Introduction to Programming Languages

Programming in C, C++, Scheme, Prolog, C#, and SOA

Fifth Edition

Yinong Chen

Arizona State University

Kendall Hunt
publishing company

Kendall Hunt
publishing company

www.kendallhunt.com
Send all inquiries to:
4050 West mark Drive
Dubuque, IA 52004-1840

ISBN 978-1-5249-1699-2

Published in the United States of America

Table of Contents

Preface

We all have witnessed the rapid development of computer science and its applications in many domains, particularly in web-based computing (Web 2.0), cloud computing, big data processing, and mobile computing. As a science discipline, the fundamentals of computer science, including programming language principles and the classic programming languages, are stable and do not change with the technological innovations. C, C++, Scheme/LISP, and Prolog belong to the classic programming languages that were developed several decades ago and are still the most significant programming languages today, both in theory and in practice. However, the technologies that surround these languages have been changed and improved, which give these languages new perspectives and new applications. For example, C++ is extended to generic classes and writing multithread programs. Functional programming principles are widely used in database query languages and the new object- and service-oriented programming languages such as C#.

This text is intended for computer science and computer engineering students in their sophomore year of study. It is assumed that students have completed a basic computer science course and have learned a high-level programming language like C, C++, or Java.

Most of the content presented in the text has been used in the "Introduction to Programming Languages" course taught by the author in the School of Computer Science at the University of the Witwatersrand at Johannesburg, and in the Computer Science and Engineering programs at Arizona State University. This text is different from the existing texts on programming languages that either focus on teaching programming paradigms, principles, and the language mechanisms, or focus on language constructs and programming skills. This text takes a balanced approach on both sides. It teaches four programming languages representing four major programming paradigms. Programming paradigms, principles, and language mechanisms are used as the vehicle to facilitate learning of the four programming languages in a coherent way. The goals of such a programming course are to make sure that computer science students are exposed to different programming paradigms and language mechanisms, and obtain sufficient programming skills and experience in different programming languages, so that they can quickly use these or similar languages in other courses.

Although there are many different programming paradigms, imperative, object-oriented, functional, and logic programming paradigms are the four major paradigms widely taught in computer science and computer engineering departments around the world. The four languages we will study in the text are the imperative C, object-oriented C++, functional Scheme, and logic Prolog. At the end of the course, students should understand

- the language structures at different layers (lexical, syntactic, contextual, and semantic), the control structures and the execution models of imperative, object-oriented, functional, and logic programming languages;
- program processing (compilation versus interpretation) and preprocessing (macros and inlining);
- different aspects of a variable, including its type, scope, name, address, memory location, and value.

More specific features of programming languages and programming issues are explored in cooperation with the four selected languages. Students are expected to have understood and be able to

- write C programs with complex data types, including pointers, arrays, and generic structures, as well as programs with static and dynamic memory allocation;

- apply the object-oriented features such as inheritance and class hierarchy, polymorphism and typing, overloading, early versus late binding, as well as the constructors, the destructor and the management of static memory, stack, and heap in C++;
- apply the functional programming style of parameter passing and write Scheme programs requiring multiple functions;
- apply the logic programming style of parameter passing, write Prolog facts, rules, and goals, and use multiple rules to solve problems;
- be able to write programs requiring multiple subprograms/procedures to solve large programming problems;
- be able to write recursive subprograms/procedures in imperative, object-oriented, functional, and logic programming languages.

The text has been revised and improved throughout in each of the new editions. In the second edition, the major change made was the inclusion of programming in Service-Oriented Architecture (SOA). Since the publication of the first edition in 2003, SOA programming has emerged as a new programming paradigm, which has demonstrated its strength to become a dominating programming paradigm. All major computing companies, including HP, IBM, Intel, Microsoft, Oracle, SAP, and Sun Microsystems, have moved into this new paradigm and are using the new paradigm to produce software and even hardware systems. The need for skill in SOA programming increases as the deployment of SOA applications increases. This new SOA paradigm is not only important in the practice of programming, but it also contributes to the concepts and principles of programming theory. In fact, SOA programming applies a higher level of abstraction, which requires fewer technical details for building large software applications. We, the authors of this book, are leading researchers and educators in SOA programming. The inclusion of the new chapter on C# and SOA programming makes the text unique, which allows teaching of the most contemporary programming concepts and practice. The new chapter also gives professors a new component to choose from, which adds flexibility for them to select different contents for different courses. As SOA has developed significantly in the past 10 years, this chapter is also updated in the fourth edition to include an introduction to Silverlight animation, phone application development, cloud computing, and big data processing.

In the third edition, we completely rewrite Chapter 5. We also discuss the modern applications of Prolog in the semantic web (Web 3.0) and its relationship with the currently used web semantic languages RDF (Resource Description Framework) and OWL (Web Ontology Language). Semantic web is considered the next generation of web that allows the web to be browsed and explored by both humans and computer programs. In the third edition revised print, this chapter is further expanded with the latest computing technologies in cloud computing and big data processing.

Since the publication of the second edition in 2006, the programming environment for Chapter 4, on Scheme, has been changed from DrScheme to DrRacket. The change does not affect the examples in the text. They all work in the new DrRacket environment, except certain notational issues. We have updated Chapter 4 to match the changes made in DrRacket.

As parallel computing and multithreading are becoming more and more important in software development, the third edition adds a new section on parallel computing and multithreading in C++, in Chapter 3. A MapReduce example is used as a case study for explaining multithreading in C++. The parallel computing concept is also emphasized in Chapter 4, where the eager evaluation and higher functions Map and Reduce are linked to parallel computing.

In the fourth edition, we added a number of new sections and many examples throughout Chapters 1, 2, 3, 4, 5, and 6 to explain the more difficulty concepts. In Chapter 1, macro expansion and execution are explained in more detail and the order of executions are discussed and showed on different programming environments. More complex examples of syntax graphs are given in Section 1.2. In Chapter 2, structure padding is discussed in Section 2.5. The file operations in Section 2.6 are extended. More recursive

examples are given in Section 2.7. A case study that puts all together is given in a new section at the end of the chapter. In Chapter 3, a new subsection on memory leak detection is added in Section 3.3 on memory management. Section 3.4 on inheritance is extended with a new subsection on type casting. A new section on C++ file operations is added as Section 3.5. In Chapter 4, the application of filtering in query languages is added in Section 4.9. In the new editions, Chapter 5 is further extended to include web application and phone application development in C#. It also extends the discussions to cloud computing and big data processing.

In the fifth edition, changes and revisions are made throughout the book. In Chapter 2, more data structures are discussed, including doubly linked list, graphs, and trees. Chapter 3, Object-Oriented Programming Language C++, is significantly extended to include inheritance, type casting, function overloading and operator overloading, and C++ file operations. A new section 3.8 Case Study is included to put together all the concepts and programming mechanisms learned in this chapter. In Appendix B, tutorials on using GNU GCC environment and Visual Studio environment to edit and debug programs are added.

This edition of the text is organized into six chapters and three appendices. Chapter 1 discusses the programming paradigms, principles, and common aspects of programming languages. Chapter 2 uses C as the example of the imperative programming paradigm to teach how to write imperative programs. It is assumed that students have a basic understanding of a high-level programming language such as C, C++, or Java. Chapter 3 extends the discussion and programming from C to C++. The chapter focuses on the main features of object-oriented programming paradigms, including class composition, dynamic memory management and garbage collection, inheritance, dynamic memory allocation and deallocation, late binding, polymorphism, and class hierarchy. Chapters 4 and 5 take a major paradigm shift to teach functional and logic programming, respectively. Students are not required to have any knowledge of functional and logic programming to learn from these two chapters. Chapter 6 gives a brief introduction to C# and service-oriented computing paradigm. A full discussion of the service-oriented computing paradigm is given in another book by the authors: Service-Oriented Computing and Web Software Integration. Assignments, programming exercises, and projects are given at the end of each chapter. The sections with an asterisk (*) are optional and can be skipped if time does not permit covering all the material. Chapters 4 and 5 are self-contained and can be taught independently, or in a different order.

The three appendices provide supplementary materials to the main text. In Appendix A, the basic computer organization and assembly language programming are introduced. If students do not have a basic computer science background, the material should be covered in the class. Appendix B starts with basic Unix commands. If the class uses a Unix-based laboratory environment, students can read the appendix to get started with Unix. Then the appendix introduces the major programming language environments that can be used to support teaching the four programming languages, including GNU GCC/G++ for C and C++, Visual Studio for C and C++, DrRacket for Scheme, and GNU Prolog. Appendix C gives the ASCII code table that is referred to in various parts of the text.

The text consists of six chapters, which can be considered reconfigurable components of a course. A half-semester course (25–30 lecture hours) can teach from two to three chapters, and a full semester course can teach four to five chapters of choice from the text. Chapter 3 (C++) is closely related to Chapter 2. If Chapter 3 is selected as a component of a course, Chapter 2, or a part of Chapter 2, should be selected as well. Other chapters are relatively independent of each other and can be selected freely to compose a course.

For a half-semester course, sensible course compositions could include: (Chapters 1, 2, 3); (Chapters 2, 3); (Chapters 1, 2, 6); (Chapters 2, 3, 6); (Chapters 1, 4, 5); and (Chapters 4, 5). For a full semester course, sensible course compositions could include: (chapters 1, 2, 3, 4, 5); (chapters 1, 2, 3, 4, 6); (Chapters 1, 2, 3, 5, 6); and (Chapters 2, 3, 4, 5, 6).

I wish to thank all those who have contributed to the materials and to the formation of this text. Particularly, I would like to thank my colleagues Scott Hazelhurst and Conrad Mueller of the University of the

Witwatersrand, and Richard Whitehouse of Arizona State University who shared their course materials with me. Parts of these materials were used in teaching the programming languages course and in preparing this text. Thomas Boyd, Joe DeLibero, Renee Turban, and Wei-Tek Tsai of Arizona State University reviewed the drafts of the text and made constructive suggestions and useful comments. Other instructors using this text have given me invaluable feedback and improvement suggestions, including Janaka Balasooriya, Calvin Cheng, Mutsumi Nakamura, and Yalin Wang. My teaching assistants helped me in the past few years to prepare the assignments and programming exercises; particularly, I would like to thank Ruben Acuna, Raynette Brodie, Justin Convery, Gennaro De Luca, Garrett Gutierrez, and Kevin Liao.

The text was written and revised while I was carrying out a full university workload. I am thankful to my family. I could not imagine that I would be able to complete the task without their generous support by allowing me to use most of the weekends in the past year to write the text.

Although I have used the materials in teaching the programming languages courses at the University of the Witwatersrand, Johannesburg and at Arizona State University for several years, the text was put together in a short period of time. There are certainly many errors of different kinds. I would appreciate it if you could send me any corrections, comments, or suggestions for improving the text. My email address dedicated to dealing with the responses to the text is <yinong.chen@asu.edu>. Instructors who use the text can contact the author for instructional support, including lecture slides in PowerPoint format and the solutions to the assignments.

Yinong Chen

December 2016

Chapter 1

Basic Principles of Programming Languages

Although there exist many programming languages, the differences among them are insignificant compared to the differences among natural languages. In this chapter, we discuss the common aspects shared among different programming languages. These aspects include:

- programming paradigms that define how computation is expressed;
- the main features of programming languages and their impact on the performance of programs written in the languages;
- a brief review of the history and development of programming languages;
- the lexical, syntactic, and semantic structures of programming languages, data and data types, program processing and preprocessing, and the life cycles of program development.

At the end of the chapter, you should have learned:

- what programming paradigms are;
- an overview of different programming languages and the background knowledge of these languages;
- the structures of programming languages and how programming languages are defined at the syntactic level;
- data types, strong versus weak checking;
- the relationship between language features and their performances;
- the processing and preprocessing of programming languages, compilation versus interpretation, and different execution models of macros, procedures, and inline procedures;
- the steps used for program development: requirement, specification, design, implementation, testing, and the correctness proof of programs.

The chapter is organized as follows. Section 1.1 introduces the programming paradigms, performance, features, and the development of programming languages. Section 1.2 outlines the structures and design issues of programming languages. Section 1.3 discusses the typing systems, including types of variables, type equivalence, type conversion, and type checking during the compilation. Section 1.4 presents the preprocessing and processing of programming languages, including macro processing, interpretation, and compilation. Finally, Section 1.5 discusses the program development steps, including specification, testing, and correctness proof.

1.1 Introduction

1.1.1 Programming concepts and paradigms

Millions of programming languages have been invented, and several thousands of them are actually in use. Compared to natural languages that developed and evolved independently, programming languages are far more similar to each other. This is because

- different programming languages share the same mathematical foundation (e.g., Boolean algebra, logic);
- they provide similar functionality (e.g., arithmetic, logic operations, and text processing);
- they are based on the same kind of hardware and instruction sets;
- they have common design goals: find languages that make it simple for humans to use and efficient for hardware to execute;
- designers of programming languages share their design experiences.

Some programming languages, however, are more similar to each other, while other programming languages are more different from each other. Based on their similarities or the paradigms, programming languages can be divided into different classes. In programming language's definition, **paradigm** is a set of basic principles, concepts, and methods for how a computation or algorithm is expressed. The major paradigms we will study in this text are imperative, object-oriented, functional, and logic paradigms.

The **imperative**, also called the **procedural**, programming paradigm expresses computation by fully specified and fully controlled manipulation of named data in a stepwise fashion. In other words, data or values are initially stored in variables (memory locations), taken out of (read from) memory, manipulated in ALU (arithmetic logic unit), and then stored back in the same or different variables (memory locations). Finally, the values of variables are sent to the I/O devices as output. The foundation of imperative languages is the **stored program concept**-based computer hardware organization and architecture (von Neumann machine). The stored program concept will be further explained in the next chapter. Typical imperative programming languages include all assembly languages and earlier high-level languages like Fortran, Algol, Ada, Pascal, and C.

The **object-oriented** programming paradigm is basically the same as the imperative paradigm, except that related variables and operations on variables are organized into classes of **objects**. The access privileges of variables and methods (operations) in objects can be defined to reduce (simplify) the interaction among objects. Objects are considered the main building blocks of programs, which support language features like inheritance, class hierarchy, and polymorphism. Typical object-oriented programming languages include Smalltalk, C++, Java, and C#.

The **functional**, also called the **applicative**, programming paradigm expresses computation in terms of mathematical functions. Since we express computation in mathematical functions in many of the mathematics courses, functional programming is supposed to be easy to understand and simple to use. However, since functional programming is very different from imperative or object-oriented programming, and most programmers first get used to writing programs in imperative or object-oriented paradigms, it becomes difficult to switch to functional programming. The main difference is that there is no concept of memory locations in functional programming languages. Each function will take a number of values as input (parameters) and produce a single return value (output of the function). The return value cannot be stored for later use. It has to be used either as the final output or immediately as the parameter value of another function. Functional programming is about defining functions and organizing the return values of one or more functions as the parameters of another function. Functional programming languages are mainly

based on the **lambda calculus** that will be discussed in Chapter 4. Typical functional programming languages include ML, SML, and Lisp/Scheme.

The **logic**, also called the **declarative**, programming paradigm expresses computation in terms of logic predicates. A logic program is a set of facts, rules, and questions. The execution process of a logic program is to compare a question to each fact and rule in the given fact and rulebase. If the question finds a match, we receive a *yes* answer to the question. Otherwise, we receive a *no* answer to the question. Logic programming is about finding facts, defining rules based on the facts, and writing questions to express the problems we wish to solve. Prolog is the only significant logic programming language.

It is worthwhile to note that many languages belong to multiple paradigms. For example, we can say that C++ is an object-oriented programming language. However, C++ includes almost every feature of C and thus is an imperative programming language too. We can use C++ to write C programs. Java is more object-oriented, but still includes many imperative features. For example, Java's primitive type variables do not obtain memory from the language heap like other objects. Lisp contains many nonfunctional features. Scheme can be considered a subset of Lisp with fewer nonfunctional features. Prolog's arithmetic operations are based on the imperative paradigm.

Nonetheless, we will focus on the paradigm-related features of the languages when we study the sample languages in the next four chapters. We will study the imperative features of C in Chapter 2, the object-oriented features of C++ in Chapter 3, and the functional features of Scheme and logic features of Prolog in Chapters 4 and 5, respectively.

1.1.2 Program performance and features of programming languages

A programming language's features include orthogonality or simplicity, available control structures, data types and data structures, syntax design, support for abstraction, expressiveness, type equivalence, and strong versus weak type checking, exception handling, and restricted aliasing. These features will be further explained in the rest of the book. The performance of a program, including reliability, readability, writability, reusability, and efficiency, is largely determined by the way the programmer writes the algorithm and selects the data structures, as well as other implementation details. However, the features of the programming language are vital in supporting and enforcing programmers in using proper language mechanisms in implementing the algorithms and data structures. Table 1.1 shows the influence of a language's features on the performance of a program written in that language.

Performance / Language features	Efficiency	Readability/ Reusability	Writability	Reliability
Simplicity/Orthogonality	✓	✓	✓	✓
Control structures	✓	✓	✓	✓
Typing and data structures	✓	✓	✓	✓
Syntax design		✓	✓	✓
Support for abstraction		✓	✓	✓
Expressiveness			✓	✓
Strong checking				✓
Restricted aliasing				✓
Exception handling				✓

Table 1.1. Impact of language features on the performance of the programs.

The table indicates that simplicity, control structures, data types, and data structures have significant impact on all aspects of performance. Syntax design and the support for abstraction are important for readability, reusability, writability, and reliability. However, they do not have a significant impact on the efficiency of the program. Expressiveness supports writability, but it may have a negative impact on the reliability of the program. Strong type checking and restricted aliasing reduce the expressiveness of writing programs, but are generally considered to produce more reliable programs. Exception handling prevents the program from crashing due to unexpected circumstances and semantic errors in the program. All language features will be discussed in this book.

1.1.3 Development of programming languages

The development of programming languages has been influenced by the development of hardware, the development of compiler technology, and the user's need for writing high-performance programs in terms of reliability, readability, writability, reusability, and efficiency. The hardware and compiler limitations have forced early programming languages to be close to the machine language. Machine languages are the native languages of computers and the first generation of programming languages used by humans to communicate with the computer.

Machine languages consist of instructions of pure binary numbers that are difficult for humans to remember. The next step in programming language development is the use of mnemonics that allows certain symbols to be used to represent frequently used bit patterns. The machine language with sophisticated use of mnemonics is called **assembly language**. An assembly language normally allows simple variables, branch to a label address, different addressing modes, and macros that represent a number of instructions. An **assembler** is used to translate an assembly language program into the machine language program. The typical work that an assembler does is to translate mnemonic symbols into corresponding binary numbers, substitute register numbers or memory locations for the variables, and calculate the destination address of branch instructions according to the position of the labels in the program.

This text will focus on introducing high-level programming languages in imperative, object-oriented, functional, and logic paradigms.

The first high-level programming language can be traced to Konrad Zuse's Plankalkül programming system in Germany in 1946. Zuse developed his Z-machines Z1, Z2, Z3, and Z4 in the late 1930s and early 1940s, and the Plankalkül system was developed on the Z4 machine at ETH (Eidgenössisch Technische Hochschule) in Zürich, with which Zuse designed a chess-playing program.

The first high-level programming language that was actually used in an electronic computing device was developed in 1949. The language was named Short Code. There was no compiler designed for the language, and programs written in the language had to be hand-compiled into the machine code.

The invention of the compiler was credited to **Grace Hopper**, who designed the first widely known compiler, called **A0**, in 1951.

The first primitive compiler, called Autocoder, was written by Alick E. Glennie in 1952. It translated Autocode programs in symbolic statements into machine language for the Manchester Mark I computer. Autocode could handle single letter identifiers and simple formulas.

The first widely used language, Fortran (FORmula TRANslating), was developed by the team headed by John Backus at IBM between 1954 and 1957. Backus was also the system co-designer of the IBM 704 that ran the first Fortran compiler. Backus was later involved in the development of the language Algol and the Backus-Naur Form (BNF). BNF was a formal notation used to define the syntax of programming languages. Fortran II came in 1958. Fortran III came at the end of 1958, but it was never released to the public. Further versions of Fortran include ASA Fortran 66 (Fortran IV) in 1966, ANSI Fortran 77 (Fortran V) in 1978,

ISO Fortran 90 in 1991, and ISO Fortran 95 in 1997. Unlike assembly languages, the early versions of Fortran allowed different types of variables (real, integer, array), supported procedure call, and included simple control structures.

Programs written in programming languages before the emergence of structured programming concepts were characterized as spaghetti programming or monolithic programming. **Structured programming** is a technique for organizing programs in a hierarchy of modules. Each module had a single entry and a single exit point. Control was passed downward through the structure without unconditional branches (e.g., *goto* statements) to higher levels of the structure. Only three types of control structures were used: sequential, conditional branch, and iteration.

Based on the experience of Fortran I, Algol 58 was announced in 1958. Two years later, Algol 60, the first block-structured language, was introduced. The language was revised in 1963 and 1968. Edsger Dijkstra is credited with the design of the first Algol 60 compiler. He is famous as the leader in introducing structured programming and in abolishing the *goto* statement from programming.

Rooted in Algol, **Pascal** was developed by Niklaus Wirth between 1968 and 1970. He further developed Modula as the successor of Pascal in 1977, then Modula-2 in 1980, and Oberon in 1988. Oberon language had Pascal-like syntax, but it was strongly typed. It also offered type extension (inheritance) that supported object-oriented programming. In Oberon-2, type-bound procedures (like *methods* in object-oriented programming languages) were introduced.

The **C** programming language was invented and first implemented by Dennis Ritchie at DEC between 1969 and 1973, as a system implementation language for the nascent Unix operating system. It soon became one of the dominant languages at the time and even today. The predecessors of C were the typeless language BCPL (Basic Combined Programming Language) by Martin Richards in 1967 and then the B written by Ken Thompson in 1969. C had a weak type checking structure to allow a higher level of programming flexibility.

Object-oriented (OO) programming concepts were first introduced and implemented in the **Simula** language, which was designed by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center (NCC) between 1962 and 1967. The original version, Simula I, was designed as a language for discrete event simulation. However, its revised version, Simula 67, was a full-scale general-purpose programming language. Although Simula never became widely used, the language was highly influential on the modern programming paradigms. It introduced important object-oriented concepts like classes and objects, inheritance, and late binding.

One of the object-oriented successors of Simula was **Smalltalk**, designed at Xerox PARC, led by Alan Kay. The versions developed included Smalltalk-72, Smalltalk-74, Smalltalk-76, and Smalltalk-80. Smalltalk also inherited functional programming features from Lisp.

Based on Simula 67 and C, a language called "**C with classes**" was developed by Bjarne Stroustrup in 1980 at Bell Labs, and then revised and renamed as **C++** in 1983. C++ was considered a better C (e.g., with strong type checking), plus it supported data abstraction and object-oriented programming inherited from Simula 67.

Java was written by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems. It was called Oak at first and then renamed Java when it was publicly announced in 1995. The predecessors of Java were C++ and Smalltalk. Java removed most non-object-oriented features of C++ and was a simpler and better object-oriented programming language. Its two-level program processing concept (i.e., compilation into an intermediate bytecode and then interpretation of the bytecode using a small virtual machine) made it the dominant language for programming Internet applications. Java was still

not a pure object-oriented programming language. Its primitive types, integer, floating-point number, Boolean, etc., were not classes, and their memory allocations were from the language stack rather than from the language heap.

Microsoft's **C#** language was first announced in June 2000. The language was derived from C++ and Java. It was implemented as a full object-oriented language without "primitive" types. C# also emphasizes component-oriented programming, which is a refined version of object-oriented programming. The idea is to be able to assemble software systems from prefabricated components.

Functional programming languages are relatively independent of the development process of imperative and object-oriented programming languages. The first and most important functional programming language, **Lisp**, short for LISt Processing, was developed by John McCarthy at MIT. Lisp was first released in 1958. Then Lisp 1 appeared in 1959, Lisp 1.5 in 1962, and Lisp 2 in 1966. Lisp was developed specifically for artificial intelligence applications and was based on the lambda calculus. It inherited its algebraic syntax from Fortran and its symbol manipulation from the Information Processing Language, or IPL. Several dialects of Lisp were designed later, for example, Scheme, InterLisp, FranzLisp, MacLisp, and ZetaLisp.

As a Lisp dialect, **Scheme** was first developed by G. L. Steele and G. J. Sussman in 1975 at MIT. Several important improvements were made in its later versions, including better scope rule, procedures (functions) as the first-class objects, removal of loops, and sole reliance on recursive procedure calls to express loops. Scheme was standardized by the IEEE in 1989.

Efforts began on developing a common dialect of Lisp, referred to as Common Lisp, in 1981. Common Lisp was intended to be compatible with all existing versions of Lisp dialects and to create a huge commercial product. However, the attempt to merge Scheme into Lisp failed, and Scheme remains an independent Lisp dialect today. Common Lisp was standardized by the IEEE in 1992.

Other than Lisp, John Backus's **FP** language also belongs to the first functional programming languages. FP was not based on the lambda calculus, but based on a few rules for combining function forms. Backus felt that lambda calculus's expressiveness on computable functions was much broader than necessary. A simplified rule set could do a better job.

At the same time that FP was developed in the United States, **ML** (Meta Language) appeared in the United Kingdom. Like Lisp, ML was based on lambda calculus. However, Lisp was not typed (no variable needs to be declared), while ML was strongly typed, although users did not have to declare variables that could be inferentially determined by the compiler.

Miranda is a pure functional programming language developed by David Turner at the University of Kent in 1985–1986. Miranda achieves referential transparency (side effect-free) by forbidding modification to global variables. It combines the main features of **SASL** (St. Andrews Static Language) and **KRC** (Kent Recursive Calculator) with strong typing similar to that of ML. SASL and KRC are two earlier functional programming languages designed by Turner at the University of St Andrews in 1976 and at the University of Kent in 1981, respectively.

There are many logic-based programming languages in existence. For example, **ALF** (Algebraic Logic Functional language) is an integrated functional and logic language based on Horn clauses for logic programming, and functions and equations for functional programming. Gödel is a strongly typed logic programming language. The type system is based on a many-sorted logic with parametric polymorphism. **RELFUN** extends Horn logic by using higher-order syntax, first-class finite domains, and expressions of nondeterministic, nonground functions, explicitly distinguished from structures.

The most significant member in the family of logic programming languages is the **Horn logic**-based **Prolog**. Prolog was invented by Alain Colmerauer and Philippe Roussel at the University of Aix-Marseille in 1971. The first version was implemented in 1972 using Algol. Prolog was designed originally for natural-language processing, but it has become one of the most widely used languages for artificial intelligence. Many implementations appeared after the original work. Early implementations included **C-Prolog**, **ESLPDPRO**, **Frolic**, LM-Prolog, Open Prolog, SB-Prolog, and UPMAIL Tricia Prolog. Today, the common Prologs in use are AMZI Prolog, **GNU Prolog**, LPA Prolog, **Quintus Prolog**, SICSTUS Prolog, SNI Prolog, and **SWI-Prolog**.

Distributed computing involves computation executed on more than one logical or physical processor or computer. These units cooperate and communicate with each other to complete an integral application. The computation units can be functions (methods) in the component, components, or application programs. The main issues to be addressed in the distributed computing paradigms are concurrency, concurrent computing, resource sharing, synchronization, messaging, and communication among distributed units. Different levels of distribution lead to different variations. **Multithreading** is a common distributed computing technique that allows different functions in the same software to be executed concurrently. If the distributed units are at the object level, this is **distributed OO computing**. Some well-known distributed OO computing frameworks are CORBA (Common Object Request Broker Architecture) developed by OMG (Object Management Group) and Distributed Component Object Model (DCOM) developed Microsoft.

Service-oriented computing (SOC) is another distributed computing paradigm. SOC differs from distributed OO computing in several ways:

- SOC emphasizes distributed services (with possibly service data) rather than distributed objects;
- SOC explicitly separates development duties and software into service provision, service brokerage, and application building through service consumption;
- SOC supports reusable services in (public or private) repositories for matching, discovery and (remote or local) access;
- In SOC, services communicate through open standards and protocols that are platform independent and vendor independent.

It is worthwhile noting that many languages belong to multiple computing paradigms; for example, C++ is an OO programming language. However, C++ also includes almost every feature of C. Thus, C++ is also an imperative programming language, and we can use C++ to write C programs.

Java is more an OO language, that is, the design of the language puts more emphasis on the object orientation. However, it still includes many imperative features; for example, Java's primitive type variables use value semantics and do not obtain memory from the language heap.

Service-oriented computing is based on the object-oriented computing. The main programming languages in service-oriented computing, including Java and C#, can be used for object-oriented software development.

The latest development in programming languages is the **visual/graphic programming**. MIT App Inventor (http://appinventor.mit.edu/) uses drag-and-drop style puzzles to construct phone applications in Android platform. Carnegie Mellon's Alice (http://www.alice.org/) is a 3D game and movie development environment on desktop computers. It uses a drop-down list for users to select the available functions in a stepwise manner. App Inventor and Alice allow novice programmers to develop complex applications using visual composition at the workflow level. Intel's IoT Service Orchestration Layer is a workflow language that allows quick development of IoT (Internet of Things) applications on Intel's IoT platforms, such as Edison and Galileo (http://01org.github.io/intel-iot-services-orchestration-layer/).

Microsoft Robotics Developer Studio (MRDS) and Visual Programming Language (VPL) are specifically developed for robotics applications (https://en.wikipedia.org/wiki/Microsoft_Robotics_Developer_Studio). They are milestones in software engineering, robotics, and computer science education from many aspects. MRDS VPL is service-oriented; it is visual and workflow-based; it is event-driven; it supports parallel computing; and it is a great educational tool that is simple to learn and yet powerful and expressive. Unfortunately, Microsoft stopped its development and support to MRDS VPL in 2014, which lead to many schools' courses, including ASU FSE100 course, using VPL without further support.

To keep ASU course running and also help the other schools, Dr. Yinong Chen, Gennaro De Luca, and the development team at IoT and Robotics Education Laboratory at ASU took the challenge and the responsibility to develop a new visual programming environment at Arizona State University in 2015: **ASU VIPLE**, standing for Visual IoT/Robotics Programming Language Environment. It is designed to support as many features and functionalities that MRDS VPL supports as possible, in order to better serve the MRDS VPL community in education and research. To serve this purpose, VIPLE also keeps similar user interface, so that the MRDS VPL development community can use VIPLE with little learning curve. VIPLE does not replace MRDS VPL. Instead, it extends MRDS VPL in its capacity in multiple aspects. It can connect to different physical robots, including LEGO EV3 and any robots based on the open architecture processors. ASU VIPLE software and documents are free and can be downloaded at: http://neptune.fulton.ad.asu.edu/WSRepository/VIPLE/

1.2 Structures of programming languages

This section studies the structures of programming languages in terms of four structural layers: lexical, syntactic, contextual, and semantic.

1.2.1 Lexical structure

Lexical structure defines the vocabulary of a language. Lexical units are considered the building blocks of programming languages. The lexical structures of all programming languages are similar and normally include the following kinds of units:

- **Identifiers**: Names that can be chosen by programmers to represent objects like variables, labels, procedures, and functions. Most programming languages require that an identifier start with an alphabetical letter and can be optionally followed by letters, digits, and some special characters.
- **Keywords**: Names reserved by the language designer and used to form the syntactic structure of the language.
- **Operators**: Symbols used to represent the operations. All general-purpose programming languages should provide certain minimum operators such as mathematical operators like +, −, *, /, relational operators like <, ≤, ==, >, ≥, and logic operators like AND, OR, NOT, etc.
- **Separators**: Symbols used to separate lexical or syntactic units of the language. Space, comma, colon, semicolon, and parentheses are used as separators.
- **Literals**: Values that can be assigned to variables of different types. For example, integer-type literals are integer numbers, character-type literals are any character from the character set of the language, and string-type literals are any string of characters.
- **Comments**: Any explanatory text embedded in the program. Comments start with a specific keyword or separator. When the compiler translates a program into machine code, all comments will be ignored.

1.2.2 Syntactic structure

Syntactic structure defines the grammar of forming sentences or statements using the lexical units. An imperative programming language normally offers the following kinds of statements:

- **Assignments**: An assignment statement assigns a literal value or an expression to a variable.
- **Conditional statements**: A conditional statement tests a condition and branches to a certain statement based on the test result (*true* or *false*). Typical conditional statements are *if-then*, *if-then-else*, and *switch* (*case*).
- **Loop statements**: A loop statement tests a condition and enters the body of the loop or exits the loop based on the test result (*true* or *false*). Typical loop statements are *for-loop* and *while-loop*.

The formal definition of lexical and syntactic structures will be discussed in Section 1.2.5.

1.2.3 Contextual structure

Contextual structure (also called **static semantics**) defines the program semantics before dynamic execution. It includes variable declaration, initialization, and type checking.

Some imperative languages require that all variables be initialized when they are declared at the contextual layer, while other languages do not require variables to be initialized when they are declared, as long as the variables are initialized before their values are used. This means that initialization can be done either at the contextual layer or at the semantic layer.

Contextual structure starts to deal with the meaning of the program. A statement that is lexically correct may not be contextually correct. For example:

```
String str = "hello";
int i = 0;
int j = i + str;
```

All declaration statements are lexically correct, but the last statement is contextually incorrect because it does not make sense to add an integer variable to a string variable.

More about data type, type checking, and type equivalence will be discussed in Section 1.3.

1.2.4 Semantic structure

Semantic structure describes the meaning of a program, or what the program does during the execution. The semantics of a language are often very complex. In most imperative languages, there is no formal definition of semantic structure. Informal descriptions are normally used to explain what each statement does. The semantic structures of functional and logic programming languages are normally defined based on the mathematical and logical foundation on which the languages are based. For example, the meanings of Scheme procedures are the same as the meanings of the lambda expressions in lambda calculus on which Scheme is based, and the meanings of Prolog clauses are the same as the meanings of the clauses in Horn logic on which Prolog is based.

1.2.5 BNF notation

BNF (Backus-Naur Form) is a meta language that can be used to define the lexical and syntactic structures of another language. Instead of learning BNF language first and then using BNF to define a new language, we will first use BNF to define a simplified English language that we are familiar with, and then we will learn BNF from the definition itself.

A simple English sentence consists of a subject, a verb, and an object. The subject, in turn, consists of possibly one or more adjectives followed by a noun. The object has the same grammatical structure. The verbs and adjectives must come from the vocabulary. Formally, we can define a simple English sentence as follows:

```
<sentence>  ::= <subject><verb><object>
<subject>   ::= <noun> | <article><noun> | <adjective><noun> |
                <article><adjective><noun>
<adjective> ::= <adjective> | <adjective><adjective>
<object>    ::= <subject>
<noun> ::= table | horse | computer
<article>   ::= the | a
<adjective> ::= big | fast | good | high
<verb> ::= is | makes
```

In the definitions, the symbol "::=" means that the name on the left-hand side is defined by the expression on the right-hand side. The name in a pair of angle brackets "<>" is **nonterminal**, which means that the name needs to be further defined. The vertical bar "|" represents an "or" relation. The boldfaced names are **terminal**, which means that the names need not be further defined. They form the vocabulary of the language.

We can use the sentence definition to check whether the following sentences are syntactically correct.

```
fast high big computer is good table                    1
the high table is a good table                          2
a fast table makes the high horse                       3
the fast big high computer is good                      4
good table is high                                      5
a table is not a horse                                  6
is fast computer good                                   7
```

The first sentence is syntactically correct, although it does not make much sense. Three adjectives in the sentence are correct because the definition of an adjective recursively allows any number of adjectives to be used in the subject and the object of a sentence. The second and third sentences are also syntactically correct according to the definition.

The fourth and fifth sentences are syntactically incorrect because a noun is missing in the object of the sentences. The sixth sentence is incorrect because "not" is not a terminal. The last sentence is incorrect because the definition does not allow a sentence to start with a verb.

After we have a basic understanding of BNF, we can use it to define a small programming language. The first five lines define the lexical structure, and the rest defines the syntactic structure of the language.

```
<letter>     ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit>      ::= 0|1|2|3|4|5|6|7|8|9
<symbol>     ::= _|@|.|~|?|#|$
<char>       ::= <letter>|<digit>|<symbol>
<operator>   ::= +|-|*|/|%|<|>|==|<=|>=|and|or|not
<identifier>     ::= <letter>|<identifier><char>
<number>     ::= <digit>|<number><digit>
<item>       ::= <identifier>|<number>
```

```
<expression> ::= <item>|(<expression>)|
                                <expression><operator><expression>
<branch>     ::= if <expr>then {<block>} |
                 if <expr>then {<block>}else {<block>}
<switch>     ::= switch<expr>{<sbody>}
<sbody>      ::= <cases> | <cases>; default :<block>
<cases>      ::= case<value>:<block> | <cases> ; case<value>:<block>
<loop>       ::= while <expr>do {<block>}
<assignment> ::= <identifier>=<expression>;
<statement>  ::= <assignment>|<branch>|<loop>
<block>      ::= <statement>|<block>;<statement>
```

Now we use the definition to check which of the following statements are syntactically correct.

```
sum1 = 0;                                                    1
while sum1 <= 100 do {                                       2
sum1 = sum1 + (a1 + a2) * (3b % 4*b); }                      3
if sum1 == 120 then 2sum - sum1 else sum2 + sum1;            4
p4#rd_2 = ((1a + a2) * (b3 % b4)) / (c7 - c8);               5
_foo.bar = (a1 + a2 - b3 - b4);                              6
(a1 / a2) = (c3 - c4);                                       7
```

According to the BNF definition of the language, statements 1 and 2 are correct. Statements 3 and 4 are incorrect because 3*b* and 2*sum* are neither acceptable identifiers nor acceptable expressions. Statement 5 is incorrect. Statement 6 is incorrect because an identifier must start with a letter. Statement 7 is incorrect because the left-hand side of an assignment statement must be an identifier.

1.2.6 Syntax graph

BNF notation provides a concise way to define the lexical and syntactic structures of programming languages. However, BNF notations, especially the recursive definitions, are not always easy to understand. A graphic form, called a **syntax graph**, also known as **railroad tracks**, is often used to supplement the readability of BNF notation. For example, the identifier and the if-then-else statement corresponding to the BNF definitions can be defined using the syntax graphs in Figure 1.1. The syntax graph for the identifier requires that an identifier start with a letter, may exit with only one letter, or follow the loops to include any number of letters, digits, or symbols. In other words, to check the legitimacy of an identifier, we need to travel through the syntax graph following the arrows and see whether we can find a path that matches the given identifier. For instance, we can verify that `len_23` is a legitimate identifier as follows. We travel through the first `<letter>` once, travel through the second `<letter>` on the back track twice, travel through the `<symbol>` once, and finally travel through the `<digit>` twice, and then we exit the definition. On the other hand, if you try to verify that `23_len` is a legitimate identifier, you will not be able to find a path to travel through the syntax graph.

Using the if-then-else syntax graph in Figure 1.1, we can precisely verify whether a given statement is a legitimate if-then-else statement. The alternative route that bypasses the else branch signifies that the else branch is optional. Please note that the definition of the if-then-else statement here is not the same as the if-then-else statement in C language. The syntax graph definitions of various C statements can be found in Chapter 2.

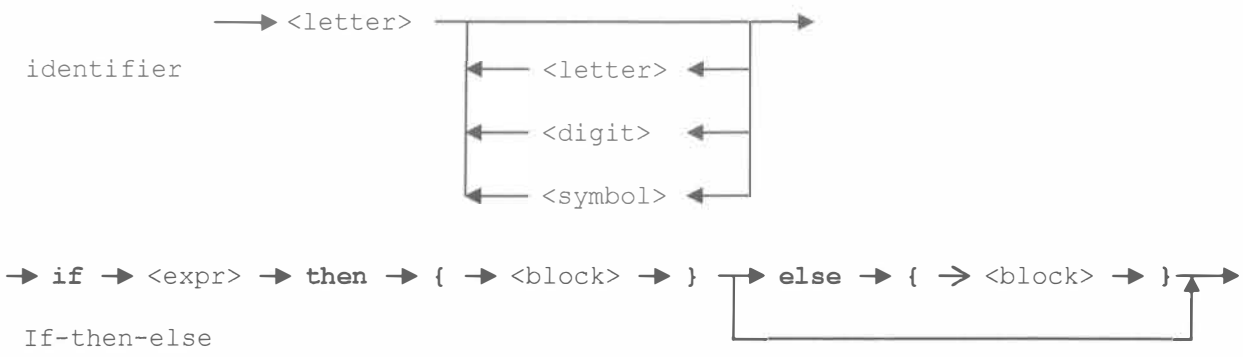


Figure 1.1. Definition of identifier and if-then-else statement.

As another example, Figure 1.2 shows the definitions of a set data structures, including the definitions of value, string, array, bool, and number.

In syntax graphs, we use the same convention that terminals are in boldfaced text and nonterminals are enclosed in a pair of angle brackets.

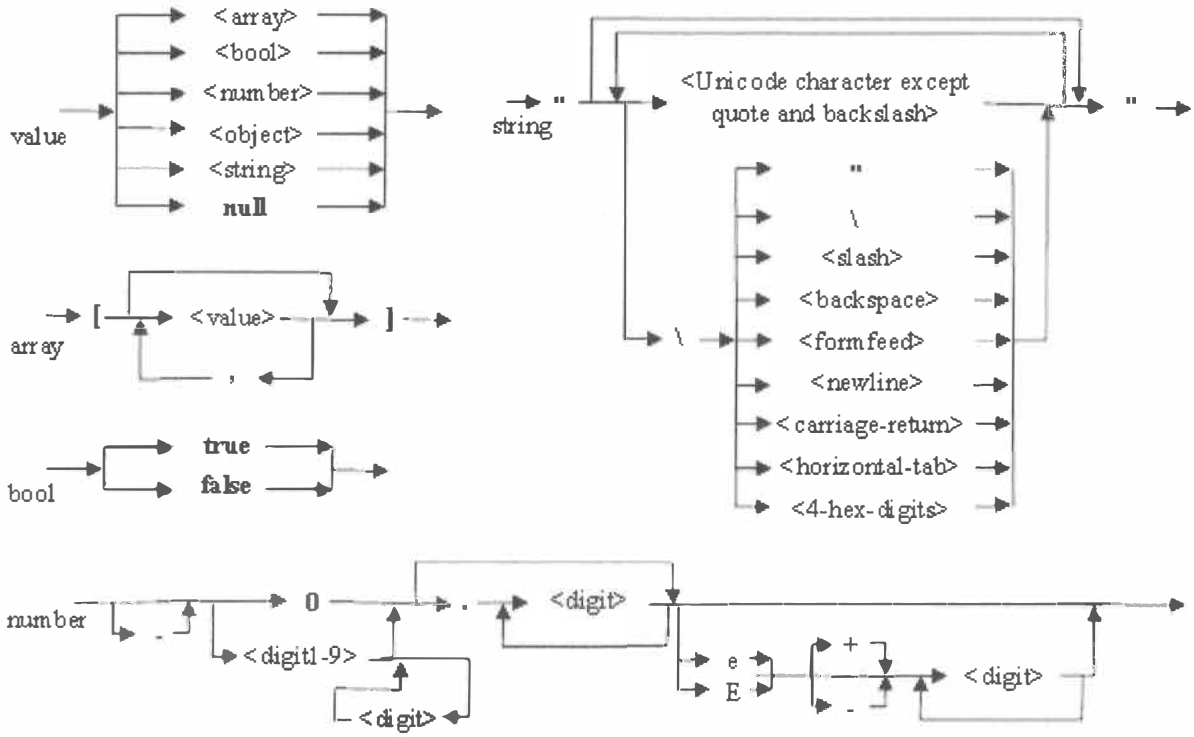


Figure 1.2. Definitions of different data structures.

1.3 Data types and type checking

In this section, we examine data types, type equivalence, and type checking that is part of the contextual structure of a programming language.

1.3.1 Data types and type equivalence

A **data type** is defined by the set of primary values allowed and the operations defined on these values. A data type is used to declare variables (e.g., integer, real, array of integer, string, etc.).

Type checking is the activity of ensuring that the data types of operands of an operator are legal or equivalent to the legal type.

Now the question is, what types are equivalent? For example, are *int* and *short* types, and *int* and *float* types in C language equivalent? Are the following operations legal in C?

```
int i = 3; short j = 5;float n, k = 3.0;
n = i + j + k;
```

The answers to these questions are related to the type equivalence policy of programming languages. There are two major type equivalence policies: structural equivalence and name equivalence. If the **structural equivalence** policy is used, the two types are equivalent if they have the same set of values (data range) and the same operations. This policy follows the stored program concept and gives programmers the maximum flexibility to manipulate data. The **stored program concept** suggests that instruction and data are stored in computer memory as binary bit patterns and it is the programmer's responsibility to interpret the meanings of the bit patterns. Algol 68, Pascal, and C are examples of languages that use structural equivalence policy. For example, in the following C code, a type *salary* and a type age are defined:

```
typedef integer salary;
typedef integer age;
int i = 0; salary s = 60000; age a = 40;
i = s + a;
```

If structural equivalence policy is used, the statement `"i = s + a;"` is perfectly legal because all three variables belong to types that are structurally equivalent. Obviously, adding an *age* value to a *salary* value and putting it into an integer type variable normally does not make sense and most probably is a programming error. Structural equivalence policy assumes programmers know what they are doing.

On the other hand, if **name equivalence** policy is used, two types are equivalent only if they have the same name. Since no programming language will normally allow two different data types to have the same name, this policy does not allow any variable of one type to be used in the place where a variable of another type is expected without explicit type conversion. If name equivalence policy is used, the statement `"i = s + a;"` is then illegal.

Ada, C++, and Java are examples where the name equivalence policy is used. Name equivalence enforces a much stronger discipline and is generally considered a good way to ensure the correctness of a program.

1.3.2 Type checking and type conversion

Besides type equivalence, type-checking policy is another important characteristic of a programming language. We can vaguely differentiate strongly typed and weakly typed languages. In a (truly) **strongly typed language:**

- every name in a program must be associated with a single type that is known at the compilation time;
- name equivalence is used; and
- every type inconsistency must be reported.

C++ and Java are strongly typed languages. Functional programming languages are normally weakly typed because mathematical functions are not typed. However, ML is strongly typed, although users did not have to declare variables that could be inferentially determined by the compiler.

On the other hand, in a **weakly typed language:**

- not every name has to be associated with a single type at the compilation time;
- structural equivalence is used; and
- a variable of a subtype is acceptable, and implicit type conversion, called **coercion**, is allowed.

Type T1 is considered the **subtype** of T2, if the set of data of T1 is a subset of data of T2 and the set of operations of T1 is a subset of operations of T2. For example, an *integer* type can be considered a subtype of *floating-point* type. C, Scheme, and Prolog are weakly typed programming languages. Typeless languages like BCPL are weakly typed.

If a data type is a subtype of another type, it does not mean that the two types have the equivalent or similar structure. For example, in a 32-bit computer, an integer number 3 is represented in a simple binary form, as shown in the upper part of Figure 1.3. On the other hand, a floating-point number is normally represented in a three-segment format (IEEE 754 Standard), with the leftmost bit representing the sign (positive or negative), the next 8 bits representing the exponent, and the remaining 23 bits representing the fraction of the number, as shown in the lower part of Figure 1.3.

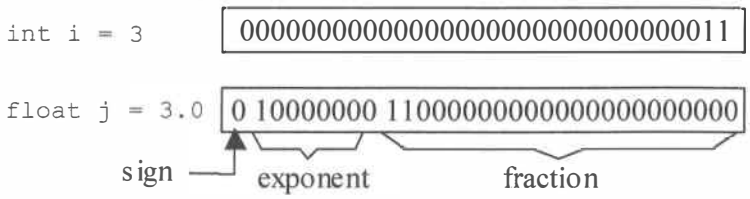


Figure 1.3. Integer and floating-point numbers have different internal structures.

In a strongly typed programming language, you can still use different types of variables in a single expression. In this case, one has to do explicit type conversion. In C/C++, explicit type conversion is called **typecasting**: the destination type in parentheses is placed before the variable that has an incompatible type. In the following statement, explicit conversion is used to convert variable `s` of *salary* type and variable `a` of *age* type into `int` type:

```
i = (int)s + (int)a;
```

Strong type checking trades flexibility for reliability. It is generally considered a good policy and is used in most recent programming languages.

1.3.3 Orthogonality

The word **orthogonality** refers to the property of straight lines meeting at right angles or independent random variables. In programming languages, orthogonality refers to the property of being able to combine various language features systematically. According to the way the features are combined, we can distinguish three kinds of orthogonality: compositional, sort, and number orthogonality.

Compositional orthogonality: If one member of the set of features S_1 can be combined with one member of the set of features S_2, then all members of S_1 can be combined with all members of S_2, as shown in Figure 1.4.

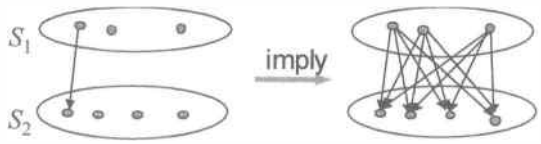


Figure 1.4. Compositional orthogonality.

For example, assume that S_1 is the set of different kinds of declarations: (1) plain, (2) initializing, and (3) constant. For example

```
(1) typex i; (2) typex i = 5; (3) const typex i = 5;
```

Set S_2 is data types: boolean, int, float, array.

If the language is compositionally orthogonal, then we can freely combine these two sets of features in all possible ways:

- Plain boolean, plain int, plain float, plain array

```
(1) boolean b; (2) int i; (3) float f; (4) array a[];
```

- Initializing boolean, initializing int, initializing float, and initializing array

```
(1) boolean b = true;
(2) int i = 5;
(3) float f = 4.5;
(4) array a[3] = {4, 6, 3};
```

- Constant boolean, constant int, constant float, constant array

```
(1) const boolean b = true;
(2) const int i = 5;
(3) const float f = 7.5;
(4) const array a[3] = {1, 2, 8};
```

Sort orthogonality: If one member of the set of features S_1 can be combined with one member of the set of features S_2, then *this* member of S_1 can be combined with all members of S_2, as shown in Figure 1.5.

For example, if we know that `int i` is allowed (the combination plain-int), then, according to sort orthogonality, plain boolean, plain int, plain float, and plain array will be allowed, that is:

```
(1) boolean b;
(2) int i;
(3) float f;
(4) array a[];
```

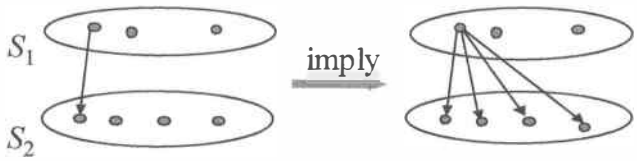


Figure 1.5. Sort orthogonality 1.

However, since sort orthogonality does not allow other members of S_1 to combine with members of S_2, we cannot conclude that initializing and constant declarations can be applied to the data types in S_2.

The sort orthogonality can be viewed from the other side. If one member of the set S_1 can be combined with one member of the set S_2, then all members in S_1 can be combined with the members of S_2, as shown in Figure 1.6.

For example, if the plain declaration can be combined with the `int` type, we conclude that the plain, initializing, and constant declarations can be applied to the `int` type.

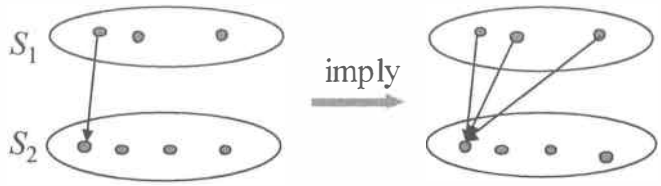


Figure 1.6. Sort orthogonality 2.

For example, if we know that `const array a` is allowed (the combination constant and array), according to the sort orthogonality, then plain array, initializing array, and constant array will be allowed:

```
(1) array a[];
(2) array a[3] = {1, 2, 8};
(3) const array a[3] = {1, 2, 8};
```

However, since the sort orthogonality does not allow other members of S_2 to combine with members of S_1, we cannot conclude that `boolean`, `int`, and `float` type can be declared in three different ways.

Number orthogonality: If one member of the set of features S is allowed, then zero or multiple features of S are allowed. For example, if you can declare a variable in a particular place (in many languages, e.g., C++ and Java, you can put declarations anywhere in the program), you should be able to put zero (no declaration) or multiple declarations in that place. In a class definition, if you can define one member, you should be allowed to define zero or multiple members.

1.4 Program processing and preprocessing

This section discusses what preparations need to be done before the computer hardware can actually execute the programs written in a high-level programming language. Typical techniques used to do the preparation work are preprocessing, interpretation, and compilation.

1.4.1 Interpretation and compilation

Interpretation of a program is the direct execution of one statement at a time sequentially by the interpreter. **Compilation**, on the other hand, first translates all the statements of a program into assembly language code or machine code before any statement is executed. The compiler does not execute the program, and we need a separate loader to load the program to the execution environment (hardware or a runtime system that simulates hardware). A **compiler** allows program modules to be compiled separately. A **linker** can be used to combine the machine codes that were separately compiled into one machine code program before we load the program to the execution environment. Figure 1.7 shows typical processing phases of programs using compilation.

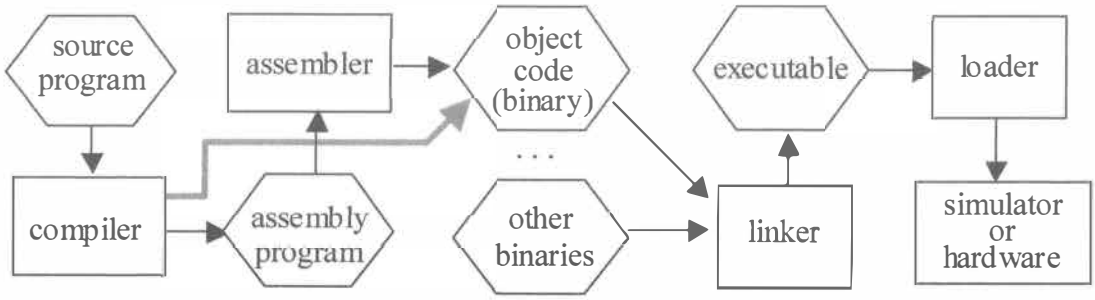


Figure 1.7. Compilation-based program processing.

The advantage of interpretation is that a separate program-processing phase (compilation) is saved in the program development cycle. This feature allows the program to be updated without stopping the system. The interpreter can immediately and accurately locate any errors. However, the execution speed with interpretation is slower than the execution of machine code after the compilation. It is also more difficult to interpret programs written in very high-level languages.

To make use of the advantages of both compilation and interpretation, Java offers a combined solution to program processing. As shown in Figure 1.8, Java source program is first translated by a compiler into an assembly language-like intermediate code, called **bytecode**. The bytecode is then interpreted by an interpreter called **Java Virtual Machine** (JVM). The advantage of using the intermediate code is that the compiler will be independent of the machine on which the program is executed. Thus, only a single compiler needs to be written for all Java programs running on any machine. Another advantage is that the bytecode is a low-level language that can be interpreted easily. It thus makes JVM small enough to be integrated into an Internet browser. In other words, Java bytecode programs can be transferred over the Internet and executed in the client's browser. This ability makes Java the dominant language for Internet application development and implementation.

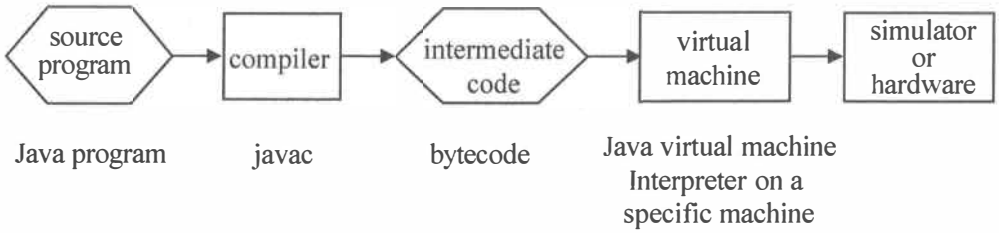


Figure 1.8. Java processing environment.

Microsoft's Visual Studio extends the Java environment's compilation and interpretation processing into a two-step compilation process. As shown in Figure 1.9, in the first compilation step, a high-level language program is compiled to a low-level language called **intermediate language** (IL). The IL is similar in appearance to an assembly language. Programs in IL are managed and executed by the **common language runtime** (CLR). Similar to Java's bytecode, the purpose of IL is to make the CLR independent of the high-level programming languages. Compared to JVM, CLR has a richer type system, which makes it possible to support many different programming languages rather than one language only, as on JVM. On the basis of the type system, nearly any programming language, say X, can be easily integrated into the system. All we need to do is to write a compiler that translates the programs of the X language into the IL. Before an IL program can be executed, it must be translated to the machine code (instructions) of the processor on which the programs are executing. The translation job is done by a **Just-In-Time** (JIT) compiler embedded in CLR. The JIT compiler uses a strategy of compile-when-used, and it dynamically allocates blocks of

memory for internal data structures when each method is first called. In other words, JIT compilation lies between the complete compilation and statement-by-statement interpretation.

Unlike the Java environment, Visual Studio is language agnostic. Although C# is considered its flagship, Visual Studio is not designed for a specific language. Developers are open to use the common libraries and functionality of the environment while coding their high-level application in the language of their choice.

1.4.2 Preprocessing: macro and inlining

Many programming languages allow programmers to write macros or **inline** procedures that preserve the structure and readability of programs while retaining the efficiency. The purpose of **program preprocessing** is to support macros and inline procedures. The preprocessing phase is prior to the code translation to the assembly or machine code. The preprocessor is normally a part of the compiler.

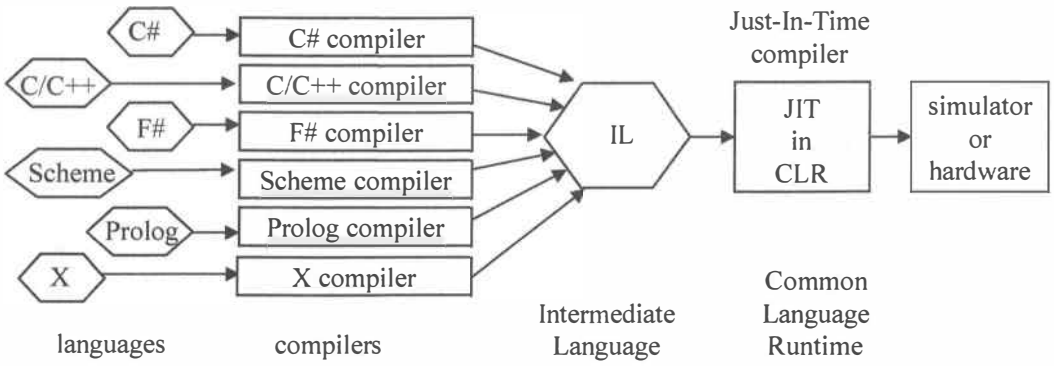


Figure 1.9. Microsoft's Visual Studio. Net programming environment.

In different programming languages, macros can be defined in different ways. In Scheme, we can introduce a **macro** by simply adding a keyword macro in the head of a procedure definition. In C/C++, a macro is introduced by a construct, which is different from a procedure definition:

```
#define name body
```

The `#define` construct associates the code in the *body* part to the identifier *name* part. The body can be a simple expression or rather complex procedure-like code, where parameter passing is allowed. For example:

```
#define MAXVAL 100
#define QUADFN(a,b) a*a + b*b - 2*a*b
...
x = MAXVAL + QUADFN(5,16);
y = MAXVAL - QUADFN(1,3);
```

The last two statements will be replaced by the macro preprocessor as:

```
x = 100 + 5*5 + 16*16 - 2*5*16;
y = 100 - 1*1 + 3*3 - 2*1*3;
```

where `MAXVAL` is replaced by `100` and `QUADFN(5,16)` is replaced by `a*a + b*b - 2*a*b`, with parameter passing: `a` is given the value `5` and `b` is given the value `16`, and `a` is given the value `1` and `b` is given the value `3`, in the two statements, respectively.

Macros are more efficient than procedure calls because the body part of the macro is copied into the statement where the macro is called. No control flow change is necessary. This process is also called **inlining**. On the other hand, if the macro is implemented as a procedure/function:

```
#define MAXVAL 100
int QUADFN(a,b) {return a*a + b*b - 2*a*b;}
...
x = MAXVAL + QUADFN(5,16);
y = MAXVAL - QUADFN(1,3);
```

a procedure will cause a control flow change that usually needs to save the processor environment, including registers and the program counter (return address), onto the stack. This process sometimes is called **out-lining**. Obviously, inlining is much more efficient than out-lining.

Macro preprocessing is a very simple and straightforward substitution of the macro *body* for the macro *name*. It is the programmer's responsibility to make sure that such a simple substitution will produce the correct code. A slight overlook could lead to a programming error. For example, if we want to define a macro to obtain the absolute value of a variable, we write the following macro in C:

```
#define abs(a) ((a<0) ? -a : a)
```

where the C statement `((a<0) ? -a : a)` returns `-a` if `a<0`; otherwise, it returns `a`.

This macro definition of the absolute-value function looks correct, but it is not. For example, if we call the macro in the following statement:

```
j = abs(2+5);    // we expect 7 to be assigned to j.
```

The statement does produce a correct result. However, if we call the macro in the following statement:

```
j = abs(2-5);    // we expect +3 to be assigned to j.
```

The statement will produce an incorrect result. The reason is that the macro-processor will replace "`abs(2-5)`" by "`((a<0) ? -a : a)`" and then replace the parameter "`a`" by "`2-5`", resulting in the statement:

```
j = ((2-5 < 0) ? -2-5 : 2-5);
```

Since `(2-5 < 0)` is true, this statement will produce the result of `-2-5 = -7`, which is assigned to the variable `j`. Obviously, this result is incorrect, because we expect `+3` to be assigned to `j`.

Examine a further example. If we write a statement:

```
j = abs(-2-5);
```

The macro-processor will replace "`abs(-2-5)`" by "`((a<0) ? -a : a)`" and then replace the parameter "`a`" by "`-2-5`," resulting in the statement:

```
j = ((-2-5 < 0) ? --2-5 : -2-5);
```

The "`--2`" in the preprocessed statement may result in a compiler error.

The problem in this `abs(a)` macro is that we expect the expression that replaces `a` to be a unit. All operations within the unit should be done before `-a` is performed. This is the case when we write a function or procedure. However, the macro replacement does not guarantee this order of operation, and the programmer must understand the difference. A correct version of the `abs(a)` macro is:

```
#define abs(a) ((a<0) ? -(a) : a)    // correct version of abs(a) macro
```

Putting the "a" in a pair of parentheses guarantees that the operations within `a` are completed before `-a` is performed.

Owing to the nature of simple textual replacement, a macro may cause side effects. A **side effect** is an unexpected or unwanted modification of a state. When we use a global variable to pass values between the caller procedure and the called procedure, a modification of the global variable in the called procedure is a side effect.

Next, examine the side effect in the correctly defined `abs(a)` macro discussed earlier. If we call the macro in the following code:

```
i = 3;
j = abs(++i);    // we expect 4 to be assigned to j.
```

According to the way a macro is preprocessed, the following code will be produced by the macro-processor:

```
i = 3;
j = ((++i < 0) ? -(++i) : ++i);
```

When the second statement is executed, variable `i` will be incremented twice. According to the definition of the expression `++i`, variable `i` will be incremented every time before `i` is accessed. There is another similar expression in C/C++: `i++`, which increments variable `i` every time after `i` is accessed. Similarly, C/C++ have expressions `--i` and `i--`, etc.

In the earlier statement, variable `i` will be accessed twice: first when we assess `(++i < 0)`, and then the second `++i` will be accessed after the condition is assessed as false. As a result, number 5, instead of 4, will be assigned to the variable `j`.

Macros can be used to bring (inline) a piece of assembly language code into a high-level language program. For example:

```
#define PORTIO __asm \
{                    \
  __asm mov al, 2    \
  __asm mov dx, 0xD007   \
  __asm out al, dx   \
}
```

The back slash \ means that there is no line break when performing macro replacement. When we make a macro call to `PORTIO` in the program, this macro call will be replaced by:

```
__asm { __asm mov al, 2 __asm mov dx, 0xD007 __asm out al, dx }
```

where `__asm` is the C/C++ keyword for assembly instructions. If the compiler is translating the program into assembly code, nothing needs to be done with a line that starts with `__asm`. If the compiler is translating the program into machine code, the compiler will call the assembler to translate this line into machine code.

For the execution of macros, different runtimes (execution engines) may process the translated code indifferent order. As an example, we consider the following code, which has two pairs of functions and macros.

```
/* Side effect, Macro versus Function */
#include <stdio.h>
#pragma warning(disable : 4996) // comment out if not in Visual Studio
#define mac1(a,b) a*a + b*b - 2*a*b
```

```
#define mac2(a,b) a*a*a + b*b*b - 2*a*b
int func1(int a, int b) { return (a*a + b*b - 2 * a*b); }
int func2(int a, int b) { return (a*a*a + b*b*b - 2 * a*b); }
main() {
   int a, b, i, j, fncout, macout;
   printf("Please enter two integers\n");
   scanf("%d%d", &a, &b);
   i = a;
   j = b;
   fncout = func1(++i, ++j);
   printf("i = %d\tj = %d\n", i, j);
   i = a;
   j = b;
   macout = mac1(++i, ++j);
   printf("i = %d\tj = %d\n", i, j);
   printf("fncout1 = %d\tmacout1 = %d\n\n", fncout, macout);
   i = a;
   j = b;
   fncout = func2(++i, ++j);
   printf("i = %d\tj = %d\n", i, j);
   i = a;
   j = b;
   macout = mac2(++i, ++j);
   printf("i = %d\tj = %d\n", i, j);
   printf("fncout2 = %d\tmacout2 = %d\n", fncout, macout);
}
```

Each of the pairs, (macl , func1) and (mac2, func2), is supposed to implement the same functionality and give the same output. If we run the code on Visual Studio 2013, the outputs are as follows:

```
Please enter two integers
5
6
i = 6   j = 7
i = 8   j = 9
fncout1 = 1     macout1 = 1

i = 6   j = 7
i = 9   j = 10
fncout2 = 475   macout2 = 1549
```

How are these outputs generated? We will manually trace the execution as follows. After the macro replacement, the two macro calls will be replaced by the following statements, respectively:

macout = mac1(++i, ++j);→ macout = ++i*++i + ++j*++j - 2*++i*++j;

macout = mac2(++i, ++j);→ macout = ++i*++i*++i + ++j*++j*++j - 2*++i*++j;

In Visual Studio, the order of execution is to apply to all the unary operations (++) first, in the order of their appearances, before doing any arithmetic calculations at all.

For func1 and mac1, the calculations are done as follows, respectively:

```
func1: 6*6 + 7*7 - 2*6*7 = 36 + 49 - 84 = 1
mac1:  8*8 + 9*9 - 2*8*9 = 64 + 81 - 144 = 1
```

In this example, func1 and mac1 happen to have generated the same result. This is pure coincidence. For func2 and mac2, the calculations are done as follows, respectively, which generated different results:

```
func1: 6*6*6 + 7*7*7 - 2*6*7 = 216 + 343 - 84 = 475
mac1:  9*9*9 + 10*10*10 - 2*9*10 = 729 + 1000 - 180 = 1549
```

Now, we run the same code on GNU GCC. The following results are generated:

```
5
6
i = 6   j = 7
i = 8   j = 9
fncout1 = 1     macout1 = -31

i = 6   j = 7
i = 9   j = 10
fncout2 = 475   macout2 = 788
```

As can be observed, the function implementations generate the same results as that generated on Visual Studio. However, the outputs of the macros on GCC are completely different from that on Visual Studio. The reason is that GCC uses a different order of evaluations. Now we explain how macout1 = –31 and macout2 = 788 are generated.

GCC calculates the unary operations for the operands of each operator in pair and makes the same variable in the operation to have the same value.

For mac1: `macout = ++i*++i + ++j*++j - 2*++i*++j;` the macro is evaluated as follows:

```
mac1:  7*7 + 8*8 - (2*8)*9 = 49 + 64 - 144 = -31
```

where the value of the first pair of variables ++i is 7 and the value of the second pair of variables ++i is 8. Then, 2 and ++i will form a pair, resulting (2*8), and its result will form a pair with the last ++i, which obtains a value 9.

For mac2: macout = ++i*++i*++i + ++j*++j*++j - 2*++i*++j; the macro is evaluated as follows:

```
mac1:  (7*7)*8 + (8*8)*9 - (2*9)*10 = 392 + 576 - 180 = 788
```

Notice that the macros generate different values when there exist side effects, for example, when ++i is used as the input. If no side effects are involved, the macros should generate the same results as their function implementations, and macros should generate the same results running different execution environments.

This discussion shows that macros are similar to, and yet different from, procedures and functions, and that both writing macros (ensuring correctness) and using macros (understanding the possible side effects) can be difficult and challenging. Can we write and use macros (obtain better efficiency) exactly in the same way as we write and use procedures and functions (obtain the same simplicity)? Efforts have been made to do that, and we are making good progress. As mentioned earlier, in Scheme, we can write macros in the

same way in which we write procedures. However, we still cannot use macros in exactly the same way we use procedures. This issue will be discussed in the chapter on Scheme programming (Chapter 4). In C++, "inline" procedures/functions are allowed. All that is needed is to add a keyword –inline (in C) and **inline** (in C++) in front of the definition of a procedure/function. For example:

```
_inline int sum(int i, int j) {
  return i + j;
}
```

However, the inline procedure/function is slightly different from a macro. A macro call will always be replaced by the body of the macro definition in the preprocessing stage. The macro-processor will not check the syntax and semantics of the definition. On the other hand, for an inline procedure/function call, the compiler (not the preprocessor) will try to replace the call by its body. There are two possibilities: If the procedure/function is simple enough, the compiler can do the replacement and can guarantee the correctness of the replacement; that is, the inlined code must work exactly in the same way as an ordinary procedure/function call. If the body of the procedure is too complicated (e.g., uses many variables and complex expressions), the compiler will not perform inlining. The inline procedure in this case will work like an ordinary procedure.

Java has a similar mechanism called **final method**. If a method is declared **final**, it cannot be overridden in a subclass. This is because a call to a final method may have been replaced by its body during compilation, and thus late binding cannot associate the call with a method redefined in a subclass.

*1.5 Program development

This section briefly introduces the main steps of the program development process, including requirement, specification, design, implementation, testing, proof, and related techniques. Understanding these steps and the related techniques involved is extremely important. However, a more detailed discussion of these topics is beyond the scope of this text.

1.5.1 Program development process

Development of a program normally goes through the following process:

Requirement is an informal description, from the user's point of view, of the functionality of the program to be developed. Normally, requirements are informal. For example, "I want the program to sort all numbers in an increasing order" is an informal requirement.

Specification is a formal or semiformal description of the requirement from the developer's point of view. The specification describes what needs to be done at the functionality level. A formal specification is defined by a set of preconditions on the inputs and a set of post conditions on the outputs. For example, a formal specification of "sorting numbers" can be defined as follows:

Input: $(x_1, x_2, ..., x_n)$

Preconditions on inputs:

$(\forall x_i)(x_i \in I)$, where I is the set of all integer numbers.

Output: $(x_{i1}, x_{i2}, ..., x_{in})$

Postconditions on outputs:

$(\forall x_{ij})(\forall x_{ik})\ ((x_{ij} \in I \wedge x_{ij} \in I \wedge j < k) \rightarrow x_{ij} \leq x_{ik})$.

The **design** step translates what needs to be done (functional specification) into how to do it (procedural steps or algorithm). For example, devising a sorting algorithm to meet the specification belongs to the design step. An algorithm is usually written in a pseudo language that does not have the mechanical details of a programming language. A **pseudo language** focuses on clear and accurate communication with humans, instead of humans and machines.

The **implementation** step actualizes or instantiates the design step using a real programming language. Writing programs in real programming languages is the main topic of this text and will be discussed in much more detail in the following chapters.

The **testing and correctness proof** step tries to show that the implementation does the work defined in the design step or in the specification step. The development process has to return to the implementation or design steps if the implementation does not meet the requirements of the design or the specification.

The **verification and validation** step tries to show that the implementation meets the specification or the user's requirements. The development has to return to design or specification steps if necessary.

In fact, numerous refined phases and iterations within and between these steps can occur during the entire development cycle.

1.5.2 Program testing

In this and the next subsections, we present more details of the testing and correctness proof step, and related techniques in the program development process.

A **test case** is a set of inputs to a program and the expected outputs that the program will produce if the program is correct and the set of inputs is applied to the program. We also use the **input case** to refer to the input part of a test case. **Program testing** is the process of executing a program by applying predesigned test cases with the intention of finding programming errors in a given environment. Testing is related to the environment. For example, assume that your program runs correctly on a GNU GCC C/C++ environment. If you move your program to a Visual Studio C/C++ environment, you need to retest your program because the environment has been changed. **Debugging** is the process of finding the locations and the causes of errors and fixing them.

If a program has a limited number of possible inputs, we could choose all possible inputs as the test cases to test the program. This approach is called **exhaustive testing**. If the outputs are correct for all test cases, we have proved the program's correctness.

However, in many cases, a program can take an unlimited number of inputs, or the number of test cases is too big to conduct exhaustive testing. We have two ways to deal with the problem: use incomplete testing or use a formal method to prove the program's correctness.

If incomplete testing is used, the question is how we should choose (generate) the limited subset of test cases. Functional testing and structural testing are two major approaches used to generate incomplete test cases. In **functional testing**, we try to generate a subset of test cases that can cover (test) all functions or subfunctions of the program under test. Functional testing is also called **black-box testing** because we generate test cases without looking into the structure or source code of the program under test. In **structural testing**, we try to generate a subset of test cases that can cover (test) particular structures of the program under test. For example, we can try to cover all:

- statements in the program,
- branches of the control flow, or

- paths from the program's entry point to the program's exit point.

Structural testing is also called **glass-box testing** or **white-box testing** because it requires detailed knowledge of the control structure and the source code of the program under test.

Both functional testing and structural testing can be considered so-called partition testing. **Partition testing** tries to divide the input domain of the program under test into different groups so that the inputs in the same group are equivalent in terms of their testing capacity. For example, if we are conducting functional testing, we can consider all inputs that will cause the same subfunction to be executed as a group. On the other hand, if we are conducting structural testing, we can consider all inputs that will cause the same program path to be executed as a group. Then, we choose:

- one or several test cases from each group of inputs, and
- one or several inputs on the boundaries between the groups

to test the program. For example, if an integer input is partitioned into two groups—negative and nonnegative—then zero is on the boundary and must be chosen as an input case. Obviously, if the partition is done properly, partition testing will have a fair coverage of all parts of the program under test.

Program testing is a topic that can take an entire semester to study. We do not attempt to teach the complete program testing theory and practice in this section. In the rest of the section, we will use a simple example to illustrate some of the important concepts related to the topic.

Example: The `gcd` function in the following C program is supposed to find the greatest common divisor of two nonnegative integers.

```
#include <stdio.h>
int gcd (int n0, int m0) { // n0 >= 0 and m0 >= 0 and (n0 ≠ 0 or m0 ≠ 0)
  int n, m;
  n = n0;
  m = m0;
  while (n != 0 && n != m) { // (n ≠ 0) AND (n ≠ m)
      if (n < m)
           m = m - n;
      else
           n = n - m;
  }
  return m;
}
main() {
  int i, j, k;
  scanf("%d\n%d", &i, &j);   // input integers i and j from the keyboard
  k = gcd(i, j);             // call function gcd
  printf("%d\n", k);         // output the greatest common divisor found
}
```

First, we "randomly" pick out the following test cases to test the program.

```
Input (i, j)        Expected Output k                    Actual
Output k
```

(6, 9)	3	3
(10, 5)	5	5
(0, 4)	4	4
(5, 7)	1	1
(8, 29)	1	1

We find that the actual output equals the expected output in all these test cases. Now the question is, is this program correct?

As we know, testing can never prove the correctness of a program unless all possible test cases have been used to test the program. However, a set of test cases that can cover different parts of the program can certainly increase the confidence of the correctness of the program.

Now we apply structural testing to generate systematically a better set of test cases. We assume that we aim at covering all the branches in the `gcd` function. To make sure we cover all branches, we first draw the function's flowchart, as shown in Figure 1.10. A **flowchart** is an abstraction of a program that outlines the control flows of the program in a graphic form. In the flowchart in Figure 1.10, each branch is marked with a circled number.

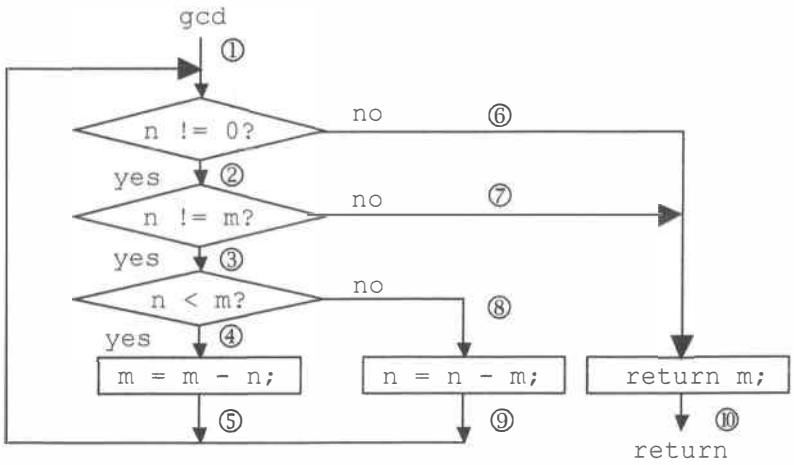


Figure 1.10. Flowchart of `gcd` function.

To obtain a good coverage of the branches, we need to analyze the features of the input domain of the program. For the given program, the input domain has the following features:

- The program takes two nonnegative integers as input. The boundary value is 0.
- The branches of the program are controlled by the relative values of a pair of integers. We should consider choosing equal and unequal pairs of numbers.
- The great common divisors are related to the divisibility of integers, or the prime and nonprime numbers. We should choose some prime numbers and some nonprime numbers.

On the basis of the analysis, we can choose, for example, these values for both `i` and `j`: boundary value 0, two prime numbers 2 and 3, and two nonprime numbers 9 and 10.

The combination of the two inputs generates the following input cases:

```
(0, 0) // This case is not allowed according to the precondition.
(0, 2), (0, 3), (0, 9), (0, 10)
(2, 0), (2 2), (2, 3), (2, 9), (2, 10)
(3, 0), (3, 2), (3, 3), (3, 9), (3, 10)
(9, 0), (9, 2), (9, 3), (9, 9), (9, 10)
(10, 0), (10, 2), (10, 3), (10, 9), (10, 10)
```

We can apply all these test cases to test the program. We can also reduce the number of test cases by partitioning the input cases into groups: two input cases belong to the same group if they cover the same branches in the same order. Table 1.2 lists the groups, a representative from each group, the expected output of the representative input case, and the branches covered by the groups.

Groups partitioned	Representative	Expected gcd output	Branches covered
(0, 2),(0, 3),(0, 9),(0, 10)	(0, 2)	2	①⑥⑩
(2 2),(3, 3), (9, 9),(10, 10)	(2 2)	2	①②⑦⑩
(2, 0),(3, 0), (9, 0),(10, 0)	(2, 0)	2	①②③⑧⑨
(2, 3),(2, 9), (3, 10), (9, 10)	(2, 3)	1	①②③④⑤⑧⑨⑩
(2, 10), (3, 9)	(2, 10)	2	①②③④⑤⑦⑩
(3, 2),(9, 2), (10, 3),(10, 9)	(3, 2)	1	①②③⑧⑨④⑤⑩
(9, 3), (10, 2)	(9, 3)	3	①②③⑦⑧⑨⑩

Table 1.2. Input case partitions and branch coverage.

If we choose the representative input from each group and the expected output as the test cases, we obtain a test case set that can cover all the branches in the flowchart. Applying this set of test cases, we will find that the input case (2, 0) will not be able to produce the expected output. In fact, a dead-looping situation will occur. Thus, we successfully find that the program is incorrect.

1.5.3 Correctness proof

To prove the **correctness** of a program, we need to prove that, for all predefined inputs (inputs that meet the preconditions), the program produces correct outputs (outputs that meet the postconditions).

Program proof consists of two steps: **partial correctness** and **termination**. A program is partially correct if an input that satisfies the preconditions is applied to the program, and if the program terminates, the output satisfies the postconditions. A program terminates if, for all inputs that meet the preconditions, the program stops in finite execution steps. A program is totally correct (**total correctness**) if the program is partially correct and the program terminates.

The idea of partial correctness proof is to prove that any input that meets the preconditions at the program entry point will be processed, step by step through the statements between the entry point and the exit point, and the postconditions will be satisfied at the exit point. Obviously, if there is no loop in the program, it is not too hard to do the step-by-step proof. However, if there is a loop in the program, the step-by-step approach will not work. In this case, we need to use the loop invariant technique to derive the condition to be proved through the loop. A **loop invariant** is a condition that is true in each iteration of the loop. To prove a condition is a loop invariant, we can use mathematical induction: We prove that the condition is true when the control enters the loop for the first time (iteration 1). We assume that the condition is true at iteration `k`, and prove that the condition will be true at the iteration `k+1`.

Finding the loop invariant that can lead to the postconditions is the most challenging task of the correctness proof. You must have a deep understanding of the problem in order to define a proper loop invariant.

Proving the termination is easy if we design the loops following these guidelines. The loop variable:

- is an enumerable variable (e.g., an integer);
- has a lower bound (e.g., will be greater than or equal to zero);
- will strictly decrease. **Strictly decrease** means that the value of the loop variable must be strictly less than the value of the variable in the previous iteration. The "<" relation is strict, while "≤" is not.

If you do not follow the guidelines, you may have trouble proving the termination of even a simple program. In an exercise given at the end of the chapter, a very simple example is given where many inputs have been tried, and the program always stops. However, so far, nobody can prove that the program terminates for all inputs.

Now we will study a similar example that we used in the last section to illustrate the proof concepts that we discussed here. The program is given in a pseudo language. Since we do not actually have to execute the program, we do not have to give the program in a real programming language.

```
gcd (n0, m0)
// precondition: (n0≥ 0 ∧ m0≥ 0) ∧ (n0≠ 0 ∨ m0≠ 0) ∧ (n0, m0 are integer)
   n = n0;
   m = m0;
   while n ≠ 0 do
// loop invariant:   (n ≥∧ m ≥ 0) ∧ (n ≠ 0 ∨ m ≠ 0) ∧
//                   max{u: u|n and u|m} = max{u: u|n0 and u|m0}
       if n ≤ m
       then m = m - n
       else swap(n, m)
   output(m)
// postconditions:   m = max{u: u|n0 and u|m0}
```

To prove the partial correctness, we need to prove, for any integer pair `(n0, m0)` that meets the preconditions, the loop invariant is true in every iteration of the loop. When the control completes all the iterations of the loop and reaches the exit point of the program, the postconditions will be true. As discussed, finding the loop invariant is the most difficult part. Now we are given the condition that should be a loop invariant, and we only need to prove that the condition is indeed a loop invariant. The given condition is:

```
(n ≥∧ m ≥ 0) ∧ (n ≠ 0 ∨ m ≠ 0) ∧
max{u: u|n and u|m} = max{u: u|n0 and u|m0}
```

We need to prove it is a loop invariant. We can use mathematical induction to prove it in the following steps:

(1) Prove the condition is true at the iteration `1`: it is obvious.

(2) Assume the condition is true at iteration `k`.

(3) Prove the condition is true at iteration `k+1`.

Since the conversion made in each iteration is:

```
gcd(n, m) ⇒ gcd(n, m - n), or
gcd(n, m) ⇒ gcd(m, n)
```

According to mathematical facts:

```
gcd(n, m) = gcd(n, m - n), and
gcd(n, m) = gcd(m, n)
```

Thus, from iteration `k` to iteration `k+1`, the condition will remain to be true. Therefore, we have proved the condition is a loop invariant.

We then need to prove that the loop invariant leads to the postconditions. It can be easily seen from the program that if the loop invariant is true when the control leaves the loop, the postconditions will indeed be true. Thus, we have proved the partial correctness of the program.

To prove that the program terminates, we can use the following facts:

(1) The loop variable is `(n, m)`. The loop variable is enumerable.

(2) If we consider the dictionary order, that is:

```
(n, m) > (n, m - n), if n ≥ m // e.g. (3, 6) > (3, 3) in dictionary order
(n, m) > (m, n),     if n > m. // (6, 3) > (3, 6) in dictionary order
```

we can see that there is a strictly decreasing order on the loop variable `(n, m)` based on the dictionary order.

(3) There is a lower bound on the value that `(n, m)` can take, that is `(0, 0)`.

Since the loop variable `(n, m)` is enumerable, it decreases strictly, and there is a lower bound `(0, 0)`; the loop must terminate.

In an exercise given at the end of the chapter, a variation of the `gcd` program is suggested. Try to prove its partial correctness and its termination.

1.6 Summary

In this chapter, we introduced in Section 1.1 the concepts of the four major programming paradigms: imperative, object-oriented, functional, and logic. We looked at the impact of language features on the performance of the programs written in the language. We briefly discussed the development of languages and the relationships among different languages. We then discussed in Section 1.2 the structures of programs at four levels: lexical, syntactic, contextual, and semantic. We illustrated our discussion on the lexical and syntactic levels by introducing the BNF notation and syntax graph. We used BNF notation to define the lexical and syntactic structures of a simple programming language. In Section 1.3, we studied the important concepts in programming languages, including data types, type checking, type equivalence, and type conversion. Orthogonality was used to examine the regularity and simplicity of programming languages. Finally, in Section 1.4, we briefly discussed program processing via compilation, interpretation, and a combination of the two techniques. The emphasis was on the macro and inline procedures/functions in C/C++. We studied how to define and use macros, and what their strengths and weaknesses are when compared to ordinary procedures/functions. Section 1.5 outlined the program development process and discussed programming testing and proof techniques through examples.

1.7 Homework and programming exercises

1. Multiple Choice. Choose only one answer for each question. Choose the best answer if more than one answer is acceptable.

1.1 Stored Program Concept (von Neumann machine) is one of the most fundamental concepts in computer science. What programming paradigm most closely follows this concept?

❑ imperative ❑ object-oriented ❑ functional ❑ logic

1.2 What computing paradigm can solve a problem by describing the requirements, without writing code in a step-wise fashion to solve the problem.

❑ imperative ❑ functional ❑ object-oriented ❑ logic

1.3 What computing paradigm enforces stateless (no variable allowed) programming?

❑ imperative ❑ object-oriented ❑ functional ❑ service-oriented

1.4 What is a feature of object-oriented computing?

❑ stateless ❑ state encapsulation ❑ platform-independent ❑ side-effect free

1.5 In contrast to Web 1.0, what is the key function of Web 2.0?

❑ Web is the computing platform ❑ Web supports graphic display

❑ Web supports semantic analysis ❑ Web is accessed over HTTP protocol

1.6 Because of hardware constraints, early programming languages emphasized

❑ efficiency ❑ orthogonality ❑ reliability ❑ readability

1.7 What factor is generally considered more important in modern programming language design?

❑ readability ❑ writability ❑ efficiency ❑ None

1.8 The main idea of structured programming is to

❑ reduce the types of control structures. ❑ increase the types of control structures.

❑ make programs execute faster. ❑ use BNF to define the syntactic structure.

1.9 Implicit type conversion is commonly refer to as:

❑ typing coercion casting paradigm

1.10 In the following pseudo code, which programming language allows the mixed use of data types?

```
int i = 1; char c = 'a';    // declaration and initialization
c = c + i;                  // execution of an assignment statement
```

❑ Ada ❑ C ❑ Java ❑ All of them

1.11 In the layers of programming language structure, which layer performs type checking?

❑ lexical ❑ syntactic ❑ contextual ❑ semantic

1.12 How many different identifiers can the following BNF ruleset generate?

```
<char>       ::= a | b | c | ... | x | y | z
<identifier> ::= <char> | <char> <identifier>
```

❑ 0 ❑ 1 ❑ 26 ❑ more than 26

1.13 Which of the following statement is correct if a language is strongly typed. Select all that apply.

- ❑ Each variable in a program has a single type associated with it.
- ❑ Variable type can be unknown at compilation time.
- ❑ Type errors are always reported.
- ❑ Coercion is automatically allowed.

1.14 Which command (construct) has a loop when expressed in syntax graphs?

❑ if-else ❑ switch ❑ for ❑ while

1.15 The contextual structure of a programming language defines

- ❑ how to form lexical units from characters.
- ❑ how to put lexical units together to form statements.
- ❑ the static semantics that will be checked by the compiler.
- ❑ the dynamic semantics of programs under execution.

1.16 If a program contains an error that divides a number by zero at the execution time. This error is a

❑ lexical error ❑ syntactic error ❑ contextual error ❑ semantic error

1.17 Interpretation is not efficient if

- ❑ the source program is small.
- ❑ the source program is written in an assembly language.
- ❑ the difference between source and destination is small.
- ❑ multi-module programming is used.

1.18 What is the difference between an inline function and a macro in C++?

- ❑ There is no difference between them.
- ❑ The *inline* functions are for Java only. There are no inline functions in C++.
- ❑ *Inlining* is a suggestion to the compiler, while a *macro* definition will be enforced.
- ❑ A *macro* definition is a suggestion to the compiler, while *inlining* will be enforced.

1.19 Macros-Processing takes place during which phase?

❑ Editing ❑ Preprocessing ❑ Compilation ❑ Execution

1.20 Assume a function requires 20 lines of machine code and will be called 10 times in the main program. You can choose to implement it using a function definition or a macro definition. Compared with the function definition, macro definition will lead the compiler to generate, for the entire program,

- ❑ a longer machine code but with shorter execution time.
- ❑ shorter machine code but with longer execution time.
- ❑ the same length of machine code, with shorter execution time.
- ❑ the same length of machine code, with longer execution time.

2. Compare and contrast the four programming paradigms: imperative, object-oriented, functional, and logic.

3. Use the library and Internet resources to compile a table of programming languages. The table must include all programming languages mentioned in Section 1.1.3 on the development of programming languages. The table should contain the following columns:
   - name of the programming language,
   - year the language was first announced,
   - authors/inventors of the language,
   - predecessor languages (e.g., C++ and Smalltalk are predecessors of Java),
   - programming paradigms (e.g., Java belongs to imperative and object-oriented programming paradigms).

   The table should be sorted by year.

4. What is strong type checking, and what are its advantages? List a few languages that use nearly strong type checking in their program compilations.

5. What is weak type checking, and what are its advantages? List a few languages that use weak type checking in their program compilations.

6. What is orthogonality? What are the differences between compositional, sort, and number orthogonality?

7. Compare and contrast a macro and an inline function in C++. Which one is more efficient in execution time? Which one is easier for programmers to write?

8. Compare and contrast the C++ inline function and Java's final method.

9. Which type equivalence leads to strong type checking, structural equivalence, or name equivalence? Explain your answer.

10. Use BNF notation to define a small programming language that includes the definition of variable, math-operator, math-expression, condition-operator, condition-expression, assignment statement, loop statement, switch statement, and a block of statements. A variable must start and end with a letter (an unusual requirement). A single letter is a valid variable. The definition of the expression must be able to handle nested expressions like `2*(x + y)` and `5*((u + v)*(x - y))`. The language must include the following statements:

    Assignment: It assigns (the value of) a variable or an expression to a variable. For example, `x = 2*(y + z)`.

    Conditional: `if-then` and `if-then-else`. The condition in the statement can be simply defined as an expression.

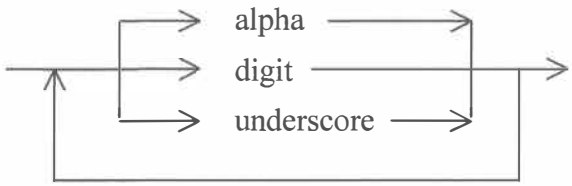
    For-loop: For example, `for (i = 0; i <10; i=i+1){`a block of statements`}`

    Switch: It must have an unlimited number of cases.

    Statement: A statement can be an assignment, conditional, for-loop, or switch statement.

Block: One statement is a block. Zero or multiple statements in curly braces and separated by ";" is also a block, according to number orthogonality. For example, `i=i+2;` is a block. `{i=i+2; for (k=0; k<i; k=k+1) {i=i+1; s=2*i;}}` is also a block.

11. The following syntax graph defines the identifiers of a programming language, where *alpha* is the set of characters "a" through "z" and "A" through "Z," *digit* is the set of characters "0" through "9," and *underscore* is the character "_."



11.1 Which of the following strings can be accepted by the syntax graph (choose all correct answers)?

❑ _FooBar25_ ❑ 2_5fooBar_ ❑ Foo&Bar ❑ 12.5 ❑ Foo2bar

11.2 Give the syntax graph that defines the identifiers always *starting* with a letter from the set *alpha*.

11.3 Give the BNF definition equivalent to the syntax graph of the question above.

12. Given the C program below, answer the following questions.

```
#define min1(x,y) ((x < y) ? x : y)
#define min 10
#include <stdio.h>
int min2(int x, int y) {
    if (x < y) return x;
    else return y;
}
main() {
    int a, b;
    scanf("%d %d", &a, &b);
    if (b < min)
        printf("input out of range\n");
    else {
        a = min1(a, b++);
        printf("a = %d, b = %d\n", a, b);
        a = min2(a, b++);
        printf("a = %d, b = %d\n", a, b);
    }
}
```

12.1 Give the exact C code of the statement "`a = min1(a, b++);`" after macro processing.

12.2 Give the exact C code of the statement "`a = min2(a, b++);`" after macro processing.

12.3 Give the exact C code of the statement "`if (b < min)`" after macro processing.

12.4 Assume 60 and −30 are entered as inputs. What is the exact output of the program?

12.5 Assume 50 and 30 are entered as inputs. What is the exact output of the program?

13. Assume a programming language has two sets of features. S_1 is the set of three different kinds of declarations: (1) plain, (2) initializing, and (3) constant. That is,

(1) `typex i;` (2) `typex i = 5;` (3) `const typex i = 5;` S_2 is the set of data types: (a) `bool`, (b) `int`, (c) `float`, (d) `array`, (e) `char`.

13.1 If the language guarantees sort orthogonality 1 in Figure 1.5, and we know that `int i` is allowed (the combination plain-int), list the allowed combinations of the features of the two sets that can be implied by the sort orthogonality.

13.2 If the language guarantees sort orthogonality in Figure 1.6, and we know that `const array a` is allowed (the combination constant-array), list the allowed combinations of the features of the two sets that can be implied by the sort orthogonality.

13.3 If the language guarantees compositional orthogonality, list the combinations of the two sets of features allowed. Write a simple C program that exercises all the declarations in this question. Each declared variable must be used at least once in the program. The purpose of the program is to test whether C supports compositional orthogonality. The program thus does not have to be semantically meaningful. Test the program on Visual Studio or GNU GCC, and submit a syntax error-free program. *Note:* a variable can be declared only once. You must use different variable names in the declarations.

14. Programming exercise.

You are given the following simple C program.

```
/* assign1.c is the file name of the program. */
#include <stdio.h>      // use C style I/O library function
main () {               // main function
    int i = 0, j = 0; // initialization
    printf("Please enter a 5-digit integer \n");
    scanf("%d", &i);  // input an integer
    i = i % 10000;    // modulo operation
    printf("Please repeat the number you entered\n");
    scanf("%d", &j);  // input an integer
    j = j % 10000;    // modulo operation
    if (i == j)       // conditional statement
        printf("The number you entered is %d\n", i);
    else              // else branch
        printf("The numbers you entered are different\n");
}
```

14.1 Enter the program in a development environment (e.g., Visual C++ or GNU GCC). Save the file as assign1.c. If you are not familiar with any programming environment, please read Section B.2 of Appendix B.

14.2 Compile and execute the program.

14.3 Read Chapter 2, Section 2.1. Modify the given program. Change <stdio.h> to <iostream>. Change printf to cout and change scanf to cin, etc. Save the file as assign1.cpp

14.4 Compile and execute the program.

15. Macros are available in most high-level programming languages. The body of a macro is simply used to replace a macro call during the preprocessing stage in compilation. A macro introduces an "inline" function that is normally more efficient than an "out-line" function. However, macros suffer from side effects, unwanted or unexpected modifications of variables. Macros should be used cautiously. The main purpose of the following program is to demonstrate the differences between a function and a macro. Other purposes include learning different ways of writing comments, formatted input and output, variable declaration and initialization, unary operation ++, macro definition/call, function definition/call, if-then-else and loop structures, etc. Study the following program carefully and answer the following questions.

```
/*  The purpose of this program is to compare and contrast a function
    to a macro. It shows the side effects of a macro and an incorrect
    definition of a macro. The macros/functions abs1(x), abs2(x), and
    abs3(x) are supposed to return the absolute value of x.  */
#define abs1(a) ((a<0) ? -(a) : (a))   // macro definition
#define abs2(a) ((a<0) ? -a : a)       // macro definition
#include <stdio.h>
int abs3(int a) {                      // function definition
    return ((a<0) ? -(a) : (a));  // --> if(a < 0) return -a else return
a;
}
main() {
    int i1 = 0, i2 = 0, i3 = 0, j1 = 0, j2 = 0, j3 = 0;
    printf("Please enter 3 integers\n");
    scanf("%d %d %d", &i1, &i2, &i3);
    while (i1 != 123) {          // 123 is used as sentinel
         j1 = abs1(++i1 - 2);  // call a macro
         j2 = abs2(++i2 - 2);  // call a macro
         j3 = abs3(++i3 - 2);  // call a function
         printf("j1 = %d, j2 = %d, j3 = %d\n", j1, j2, j3);
         printf("Please enter 3 integers\n");
         scanf("%d %d %d", &i1, &i2, &i3);
    }
}
```

15.1 Desk check (manually trace) the program. What would be the outputs of the program when the following sets of inputs are applied to the program?

```
i1, i2, i3 = 9, 9, 9   j1, j2, j3 =
i1, i2, i3 = -5, -5, -5    j1, j2, j3 =
i1, i2, i3 = 0, 0, 0   j1, j2, j3 =
```

15.2 Enter the program into a programming environment and execute the program using the inputs given in the previous question. What are the outputs of the program?

15.3 Explain the side effects that occurred in the program.

15.4 Which macro is incorrectly defined? Explain your answer.

15.5 Change the macros in the program into inline functions.

16. Consider the `gcd` program in Section 1.5.3. What would happen if the else-branch `swap(n, m)` in the program were changed to `n = n - m`?

16.1 Can we still prove the partial correctness?

16.2 Can we prove the termination?

16.3 Write a C program to implement the original algorithm and find a set of test cases to test your program. The test case set must cover the branches of the program.

17. Given the following algorithm in pseudo code:

```
termination(n)// precondition: n is any integer
    while n ≠ 1 do
        if even(n)
        then     n := n/2
        else     n := 3n + 1
    output(n) // postcondition: n = 1
```

17.1 Prove the program is partially correct.

17.2 Discuss whether this program terminates or not.

17.3 Write a C program to implement the algorithm, generate a set of test cases that can cover all branches of the program, and use the set of test cases to test the program.

Chapter 2

The Imperative Programming Languages, C/C++

As we discussed in Chapter 1, the imperative paradigm manipulates named data in a fully specified, fully controlled, and stepwise fashion. It focuses on how to do the job instead of what needs to be done. Imperative programs are algorithmic in nature: do this, do that, and then repeat the operation n times, etc., similar to the instruction manuals of our home appliances. The coincidence between the imperative paradigm and the algorithmic nature makes the imperative paradigm the most popular paradigm among all possible different ways of writing programs. Another major strength of the imperative paradigm is its resemblance to the native language of the computer (von Neumann machine), which makes it efficient to translate and execute the high-level language programs in the imperative paradigm.

In this chapter, we study the imperative programming languages C/C++. We will focus more on C and the non-object-oriented part of C++. We will study about the object-oriented part of C++ in the next chapter.

By the end of this chapter, you should

- have a solid understanding of the imperative paradigm;
- be able to apply the flow control structures of the C language to write programs;
- be able to explain the execution process of C programs on a computer;
- be able to write programs with complex control structures including conditional, loop structures, function call, parameter passing, and recursive structures;
- be able to write programs with complex data types, including arrays, pointers, structures, and collection of structures.

The chapter is organized as follows. Section 2.1 gives a quick tutorial on C/C++ so that students can start their laboratory work on C. Section 2.2 introduces the C/C++ control structures. Sections 2.3, 2.4, and 2.5 discuss data declaration; scope and basic data types; constant, array, pointer, and string; and type construction, including enumeration type, union type, structured types, and file types. Section 2.5 also presents several large program examples using array, pointer, and structures. Section 2.6 studies the functions, function calls, and parameter-passing mechanism in the C language. Section 2.7 teaches a unique technique of understanding recursion and writing recursive programs in four easy steps. Finally, Section 2.8 briefly discusses how to construct C programs into modules and how to use modules to form larger programs.

Imperative programming is largely based on computer architectures and assembly language programming. We will briefly discuss the basic computer architectures as the background material in Appendix A.

2.1 Getting started with C/C++

In this section, we first introduce how to write your first C/C++ program and how to perform input and output. You can develop your programs in different programming environments. Two of the most frequently used programming environments—GNU GCC and MS Visual Studio—are introduced in Appendix B.

2.1.1 Write your first C/C++ program

A C program consists of one or more functions. There are two kinds of functions:

- Built-in functions are prewritten and exist in **libraries**, for example, input and output functions (printf, scanf in C and cin, cout in C++), mathematical functions (abs, sin, cos, sqrt);
- User defined functions are written by the programmers.

The `main()` is a function that all C/C++ programs must have, which is the entry point of the programs (i.e., execution of all programs begin at the first statement of the main function). The shortest and simplest C/C++ program is:

```
main() { }
```

Obviously, this program does nothing. Usually, `main` will have some statements and invoke other user written or library functions to perform some job. For example:

```
/* My first program, file name hello.c
   This program prints "hello world" on the screen */
#include <stdio.h>   // the library functions in stdio will be used
main( ) {
  printf("hello world\n");
}
```

The simple C program will call a library function `printf` to print

```
hello world
```

The first two lines are comments. There are two ways to write comments in C/C++. Multi-line comments can be quoted in a pair of `/*` and `*/`, while single line comments can simply follow double slashes `//`.

The third line of the program specifies what library package will be used. In this program, we use `printf` that is defined in the `stdio` package. In the print statement, "\n" is the "newline" control symbol that puts this output on a line by itself. Another useful control symbol is "\t" for tab.

A C/C++ function may return a value (like a Pascal function) or return no value (like a Pascal procedure). A function may take zero or a number of parameters. The following are several forms of the main function:

```
main () { ... }       // acceptable for C
void main( ) { ... } // in C++, void must be used.
int main() {... return 0;}
void main (int argc, char *argv [ ]) {...}
```

The first and the second forms do not require the function to return a value. The third form requires the function to return an integer value. The fourth form does not require a return value but requires parameter inputs.

You may ask how do we or why do we need to pass values to the function that will be called before any other statements or functions are executed? The answer is that the parameters to the `main()` function allow it to take command line inputs, used to specify, for example, the name of a data file.

For example, if we compiled our first program `hello.c` into the executable code `hello.exe`, we can execute the program by simply typing the name `hello` and the `Enter` key. However, if we have a program, say, `letterReader.exe` that reads a text file, say, `letter.txt`, then we need to execute the program by typing

```
letterReader letter.txt
```

where the file name "`letter.txt`" will be passed to the main function of the program `letterReader` as a parameter.

Unlike Java, C/C++ functions and variables may exist outside any class or functions. These functions and variables are **global**. Function `main()` is always a global function.

2.1.2 Basic input and output functions

Generally, input in C/C++ is reading from a file and output is writing to a file. The keyboard is considered the standard input file and the monitor screen is considered the standard output file. The functions `getchar()` and `putchar(x)` are the basic I/O library functions in C. `getchar()` fetches one character from the keyboard every time it is called, and returns that character as the value of the function. After it reaches the last character of a file, it returns `EOF` (end of file), signifying the end of the file. On the other hand, `putchar(x)` prints one character (the character stored in variable `x`) on the screen every time it is called. The following program reads a line of characters from the keyboard and prints it on the screen. Since both standard input and output are in fact files, a similar program can be used to copy the contents of one file to the other.

```
#include <stdio.h>
main( ) {
   char c;              // declare c as a character type variable
   c = getchar( );      // input one character from the keyboard
   while (c != '\n') {    // while c ≠ the newline control symbol
       putchar(c);      // print to screen
       c = getchar( );    // input another character from the keyboard
   }
}
```

To input a stream of characters, you can use fget:

char *fgets(char *tempstr, int n, FILE *inputfile)

where `tempstr` will point to the string read from the file pointer `inputfile`. Using this read operation, we can read from any text file. For now, we will consider the input file is the keyboard and the file name is stdin. The int variable `n` is the maximum number of characters (bytes) we want to read. The operation `fgets` will stop when `n-1` characters are read or a newline character is read. It returns `null` if nothing is read into the tempstr. Otherwise, it will return the value of `tempstr`.

The following snippet of code shows an example of using fgets.

```
char tempstr[256];
char name[32]; char breed[32]; char owner[32];
```

```
printf("Please enter the dog's info in the following format:\n");
printf("name:breed:owner\n");
fgets(input, sizeof(tempstr), stdin); // read from keyboard
// change '\n' char attached to tempstr into null terminator
tempstr[strlen(tempstr) - 1] = '\0';
char* name = strtok(tempstr, ":");  // parse to ":"
char* breed = strtok(NULL, ":");    // remove separator
char* breed = strtok(tempstr, ":"); // parse to ":"
char* owner = strtok(NULL, ":");    // remove separator
```

where strtok function is used to parse the string and extract the part of the string separated by a separator. In this example, ":" is used. You can use other separators, such as " " or ",". In the program, function call `strtok(tempstr, ":")` will read `tempstr` upto ":", while `strtok(NULL, ":")` will remove ":" and return the remaining string.

2.1.3 Formatted input and output functions

The basic input/output functions allow us to read and write a character at a time. They cannot be used to read and write other types of variables and cannot control the format of output.

The formatted input/output functions are `printf` and `scanf` that take an argument for formatting information. The following program demonstrates a simple use of the functions.

```
/* The program takes a number from the keyboard, processes the number,
 and then prints the result. */
#include <stdio.h>
main () {
  int i;        // i is an integer type variable
  float n = 5.0;// n is floating-point type and is initialized to 5.0
  printf("Please enter an integer\n");
  scanf("%d", &i);  // An integer is expected from the keyboard
  if (i > n)
      n = n + i;
  else
      n = n - i;
  printf("i = %d\t n = %f\n", i, n); //%d, \t, %f, and \n control formats
}
```

Assume a number 12 is entered when `scanf` is executed; the output of the program is

```
i = 12 n = 17.0
```

Generally, the formats of `scanf` and `printf` are

```
scanf ("control sequence", &variable1, &variable2, ... &variablek);
printf ("control sequence", expressions);
```

In the `scanf` function, the ampersand "`&`" is the address-of operator that returns the address of the variable. Using the address-of operator in the argument of a function (e.g., `&i` in `scanf`) enforces the parameter passing by reference. Parameter-passing mechanisms will be explained in detail later in Section 2.6.

In the `printf` function, the "`expressions`" is a list of expressions whose values are to be printed out. Each expression is separated by a comma.

The control sequence includes constant strings to be printed (e.g., "`i =`"), and control symbols to be used to convert the input/output from their numeric values that are stored in the computer to their character format displayed. The control symbol "`%d`" in the `scanf` and `printf` signifies that the next argument in the argument list is to be interpreted as a decimal number and "`%f`" signifies that the next argument is to be interpreted as a floating-point number. The other control characters include "`%c`" for characters and "`%s`" for strings of characters. The symbols "`\n`" and "`\t`" signify the "`newline`" that puts the next output on a new line, and "`tab`" puts the next output after a tab. If there is no "`newline`" or "`tab`" at the end of the first output line, successive calls to `printf` (or `putchar`) will simply append the string or character to the previous output line.

In C++, a different library package and different I/O functions are used. When you use C++ specific features, your program name must have an extension `.cpp` for C++ program. If you use extension `.c`, your program will be considered to be a C program only, and you will obtain compilation errors for C++ specific features.

```
#include <iostream>  // iostream is the C++ library I/O package
using namespace std;
void main() {
  int i, j, sum;         // declaration
  cout << "Enter an integer" << endl;  // prompt for input
  cin >> i;          // read an integer and put in variable i
  cout << "Enter an integer" << endl;
  cin >> j;          // read an integer and put in variable j
  sum = i + j;
  cout << "Sum is " << sum << endl; // print sum
}
```

A scenario of execution of the program is

```
Enter an integer
5
Enter an integer
7
Sum is 12
```

In the program, the functions `cin` and `cout` are C++ standard input and output functions. The function `endl` is the C++ `newline` function corresponding to C's "`\n`".

In C-formatted I/O, a programmer must specify the type of variables to be printed. In C++, the types are automatically recognized. This improvement simplifies printing statements in most cases. Unfortunately, we still have situations where we have to tell the program what type of data to print. For example, a character type in C/C++ is the same as an integer type. How do we tell the program that we want to print a character or an integer? The solution is type casting. The following example shows how a character type variable `c`, initialized to 68 and corresponding to the ASCII character "`D`," is printed as integer 68 and as character "`D`" using `printf` and `std::cout`, respectively. The ASCII table is given in Appendix C.

```
#include <iostream>
using namespace std;
```

```
void main(void) {
   char c = 68;
   printf("c = %d", c);
   printf("\tc = %c\n", c);
   cout<<"c = "<<(int) c;
   cout<<"\tc = "<<c<<endl;
}
```

The output of the program is

```
c = 68 c = D
c = 68 c = D
```

Please note that C++ I/O package `<iostream>` contains all C-styled I/O functions and control symbols like `printf`, `scanf`, "`\n`" and "`\t`."

2.2 Control structures in C/C++

In this section, we briefly review the basic control structures in C/C++, which are similar in all imperative programming languages. The topics we will discuss are

- operators and the order of evaluation,
- basic selection structures,
- multiple selection structures, and
- iteration structures.

Recursion structures are much more complex and will be discussed in Section 2.7 in detail. Chapters 4 and 5 will have even more discussion on this topic.

2.2.1 Operators and the order of evaluation

C/C++ provides a set of **operators** to allow programmers to write complex arithmetic and logical expressions. The **precedence** and **associativity** of C/C++ operators affect the grouping and evaluation of operands in expressions. Table 2.1 summarizes the precedence and associativity (the order in which the operands are evaluated) of C operators, listing them in the order of precedence from highest to lowest. Operators with higher precedence are evaluated first. If two operators have equal precedence (they appear at the same level in the table), they are evaluated according to their associativity, either from right to left or from left to right, as defined in the right column of the table.

Table B.4 in Appendix B gives a complete list of the C/C++ operators, their precedence, description, and associativity.

Please note that C/C++ use a **lazy evaluation** policy; that is, an expression will be evaluated only if its value is needed. For example, if we have an expression

```
(i == 0) && j++
```

the second operand, `j++`, will be evaluated only if `i == 0` is true (nonzero). Thus, `j` will not be incremented if `i == 0` is false (0).

Operators	Type of operation	Associativity
[ ] ( ) . -> postfix ++ and postfix —	Expression	Left to right
prefix ++ and prefix — sizeof & * + – ~ !	Unary	Right to left
Typecasts	Unary	Right to left
* / %	Multiplicative	Left to right
+ –	Additive	Left to right
<<>>	Logical bitwise shift	Left to right
<><= >=	Relational	Left to right
== !=	Equality	Left to right
&	Bitwise-AND	Left to right
^	Bitwise-exclusive-OR	Left to right
|	Bitwise-inclusive-OR	Left to right
&&	Logical-AND	Left to right
||	Logical-OR	Left to right
? :	Conditional-expression	Right to left
= *= /= %= += –= <<= >>= &= ^= |=	Assignment	Right to left
,	Sequential evaluation	Left to right

Table 2.1. C/C++ operators and their precedence.

2.2.2 Basic selection structures (if-then-else and the conditional expression)

The **basic selection structure** in C/C++ is implemented by `if-then` and `if-then-else` statements, which can be defined by the syntax graph in Figure 2.1.

Figure 2.1. Syntax graph for `if-then-else` in C/C++.

In the syntax graph, `<block1>` and `<block2>` contain zero, one, or a block of statements. A block of statements are enclosed within curly braces, which can contain zero, one, or multiple statements. Local declaration can be given in each block. The `<condition>` is any expression that evaluates to integer value. If the expression evaluates to 0 (considered "false"), `<block2>` will be executed, otherwise (any nonzero value will be considered "true"), `<block1>` will be executed. In the syntax graph, we omitted the arrow before the keyword **`if`**. In C, there is no Boolean type, while in C++ a Boolean type is predefined.

The following example illustrates the use of conditional structure and logic and relational operators.

```
if (a == b && c <= d)
  x = 0;
else {
  x = 1;
  y = 2;
```

```
}
```

The character sequence "&&" is a **logical operator** for AND. The character sequences "==" and "<=" are called **relational operators**. A complete set of both arithmetic and logical operators is given in Table 2.1.

C/C++ also provides a ternary **conditional operator** "?:" to form a **conditional expression**. The conditional operator takes three operands and performs a similar selection function as an if-then-else statement. The general form of the conditional operator is given in Figure 2.2.

In the syntax graph below, <operand1> and <operand2> can be any expression that returns a value or a simple assignment statement, in which case, the assigned value will be considered the return value of the operand. When the conditional expression is executed, the <condition> is first tested. If it returns a nonzero or true value, <operand1> will be evaluated; otherwise, <operand2> will be evaluated.

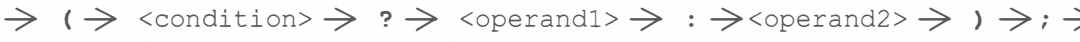


Figure 2.2. Syntax graph for the conditional operator in C/C++.

The following example illustrates different ways of using the conditional expression and their effects. A conditional expression can be used as an expression in the right-hand side of an assignment statement or as a stand-alone statement.

```
i = 0; j = 0;              // i = 0 and j = 0 represent false value
(i? i=5 : i=9);            // 9 will be assigned to i;
k = (j? i=5 : j=9);        // 9 will be assigned to j and to k
i = 1; j = 2               // i ≠ 0 and j ≠ 0 represent true value
(i? i=5 : i =9);           // 5 will be assigned to i;
k = (j? j=5 : j=9);        // 5 will be assigned to j and to k
```

2.2.3 Multiple selection structure (switch)

The basic selection statements select one out of two cases. If we have multiple choices, we need to use nested if-then-else statements. For example:

```
if (ch == '+')   x = a + b;
  else if (ch == '-')    x = a - b;
  else if (ch == '*')    x = a * b;
  else if (ch == '/')    x = a / b;
  else printf("invalid operator");
```

It is often more convenient in such situations to use a multiple selection statement switch, as defined in Figure 2.3.

The case statements label different actions we want to execute. The loop in the definition signifies that we can have any number of case statements (the number must be greater than or equal to one). The break statements, which exit the switch construct if a case is satisfied, are optional (there is a bypass route). The default case is performed if none of the other cases is satisfied. According to the definition, default is optional (there is a bypass route). If default is not included and none of the cases match, no action will be executed. For example, the following piece of code selects one of the four operations.

```
switch (ch) {
  case '+': x = a + b; break;
```

```
   case '-': x = a - b; break;
   case '*': x = a * b; break;
   case '/': x = a / b; break;
   default: printf("invalid operator");
}
```

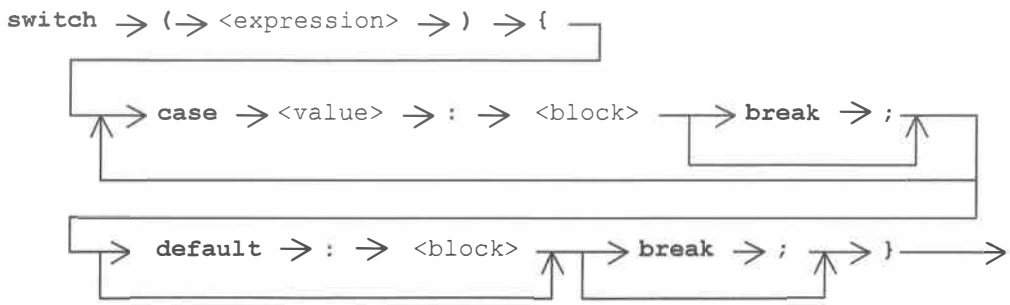


Figure 2.3. Syntax graph for `switch` in C/C++.

Including the `break` statements in the code is not an efficiency issue, as many people believe. What would happen if any one of the four `break` statements is omitted? Examine the following program without the `break` statements.

```
/* This C program demonstrates the switch statement without using breaks.
   The program is tested on MS Visual C++ platform */
#include <stdio.h>
void main() {
   char ch = '+';
   int f, a=10, b=20;
   printf("ch = %c\n", ch);
   switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
   }
   ch = '-';
   printf("ch = %c\n", ch);
   switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
   }
   ch = '*';
   printf("ch = %c\n", ch);
```

```
    switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
    }
    ch = '/';
    printf("ch = %c\n", ch);
    switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
    }
    ch = '%';
    printf("ch = %c\n", ch);
    switch (ch) {
        case '+': f = a + b; printf("f = %d\n", f);
        case '-': f = a - b; printf("f = %d\n", f);
        case '*': f = a * b; printf("f = %d\n", f);
        case '/': f = a / b; printf("f = %d\n", f);
        default: printf("invalid operator\n");
    }
}
```

The `switch` statements in this program are all syntactically correct, but they do not implement the selection at all. The omission of the `break` statements leads to the "fall through" execution of all the following cases of statements. The output of the program is

```
ch = +
f = 30
f = -10
f = 200
f = 0
invalid operator
ch = -
f = -10
f = 200
f = 0
invalid operator
ch = *
f = 200
f = 0
```

```
invalid operator
ch = /
f = 0
invalid operator
ch = %
invalid operator
```

This rather "unexpected" output is due to the "jump-table" implementation of `switch` statements at the assembly language level as shown in Figure 2.4.

The variable `ch` will be compared with the label values stored in the **jump-table**. If a match is found, the control will jump to the right address of the statement-table. Obviously, if no break statement appears at the end of each case, the machine would continue to execute the next statement. In some languages (e.g., Pascal), the compiler automatically adds a break statement at the end of each case. The advantage is the elimination of a possible error source and the drawback is that the programmer loses a bit of writability—in a rare case, a programmer may want to execute all the following cases once a condition is met.

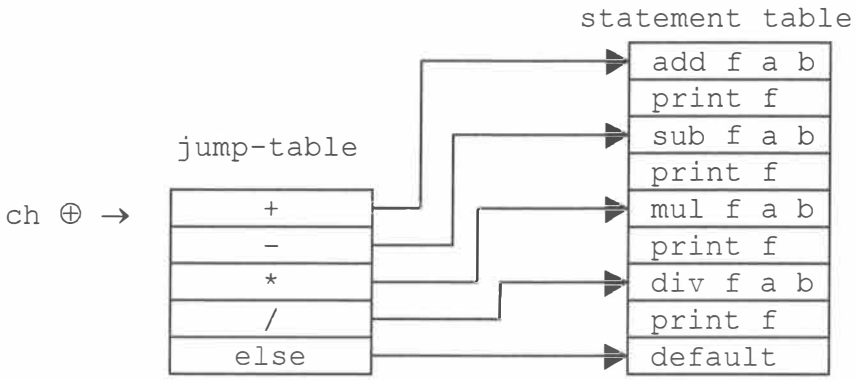


Figure 2.4. The assembly language level implementation of the `switch` statement.

2.2.4 Iteration structures (while, do-while, and for)

The basic looping structure in C/C++ is the **while-loop**. The syntax graph of a while-loop is given in Figure 2.5.

Figure 2.5 Syntax graph for `while` statement in C/C++.

In a while-loop, the `block` of statements, called **loop body**, will execute repeatedly as long as the `condition` statement produces a `true` (nonzero) value. However, there is no looping in the syntax definition. This is because there is only a semantic level looping for the `while`-statement, but no looping at the syntactic level. This is also true for the `for`-loop. On the other hand, there is a looping structure in the syntax graph of the `switch` statement, but there is no looping for the statement at the semantic level. The following program counts the number of inputs that are greater than 90. The program stops when a negative number is entered.

```
#include <stdio.h>
main () {
  int i, c = 0;
```

```
    scanf("%d", &i);
    while( i >= 0 ) {
        if (i > 90)
            c++;      // counting: same as c = c + 1;
        scanf("%d", &i);
    }
}
```

A variation of the `while-loop` is the **do-while-loop** that tests the condition after the loop body has been executed once. Using a `do-while-loop`, we need only one `scanf` statement for the example above:

```
#include <stdio.h>
main () {
  int i, c = 0;
  do { scanf("%d", &i);
       if (i > 90)
           c++;
  }
  while (i >= 0);
}
```

The `for`-**loop** can be considered a more compact form of the `while`-loop. It allows us to put the initialization, condition-test, and increment parts of a loop in a single statement. The syntax graph of the `for`-loop is given in Figure 2.6.

In the syntax graph, the `<initialization>` and `<increment>` can be single statements or multiple statements separated by commas. The function of the `for-loop` is equivalent to the function of the following code with a `while-loop`:

```
<initialization>
while(<condition>) {
  <block>
  <increment>
}
```

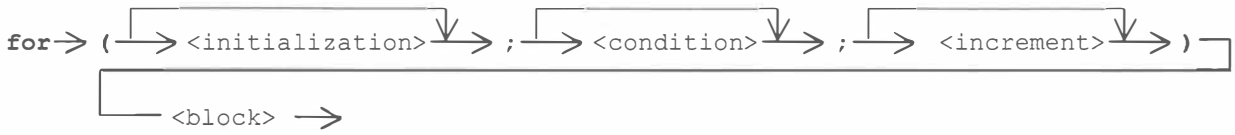


Figure 2.6. Syntax graph for the `for` statement in C/C++.

For example, the following program does a similar job as the program with a `while-loop`, except that the program with a `while-loop` terminates if a negative number is entered, while the following program terminates when exactly `n` numbers are entered.

```
main () {
  int i, k, n = 10, c=0;
  for(k=0; k<n; k++) {
      scanf("%d", &i);
      if (i > 90)
```

```
            c++;
        }
    }
```

All three components in the parentheses of a `for-loop` are optional. A `for-loop` with all three components absent creates an infinite loop:

```
for(;;) <block>
```

In some programming languages (e.g., Pascal), the loop iteration variable `k` and the loop boundary variable `n` may not be modified in the loop body, which means that the `for-loop` can only iterate a fixed number of times. In C/C++, both variables can be modified and thus the `for`-loop can iterate a variable number of times. However, it is not a good programming practice to modify any of the two variables even if we are allowed to modify them. Normally, we use a `while-loop` or a `do-while-loop` if the number of iterations is unknown and we use a `for-loop` if the number of iterations is fixed.

2.3 Data and basic data types in C/C++

The key concepts of data in a programming language include:

- Type: What values and operations are allowed on the type of data?
- Location: Where is data stored in memory?
- Address/Reference (of location in memory): How do we find the location where a particular piece of data is stored?
- Name: How do we conveniently access the locations of data?
- Value: What is stored in a memory location?
- Scope (visibility and lifetime): Where and when is a piece of data visible or accessible?

We will look at these concepts while studying basic data types in C/C++.

2.3.1 Declaration of variables and functions

At machine level, all data and instructions are stored in memory locations in sequences of binary bits: 001011. It is up to the programmer to manage and interpret the bit patterns.

A variable **declaration** in a high-level programming language binds a name to a location in memory and describes the attributes of the value in the location, so that the programmer can use the name to access the memory location and the value stored in the location conveniently. A variable declaration describes the following attributes of the value:

- type
- scope
- qualifier (modifiability, e.g., constant)
- variable initialization

Typically, the compiler allocates memory for the variable and binds the name to that location when a variable is declared.

The general form of variable declaration in C/C++ is

```
qualifier typename variable_names separated by a comma.
```

For example:

```
int i = 0, j, k;
const double pi = 3.1415926;
float x = 3.0, y, z = 2.5;
```

The general form of function declaration in C/C++ is

```
typename function_name(typename name, ..., typename name){ <body> }
```

The `typename` before the `function_name` specifies the return-value type of the function. The list in the parentheses is the list of parameters with their types. If a function does not return a value, we can either write the type name `void` or write nothing. Similarly, if a function does not have any parameter, we can either write void or nothing in the parentheses.

For example, the following program declares a `max()` function that returns the larger value between two parameter values. The function is called twice in the `main()` function.

```
#include <stdio.h>
int max(int first, int second){      // function declaration
  if (first > second)
      return first;
  else return second;
}
void main (void) {       // main function
  int i = 7, j = 5, k = 12, f;
  f = max(i, j);      // function call
  f = max(k, f);      // call the function again with different parameters
}
```

2.3.2 Scope rule

The **scope rule** of a C/C++ declaration: The scope of a variable is from its declaration to the end of the block defined by a pair of curly braces. The idea of the scope rule is declare-before-use: any variables or functions must be declared before they can be used. For example:

```
{
  int height = 6; int width = 6;
  int area = height*width;
  . . .
} // block ends
```

In this example, the variable `area` is initialized to `height*width`, which are declared just before the `area` is declared. According to the scope rule, the declaration is correct. On the other hand, if we swap the order of the first two lines

```
{
  int area = height*width;
  int height = 6; int width = 6;
  . . .
} // block ends
```

we will have a compilation error complaining that `height` and `width` are not declared when their values are used.

There is a subtle difference between the scope rules of an imperative language and a functional language. In a functional language, the scope rule normally says that "the scope of a variable is within the block in which the variable is declared/defined." Should the scope rule of C/C++ say that "the scope of a variable is within the block in which the variable is declared," no compilation error would occur if the declaration of variable `area` is placed before `height` and `width`.

The declare-before-use principle is simple to understand and use for variables, but may cause problems for declarations of mutually recursive functions. For example, a function `F` calls function `G` and function `G` calls function `F`. In this case, which function should be declared first?

There are two possible solutions to this dilemma:

Multi-scan compilation: The compiler scans the program multiple times. For example, in the first round of scan, all names (variables and functions) are stored in a name table, and in the second round of scan, binding between names and memory locations is made.

Forward declaration: Each function is declared in two steps: a forward declaration and a genuine declaration. The forward declaration makes a name known in advance (before it is used) and thus needs to specify only the return type, function name, parameter types, and parameter names (parameter names are optional). In the following program segment, for example, function `bar` calls function `foo` and function `foo` calls function `bar`. In such a case, we cannot satisfy the declare-before-use requirement without using forward declaration.

```
void bar(float, char);   // forward declaration to satisfy scope rule
int foo(void);           // forward declare all functions
...
int foo(void) {          // genuine declaration
  . . .
  bar(2.5, '+');         // call function bar()
  . . .
}
void bar(float f, char c) {  // genuine declaration
  . . .
  k = foo();             // call function foo()
  . . .
}
```

Most C/C++ compilers today use the multi-scan technique and thus forward declaration is not necessary for mutually recursive functions. However, forward declaration is still frequently used for two reasons:

- To make the program independent of the compiler
- Better readability: The forward declarations serve as an index to (overview of) all functions

2.3.3 Basic data types

C defines five basic **data types**, sometimes called value types. They are:

- Character (char)
- Integer (int)

- Floating-point (float)
- Double precision floating-point (double)
- Valueless (void)

C++ adds two more basic data types:

- Boolean (bool)
- Wide-character (wchar_t)

There is no Boolean type in C. The logic values are represented by integer: 0 for `false` and any other value will be interpreted as `true`. The character type in C is based on the 7-bit ASCII code, which allows 128 characters. C++'s wide-character type is based on the 16-bit Unicode, which allows $2^{16} = 65,536$ characters. Java's character type is also based on the Unicode.

Several of these basic types can be modified using one or more of these modifiers:

```
signed, unsigned, short, long, register
```

The type modifiers `signed` and `unsigned` explicitly specify that the integer type is signed and unsigned, respectively, although by default an integer type without any modifier is signed. For a signed integer, a "1" at the most significant bit indicates a negative number, while for an unsigned integer, no bit is used for the sign and only nonnegative numbers can be represented. Thus a "1" at the most significant bit indicates a large positive number. The type modifiers `short` and `long` indicate the data ranges of the integers of these types. It is more efficient to specify a short integer if you know your integer will not be very large. The type modifier `register` suggests to the compiler that the programmer wants to access the variable as fast as possible. Obviously, a variable can be accessed in the fastest way if it is put in a register. Since a processor has a very limited number of registers, you should use the `register` modifier sparingly. The `register` modifier is normally used for variables that need to be accessed frequently in a short period of time, such as loop variables. Please note that the `register` modifier is only a suggestion to the compiler. The compiler will take it into consideration where it is possible. However, there is no guarantee that the compiler can keep the variable in a register longer than the other variables.

Table 2.2 summarizes the basic data types available in C/C++. Since C/C++ can be implemented on machines of different sizes (e.g., word length = 8, 16, 32, and 64), the number of bits used to implement a particular data type can vary. However, the language requires that a minimum number of bits must be guaranteed for each data type. The larger machines can use more bits to provide extra data range and/or higher precision. The second and third columns of the table list the guaranteed minimum number of bits and the minimum data range for each of the data types.

To find the exact size of each type on a particular machine, you can call the `sizeof` function using the type name as the parameter `sizeof(type_name)`. For example:

```
printf("size of long type = %d\n", sizeof(long));
```

will print the "`size of long type = 4`" if the machine uses 4 bytes to store a `long` integer. If we call

```
printf("bool-size = %d, true = %d, false = %d\n", sizeof(bool), true,
false);
```

The output would be

```
bool-size = 1, true = 1, false = 0
```

which means C++ uses one byte to store a `bool` type variable, the internal value of `true` is 1 and the internal value of `false` is 0.

Type	Minimum bits	Minimum range
`bool`(C++ only)	1	true/false
char	8	from −127 to 127
signed char	8	from −127 to 127
unsigned char	8	from 0 to 255
`wchar_t`(C++ only)	16	from 0 to 65 535
int	16	from −32 768 to 32 768
signed int	16	same as `int`
unsigned int	16	from 0 to 65 535
short int	16	from −32 768 to 32 768
signed short int	16	same as `short int`
unsigned short int	16	same as unsigned int
long int	32	±2 147 483 647
signed long int	32	same as `long int`
unsigned long int	32	from 0 to 4 294 967 295
float	32	6 decimal digits of precision
double	64	10 decimal digits of precision

Table 2.2. Basic data types in C/C++.

To see the relationship among the types, we can classify the data types into four categories: scalar, function, aggregate, and valueless (`void`) types, as shown in Figure 2.7. The scalar types can be further divided into pointer, arithmetic, and enumeration types. In the diagram, boldfaced words are keywords, and italic words are optional keywords. Other names are generic terms. The basic data types we discuss in this section belong to arithmetic types. Functions are considered a special data type. In the following sections, we will discuss pointer, enumeration, and aggregated types.

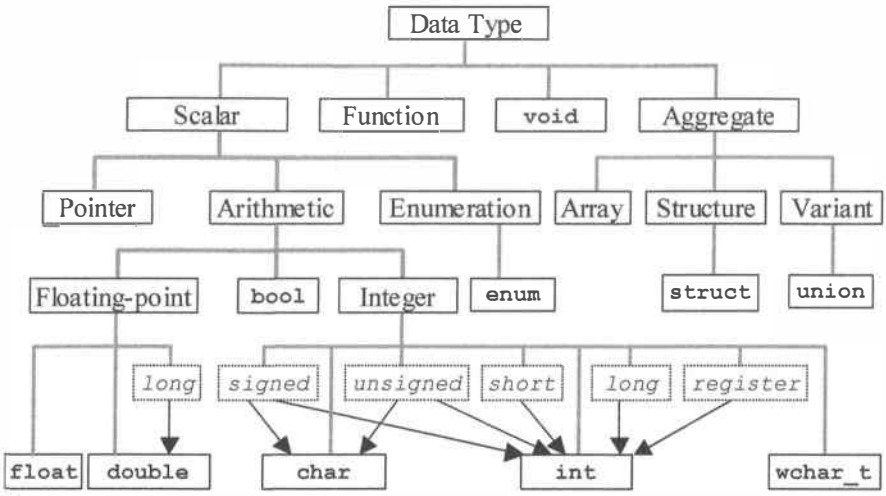


Figure 2.7. Classification of data types.

2.4 Complex types

In the previous section, we discussed basic data types. In this section, we discuss more complex data types including array, string as array of characters, pointer, constant, and enumeration.

2.4.1 Array

Array is a homogeneous collection of data elements that are stored in a consecutive block of memory locations. At the assembly language level, we use the initial address of the block plus the offset (index) of the element to access a particular element. At the high-level language level, we use the array variable name and the index to access an array element. An array is declared by

```
typename variablename[length] = {v0, v1, v2, ..., vlength-1};
```

The `length` and the initialization part, = {v0, v1, v2, ..., vlength-1}, are optional, which produces four possible combinations:

```
1. typename variablename[length];
2. typename variablename[] = {v0, v1, v2, ..., vlength-1};
3. typename variablename[];
4. typename variablename[length] = {v0, v1, v2, ..., vlength-1};
```

The first two array declarations are correct and are most frequently used. In the first declaration, the array variable and its length are declared. However, array elements are not initialized. In the second declaration, the length of the array is implied by the number of elements in the initialization list.

The third array declaration will immediately cause a compilation error because the compiler needs to know the size of the array to allocate the right amount of memory space for the array, and the size of the array is missing in the declaration.

The fourth declaration is syntactically correct, but one can easily make a contextual error in using this declaration! There are three possible cases when we use both explicit and implicit mechanisms to specify the length of the array:

- If length = n (the number of elements given in the initialization list), no problem will occur. However, this case is exactly the same as the second way of declaration.
- If length < n, a compilation (contextual) error will occur: there are not enough places to hold the elements given in the list.
- If length > n, no compilation error will occur. The n elements in the initialization list will be put in the first n places 0, 1, 2, ..., n−1. This is a case that is not covered by the first two ways of array declaration. Maybe this is the only case where we really need to use the fourth way of array declaration.

In the declaration of arrays, the `length` must be an integer value or a simple expression with integer operations like 20+5−1. It cannot contain a variable, even if the variable has been initialized.

The following piece of code illustrates the different ways of array declaration.

```
void main() {
  int a[3], sa, sb, sc, sd;  // a is correctly declared without
initialization
  int b[] = {2, 3, 9, 4};    // b is correctly declared and initialized
  int c[2] = {15, 14};       // c is correct, but the length is unnecessary
  int d[5] = {15, 14, 18};   // the first 3 elements of d are initialized
```

```
//int d1[2] = {15, 14, 18};  // incorrect: not enough places
//int e[];                   // incorrect: no length indication
  a[0] = 20;                 // array index always starts from 0
  a[1] = 30;
  a[2] = 90;
  sa = sizeof(a);            // number of bytes used by a is 12
  sb = sizeof(b);            // number of bytes used by b is 16
  sc = sizeof(c);            // number of bytes used by c is 8
  sd = sizeof(d);            // number of bytes used by d is 20
  printf("sa = %d\t sb = %d\t sc = %d\t sd = %d\n", sa, sb, sc, sd);
  printf("d0 = %d\t d1 = %d\t d2 = %d\t d3 = %d\t d4 = %d\n",
          d[0], d[1], d[2], d[3], d[4]);}
```

The output of the program is

```
sa = 12     sb = 16  sc = 8   sd = 20
d0 = 15     d1 = 14  d2 = 18  d3 = 0  d4 = 0
```

The first four lines of comments explain the different ways of declaration. The two incorrect declarations are commented out so that the program can be compiled and executed.

In the program, the system function `sizeof` returns the number of bytes (a byte = 8 bits) of the variable. The program is compiled on a 32-bit PC, an integer type variable takes 32 bits (4 bytes). If you compile the same program on a different machine (e.g., a 16-bit or 64-bit machine), the `sizeof` function will return different integer sizes. In Table 2.2, the minimum integer size given is 16 bits (2 bytes). When you write C/C++ programs, you need to handle the word length of the machine on which your program runs. You can use the `sizeof` function to find the word length of your computer and use the `sizeof` function to make your program independent of the word length. More uses of the `sizeof` function will be seen later in the text.

In Java, array declaration is different: We can declare an array without indicating its size and later give the size when we create the array object during the execution. This way of memory allocation is called **dynamic memory allocation**. The array declaration we discussed in this section is based on the **static memory allocation** by the compiler. However, C/C++ does provide the dynamic memory allocation mechanism for array and other structured data types. This will be discussed in conjunction with the pointer type.

We can define an array of `int`, `char`, and `float`, etc. Can we have an **array of arrays**? The answer is yes. C and C++ use array of arrays to represent **multidimensional arrays**. Array of arrays are declared and initialized like this:

```
char mac[5][7];
int mai[2][3] = {{4, 2, 3}, {7, 8, 9}};
```

Conceptually, array `mai` is stored in a matrix of 2 by 3, and its elements are accessed using the array name and the two indices `mai[i][j]`. Structurally, array `mai` is stored in a block of consecutive memory locations like this:

4	2	3	7	8	9

The following program illustrates the use of multidimensional arrays. Please note that `maxrow` and `maxcolumn` are defined as macros. The compiler would not accept the declaration of the `maze[maxrow][maxcolumn+1]` if they were defined as constant variables by using "`const`."

```
#define maxrow 50
#define maxcolumn 100
#include <stdio.h>
//const int maxrow = 100, maxcolumn = 100;
char ma[maxrow][maxcolumn+1];
void main(void) {
  int i, j;
  for (i=0; i < maxrow; i++)
      for (j = 0; j < maxcolumn + 1; j++)
          ma[i][j] = 'x';
}
```

2.4.2 Pointer

Pointer type is the most challenging data type in C/C++. This is especially true for Java programmers. Pointers provide programmers flexibility in accessing memory locations and modifying their values. On the other hand, the flexibility can easily create incorrect programs due to lack of understanding of computer organization and the relationships among different data types. This section will explain the principle of pointer type and the correct ways of using pointer variables.

We start by exploring different aspects of a **variable**:

- **Value:** A variable will hold a single value or a set of values. For example, an integer variable holds a single value and an array variable holds a set of values. A value can appear on the right-hand side of an assignment statement only and thus is called an **r-value** (for right-hand-side value).
- **Location:** A variable will be associated with a location or a set of memory locations. The value of a variable is stored in the location.
- **Address:** The address of a variable is a natural number directly associated with a memory location by the hardware. The address provides a direct way for programmers to access (read or write) a memory location or variable. The address refers to the literal number and thus is also an r-value.
- **Name:** The name of a variable is a mnemonic symbol that provides a convenient way for programmers to access a memory location or variable. A name is associated with a memory location (or translated into the address of the location) by the compiler. Some languages only use names to access memory locations (e.g., Java). Some languages allow using both names and addresses to access memory locations (e.g., C/C++). A variable name can appear in the left-hand side and right-hand side of an assignment statement and is called **l-value** (for left-hand-side value). A variable name has two faces: If it is used on the left-hand side of an assignment, it refers to the memory location. If it is used on the right-hand side of an assignment, it refers to the value stored in the memory location.

We can use an analogy to understand these aspects. Consider the soccer teams attending the World Cup. Each team consists of a number of members, corresponding to the set of values stored in a variable. Each team member will stay in a location (i.e., a hotel room). Each location will have a unique address (i.e., street address of the hotel plus room number). The team and each team member can be accessed by the address. In computer memory, the set of values related to a variable is normally stored in a consecutive block of memory locations, and thus, we can use the initial address of the block to access the values starting at that address. The hotel and rooms may also have names. If the context (scope) is clear, we can also use the name of the hotel and the name of a room to access a team member staying in the room. Read Appendix A, Section A.2 for a more detailed example.

Why do we need the name if we have the address of a memory location? Humans are better at reading, understanding, and remembering names than long tedious numbers.

Why do we need addresses in high-level language programming if we have names? The reasons are twofold. First, every memory location in a computer has an address, but not every memory location has a name. We can use addresses to access unnamed variables. Second, memory addresses are numbers and can be manipulated. For example, we can easily increment the address of the current location to obtain the address of the next location, or compare the two addresses to determine which address is smaller. As a result, it is more powerful and more flexible to access memory locations using addresses than using names.

What is a pointer? To take advantage of names (easy for humans to remember) and addresses (flexible in programming), we give a name to an address. The name of an address is a **pointer**. In other words, a pointer variable contains the address of another variable. Like any variable, a pointer variable is an l-value and the address stored in the pointer variable is an r-value.

Pointer as a data type is common in most imperative languages. The data range is the address space of the programming language. In C/C++, the data range is the same as an unsigned integer. The operations on pointers include the following:

- **Assignment operation:** An address value can be assigned to a pointer variable.
- **Integer operations:** A pointer variable can be operated like an integer variable.
- **Referencing operation:** Obtain the address of a variable `x` from the variable name: `&x`. The ampersand `&` is called the **address-of** operator that returns the address value of the variable it precedes. For example, if integer `x` is allocated at memory address = 2000, then `&x` will return 2000. Please note that `&x` returns the address value, not a pointer variable containing that address value, and thus `&x` is an r-value and can never appear on the left-hand side of an assignment statement.
- **Dereferencing operation:** To access the variable pointed to by a pointer variable `y`, we can use the **dereferencing** operator`*y`. In other words, the dereferencing operator `*` creates a new name for the variable pointed to by the pointer variable `y`. Please note that `*y` is a new name of the variable pointed to by the pointer variable that `*` proceeds. `*y` is an l-value and can appear on both sides of an assignment statement.

Although C/C++ has a pointer type, there is no type name for pointers. A pointer is declared by the type to which it points. For example:

```
int i = 137, *j;
j = &i;
```

Variable `i` is an integer and `j` is a pointer variable pointing to the integer variable `i` or `*j`, which becomes an alias (another name) of the variable. In other words, variable `i` has two names: `i` and `*j`. Assume that the compiler has associated the variable `i` with the address 100, then the statement "`j = &i;`" will assign 100 to `j`.

A pointer variable is a variable too. We can define another pointer variable to point to a pointer variable. For example, we can extend the above example to

```
int i = 137, *j = 0, **k = 1;                                    // 1
j = &i;                                                          // 2
k= &j;                                                           // 3
*j = 0;                                                          // 4
**k = 1;                                                         // 5
```

In the example, k is a pointer variable pointing to the pointer variable j. Assume the compiler has allocated address 100 to i, 160 to j, and 120 to k. Initially, the three variables are independent, as shown in Figure 2.8. Please note the initializations "*j = 0, **k = 1;" at line 1 put the value 0 in variable j and put the value 1 in variable k. This is different from the assignment statements at lines 4 and 5, where 0 and 1 are put into the variable *j and **k, respectively! Here, you can see again that static semantics (context) and dynamic semantics are different!

Figure 2.8. Variables i, j, and k are declared as independent variables at line 1 of the code.

After the execution of the statement at line 2, the address of variable i is put in j, resulting in j pointing to i (holding i's address); and after the execution of the statement at line 3, the address of variable j is put in k, resulting in k pointing to j (holding j's address). The new relationship between the three variables i, j, and k after statements at lines 2 and 3 is illustrated in Figure 2.9.

Variable i is initialized to value 137. Using the address-of operator, statement "j = &i;" puts the address of i, 100, into pointer variable j. Statement "k = &j;" puts the address of j, 160, into pointer variable k. On the other hand, we use the dereferencing operator to access the variable pointed to by the pointers. Since k holds the address of j, we can use *k to access j, or *k becomes an alias of j. Similarly, since j holds the address of i, we can use *j to access i, or *j becomes an alias of i. Furthermore, since *k is an alias of j, **k is an alias of i too, that is, i has two aliases: *j and **k. However, since &i is an r-value (not a variable), we cannot perform an &&i operation.

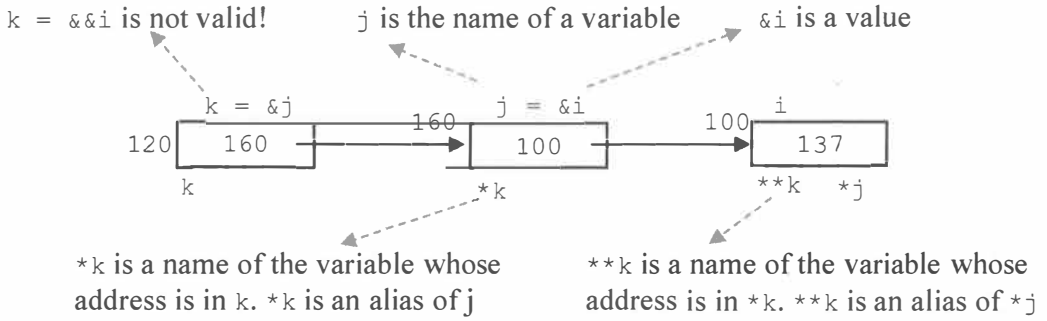


Figure 2.9. Relationship between variables and their pointers after lines 2 and 3.

At lines 4 and 5, both assignment statements modify variable i (*j and **k are aliases of i), resulting in the variable i being first changed from 137 to 0, and then changed from 0 to 1. If we compare the effect of these two statements with the initialization at line 1, we can see that similar assignment operations in the initialization part (contextual structure) and in the execution part (semantic structure) have different effects.

In this section, we discussed only the concept, the declaration, and the assignment of pointer variables. We will discuss more applications of pointers in the following sections. It will make a lot more sense when we combine pointer type with other complex data types.

2.4.3 String

There is no specific string type in C. Any array of characters can be considered a string, and thus a string variable can be declared as an array of characters, for example:

```
char str1[] = {'a', 'l', 'p', 'h', 'a'}; // initialized as an array
char str2[] = "alpha";                   // initialized as a string
char str3[5];                          // without initialization
```

As can be seen from the example, there are two different ways to **initialize** an array of characters in the declaration. The effect of these two initializations is slightly different.

The first declaration and initialization uses exactly the same method that declares and initializes any array, and thus `str1` is 100% an array of characters. On the other hand, we can also consider and use `str1` as a string. It has all the features that an array of characters should have. For example, we can modify the string in the following code and print the modified string

```
char str1[] = {'a', 'l', 'p', 'h', 'a'};
for (i = 0; i < sizeof(str1); i++) {
  str1[i] += 1; // same as str1[i] = str1[i]+1;
  printf("%c", str1[i]);
}
printf("\t sizeof(str1) = %d\n", sizeof(str1));
```

As expected, the output of the code is

```
bmqib  sizeof(str1) = 5
```

The second initialization indicates to the compiler that the array of characters is considered a string. In this case, the compiler will append a null terminator (null character) '`\0`' to the end of the string. In the ASCII table, the code for the null character is `0` (seven binary zeros). Please notice that the code for the digit '`0`' is 48. Appending the null character to the end of a string increases the size of the string by one, as shown in the following code.

```
char str2[] = "alpha";
for (i = 0; i < sizeof(str2); i++) {
  str2[i] += 1;
  printf("%c", str2[i]);
}
printf("\t sizeof(str2) = %d\n", sizeof(str2));
```

The output of the code is:

```
bmqib  sizeof(str2) = 6  // '\0'is not a printable character
```

To have the same effect, one can use the following initialization to append the '`\0`' to the end of the string:

```
char str4[] = {'a', 'l', 'p', 'h', 'a', '\0'};
```

As we discussed in the array section, we can specify the size in the declaration. If the size of the array is specified and the size is smaller than "string_length+1," the '`\0`' character and possibly some characters of the string cannot be stored in the variable. This is called **truncated initialization**. For example:

```
char str5[5] = "alpha";
char str6[4] = "alpha";
```

In this example, the null terminator '`\0`' will not be stored in `str5`; furthermore, the last "`a`" and the null terminator are not stored in `str6`. Figure 2.10 shows the memory map and initialization of `str1` through `str6`.

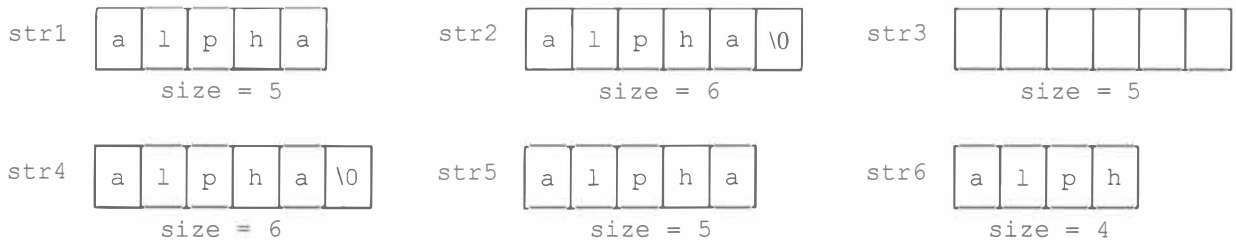


Figure 2.10. Memory allocation and initialization of six strings.

A number of string functions have been defined in the string package `<string.h>`and related library packages. A list of useful string and character manipulation functions is given in Table 2.3.

Having introduced the string functions, we can use the `strlen(str)` to replace the `sizeof(str)` in the previous example. In fact, `sizeof(str)` worked in that example because each character takes exactly one byte and `sizeof(str)` returns the number of bytes. The program would not work if we had used the `wchar_t` (two bytes per character) type instead. However, `strlen(str)` will work in both cases. Another difference, `sizeof` will include the byte used to store the null terminator '`\0`', while `strlen` does not.

Notice that some environment, such as Visual Studio, implemented a new set of string library functions with an extended name, such as `strcpy_s(str1, str2)` and `strcmp_s(str1, str2)`. These functions added security features to prevent code attacks at the instruction level, such as the return-oriented programming (ROP) attacks. An ROP attack combines code sequences in library functions to create malicious functions by changing the return address of function calls. This kind of attack does not need to inject malicious code into a system; instead, it modifies the return addresses on system stack.

So far, we have discussed the string as an array of characters. A string can also be defined by a pointer to a character or, more accurately, a pointer to the first character of a string. For example, the declaration

```
char *p = "hello, ", *q = "world", *s;
```

declares three pointer variables `p`, `q`, and `s`, each pointing to a character type variable. Pointer variables `p` and `q` are initialized to values that point to a string, while `s` is not initialized. Now the question is, what is the difference between the array-based strings and the pointer-based strings?

Library	Function	Description	Example
stdlib.h	atoi(str)	Convert a numeric string into an integer	`atoi("356")` returns 356 as an integer
	itoa(i, str, base)	Convert an integer `i` to a string using the specified `base` and link the result to pointer *str*	`itoa(356, str, 10)` results in pointer `str` pointing to string `"356"`.
stdio.h	getc(stdin)	Read a character from keyboard	ch = getc(stdin);
	gets(str)	Read a string from keyboard	gets(s); strcpy(str, s);
	putc(ch, stdout)	Print a character onto screen	ch = 'a'; putc(ch, stdout);
string.h	strcat(str1, str2)	Concatenate str2 to the end of str1	strcat (str, "hello world");
	strncat(str1, str2, n)	Concatenate the first n characters (substring) of str2 to the end of str1	strcat (str, "hello world", 3);
	strcmp(str1, str2)	Return 0 if str1 == str2, Return <0 if str1 < str2, Return >0 if str1 > str2,	if (strcmp(s1, s2)) y = x+1;
	strncmp(str1, str2, n)	Same as strcmp, except only compare the first n characters	if (strncmp(s1, s2, 3)) y = x+1;
	stricmp(str1, str2)	Same as strcmp, except letter comparisons are case insensitive	if (stricmp(s1, s2)) y = x+1;
	strcpy(str1, str2)	Copy str2 into str1	strcpy(str, "hello world");
	strlen(str)	Return the length of str	L = strlen(str);
ctype.h	tolower(ch)	Return the lowercase equivalent	tolower('D') returns 'd'
	toupper(ch)	Return the uppercase equivalent	toupper('b') returns 'B'

Table 2.3. Useful library functions for string and character manipulation.

We can examine the following example to see in detail the differences and similarities between array-based strings and pointer-based strings.

```
#include <stdio.h>
#include <string.h>
void main (void) {
  char p1[] = "hello", q1[] = "this is an array-string", s1[6];          //1
  char *p2 = "Hi", *q2 = "this is a pointer-string", *s2=0;              //2
  char *temp;                                                            //3
//s1 = p1;           // Array name cannot be a L-value                   //4
//s1 = "hi";         // Array name cannot be a L-value                   //5
  strcpy(s1, p1);    // We must use string-copy function                 //6
  printf("s1 = %s\t len-s1 = %d\n", s1, strlen(s1));                     //7
  strcpy(s1, q1);                                                        //8
  printf("s1 = %s\t len-s1 = %d\n", s1, strlen(s1));                     //9
```

```
  printf("s1 = %s\t size-s1 = %d\n", s1, sizeof(s1));                    //10
  for (temp = s1; temp < s1+strlen(s1); temp++)                           //11
      *temp += 1;                                                        //12
  printf("s1 = %s\n", s1);                                               //13
  for (temp = &s1[0]; temp <&s1[0] + strlen(s1); temp++)                 //14
      *temp -= 1;                                                        //15
  printf("s1 = %s\n", s1);                                               //16
//strcpy(s2, p2);                                                        //17
  s2 = q2;                                                               //18
  printf("s2 = %s\t len-s2 = %d\n", s2, strlen(s2));                     //19
  printf("s2 = %s\t size-s2 = %d\n", s2, sizeof(s2));                    //20
  for (temp = s2; temp < s2+strlen(s2); temp++){                         //21
//    *temp += 1;                                                        //22
  }
  strcpy(s1, q2);                                                        //23
  for (temp = s1; temp < s1+strlen(s1); temp++)                          //24
      *temp += 1;                                                        //25
  printf("s1 = %s\t len-s1 = %d\n", s1, strlen(s1));                     //26
}
```

All incorrect statements are commented out so that the program can be compiled and executed. The output of the program is

```
s1 = hello                          len-s1 = 5
s1 = this is an array-string        len-s1 = 23
s1 = this is an array-string        size-s1 = 6
s1 = uijt!jt!bo!bssbz.tusjoh
s1 = this is an array-string
s2 = this is a pointer-string       len-s2 = 24
s2 = this is a pointer-string       size-s2 = 4
s1 = uijt!jt!b!qpjoufs.tusjohlen-s1 = 24
```

Now we explain each statement in the program.

Statement 1 declares three array-based strings `p1`, `q1`, and `s1`. Variables `p1` and `q1` are initialized to a string while `s1` is not initialized.

Statement 2 declares three pointer variables `p2`, `q2`, and `s2`, each pointing to a character type variable. Variables `p2` and `q2` are initialized to values pointing to a string while `s2` is not initialized.

Statement 3 declares a pointer variable "`temp`," to be used as a temporary pointer variable.

Statement 4 tries to assign the string variable `p1` to string variable `s1`. A compilation error occurs, because `s1` is in fact an array name. We cannot assign anything to an array name. We can assign a value only to an element of an array (e.g., "`s1[0] = 'a';`" is a valid assignment).

Statement 5 tries to assign a string literal (value) to `s1`. For the same reason stated above, the statement causes a compilation error.

The correct way to assign a string variable or a string literal to an array-based string is to use the library function `strcpy(s1, p1)`. In statement 6, string `p1` is copied into string `s1` correctly and printed correctly in statement 7.

Statement 8 copies `q1` into `s1`. String `s1` and its length `strlen(s1)` are correctly printed in statement 9 (see output of the program). Please note that the length of `s1` is declared to be 6. How can the program put 24 characters in 6 places? This is in fact a semantic error that the compiler does not check. The runtime system could handle the error by checking the sizes and lengths of the two arrays in `strcpy` and prevent a longer array to be string-copied into a shorter array. However, these kinds of checks will slow down the execution of the program, and the designers of C decided to leave the responsibility to the programmers! As a programmer, you should know the lengths of the two arrays. If you really do not, you can always use the `strlen` function to find the lengths; for example, you can use the following statement to replace statement 6:

```
if (sizeof(s1) >= strlen(p1)) strcpy(s1, p1); else printf("error\n");
```

We still have not answered the question of how to put 24 characters in 6 places. What has happened is that the 18 extra characters are appended to the 6 declared memory locations, as shown in Figure 2.11.

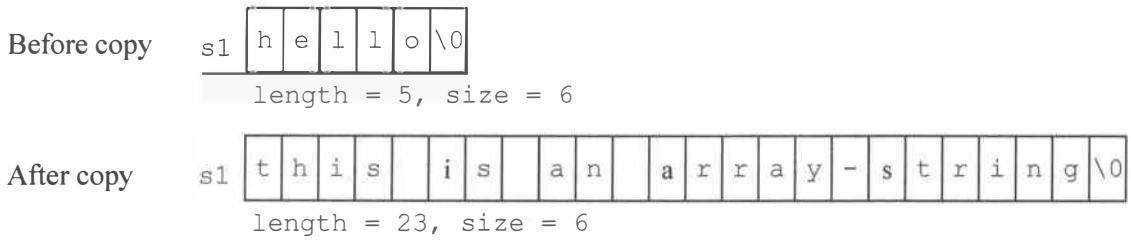


Figure 2.11. A `strcpy` operation may illegally use more space than what is declared.

Before we perform the `strcpy`, `s1` contains 5 characters and the size of `s1` is 6, as specified in the declaration. After we have performed the `strcpy`, a 23-character string is copied to the memory location starting from address `s1`. Obviously, the string goes beyond the limit of the size of `s1`. That is why we can still print the string `s1` with all characters, because the `printf` function starts from `s1` and stops when the character '`\0`' is detected.

The problem is that the use of memory beyond the declared boundary is unknown to the compiler and the runtime system. Please note that in the output of print-statement 10, the size of `s1` is still 6, even if a 23-character long string has been copied into `s1`. There are three possibilities:

(1) The locations are not allocated to any variable and you are lucky;

(2) The locations are allocated to other variables and you have overwritten the values of those variables;

(3) The locations are allocated to other variables and your values will be overwritten later.

In cases (2) and (3), your program may crash or, even worse, still behave normally but produce incorrect results that go undetected and cause much more damaging consequences. We explained in Chapter 1 that C/C++ use weak type checking. As you can see here, C/C++ are also weak in runtime checking, which leaves a huge responsibility entrusted to the programmers.

Now we continue to discuss the example. In statement 11, "`temp = s1;`" means to assign the address of `s1` (or the address of the first element `s1[0]`) to a pointer variable `temp`. Note that we use `&x` to obtain the address of a simple variable `x`, and we simply use the array name `s1` to obtain the address of an array `s1`

(or the address of the first element of the array). In some C compilers, you can use either `&s1` or `s1` to obtain the address of array `s1`. However, in C++, you can use the array name only to obtain the address of the array: Please also note that "`temp = s1;`" is same as "`temp = &s1[0];`" because `s1[0]` is a simple variable of character type, not an array (see the use in statement 14). In the next part of the for-loop, we use "`temp < s1+strlen(s1);`" to test if the value (address) of `temp` is less than the initial address of `s1` plus the length of `s1`. And then we increment `temp` in the next part.

In statement 12, we do "`*temp += 1;`", which means we increment the value pointed to by the pointer variable `temp`. The statement is the same as "`(*temp)++;`", but not the same as "`*(temp++);`", which increments the pointer value, instead of the pointed value. Please note that "`*temp++;`" is the same as "`*(temp++);`", because the unary operators * and ++ operate at the same precedence level. However, they associate from RIGHT to LEFT! Therefore, in "`*temp++;`", `temp` associates with ++ before *, and hence "`*(temp++);`" gets evaluated as "`*temp++;`".

Statement 13 will print the string in which every character is changed to the next character in the ASCII code. For example, `s` is changed to `t`, `h` is changed to `i`, `i` is changed to `j`, etc.

Statement 14 is equivalent to statement 11, and statement 12 is equivalent to statement 13 in structure, and it reverses (decrypts) the encryption in statement 12. Statement 16 prints the decrypted string that is same as the string before encryption.

We have discussed array-based strings so far, and now we turn to discuss pointer-based strings.

In statement 17, we try to do what we did in statement 6: string-copy `p2` to `s2`. However, the attempt will cause a compilation error. Thus, we commented the statement out so that we can continue with the other statements. The reason for this compilation error is that `s2` is a pointer variable and there is no memory allocated for a string. In the declaration in line 2, "`char ... *s2=0;`" means that `s2` is declared as a pointer variable to char and the pointer is initialized to `0`. It does not mean that the pointer is initialized to the address of the string "`0.`" However, should we use "`char ... *s2="0";`", it does mean that the pointer is initialized to the address of string "`0.`"

In statement 18, we assign `q2` to `s2`. We assign the value of `q2` (a pointer value) to pointer `s2`. Both pointers point to the same string. Here only pointer manipulation is involved. No string duplication is performed, as shown in Figure 2.12.

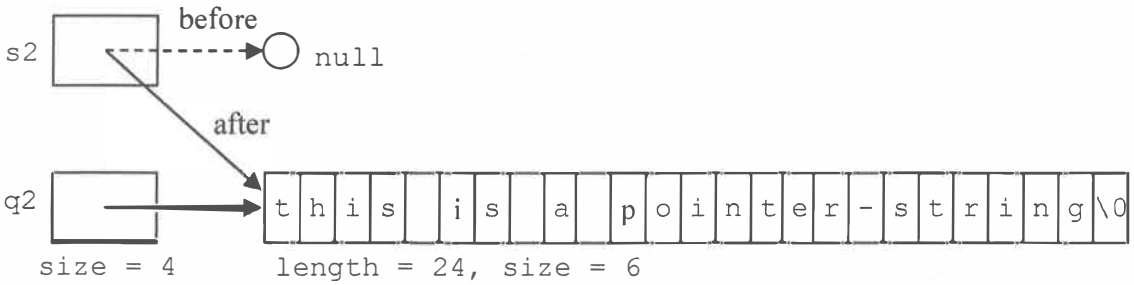


Figure 2.12. Both pointers `q2` and `s2` point to the same string.

Statement 19 prints the string pointed to by `s2` and its length. Statement 20 does not print the size of the string; instead, it prints the size of the pointer variable, which is 4 bytes (same as the size of an integer).

Statement 21 is similar to statements 11 and 14. Statement 22 tries to modify the character pointed to by temp, as we did in statements 12 and 15. However, we will have a runtime error. In C/C++, if a string is

assigned to a pointer-based string variable, the string is a string literal and cannot be modified. If we try to modify it, we will encounter a runtime error. Thus, we commented out this statement so that we can continue to compile other statements.

Although we cannot modify a string initialized as a pointer-based string, we can modify the string if it is copied into an array. Statement 23 copies pointer-based string q2 into s1 and then we can modify the string in s1 in statements 24 and 25. Statement 26 prints the modified string.

Through this example, we explained the following aspects of a string in C/C++.

- We can use the array of characters to declare a string variable and initialize the string variable to a string literal. We can access (read and write) the characters in the string as array elements. We can assign the initial address of the array (string) to a pointer variable and use this pointer to access (read and write) the characters in the string.
- We can declare a string variable using a pointer to character type and initialize the string to a string literal. We can read the characters in the string, but we cannot modify the characters.
- We can copy a pointer-based string into an array and we can modify the characters in the array.

A multidimensional array is stored in memory as a sequence of its elements or a one-dimensional array and can be processed easily using a pointer. We start with an example of 2-D array of characters.

```
#include <stdio.h>
void main(void) {
  char *p = 0, ma[2][4]; // declare a 2x4 array of characters
  ma[0][0] = 'C'; ma[0][1] = 'a'; ma[0][2] = 'r'; ma[0][3] = 'B';
  ma[1][0] = 'i'; ma[1][1] = 'k'; ma[1][2] = 'e'; ma[1][3] = '\0';
  p = &ma[0][0];
  while (*p != 0) {
      printf("%c", *p);
      *p = *p+1;
      p++;
  }
  printf("\n");
  p = &ma[0][0];
  while (*p != 0) printf("%c", *p++);
}
```

In the example above, a 2-D array is declared and initialized. We use pointer variable `p` to parse through each element, adding 1 to each element. The characters before the addition and after the addition are printed and the output of the program is shown in the screenshot.

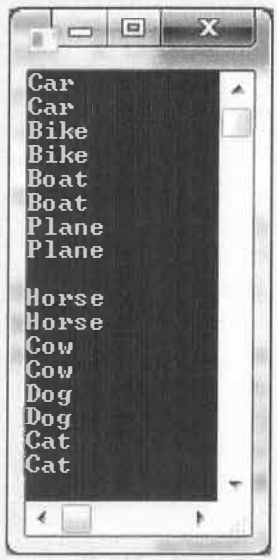
We can further define 3-D arrays. The code below defines a 2-D array of strings. As a string is an array of characters, the array is in fact a 3-D array of characters.

As we initialize the 2-D array using string values, the null terminator is appended to the end of each string. To print these strings, we use character print, and thus use the 3-D array element `ma[i][j][k]` to print each character. We use `ma[i][j][k]!='\0'` as the termination condition of the inner-most for-loop. Another option is to use the string length operation `strlen()` in the condition for the inner-most for-loop: `for (k=0; k < strlen(ma[i][j]); k++)`.

```
#include <stdio.h>
void main() {
   char *ma[2][4] = {{"Car", "Bike", "Boat", "Plane"},
                     {"Horse", "Cow", "Dog", "Cat"}};
   int i=0, j=0, k=0;
   for (i=0; i<2; i++) {
        for (j=0; j<4; j++) {
             for (k=0; ma[i][j][k]!='\0'; k++)
                  printf("%c", ma[i][j][k]);
             printf("\n");
        }
        printf("\n");
   }
}
```

We can also use pointer operations to access the 3-D array, in which we obtain the address of each string through the operation char `*p = ma[i][j]`, and then print each string, as shown in the following code.

```
#include <stdio.h>
void main() {
   char *ma[2][4] = {{"Car", "Bike", "Boat", "Plane"},
                     {"Horse", "Cow", "Dog", "Cat"}};
   int i = 0, j = 0, k = 0;
   char *p = 0;
   for (i = 0; i<2; i++) {
        for (j = 0; j<4; j++) {
             p = ma[i][j];          // Do not use &ma[i][j]
             printf("%s\n", p);     // print string
             while (*p!=0)
                  printf("%c", *p++); // print char
             printf("\n");
        }
        printf("\n");
   }
}
```



Each word is printed twice, because we used two different ways to print the words. First, we printed the entire word as a string, and then we printed each character of a string in a while loop. Notice that we use `p = ma[i][j];` instead of `p = &ma[i][j];` because the element of the 2-D array is an array of character, and the array name is the initial address of the array.

In C++, you can use all the C functions for string processing. However, a C++ library <string> is added to allow string declaration and processing without explicitly declaring an array of characters. The following code shows an example of using the string library.

```
#include <iostream>
#include <string>  // This is the library for C++ string operations
using namespace std;
```

```
void main() {
   string cat1 = "Max", cat2, temp; int  length;
   cout << "please enter a name for a cat" << endl;
   cin >> cat2;
   temp = cat1; cat1 = cat2; cat2 = temp; // swap the names of cat1 & cat2
   // One can also treat the string as an array of characters
   length = cat1.size();
   cout << "The length of cat1 is: " << length << endl;
   for (int i = 0; i < length; i++)
       cout << cat1[i];
   cout << endl;
}
```

Notice that we do not need to use strcpy; instead, we can simply use assignment cat1 = cat2 to copy the name in cat2 into cat1. Instead of using strlen(cat1) or sizeof cat1, we use cat1.size(), where cat1 is an object, and .size() is a member function of the object. We will discuss more details about class and object in the following chapter. The console output of the program is shown as follows:

```
please enter a name for a cat
Molly
The length of cat1 is: 5
Molly
```

2.4.4 Constants

Most programming languages allow constants to be declared. However, their implementations depend on the language definition and the compiler technologies.

C/C++ provides three different ways to introduce constants:

- **Macro**: As discussed in Section 1.4.2, we can use a macro definition to introduce a constant. The constant will substitute for the name at the preprocessing time. The advantage of a macro constant is its efficiency. A small constant may fit in an immediate-type of a machine instruction and thus save a memory access. The disadvantage is that the way a macro is defined is different from the way a variable is declared and initialized (nonorthogonal).
- **const qualifier**: A constant is a "variable" that a program cannot modify. The advantage is that the constant is declared and initialized in the same way as a variable is declared and initialized (orthogonal). However, a memory access is needed in order to access the constant (slower). We will discuss this kind of constant in this section.
- **Enumeration constant**: We can introduce constants by defining an enumeration type variable. This topic will be discussed in the following section.

The simplest way to introduce constants in C/C++ is to use the qualifier `const` before the variable declaration. For example:

```
const int min = 5, max = 100, pi = 3.14159265358979;
const char x = 'a', y = 's';
```

Constants increase the readability of programs. For example, the statement

```
if (x >= min and x <= max) x = x*x*pi;
```

is easier to understand than the statement

```
if (x >= 5 and x <= 100) x = x*x*3.14159265358979;
```

Constants also prevent us from making semantic errors. For example, if we try to modify a constant in a statement like

```
max = max + 10;
```

the compiler will raise a compilation error because `max` is a constant.

A constant defined by qualifier `const` is actually a "constant variable" and thus it has a memory address. We can apply the dereferencing operator on the constant. For example, the statement

```
temp = &max;
```

will put the memory address of constant `max` into the pointer variable `temp`.

In the following example, we demonstrate that we can even modify a constant variable if we can get around the compiler's check.

```
void main() {
  const int max = 100;
  int *temp;                  // temp is a pointer to an integer
//max = max + 10;             // Compilation error would occur
  temp = &max;                // assign the address of max to temp
  *temp = *temp + 10;         // max is modified through pointer temp
  printf("max = %d\n", max); // The output is: max = 110
}
```

Self-checking question: What would happen if we used "`#define max 100`" to define the constant? Will the statement "`temp = &max;`" work?

Through this example, we can see that a constant defined by `const` qualifier is in fact a variable.

It has a memory location and memory address and we can use the `&` operator to obtain its address.

Compiler protection is used. A compilation error will occur if you try to modify a `const` variable. In some versions of the compiler, a warning, instead of an error, will be given.

It can be modified if you can get around the compiler; for example, using an alias, you can modify a `const` variable.

2.4.5 Enumeration type

We have discussed data types predefined in C/C++. Most modern programming languages also provide mechanisms (type constructors) to allow programmers to define more complex data types. We will discuss the enumeration type in this section and other complex data types in the following sections.

Enumeration type is usually used for variables that can take an enumerable ordered set of values. Each of these values is given a name and we use the name to access the corresponding value. These names are associated with integer values starting from 0. Each enumeration type is a distinct data type.

Enumeration types in C/C++ are defined using the keyword **enum**. For example:

```
#include <stdio.h>
typedef enum {
      Sun, Mon, Tue, Wed, Thu, Fri, Sat
```

```
  } Days;
Days x = Sun, y = Sat;
void main (void) {
  while (x <= y) {
       printf("x = %d\t", x);
       x++;
  }
  printf("\n");
}
```

The names (constants) in the `Days` are not initialized and integers starting from 0 will be associated with each name in the given order. Thus, the type definition above defines seven constants equivalent to:

```
const int Sun = 0;
const int Mon = 1;
const int Tue = 2;
const int Wed = 3;
const int Thu = 4;
const int Fri = 5;
const int Sat = 6;
```

The output of the program is

```
x = 0  x = 1  x = 2  x = 3  x = 4  x = 5  x = 6
```

We can also initialize the names in the definition. For example, if we define `Days` as follows

```
typedef enum {
       Sun = 1, Mon = 2, Tue = 3, Wed = 4, Thu = 5, Fri = 6, Sat = 7
  } Days;
```

then the output of the program will be

```
x = 1  x = 2  x = 3  x = 4  x = 5  x = 6  x = 7
```

We now show a longer example demonstrating the use of enumeration types.

```
#include <stdio.h>
#include <time.h>
typedef enum {
       red, amber, green
  } traffic_light;
void sleep(int wait);    // forward declaration
main() {
traffic_light x = red;
printf("Red:\tStop!\n");
while (1)
  switch (x) {
       case amber:
            sleep(1);          //sleep 1 second
            x = red;
```

```
            printf("Red:\tStop!\n"); break;
        case red:
            sleep(6);          //sleep 6 second
            x = green;
            printf("Green:\tGo>>>\n"); break;
        case green:
            sleep(12);         //sleep 12 second
            x = amber; printf("Amber:\tBrake...\n");
    }
}
void sleep(int wait) {   // Sleep for a specified number of seconds.
    clock_t goal;        // clock_t defined in <time.h>
    goal = wait * CLOCKS_PER_SEC + clock();
    while( goal > clock() )
        ;
}
```

In this program, we defined an enumeration type called `traffic_light` with three possible values: `red`, `amber`, and `green`. We could use an `int` type instead. However, the program would be less readable and prone to error. A snapshot of the output is given as follows.

```
Red:    Stop!
Green:  Go>>>
Amber:  Brake...
Red:    Stop!
Green:  Go>>>
Amber:  Brake...
...
```

In this example, the time function `clock()` in `<time.h>` is used to obtain the number of clock cycles from a given point. This function can be used to measure the time between any two points. For example, the following piece of code can measure the time used by a function `foo()`.

```
c1 = clock();                          // time stamp 1
foo();
c2 = clock();                          // time stamp 2
interval = (double) (c2 - c1) / CLOCKS_PER_SEC; // time difference
```

The time difference computed in this example is in seconds. The precision of this method is 0.001 second.

There is another time function `time()` that can be used to measure the time in seconds. The following code shows the use of the time function and other related functions:

```
#include <stdio.h>
#include <time.h>
main() {
  int n; time_t start, finish;  double result, duration;
  time( &start );    // get the initial time
  for( n = 0; n < 900000000; n++ )
```

```
        result = 3.1415 * 2.23;
    time( &finish );  // get the end time
    duration = difftime( finish, start);  // compute difference
    printf( "\nThe program takes %2.4f seconds\n", duration );
}
```

2.5 Compound data types

In this section, we discuss compound data types that are composed of several data types, including structure, union, array of structures, linked list of structures connected by pointers, and file types.

2.5.1 Structure types and paddings

A structure is created using the keyword `struct`. The general way to define a structure type is

```
struct type_name {
        type field1;
        type field2;
        . . .
        type fieldn;
    } struct_variable_name;
```

For example:

```
struct stype {
    char ch;
    int x;
} u, v;          // We can declare variables of the type here.
void main() {
    struct stype s, t; // We can use the type to declare variables here too.
    ...                // The keyword struct must be used before type name.
}
```

Now we will study an example with a structure type.

```
struct Contact { // define a type that can hold a person's detail
    char name[30];
    long phone;
    char email[30];
};
void main() {
    struct Contact x, y, z;
    strcpy(x.name, "Mike Smith");
    x.phone = 9650022;
    strcpy(x.email, "mike.smith@asu.edu");
    strcpy(y.name, "Jane Miller");
    y.phone = 9650055;
    strcpy(y.email, "jane.miller@asu.edu");
}
```

As you can see from the example, we use `x.name` notation to access the `name` field of the variable `x`. We will see more examples of structures in the following sections, where we will combine the structure types with the array and pointer types.

The processor reads integers and floats in words. If a structure contains an integer, a pointer, or a float, the structure size must be aligned into multiples of 4 in a 32-bit computer and multiples of 8 in a 64-bit computer. Consider a 32-bit computer. If a structure has a member of integer, pointer, or float, and a member whole size is not a multiple of 4, padding is needed to align the data in memory. It may need padding of one, two, or three bytes to pad one part of a structure. Consider the following snippet of code:

```
struct personnel {
   char name[16];
   int phone;
   char address[24];
   char gender;        // F or M
   // char CSmajor;    // Y or N
} person;
printf("struct size = %d", sizeof person);
```

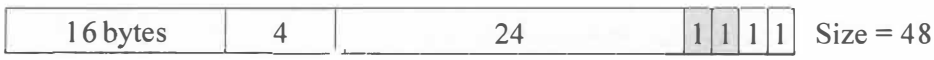
If you count the bytes, the `struct` variable person will need 45 bytes. However, the `printf` will print the size of person = 48.

The reason is as follows. The structure contains an integer variable phone. An integer will be read at machine language level by a "Load Word" instruction, and thus the entire structure will be read using the Load Word instructions. As shown in the following memory map, the compiler will add three bytes before the character type variable to make the byte into a four-byte word. As the result, the total size of the person variable is 48.

In the code, uncommenting the two lines of code adds one more character variable into the structure, as shown below:

```
struct personnel {
   char name[16];
   int phone;
   char address[24];
   char gender;     // F or M
   char CSmajor;    // Y or N
} person;
printf("struct size = %d", sizeof person);
```

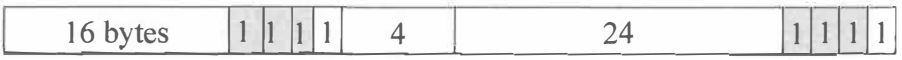
The total size of the structure will remain unchanged as 48. In this case, the three bytes of padding will reduce to two bytes only, as shown in the following map:



The order of the variables in a structure will be preserved by the compiler when it allocates the memory. If the character types are separated by a Word-type variable, multiple paddings are required. Consider the following snippet of code, where the variable gender is moved before the variable phone:

```
struct personnel {
   char name[16];
   char gender;     // F or M
   int phone;
   char address[24];
   char CSmajor;  // Y or N
} person;
printf("struct size = %d", sizeof person);
```

The memory map is given as follows:



Two paddings of three bytes each are added by the compiler, resulting in a structure of 52 bytes. These examples show that it will result in a more efficient code if the character type variables are kept together in the structure definition.

Padding is required only for word type of variables, such as int, float, and pointer. If a structure contains character type of variables only, no padding will be added, irrespective of whether the total size is a multiple of four or not. The compiler will use "Load Byte" instructions to read the structures that contain character (byte) type only. Consider the following code, where the int variable phone is removed:

```
struct personnel {
   char name[16];
   char gender;     // F or M
   char address[24];
   char CSmajor;  // Y or N
   char KnowJava;    // Y or N
} person;
printf("struct size = %d", sizeof person);
```

The program will print the size of person = 43, which is not a multiple of 4.

Consider the following snippet of code in a 32-bit computer:

```
struct contact {
    char    name[30];
    int     phone;
    char    email[30];
} x;
```

What is the size of variable x in bytes? It is incorrect to simply add 30+4+30. Because there is an integer type involved, two padding bytes must be added to name and email arrays, and thus, the total number of bytes for variable x is 68, instead of 64. However, if we keep the name and email members together in the order, the size will become 64.

2.5.2 Union

A `union` type variable is a region of shared memory that, over time, can contain different types of values. At any given moment, a union can contain only one value. Programmers must make sure the proper type is used at the proper time. The general way to define a union type is

```
union union_name {
```

```
    type field1;
    type field2;
    . . .
    type fieldn;
} union_variable_name;
```

For example:

```
union utype {
    char ch;
    int  x;
} v;
void main(){
    union utype s, t;  //We can use the type to declare variables here too.
    ...                // The keyword union must be used before type name.
}
```

In this example, we define a union type called `utype` and declare a variable `v` of `utype`. Similar to a structure type, we can have multiple data fields in the type definition. In this example, there are two data fields. The field variable `x` belongs to the `int` type and takes 32 bits or 4 bytes (in a 32-bit machine) and `ch` takes 8 bits or 1 byte. If the union type is defined as a structure type, the variable `v` will have 4+1 bytes of memory allocated. However, in the union type, all data fields share the same memory. If these data fields require different sizes of memory, the largest size among the data fields will be allocated and the smaller sized fields will occupy a part of the memory. In this example, 4 bytes of memory will be allocated and the smaller field `ch` will share the first byte of `x`.

The way we access a union type variable is similar to that of the structure type variable. For example,

```
v.x = 124000;    // put an integer value into the data field x
v.ch = 'C'; // put a character value into the data field ch.
```

Since the two data fields share a part of the memory, the second assignment will overwrite the first byte of `v.x`, destroying the integer value in `v.x`. Obviously, if we do not use the data fields carefully, we can easily make mistakes in programming.

The question is why do we need such an unsafe data structure? The reason is that it could be useful in certain situations. The following example depicts such a situation where union type variables make the program more elegant.

Assume we want to define a data type to store personnel information for both faculty members and students in a university. The faculty and students have ID numbers with different lengths. A person in a university has either a faculty ID or a student ID. If we use two separate data fields for faculty ID and student ID, we will use only one of the two data fields for every record. If we use only one data field and leave the extra bytes free when an ID number does not have enough characters to fill all bytes, we could lose our view of whether we are dealing with a student record or a faculty record. A union type would solve the problem, as shown in the following program:

```
#include <stdio.h>
#include <string.h>
struct Personnel {    // Define a structure type called Personnel
  char name[30];
  long phone;
```

```
    union identity {   // Define a union type inside the structure type
        char facultyid[8]; // Two alternative data fields are defined here
        char studentid[12];
    } id;     // We declare a variable of the union type here.
};

main(){
    struct Personnel x, *p;  // Declare a Personnel type variable and a
pointer
    strcpy(x.name, "Mike Lee");     // Copy a name into the name field
    x.phone = 21400000;             // Assign a number to phone field
    strcpy(x.id.studentid, "1999eas1234"); // Copy student ID
    printf("x.id.studentid = %s\n", x.id.studentid);
    strcpy(x.name, "Jane Smid");    // Use the same x for a faculty record
    x.phone = 9659876;
    strcpy(x.id.facultyid, "cse1234");
    printf("x.id.facultyid = %s\n", x.id.facultyid);
    p = &x;
    printf("p->id.studentid = %s\n", p->id.studentid);
    printf("p->id.facultyid = %s\n", p->id.facultyid);
}
```

In this example, the same variable is used for a student record and a faculty record. The different ID field names allow us to differentiate which record we are handling. In the next chapter, we will discuss the generic class in C++, which is a more general way of associating different classes with a class reference.

2.5.3 Array of structures using static memory allocation

Structure types will make more sense if we combine them with array and pointer types to form collections of structures. In the following example, we define an array of structures to form a database.

In the following program, `Contact` is a structure type with three data fields. The declaration

```
struct Contact ContactBook[max];
```

declares an array of structures with 100 entries. Then we use the `tail` variable as the index to access the next unused element of the array: `ContactBook[tail]`. Figure 2.13 shows the structure of the array.

Since the element of the array is of the `structcontact` type with three data fields, we use the **dot-notation** to access the data fields of the i^{th} element:

```
ContactBook[i].name
ContactBook[i].phone
ContactBook[i].email
```

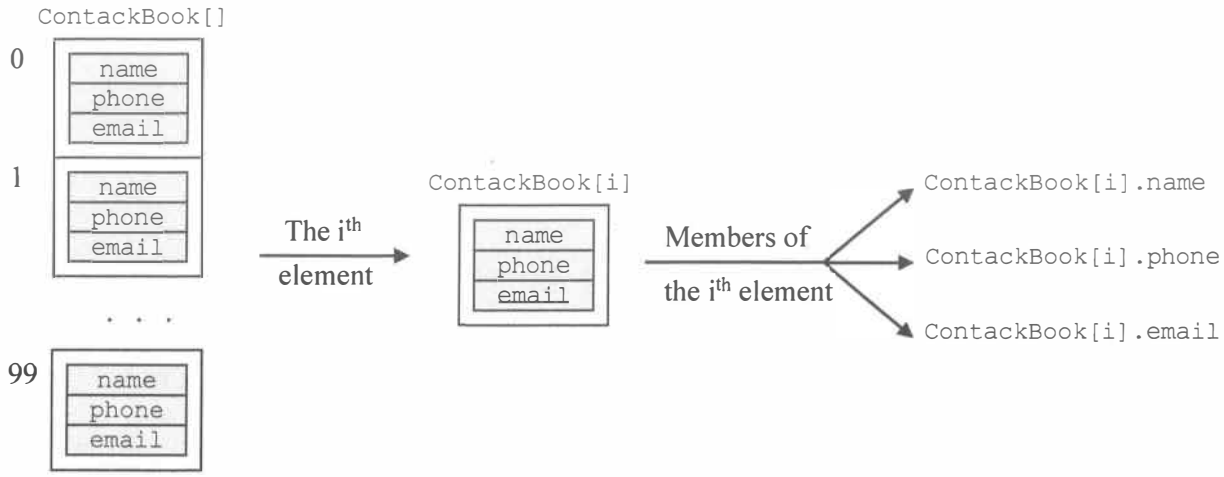


Figure 2.13. Array of structures, its element, and members of the element.

The array variable `ContactBook[max]` is a **global variable**, that is, a variable that is outside all functions. The memory locations for global variables are statically allocated by the compiler during compilation time. We call this kind of memory allocation **static memory allocation**. In this example, the compiler will allocate an array of 100 (`max`) elements before the program starts. Assume a `long` integer takes 4 bytes, and the name and email take 30 bytes each. The total number of bytes needed for one array element is then 64 bytes. The array of 100 elements will take 6400 bytes.

```
/* This program demonstrates how to define an array of structures.
   It statically allocates memory for the variables of structure type */
#include <stdio.h>
#include <string.h>
#define max 100
struct Contact { // define a node that can hold a person's detail
   char name[30];
   long phone;
   char email[30];
};
struct Contact ContactBook[max];  // an array of structures, 100 entries
int tail = 0;                     // tail is defined here as a global variable
void branching(char c);  // forward declaration of a function
int insertion();         // forward declaration of a function
int search();            // forward declaration of a function
// void deletion();      // not implemented in this example
// void printall();      // not implemented in this example
void main() {            // main() first prints a menu for selection
   char ch = 'a';
   while (ch != 'q') {
       printf("enter your selection\n");
       printf("      i: insert a new entry\n");
       printf("      s: search an entry\n");
       printf("      d: delete an entry\n");    // not implemented
```

```
        printf("      p: print all entries\n"); // not implemented
        printf("      q: quit\n");
        fflush(stdin);                  // flush input buffer to make
        ch = getc(stdin);               // sure getc reads correctly
        branching(ch);
    }
}
void branching(char c) { // branch to different tasks
    switch(c) {
        case 'i': insertion(); break;
        case 's': search(); break;
        case 'q': printf("You exit the program\n"); break;
        default:  printf("Invalid input\n");
    }
}
int insertion() {    // insert a new entry
    if (tail == max) {
        printf("There is no more place to insert\n");
        return -1;
    }
    else {
        printf("Enter name, phone, email\n");
        scanf("%s", ContactBook[tail].name);
        scanf("%d", &ContactBook[tail].phone);
        scanf("%s", ContactBook[tail].email);
        tail++;
        printf("The number of entries = %d\n", tail);
        return 0;
    }
}
int search() {   // search and print phone and email via name
    char sname[30];
    int  i;
    printf("please enter the name to be searched\n");
    scanf("%s", sname);
    for (i=0; i<tail; i++)
        if (stricmp(sname, ContactBook[i].name)== 0) {
            printf("phone = %d\n", ContactBook[i].phone);
            printf("email = %s\n", ContactBook[i].email);
            return 0;
        }
    printf("The name does not exist\n");
    return -1;
}
```

In the next chapter, we will discuss in detail the three different memory areas: static, stack, and heap, the mechanisms for allocating memory from these three areas, as well as how memory is deallocated (garbage collection).

2.5.4 Linked list using dynamic memory allocation

The advantage with static memory allocation is that the memory for the variables is already available when we want to store data in them. The problem is that we need to know the maximum number of elements in advance, which is possible in some cases and not possible in some other cases. If we overestimate the data amount, we waste memory. If we underestimate the data amount, we have to stop the program, modify the `max` value, and recompile the program. To solve this problem, we can use **dynamic memory allocation** that allocates memory to variables during the execution by a function call.

In C, the function that dynamically allocates memory is

```
void *malloc(size_t size);
```

The function takes one parameter that is of type `size_t`. The type `size_t` is usually an `unsigned int`. The parameter specifies the number of bytes to be allocated. For example, if you need a memory location for an integer variable, then you can call

```
p = malloc(4);
```

However, this statement will work only on a machine that uses 4 bytes for an integer. If you run your program on another machine with a different word-length, the statement will cause a problem. A better way to allocate memory for a given type of variable is to call `malloc (sizeof(type_name))`. For example, if you need memory for an integer variable, it is better to do

```
p = malloc(sizeof(int));
```

The function `malloc` returns a pointer to the initial address of the memory. If the runtime system runs out of memory, it returns `null`.

Please notice that the notation "`void *`" means here that the `malloc` function returns a generic pointer that can point to a variable of any data type. This is possible because all pointer types are structurally equivalent and C mainly uses structural type equivalence in its type checking. Of course, you can also make an explicit type casting to convert the generic pointer type to the specific type, for the purpose of readability, for example:

```
p = (int *) malloc(sizeof(int));
```

casts the return value to an integer type pointer.

In C++, a new dynamic memory allocation operator has been introduced:

```
class_name p = new class_name;
```

The `new` operator allocates the right amount of memory for a variable (object) of the given class and returns a pointer of that class. Java uses a similar operator to dynamically allocate memory. The `new` operator will be explained in more detail in the next chapter.

Since the `malloc` function returns a generic pointer, we often combine dynamic memory allocation with pointers to define a collection of structures. The following example re-implements the array of structures using dynamic memory allocation.

```
/* This program demonstrates how to define a linked list of structures.
   It dynamically allocates memory for the variables of structure type.
   Only the parts that are different from the array of structure example
```

```
   are given here. */
#include <stdio.h>
#include <stdlib.h>      // used for malloc
struct Contact {         // define a node holding a person's detail
   char name[30];
   long phone;
   char email[30];
   struct Contact *next; // pointer to Contact structure
} *head = NULL;          //head is a global pointer to first entry
void branching(char c); // function forward declaration
int insertion();
int search();
// void deletion();
// void printall();

int insertion() {        // insert a new entry at the beginning
   struct Contact *p;
   p = (struct Contact *) malloc(sizeof(struct Contact));
   if (p == 0) {
       printf("out of memory\n");  return -1;
   }
   printf("Enter name, phone, email \n");
   scanf("%s", p->name);
   scanf("%d", &p->phone);
   scanf("%s", p->email);
   p->next = head;
   head = p;
   return 0;
}
int search() {   // print phone and email via name
   char sname[30];
   struct Contact *p = head;
   printf("please enter the name to be searched\n");
   scanf("%s", sname);
   while (p != 0)
       if (strcmp(sname, p->name)== 0) {
            printf("phone = %d\n", p->phone);
            printf("email = %s\n", p->email);
            return 0;
       }
       else p = p->next;
   printf("The name does not exist\n");
   return -1;
}
```

In the example, the `Contact` type is redefined with an extra field `next`:

```
struct Contact *next;  // pointer to Contact structure
```

The `next` field is a pointer to a `Contact` type variable. We use it to form a linked list. Please note that we need to use the keyword **`struct`** whenever we refer to a structure type.

In the insertion function, we use

```
p = (struct Contact *) malloc(sizeof(struct Contact));
```

to allocate the right amount of memory for a variable of `Contact` type, and we link the initial address of this memory chunk to a pointer variable `p`. The type casting makes it clearer that the memory is allocated for a `Contact` type variable. Using malloc(sizeof(struct Contact)), instead of using malloc(68), can avoid calculation error, and particularly, avoid calculating the padding bytes.

Figure 2.14 illustrates the insertion process. Assume that the linked list already has two nodes and a new node is being inserted.

This insertion function inserts the new node at the beginning of the linked list. You can also insert the new node at the end (or at any required position). In this case, you can use a temporary pointer, say `temp`, and move temp to the last node before performing insertion, as shown in the following code.

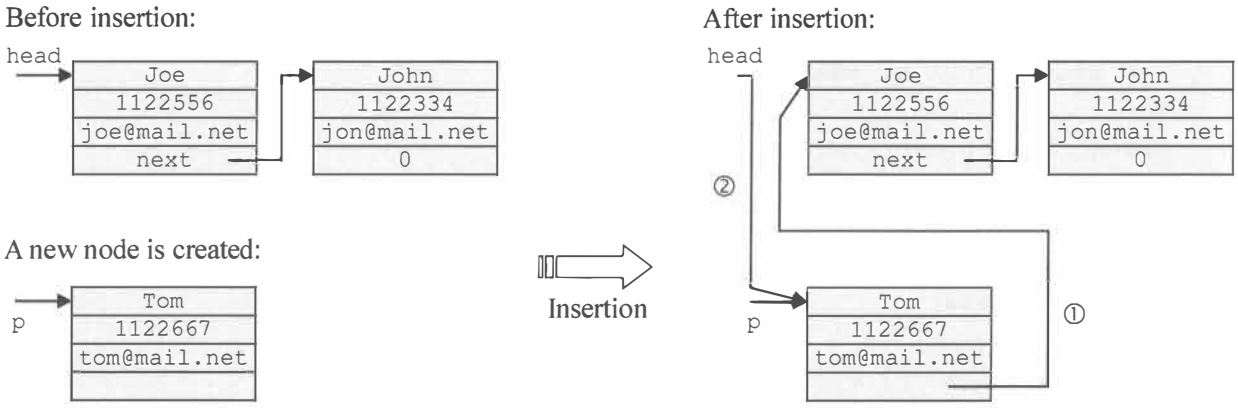


Figure 2.14. Insert a new node at the beginning of a linked list.

```
int insertion_at_end() {      // insert a new entry at the end
  struct Contact *p, *temp;
  p = (struct Contact *) malloc(sizeof(struct Contact));
  if (p == 0) {
      printf("out of memory\n");  return -1;
  }
  printf("Enter name, phone, email \n");
  scanf("%s", p->name); scanf("%d", &p->phone); scanf("%s", p->email);
  p->next = 0;
  if (head == 0) head = p;
  else {
      while (temp->next != null)
          temp = temp->next;        // Find the last node
      temp->next = p;               // Link the new node
```

```
    }
}
```

Generally, a node can be inserted in any position in a linked list. Figure 2.15 illustrates the insertion process. It consists of three steps: (1) Find the position where the new node is to be inserted. Use a temporary pointer variable `temp` to point to this position. (2) Set the new node's next pointer to the node next to the node pointed to by `temp`. (3) Set the next pointer of the node pointed to by `temp` to the new node.

In the earlier example of the array of structures, we used the dot-notation to access the data field of a structure variable. It is different when referring to a data field of a structure pointed to by a pointer variable. We use the **arrow operator** (sometimes called pointer-to-member operator) instead; that is, we use

```
p->name
p->phone
p->email
p->next
```

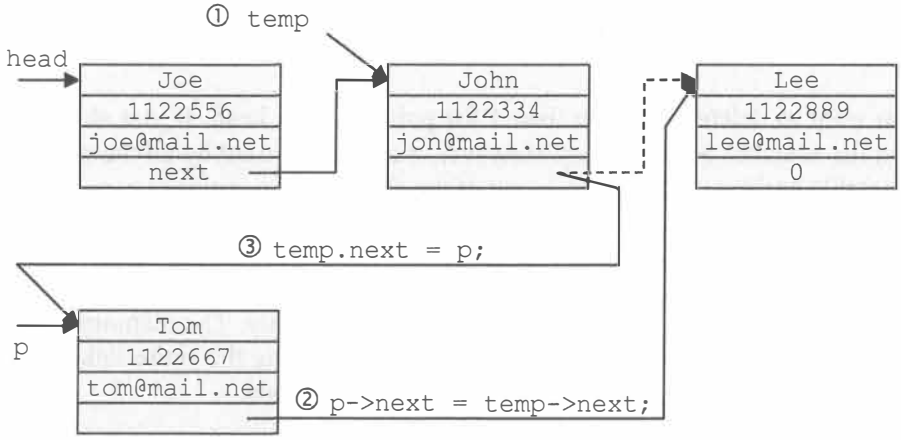


Figure 2.15. Insert a new node in the middle of a linked list.

to access the four fields of a `Contact` structure variable pointed to by `p`. The differentiation is necessary because their meanings are different. Examine the piece of code:

```
struct Contact *p, q;
p = (struct Contact *) malloc(sizeof(struct Contact));
strcpy(p->name, "smith");     // or strcpy(*p.name, "smith");
strcpy(q.name, "miller");
free(p);     // return the memory allocated by malloc to the memory heap
p = &q;      // p is now pointing to variable q.
```

In the example, `p` is a pointer to a `Contact` structure variable and `p` has only 4 bytes of memory allocated, while `q` is the name of a variable of `Contact` type, as shown in Figure 2.16. The compiler has allocated the entire memory that can hold all four data fields to `q`. Thus, we can directly copy a name "`miller`" into the name field of `q`. Before we can copy anything into the variable pointed to by `p`, we must first use `malloc` to obtain the memory for that variable.

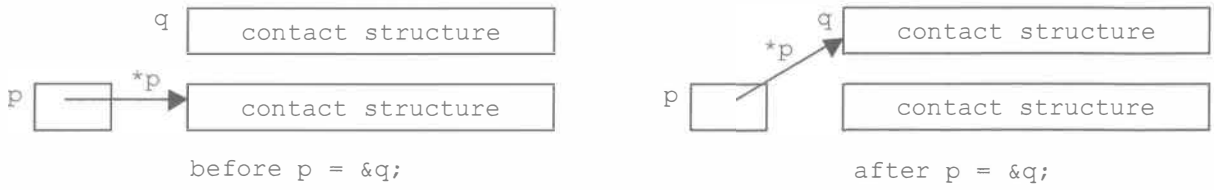


Figure 2.16. Variable `q` is allocated statically, while the variable pointed to by `p` is dynamically allocated.

The last statement in the example assigns the address of `q` to `p`. Now `p` is pointing to the variable `q`. In other words, now `q` has another name which is `*p`. If we do not free (delete) the `Contact` variable pointed to by `p` before we assign the address of `q` to `p`, the variable will be completely inaccessible and becomes a piece of **garbage**. The "`free`" function is, in fact, doing the job of garbage collection.

The function `free(p)` is the opposite of the function p = `malloc(size)`, in that it returns the memory linked to `p` to the **heap**, the pool of free memory. If we keep using `malloc` to get memory from the heap, but do not collect the garbage, the heap will eventually be empty and we thus run out of memory. Not collecting garbage is also called a **memory leak**. The following examples show memory leak when deleting a linked list.

Assume that you want to delete the entire linked list pointed to by head. If you simply assign `head = null`, the linked list becomes an empty list. However, the memory used by all the nodes in the linked list becomes uncollectable garbage. What is the result of the following operations?

```
free(head);
head = null;
```

This snippet of code will free the memory used by the first node only. The memory used by all the other nodes will become uncollectable garbage. The correct way of deleting the entire linked list is to use a loop to free each and every node, as shown in the following snippet of code.

```
temp = head;
while (temp ! =null) {
   temp = temp->next;
   free(head);
   head = temp;
}
```

Garbage collection and memory leak will be discussed in the section on memory management in the next chapter.

2.5.5 Doubly linked list

When traversing a linked list, you can easily move a temp pointer forward from the head pointer to the end of the linked list. However, you cannot move the pointer backward along the linked list. If you need a data structure that can move both forward and backward, doubly linked list is a good solution.

A doubly linked list node has two pointers pointing to the previous node and the next node. The following code shows a simple example of a doubly linked list. An insertion function is given, which inserts a new node at the sorted place by name. The id of the Node `struct` is generated automatically, and the names are entered from the keyboard. This program can be combined with the complete program of the singly linked list to allow full functionalities.

```
#include<stdio.h>
```

```
#include <stdlib.h>
#include<string.h>
#pragma warning(disable: 4996) // disable warning in Visual Studio
struct Node {
  int id;                    // int size is 4 bytes
  char *name;                // name is a pointer, not an array
  struct Node* previous;     // pointer to previous node
  struct Node* next;         // pointer to next node in list
};
struct Node *head = NULL, *tail = NULL;
int insertion(int i, char* n) {
  struct Node *temp = (struct Node*)malloc(sizeof(struct Node));
  if (temp == NULL) {
      printf("out of memory\n"); return -1;
  }
  temp->id = i;
  temp->name = n;
  if (head == NULL) {  // Case 0: the linked list is empty
      head = temp;
      head->next = NULL;
      head->previous = NULL;
      tail = temp;
      return 0;
  }
  else {  // Case 1: The list is not empty, insert at the beginning
      if (strcmp(temp->name, head->name) < 0) {
          temp->next = head;
          head->previous = temp;
          head = temp;
          head->previous = NULL;
          return 1;
      }
  };
  struct Node *iterator = head;
  struct Node *follower = iterator;
  while (iterator != NULL) {  // Case 2
      if (strcmp(temp->name, iterator->name) < 0) {
          temp->next = iterator;
          iterator->previous = temp;
          temp->previous = follower;
          follower->next = temp;
          return 2;
      }
      follower = iterator;
```

```
        iterator = iterator->next;
    }
    follower->next = temp;    // Case 3
    temp->previous = follower;
    temp->next = NULL;
    tail = temp;
    return 3;
}
int main() {
    int identity = 0;
    char *name1 = malloc(32);
    char *name2 = malloc(32);
    char *name3 = malloc(32);
    struct Node *temp1, *temp2;
    printf("Please enter 3 names:\n");
    scanf("%s", name1);    // enter John
    scanf("%s", name2);    // enter Mary
    scanf("%s", name3);    // enter David
    insertion(identity++, name1);
    insertion(identity++, name2);
    insertion(identity++, name3);
    temp1 = head;
    temp2 = tail;
    printf("ID = %d, name = %s\n", temp1->id, temp1->name);
    printf("ID = %d, name = %s\n", temp1->next->id, temp1->next->name);
    printf("ID = %d, name = %s\n", temp2->id, temp2->name);
    return 0;
}
```

Notice that dynamic memory is used for `name1`, `name2`, and `name3` in the main program. It is necessary in order to keep the memory in the linked list. If a local variable is used for the names, the memory will go out of scope when function exits. Memory management will be discussed in detail in the next chapter. The output of the program is as follows:

```
Please enter 3 names:
John
Mary
David
ID = 2, name = David
ID = 0, name = John
ID = 1, name = Mary
```

2.5.6 Stack

A **stack** is a data structure that can contain a set of ordered items. The items are ordered in such a way that an item can be inserted into the stack or removed from the stack at the same end. This end is called the top of the stack.

The stack is one of the most important data structures in computer hardware and software design. This section introduces the basic concept of stack through an example. More applications of the stack will be further discussed in Chapter 3 when we study memory management, in Section A.2 in Appendix A when we introduce basic computer architectures, and in A.3 when we discuss the implementation of function calls at the assembly language level.

Like any structured data type or a data structure, a stack is defined on simpler data types and the new operations on the data types.

Typically, a stack is defined on an array type. The basic operations defined on the stack are **push** (add an element onto the stack top) and **pop** (remove the top element from the stack). The code below shows the definition of a stack:

```
elementType stack[stackSize];
int top = 0;
void push(elementType Element) {
  if (top < stackSize) {
       stack[top] = Element;
       top++;
  }
  Printf("Error: stack full\n");
}
elementType pop() {
  if (top > 0) {
       top--;
       return stack[top];
  }
  Printf("Error: stack empty\n");
}
```

Now we use the stack to implement a four-function calculator that supports addition, subtraction, multiplication, and division operations on floating point numbers. The basic part of the implementation is the same as the code above, except that the `elementType` is now `float`, and four extra arithmetic functions are included. To perform operations, data are first pushed onto the stack. Every time an operation is performed, the two data items on the stack top are popped out for operation and the result is pushed back onto the stack.

```
#define stackSize 8  // a sample value
#include <stdio.h>
float stack[stackSize];
int top = 0;
void push(float Element) {
  if (top < stackSize) {
       stack[top] = Element;
       top++;
  } else
  printf("Error: stack full\n");
}
float pop() {
```

```
  if (top > 0) {
     top--;
     return stack[top];
  } else
  printf("Error: stack empty\n");
}
float add() {
  float y;
  y = pop() + pop(); push(y);
}
float sub() {
  float y;
  y = pop() - pop(); push(y);
}
float mul() {
  float y;
  y = pop() * pop(); push(y);
}
float div() {
  float y;
  y = pop() / pop(); push(y);
}
void main() {
  float x1 = 1.5, x2 = 2.5, x3 = 3.5, x4 = 4.5, x5 = 5.5, x6 = 6.5;
  push(x1); push(x2); push(x3);
  push(x4); push(x5); push(x6);
  add(); sub(); mul(); div(); add();
  printf("final value = %f\n", pop());
}
```

What is computed in the main program by the sequence of operations `add()`, `sub()`, `mul()`, `div()`, and `add()`? Figure 2.17 shows the stack status after each `push` operation and after each arithmetic operation. Initially, stack `top` = 0. It increments after each `push` operation. In each arithmetic operation, two `pop` operations and one `push` operation are performed, resulting in the `top` being decreased by one. The final value computed is 12.0. After the `pop` operation performed in the `printf` statement, the `top` returns to zero.

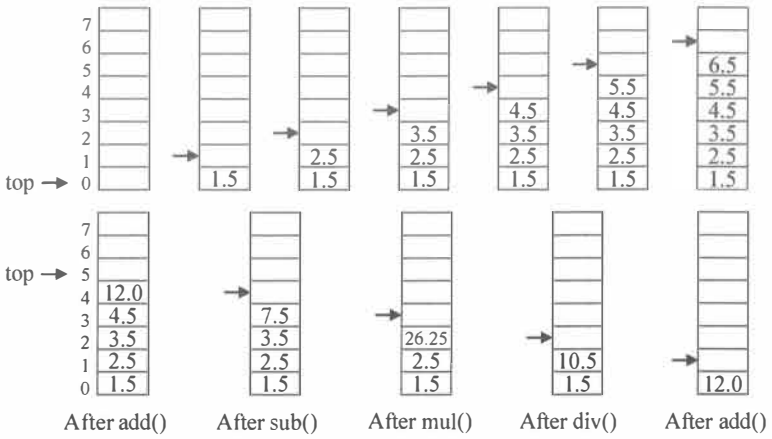


Figure 2.17. Stack status after each operation.

2.6 Standard input and output, files, and file operations

So far, we have discussed using memory (variables) to store data. However, memory is only a temporary place to store data. When we quit a program, all memory allocated to the program is taken back by the operating system for reuse. If our program has data that need to be stored for future use, we need to store the data into the permanent storage of a computer—the disk.

2.6.1 Basic concepts of files and file operations

Data stored on disk are organized in **files**. We consider a file as a structured data type and we access data in a file using a pointer to an object of type FILE, which records whatever information is necessary to control the stream of data.

As we know that disk operations are extremely slow, million times slower than memory operations, as it involves mechanical rotations of the disk and sliding of the read/write heads. The challenge is to make file operations faster. The solution is to use a buffer in the memory to hold a large block (e.g., 1024 bytes) of data. Each disk operation will transfer a block of data, instead of a byte or a word of data. Figure 2.18 shows how read and write operations are implemented.

For the read operations, the process is as follows:

- Declare a pointer f to a FILE type;
- Open a file for read: Create a buffer that can hold a block of bytes (e.g., 1024 bytes);
- Copy the first block of a file into the buffer;
- A program uses the pointer to read the data in the buffer;
- When the pointer moves down to the end of the buffer, copy the next block into the buffer;
- Close the file at the end of use.

For the write operation, the process is as follows:

- Declare a pointer f to a FILE type;
- Open a file for write: Create a buffer that can hold a block of bytes (e.g., 1024 bytes);
- A program uses the pointer to write the data in the buffer;
- When the buffer is full, copy the block into the disk;
- Move the pointer to the beginning for more write-operations;

- Close the file at the end of use.

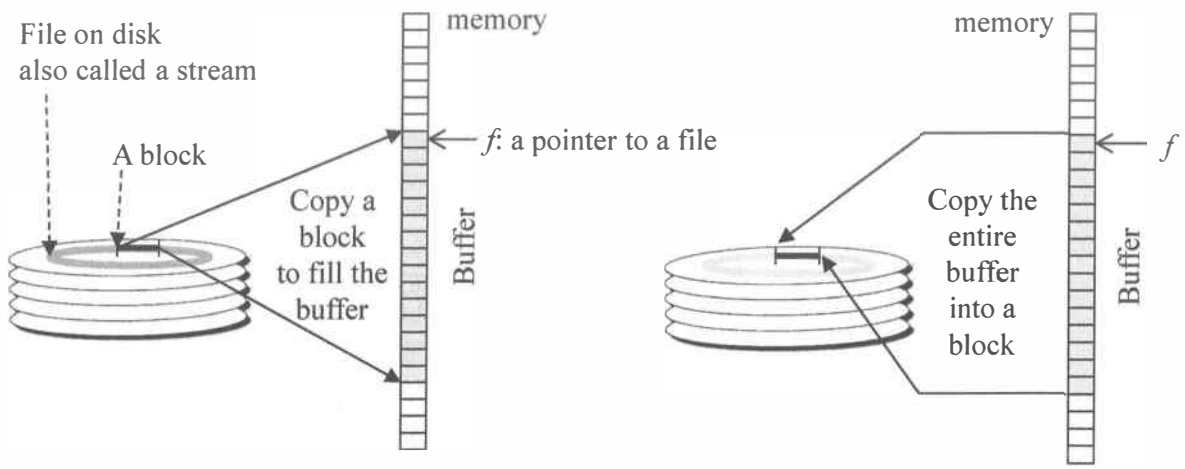


Figure 2.18. File read and write operations.

2.6.2 File operations in C

We focus on C file operations in this section, and we will discuss the C++ file operations in the next chapter. We will use the following example to demonstrate the basic file operations, including opening, reading, writing, and closing a file.

```
// demonstrate the use of fopen, fclose, feof, fgetc and fputc operations
#include <stdio.h>
#include <string.h>
// This function reads all characters in a file and puts them in a string
void file_read(char *filename, char *str) {
  FILE *p;
  int index=0;
  p=fopen(filename, "r");    // Open the file for "read".
  // Other options are "w" (write), "a" (append), and "rw" (read & write).
  while(!feof(p))// while not reaching the end-of-file character
       *(str+index++)=fgetc(p); //read a character from file and put it
                    // in str. p is incremented automatically.
  str[index]='\0';  // add the null terminator
  puts(str);        // print str. You can use printf too.
  fclose(p);        // close the file
}
// This function creates a new file (or opens an existing file), and then
// stores (puts) all characters in the string str into the file.
void file_write(char *filename, char *str) {
  int i, l;
  FILE *p;                  // declare a pointer to file type
  p=fopen(filename, "w");   // open/create a file for "write".
  l = strlen(str);          // get string-length
  for(i=0;i<l;i++)
```

```
        fputc(*(str+i),p);          // write a character to the file pointed
                                    // by p. p is incremented automatically.
    fclose(p);                      // Close the file.
}
// This function cipher-encrypts the string in variable str.
void encrypt(int offset, char *str) {
    int i,l;
    l=strlen(str);
    printf("original str = \n%s\n", str);
    for(i=0;i<l;i++)
        str[i] = str[i]+offset;
    printf("encrypted str = \n%s \nlength = %d\n", str, l);
}
void main() {
    char filename[25];
    char strtext[1024];
    printf("Please enter the name of the file to be read\n");
    // you should enter the name of an existing text file, e.g., letter1.txt
    scanf("%[^\n]s", filename);     //Read a line till the end-of-line "\n"
    file_read(filename, strtext);   //read text from file & put it in strtext
    encrypt(5, strtext);            //manipulate the string strtext
    printf("Please enter the name of the file to be written\n");
    scanf("%[^\n]s", filename);     //Read a line till the end-of-line "\n"
    file_write(filename, strtext); //write the text into the given file
}
```

This program first takes a file name from the keyboard, reads the file (we assume the file exists), and puts the entire contents of the file in a string variable `strtext`. Then we call the encrypt function to encrypt the string. Finally, we write the encrypted string into another text file.

In the program, we use the following basic file open operation:

```
p = fopen(filename, "r");
```

to open the file in "read" mode. The pointer `p` points to the first character in the text file (buffer). Other mode options are "`w`" for "write" and "`a`" for "append" data at the end of the file. In addition to these modes, the following characters can be included in *mode* to specify the translation mode for newline characters: "t": Open in text (translated) mode. In this mode, CTRL+Z is interpreted as an end-of-file character on input. "b": Open in binary (untranslated) mode; translations involving carriage-return and linefeed characters are suppressed. The letter "b" must be placed at the end of the mode string. If you want to open a file for both read and write operations, you can use an "+." Table 2.4 summarizes the mode definitions:

Mode	Description
r	Open an existing file for reading.
w	Open a file for writing. If the file does not exist, a new file is created. If the file exists, its content is cleared, and the file is written as an empty.
a	Open a file for appending. If the file does not exist, a new file is created.
r+	Open an existing file for reading and writing.
w+	Same as w mode, but allow both write and read.
a+	Same as a mode, but allow append and read.
b	Binary mode. The letter b can appear at the end or before +, e.g., both w+b and wb+ are acceptable.

Table 2.4. File operation modes.

Having opened a file, we can use the function

```
ch = fgetc(p);
```

to read the first character from the file. After each `fgetc` call, the pointer is automatically moved to the next position, ready for reading the next character. Another function is

```
fputc(ch, p);
```

that puts the character in parameter `ch` into the file at the position pointed to by `p`.

After we have completed file operations (read or write), we must close a file by using the file close operation

```
fclose(p);
```

If a file is not closed, the file descriptor that is used by the operating system to identify the file will not be freed. The total number of file descriptors that an operating system can issue is usually very limited. For example, in the Unix operating system, the file descriptor must be between 0 and 20. File descriptors 0, 1, and 2 are reserved for three system files: standard input, standard output, and standard error output, leaving only 18 file descriptors for all users concurrently using the operating system. If no file descriptors are available, the operating system will not be able to open any file for any user applications.

In the statement `scanf("%[^\n]s," filename)` in the program example above, the control sequence `[^\n]` ensures that the `scanf` reads until the `newline` symbol "`\n,`" including the spaces in the line.

The other file operations include:

- fread(buffer, size, count, fileName); // unformatted read
- fwrite(buffer, size, count, filename); // unformatted write
- scanf(control sequence, parameter list); // formatted input from keyboard
- printf(control sequence, parameter list); // formatted output to screen
- fscanf(filename, control sequence, parameter list);
  // This function is the same as scanf, except it inputs from a file
- fprintf(filename, control sequence, parameter list);
  // This function is the same as printf, except it outputs to a file

The definitions of the functions `fread` and `fwrite` are:

```
size_t fread(void *buffer, size_t size, size_t count, FILE *fileName );
size_t fwrite(const void *buffer, size_t size, size_t count, FILE *fileName
);
```

```
// Using these two functions requires including a <stdio.h> header
```

The functions are defined in the standard library `stdio.h`. The function `fread` returns the number of full items actually read, which may be less than `count` if an error occurs or if the end of the file is encountered before reaching `count`. You can use the `feof` or `ferror` function to distinguish a read error from an end-of-file condition. If `size` or `count` is 0, `fread` returns 0, and the `buffer` contents are unchanged.

The function `fwrite` returns the number of full items actually written, which may be less than `count` if an error occurs. Also, if an error occurs, the file-position indicator cannot be determined.

The parameters in the functions are specified as follows:

1. `buffer: pointer to the source variable (for fwrite) or to the destination variable (for fread)`
2. `size: item size in bytes`
3. `count: maximum number of items to be read or written. Normally, use 1.`
4. `fileName: pointer to FILE structure`

The following segment of code shows the application of `fread` and `fwrite`. It consists of two functions. The `save_file` function saves a linked list of nodes into a file called `fileName`, and the `load_file` function reads the file called `fileName`, and rebuilds the linked list according to the saved data. The segment of code demonstrates how to write and read strings and integers to and from a file.

```
// demonstrate the use of fopen, fclose fread and fwrite operations
void save_file() {
  FILE *fileName;
  personnel *node;
  char ch;
  long sid;
  fileName = fopen(file_name, "wb"); // w for write, b for binary mode
  if(fileName != NULL) {
      node = head;
      while(node != NULL) {
           fwrite(node->getName(), 30, 1, fileName);
           fwrite(node->getBirthday(), 11, 1, fileName);
           sid = node->getId();
           fwrite(&sid, sizeof(long), 1, fileName); // binary write
           node = node->getNext();
      }
  }
  else
      printf ("ERROR - Could not open file for saving data !\n");
}
void load_file() {
  FILE *fileName;
  personnel *node, *temp;
  char sname[30];
  char sbirthday[11];
```

```
    long sid;
    fileName = fopen(file_name, "rb");     // r for read, "b" for binary
    if(fileName != NULL) {
        while(fread(sname, 30, 1, fileName) == 1) {
            fread(sbirthday, 11, 1, fileName);
            fread(&sid, sizeof(long), 1, fileName); // read binary
            node = new personnel(sname, sbirthday);
            node->setId(sid);
            if(head != NULL)
                temp->setNext(node);
            else
                head = node;
            temp = node;
        }
        fclose(fileName);
    }
}
```

We have used `scanf` and `printf` to read from the keyboard and print to the screen. In fact, in C/C++, the keyboard is considered to be a read-only file (standard input file) and the screen is considered to be a write-only file (standard output file). Their file names are `stdin` and `stdout`, respectively.

The functions `fscanf` and `fprintf` are more general forms of file operations in which we can specify what file we want to read and write. The standard input and output functions `scanf` and `printf` are special cases of them and are equivalent to

```
fscanf(stdin, control sequence, parameter list);
fprintf(stdout, control sequence, parameter list);
```

The following example shows the application of `fprintf` and `fscanf`. First, an integer number and a float number are written in the file named `PersonData`. Then the file is closed and reopened for read. An integer number and a float number are read into two variables `len` and `hei`, respectively:

```
#include <stdio.h>
void main() {
  FILE *fileID;
  int length = 35429, len;
  float height = 5.8, hei;
  fileID = fopen("PersonData", "wb");    // open for write
  if(fileID != NULL) {
      fprintf(fileID, "%d\n", length);  // write an integer into a file
      fprintf(fileID, "%f\n", height);  // write a float into a file
  }
  else
      printf ("ERROR - Could not open file for saving data !\n");
  fclose(fileID);
  fileID = fopen("PersonData", "rb");    // open for read
  if(fileID != NULL) {
```

```
        fscanf(fileID, "%d", &len);        // read an integer from a file
        fscanf(fileID, "%f", &hei);        // read a float from a file
        printf("length = %d, height = %f\n", len, hei);
    }
    fclose(fileID);
}
```

In C++, similar input and output functions are defined:

```
cin >>
cout <<
cin.ignore();
cin.get(strvar, strlength, achar);
cin.getline(strvar, strlength, achar);
```

More details of C++ input, output, and file operations will be discussed in Chapter 3.

2.6.3 Flush operation in C

In the aforementioned programs, we used `fflush(stdin)` to remove the delimiter (a space, a newline, etc.) before using `getc(stdin)`. The reason is, the formatted input function `scanf` will read only up to the delimiter and leave the delimiter in the input buffer. If we do not call `fflush(stdin)`, the left delimiter will be read by an unformatted input function such as `getc(stdin)` and `gets(stdin)`. Thus, we must call function `fflush(stdin)` to flush the buffer of the standard input file `stdin`. It is not a problem if two consecutive `scanf` functions are called because a formatted input function can automatically remove the delimiter. The C++ function equivalent to `fflush` is `cin.ignore`, which will be discussed in Chapter 3.

C-styled `fflush(stdin)` function that flushes the input buffer to remove the remaining delimiters in the buffer of the standard input file `stdin` after a `scanf` operation. Consider the following snippet of code:

```
#include <stdio.h>
#pragma warning(disable: 4996) // comment out if not in Visual Studio
int main() {                   // Test fflush() function
  char strvar[8], ch;
  scanf("%s", strvar);  // Enter: Hi
  printf("%s\n", strvar);
  //fflush(stdin);      // Try the program with and without fflush()
  ch = getc(stdin);     // enter a character 'x'
  printf("%c\n", ch);
  printf("%s\n", strvar);
}
```

The inputs and outputs of the program with the fflush() (commented out) and without the fflush, respectively, are shown as follows:

With fflush: 

Without fflush: 

Figure 2.19 explains the reason why the character read getc(stdin) does not read the character into the variable `ch`, by illustrating the states of the input variable `strvar` and the input buffer in the execution process of the program above. Notice that the newline character '`\n`' is left in the buffer after the `scanf` operation is completed, and thus the next input character is appended to the character. The `getc` function reads a character from the input buffer, which will read '`\n`', without giving a chance to read the keyboard. As a result, no input is needed, '`\n`' is read into the input variable `ch`, and thus the keyboard operation is skipped.

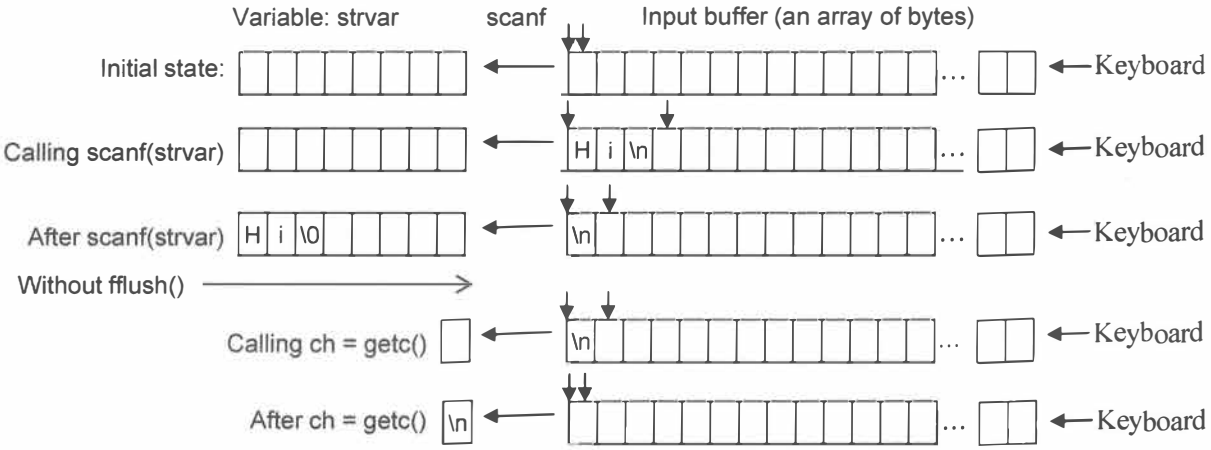


Figure 2.19. The input buffer between the keyboard and the input variable.

If the library function `fflush` does not exist in your environment, you can also write your own flush function to flush all the characters in the file buffer. The following snippet of code shows a simple implementation:

```
void myFlush(){ // Manually flush all characters in the stdin buffer
  int c;
  do {
      c = getchar();
  } while (c != '\n' && c != EOF); // EOF: End of File flag
};
```

When you switch back from unformatted input to formatted input, you normally do not need to put the delimiter back. However, it is a good idea to restore the character that is flushed. You can call the library function `ungetc('\n', stdin)` to put the newline character back into the buffer.

2.7 Functions and parameter passing

Functions, also called procedures or subroutines in some other programming languages, are named **blocks** of code that must be explicitly called. The purpose of functions is twofold:

- **Abstraction**: Statements in a function form a conceptual unit.
- **Reuse**: Statements in a function can be executed multiple times in the program.

As a part of a program, a function must communicate with the rest of the program. To pass values into a function (**in-passing**), we usually have two methods: global variable and parameter passing. To pass values out of a function (**out-passing**), we usually have three methods: global variable, parameter passing, and return value. Different programming languages have different value passing policies and mechanisms.

- In imperative and object-oriented programming languages like C/C++ and Java, all combinations of the in-passing and out-passing methods are allowed.
- In functional programming languages like Scheme or Lisp, parameter passing is the only in-passing method and return value is the only out-passing method allowed.
- In logic programming languages like Prolog, parameter passing is the only in-passing and the only out-passing method allowed.

Using a global variable to pass a value in or out of a function causes unwanted side effects and thus it is generally not recommended to use global variables for passing values. It is conceptually simple to use a return value to pass a value out of a function. We will thus focus on the parameter-passing mechanisms that pass values in and out of functions.

When we discuss parameter passing, we need to differentiate two kinds of parameters: formal parameters and actual parameters. **Formal parameters** are the parameters we use when we declare (define) a function. Formal parameters are local variables of the function. **Actual parameters** are the values or variables we use to substitute for the formal parameters when we call a function. Actual parameters are variables/values of the caller before the control enters the function. They become the variables/values of the function after the control enters the function.

Now the question is what would happen if we modify the formal parameters in the function. Will the modification have an impact on the actual parameters? The answer to the question depends on what kind of parameter-passing mechanisms we use. The most frequently used parameter-passing mechanisms are call-by-value, call-by-alias, and call-by-address.

Call-by-value: The formal parameter is a local variable in the function. It is initialized to the value of the actual parameter. It is a copy of the actual parameter. The modification of formal parameters has no impact on the actual parameters. In other words, call-by-value can only pass values into a function, but it cannot pass values outside the function. Functions using call-by-value must use return-value to pass a value to the outside. The advantage of call-by-value is that it has no side effects and it is considered a reliable programming practice. The drawback is that it is not convenient to handle structured data types.

The following piece of code demonstrates value in-passing by global variable and call-by-value mechanisms:

```
#include <stdio.h>
int i = 1;                  // i is a global variable outside any function
foo(int m, int n) {         // m and n are formal parameters
  printf("i = %d m = %d n = %d\n", i, m, n);
  i = 5; m = 3; n = 4;      // Modify i, m and n.
```

```
  printf("i = %d m = %d n = %d\n", i, m, n);
}
main() {
  int j = 2;              // j is a local variable, local to main() function
  foo(i, j);              // i and j are actual parameters of function foo
  printf("i = %d j = %d\n", i, j);
}
```

The output of the program is

```
i = 1  m = 1n = 2
i = 5  m = 3n = 4
i = 5  j = 2
```

As you can see, the global variable `i` is changed in the function and `i` remains changed after leaving the function. On the other hand, `j` is passed to formal parameter `n` and `n` is modified in the function. The modification to `n` has no impact on `j`.

Call-by-alias: It is also called call-by-reference or call-by-variable. The formal parameter is an *alias* name of the actual parameter. Call-by-alias can pass a value into and out of a function. However, it has a side effect, that is, a variable outside a function can be changed by an action in a function.

For call-by-alias, there is only one variable (memory location) with two names for the formal and actual parameters, respectively. Changing the formal parameter immediately changes the actual parameter. The actual parameter must be a variable. It cannot be a literal value because a value cannot have an alias. This mechanism is supported by C++, but not by C.

To declare a formal parameter `x` in call-by-alias, an ampersand symbol is prefixed to the parameter: `&x`. The following code demonstrates parameter passing by the call-by-alias mechanism, where the second parameter of the `foo` function is an alias to the corresponding actual parameter:

```
#include <iostream>
void foo(int, int &);     // forward declaration
int i = 1;
void main() {
  int j = 2;              // j is a local variable, local to main() function
  foo(i, j);              // i and j are actual parameters of function foo
  printf("i = %d j = %d\n", i, j);
  foo(j, i);              // i and j are swapped
  printf("i = %d j = %d\n", i, j);
}
void foo(int m, int &n) { // call-by-alias is applied to parameter n
  printf("i = %d m = %d n = %d\n", i, m, n);
  i = 5; m = 3; n = 4;    // Modify i, m and n.
  printf("i = %d m = %d n = %d\n", i, m, n);
}
```

The output of the program is

```
i = 1 m = 1 n = 2
```

```
i = 5 m = 3 n = 4
i = 5 j = 4              // notice that j is changed
i = 5 m = 4 n = 5
i = 4 m = 3 n = 4    // notice that i is changed immediately
i = 4 j = 4
```

This program is basically the same as the call-by-value example, except that the modification to the variable `n` in the `foo` function is passed to the variable j in the main program, in the first call, and is immediately passed to i in the second call to `foo`.

Call-by-address: It is also called call-by-pointer. The address of the actual parameter is passed into a local variable of the function. The actual parameter can be an address value or a pointer variable. You can use the address to modify directly the actual parameter pointed to by the address. You can also modify the address value stored in the formal parameter. However, this modification will not modify the actual parameter. In fact, for the pointer variable itself, call-by-value is applied. In the following example, we demonstrate parameter passing by call-by-address.

```
#include <stdio.h>
void foo(int *n) {              // declare call-by-address parameter
  printf("n = %d\n", *n);       // print the variable value pointed to by n
  *n = 30;                      // modify the variable value pointed to by n
  printf("n = %d\n", *n);       // print the variable value pointed to by n
again
  n = 0;                        // Modify the pointer itself.
}
void main() {
  int i = 15;
  foo(&i);
  printf("i = %d\n", i);
  i = 10;
  foo(&i);
  printf("i = %d\n", i);
}
```

The output of the program is

```
n = 15
n = 30
i = 30
n = 10
n = 30
i = 30
```

In the main program, we have a local variable `i`, initialized to 15. We then call function `foo` using `&i`, the address of variable `i`. The actual parameter is the address value (a pointer) to `i`, not the variable `i` itself. In the function definition body, the formal parameter is a pointer to an integer type. We use the pointer `n` to modify the variable `*n` pointed to by `n`, which is in fact `i`. Thus, we have indirectly modified variable `i` in the `main()` function. When the control exits the function, `i` remains modified. As the last statement in the function, we modify `n` itself. However, this modification will not have an impact on the address value

passed to n when the control exits the function. As you can see here, call-by-address and call-by-value are relative. We use call-by-address if our intention is to pass the variable pointed to by the pointer (address value) to the function. If we change the variable through its address, the variable remains changed after exiting the function. On the other hand, if the address value is what we really want to pass into the function, instead of the variable pointed to by the address, we are actually doing call-by-value.

Another application of call-by-address is in the situation where we want to pass a structure variable (an array, a string, or a structure) to a function. It is more convenient to pass the pointer of the structure variable, instead of passing the structure itself.

In Java, parameter passing is limited in such a way that:

- If the parameter is of a primitive type, only call-by-value is allowed;
- If the parameter is of a class-based type, only call-by-address is allowed.

Some books say that Java only supports call-by-value. They are referring to the pointer variable (reference) itself passed to a function. However, since the intention of passing the reference variable is to access the object pointed to by the reference, it is better to say that it supports call-by-address, instead of call-by-value. Notice that the pointer in Java is called "reference." However, Java's parameter passing for objects is not call-by-alias or call-by-reference, according to the definition of call-by-reference here.

In C/C++, all combinations of parameter passing are allowed. You can pass a pointer (call-by-address) or an alias (call-by-alias) to a simple variable (e.g., of integer type), or pass a complex type of variable using call-by-value, call-by-alias, or call-by-address.

The following program shows how to pass a string into and out of a function using call-by-address:

```
// file name: strop.c
#include <stdio.h>
#include <string.h>
char *getString(char *str){  //The function returns a pointer to its
parameter.
  return str;                // returns a pointer.
}
void setString(char *str1, char *str2) {  // Copy str2 into str1
  strcpy(str1, str2);
}
void main() {
  char *p, q[8] = "morning", *s = "hello";
  printf("s = %s\n", s);
  p = getString(s);
  printf("p = %s\n", p);
  setString(q, p);   // q is the address of the array-based string
  printf("q = %s\n", q);
}
```

The output of the program is

```
s = hello
p = hello
q = hello
```

2.8 Recursive structures and applications

Section 2 discussed basic control structures in C/C++. This section studies the complex recursive structures. We first compare recursive structures with the iterative structures. Then we formulate the generic steps of writing recursive functions. Finally, we use a longer example as a case study to go through the design steps. More examples of recursion will be studied in Chapters 4 and 5.

2.8.1 Loop structures versus recursive structures

A function (or procedure) is said to be **recursive** if the function calls itself. A recursive function can call itself anywhere, except the first statement, and once or multiple times, in the body of the function. If a recursive function calls itself only once and in the last statement, the function is said to be **tail-recursive**. Tail-recursion has the simplest structure.

Figure 2.20 compares three different repetition structures: (a) while-do-loop, (b) nontail-recursion, and (c) tail-recursion.

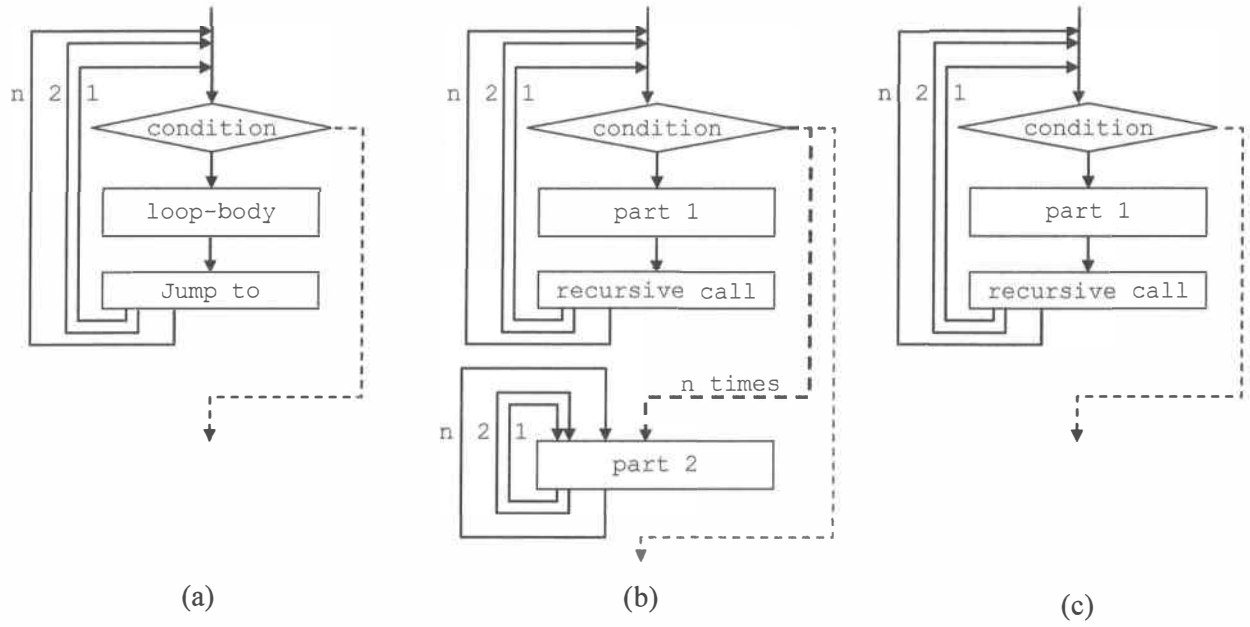


Figure 2.20. Three different repetition structures.

Although other loop structures do exist, like for-loop and do-while-loop, while-do-loop is sufficient to implement other possible loop structures. The nontail-recursion breaks its loop-body down into two parts, separated by the recursive call. Part 1 is first repeatedly executed `n` times, and then part 2 is executed `n` times. It is important to recognize that part 2 is executed the same number of times as part 1. The partially completed computations are stored on the stack. When part 1 is eventually repeated, the sufficient number of times or the stopping condition is satisfied, the control exit part 1 and enters part 2. Then part 2 will be executed the same number of times, and finally exit at the end of part 2.

In the case of tail-recursion, the recursive call is the last statement, and thus, there is no part 2. As we can see, the general recursive structure is very different from the iterative loop structure. However, the tail-recursive structure has exactly the same control structure as the while-do-loop. In other words, the while-do-loop structure is a special case of the recursive structure. We will see in Chapter 4 that functional program languages can use recursion as their only repetition structure, completely removing loop structures from the languages.

Here we are taking a glass-box approach to understand the recursive function; that is, we try to study recursion by trying to understand the structure and the control flow of the function. This is one of the most common approaches taken by many programmers. It works fine for simple recursive functions. However, if a recursive function has multiple recursive calls in its body, the structure will be far too complex to understand. We will take an innovative approach in this book to study the recursive function: the black-box approach, or so-called **abstract approach**. This approach works fine for both simple and complex recursive functions. We will see soon that this approach is far easier to understand and to apply to solve all kinds of recursive problems in all possible programming languages.

2.8.2 The fantastic-four abstract approach of writing recursive functions

The idea of recursion may not be as straightforward as iterative looping. However, writing recursive functions can be as simple as writing iterative functions, as long as you strictly follow the **fantastic-four abstract approach**. The approach was first proposed by one of the authors in the first edition of this book and was called "simple steps for writing recursive procedures." The approach has been rated by all students who learn it to be the most efficient method of teaching and learning recursion and was called by the students "fantastic-four." In the second edition, it is formally named the fantastic-four abstract approach, which consists of the following steps:

1. **Formulate the size-n problem:** Recursion is necessary only if you want to solve a problem that needs to repeat the same operations for a number of times. We assume the number of repetition is `n`. In most cases, `n` is obvious. For example, if we want to compute factorial `n!`, the size `n` is already given. Formulating a size-n problem is merely choosing a function name, using `n` as the parameter, and defining the return type (not the return value) of the function. It is similar to writing the forward declaration of a function in C. Thus, the size-n problem for a factorial problem is

```
int factorial(int n);
```

The return value of the size-n problem is what the function is supposed to compute, or the value we are looking for. In this step, we do not need to design the solution for the size-n problem.

2. **Find the stopping condition and the corresponding return value:** The body of a recursive function should begin with checking the stopping condition. If the stopping condition is true, the function returns the corresponding value and exits. Otherwise, it calls the function itself. In most cases, identifying the stopping condition and corresponding value is trivial or given. For example, the stopping condition of `factorial(n)` is `n = 0`, and the corresponding value is 1.

3. **Select `m` and formulate the size-m problem:** After we have formulated the size-n problem, the size-m problem is easy: We simply replace parameter `n` by `m` in the size-n problem, where `m < n`. Size m is determined by how much we can reduce the size of the problem in one iteration. If we can only reduce the problem size by 1, `m` is `n-1`, and thus our task in this step is formulating a size-(n−1)problem. For example, the size-(n−1) factorial problem is simply `factorial(n-1)`. Sometimes, we may need to find an `m` that is not `n-1`. It is application-specific to find a proper `m`. We will use several examples to illustrate this point in this section and study many more examples in Chapter 4. Most students who have difficulty comprehending recursion misunderstand this step: They try to define a solution, or the return value, of size-m problem here in this step! It is not possible and it is not necessary to produce the return value in this step. All we need to do about the return value here is exactly the same as what we did in step 1. We simply assume the size-m problem will return a value and use this value in step 4. For example, the return value of size-(n−1) factorial problem is `factorial(n-1)`.

4. **Construct the solution of the size-n problem:** In this step, we will use the assumed solution or return value for size-m or size-(n−1) problem to construct the solution of the size-n problem. Again, this is

application-specific. In the case of the factorial problem, the solution of the size-n problem is `n*factorial(n-1)`.

Sometimes, we may need to use the return values of multiple size-m problems, where $0 \leq m < n$ (assume size-0 is the stopping condition), to construct the solution of the size-n problem.

Strictly following these steps, we can define the complete factorial function as follows:

```
int factorial(int n) {          // size-n problem
  if  (n == 0)                  // Stopping condition
      return 1;                 // Return value at the stopping condition
  else
      return n * factorial(n - 1); //use size-(n-1) problem's assumed
      // solution to construct size-n problem's solution
}
```

2.8.3 Hanoi Towers

The Hanoi Towers game is a good example used for explaining recursion. As shown in Figure 2.21, the rules of playing the game are:

- There are three pegs, and `n` successively smaller disks are initially placed on the left peg. In the example in Figure 2.21, `n` = 4. The objective is to move all disks to the right peg. The center peg can be used as an auxiliary holding (spare) peg.
- Disks may be moved from one peg to another. Only one disk may be moved at a time.
- The only disks that may be moved are the top disks on one of the three pegs.
- At no time may a larger disk may be placed on a smaller disk.

Now we follow the fantastic-four abstract approach to define a solution for the Hanoi Towers problem.

1. Formulate the size-n problem

We can simply formulate the size-n problem as `void hanoi(int n)`. However, in the return value (solution), we need to print how to move one disk from one peg to another in each step, and we need to name these three pegs. We could hard code the names as p1, p2, and p3; or left, center, and right. To increase the flexibility of the code, we add three parameters to the function, so that the user can pass different names into the function. Thus, we formulate the problem as

```
void hanoitowers(int n, char *left char *center char *right);
```

Notice that the function does not return a value; instead, it prints instructions (steps) for how to move `n` disk from the `left` peg, using the `center` peg as the auxiliary, to the `right` peg.

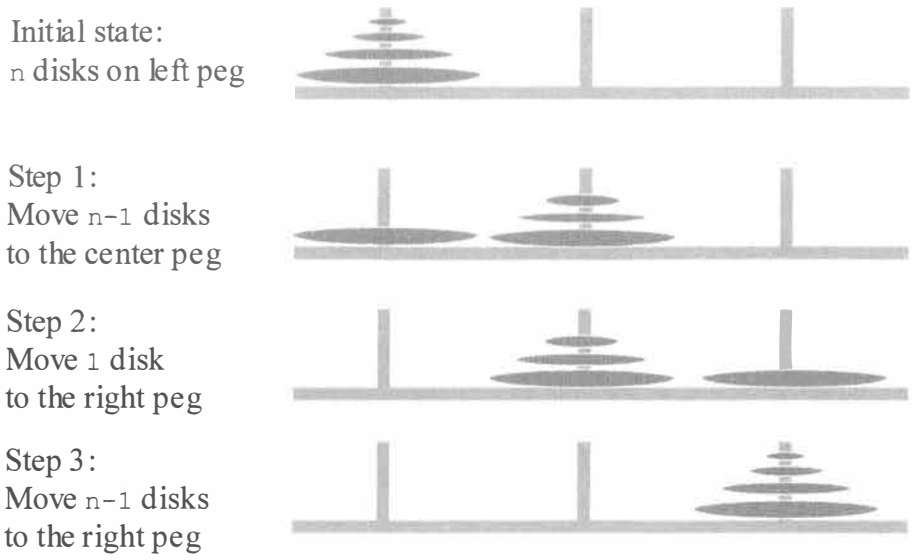


Figure 2.21. Solving Hanoi Towers problem.

2. Find the stopping condition and the corresponding return value

The stopping condition is `n = 1`. In this case, the size-1 problem is `hanoitowers(1, left, center, right)`, and the solution is to print "move the disk from the `left` peg to the `right` peg."

3. Select m and formulate the size-m problem

Since we can move only one disk at a time, it is obvious that we can only reduce the size by one in one iteration. Furthermore, since we have multiple parameters in the function, we could have multiple size-(n−1) problems. The following are six possible size-(n−1) problems:

(1) move `n-1` disks from `left` to `right`, using `center` as auxiliary:

```
hanoitowers(n-1, left, center, right)
```

(2) move `n-1` disks from `left` to `center`, using `right` as auxiliary:

```
hanoitowers(n-1, left, right, center)
```

(3) move `n-1` disks from `center` to `left`, using `right` as auxiliary:

```
hanoitowers(n-1, center, right, left)
```

(4) move `n-1` disks from `center` to `right`, using `left` as auxiliary:

```
hanoitowers(n-1, center, left, right)
```

(5) move `n-1` disks from `right` to `left`, using `center` as auxiliary:

```
hanoitowers(n-1, right, center, left)
```

(6) move `n-1` disks from `right` to `center`, using `left` as auxiliary:

```
hanoitowers(n-1, right, left, center)
```

4. Construct the solution to the size-n problem

Use the solutions for size-(n−1) problems to construct the solution for the size-n problem.

Figure 2.21 and the text on the left-hand side showed how we construct the solution for the size-n problem based on the solutions for size-(n−1) and size-1 problems, that is,

```
hanoitowers(n-1, left, right, center)    // move n-1 disks left -> center
hanoitowers(1, left, center, right)      // move 1 disk left -> right
hanoitowers(n-1, center, left, right)    // move n-1 disks left -> center
```

In words, the solution for the size-n problem is: (1) Move `n-1` disks from `left` peg to the `center` peg. We simply assume that we can do it, because it is a size-(n−1) problem. (2) Move the remaining disk from `left` to `right`. (3) Move `n-1` disks from `center` to the `right`.

Once we have designed the solution, we can easily obtain the C program that solves the Hanoi Towers problem as follows:

```
#include <stdio.h>
void hanoitowers(int n, char *S, char *M, char *D) {
  if (n == 1) {          // stopping condition
     printf("move top from %s to %s\n", S, D);
     // output at stopping condition
  } else {               // from size-(n-1) to size-n problem
     hanoitowers(n-1, S, D, M);
     hanoitowers(1, S, M, D);
     hanoitowers(n-1, M, S, D);
  }
}
void hanoi(int n) {      // define a simpler human-interface
  hanoitowers(n, "Left", "Center", "Right");
}
void main() {
  hanoitowers(3, "Source", "Spare", "Destination");
  printf("-----------------------------------\n");
  hanoi(4);
}
```

In the program, we defined a one-parameter function `hanoi(n)` as a simpler user interface, in case the user wants the hard-coded peg names. The function with more parameters is defined as a recursive function. When the `main()` function is executed, the functions `hanoitowers(3, "Source", "Spare","Destination")` and `hanoi(4)` will be called, resulting in the following output describing how to solve the size-3 and size-4 Hanoi Towers problems:

```
move top from Source to Destination
move top from Source to Spare
move top from Destination to Spare
move top from Source to Destination
move top from Spare to Source
move top from Spare to Destination
move top from Source to Destination
-----------------------------------
move top from Left to Center
```

```
move top from Left to Right
move top from Center to Right
move top from Left to Center
move top from Right to Left
move top from Right to Center
move top from Left to Center
move top from Left to Right
move top from Center to Right
move top from Center to Left
move top from Right to Left
move top from Center to Right
move top from Left to Center
move top from Left to Right
move top from Center to Right
```

As you can see from the example, the most important idea of recursive functions is that we simply assume that we have the solution for the size-(n−1) problem and we do not need to solve it. Why does it work? Because the recursive mechanism will actually solve the problem from size-1 upward automatically; that is, it will solve the size-1 problem, then it will use the solution to the size-1 problem to construct the solution to the size-2 problem, and so on. Since we have given the solution to the size-1 problem and we have defined how to find the solution to the size-n problem based on the solution to the size-(n−1), we basically have given solutions to the problem of all sizes of problems!

2.8.4 Insertion sorting

Now we will follow the fantastic-four abstract approach to solve the **sorting problem** in a simple way (insertion sorting) to demonstrate recursion. Assume that we have an array containing `n` integers: `A[n]`. The task is to sort the `n` numbers in ascending order.

1. Formulate the size-n problem.

We can simply formulate the size-n problem as

```
int* sorting(int *A, int n);
```

where `A` is the initial address of the array to be sorted and `n` is the size of the array. The function will return the initial address of the sorted array.

2. Find the stopping condition and the corresponding return value.

The stopping condition is `n = 1`. In this case, the size-1 problem is `sorting(int *A, 1)`, and the solution or return value is the address of `A`. `A` is not changed because `A` has only one element and is already sorted.

3. Select m and formulate the size-m problem.

Here we take a simple approach by reducing the size of the problem by 1. Thus, `m = n - 1` and the size-(n−1) problem is

```
sorting(B, n-1);
```

where `B` is the address of an array of size-(n−1). We **assume** `B` will be sorted if we call this function. This is very important!

4. Construct the solution of the size-n problem.

Since step 3 can solve the size-(n−1) problem, it is easy to solve the size-n problem:

(1) We split array `A` into two parts: The subarray of the first n−1 elements is `B` and the remaining element is `x`.
(2) We call the function in step 3 to sort the size-(n−1) array `B`.
(3) We find the right position `p` for inserting `x` into `B`.
(4) We make space for `x` by shifting the elements after position `p` one place right.
(5) We insert `x` at the position `p`.

Figure 2.22 graphically illustrates the four steps of implementing the recursive sorting algorithm.

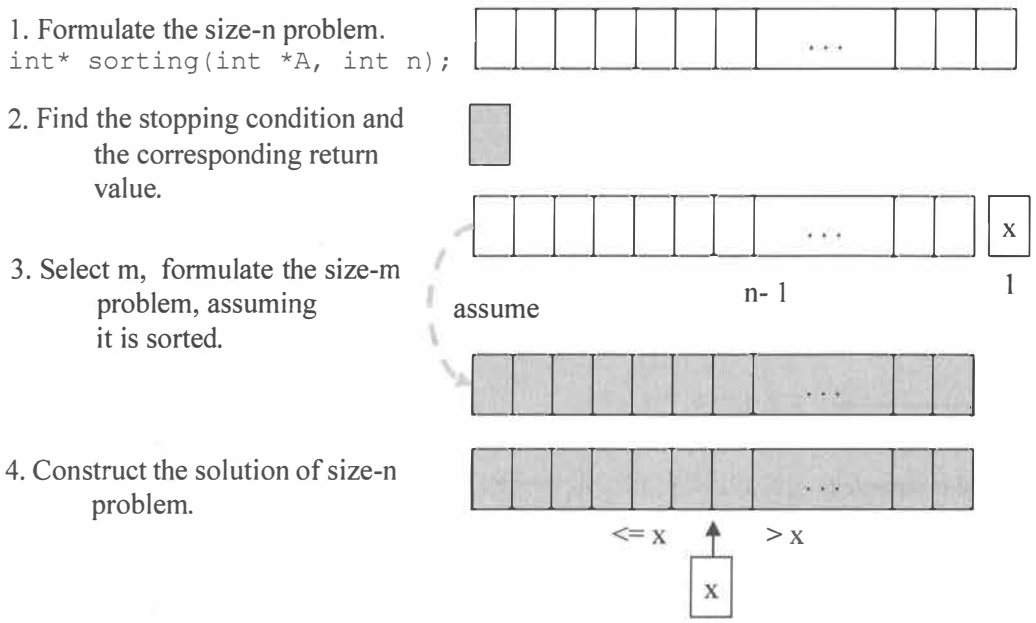


Figure 2.22. The fantastic-four steps sorting an array recursively.

The following program implements the four steps in the abstract approach. The comments in the program associate the statements with the four steps described above.

```
#include <stdio.h>
int* sorting(int *A, int n) {
  int *B, i, j, p = n-1, x;
  if (n==1) return A;          // stopping condition and return value
  else {
     x = A[n-1];               // Store the last element in x
     B = sorting(A, n-1);      // size-(n-1) problem
     i = 0;
     while (i < n-1) {         // Start to construct size-n solution
          if (x < B[i]) {
               p = i;  // locate the position p for x
               i = n;  // exit the loop
          }
```

```
            else i++;
        }                       // x should be inserted at position p
        for (j = p; j < n-1; j++)  // make space
            B[n-1-(j-p)] = B[n-1-(j-p)-1];
        B[p] = x;               // put x in the right place
        return B;
    } // end of else branch
}
void main() {
    int *SA, i, k, A[] = {3, 2, 4, 2, 9, 7, 1, 6}; // sample array
    k = (int)sizeof(A)/sizeof(int);    // get the length of the array
    SA = sorting(A, k);
    for (i = 0; i < k; i++)
        printf("%d, ", SA[i]);
}
```

Figure 2.23 illustrates the execution process and the changes of the array. In part 1 of the recursive function (before the recursive call), the array size is reduced by 1 every time the recursive function is called, till n = 0. Since the array index starts from 0, the array has one element when n = 0, as shown on the left-hand side of the figure. On the right-hand side, corresponding to part 2 (after the recursive call), the last element is inserted into the right position to form the size-n problem's solution at each level of recursion.

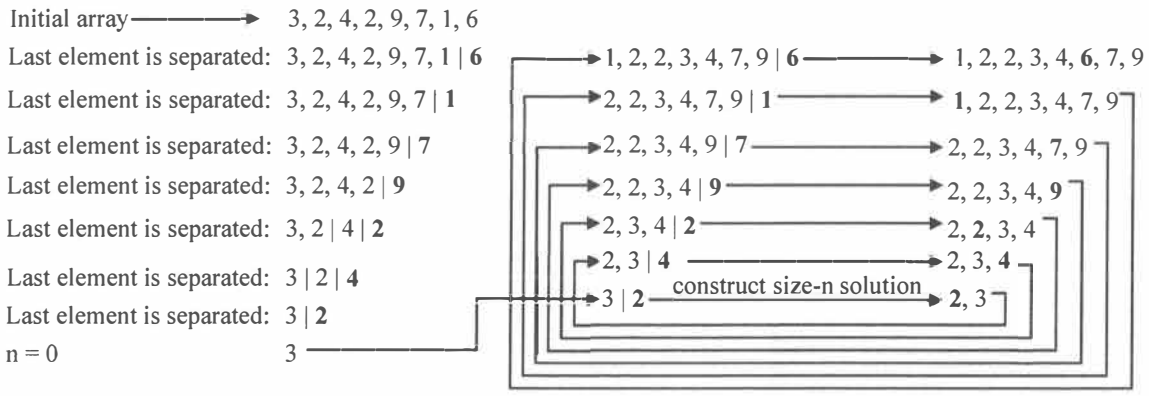


Figure 2.23. Three different repetition structures.

2.8.5 Merge sort algorithm

For some problems, it is possible to reduce the size by more than 1, resulting in a more efficient solution. For example, in step 3 of the above sorting example, we could select `m` = ⌊`n/2`⌋ (floor of `n/2`). In other words, we divide the size-n problem into two approximately equal-sized problems by dividing the array `A` into two half-sized arrays `B1` and `B2`. Then we call

```
sorting(B1, ⌊n/2⌋);   // floor of n/2
sorting(B2, ⌈n/2⌉);   // ceiling of n/2
```

respectively and have both `B1` and `B2` sorted. Then we merge `B1` and `B2` into an array `B` by comparing the elements of the two subarrays sequentially. This sorting algorithm is called **merge sort**, which is one of the

most efficient sorting algorithms with a complexity of O(*n*log*n*). The pseudo code in Figure 2.24 shows the sorting process through an example, illustrating how each subarray is split, sorted, and merged.

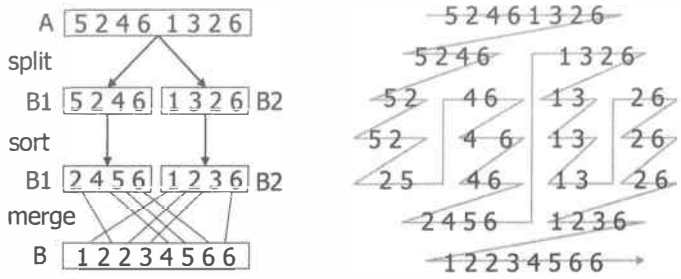
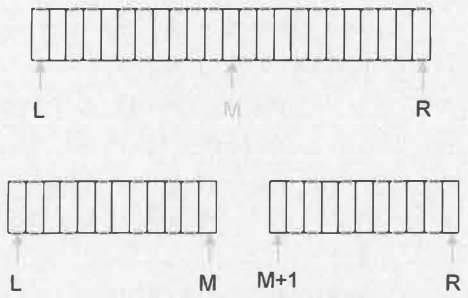


Figure 2.24. Merge sort and its sorting process.

The pseudo code below shows the recursive algorithm that implements merge sort. In the algorithm, the floor(x) function rounds x downward, returning the largest integer value that is not greater than x.

```
mergesort (A, L,R) {
   if R > L then
        { M = floor((R+L)/2);  // rounds down (R+L)/2 to integer
        mergesort (A, L, M);
        mergesort (A, M+1,R);
        merge(A, L, M, R); }
   else
        return A;
}
merge (A, L, M, R) {
   for i = M down to L do B[i] = A[i];
   for j = M+1 to R do  B[R+M+1-j] = A[j];
   i = L;
   j = R;
   for k=L to R do {
      if B[i] < B[j] then
        { A[k]=B[i]; i = i+1; }
      else
        { A[k]=B[j]; j = j-1; }
   }
}
```



Please study the pseudo code above and identify the code for defining:

1. size-n problem
2. stopping condition and return value
3. size-m problems
4. the size-n solution from the size-m solutions

2.8.6 Quick sort algorithm

Merge sort has the best complexity O($n$log$n$) in all the comparison-based sorting algorithms. The big O notation is defined based on the worst case execution time. However, merge sort is not the fast algorithm in terms of the average execution time, which is calculated based on the mean value of all possible input combinations of the input array. Quick sort, on the other hand, the complexity O(n^2). However, the average execution time beats merge sort.

The idea of quick sort is to pick a value (any value) in the array as the pivot value to divide the array into two subarrays, one with all its elements less than the pivot value, and the other with all its value greater than or equal to the pivot value. The pseudo code below shows the recursive algorithm that implements quick sort. In Chapter 5, we will give a full implementation of quick sort in Prolog.

```
void Quicksort (A, L, R) {
  if R = L then return; else {
       k = split(A, L, R);
       Quicksort (A, L, k-1);
       Quicksort (A, k+1, R);
  }
}
int split(A, L, R) {
       int pivot = A[R]; i = L; j = R;
       while i < j do {
            while (A[i] < pivot) do i = i+1;
            while ((j > i) && (A[j] >= pivot)) do j = j - 1;
            if  (i < j) then
                swap(A[i], A[j]);  // swap the values
       }
       swap(A[i], A[R]);
       return i;
}
```

2.8.7 Tree operations

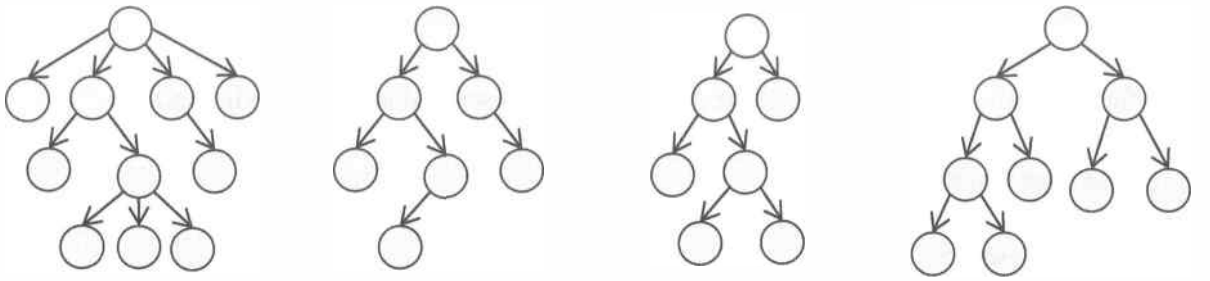
Graphs and trees are widely used data structures. A graph is a mathematical model and a data structure that is widely for representing related objects. A graph consists of a set of nodes and a set of edges between the nodes. A graph is a **directed graph** if a direction is defined for each edge, and a graph is an **undirected graph** if there is no direction defined for any edge. Assuming that the direction of the edge is the driving direction of streets, a directed graph allows the flow following the edge directions. An undirected graph allows the flow in both directions of the edge. A **path** from Node `N1` to Node `N2` is a sequence of edges connecting `N1` to `N2`, for example, `(N1, X1), (X1, X2), (X2, X3), …, (Xk, N2)`.

A directed graph is a tree, if it does not contain a loop and there is no more than one path between any two nodes. A tree is a rooted tree if it has a unique node that does not have an incoming edge. This node is the root of the tree. A node that does not have outgoing edges is a leaf. The height or depths of a tree is the number edges from the root to the deepest (farthest) leaf.

A tree is a binary tree, if any of its nodes can have at most two child (next) nodes. In a full binary tree, each node has either no child or two children. A balanced binary tree is a tree in which the heights (depths) of

the two subtrees of every node never differ by more than 1. Its height is $O(\log_2 n)$. Figure 2.25 shows a rooted tree, a binary tree, a full binary tree, and a balanced binary tree.

A binary search tree stores data in such a way that it keeps keys (used for indexing data) in sorted order, so that data search will be much faster.



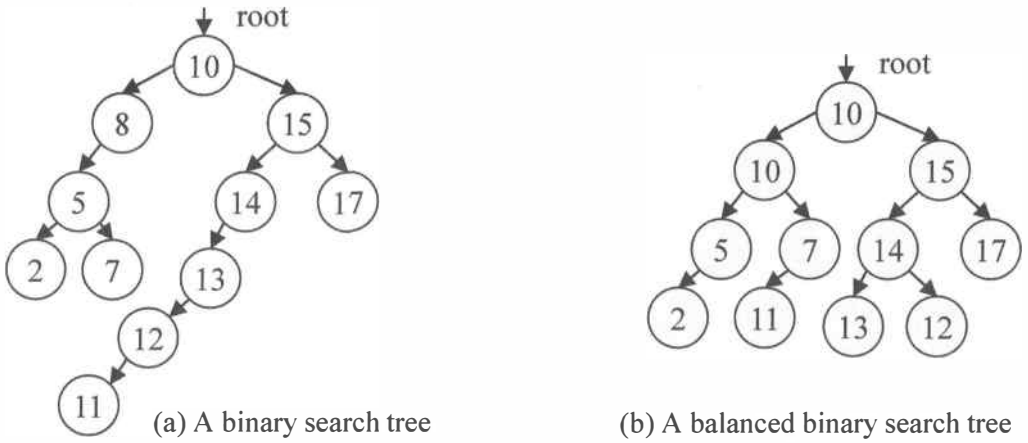
(a) Rooted tree (b) Binary tree (c) Full binary tree (d) Balanced binary tree

Figure 2.25. Binary trees.

When inserting data into a binary search tree, the simplest algorithm is

1) If the tree is empty, insert the first number as the root;
2) If the tree is not empty:
    a. If the incoming number is small than the key of the current node, insert the number to the left subtree using recursion;
    b. If the incoming number is greater than or equal to the key of the current node, insert the number to the right subtree using recursion.

Using this simple insertion algorithm, the binary search will not be balanced. In the worst-case scenario, when the input numbers are sorted, the tree becomes a linked list. Figure 2.26 shows a not balanced binary search tree and a balanced binary search tree.



(a) A binary search tree (b) A balanced binary search tree

Figure 2.26. A binary search tree and a balanced binary search tree.

To maintain the binary search tree balanced when inserting, a much more complex algorithm must be applied. Red-black tree is a data structure that attempts to have the tree balanced during insertion, with a topic of algorithm class.

As data are searched much often than data are inserted, it is critical to make search faster than insertion. The complexity of search algorithms on different data structures is listed as follows:

- The complexity is O(n) for linear search of data stored in arrays or linked lists.
- The complexity is O(n) for binary search is $O(\log_2 n)$, if the binary tree is balanced.
- The complexity is O(n) for binary search is $O(\log_2 n)$, if the binary tree is binary.

The following code shows a simple implementation of search and insertion of binary search tree, where the common functions `main` and `branching` are similar to those discussed in the linked list section.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h> // used by malloc
#include <time.h>
struct treeNode {
  int data;
  struct treeNode *left, *right; // pointers to left and right
} *root = 0;          //root is a global pointer to the root entry
void branching(char);             // function forward declaration
void insertion();
struct treeNode *search(struct treeNode *, int);
void traverse(struct treeNode *);
main( ) {   // print a menu for selection
  char ch = 'i';
  srand( (unsigned)time( 0 ) );  // Use current time as seed
  while (ch != 'q') {
      printf("Enter your selection\n");
      printf("      i: insert a new entry\n");
      printf("      s: search an entry\n");
      printf("      t: traverse the tree and print\n");
      printf("      q: quit \n");
      fflush(stdin);    // flush the input buffer
      ch = tolower(getchar());
      branching(ch);
  }
}
void branching(char c) {      // branch to different tasks
  int key;
  switch(c)  {
  case 'i':
      insertion();           // Not passing root, but use it as global
      break;
  case 's':
      printf("Enter the key to search\n");
      scanf("%d", &key);
      search(root, key);     // root call-by-value
```

```
        break;
    case 't':
        traverse(root);        // print all data
        break;
    default:
        printf("Invalid input\n");
    }
}
struct treeNode * search(struct treeNode *top, int key) {
    struct treeNode *p = top;
    if (key == p->data)
        printf("data = %d\n", p->data);
    if (key <= p->data) {
        if (p->left == 0) return p;
        else search(p->left, key);
    }
    else {
        if (p->right == 0) return p;
        else search(p->right, key);
    }
}
```

Deletion is more complex than insertion and search. Three cases need to be considered, as shown in Figure 2.27, where the shaded node is the node to be deleted.

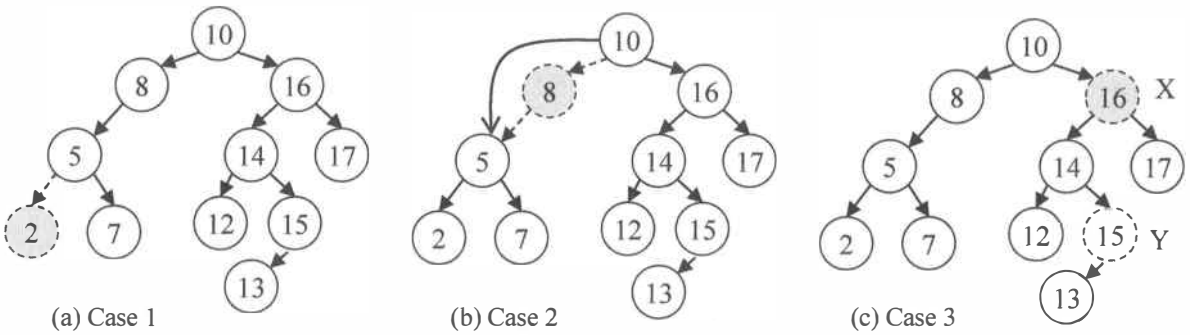


Figure 2.27. Three cases in deletion of a node.

Case 1: If the node to be deleted is a leaf, the node can be simply deleted.

Case 2: If the node to be deleted has only one child node, we can link the parent node to the child node.

Case 3: If the node X to be deleted has two child nodes, we can use a search function to find Y, the successor of X, which is the large node that is smaller than X. We use Y to replace X. Then, the problem becomes deleting Y from the tree. We repeat the same process with three possible cases for deleting Y. In the example in Figure 2.27, Y falls into case 2, and we can use node 14 to 13.

2.8.8 Gray code generation

The Gray code, named after Frank Gray, also known as reflected binary code, is a binary coding system where two successive values differ in only one digit. The Gray code was originally designed for preventing spurious output from electromechanical switches. Today, the Gray code is widely used in facilitating error correction in digital communications such as digital terrestrial television and some cable TV systems.

The main feature of the Gray code is that an n-bit Gray code can be constructed from (n-1)-bit code in the process shown in Figure 2.28.

Following the fantastic-four abstract approach, we can formulate the problem and its solution in the following four steps:

1. Formulate the size-n problem.

   char *gcode(int n); // will return the array of n-bit gcode

2. Find the stopping condition and the corresponding return value.

   If n = 1, return array {'0', '1'};

3. Formulate the size-(n-1) problem, assuming the gcode is found for the size-(n-1) problems.

   gcode(n-1) will return the (n-1)-bit gcode

4. Construct the solution of size-n problem.

   part1 = gcode(n-1);
   part2 = reverse(part1)
   left-append '0' to each item in part1
   left-append '1' to each item in part2
   return: part1 and part2

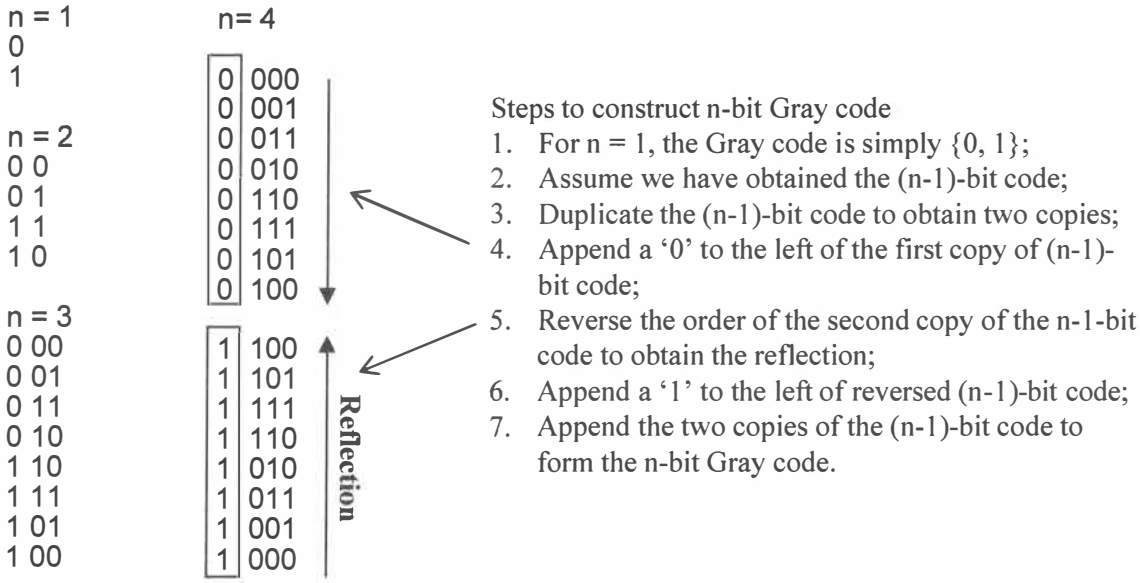
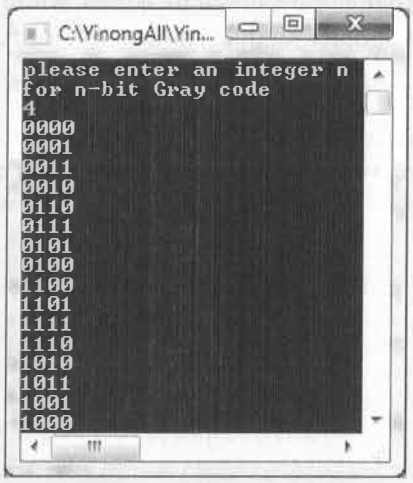


Figure 2.28. Generating n-bit Gray code from (n-1)-bit Gray code.

The complete Gray code generation function and a sample main program are given as follows:

```
#include <stdio.h>
#include <stdlib.h>  // malloc
#include <math.h>    // double pow(double, double)
#define columns 8    // The example limits the size to 7 columns
#define rows 256     // 256 = pow(2, 8)
char **gcode(int n);
void main() {
  char **g; int i, n, p;
  printf("please enter an integer n for n-bit Gray code\n");
  scanf("%d", &n);   // Note: 0 < n < columns
  g = gcode(n); // Call recursive gcode function
  p = (int) pow(2, n);
  for (i=0; i<p; i++)
     printf("%s\n", g[i]); // Print each element in array
}
char **gcode(int n) {
  int i, j, p, q;
  char **sizem, **sizen;     // pointers to 2-D arrays
  p = (int) pow(2, n);  // The length of size-n-code
  q = (int) p/2;        // create an array of pointers
  sizen = (char **) malloc(sizeof(char[rows]));
  for (i =0; i<p; i++)
     sizen[i] = (char *) malloc(sizeof(char[columns]));
  if (n<=1) {              // stopping condition
     sizen[0][0] = '0';
     sizen[0][1] = '\0';    // add terminator
     sizen[1][0] = '1';
     sizen[1][1] = '\0';    // add terminator
     return sizen;
  }
  else {
     sizem = gcode(n-1);
     for (i = 0; i < q; i++) {
          sizen[i][0] = '0';
          for (j = 1; j<=n; j++)
               sizen[i][j] = sizem[i][j-1];
     }
     for (i = 0; i < q; i++) {
          sizen[q+i][0] = '1';
          for (j = 1; j<=n; j++)
               sizen[q+i][j] = sizem[q-i-1][j-1];
     }
     for (i = 0; i < q; i++)
          free(sizem[i]);  // free each row
```



```
    free(sizem);                    // free the index
        return sizen;
    }
}
```

2.9 Modular design

Functions bring a level of abstraction into our programs. The abstraction makes our program easier to understand and to manage. However, the programming task can still become too large to understand. We need to introduce another level of abstraction, that is, modular design. Other advantages of modular design include:

- **Sharing**: We can group some frequently used functions and data into a module for being shared with other programmers (e.g., library functions).
- **Separate compilation**: This is a maintenance issue. If we find a programming error or we need to make functional modifications in a part of the program, we do not have to recompile the entire program.
- **Expandability**: We can easily add new modules into the system.

So far, we have been focusing on designing a program to solve relatively small problems. This is called **programming-in-the-small**. We need the skill of programming-in-the-small before we can do **programming-in-the-large**, which combines programming modules into a large program.

To design a module in a large system, we need to separate the specification from the implementation. The **specification** part tells what the module does and gives an external view of the module, while the **implementation** part gives code that implements the specification. Variable and function names given in the specification part are available to users inside the module as well as outside the module.

All programming languages provide mechanisms to support modular design. In C, specifications of programs are stored in .h files, while implementations are stored in .c files. In order to use functions defined in another module named, say, `modulename.c`, we need to use

```
#include "modulename.h"  // user-defined header files are quoted by "..."
```

Consider the traffic light example in Section 2.4. Since the `sleep` function may be used by other programs, we want to put this function and possibly other frequently used functions into a module, say, called `mylib.c`, which contains the following code:

```
// file name: mylib.c
#include<time.h>
const float pi = 3.14159265;
// Sleep for a specified number of seconds.
void sleep(int wait) {
  clock_t goal;                 // clock_t defined in <time.h>
  goal = wait * CLOCKS_PER_SEC + clock();
  while( goal > clock() )
      ;
}
// This function computes the volume of a cylinder:
double cylinder (int h, int r) {  //h: height, r: radius
```

```
    const double pi = 3.14159265;
    return pi*r*r*h;
}
```

Notice that a module does not need to have a `main()` function.

Then, we can put the headers of all functions, the type definitions, as well as the global variables to be shared among different modules, in the header file called `mylib.h`. In this example, we have only two functions and one type definition. Thus, the header file should look like:

```
typedef enum {red, amber, green} traffic_light;
void sleep(int wait);
double cylinder (int h, int r);
```

In the main program of the traffic light example, we do not need the function `sleep`, or the type definition. The code of the main function is as follows:

```
#include <stdio.h>  // system library uses angle brackets to include
#include "mylib.h"  // user library uses quotes to include
main() {
traffic_light x = red;
printf("Red:\tStop!\n");
while (1)
   switch (x) {
        case amber:
             sleep(1);          //sleep 1 second
             x = red;
             printf("Red:\tStop!\n"); break;
        case red:
             sleep(6);          //sleep 6 second
             x = green;
             printf("Green:\tGo>>>\n"); break;
        case green:
             sleep(12);         //sleep 12 second
             x = amber;
             printf("Amber:\tBrake...\n");
   }
}
```

In Visual Studio programming environment, all modules (.c files) should be placed in the folder "Source Files" and all the user-defined header files should be placed in the folder "Header Files," as shown in Figure 2.29.

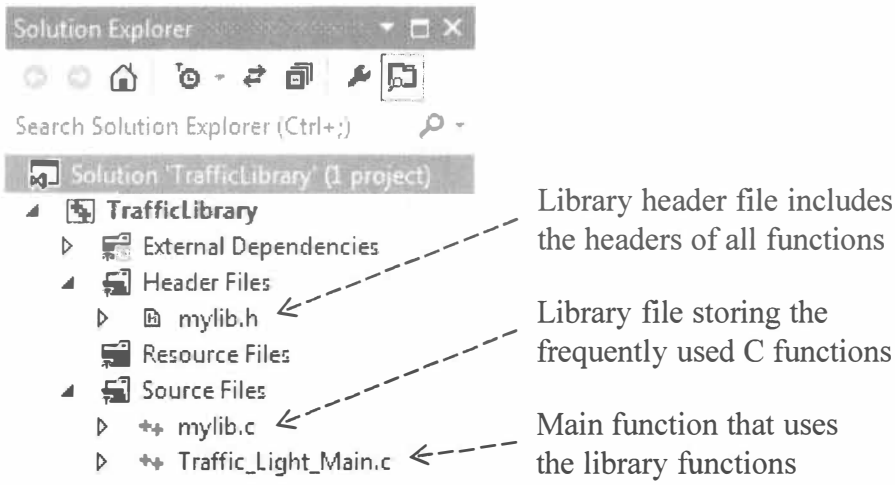


Figure 2.29. Organizing the modules and header files.

In object-oriented computing, better modularity is the main focus. We will discuss program design with multiple classes and multiple modules in more detail in Chapter 3.

2.10 Case Study: Putting All Together

In this section, we give an example that applies many of the data structures and programming techniques learned in this chapter, including array, string, enumeration type, pointer, pointer to pointer, linked list, global variable versus local variable, call-by-value and call-by-address parameter-passing mechanisms, memory management and garbage collection, and recursion. The memory management and garbage collection will be further discussed in the C++ chapter in more detail, where memory leak detection and detection tool will be introduced.

The first part of the example includes the declaration, forward declaration, the main function, and the branching function.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#pragma warning(disable: 4996) // comment out if not in Visual Studio
typedef enum { diploma = 0, bachelor, master, doctor } education;
// A struct for nodes of the linked list.
struct container {
  struct person *plink; // points to a struct person.
  struct container *next;
};
// A struct to hold attributes of a person
struct person {
  char name[32];
  char email[32];
  int phone;
```

```
  education degree;
};
void branching(char c);
char* get_name();
void print_list(struct container* root);
int insertion(struct container** ptrToHead); // note: pointer to pointer
struct container* search(struct container* root, char* sname);
void deleteOne(struct container** ptrToHead);
void deleteAll(struct container** ptrToHead);
void print_all(struct container* root);
/*****************************************************************/
int main(){
  // Declare head as a local variable of main function
  struct container* head = NULL;
  char ch = 'i';
  do {          // Print a menu for selection
      printf("Enter your selection\n");
      printf("\ti: insert a new entry\n");
      printf("\td: delete one entry\n");
      printf("\tr: delete all entries\n");
      printf("\ts: search an entry\n");
      printf("\tp: print all entries\n");
      printf("\tq: quit \n");
      fflush(stdin); // Flush the input buffer. Read section 2.6.3
      ch = tolower(getchar());   // Convert uppercase char to lowercase.
      branching(ch, &head);
      printf("\n");
  } while (ch != 113);            // 113 is 'q' in ASCII
  return 0;
};
/*****************************************************************/
// Branch to different tasks: insert a person, search for a person,
// delete a person, and print all added persons.
void branching(char c, struct container** ptrToHead){
  char *p;
  switch (c) {
  case 'i':
      insertion(ptrToHead);break;
  case 's':
      p = get_name();
      search(*ptrToHead, p);break;
  case 'd':
      deleteOne(ptrToHead);break;
  case 'r':
```

```
        deleteAll(ptrToHead);break;
    case 'p':
        print_all(*ptrToHead);break;
    case 'q':
        deleteAll(ptrToHead); // free all memory when quit
        break;
    default:
        printf("Invalid input\n");
    }
};
```

The relationship between the container struct and the person struct is illustrated in Figure 2.30. Notice that the head pointer is declared as a local variable in the main function, which is not visible in the other functions, and thus, we need to use parameter passing to access and to modify the head pointer. To read the head pointer, we can use call-by-value parameter passing, which are used in the functions search and printAll. In the functions deleteOne and deleteAll, we need to modify the head pointer, and we need to pass the address of head pointer into these functions, which is thus a pointer to a pointer.

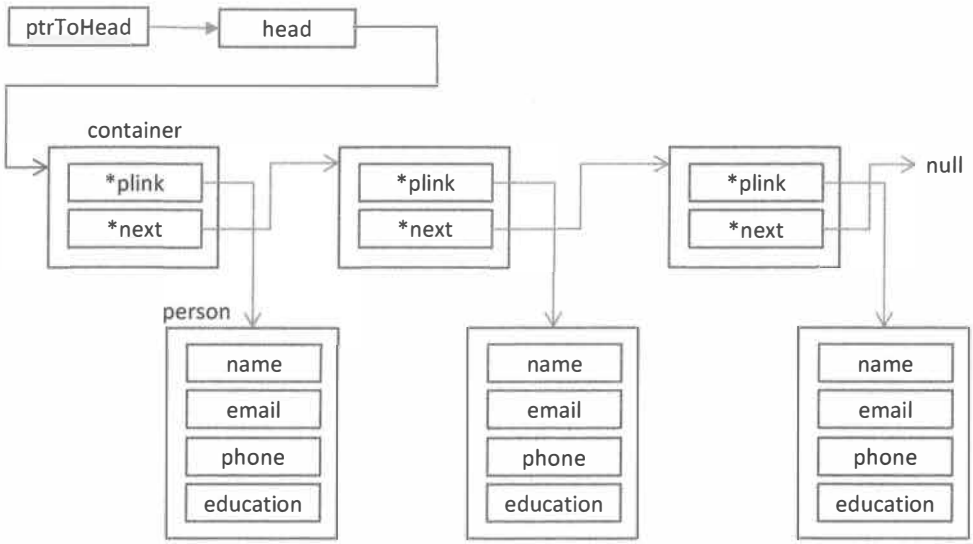


Figure 2.30. A linked list of containers, with pointers to a person node and to the next container.

In the following, we provide the remaining functions listed in the forward declaration part and discuss their implementation.

```
/*************************************************************/
// Delete the first person node in the linked list.
void deleteOne(struct container** ptrToHead) {
  int i = 0;
  struct container *toDelete = NULL;
  if (*ptrToHead == NULL) {
      printf("\nThe list is empty. Nothing was deleted.\n");
  }
  else if ((*ptrToHead)->next == NULL) {
      free((*ptrToHead)->plink);
```

```
        free(*ptrToHead);
        *ptrToHead = NULL;
    }
    else {
        toDelete = *ptrToHead;
        *ptrToHead = (*ptrToHead)->next;
        free(toDelete->plink);
        free(toDelete);
        toDelete = NULL;
    }
    printf("\nA container node was deleted.\n");
};
/*****************************************************************/
// Recursively delete the entire list given the head of a linked list.
void deleteAll(struct container** ptrToHead) {
    struct container* pnext;
    if (*ptrToHead == NULL)
        return;
    else {
        deleteOne(ptrToHead);
        deleteAll(ptrToHead);
    }
};
// Read the input from the user.
char * get_name() {
    char *p = (char *) malloc(32); // Use dynamic memory which does not go
out of scope
    printf("Please enter a name for the search: ");
    scanf("%s", p);
    return p;
};
/*****************************************************************/
// Inserts the person to the sorted place. Note: A < a, and A will be
ordered first.
int insertion(struct container** ptrToHead) {
    int i = 0;
    struct container* newNode = NULL, *iterator = NULL, *follower = NULL;
    struct person* newPerson = NULL;
    newNode = (struct container*) malloc(sizeof(struct container));
    // Case 1: The program is out of memory.
    if (newNode == NULL) {
        printf("Fatal Error: Out of Memory. Exiting now.");
        return 0;
    }
```

```
  // Case 2: The structure still has unfilled slots.
  else {
       newPerson = (struct person*) malloc(sizeof(struct person));
       if (newPerson == NULL) {
            printf("Fatal Error: Out of Memory. Exiting now.");
            return 0;
       }
       else {
            printf("Enter the name:\n");
            scanf("%s", newPerson->name);
            printf("Enter the phone number:\n");
            scanf("%d", &newPerson->phone, sizeof(newPerson->phone));
            printf("Enter the e-mail:.\n");
            scanf("%s", newPerson->email);
            do {
                printf("Enter the degree: select 0 for diploma, select 1
for bachelor, select 2 for master, or select 3 for doctor:\n");
                scanf("%d", &newPerson->degree);
                if (newPerson->degree< diploma || newPerson->degree >
doctor) {
                     printf("Please enter a value from 0 to 3.\n");
                }
            } while (newPerson->degree < diploma || newPerson->degree >
doctor);
            newNode->plink = newPerson;
            if (*ptrToHead == NULL) {
                *ptrToHead = newNode;
                (*ptrToHead)->next = NULL;
                return 0;
            }
            else {
                if (strcmp(newPerson->name,(*ptrToHead)->plink->name)<0){
                     newNode->next = *ptrToHead;
                     *ptrToHead = newNode;
                     return 0;
                }
                iterator = *ptrToHead;
                follower = iterator;
                while (iterator != NULL) {
                     if (strcmp(newPerson->name,iterator->plink->name)<0){
                         newNode->next = iterator;
                         follower->next = newNode;
                         return 0;
                     }
```

```
                follower = iterator;
                iterator = iterator->next;
            }
            follower->next = newNode;
            newNode->next = NULL;
            return 0;
        }
    }
  }
  return 0;
};
/*************************************************************/
// Print the name, e-mail, phone, and education level of each person.
// It calls the helper printFirst to recursively print the list
void print_all(struct container* root) {
  struct container* iterator = root;
  //Case 1: The structure is empty
  if (iterator == NULL) {
      printf("\nNo entries found.\n");
      return;
  }
  // Case 2: The structure has at least one item in it
  else{
      printFirst(root);
      return;
  }
};
void printFirst(struct container* root) {
  if (root != NULL){
      printf("\n\nname = %s\n", root->plink->name);
      printf("email = %s\n", root->plink->email);
      printf("phone = %d\n", root->plink->phone);
      switch (root->plink->degree) {
      case diploma:
           printf("degree = diploma\n");
           break;
      case bachelor:
           printf("degree = bachelor\n");
           break;
      case master:
           printf("degree = master\n");
           break;
      case doctor:
           printf("degree = doctor\n");
```

```
            break;
        default:
            printf("System Error: degree information corruption.\n");
            break;
        }
        printFirst(root->next);
    }
};
/*****************************************************************/
//Find a person by comparing names given the head of the linked list.
struct container* search(struct container* root, char* sname) {
    struct container* iterator = root;
    while (iterator != NULL) {
        if (strcmp(sname, iterator->plink->name) == 0) {
            printf("\n\nname = %s\n", iterator->plink->name);
            printf("email = %s\n", iterator->plink->email);
            printf("phone = %d\n", iterator->plink->phone);
            switch (iterator->plink->degree) {
            case diploma:
                printf("degree = diploma\n");
                break;
            case bachelor:
                printf("degree = bachelor\n");
                break;
            case master:
                printf("degree = master\n");
                break;
            case doctor:
                printf("degree = doctor\n");
                break;
            default:
                printf("System Error: degree information corruption.\n");
                break;
            }
            free(sname); // garbage collection
            return iterator;
        }
        iterator = iterator->next;
    }
    printf("The name does not exist.\n");
    free(sname); // garbage collection
    return iterator;
};
```

2.11 Summary

In this chapter, we started from basic issues in writing imperative C/C++ programs and went through important and advanced topics in the languages. The focus is on the topics that are significantly different from Java. The major topics we discussed are:

- Getting started with writing simple C/C++ programs;
- Control structures in C/C++ using syntax graphs;
- Relationships among memory locations, memory addresses, variable names, variable addresses, and the value stored in a variable;
- Pointers and pointer variables, referencing, and dereferencing;
- Array-based and pointer-based string operations;
- Three different ways of introducing constants: macro, const, and enumeration types;
- Structure types and compound data types;
- File type and file operations;
- Three major parameter-passing mechanisms: call-by-value, call by address, and call-by-alias;
- Recursive structures; and
- A brief introduction to modular design. More modular design will be discussed in the C++ chapter.

Table 2.5 summarizes the features supported by C and C++, as well as by Java. As can be seen from the table, C and C++ allow more flexibility, whereas Java is more restricted.

Feature	C and C++	Java
Macro	YES	NO
Inlining	YES	YES
Global variables	YES	NO
Static variables	YES	YES
Pointer	YES	NO
Value semantics for all types	YES	Primitive types
Reference semantics for all types	YES	Reference types
String type	char array	YES
Union type	YES	NO
Parameter passing: call-by-value	YES	Primitive types
Parameter passing: call-by-alias	C++ only	NO
Parameter passing: call-by-address	YES	Reference types
Recursive call and application of the fantastic-four abstract approach	YES	YES

Table 2.5. Feature comparison between C/C++ and Java.

2.12 Homework, programming exercises, and projects

1. Multiple Choice. Choose only one answer for each question. Choose the best answer if more than one answer is acceptable.

1.1 Forward declaration in modern programming practice

- ❑ provides a level of abstraction.
- ❑ is never necessary.
- ❑ is not required if `<iostream>` is included.
- ❑ is useless.

1.2 C language does not have a Boolean type because

- ❑ C is not designed to handle logic operations.
- ❑ C uses strong type checking.
- ❑ Boolean values can be represented as integers.
- ❑ C++ has already defined a Boolean type.

1.3 Two functions are said to be mutually recursive if

- ❑ one function is defined within the other function.
- ❑ they call each other.
- ❑ each function calls itself.
- ❑ they are independent of each other.

1.4 Assume that a string is declared as `char str[] = "alpha"`, what is the return value of `sizeof (str)`?

❑ 1 ❑ 5 ❑ 6 ❑ 7 ❑ 40

1.5 Assume that two pointers are declared as: `char *str1 = "alpha", *str2;`
Which assignment statement below will lead to a semantic error?

❑ `str2 = str1;` ❑ `str2 = 0;` ❑ `str1 = str1+1;` ❑ `*str2 = "Hi";`

1.6 Which of the following declarations will cause a compilation error?

- ❑ `char s[5];`
- ❑ `char s[3] = "hello";`
- ❑ `char s[];`
- ❑ `char s[] = {'s', 't', 'r'};`

1.7 Given a declaration: int i = 25, *j = &i, **k = &j;
which of the following operations will change the value of variable i?

❑ `j++;` ❑ `k++;` ❑ `(*k)++;` ❑ `(**k)++;`

1.8 Given a declaration: int i = 25, *j = &i, **k = &j;
which of the following operations will cause a compilation error?

❑ `i++;` ❑ `(&i)++;` ❑ `(*j)++;` ❑ `(**k)++;`

1.9 What is the maximum number of padding bytes that a compiler can add to a structure?

❑ `1` ❑ `2` ❑ `3` ❑ `more than 3`

1.10 The enumeration type of values are stored in the memory as

❑ bool ❑ double ❑ int ❑ string

1.11 If we want to store a linked list of structures, with each structure containing different types of data, into a disk file, what file type should we choose?

❑ array file ❑ binary file ❑ text file ❑ structure file

1.12 The reason that we need to call fflush() or cin.ignore() is because the previous

❑ output leaves a character in the file buffer. ❑ output fails to complete its operation.

❑ input leaves a character in the file buffer. ❑ input fails to complete its operation.

1.13 Assume the following structure is defined in a 32-bit programming environment.

```
struct myNode {
    char     name[30];
    char     location[32];
    struct   myNode* next;
} x, *y;
```

what is the size of x?

❑ 4 bytes ❑ 66 bytes ❑ 68 bytes ❑ 72 bytes

what is the size of x?

❑ 4 bytes ❑ 66 bytes ❑ 68 bytes ❑ 72 bytes

1.14 What parameter-passing mechanism cannot change the variable values in the caller?

❑ call-by-value ❑ call-by-alias ❑ call-by-address ❑ None of them

1.15 What parameter-passing mechanism requires the actual parameter to be a variable?

❑ call-by-value ❑ call-by-alias ❑ call-by-address ❑ None of them

1.16 Given the forward declaration: void foo(char c, int &n); what parameter passing mechanisms are used? Select all that apply.

❑ call-by-value ❑ call-by-alias ❑ call-by-address ❑ None of them

1.17 What type of recursive function is structurally equivalent to a while-loop?

❑ head-recursion ❑ middle-recursion ❑ tail-recursion ❑ mutual recursion

The Ackermann function is defined recursively for two nonnegative integers k and n as follows. Answer the following three questions based on the function and the fantastic-four abstract approach.

$$A(s, t) = \begin{cases} t+1, & if\ s = 0 \\ A(s-1, 1), & if\ s > 0\ and\ t = 0 \\ A\big(s-1, A(s, t-1)\big), & if\ s > 0\ and\ t > 0 \end{cases}$$

1.18 What is the size-n problem?

❑ (s, t) ❑ A(s, t) ❑ A(n) ❑ n

1.19 What is the stopping condition and return value at the stopping condition?

❑ s = 0 and t+1 ❑ s = 0 and t = 1 ❑ s > 0 and t = 0 ❑ s > 0 and t > 0

1.20 What is the size-m problem that can be used for calculating the size n problem? Select all that apply.

❑ A(s, t) ❑ A(s-1, 1) ❑ A(s, t-1) ❑ A(s-1, A(s, t-1))

1.21 The data stored in a binary search tree is sorted, if the tree is traversed in

❑ preorder ❑ postorder ❑ inorder ❑ in any order

1.22 Consider an array, a linked list, doubly linked list, and a binary search tree. Which data structure requires fewest comparisons in average to search an element stored in the data structure?

❑ binary search tree ❑ array ❑ doubly linked list ❑ linked list

2. What is a byte and what is a word in memory? What is the name of a variable? What is the address of a memory location? What is the content of a memory location?

3. What is the difference between a memory location and a register? How do we access a memory location and a register?

4. A variable has several aspects (name, address, value, location), and different aspects are used in different places.

4.1 If a variable is used on the left-hand side of an assignment statement, which aspect is actually used?

4.2 If a variable is used on the right-hand side of an assignment statement, which aspect is actually used?

4.3 If we apply the address-of operator "`&`" to the variable (i.e., `&v`), which aspect is returned?

5. Given a piece of C code

```
1   #include <stdio.h>
2   void main() {
3         char str[] = "hello", *p;
4         p = str;
5         while (*p != '\0')
6              (*(p++))++;
7         printf("str = %s, p = %s\n", str, p);
8   }
```

5.1 What is the exact output of the printf statement?

5.2 At line 3, if we replace `char str[] = "hello"` by `char *str = "hello"`, it will cause

❑ compilation error. ❑ runtime error. ❑ no error at all. ❑ incorrect output.

5.3 At line 3, if we replace `*p;` with `char *p = str;` it will cause

❑ compilation error. ❑ runtime error. ❑ no error at all. ❑ incorrect output.

5.4 In lines 5 and 6, the string is accessed using a pointer and pointer operations. Rewrite the program from line 3 to line 6 so that only array operations are used to access the string.

6. What are the three different methods of defining constants in C/C++? What are the differences of the constants defined in these methods?

6.1 Can a constant defined by `const` ever be modified? If yes, how and why? If no, why?

6.2 Can a constant defined by `#define` ever be modified? If yes, how and why? If no, why?

6.3 What are the advantages of defining an enumeration type instead of using an integer type directly?

7. What is the difference between a structure type and a union type? In what circumstances are union types useful?

8. Parameter passing

8.1 What is a formal parameter and what is an actual parameter?

8.2 What is the difference between call-by-value and call-by-alias?

8.3 Where do you need to use call-by-value and where do you need to use call-by-alias?

8.4 How do you use call-by-value and call-by-alias?

9. Structure type

9.1 How do you define a structure type? How do you declare a variable of a structure type? How do you declare a pointer to a structure type variable?

9.2 How do you obtain memory statically for a structure type variable? How do you create dynamic memory and link it to a pointer?

9.3 How do you use the name of a structure and a pointer to a structure to access the fields in the structure?

10. Programming exercise. This question gives you practice in using declarations, forward declarations and scopes of functions and variables, and type checking in C and C++.

Given the C program below, answer the following questions.

```
// This program shows function and variable declarations and their
scopes.
#include <stdio.h>
int keven = 0, kodd = 0;
long evennumber(short);
long oddnumber(short);
int even(int);
int evennumber(int a) {          // genuine declaration
    if (a == 2) {
        printf("keven = %d, kodd = %d\n", keven, kodd);
        return keven;
    }
    else {
        a = (int)a/2;
        if (even(a)) {    // Is an even?
            keven++;
            return evennumber(a);
        }
```

```
            else {
                kodd++;
                return oddnumber(a);
            }
        }
        // return a;
}
int oddnumber(int b) {      // genuine declaration
        if (b == 1) {
            printf("keven = %d, kodd = %d\n", keven, kodd);
            return kodd;
        }
        else {
            b = 3*b+1;
            if (!even(b)) {   // Is b odd?
                kodd++;
                return oddnumber(b);
            }
            else {
                keven++;
                return evennumber(b);
            }
        }
        // return b;
}
int even(int x) {       // % is modulo operator.
        return ( (x%2 == 0) ? 1 : 0);
}
void main() {
        register short r = 0;       // a register type variable is faster,
        int i = r;                  // it is often used for loop variable
        float f;
        for (r = 0; r < 3; r++) {
            printf("Please enter an integer number that is >= 2\n");
            scanf("%d", &i);
            if (even(i))
                f = evennumber(i);
            else
                f = oddnumber(i);
        }
}
```

10.1 Save the file as `declaration.c`. (consider the program to be a C program). Choose the commands under the menu "`Build`":

```
Compile declaration.c
Build declaration.exe
Execute declaration.exe
```

What errors or warning messages are displayed?

10.2 Save the file as `declaration.cpp`. Repeat question 10.1.

10.3 Analyze the type requirement of functions and variables in the given program. Make minimum changes to `declaration.cpp` to remove all compilation errors and warnings.

10.4 Explain global variables and local variables. List the global variables and local variables in the program. The parameters of a function are local variables too.

10.5 Can we swap the order of the two variable declarations: "`register short r = 0;`" and "`int i = r;`"? Explain your answer according to C/C++'s scope rule.

10.6 Explain the forward declaration. If the forward declarations in the program were removed, what would happen?

10.7 Explain type casting and type coercion. List all type castings and type coercions used in the program.

10.8 According to the analysis above and the definition of strong type checking, are C and C++ strongly typed? Which language's typing system is stronger, C or C++?

10.9 Program correctness/reliability issue. A correct program must terminate for all valid inputs. The given program has been tested by many people. It has always terminated for the inputs used. However, nobody so far can prove that this program can terminate for any integer input. Thus, this program is often used as an example of improperly designed loop structure or recursive function. A good programming practice in writing loop or recursive function is to guarantee that the loop variable or the size-related-parameter (they control the number of iterations) is defined on an enumerable set (e.g., integer), has a lower bound (e.g., 0), and decreases strictly (e.g., 9, 6, 5, 3, 2, 1). Add a print-statement in functions `evennumber` and `oddnumber` to print the size-related parameter value and use input values i = 3, 4, and 7, respectively, to test the program. Give the three sequences of values printed by the added print-statements.

10.10 Compare questions 10 with homework question 17 in Chapter 1 and explain why it is difficult to prove that the program can terminate for any integer input.

11. The following program will open and read an existing text file called `file1.txt`, add a number between 1 and 25 to each and every character, and then write the modified text into a new file called `file2.txt`. Read this program carefully and answer the questions following the program.

```
// Filename: encryption0.c
#include <stdio.h>
#include <string.h>

// Read all characters in the file and put them in a string str
void file_read(char *filename, char *str) {
```

```
    FILE *p;        // p is declared as a pointer to the FILE type.
    int index=0;
    p=fopen(filename, "r");     // Open the file for "read".
                      // Other options incl. "w" (write) and "a"
(append)
    while(!feof(p))         // while not reaching the end-of-file
character
        *(str+index++)=fgetc(p); //read a character from file and put
it
                          // in str. Then p is increased automatically.
    str[index-1]='\0';          // add the string terminator
    puts(str);                  // print str. You can use printf too.
    fclose(p);                  // close the file
}
void encrypt(int offset, char *str) {
    int i,l;
    l=strlen(str);
    printf("unencrypted str = \n%s\n", str);
    for(i=0;i<l;i++)
        str[i] = str[i]+offset;
    printf("encrypted str = \n%s \nlength = %d\n", str, l);
}
void file_write(char *filename, char *str) {
    int i, l;
    FILE *p;
    p=fopen(filename, "w");         // open the file for "write".
    l = strlen(str);                // string-length
    for(i=0;i<l;i++)
        fputc(*(str+i),p);          // write a character in the file
pointed
                                    // by p. p is increased automatically
    fclose(p);                      // close the file
}
void main(void) {
    char filename[25];
    char string[1024];
    strcpy(filename, "file1.txt");
    file_read(filename, string);
    encrypt(7, string);
    strcpy(filename, "file2.txt");
    file_write(filename, string);
}
```

11.1 Enter the following text in a text file named `file1.txt`.

```
Politician A said "Politician B is a liar, because he promised in last
year's election that he would give every homeless person a home".
```

```
Politician B said "Politician A is a liar, because he promised in last
year's election that he would give every jobless person a job". Are these
functions mutually recursive?
```

Enter the following text in a text file named `file4.txt`.

```
Politician B said "Politician A is a liar, because he promised in last
year's election that he would give every homeless person a home".
Politician A said "Politician B is a liar, because he promised in last
year's election that he would give every jobless person a job". These
quotations are not mutually recursive.
```

Use `file1.txt` file and the `key = 7` as the test case for the program `encrytion0.c`. Load the program into Visual C++ and execute the program. The program should generate a file called `file2.txt`.

Hint: `file1.txt` must be in the same directory as the program.

11.2 Rewrite the `void encrypt(int offset, char *str)` function in the given program using pointer operations to replace array operations, for example, replace `str[i]` with `*(str+i)`. Replace the for-loop with a while-loop, where you must use a terminator to detect the end of the string. Comment the code where changes have been made.

11.3 Write a function called `int difference(char *filename1, char *filename2)` that compares two files. The file's names must be passed to the parameters `filename1` and `filename2`, respectively. The program should return the number of characters that are different (mismatches). For example, if the two files are exactly the same, the function returns 0. If the program detects 10 mismatches, it returns 10. If one file is longer than the other, the extra characters count as differences. This function must use string and pointer operations and compare each character one after another. You may not use the library function for string comparison. Write at least two lines of comment at the beginning of the function, describing what this function does and how it is implemented.

11.4 Write a function `void faultinjection(char *filename1, char *filename2, int n)` that injects `n` character faults into the file specified by the parameter `filename1`. Each character fault is a modification to a character by adding a random number between −10 and 10. The positions of the `n` faults are chosen randomly between the first character and the last character in the file. The modified file is stored in the file specified by the parameter `filename2`.

Note: To generate a random number, you can call a library function, for example, `rand()`. For a simple example, the following code prints 20 pseudo random numbers between 0 and 99. The function `srand(unsigned)` seeds the random-number generator with the current time so that the numbers generated by `rand()` will be different every time you call it.

```
#define size 100
#include <stdio.h>
#include <stdlib.h>    // function rand() is defined in this library
#include <time.h>      // function time(NULL)is defined in this library
main() {
    int i, rdm;
    srand((unsigned)time(NULL));  // Use current time as seed
    for (i = 0; i<20; i++) {
        rdm = rand() % size;      // modulo operation
        printf("random = %d\n", rdm);
```

```
    }
}
```

11.5 Rewrite the `main()` function to perform the following operations described in the pseudo code.

```
encrypt file1.txt into file2.txt
decrypt file2.txt into file3.txt  // use a negative key
Find the differences between file1.txt and file3.txt
Find the differences between file1.txt and file4.txt
Call faultinjection (file4.txt, file5.txt, n); // choose n = 5
Find the differences between file4.txt and file5.txt
```

12. The following program generates maps representing mazes, where blank (space) characters represent open rooms through which a path may pass, while "X" characters represent closed rooms that cannot be used on any path. The starting position is marked by a character "S" and the goal position is marked by a character "G." For a given maze, one can write a program to check if there is path from "S" to "G," and print the paths if they exist. In this exercise, we generate only the mazes and do not attempt to write a program to find the paths. You are given the following C code, try to understand what it does and make the changes given in the following questions.

```
// This program exercises the operations on multidimensional array
#include <stdio.h>
#pragma warning(disable: 4996) // comment out if not in Visual Studio
#define maxrow 50
#define maxcolumn 50
char maze[maxrow][maxcolumn]; // Define a static array of arrays of
characters.
int lastrow = 0;
// Forward Declarations
int triple(int);
void initialization(int, int);
void randommaze(int, int);
void printmaze(int, int);

int triple(int x) {        // % is modulo operator.
    return ((x % 3 == 0) ? 1 : 0);
}
void initialization(int r, int c) {
    int i, j;
    for (i = 0; i < r; i++){
        maze[i][0] = 'X';     // add border
        maze[i][c - 1] = 'X'; // add border
        maze[i][c] = '\0';    // add string terminator
        for (j = 1; j < c - 1; j++)
        {
            if ((i == 0) || (i == r - 1))
                maze[i][j] = 'X'; // add border
            else
```

```
                maze[i][j] = ' '; // initialize with space
            }
        }
    }
// Add 'X' into the maze at random positions
void randommaze(int r, int c) {
    int i, j, d;
    for (i = 1; i < r - 1; i++) {
        for (j = 1; j < c - 2; j++) {
            d = rand();
            if (triple(d))
            {
                maze[i][j] = 'X';
            }
        }
    }
    i = rand() % (r - 2) + 1;
    j = rand() % (c - 3) + 1;
    maze[i][j] = 'S'; // define Starting point
    do {
        i = rand() % (r - 2) + 1;
        j = rand() % (c - 3) + 1;
    } while (maze[i][j] == 'S');
    maze[i][j] = 'G'; // define Goal point
}

// Print the maze
void printmaze(int r, int c) {
    int i, j;
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++)
            printf("%c", maze[i][j]);
        printf("\n");
    }
}
void main() {
    int row, column;
    printf("Please enter two integers, which must be greater than 3 and
less than maxrow and maxcolomn, respectively\n");
    scanf("%d\n%d", &row, &column);
    while ((row <= 3) || (column <= 3) || (row >= maxrow) || (column >=
maxcolumn)) {
        printf("both integers must be greater than 3. Row must be less
than %d, and column less than %d. Please reenter\n", maxrow, maxcolumn);
```

```
        scanf("%d\n%d", &row, &column);
    }
    initialization(row, column);
    randommaze(row, column);
    printmaze(row, column);
    //encryptmaze(row, column);
    //printmaze(row, column);
    //decryptmaze(row, column);
    //printmaze(row, column);
}
```

The program above can be written using pointer operations, instead of using array indices, The code below shows the pointer version of the initialization function.

```
void initialization(int r, int c) {
    int i, j;
    char *p = 0;
    for (i = 0; i <= r; i++){
        p = &maze[i][0];
        // Tt points to initial address of the ith row of the maze
        *p = 'X';         // add left border
        *(p + c - 1) = 'X';   // add right border
        *(p + c) = '\0'; // add string terminator
        for (j = 1; j < c - 1; j++){
            if ((i == 0) || (i == r - 1))
                *(p + j) = 'X'; // add top and bottom borders
            else
                *(p + j) = ' '; // initialize inner maze with space
        }
    }
}
```

Now, you can follow the example to complete the following exercise questions.

12.1 Rewrite the function int triple(int) using macro definition.

12.2 Rewrite the function randommaze(row, column) by substituting pointer operations for all array operations. You may not use indexed operation like maze[i][j], except getting the initial value of the pointer.

12.3 Rewrite the function printmaze(row, column) by substituting string operations for all character operations.

12.4 Write the function encryptmaze(row, column) based on the pointer operations. The function will encrypt the maze in the following secret rules:

- An integer i will be added to each space character, where i is the row number of the character.
- An integer j will be added to each non-space character (X, S, and G), where j is the column number of the character. Do not encrypt the terminator character '\0'.

12.5 Write the function decryptmaze(row, column) to decrypt the maze.

12.6 Call encryptmaze(row, column) and decryptmaze(row, column) functions in the main() function by removing the comment marks, and call the printmaze(row, column) after encryption and after decryption.

13. You are given the following program. Save the given program under the name `contactbook.c`. The program takes a **command line parameter**: the database's name in which the contact records are to be saved, assuming the database name is `person.dbms`.

You can pass command line parameters within the Visual Studio environment as follows:

(1) Use Visual Studio to compile and build the program `contactbook.c`.
(2) Choose menu "`Project`" and "`Properties...`".
(3) Under the item "`Configuration Properties,`" click on "`Debugging,`" and enter the file name `person.dbms` to the right of the field "`Command Arguments.`"
(4) Click OK to return.
(5) Now you can execute the program.

II. You can also pass command line parameters using the following method:

(1) Compile and build the program.
(2) Choose MS Windows "`Start`" Menu.
(3) Choose "`Run ...`".
(4) Click on "`Browse ...`".
(5) Browse to the folder where your `contactbook.c` is stored.
(6) Go into the folder "`Debug.`" (This folder should have been created by the compile and build commands in step 1.)
(7) Choose the executable program called `contactbook` and then click "`open.`"
(8) You should see the path "`...\Debug\contactbook.exe.`"
(9) Append the file name `person.dbms` to the end of the path, with a space in between. The entire command sequence should look like: "`...\Debug\ contactbook.exe`" `person.dbms`.
(10) Click "`OK.`" The program should start to run.

The tasks of this assignment are as follows.

13.1 Read the program carefully and make sure you understand the program and each function in the program. Then add at least two lines of comments below each function's forward declaration to explain what the function does.

13.2 Write a function called `sort()`. The function should sort the existing linked list by the name field in dictionary order. Use the simplest sorting algorithm. For example, the selection sort: find the name with the smallest dictionary value and place it in the first place in the linked list, and then find the name with the next smallest dictionary value and put it in the second place, and so on.

13.3 Add your `sort()` function into the program and modify the program to offer users an extra option "sort" in the menu.

13.4 Test each function of the program: insert, delete, search, and sort. You can quit the program and restart the program. The records stored in the linked list should be saved and reloaded into the list. To copy the output in Visual Studio: Highlight the text you want to copy, click on the small icon

"c:\" at the top-left corner of the output window. Choose Edit-Copy. Then you can paste the output into a text file.

```
// Manipulation of files and singly linked list
// Command line parameter inputs
#include <stdio.h>
#include <stdlib.h>    // malloc is defined in this library
#include <string.h>    // string operations are defined in this library
struct contact {
  char name[30];
  char phone[20];
  char email[30];
  struct contact *next;
}*head = NULL;
char *file_name;
// forward declaration
void menu();
void branching(char c);
struct contact* find_node(char *str, int *position);
void display_node(struct contact *node, int index);
int insert();
int deletion();
int modify();
int search();
void display_all();
void load_file();
void save_file();
int main(int argc, char *argv[]) {
     char ch;
     if(argc != 2) {  // Two command line parameters required
          printf("Command Line Parameters Required !\n");
          printf("Try again......\n");
          getchar();  // enter any character to return
          return -1;
     }
     printf("SINGLY LINKED LIST\n");
     printf("******************");
     file_name = argv[1];
     load_file();

     do {
          menu();
          fflush(stdin);              // Flush the standard input buffer
          ch = tolower(getchar()); // read a char, convert to lower case
          branching(ch);
```

```
    } while (ch != 'q');
    return 0;
}
void menu() {
    printf("\n\nMENU\n");
    printf("----\n");
    printf("i: Insert a new entry.\n");
    printf("d: Delete an entry.\n");
    printf("m: Modify an entry.\n");
    printf("s: Search for an entry.\n");
    printf("p: Print all entries.\n");
    printf("q: Quit the program.\n");
    printf("Please enter your choice (i, d, m, s, p, or q) --> ");
}
void branching(char c) {
    switch(c) {
        case 'i':   if(insert() != 0)
                        printf("INSERTION OPERATION FAILED.\n");
                    else
                        printf("INSERTED NODE IN THE LIST
SUCCESSFULLY.\n");
                    break;
        case 'd':   if(deletion() != 0)
                        printf("DELETION OPERATION FAILED.\n");
                    else
                        printf("DELETED THE ABOVE NODE SUCCESSFULLY.\n");
                    break;
        case 'm':   if(modify() != 0)
                        printf("MODIFY OPERATION FAILED.\n");
                    else
                        printf("MODIFIED THE ABOVE NODE SUCCESSFULLY.\n");
                    break;
        case 's':   if(search() != 0)
                        printf("SEARCH FAILED.\n");
                    else
                        printf("SEARCH FOR THE NODE SUCCESSFUL.\n");
                    break;
        case 'p':   display_all();
                    break;
        case 'q':   save_file();
                    break;
        default: printf("ERROR - Invalid input.\n");
                    printf("Try again.....\n");
                    break;
```

```
    }
    return;
}
int insert() {
    struct contact *node;
    char sname[30];
    int index = 1;
    printf("\nInsertion module.....\n");
    printf("Enter the name of the person to be inserted: ");
    scanf("%s", sname);
    node = find_node(sname, &index); // find duplicates
    if(node != NULL) {
        printf("ERROR - Duplicate entry not allowed.\n");
        printf("A entry is found in the list at index %d.\n", index);
        display_node(node, index);
        return -1;
    }
    else {
        node = (struct contact*) malloc(sizeof(struct contact));
        if (node == NULL) {
           printf("ERROR - Could not allocate memory !\n");
           return -1;
        }
        strcpy(node->name, sname);
        printf("Enter his telephone number: ");
        scanf("%s", node->phone);
        printf("Enter his email address: ");
        scanf("%s", node->email);
        node->next = head;
        head = node;
        return 0;
    }
}
int deletion() {
    char sname[30];
    struct contact *temp, *prev;
    int index = 1;
    printf("\nDeletion module.....\n");
    printf("Please enter the name of the person to be deleted: ");
    scanf("%s", sname);
    temp = head;
    while (temp != NULL) // search for the node to be deleted
        if (stricmp(sname, temp->name) != 0) { //case insensitive
strcmp
```

```
            prev = temp;
            temp = temp->next;
            index++;
        }
        else {
            printf("Person to be deleted is found at index %d.",
index);
            display_node(temp, index);
            if(temp != head) prev->next = temp->next;
            else head = head->next;
            free(temp);  // Garbage collection
            return 0;
        }
    printf("The person with name '%s' does not exist.\n", sname);
    return -1;
}
int modify() {
    struct contact *node;
    char sname[30];
    int index = 1;
    printf("\nModification module.............\n");
    printf("Enter the name whose record is to be modified in the\n");
    printf("database: ");
    scanf("%s", sname);
    node = find_node(sname, &index);
    if(node != NULL) {
        printf("Person to be modified is found at index %d.", index);
        display_node(node, index);
        printf("\nEnter the new telephone number of this person: ");
        scanf("%s", node->phone);
        printf("Enter the new email address of this person: ");
        scanf("%s", node->email);
        return 0;
    }
    else {
        printf("The person with name '%s' does not exist \n", sname);
        printf("database.\n");
        return -1;
    }
}
int search() {
    struct contact *node;
    char sname[30];
    int index = 1;
```

```
     printf("\nSearch module.............\n");
     printf("Please enter the name to be searched in the database: ");
     scanf("%s", sname);
     node = find_node(sname, &index);
     if(node != NULL) {
          printf("Person searched is found at index %d.", index);
          display_node(node, index);
          return 0;
     }
     else {
          printf("The person '%s' does not exist.\n", sname);
          return -1;
     }
}
void display_all() {
    struct contact *node;
    int counter = 0;
    printf("\nDisplay module.............");
    node = head;
    while(node != NULL) {
         display_node(node, ++counter);
         node = node->next;
    }
    printf("\nNo more records.\n");
}
void load_file() {
    FILE *file_descriptor;
    struct contact *node, *temp;
    char str[30];
    file_descriptor = fopen(file_name, "rb"); // "b" for binary mode
    if(file_descriptor != NULL) {
         while(fread(str, 30, 1, file_descriptor) == 1) {
              node = (struct contact*) malloc(sizeof(struct contact));
              strcpy(node->name, str);
              fread(node->phone, 20, 1, file_descriptor);
              fread(node->email, 30, 1, file_descriptor);
              if(head != NULL) temp->next = node;
              else head = node;
              node->next = NULL;
              temp = node;
         }
         fclose(file_descriptor);
    }
}
```

```
void save_file() {
    FILE *file_descriptor;
    struct contact *node;
    file_descriptor = fopen(file_name, "w");
    if(file_descriptor != NULL) {
        node = head;
        while(node != NULL) {
            fwrite(node->name, 30, 1, file_descriptor);
            fwrite(node->phone, 20, 1, file_descriptor);
            fwrite(node->email, 30, 1, file_descriptor);
            node = node->next;
        }
    }
    else {
        printf("\nERROR - Could not open file for saving data !\n");
        getchar();
        exit(-1);
    }
}
struct contact* find_node(char *str, int *position) {
    struct contact *temp = head;
    while (temp != NULL) {
        if (stricmp(str, temp->name) != 0) {
            temp = temp->next;
            (*position)++;
        }
        else return temp;
    }
    return NULL;
}
void display_node(struct contact *node, int index) {
    printf("\nRECORD %d:\n", index);
    printf("\t\tName:\t\t%s\n", node->name);
    printf("\t\tTelephone:\t%s\n", node->phone);
    printf("\t\tEmail Address:\t%s\n", node->email);
}
```

14. Follow the fantastic-four abstract approach to write a recursive function to find the largest number in a given array of integers.

15. In Section 2.7, an array is sorted by a simple recursive function. In step 3 of the fantastic-four abstract approach, the `m` is selected to be `n - 1`. Rewrite the sorting program, but select `m` to be `n/2` (merge sort). You can assume that the initial size `n` is a power of 2 to simplify the problem.

16. Fibonacci numbers are defined by

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n \geq 2 \end{cases}$$

Follow the fantastic-four abstract approach to implement the function in C.

16.1 Define the size-n problem.

16.2 Define the stopping conditions and the return values.

16.3 Define the size-m problem.

16.4 Explain how you construct the size-n solution from the size-m solutions.

16.5 Implement the function in C that can be used to compute Fibonacci numbers for the integer $n \geq 0$.

16.6 Write a main program that takes the input of n from the keyboard; call the recursive function, and then print the result.

17. The Ackermann function is defined recursively for two nonnegative integers s and t as follows:

$$A(s, t) = \begin{cases} t + 1, & if\ s = 0 \\ A(s-1, 1), & if\ s > 0\ and\ t = 0 \\ A(s-1, A(s, t-1)), & if\ s > 0\ and\ t > 0 \end{cases}$$

17.1 Follow the fantastic-four abstract approach to implement the function in C. The function should take two integer numbers, m and n, and return the value of A(s, t), which is a long integer. Notice that the Ackerman function is a very rapidly growing function. Even values of 4 for m and n will yield an extremely large number, and thus using a long integer as the return value is necessary.

17.2 Write a main program that takes inputs of m and n from the keyboard; call the recursive function, and then print the result.

Chapter 3

The Object-Oriented Programming Language, C++

We discussed C/C++ as an imperative programming language in Chapter 2. Although we mainly discussed C, we used the notion of C/C++. The reason is that C is part of C++, or C is the imperative part of C++. In this chapter, we study C++ as an object-oriented programming language, or we will focus on the object-oriented features of the language.

The main idea of object-oriented programming is to use abstract and extendable data types as the building blocks of programs. The principle behind the object-oriented paradigm consists of a number of programming concepts, including the following:

- **Abstract data type (class):** Encapsulates related information in a class. A class (type) is used to declare objects (variables) that consist of data and operations (functions). Access to data members can be accurately controlled through operations defined in the data type.
- **Inheritance:** Derives a new class (derived class) based on an existing class (base class). Inheritance allows us to reuse and extend the data structures and functions that we or others have defined.
- **Class hierarchy:** Links all related classes together using inheritance.
- **Late binding and polymorphism:** Support virtual functions that use late binding. Allow pointers to objects in the class hierarchy to move downward, and allow the same function call to bind to different implementations of the function.

In this chapter, we will describe the object-oriented features of C++. At the end of the chapter, you should:

- have a good understanding of the object-oriented programming paradigm;
- be able to define classes with data members (also called member variables) and member functions (also called methods) in C++;
- understand the differences and similarities between data types and classes;
- understand the memory allocation mechanism in programming languages and the three memory allocation mechanisms: static, stack, and dynamic memory;
- be able to apply major object-oriented concepts in program design, including inheritance, class hierarchy, polymorphism, dynamic memory allocation, and late binding.

The chapter is organized as follows. In Section 3.1, a complete example of a C++ program is presented. The program will be used and explained in the following sections. In Section 3.2, the composition and definition of classes are discussed. The static and dynamic memory allocation and deallocation are studied in Section 3.3. Section 3.4 discusses the main features of the object-oriented programming paradigm,

including class inheritance, inheritance-based class hierarchy, polymorphic pointer, virtual functions, and late binding. In Section 3.5, C++ exceptions and exception handling are discussed.

3.1 A long program example: a queue and a priority queue written in C++

In this section, we present a complete C++ program that illustrates various features that we will discuss in the following sections:

- class definition;
- the in-class implementation of member functions and the out-class implementation of member functions using scope resolution operators;
- constructor and destructor;
- overloading of member functions;
- creation of stack and heap objects;
- access of class members using dot operator and pointer-to-member operators.

The following program is only explained through comments embedded in the program in this section. Different parts of the program will be further explained and studied in the following sections.

```
#include <iostream>
using namespace std;
class Queue {    // Class definition
private:         // private members can only be accessed in the class
  int queue_size;
protected:       // protected members can be accessed in derived classes
  int *buffer;   // pointer to first element of array
  int front;     // used for removing an element from the queue
  int rear;      // used for adding an element into the queue
  static int counter;    // It must be initialized outside the class
public:          // public members can be accessed by all functions
  Queue(void) { // constructor with no parameters
      front = 0;
      rear = 0;
      queue_size = 10;
      buffer = new int[queue_size]; // create a heap object of array
      if (buffer != NULL) counter++;     // counting objects
  }
  Queue(int n) {         // overloaded constructor with one parameter
      front = 0;
      rear = 0;
      queue_size = n;
      buffer = new int[queue_size]; // create a heap object of array
      if (buffer != NULL) counter++;     // counting objects
  }
  virtual ~Queue(void) {          // destructor
      delete buffer; // You must use "delete [] buffer;" if buffer points
```

```
        buffer = NULL; // to an array of objects, instead of integers.
        counter--;     // decrement the # of objects
    }
    void enqueue(int v) { // add an element at the end of the queue
        if (rear < queue_size)
             buffer[rear++] = v;
        else
             if (compact())
                 buffer[rear++] = v;
    }
    int dequeue(void){ // return and remove the 1st element from the queue
        if (front < rear)
             return buffer[front++];
        else {cout<< "Error: Queue empty"<<endl; return -1;}
    }
private:
    bool compact(void);              // implementation outside class
};
int Queue::counter = 0;
// static class member of Queue is initialized here outside class
// End of the class definition

class PriQueue : public Queue {   //PriQueue is derived from Queue
public:
    int getMax(void); // return and remove the max value from priority queue
    PriQueue(int n) : Queue(n) { };
    // base class constructor may not be inherited. It has to be explicitly
    // called. PriQueue's constructor simply calls Queue's constructor;
    ~PriQueue() {      // base class destructor may not be called or
        delete buffer;    // inherited. We must explicitly use delete.
        buffer = NULL;
        counter--;        // decrement the counter of objects
    }
};

bool Queue::compact(void){ // using scope resolution operator
    if (front == 0) {
        cout<<"Error: Queue overflow"<<endl;
        return false;
    }
    else {
        for (int i=0;i<rear-front;i++)
             buffer[i]=buffer[i+front];
        rear = rear - front;
```

```
        front = 0;
        return true;
    }
}
int PriQueue::getMax(void) { // get & remove max value from priority queue
    int i, max, imax;
    if (front < rear) {
        max = buffer[front];
        imax = front; // imax holds the index of current max value
        for (i=front;i<rear;i++){
            if (max < buffer[i]) {
                max = buffer[i];
                imax = i;
            }
        }
        for (i=imax;i<rear-1;i++)
            buffer[i]=buffer[i+1];   // remove the max value
        rear = rear - 1;
        return max;
    }
    else {
        cout<< "Error: Queue empty"<<endl;
        return -1;
    }
}
void main() {                  // main function must be outside any class
    Queue Q1(5);               // will call constructor Queue(int)
    Q1.enqueue(2);          // insert 2 into Q1
    Q1.enqueue(8);             // insert 8 into Q1
    int x = Q1.dequeue();
    int y = Q1.dequeue();
    cout << "x = " << x << endl << "y = " << y << endl;
    Queue *Q2 = new Queue(4);  // will call constructor Queue(int)
    Q2->enqueue(12);           // insert 12 into Q2
    Q2->enqueue(18);           // insert 18 into Q2
    x = Q2->dequeue();
    y = Q2->dequeue();
    cout << "x = " << x << endl << "y = " << y << endl;
    PriQueue *Q3 = new PriQueue(4); // will call constructor Queue(int)
    Q3->enqueue(12);           // insert 12 into Q3
    Q3->enqueue(18);           // insert 18 into Q3
    Q3->enqueue(14);           // insert 14 into Q3
    x = Q3->getMax();
    y = Q3->getMax();
```

```
  cout << "x = " << x << endl << "y = " << y << endl;
  delete Q2; Q2 = NULL;
  delete Q3; Q3 = NULL;
}
```

3.2 Class definition and composition

The class is the fundamental building block of C++ programs. A **class** consists of a number of **class members**. A member can be a data member or a function member (also called a member function or method).

3.2.1 Class definition

The syntax of the class definition is given in Figure 3.1. The access privilege of a member has three levels. If a member is prefixed with the **public** access privilege, the member can be accessed by any functions inside or outside the class. If a member is prefixed with no access privilege (default access privilege) or the **private** access privilege, the member can only be accessed by member functions within the same class. If a member is prefixed with the **protected** access privilege, the member can only be accessed by member functions within the same class and the inherited classes (derived classes).

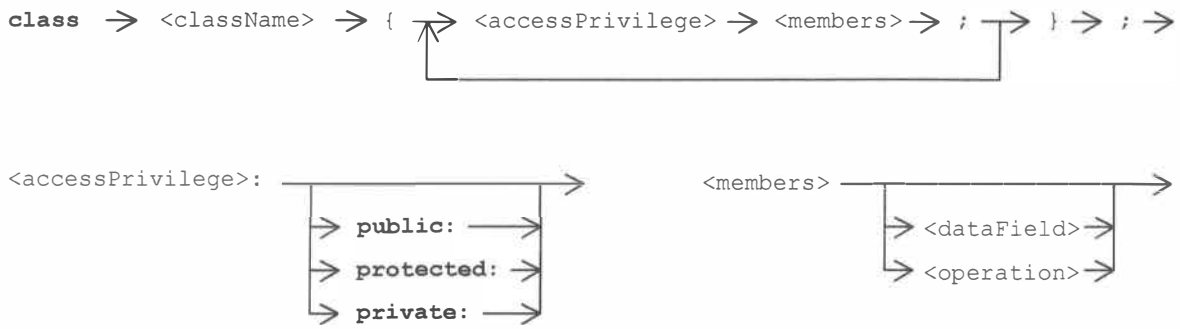


Figure 3.1. Syntax graphs of class definition.

Public and protected access privileges give access to the group of classes that meet the conditions. C++ also allows a class to give access to any particular class or function through the definition of friend access. For example, when you define class A, you can list class B and function C as friends. Then, all functions in class B and function C will be able to access all public, protected, and private data and functions in A. In other words, a friend function has the same access privilege as a function inside the class. Notice that the friendship is not mutual. If class A lists class B as a friend, all functions in class B can access all members in class A. However, the functions in class B cannot access the protected and private members. The following code shows an example of declaring a friend class.

```
#include <iostream>
using namespace std;
class Circle {
friend class Cylinder;  // Declare class Cylinder as a friend
private:
  double radius;
  double area(double r) { return 3.1416*r*r ; }
};
```

```
class Cylinder {
public:
   double volume(Circle &c, double r, double h) {
        double a = c.area(r) ; // area function is private in Circle class
        return a*h ;
   };
};
void main() {
   double v, r = 5, h = 10 ;
   Circle c;
   Cylinder y ;
   v = y.volume(c, r, h) ;
   cout << "cylinder volume = "  << v << endl ;
   // The program will output: cylinder volume = 785.4
}
```

Now we will examine the class definition in the `Queue` example given in Section 3.1. In this example, we define a `Queue` class that has several data and function members. There is one data member that is private. There are three data members that are protected. All function members are declared as public so that they can be accessed by all functions in the program.

In the previous chapter, we discussed that a C structure consists of a number of data members. As you can see in the definition of a C++ class, the C structure is a special case of a class when:

- there are only data members and no member functions;
- all members are public members.

If you are familiar with Java, you can see that the class definitions in the two languages are similar. However, C++ allows you to put the implementation part of a member function outside the class to separate the specification (function declaration) from the implementation (the body of the function) by using a scope resolution operator.

3.2.2 Scope resolution operator

In Java, implementations of member functions (methods) are always within the class definition. In C++, they can be inside the class definition (for short functions) or outside the class definition (for longer functions). It is more efficient to have function implementations in the class because we save the time needed to jump to another memory area to access the implementations. However, it is structurally clearer to separate the implementation from the specification.

If the implementation of a function is outside the class, we must specify the class to which the implementation belongs. The **scope resolution operator** in C++ serves this purpose. It consists of the class name and two consecutive colons. Generally, the format is

```
return_type class_name::function_name(parameters){implementation};
```

For the `Queue` example we discussed above, we put the shorter functions `enqueue` and `dequeue` in the class, while putting the longer function's implementation outside the class using the **scope resolution operator**, that is "`Queue::`," as shown in the code below.

```
bool Queue::compact(void){    // Queue:: is the scope resolution operator
  if (front == 0) {
```

```
        cout<<"Error: Queue overflow"<<endl;
        return false;
    }
    else {
        for (int i=0;i<rear-front;i++)
            buffer[i]=buffer[i+front];
        rear = rear - front;
        front = 0;
        return true;
    }
}
```

For much longer implementation, it makes the structure better to put the body of a member function outside the class definition, if the implementation is very long (e.g., over a hundred lines).

3.2.3 Objects from a class

Like a structure in C, a class in C++ can be used to declare variables. The variable declared by a class is called an **object** of the class. For example:

```
Queue r;              // r is a variable (object) of Queue class.
Queue *s;             // s is a pointer to a Queue variable.
r.enqueue(25);        // push/add a number into the Queue variable
x = r.dequeue();      // pop/remove the front element from the Queue
s = new Queue;        // create a heap object of the Queue and link it to s
s->enqueue(25);       // push a number into the object pointed to by s
y = s->dequeue();     // pop the front element from the Queue object.
```

As can be seen in this example, we can allocate memory for a variable (object) from the stack or from the heap. In this example, we declare `r` as a variable of `Queue` class. The variable obtains memory (or the object is created) from the stack during the compilation. We can immediately push an integer into the `r` object by the `r.enqueue(int)` operation.

On the other hand, variable `s` is not an object of the `Queue` class. It is a pointer to a `Queue` object from the heap. The object is dynamically created by a `new` operation:

```
s = new Queue();
```

Having created the object, we push an integer into the object linked to `s` by the `s->enqueue(int)` operation.

3.2.4 Definition of constructor and destructor

A **constructor** in a class is a special member function whose name is the same as the class name. A constructor is used to automatically initialize objects when an object is created. We can also define multiple constructors with different numbers of parameters or different types of parameters. Defining two or more functions with the same name is called **overloading**. As long as these functions have different parameter lists, they are considered different functions by the compiler.

A **destructor** is a special member function whose name is the same as the class name but prefixed by the tilde character "~." The tilde character is often used for complement operation, suggesting that the destructor is the complement operation of the constructor. A destructor is used to delete objects (collect

garbage) created within the class, normally within the constructors. A destructor cannot have any parameters or a return value, and thus it cannot be overloaded. The definition and the use of constructors are basically same as those in Java. However, there is no destructor in Java programs. Instead, Java uses a built-in garbage collector to collect automatically all objects that have no references.

We defined two overloaded constructors and a destructor in the `Queue` class in the example program. The first constructor has no parameters and it creates an object of array with 10 elements. The second constructor takes a parameter and creates an object of array with the given number of elements. Both constructors initialize the `front` and `rear` pointer to zeros, indicating an empty queue.

The definitions of constructors and the destructor are relatively simple. However, understanding how memory is allocated by the constructor and deallocated by the destructor is not trivial. We will devote the entire next section to discussing the general memory management in imperative and object-oriented programming languages like C, C++, and Java.

3.3 Memory management and garbage collection

When a program is started, the operating system (OS) will allocate a segment of memory to the program. The memory allocated to a program is managed by the programming language environment (runtime system) and it is divided into three areas: **static**, **stack,** and **heap** areas as shown in Figure 3.2.

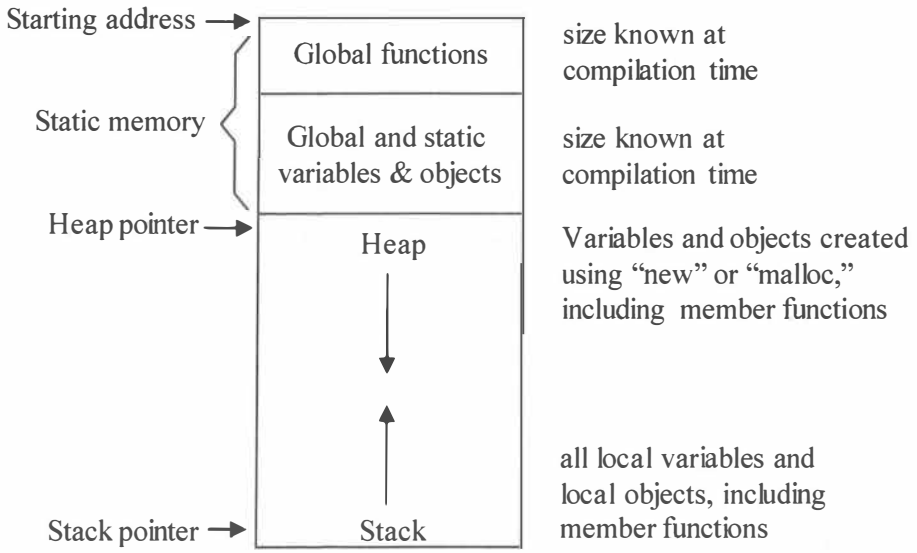


Figure 3.2. Partition of the memory allocated to a program.

The programmers can choose from which area to obtain memory by declaring their variables in different ways. In C/C++:

All **static** local variables in functions obtain memory from static memory. In other words, if we want to have memory from the static area for any variable, we can add the qualifier `static` before the variable declaration, for example, "`static int s;`" will declare a static integer variable. All global variables (variables declared outside any functions) obtain memory from static memory, whether the qualifier `static` is used or not.

All non-static local variables in functions obtain memory from the stack. These functions include the global functions and the member functions in objects. This stack is also called **program runtime stack** to differentiate it from other possible stacks used in the computer system.

All dynamically allocated variables obtain memory from the heap. In C, the memory allocated by the function `malloc` is from the heap area. In C++, the memory allocated by using `new` is from the heap area. If a class contains data members and function members, the data members will obtain memory from the heap when an object is created using new. However, local variables in the member functions will not obtain memory until the functions are being executed, and the local variables in these functions will obtain memory from the language stack.

Now the question is what difference will it make to a programmer to use these different memory areas? This question will be answered in the following subsections.

3.3.1 Static: global variables and static local variables

Static memory is allocated statically, that is, during the compilation stage (before the program is executed). There is only one copy of the static memory. Changes made to a static variable will have an impact on all the other functions that use the variable. A static variable will go out of scope only if the program is terminated. Why do we need a static local variable?

The following function shows a situation where using a static local variable is better than using a global variable:

```
void login() {
  static int counter = 0;     // will be initialized only once
  readId_pwd( );
  if (verified( ))
      counter++;              // count the # of users logged in
}
```

This function allows users (callers) to login to a secure resource and keeps track of the number of users who entered the resource. If the `counter` variable is not declared as `static`, the variable will be re-allocated and initialized every time the function is called. As a `static` variable, `counter` will be allocated and initialized only once. Thus, the variable can keep the history of the logins.

If a static local variable is declared as a class member, it may not be initialized when it is declared. It must be initialized outside the class using the scope resolution operator. In the `Queue` class example, a static local variable `counter` is declared in the class. It has to be initialized outside the class using the scope resolution operator:

```
class Queue {
  ...
  static int counter;     // declaration is inside the class
  ...
}
int Queue::counter = 0;  // initialization is outside the class
```

The variable `counter` is incremented in the constructors and decremented in the destructor. The variable can keep track of how many objects of the `Queue` class exist.

Generally, a static local variable could be declared as a global variable. The advantages, however, of using a static local variable are twofold:

- It puts the variable declaration in the place where the variable is actually used. It makes the program easier to read and to understand.
- It prevents other functions from accessing the variable. As a global variable, all other functions can read and modify it.

Although a static local variable exists all the time, it is invisible outside the function. In the following program example, we define a static local variable `counter` in the `login()` function and a global variable also called `counter`. Although both variables obtain their memory from the static memory area, they are two independent variables and there is no name conflict.

```
#include <iostream>
using namespace std;
int counter = 0;                // Global variable
void login(void) {
  static int counter = 0;       // Static local variable
  counter++;                    // Modify local variable
  cout << "login counter = " << counter << endl;
}
void main(void) {
  int i;
  for (i=0;i<5;i++) {
      counter = counter + 2;
      login();
  }
  cout << "global counter = " << counter << endl;
}
```

The output of the program is as follows. It can be seen that, within the function `login()`, only the static local variable `counter` is visible. Outside the function, only the global variable is visible.

```
login counter = 1
login counter = 2
login counter = 3
login counter = 4
login counter = 5
global counter = 10
```

3.3.2 Runtime stack for local variables

Local variables are variables declared within a function. When the control enters a function, a block of memory (called a **stack frame**) is created on the stack. All non-static local variables obtain memory from the stack frame. When the control leaves the function, all these local variables are freed and the contents of these variables are no longer valid (no longer accessible). More details on stack frames used to accommodate local variables and to support reentrant and recursive function calls are discussed in Appendix A.

The stack memory allocation is illustrated in the example in Figure 3.3. As shown in the left part of Figure 3.3, the program consists of two functions. The `main` function has one local variable `i`. The function `bar` has two local variables `j` and `k`. Please note that the formal parameter of a function is a local variable to the function.

The state (0) shows the initial state of the stack before the main function is executed. When the control enters the main function, the local variable i obtains the memory on the top of the stack, as shown in stack state (1). The value of i is initialized to 0 and then incremented to 1. Then i is passed as the actual parameter to the function bar. When the control enters the function bar, the two local variables j and k obtain memory on the top of the stack, as shown in state (2). The value of i is passed to the formal parameter j. Please note that j and i have different memory locations. j has a copy of i's value. When j is modified in the function bar, the modification has no impact on variable i.

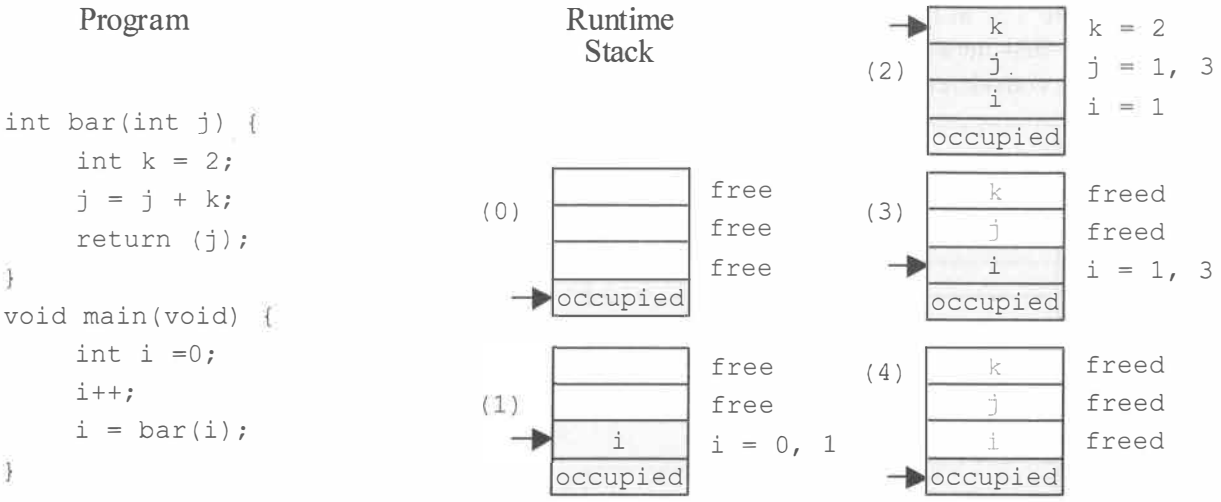


Figure 3.3. A simple program and its runtime stack.

When the control exits the function bar, variables j and k go out of scope. The stack pointer returns to its original position before it entered the function. The memory used for variables j and k, is thus freed, as shown in stack state (3). Therefore, we cannot access j and k outside the function. Finally, when the control exits the main function, variable i is also freed and the stack pointer returns to the position it held before it entered the main function, as shown in state (4) in Figure 3.3.

Since local variables are automatically garbage-collected by the runtime stack when they go out of scope, there is no need for programmers to explicitly return the memory to the system.

Having understood the stack used to allocate memory for local variables, we can easily understand how recursive functions are implemented. In fact, no special mechanism is needed. The stack that handles all local variables handles the variables in recursive functions, too.

Now we examine the following recursive function fac(n). There are two local variables in the function: The formal parameter n and a temporary variable fac that holds the return value from the $(n-1)^{th}$ iteration. Figure 3.4 shows the runtime stack before and during the execution of the recursive function fac(3).

```
#include <iostream>
using namespace std;
int fac(int n) {
  if (n <= 1) return 1; else return n* fac(n - 1);
}
void main() {
  int i = 3, j;
  j = fac(i);
```

```
    cout << "j = " << j<< endl;
}
```

State `(0)` is the state before the function `fac(3)` is called. When the control enters the function `fac(n)` for the first time, the two local variables `n` and `fac` obtained memory on the stack, as shown in state `(1)`. Formal parameter `n` is initialized to the actual parameter `3`, but variable `fac` is not given a value yet. Within the first iteration, `fac(n-1)` is called and the function is reentered. Again, the two local variables obtain memory from the stack. Now, `n` is initialized to the actual parameter value `2` and `fac` is not given a value, as shown in state `(2)` in Figure 3.4. The variables `n` and `fac` in the second iteration are different from `n` and `fac` in the first iteration, although they happen to have the same names. Since they have different scopes, they are considered different variables.

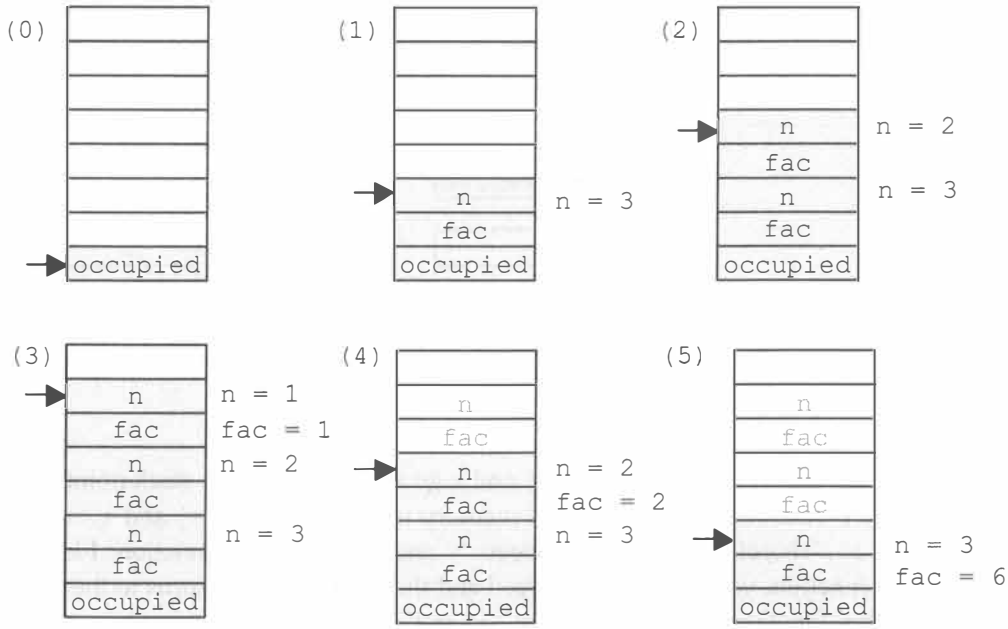


Figure 3.4. The runtime stack of a recursive program.

Within the second iteration, `fac(n-1)` is called again. In this iteration, `n` is initialized to `1` and the condition `(n <= 1)` is true. Now the function `fac(1)` is actually completed. It did not complete before. Now a value is then returned to fac in this iteration, as shown in state `(3)`. The return of iteration 3 completed the function call `fac(1)` in iteration 2 and the return value `fac = 1` is passed into iteration 2. The operation `n*fac(1)` then will produce a value `2`, as shown in state `(4)`. The return value `2` will, in turn, be passed to iteration 1 and produces a value `6`, as shown in state `(5)` in Figure 3.4. When the final iteration is completed, the `fac(n)` function exits and the stack pointer will return to its original state `(0)`.

If you compare the recursive function call here and the ordinary function call in the previous example, you can see that the processes of variable allocation on the stack are handled in exactly the same way.

In fact, at the assembly language (or machine code) level, the variable `fac` used for holding the return value is not on the stack. Instead, a register is used. A register can be considered a global variable used by the compiler, which is invisible to the high-level language programmers. Since the concept of register is not a part of high-level language programming, we use a stack variable `fac` here to make the value passing visible on the stack.

3.3.3 Heap: dynamic memory allocation

The third area in the data section is the heap. Heap is used for dynamic memory allocation requested by operations like `malloc(size)` in C and `new class_name` in C++. For example, in C,

```
struct Contact { // define a structure
   char name[30];
   int phone;
} *p;
p = (struct Contact *) malloc(sizeof(struct Contact));
```

The function `malloc` will acquire a memory block from the heap. The size of the memory block should be the size of the `Contact` type variable, and we cast the address of the piece of memory to the `Contact` pointer type. In C++, we use the `new` operator to acquire the memory for an object of `Contact` class, as shown in the code below:

```
class Contact { // define a class
   char name[30];
   int phone;
};
Contact *p = new Contact();
```

The data types that acquire memory from the heap are called **reference types** because their variables take memory addresses (references) as their values.

3.3.4 Scope and garbage collection

So far, we have explained when we should use static, stack, and heap memory. We have also explained how we acquire memory from static, stack, and heap areas. The last question we need to answer is, do we need to worry about **garbage collection**? In other words, do we need to deallocate memory that we allocated? The answer to the question depends on where we acquire the memory.

According to the definitions, global and static variables should exist for the entire lifetime of the program (even when they are invisible), and thus they should never be garbage-collected by the programmer or by the runtime system. When the `main` function exits, the OS that starts the program will reclaim the entire memory segment allocated to the program. Thus, if a variable or an object is static or global, we do not need to worry about collecting its memory.

If a variable or an object obtains its memory from the stack, the memory will be deallocated automatically by the system. As explained in Section 3.3.2, when the control enters a function, local variables or objects obtain memory from the stack. When the control exits the function, the stack pointer moves back to the original position it held when it entered the function; that is, the memory allocated to the local variables is returned back to the stack. This memory deallocation is managed by the scope rule of the language: The scope of a variable starts from the declaration and ends at the end of the block. When a variable goes out of scope, the memory allocated to the variable returns to the system.

However, if variables or objects acquire their memory from the heap, it will **not** be automatically deallocated or freed. We will have to explicitly use `free` and `delete` operations to return the memory allocated by `malloc` in C and by `new` in C++, respectively.

Now we go back to the `Queue` example discussed earlier in this chapter. We now focus on the constructor and the instantiation of objects in another function, as shown in the following segment of the code:

```
Queue(int n) {          // overloaded constructor with one parameter
```

```
    front = 0;
    rear = 0;
    queue_size = n;
    buffer = new int[queue_size]; // create a heap object of array
}
void foo() {
    Queue Q1(5);                  // will call constructor Queue(int)
    Q1.enqueue(2);              // insert 2 into Q1
    Q1.enqueue(8);                // insert 8 into Q1
    int x = Q1.dequeue();
    int y = Q1.dequeue();
    cout << "x = " << x << endl << "y = " << y << endl;

    Queue *Q2 = new Queue(4);  // will call constructor Queue(int)
    Q2->enqueue(12);              // insert 12 into Q2
    Q2->enqueue(18);              // insert 18 into Q2
    x = Q2->dequeue();
    y = Q2->dequeue();
    cout << "x = " << x << endl << "y = " << y << endl;

    delete Q2; Q2 = NULL;
    delete Q3; Q3 = NULL;
}
```

The declaration "`Queue Q1(5);`" will create a stack object `Q1` shown in the left part of the diagram in Figure 3.5. Object `Q1` has four data members: `queue_size`, `front`, `rear`, and a pointer variable `*buffer`. When `Q1` is created, the constructor `Queue(int n)` will be called and a heap object `int[5]` will be created and linked to the pointer `*buffer`, as shown in the right part of the diagram in Figure 3.5.

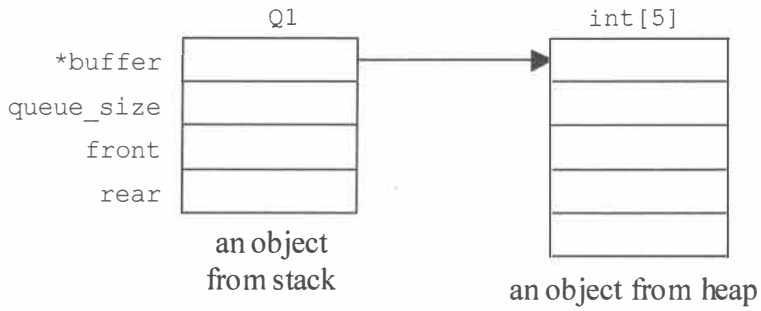


Figure 3.5. A stack object is first created and then a heap object is created.

On the other hand, the declaration with the `new` operator "`Queue *Q2 = new Queue(5);`" will create a pointer variable `Q2` on the stack, and then create a `Queue` object with four data members from the heap. The constructor `Queue(int n)` will create a further object `int[5]` from the heap and link it to the pointer `*buffer`, as shown in Figure 3.6.

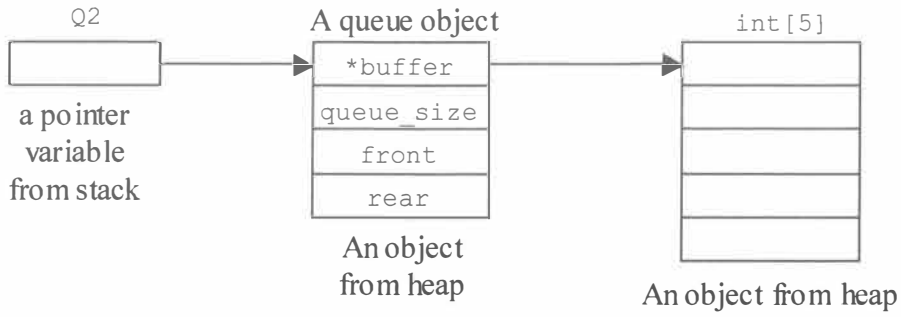


Figure 3.6. A pointer variable is created from the stack; a `Queue` object and an `int[5]` object are created from the heap.

Since any program will be allocated to only a limited amount of memory, garbage collection is extremely important, especially for those programs that continuously add and delete data. In Java, an automatic and expensive garbage-collection mechanism is implemented to collect unused memory. It is expensive because the system has to maintain a reference counter associated with each object created. When the reference counter drops to zero, the object is no longer accessible and thus can be collected. C++ does not have an automatic garbage-collection mechanism. The garbage is partially collected by the system and partially the responsibility of the programmers. Table 3.1 summarizes the memory allocation and deallocation mechanisms, and the applications of static, stack, and heap memory.

Memory	Allocation	Deallocation	Application
Static	Global variables: declared outside any class or function; local variables with `static` prefix	Deallocated by the system when the main function exits	Unique copy in global or local context
Stack	Local variables declared in a function	Deallocated by the system when the function exits	Temporary variable used in a function
Heap	Use `malloc` in C; or use `new` in C++	The programmer must explicitly use `free` in C; or use `delete` in C++	Variables that should never go out of scope unless explicitly deleted

Table 3.1. Allocation and deallocation of different kinds of memory.

As we can see from the table, the heap is the only memory that programmers need to garbage-collect. If you write a constructor and create an object in the constructor, you must write a destructor and call the `delete` operation in the destructor to delete the object. If you are using a class defined by somebody else, you should not worry about garbage-collecting the objects created in the constructor. The destructor should collect the object automatically.

If you create an object in your program, you must call the `delete` operation in an appropriate place to delete the object. For example, assume you are writing a program to maintain a database. You create an object in a function called `insertion()` that adds a new data item into your database. Then you should have a function called `deletion()` that removes an unwanted data item from the database. In the deletion function, you should not only remove the links to the data item, but also use the `delete` operation to garbage-collect the memory that holds the data item. If you fail to do so, you have a memory leak that will eventually lead to failure (out of memory) of the entire system. Now consider memory deallocation in the `foo()` function in the `Queue` example. We created a stack object `Q1` and a heap object is created in the

constructor and linked to the `buffer` pointer in `Q1`, as shown in Figure 3.5. When `Q1` goes out of scope, `Q1` is returned to the stack. Now what happens to the heap object (integer array) linked to `Q1`? If `Q1` is simply deallocated, the object from the heap will hang up and can never be accessed or garbage-collected. To avoid this situation, the C++ runtime system will call the destructor whenever a function exits, to delete any objects that are created in a constructor and are linked to an object that is going out of scope.

In the `foo()` function in the example above, we also created a heap object and linked it to `Q2`. Since this object is not created in a constructor, we must call the `delete` operation outside the destructor to delete the object. We call the `delete` operation before exiting the `foo()` function. When the object linked to `Q2` is deallocated, what happens to the heap object (integer array) linked to the object, as shown in Figure 3.6? Again, to garbage-collect the heap object created in the constructor, the `delete` operation will call the destructor to delete any objects that are created in a constructor and are linked to an object that is being deleted.

In summary, the destructor will be called in the following situations:

1. When a function exits and a local object (from stack) goes out of scope. The destructor will be called to garbage-collect any objects linked to the local object before the memory of the local object goes out of scope.
2. When a program (the `main` function) ends and global or static objects exist. Since the `main` function is treated like any other functions, the destructor will be called. However, when the `main` function exits, the entire program exits, the OS will take back the entire memory segment allocated to the program.
3. When the `delete` function is called. An object allocated using the `new` operator (from heap) outside the class must be explicitly deallocated using the `delete` operator. The destructor will be called to garbage-collect any objects linked to the object to be deleted.
4. When the destructor is explicitly called. The destructor is simply a function. A user can call the destructor function like calling any other functions.

Furthermore, to garbage-collect the memory in a collection of objects, we need to delete each object through an iteration structure. For example, to delete a linked list of object, one cannot simply set the head pointer to null or delete the head only. Below is an example of deleting a linked list using a while loop, which delete the objects one by one from the head to the tail.

```
temp = head;
while (temp != null) {
  temp = temp->next;
  delete head;
  head = temp;
}
```

One can also use a recursive procedure to delete the objects backward from the tail to the head, as shown in the code below:

```
void deleteList(ListNode *p) {
  if (p == null)
     return;
  else {
     deleteList(p->next); // size-(n-1) problem
     delete p;
  }
```

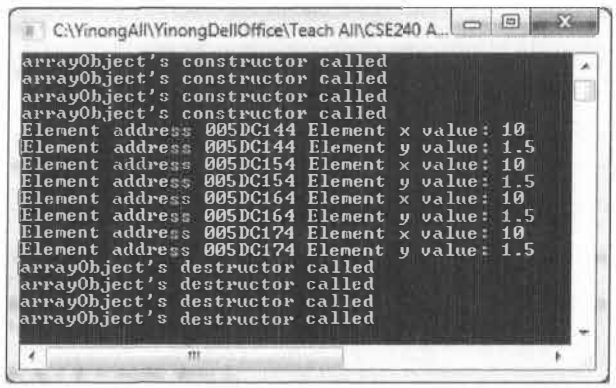
```
}
```

How do we delete an array of objects? We can use a loop to delete each element, just like we did in the examples above.

However, the language provides a library function to delete all the elements one by one without the user to explicitly use a loop. The operation is delete[] p; where p is pointing to an array of objects or structures. The code below shows an example of using delete[] p operation:

```
#include <iostream>
using namespace std;
#define size 4
class arrayObject {
public:
  int x; double y;
  arrayObject(){  // constructor showing it is called by printing
      cout << "arrayObject's constructor called" << endl;
  }
  ~arrayObject(){  // destructor showing it is called by printing
      cout << "arrayObject's destructor called" << endl;
  }
};
void main() {
  arrayObject *p, *q; // declare two pointer variables to the object
  p = new arrayObject[size]; // Create an array of objects
  for (q = p; q < p + size; q++) {// Initialize the objects
      q->x = 10;
      q->y = 1.5;
  }
  for (q = p; q < p + size; q++) {
      cout<<"Element address "<<q<<" Element x value: "<<q->x<<endl;
      cout<<"Element address "<<q<<" Element y value: "<<q->y<<endl;
  }  delete[] p;
}
```

The output of the program shows that the destructor is called four times (size of the array), because the delete operation is called four times automatically to delete each element of the array.



3.3.5 Memory leak detection

To make sure that your program does not have a memory leak, you should create a memory allocation and deallocation table that lists all the malloc (new) calls and the corresponding free (delete) calls that garbage-collect the memory created by the malloc (new) calls.

Tools have been developed to detect memory leak. Visual Studio has a built-in tool to detect possible leak. To use the tool, you will need to define the flag to map malloc calls to C Run-Time (CRT) debugger and to include necessary library functions.

We will use the following program to demonstrate memory leak detection in Visual Studio. The program is the C++ version of the case study discussed in Chapter 2, Section 2.10.

```
#include <iostream>
#include <string>
#include <stdlib.h> // include memory leak detection library functions
#include <crtdbg.h> // include memory leak detection library functions
#pragma warning(disable: 4996) // comment out if not in Visual Studio
using namespace std;
/************************* Macros ************************/
#define _CRTDBG_MAP_ALLOC // Define flag to map malloc calls to debugger
#define ARRAY_SIZE 32
/****************** Forward Declarations *****************/
class container;
class person;
void branching(char c, container** pointerToHead);                    // given
char* get_name();
void printFirst(container* root);
int insertion(container** pointerToHead);
person* search(container* root, char* sname);
void deleteFirst(container** pointerToHead);
void deleteAll(container** pointerToHead);
void printAll(container* root);
/****************** Class Definitions *****************/
// A class to hold attributes of a person
class person{
public:
  char *name;   // a pointer to string. No memory for the string
  char *email;  // a pointer to string. No memory for the string
  int phone;
  person() { // Default constructor.
      name = NULL;
      email = NULL;
      phone = 0;
  };
  // Constructor WITH parameters to initialize the class properties
  person(char *cName, char *cEmail, int iPhone) {
```

```
        name = new (char[ARRAY_SIZE]);    // acquiring memory for storing
name
        email = new (char[ARRAY_SIZE]);   // acquiring memory for storing
email
        strcpy(name, cName);          // initialize name
        strcpy(email, cEmail);            // initialize email
        phone = iPhone;               // initialize phone
    };
    virtual ~person() {   // Destructor: Deallocate all heap memory.
        delete name;
        delete email;
    };
};
class container {    // A class for nodes of the linked list.
public:
    person *plink;     // points to a person object
    container *next;
    container() { // Default Constructor
        plink = NULL;
        next = NULL;
    };
    // Constructor initializing plink and next.
    container(person *p, container *n) {
        plink = p;
        next = n;
    };
};
int main()  { // Entry point of the program.
    container* head = NULL; // Declare head as a local variable of main
    // Print a menu for selection
    char ch = 'i';
    do {
        std::cout << "Enter your selection" << endl;
        std::cout << "\ti: insert a new entry" << endl;
        std::cout << "\td: delete one entry" << endl;
        std::cout << "\tr: delete all entries" << endl;
        std::cout << "\ts: search an entry" << endl;
        std::cout << "\tp: print all entries" << endl;
        std::cout << "\tq: quit" << endl;
        std::cin >> ch;
        ch = tolower(ch); // Convert any uppercase char to lowercase.
        branching(ch, &head);  // passing pointer to head pointer
        std::cout << endl;
    } while (ch != 'q');
```

```
    return 0;
};
// Branch to different tasks:, search, delete, print
void branching(char c, container** pointerToHead) {
    char *p;
    switch (c) {
    case 'i':
        insertion(pointerToHead); break;
    case 's':
        p = get_name();
        search(*pointerToHead, p); break;
    case 'd':
        deleteFirst(pointerToHead); break;
    case 'r':
        deleteAll(pointerToHead); break;
    case 'p':
        printAll(*pointerToHead); break;
    case 'q':
        deleteAll(pointerToHead); // free all memory
        _CrtDumpMemoryLeaks(); // Call memory leak report function
        break;
    default:
        std::cout << "Invalid input. Please select again" << endl;
    }
};
// Recursively delete the entire list given the head of a linked list.
void deleteAll(container** pointerToHead) {
    if (*pointerToHead == NULL) {
        std::cout << "\nThe list is empty. Nothing was deleted." << endl;
        return;
    }
    else{
        // Delete the next item in the list.
        deleteAll(&(*pointerToHead)->next); // recursive call
        deleteFirst(pointerToHead);
    }
};
// Delete the first person node in the linked list.
void deleteFirst(container** pointerToHead) {
    int i = 0;
    container *toDelete = NULL;
    if (*pointerToHead == NULL) { // *pointerToHead is the head of the list
        std::cout << "\nThe list is empty. Nothing was deleted." << endl;
    }
```

```
    // (*pointerToHead)->next is equivalent to saying head->next
    else if ((*pointerToHead)->next == NULL) {
        // delete (*pointerToHead)->plink; // delete the person object
        delete*pointerToHead;
        *pointerToHead = NULL;
    }
    else{
        toDelete = *pointerToHead; // toDelete = head;
        *pointerToHead = (*pointerToHead)->next; //head = head->next;
        // delete toDelete->plink; // delete the person object
        delete toDelete;
        toDelete = NULL;
        std::cout << "\nA container node was deleted." << endl;
    }
};
/* Inserts the person lexigraphically given the head of a list.
Note: A < a so all capital letters will be ordered first. */
int insertion(container** pointerToHead) {
    int i = 0;
    container* newNode = NULL, *iterator = NULL, *follower = NULL;
    person* newPerson = NULL;
    char name[ARRAY_SIZE];
    char email[ARRAY_SIZE];
    int phone = 0;
    newNode = new container();
    // Case 1: The program is out of memory.
    if (newNode == NULL) {
        std::cout << "Fatal Error: Out of Memory. Exiting now." << endl;
        return 0;
    }
    // Case 2: The structure still has unfilled slots.
    else{
        cin.ignore();
        std::cout << "Enter the name:" << endl;
        std::cin.getline(name, ARRAY_SIZE, '\n');
        std::cout << "Enter the phone number:" << endl;
        cin >> phone;
        std::cout << "Enter the e-mail:" << endl;
        cin.ignore();
        std::cin.getline(email, ARRAY_SIZE, '\n');
        newPerson = new person(name, email, phone);
        if (newPerson == NULL) {
            std::cout << "Fatal Error: Out of Memory. Exiting now." <<
endl;
```

```
            return 0;
        }
        else{
            newNode->plink = (newPerson);
            if (*pointerToHead == NULL) {
                *pointerToHead = newNode;
                (*pointerToHead)->next = (NULL);
                return 0;
            }
            else{
                if (strcmp(newPerson->name, (*pointerToHead)->plink->name)
< 0){
                    newNode->next = (*pointerToHead);
                    *pointerToHead = newNode;
                    return 0;
                }
                iterator = *pointerToHead;
                follower = iterator;
                while (iterator != NULL) {
                    if (strcmp(newPerson->name, iterator->plink->name) <
0) {
                        newNode->next = (iterator);
                        follower->next = (newNode);
                        return 0;
                    }
                    follower = iterator;
                    iterator = iterator->next;
                }
                follower->next = (newNode);
                newNode->next = (NULL);
                return 0;
            }
        }
    }
    return 0;
};
char * get_name() {// Read the input from the user.
    // Use dynamic memory which does not go out of scope
    char *p = new(char[ARRAY_SIZE]);
    std::cout << "Please enter a name for the search: " << endl;
    std::cin >> p;
    return p;
};
// Print the name, e-mail, phone, and education level of each person.
```

```
// It calls the helper printFirst to recursively print the list
void printAll(container* root) {
  container* iterator = root;
  //Case 1: The list is empty
  if (iterator == NULL) {
      std::cout << "\nNo entries found." << endl;
      return;
  }
  // Case 2: The list has at least one item in it
  else{
      printFirst(root);
      return;
  }
};
void printFirst(container* root){// Print the root of the linked list
  if (root != NULL){
      std::cout << "\n\nname = " << root->plink->name << endl;
      std::cout << "email = " << root->plink->email << endl;
      std::cout << "phone = " << root->plink->phone << endl;
      printFirst(root->next);
  }
};
//Find a person by comparing names given the head of the linked list.
person* search(container* root, char* sname) {
  container* iterator = root;
  while (iterator != NULL) {
      if (strcmp(sname, iterator->plink->name) == 0) {
          std::cout << "\n\nname = " << iterator->plink->name << endl;
          std::cout << "email = " << iterator->plink->email << endl;
          std::cout << "phone = " << iterator->plink->phone << endl;
          delete sname; // garbage collection
          return iterator->plink;
      }
      iterator = iterator->next;
  }
  std::cout << "The name does not exist." << endl;
  delete sname; // garbage collection
  return NULL;
}
```

In the deleteFirst function, two delete operations are intentionally commented out:

```
      // delete (*pointerToHead)->plink; // delete the person object
      // delete toDelete->plink; // delete the person object
```

which will cause memory leak, as the person object linked to the container object will not be deleted when the container object is deleted. This memory leak will be detected if we draw a memory allocation and deallocation table, as shown in Table 3.2 that lists all memory allocation and corresponding deallocation. The simple principle is: for each new (or malloc) operation, there must be a delete (free) operation to deallocate the memory.

Function name	new() calls	Function name	delete calls
constructor person()	name = new (char[ARRAY_SIZE]) email = new (char[ARRAY_SIZE])	destructor ~person()	delete name delete email
get_name()	char *p = new (char[ARRAY_SIZE])	search()	delete sname
insertion()	newNode = new container()	deleteFirst()	delete *pointerToHead or delete toDelete
insertion()	newPerson = newperson(name, email, phone);	deleteFirst()	No delete operation is found to match this new() call.

Table 3.2. Memory allocation and deallocation table.

If we uncomment the following two lines of code in the deleteFirst function, the table will be complete and the memory leak problem will be solved.

```
// delete (*pointerToHead)->plink; // delete the person object
// delete toDelete->plink; // delete the person object
```

Another possible implementation is to delete the person object in destructor of the container class, instead of in the deleteFirst function:

```
virtual ~container() {
    delete plink;
};
```

The advantage of using the destructor is to make sure that no memory leak can occur even if the user's function deleteFirst overlooks the need of delete plink call.

Various tools have been developed to detect memory leak. Visual Studio has a built-in tool to detect possible leak. To use the tool, you will need to define the flag to map malloc calls to CRT debugger and include necessary library functions, as shown in the snippet of code.

```
#define _CRTDBG_MAP_ALLOC // Define flag to map malloc calls to debugger
#include <stdlib.h>  // include memory leak detection library functions
#include <crtdbg.h>  // include memory leak detection library functions
// your other code here
_CrtDumpMemoryLeaks();   // Call memory leak reporter before returning
Return;
```

The memory leak detection code has been included in the foregoing example. When we run the code in the debugging mode (start with Debugging), with the two delete operations commented out, the following memory leak report will be generated:

```
Detected memory leaks!
Dumping objects ->
{188} normal block at 0x0068C440, 32 bytes long.
 Data: <john@asu.edu    > 6A 6F 68 6E 40 61 73 75 2E 65 64 75 00 CD CD CD
```

```
{187} normal block at 0x0068C3E0, 32 bytes long.
 Data: <John            > 4A 6F 68 6E 00 CD CD CD CD CD CD CD CD CD CD CD
{186} normal block at 0x0068C390, 16 bytes long.
 Data: <      h @ h 9   > 94 FD 1E 01 E0 C3 68 00 40 C4 68 00 39 9C DC 02
Object dump complete.
The program '[9044] MyCppProject.exe' has exited with code 0 (0x0).
```

After we uncomment these two delete operations, or include the destructor ~container() with delete plink operation, the memory leak problem will be fixed.

You can find more detail of Visual Studio memory leak detection tool at:

https://msdn.microsoft.com/en-us/library/x98tx3cf.aspx

Memory leak detection tools are also developed for other programming environments. For example, Valgrind is a GNU GCC/G++ memory leak detection tool. Its detail can be found at:

http://www.valgrind.org/docs/

The tool is installed in GNU GCC/G++ in ASU general server. The following commands can be used to include the memory leak detection tool:

```
g++ -o myProg -g myProg.cc // Compile
// The use of -g allows exact line numbers in error messages
valgrind --leak-check=full --tool=memcheck ./myProg
```

3.4 Inheritance

3.4.1 Class containment and inheritance

C language does not support the inheritance from one structure to another. However, we can still share the structures defined before. The way we reuse the existing structures is through a **class containment** mechanism, that is, we can use a structure to declare a variable within another structure, and, thus, we do not have to redefine the existing structure. Such a containment mechanism is also available in C++. For example, assume that we have defined a class `Employee` as follows:

```
class Employee {
  char      name[30];
  long      id;
  char      department[50];
  int       salary(int base, int bonus) { ... };
  int       tax(int thisMonth, int thisYear) { ... };
}
```

Now we want to define a linked list to hold the information of all the employees. We can then define an employee node class in the following two different ways:

```
// Definition 1:
class EmployeeNode1 {
  Employee      data;   // containing the class Employee
  EmployeeNode *next;
}
```

```
// Definition 2:
class EmployeeNode2 {
  char          name[30];
  long          id;
  char          department[50];
  int           salary(int base, int bonus) { ... };
  int           tax(int thisMonth, int thisYear) { ... };
  EmployeeNode  *next;
}
```

In the definition of `EmployeeNode1`, we contain an `Employee` class in the `EmployeeNode1` class, while in the definition of `EmployeeNode2`, we copy (rewrite) all the data members and member functions into the new class. What are the advantages and disadvantages of these two definitions?

The advantages of using a containment mechanism are:

- The new class is more concise;
- There is a level of abstraction in the `EmployeeNode1`. This class has only two members: A data member that contains employee information and a pointer that points to the next node. Normally, we do not need the detail of the data. However, if we do need to access the detail of the data member, it will be a bit less convenient. For example, if we declare a stack object by "`EmployeeNode x;`", then we will have to use `x.data.name` (three sections), `x.data.id`, `x.data.department`, `x.data.salary()`, and `x.data.tax()` to access the members; and use `x.next` (two sections only) to access the `next` member in `EmployeeNode1` object. Since the members from the `Employee` class are not semantically related to the next member in the `EmployeeNode1`, it makes sense to separate them in two different levels.

The second advantage could become a disadvantage if these members are semantically related and should not be separated. Now we define two new manager classes based on the employee class as follows:

```
// Definition 1:
class Manager1 {
  Employee  empl;    // containing the class Employee
  int       rank;
}

// Definition 2:
class Manager2 {
  char      name[30];
  long      id;
  char      department[50];
  int       salary(int base, int bonus) { ... };
  int       tax(int thisMonth, int thisYear) { ... };
  int       rank;
}
```

Here the extra member `rank` defined in the manager classes is related to the members in the `Employee` class. It should be put at the same level as the other members, so that members' `name`, `id`, `department`,

`salary()`, and `tax()` can be accessed in the same way as the member rank. Thus, the class definition `Manager2` is more suitable than `Manager1`.

What is wrong with the definition of the `Manager2` class? There are several problems associated with the approach of repeating the members in another class:

- It wastes time and space, especially when the member functions `salary()` and `tax()` in the `Employee` class are very long.
- A more serious problem is data integrity: Redundant structures exist in your program. When you change the `Employee` class, you must change the `Manager` class to preserve data integrity. If you have multiple redundant data in different places, it is very difficult to maintain the code.

Object-oriented programming languages introduced the inheritance mechanism to address the problem. The **inheritance mechanism** supports the definition of a new class based on an existing class. In C++, the existing class is called the **base class** and the new class is called the **derived class**. In Java, they are called parent (super) class and child (sub) class, respectively. The inheritance mechanism allows the derived class to

- inherit all members (data members and member functions) of the base class without having to repeat any of them;
- be able to add new members;
- be able to redefine members of the base class.

Using the inheritance mechanism, we can define a new manager class as follows:

```
class Manager3 : public Employee {
  int       rank;
}
```

The `Manager3` class can be used in exactly the same way as the `Manager2` class; that is, we can access the inherited members from the `Employee` class in the same way as using the new member `rank`. The only difference is that when the members of the `Employee` class are modified, the inherited members of the `Manager3` class are automatically modified.

As we can see from the above discussion, containment and inheritance mechanisms have different applications. Now the question is how can we decide what mechanism is the right one for a particular situation? As mentioned above, we can make the decision based on whether the new members are really related to the members in the base class. Another way to help you make the decision is to characterize the containment relation and the inheritance relation as "***has-a***" relation and "***is-a***" relation. For example, an employee node (in a linked list) has an employee as its data, a car *has-a* wheel, a university *has-a* student; and a manager *is-an* employee, pixel *is-a* point, etc. Whenever the *has-a* relation holds, we apply the containment mechanism, and whenever the *is-a* relation holds, we apply the inheritance mechanism. For example:

- Since it is better to consider that the employee node *has-an* (contains an) employee than to consider that the employee node *is-an* employee, we should apply the containment mechanism.
- Since it is better to consider that the manager *is-an* employee than to consider that the manager *has-an* (contains an) employee, we should apply the inheritance mechanism.

Now we examine our `Queue` and `PriQueue` example at the beginning of the chapter. Obviously, it is better to consider that the **priority queue** `PriQueue` *is-a* `Queue` than to consider that the `PriQueue` *has-a* `Queue`. Thus, we used the inheritance mechanism to define the `PriQueue`.

```
class PriQueue : public Queue {
public:
  int getMax(void); // return and remove the max value from the queue
  PriQueue(int n) : Queue(n) { };
  // base class constructor may not be inherited. It has to be explicitly
  // called. PriQueue's constructor simply calls Queue's constructor;
  ~PriQueue() {          // base class destructor may not be called here or
      delete buffer;     // inherited. We must explicitly use delete.
      buffer = 0;
  };
};
```

The third line defines a new member in the `PriQueue` class that returns and removes the element with the maximum value from the queue. The fourth line defines the constructor of the `PriQueue` class. Please note that the base class's constructor may not be inherited. It has to be explicitly called. `PriQueue`'s constructor simply calls the `Queue`'s constructor. The next member function is the destructor of the `PriQueue` class. The base class's destructor may not be inherited and may not be called in the derived class's destructor. Thus, we have to use the explicit delete operation to delete the objects created in the constructor of the `PriQueue` class.

Inheritance is useful in associating many related classes and organizing them into a hierarchy of classes. From this section and in the following sections, we will use a more complicated example to illustrate the major properties of object-oriented programming languages, including inheritance, class hierarchy, virtual function, late binding, polymorphism, and type checking.

3.4.2 Inheritance and virtual function

In using inheritance, if a member only shares the name with its base class and has a different data structure or functionality, we can redefine or override the member in the derived class. For any member of a class to be redefined, we must put the keyword `virtual` before the name. In the derived classes, we may, but we do not have to, use the keyword `virtual` before the member that has been declared as `virtual`. If a member is declared as `virtual`, all redefined members in the derived classes will be virtual, no matter whether the keyword `virtual` is used or not. The destructor of a class must be defined as `virtual` if the class is used as a base class and there are destructors defined in the derived classes.

The following program defines four classes: Color, Shape, Rectangle, and Triangle. The Shape is the base class and the Rectangle and Triangle are its child class. The Color does not have an inheritance relationship with the other classes. It is used as a member in the Shape class. The program demonstrates the use of inheritance and virtual functions.

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <string>
using namespace std;
#pragma warning(disable: 4996) // comment out if not in Visual Studio
class Color {
public:
  string name;
```

```
public:
   Color(string color){
        name = color;
   }
};
class Shape {
protected:
   int width, height;
   string *shapeName;
   Color *color;
public:
   Shape(int edge = 0, const char *name = "Shape") {
        width = edge;
        shapeName = new string(name);
        color = NULL;
   }
   ~Shape() {
        delete shapeName;
   }
   int area() {
        cout << "Called Shape's area." << endl;
        return 0;
   }
   virtual void print_name() {
        cout << "I am a: " << *shapeName << endl;
   }
   virtual void print_color() = 0;
};
class Rectangle : public Shape {
public:
   Rectangle(int rectWidth = 0, int rectHeight = 0) : Shape(rectWidth,
"Rectangle") {
        height = rectHeight;
        color = new Color("Red");
   }
   ~Rectangle() {
        delete color;
   }
   int area() {
        cout << "Called Rectangle's area." << endl;
        return (width * height);
   }
   void print_name() {
```

```
        cout << "I am a: " << width << " x "<< height << " " << *shapeName
<< endl;
    }
    void print_color() {
        if (color != NULL)
            cout << "I am " << color->name << endl;
    }
};
class Triangle : public Shape {
public:
    Triangle(int triBase = 0, int triHeight = 0) : Shape(triBase, "Triangle")
    {
        height = triHeight;
        color = new Color("Blue");
    }
    ~Triangle() {
        delete color;
    }
    int area() {
        cout << "Called Triangle's area." << endl;
        return (width * height / 2);
    }
    void print_name() {
        cout << "I am a: " << *shapeName << " with a right angel" << endl;
    }
    void print_color() {
        if (color != NULL)
            cout << "I am " << color->name << endl;
    }
};
int main(int argc, char *argv[]) {
    Shape *shape;
    Rectangle rect(12, 7);
    Triangle  tri(15, 8);
    rect.area();
    tri.area();
    shape = &rect;
    shape->print_name();
    shape->area();
    shape->print_color();
    shape = &tri;
    shape->print_name();
    shape->area();
    shape->print_color();
```

```
    return 0;
}
```

The output of the program is given as follows:

```
Called Rectangle's area.
Called Triangle's area.
I am a: 12 x 7 Rectangle
Called Shape's area.
I am Red
I am a: Triangle with a right angle
Called Shape's area.
I am Blue
```

In the base class of the program, print_name function is defined as virtual, and the same function is redefined in the Rectangle and Triangle class to print different information.

You can also define an **abstract function**, an interface, or a **pure virtual function**, by assigning the virtual function to 0. In the base class of the program, the print_color function is an abstract function:

```
virtual void print_color() = 0;
```

An abstract function must be overridden in the derived classes. If a class contains an abstract function, the class is an **abstract class** and it cannot be used to instantiate an object. It can only be used to derive new classes. In this example, Shape class is an abstract class.

3.4.3 Inheritance and hierarchy

Assume that we are developing a database for a university to hold personnel information of students and employees. We break the information to be stored into multiple classes and organize the classes in a hierarchy using inheritance. As shown in Figure 3.7, the class `Personnel` is the root of the inheritance tree. Classes `Student` and `Employee` inherit `Personnel`. Classes `Faculty`, `Staff`, and `Consultant` inherit `Employee`, etc. In each class, a number of members are defined. According to the inheritance principle, a child (derived) class will inherit the members of its parent (base) class and, in turn, its grandparent class, great grandparent class, etc.

In Figure 3.7, the `display()` function in each node will print each data member in the class. Since each class has different data members, we have to write a different display function in each class. It is thus defined as a virtual function. Of course, we could use different names in different classes. However, we will shortly see that it is a much better way to redefine the display function than to use different names.

C++ also supports **multiple inheritances**, that is, a class can inherit members from more than one class. The need for multiple inheritances arises when a class has an *is-a* relation with multiple classes. For example, the `Consultant` class inherits the `Employees` and `ConsultingCo` classes, as shown in Figure 3.7. Syntactically, the use of multiple inheritances is simple; we simply list the classes from which the derived class wants to inherit. For example, using the multiple inheritances, we defined the `Consultant` class as follows:

```
class Consultant: public Employee, public ConsultingCo{ // inherit two
classes
  ... // new members here
}
```

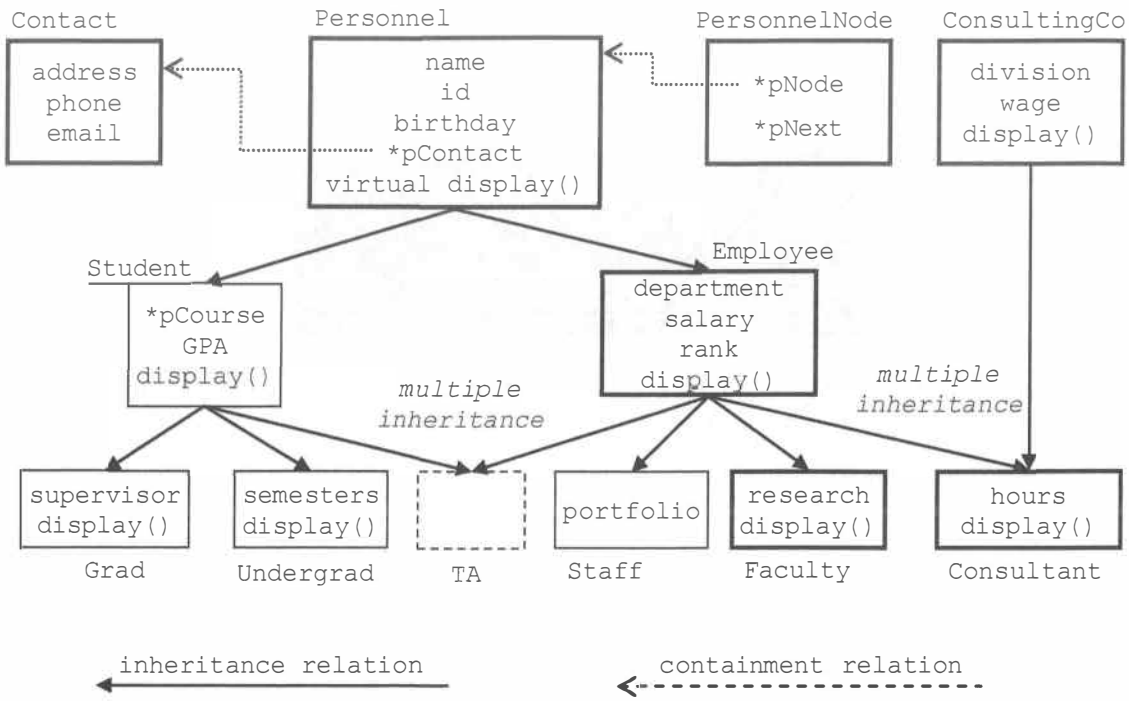


Figure 3.7. Organizing classes in a hierarchy.

However, the semantics of multiple inheritances are complex and error prone. Inheritance must be used with caution. In this example, the `Consultant` class inherits from two classes that are not in the same inheritance hierarchy. If a class inherits two classes in the same hierarchy, we must use a **virtual base class** to avoid the duplication of members in the derived class. For example, assume we have a `TA` (teaching assistant) class that inherits from `Employee` and `Student` classes. We must define the `Personnel` class as a virtual base class. Then, when we derive the `Employee` and `Student` classes, the members in `Personnel` (i.e., `name`, `id`, etc.), will not be doubly copied into the `TA` class through `Employee` and `Student`, as shown in the following piece of code:

```
class Personnel {
   // members
};
class Employee : virtual public Personnel {
   // members
};
class Student : virtual public Personnel {
    // members
};
class TA : public Employee, public Student {
    // Member list
};
void main() {
   ...
}
```

Please note that we defined two classes, `Contact` and `PersonnelNode`, which do not have the inheritance or *is-a* relation with other classes. Containment relation is used to relate them to a class in the hierarchy. Class `Contact` is contained in the class `Personnel`, and class `PersonnelNode` contains a `Personnel` class.

We could put a particular member in different classes. The principle is to put a member in a class where all its derived classes share the member. For example, features like `name`, `id`, and `birthday` that everyone shares should be in the `Personnel` class that is the root of the hierarchy.

A C++ implementation of a part of the hierarchy in Figure 3.7 is given below. The program includes the classes `Contact`, `Personnel`, `Employee`, `Faculty`, `ConsultingCo`, `Consultant`, `PersonnelNode`, and a list of global functions. In the global functions, the menu function allows users to select desired operations (e.g., insertion of a new node and printing the node information). Different insertion functions are used to create a new object of different classes and insert it into the linked list. A single `remove()` function deletes an object of any class from the linked list and garbage-collects the memory allocated to the object using an explicit delete operation. The `display_all()` function displays the object information stored in the linked list. The `display_all()` function will call the polymorphic member functions defined in each class to display class-specific information. More details of the program are given as comments in the program.

The purpose of this program is to illustrate:

- **Class definition**: `public`, `protected`, and `private` members, and use `public` member functions to access `private` members.
- **Class inheritance and hierarchy**: `Employee` inherits `Personnel` and `Faculty` inherits `Employee`;
- **Multiple inheritance**: `Consultant` inherits `Employee` and `ConsultingCo` classes.
- **Class containment**: `Personnel` class contains `Contact` class, and `PersonnelNode` class contains `Personnel` class.
- **Constructor**: The constructor is defined in every class. The constructor of a derived class can call the constructor of the base class.
- **Destructor**: Destructors are defined in `Personnel`, `Employee`, and `Faculty` classes. The destructor of a derived class cannot call the destructor of the base class. It has to repeat the statements if it wants to perform the same operations.
- **Explicit garbage collection**: Objects created in the insertion functions have to be deleted (garbage-collected) explicitly. The program contains a `remove()` function. Heap objects created in the insertion functions are explicitly deleted using the `delete` function.
- **Polymorphism**: Polymorphic pointer: Use a pointer to `Personnel` class object to access objects of derived classes. Polymorphic function: `display()` function is defined as `virtual` and a different function is called when the polymorphic pointer points to a different class object.
- **Linked list**: In the `PersonnelNode` class, a pointer `pNext` to `PersonnelNode` (itself) is declared. The pointer forms the links of the linked list.
- **Global functions**: Member functions in each class are associated with the object of the class. Global functions are not associated with any class and can be called anywhere in the program.
- **Global variable**: `head` is defined as a global variable that holds the starting address of the linked list; sometimes called a head pointer.
- **Static local variable**: In the constructor of the `Personnel` class, a static local variable `iIdCounter` is declared to hold the current ID number that has been issued. The `static` qualifier

ensures that the variable is initialized only once and incremented after each instantiation of a new object.

```
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
using namespace std;
class Contact {
  private:
       char cAddress[75];
       char cPhone[25];
       char cEmail[75];
  public:
       void setAddress(char *cAddress) {strcpy(this->cAddress, cAddress);}
       void setPhone(char *cPhone) { strcpy(this->cPhone, cPhone); }
       void setEmail(char *cEmail) { strcpy(this->cEmail, cEmail); }
       char* getAddress() { return cAddress; }
       char* getPhone() { return cPhone; }
       char* getEmail() { return cEmail; }
       Contact(char *cAddress, char *cPhone, char *cEmail) { // constructor
            setAddress(cAddress);
            setPhone(cPhone);
            setEmail(cEmail);
       }
};
class Personnel {
  private:
       char cName[50];
       int iId;
       char cBirthday[11];
  protected:
       Contact *pContact;
  public:
       void setName(char *cName) { strcpy(this->cName, cName); }
       void setBirthday(char *cBirthday) {strcpy(this->cBirthday,
cBirthday);}
       void setId(int iId) { this->iId = iId; }
       char* getName() { return cName; }
       int getId() { return iId;}
       char* getBirthday() { return cBirthday; }
       Contact * getContact() { return pContact; }
       virtual void display() {
            cout << "PERSONNEL" << endl;
            cout << "Name:\t" << getName() << endl;
```

```
            cout << "Id:\t" << getId() << endl;
            cout << "Birthday:\t" << getBirthday() << endl;
            cout << "Address:\t" << pContact->getAddress() << endl;
            cout << "Phone:\t" << pContact->getPhone() << endl;
            cout << "Email:\t" << pContact->getEmail() << endl;
      }
      Personnel(char *cName, char *cBirthday, char *cAddress, char
*cPhone, char *cEmail) {                                    // constructor
            static int iIdCounter = 0;
            setName(cName);       // set Name
            setId(iIdCounter);    // set Id by a self-incremental counter
            iIdCounter++;         // increment the ID generator;
            setBirthday(cBirthday); // set Birthday
            pContact = new Contact(cAddress, cPhone, cEmail);
      }
      virtual ~Personnel() {            // destructor
            delete pContact;      // delete the object linked to pContact
            pContact = NULL;      // Make sure pContact not point to an
object
      }
};
class Employee : public Personnel  { // inherit from Personnel
  private:
      char cDepartment[75];
      float fSalary;
      char cRank[75];
  public:
      void setDepartment(char *cDepartment) {strcpy(this->cDepartment,
cDepartment);}
      void setSalary(float fSalary) { this->fSalary = fSalary; }
      virtual void setRank(char *cRank) { strcpy(this->cRank, cRank); }
      char* getDepartment() { return cDepartment; }
      float getSalary() { return fSalary; }
      virtual char* getRank() { return cRank; }
      void display() {
            Personnel::display();
            cout << "EMPLOYEE" << endl;
            cout << "Department:\t" << getDepartment() << endl;
            cout << "Salary:\t" << getSalary() << endl;
            cout << "Rank:\t" << getRank() << endl;
      }
      Employee(char *cName, char *cBirthday, char *cAddress, char *cPhone,
char *cEmail, char *cDepartment, float fSalary, char *cRank)
            : Personnel(cName, cBirthday, cAddress, cPhone, cEmail)
      {
```

```
            setDepartment(cDepartment);
            setSalary(fSalary);
            setRank(cRank);
        }
        virtual ~Employee() {              // destructor
            delete pContact;        // delete the object linked to pContact
            pContact = NULL;        // Make sure pContact does not point to
        }                           // any object
};
class Faculty : public Employee { // inherit from Employee
  private:
        char cResearch[75];
  public:
        virtual void setResearch(char *cResearch) {
            strcpy(this->cResearch, cResearch);
        }
        char* getResearch() { return cResearch; }
        virtual void display() {
            Employee::display();
            cout << "FACULTY" << endl;
            cout << "Research\t" << getResearch() << endl;
        }
        Faculty(char *cName, char *cBirthday, char *cAddress, char *cPhone,
char *cEmail, char *cDepartment, float fSalary, char *cRank, char
*cResearch)
            : Employee(cName, cBirthday, cAddress, cPhone, cEmail,
cDepartment, fSalary, cRank)
        {
            setResearch(cResearch);
        }
        virtual ~Faculty() {               // destructor
            delete pContact;        // delete the object linked to pContact
            pContact = NULL;        // Make sure pContact does not
        }                           // point to any object
};
class ConsultingCo {               // not inherit from a class
  private:
        char division[30];
        float wage;
  public:
        virtual void display(){
            cout << "ConsultingCo" << endl;
            cout << "Division: " << getDivision() << endl;
            cout << "Wage: " << getWage() << endl;
```

```
        }
        char* getDivision(){
             return division;
        }
        void setDivision(char *division_){
             strcpy(this->division, division_);
        }
        float getWage(){
             return wage;
        }
        void setWage(float wage_){
             this->wage = wage_;
        }
        ConsultingCo(char *division_, float wage_) {
             setDivision(division_);
             setWage(wage_);
        }
};
// multiple inheritance: Consultant class inherit from two classes
class Consultant : public Employee, public ConsultingCo{
  private:
       int hours;
  public:
       virtual void display() {
            Employee::display();
            ConsultingCo::display();
            cout << "Consultant" << endl;
            cout << "Hours: " << getHours() << endl;
       }
       int getHours(){ return hours;}
       void setHours(int hours_) {this->hours = hours_;}
       Consultant(char *cName, char *cBirthday, char *cAddress, char
*cPhone, char *cEmail, char *cDepartment, float fSalary, char *cRank, char*
division_, float wage_, int hours_)
            : Employee(cName, cBirthday, cAddress, cPhone, cEmail,
cDepartment, fSalary, cRank), ConsultingCo(division_, wage_) {
            setHours(hours_);
       }
       virtual ~Consultant() {           // destructor
            delete pContact;       // delete the object linked to pContact
            pContact = NULL;       // Make sure pContact does not point to
       }                           // any object
};
```

```
class PersonnelNode {          // This is a container class
  private:
      Personnel     *pNode; // It contains a Personnel class
      PersonnelNode *pNext; // pointer used to form a linked list
  public:
      void setNode(Personnel *pNode) { this->pNode = pNode; }
      void setNext(PersonnelNode *pNext) { this->pNext = pNext; }
      Personnel* getNode() { return pNode; }
      PersonnelNode* getNext() { return pNext; }

      PersonnelNode() {      // constructor
           pNode = NULL;
           pNext = NULL;
      }
} *head = NULL; // declare a global pointer variable head
// Forward declaration of global functions outside any class
int main();              // The main function
void menu();             // display main choices
void branching(char);    // branch to the chosen function
void sub_menu();         // display different insertion options
void insert();           // call sub_menu() and branch to chosen function
int insert_personnel();  // insert a personnel node
int insert_employee();   // insert an employee node
int insert_faculty();    // insert a faculty node
int insert_consultant();
void remove();               // call remove function
void display_all();      // display members in all nodes in the linked list
void display_node(Personnel*, int);  // display the members in one node

int main() {     // main function is the entry point of the program
  char ch;
  cout << "CLASSES INHERITANCE, HIERARCHY AND POLYMORPHISM" << endl;
  cout << "**********************************************" << endl;
  do {
       menu();                    // display choices
       cin >> ch;                 // enter a choice from the keyboard
       ch = tolower(ch);     // convert to lower case
       branching(ch);        // branch to the chosen function
  }
  while (ch != 'q');     // 'q' for quit
  return 0;
}
void menu() {
  cout <<   endl << endl << "MENU" << endl;
```

```
   cout << "----" << endl;
   cout << "i: Insert a new entry." << endl;
   cout << "d: display all entries." << endl;
   cout << "r: remove an entry." << endl;
   cout << "q: Quit the program." << endl;
   cout << endl << "Please enter your choice (i, d, r, or q) --> ";
}
void branching(char c) {      // branch to chosen function
    switch(c) {
       case 'i': insert();
                 break;
       case 'd': display_all();
                 break;
       case 'r': remove();
                 break;
       case 'q': cout << endl << "@Exiting the program......" << endl;
                 cin.get();   //type any key.
                 break;
       default:  cout << endl << "@ERROR - Invalid input." << endl;
                 cout << "@Try again......" << endl;
    }
}
void sub_menu() {     // display insertion options
   cout <<   endl << endl << "INSERTION SUB-MENU" << endl;
   cout << "------------------" << endl;
   cout << "p: insert a personnel entry." << endl;
   cout << "e: insert an employee entry." << endl;
   cout << "f: insert a faculty entry." << endl;
   cout << "c: insert a consultant entry." << endl;
   cout << "q: Quit insertion and back to main menu." << endl;
   cout << endl << "Please enter your choice (p, e, f, c, or q) --> ";
}
// insert() is the umbrella insertion function that calls different
// insertion functions according to the selection in the sub-menu.
void insert() {
   char ch;
   cout << endl << "@Insertion module.............";
   do {
       sub_menu();
       cin >> ch;
       ch = tolower(ch);
       switch(ch) {
            case 'p':    if(insert_personnel() != 0)
                             cout << "@INSERTION FAILED." << endl;
```

```
                        else
                            cout << "@INSERTED SUCCESSFULLY." << endl;
                            break;
            case 'e':   if(insert_employee() != 0)
                            cout << "@INSERTION FAILED." << endl;
                        else
                            cout << "@INSERTED SUCCESSFULLY." << endl;
                            break;
            case 'f':   if(insert_faculty() != 0)
                            cout << "@INSERTION FAILED." << endl;
                        else
                            cout << "@INSERTED SUCCESSFULLY." << endl;
                            break;
            case 'c':   if(insert_consultant() != 0)
                            cout << "@INSERTION FAILED." << endl;
                        else
                            cout << "@INSERTED SUCCESSFULLY." << endl;
                            break;
            case 'q':   cout << endl << "@Exiting the insertion..." <<
endl;
                        cin.get();
                        break;
            default:    cout << endl << "@ERROR - Invalid input." << endl;
                        cout << "@Try again....." << endl;
      }
   }
   while (ch != 'q');          // do-while statement
}
int insert_personnel() {       // insert a Personnel node
   Personnel *person = NULL;
   PersonnelNode *node = NULL;
   char name[50], birthday[11], address[75], phone[25], email[75];
   cout << endl << "@Inserting personnel node........." << endl;
   cout << "Enter the name: ";
   cin.ignore(); // fflush()
   cin.getline(name, 50);
   cout << "Enter the birthday, e.g., 05/24/1985: ";
   cin.getline(birthday, 11);
   cout << "Enter the address: ";
   cin.getline(address, 75);
   cout << "Enter the phone number: ";
   cin.getline(phone, 25);
   cout << "Enter the email: ";
   cin.getline(email, 75);
```

```
    person = new Personnel(name, birthday, address, phone, email);
    node = new PersonnelNode();
    if((person != NULL) && (node != NULL)) {
        node->setNode(person);
        node->setNext(head);
        head = node;
        return 0;
    }
    else {
        cout << endl << "@ERROR - Could not allocate enough memory!" <<
endl;
        return -1;
    }
}
int insert_employee() {       // insert an Employee node
    Personnel *person = NULL;
    PersonnelNode *node = NULL;
    char name[50], birthday[11], address[75], phone[25], email[75],
department[75], rank[75];
    float salary;
    cout << endl << "@Inserting employee node....." << endl;
    cout << "Enter the name: ";
    cin.ignore();
    cin.getline(name, 50);
    cout << "Enter the birthday, e.g., 05/24/1985: ";
    cin.getline(birthday, 11);
    cout << "Enter the address:";
    cin.getline(address, 75);
    cout << "Enter the phone number: ";
    cin.getline(phone, 25);
    cout << "Enter the email: ";
    cin.getline(email, 75);
    cout << "Enter the department: ";
    cin.getline(department, 75);
    cout << "Enter the salary. It must be a float number: ";
    cin >> salary;
    cout << "Enter the rank: ";
    cin.ignore();
    cin.getline(rank, 75);
    person = new Employee(name, birthday, address, phone, email, department,
salary, rank);
    node = new PersonnelNode();
    if((person != NULL) && (node != NULL)) {
        node->setNode(person);
```

```
        node->setNext(head);
        head = node;
        return 0;
    }
    else {
        cout << endl << "@ERROR - Could not allocate enough memory!" <<
endl;
        return -1;
    }
}
int insert_faculty() {        // insert a Faculty node
    Personnel *person = NULL;
    PersonnelNode *node = NULL;
    char name[50], birthday[11], address[75], phone[25], email[75],
department[75], rank[75], research[75];
    float salary;
    cout << endl << "@Inserting faculty node....." << endl;
    cout << "Enter the name: ";
    cin.ignore();
    cin.getline(name, 50);
    cout << "Enter the birthday, e.g., 05/24/1985: ";
    cin.getline(birthday, 11);
    cout << "Enter the address: ";
    cin.getline(address, 75);
    cout << "Enter the phone number: ";
    cin.getline(phone, 25);
    cout << "Enter the email: ";
    cin.getline(email, 75);
    cout << "Enter the department: ";
    cin.getline(department, 75);
    cout << "Enter the salary. It must be a float number: ";
    cin >> salary;
    cout << "Enter the rank: ";
    cin.ignore(); cin.getline(rank, 75);
    cout << "Enter the research: ";
    cin.getline(research, 75);
    person = new Faculty(name, birthday, address, phone, email, department,
salary, rank, research);
    node = new PersonnelNode();
    if((person != NULL) && (node != NULL)) {
        node->setNode(person);
        node->setNext(head);
        head = node;
        return 0;
```

```
  }
  else {
      cout << endl << "@ERROR - Could not allocate enough memory!" <<
endl;
      return -1;
  }
}
int insert_consultant() {          // insert a Faculty node
  Personnel *person = NULL;
  PersonnelNode *node = NULL;
  char name[50], birthday[11], address[75], phone[25], email[75],
department[75], rank[75], division[30];
  float salary, wage;
  int hours;
  cout << endl << "@Inserting consultant node....." << endl;
  cout << "Enter the name: ";
  cin.ignore();
  cin.getline(name, 50);
  cout << "Enter the birthday, e.g., 05/24/1985: ";
  cin.getline(birthday, 11);
  cout << "Enter the address: ";
  cin.getline(address, 75);
  cout << "Enter the phone number: ";
  cin.getline(phone, 25);
  cout << "Enter the email: ";
  cin.getline(email, 75);
  cout << "Enter the department: ";
  cin.getline(department, 75);
  cout << "Enter the salary. It must be a float number: ";
  cin >> salary;
  cout << "Enter the rank: ";
  cin.ignore();
  cin.getline(rank, 75);
  cout << "Enter the division: ";
  cin.getline(division, 30);
  cout << "Enter the wage. It must be a float number: ";
  cin >> wage;
  cout << "Enter the hours. It must be an integer: ";
  cin >> hours;
  person = new Consultant(name, birthday, address, phone, email,
department, salary, rank, division, wage, hours);
  node = new PersonnelNode();
  if((person != NULL) && (node != NULL)) {
      node->setNode(person);
```

```
        node->setNext(head);
        head = node;
        return 0;
    }
    else {
        cout << endl << "@ERROR - Could not allocate enough memory!" <<
endl;
        return -1;
    }
}
/* void remove() function removes a node in the linked list. In the remove
function, an id number will be entered as the key. The node whose id field
stored an id number that is equal to the entered id number will be removed.
*/
void remove() {
    int id;
    PersonnelNode *temp, *prev;
    Personnel *person;
    int index = 1;
    cout<<"Remove module.............\n" << endl ;
    cout<<"Please enter the ID number of the person to be deleted: "<<endl;
    cin>> id;
    temp = head;
    while (temp != NULL) {
        person = temp->getNode();
            if (id != person->getId()){
                prev = temp;
                temp = temp->getNext();
                index++;
            }
            else {
                cout <<"Person to delete is found at index"<<index<<endl;
                display_node(person, index);
                if(temp != head)
                prev->setNext(temp->getNext());
                else
                    head = head->getNext();
                delete person;    // explicit garbage-collection
                person = NULL;    // Make it not to point to any object
                delete temp;      // explicit garbage-collection
                temp = NULL;      // Make it not to point to any object
                return;
            }
    }
    cout <<"The person with ID = << id << does not exist."<< endl;
```

```
}
void display_all() { // Display contents of all nodes in the linked list
   PersonnelNode *node;
   int node_count= 0;
   cout << endl << "@Display module......";
   node = head;
   while(node != NULL) {
        display_node(node->getNode(), ++node_count);
        node = node->getNext();
   }
   cout << endl << "@No more records." << endl;
}
void display_node(Personnel *node, int index) {// Display contents of node
   cout << endl << "Record " << index << "." << endl;
   node->display();   // Polymorphic call. Depending on the object pointed
}                              // to by node, a different display() will be
invoked.
```

3.4.4 Inheritance and polymorphism

Polymorphism is the ability to apply the same operation to different objects and to receive different forms of responses. Polymorphism in C++ is applied to different classes related by inheritance. With at least one virtual member defined, it allows a pointer declared to point to an object of class *A* to point to an object of class *B* if *B* is a descendant class of *A*. Therefore, the pointer of a base class is polymorphic. The main purpose of polymorphism is to access the virtual members. A member function of a class can be defined as a virtual function and redefined in the derived classes. Then, the same call to a virtual function in different classes will cause different functions to be invoked. Thus, the calls are polymorphic.

Polymorphism has been illustrated in the long example in Section 3.4.3, where the display() functions in different classes are virtual.

Now we examine how polymorphism simplifies the design. In the program, we defined the linked list node to have two pointer variables.

```
class PersonnelNode {
      Personnel     *pNode;
      PersonnelNode *pNext;
}
```

The first pointer variable points to an object of the `Personnel` class that is the data portion of the linked list. The second pointer variable points to an object of `PersonnelNode` that is the link portion of the linked list. Since the `pNode` is declared as a pointer to the root class's objects, it can be used to point to objects of all derived classes: `Personnel`, `Employee`, `Faculty`, `Staff`, `Consultant`, etc. In other words, we can link different objects in the same linked list, as shown in Figure 3.8.

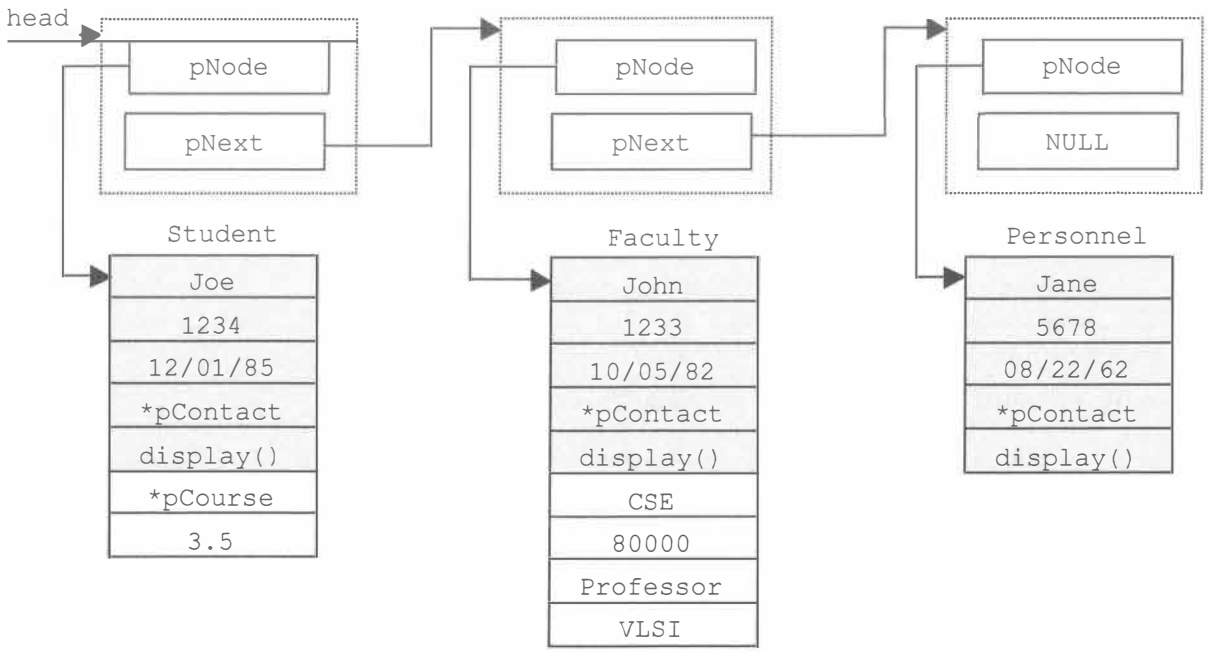


Figure 3.8. Different objects are linked into the same list.

Having inserted different nodes into the linked list, we could use the following code to print members defined in the `Personnel` object (the root node) using a pointer `pl` to the `Personnel` object:

```
PersonnelNode *pn;
for (pn = head; pn != NULL; pn = pn->next) {// traverse the linked list
  Personnel *pl = pn->pNode;                // pl is a Personnel class pointer
  cout << "name = " << pl->cName << endl;
  cout << "id = " << pl->iId << endl;
  cout << "birthday = " << pl->cBirthday << endl;
  cout << "Contact->address = " << pl->pContact->cAddress << endl;
  cout << "Contact->phone = " << pl->pContact->cPhone << endl;
  cout << "Contact->email = " << pl->pContact->cEmail << endl;
}
```

Now the question is can we add the following statements to print members defined in the `Student` class?

```
  cout << "GPA = " << pl->GPA << endl;
  cout << "department = " << pl->cBirthday << endl;
```

The answer is no, because the type checking mechanism in C++ will prevent us from using the `Personnel` pointer `pl` to access the members that do not exist in the base class `Personnel` (see next subsection). In order to print the complete information in different kinds of nodes in the linked list, we defined a virtual function `display()` in each class in the hierarchy. When pl points to a particular object, the `display()` function specifically defined in the class of the object is invoked, as shown in the following piece of code:

```
PersonnelNode *pn;
for (pn = head; pn !=0; pn = pn->pNext) {
  Personnel *pl = pn->pNode; // pl can point to any object
```

```
  p1->display();          // the function defined in current class is
invoked
}
```

Since the `Personnel` class also has the function `display()`, the compiler will not complain. However, since `display()` is redefined in the derived classes, polymorphism will cause the redefined `display()` function to be called, thus allowing the information specific to the node to be printed. For example, the `display()` function in the `Student` class could be defined as follows, in which the class-specific members GPA and major can be printed.

```
Student::display() {
  cout << "name = " << name << endl;
  cout << "id = " << id << endl;
  cout << "birthday = " << birthday << endl;
  cout << "Contact->address = " << Contact->address << endl;
  cout << "Contact->phone = " << Contact->phone << endl;
  cout << "Contact->email = " << Contact->email << endl;
  cout << "GPA = " << GPA << endl;
}
```

3.4.5 Polymorphism and type checking

As we have seen in the previous subsection, although polymorphism allows us to move a pointer from a base class object to a derived class object, it only allows us to access members inherited from the base class. It does not allow us to access the new members defined in the derived class. Observe a further example:

```
Personnel *p = new Personnel();   // link a Personnel object to p.
Employee *e = new Employee();        // link an Employee object to e.
Faculty *f = new Faculty();       // link a Faculty object to f.
```

then, according to polymorphism, we can use the pointer p to access the objects of the derived class:

```
p = e;          // p is now pointing to the object pointed by e.
p->id = 123;    // Now we use p to access members in Employee object.
p = f;          // p is now pointing to the object pointed by f.
p->id = 124;    // Now we use p to access members in Faculty object.
```

However, we cannot access the members that do not exist in the base class. For example, if we write the following statements in our program:

```
p = e;
strcpy(p->department, "computer science");
```

we will have a compilation error. The reason is: classes in C++ are essentially user-defined types. Static type checking is enforced by the compiler. After we have moved the `Personnel` class pointer to an `Employee` class's object and then access `p->id`, the compiler will consider that we are still accessing the `Personnel` class's member, and thus it will not complain. However, if we try to access `p->department`, the compiler will not be able to find the member `department` in the `Personnel` class and thus the compiler will report a compilation error.

3.4.6 Polymorphism and late binding

When a member function is declared as a virtual function, late binding will be used; that is, the function name will be bound to the actual memory locations that store the function's code, not during the compilation, but during the execution of the program. Binding a name to its memory locations during the execution stage is called **late binding** or **dynamic binding**. If late binding is applied, the function can be redefined. On the other hand, binding a name to its memory locations during the compilation stage is called **early binding** or **static binding.** If early binding is applied, the member cannot be redefined. In C++, only member functions can be dynamically bound. Data members can only be bound statically. Early binding is considered a feature of the imperative programming paradigm, while late binding is considered a feature of the object-oriented programming paradigm. In Java, late binding is applied to all member functions of a class by default. If you specifically want to apply early binding to a member function, you must use the keyword "final" to enforce the early binding. Polymorphic function calls are possible only if late binding is applied.

Why do we need early binding? For example, if you write a password verification member function in a class and do not want anyone to redefine/override the function in a derived class, you can apply early binding to prevent overriding. Another advantage of early binding is its efficiency. Late binding during execution is less efficient than early binding. For efficiency reasons, C++ applies early binding to all members of a class by default. If you specifically want to apply late binding to a member function, you must use the keyword "virtual" to enforce the late binding.

3.4.7 Type Casting in C++

In C, we can simply put the destination type in front of the variable to change the variable type to the destination type in the syntax (type) variable or type (variable). For example:

```
int x = 7;
double f = (double) x + 5;
```

Even though C-style casting can be used in C++, C++ has a set of its own casting functions. The most commonly used for implementing the aforementioned casting is:

```
int x = 7;
double f = static_cast<double>(x);
```

The options available in C++ to cast a variable or an object include:

- const_cast: It converts a variable or object into a constant one. This is cast useful if you want to allow a variable or object to be modifiable before a certain point in the programming execution, but treat it as a constant of that point.
- static_cast: It removes the type restriction to allow the object to be used as a different object, as long as the structures of the source type and the destination type are equivalent. static_cast is the common use of casting and works in most cases. The static_cast is type-checked by the compiler and is safe. The other options should be considered only if static_cast does not work.
- dynamic_cast : It is normally used to cast **pointer** among the classes in an inheritance hierarchy. There are two common applications. (1) The polymorphism allows a base class pointer to be used to point to the objects of its derived classes. In order to use a pointer of a derived class to point to an object of its base class, dynamic_cast can be used. (2) It can be used to cast a base class pointer to a derived class to obtain the full access to all the members in the derived class. In addition, it can also be used to covert a pointer in an inheritance hierarchy not only up and down, but also sideways. But this is not a common use. The dynamic_cast is not type-checked by the compiler, and the

programmer is responsible for the program's dynamic behaviors. Figure 3.9 shows an example of applying dynamic_cast.

- reinterpret_cast: This is a more powerful but dangerous version of dynamic_cast. It can be applied to convert not only pointers, but also objects. It can covert unrelated objects with different types and different sizes through truncation. It can covert a larger object to a smaller object in terms of memory use. If truncation is performed, the object cannot be casted back to the original type. Avoid using this cast unless static and dynamic casts do not work.

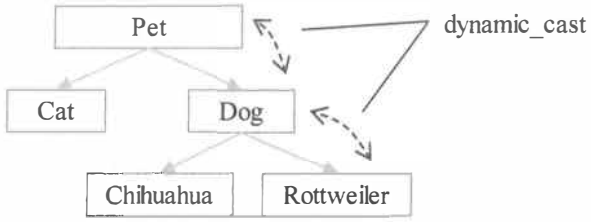


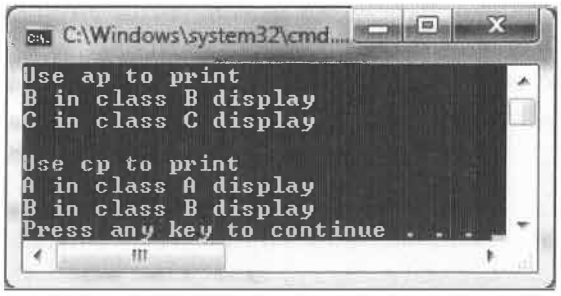
Figure 3.9. Common dynamic_cast along the inheritance hierarchy.

The following snippet of code shows moving the pointers up and down in an inheritance hierarchy consisting of three classes A, B, and C, where A is the base class, B is derived from A, and C is derived from B. We declare a pointer from each class: ap, bp, and cp. We use ap pointer to access objects of classes B and C. The polymorphism allows pointer of A to access objects of classes B and C without using casting. We then use the cp pointer to access objects of B and C. In this case, casting is required. We use static_cast to cast the objects of A and B to type C before accessing the object.

```
#include <iostream>
using namespace std;
class A {
  public: virtual void display(){ // virtual method to be redefined
       char s0 = 'A';
       cout << s0 << " in class A display\n";
  };
};
class B : public A {
  char s1 = 'B';
  public: void display() {
       cout << s1 << " in class B display\n";
  }
};
class C : public B {
  char s2 = 'C';
  public: void display() {
       cout << s2 << " in class C display\n";
  }
};
int main() {
  A *ap, objA;
  B* bp = new B();
```

```
    C *cp = new C();
    // Move pointer downwards
    cout << "Use ap to print \n";
    ap = bp;
    ap->display();// Use ap to call the display method in B
    ap = cp;
    ap->display();// Use ap to call the display method in C
    // Move pointer upwards
    // cp = bp; // Without casting, this line will cause an error
    cout << "\nUse cp to print \n";
    cp = static_cast<C*>(&objA); // use static_cast
    cp->display();// Use cp to call the display method in A
    cp = static_cast<C*>(bp); // use static_cast
    cp->display();// Use cp to call display method in B
}
```

The output of the code is shown as follows:



3.5 Function and Operator Overloading

Overloading is a useful feature allowing programmers to define and apply the same function name or operator name to different type of data. C++ allows both function overloading and operator overloading.

3.5.1 Function overloading

Function overloading: This is a feature of all object-oriented programming languages. It allows programmers to define multiple functions with the same name. However, these overloaded functions must have at least one parameter that has a different type, so that the compiler can differentiate which function to bind when a function call is made. The compiler treats them as different functions.

At the beginning of the chapter, we discussed constructor overloading. Overloading can be applied to any function in C++. In the Queue example, we have the following function:

```
void enqueue(int v) { // add an element at the end of the queue
    if (rear < queue_size)
        buffer[rear++] = v;
    else
        if (compact())
            buffer[rear++] = v;
```

```
}
```

We can define an overloaded function:

```
void enqueue(double v) {
     if (rear < queue_size)
          buffer[rear++] = v;
     else
          if (compact())
               buffer[rear++] = v;
}
```

These two functions have exactly the same functionality, but they deal with different types of values. Overloading is widely used in defining the same functions for different types of data.

Overloaded functions require to have different parameter lists. It is unfortunate that different return types cannot be used for overloading. For example, in the Queue example, we defined the function:

```
int dequeue(void){ // return and remove the 1st element from the queue
     if (front < rear)
          return buffer[front++];
     else {cout<< "Error: Queue empty"<<endl; return -1;}
}
```

We cannot define the following overloaded function:

```
double dequeue(void){
     if (front < rear)
          return buffer[front++];
     else {cout<< "Error: Queue empty"<<endl; return -1;}
}
```

The compiler will throw an error, as this function has the same parameter as the int dequeue(void) function. Both Java and C++ have the same definition of overloading.

What is the benefit of defining overloaded functions? It is more convenient for the users to call the overloaded functions. Without function overloading feature, we would have to write two different functions such as enqueueInt(int v) and enqueueDouble(double v), and the caller must use the correct function name with the parameter types. With the overloading feature, the user can call the same function using different parameter types.

Overloading allows us to have multiple constructors. Recall that a constructor must have the same name as the class. Without overloading, we can have one constructor only. A destructor cannot be overloaded, because a destructor cannot have any parameters or a return value. Two destructors could not be differentiated by the compiler.

3.5.2 Operator overloading

An operator in C++ is a built-in function that is often defined using a symbol name and the mathematical style. For example, +, -, *, /, <, ++, >>, and <<. These operators are predefined for primitive types of data. For example, +, -, *, and / are defined for numerical types, such as int and double. It is very convenient that we can overload these operators to user-defined object types. For example:

- Define a string assignment operator to allow string1 = string2; instead of using strcpy(string1, string2);
- Define a string comparison operator to allow string1 >= string2, instead of using strcmp(string1, string2);
- Define + operator on an object rectangleArea(length, width) to allow the addition of two objects: rectangleArea(3, 5) + rectangleArea(2, 6), and define a comparison operator < to allow the comparison: rectangleArea(3, 5) < rectangleArea(2, 6);
- Define + operator to add one to a Date(year, month, day) with a number (of days), and to compare two date objects: Date(2016, 10, 23) < Date(2016, 8, 31);

According to a language's orthogonality, if one operator can be overloaded, all the operators should be able to overloaded. Most of the C and C++ operators can be overloaded. However, due to their nature, there are some operators that cannot be overloaded. Table 3.3 lists a few such operators.

Operator	Description	Example
::	Scope resolution operator (C++ only)	Queue::enqueuer(int v);
.	Object member selector through value semantics	Contact.name
.*	Member of object Contact selected by pointer-to-member name	Contact.*name
?:	Ternary conditional	((a<0) ? −a : a)
sizeof	Size of an object	sizeof(Contact)
static_cast	Static cast for simple variables	static_cast <double>(i);
dynamic_cast	Static cast for pointers with inheritance relation	dynamic_cast<Contact>(p)
reinterpret_cast	For converting between different objects	reinterpret_cast<type>(a)

Table 3.3. Operators that cannot be overloaded.

In this section, we will define a few overloaded operators for object types in two examples. In the first example, we define three operators: +, -, and <, on the user-defined cylinder object. In the second example, we define prefix ++d operator and postfix d++ on user-defined Days objects.

In the first example, we define the add operator overloading as follows:

Cylinder operator+(const Cylinder &c)

The first part is the return type of the operator. It returns a Cylinder object. The second part operator+ define the operator name is +. The operator is defined as a class member, and it adds a class member with the parameter. Thus, only one parameter is defined in the parameter list. We use &c to specify the parameter-passing mechanism call-by-alias. The operation- is defined in the same way. We defined the relational operator > as follows:

```
bool operator>(const Cylinder &c)
```

To define relational operator, such as greater than, greater than or equal to, we will specify the return type as bool.

```
#define _USE_MATH_DEFINES
#include <iostream>
#include <cmath>
using namespace std;
class Cylinder {
```

```
private:
   double radius;       // radius of cylinder
   double height;       // height of cylinder
public:
   double getVolume(void) {
       return M_PI * radius * radius * height; // M_PI defined in <cmath>
   }
   void setRadius(double r) {
       radius = r;
   }
   void setHeight(int h) {
       height = h;
   }
   // Overload + operator to add two Cylinder objects.
   Cylinder operator+(const Cylinder &c) {
       Cylinder cylinder;
       cylinder.radius = this->radius + c.radius;
       cylinder.height = this->height + c.height;
       return cylinder;
   }
   // Overload - operator to subtract two Cylinder objects.
   Cylinder operator-(const Cylinder &c) {
       Cylinder cylinder;
       cylinder.radius = this->radius - c.radius;
       cylinder.height = this->height - c.height;
       return cylinder;
   }
   // Overload - operator > (greater than of two Cylinder objects.
   bool operator>(const Cylinder &c) {
       Cylinder cylinder; double vol0, vol1;
       vol0 = this->getVolume();
       vol1 = cylinder.getVolume();
       if (vol0 > vol1) return true;
       else return false;
   }
};
int main() {
   Cylinder cylinder1, cylinder2, cylinder3;
   double volume = 0.0;
   // cylinder1 and cylinder2 initialization
   cylinder1.setRadius(5.0); cylinder1.setHeight(5.0);
   cylinder2.setRadius(4.0); cylinder2.setHeight(10.0);
   // get and print volumes of cylinder1 and cylinder2
   volume = cylinder1.getVolume();
```

```
    cout << "Volume of cylinder1 : " << volume << endl;
    volume = cylinder2.getVolume();
    cout << "Volume of cylinder2 : " << volume << endl;
    // Add two objects using overloaded operator +, and get and print volume
    cylinder3 = cylinder1 + cylinder2;
    volume = cylinder3.getVolume();
    cout << "Volume of cylinder3 : " << volume << endl;
    // Subtract two object as follows:
    cylinder3 = cylinder1 - cylinder2;
    // get and print volume of cylinder 3
    volume = cylinder3.getVolume();
    cout << "Volume of cylinder3 : " << volume << endl;
    if (cylinder1 > cylinder2) // using overloaded operator >
        cout << "cylinder1 volume is greater than cylinder2 volume" << endl;
    else
        cout<<"cylinder1 volume isn't greater than cylinder2 volume"<< endl;
    return 0;
}
```

In the main program, we use the overloaded operators in the following statements:

```
cylinder3 = cylinder1 + cylinder2;
cylinder3 = cylinder1 - cylinder2;
if (cylinder1 > cylinder2)
```

The output of the program is given as follows:

```
Volume of cylinder1 : 392.699
Volume of cylinder2 : 502.655
Volume of cylinder3 : 3817.04
Volume of cylinder3 : -15.708
cylinder1 volume is greater than cylinder2 volume
```

In the second example, we define a Days class; it contains an enumeration type variable that can take these values: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. We want to define an incremental operation ++ that can change Mon to Tue, Tue to Wed, and Sat to Sun.

In this example, we use the following two lines of code to the prefix ++operator overloading and the postfix ++ operator overloading, respectively:

```
Days operator++()
Days operator++(int)
```

The first definition does not have a parameter to indicate that ++ will be placed before the object, and the second definition has an int type parameter to indicate that ++ will be placed after the object.

```
#include <iostream>
using namespace std;
typedef enum { Sun = 0, Mon, Tue, Wed, Thu, Fri, Sat } DayType;
class Days {
private:
```

```
  DayType day;
public:
  Days() { day = Sun; }           // constructor without parameter
  Days(DayType d) { day = d; }    // constructor with a parameter
  DayType getDay(void) { return day; }
  void setDay(DayType d) { if (d >= Sun && d <= Sat) this->day = d; }
  void display() {
      switch (day) {
      case Sun: cout << "Sun" << endl; break;
      case Mon: cout << "Mon" << endl; break;
      case Tue: cout << "Tue" << endl; break;
      case Wed: cout << "Wed" << endl; break;
      case Thu: cout << "Thu" << endl; break;
      case Fri: cout << "Fri" << endl; break;
      case Sat: cout << "Sat" << endl; break;
      default: cout << "Incorrect day" << endl;
      }
  }
  // Overload prefix ++ operator to add one to Days object: ++days.
  Days operator++() {
      Days days(day); // Save the original value
      switch (this->day) {
      case Sun: this->day = Mon; break;
      case Mon: this->day = Tue; break;
      case Tue: this->day = Wed; break;
      case Wed: this->day = Thu; break;
      case Thu: this->day = Fri; break;
      case Fri: this->day = Sat; break;
      case Sat: this->day = Sun; break;
      default: cout << "Incorrect day" << endl;
      }
      days.day = this->day; // What happens if remove this line of code?
      cout << "For debugging: the value that prefix ++ returns is: "
           << days.day << endl;
      return days; // return the value that has been increased
  }
  // Overload postfix ++ operator to add one to Days object: days++.
  Days operator++(int) {//This parameter indicates ++ follows a parameter
      Days days(day);   // Save the original value
      switch (this->day) {
      case Sun: this->day = Mon; break;
      case Mon: this->day = Tue; break;
      case Tue: this->day = Wed; break;
      case Wed: this->day = Thu; break;
```

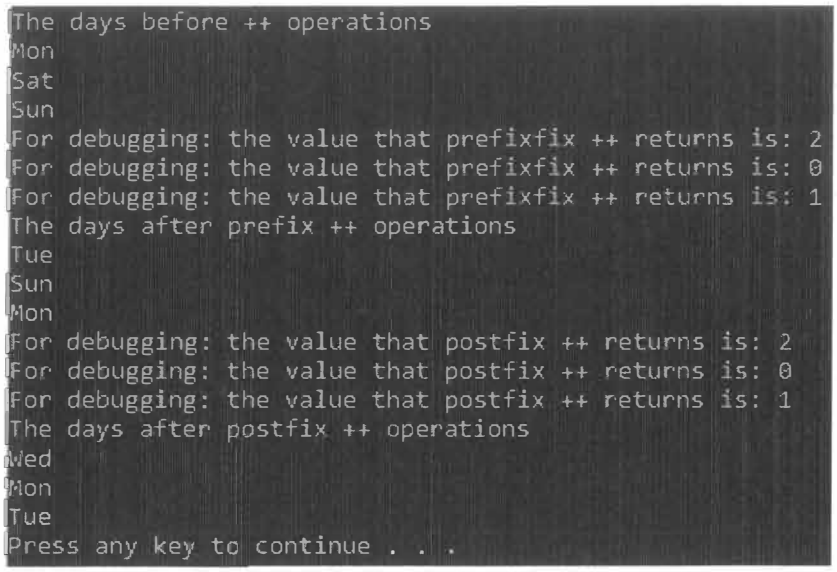
```
        case Thu: this->day = Fri; break;
        case Fri: this->day = Sat; break;
        case Sat: this->day = Sun; break;
        default: cout << "Incorrect day" << endl;
        } // The value in the this object has been changed.
        cout << "For debugging: the value that postfix ++ returns is: "
             << days.day << endl;
        return days; // return the value before the changes.
    }
};
int main() {
    Days day1(Mon), day2, day3;
    day2.setDay(Sat);
    day3.setDay(Sun);
    cout << "The days before ++ operations" << endl;
    day1.display();
    day2.display();
    day3.display();
    ++day1; ++day2; ++day3;
    cout << "The days after prefix ++ operations" << endl;
    day1.display();
    day2.display();
    day3.display();
    day1++; day2++; day3++;
    cout << "The days after postfix ++ operations" << endl;
    day1.display();
    day2.display();
    day3.display();
    return 0;
}
```

Notice the difference in the definition of the prefix increment and postfix increment. The difference is that the prefix increment returns the modified object, while the postfix increment returns the object before being changed. In both operations, the objects are changed.

In the main program, we use the overloaded operators, prefix increment and postfix increment, respectively, in the following statements:

```
++day1; ++day2; ++day3;
day1++; day2++; day3++;
```

The output of the program is given as follows. Notice that Sun is changed to Mon after the increment, which is exactly what we want.



Debugging information is also printed just before return statements in the two ++ operators' definition. This debugging information shows that return values in the prefix increment operator definition and values printed after by calling the display() functions are the same, because the values are pre-incremented. However, the return values in the postfix increment operator definition and values printed after by calling the display() functions are different, because the return values are not incremented yet in the operator definition. The printout also shows that the enumeration values are print in integers if we directly print the values. In order to print the word values, we need to use a switch statement to map the integers their word values.

Table B.4 in the Appendix B gives a complete list of the C/C++ operators, their precedence, description, and associativity. The operators that cannot be overloaded are also indicated in the description part of the table.

3.6 File Operations in C++

In Chapter 2, we discussed basic concepts of standard input/output, files, and file operations in C. Please review Section 2.5 on basic concepts discussion which is applicable to C++ file operations. In this section, we discuss the C++ specific file operations.

3.6.1 File objects and operations in C++

C++ provides a different set of stream classes to perform input/output and file operations.

- ofstream: Stream class for writing on files.
- ifstream: Stream class for reading from files.
- fstream: Stream class for both reading from and writing to files.

These classes are derived from the library classes istream and ostream. As standard input and output, we have used their objects cin and cout: cin is an object of class istream and cout is an object of class ostream. These classes include member functions open, is_open, and close. In this section, we will use these classes for general file read and write operations. The only difference with the standard input and output is that, instead of using the default file names stdin and stdout, we will specify a different file name for reading or

writing. Like C file operations, you can also specify the options for different types of read and write. Table 3.4 shows the available operations.

ios::in	Open file for input operations. Not required of ifstream class is used.
ios::out	Open for output operations. Not required of ofstream class is used.
ios::binary	Open in binary mode.
ios::ate	Set the cursor position at the end of the file for reading or writing.
ios::app	The write operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

Table 3.4. Options for C++ file operations

The following program shows a part of a C++ linked list program that saves the linked list into file before exiting, loads the data from file, and creates a linked list at starting.

```
#include <iostream>
#include <fstream> // include ifstream and ofstream classes
#include <string>
using namespace std;
class Contact {
  public:   string name;
            int phone;
            Contact *next;
};
Contact *head = NULL;
int save_file(string myFileName); // myFileName is "SavedList"
void load_file(string myFileName); // myFileName is "SavedList"

void load_file(string myFileName)  { // myFileName is "contactFile"
  int count = 0;
  Contact * temp = head;  // head is global
  ifstream myFileVar; // declare a variable of File type
  myFileVar.open (myFileName); // open the same file name
  if(myFileVar.is_open())  {
      myFileVar >> count;  // read count from file
      for (int i = 0; i < count; i++) {
           Contact* temp= new Contact();
           myFileVar >> temp->name;
           myFileVar >> temp->phone;
           temp->next = head;
           head = temp;
      }
      myFileVar.close();
  }
```

```
}
// Note, save_file does not delete the linked list when saving into file
int save_file(string myFileName)  { // myFileName is "contactFile"
   int count = 0;
   Contact *temp = head;  // head is global
   ofstream myFileVar; // declare a variable of File type
   while (temp != NULL) // count number of Contact nodes in linked list {
        temp = temp->next; count++;
   }
   myFileVar.open(myFileName);
   if(myFileVar.is_open())  {
        myFileVar << count;  // save count into file
        while (temp != NULL) {
             myFileVar << temp->name << endl;
             myFileVar << temp->phone << endl;
             temp = temp->next;
        }
        myFileVar.close();
        return 0;
   }
}
void main() {
   string myFile = "SavedList";
   load_file(myFile);
   // Perform other functions
   save_file(myFile);
   deleteList(head); // delete all notes in a loop
}
```

Figure 3.10 illustrates how the linked list data is saved into the file. We do not need to save the next member. It will be created in load_file function.

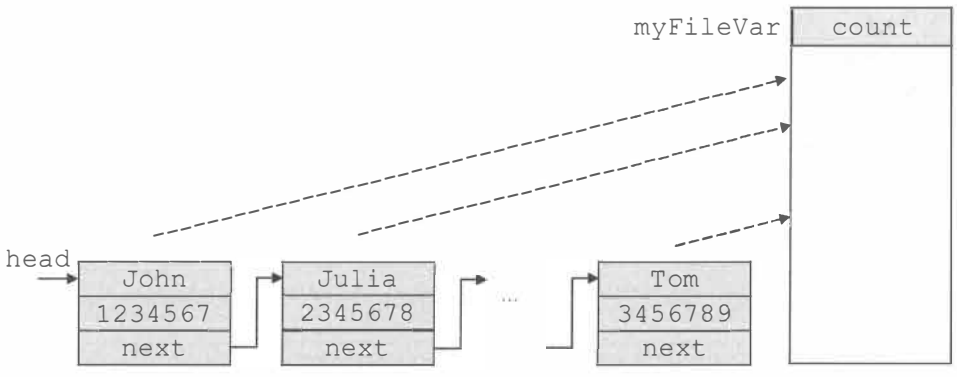


Figure 3.10. Different objects are linked into the same list.

Similar to C file I/O and file operations, there are different ways of reading and writing files in C++. The following program example shows a more complex example of writing a structure of data consisting of first name, last name, phone, and email into a .csv (Comma Separated Value) file, which is one of the data formats of Excel. To illustrate the different ways available, we will use a different way of reading and writing the .csv file. Table 3.5 shows a .csv file entered using the writingFile function in the program. In the program, phone is considered as an integer and processed as an integer.

First	Last	Phone	Email
David	Smith	3456789	david@gmail.com
Mary	Miller	7654321	mary@yahoo.com
Jane	Jones	9650000	jane@live.com
John	Lee	1234567	john@asu.edu

Table 3.5. Example of .csv file generated by the program.

```
#include <iostream>
#include <string>
#include <fstream>   // for file reading and writing functions.
using namespace std;
void readingFile();  // Forward Declarations
void writingFile();  // Forward Declarations
ofstream wFile;
int main() {
  char choice;
  do {
       cout << "\nSelect a function" << endl;
       cout << "(w) For writing file" << endl;
       cout << "(r) For reading file that exist" << endl;
       cout << "(q) For quitting" << endl;
       choice = getc(stdin);
       cout << endl;
       if (choice == 'r')
            readingFile();
       else if (choice == 'w')
            writingFile();
       else{
            cout << "Invalid input\n" << endl;
            cin.ignore();
       }
  } while (choice != 'q');
  return 0;
}
void writingFile() { // This function writes into a file
  // Declare object to read file of Comma Separated Value csv file
  wFile.open("person.csv", ios::out|ios::app);
```

```
  // Open for writing and appending
  // These strings are declared to store the input data.
  string firstName, lastName, phone, email;
  int iPhone;
  cout << "Please enter first name, last name, phone, and email" << endl;
  cin >> firstName;
  cin >> lastName;
  cin >> phone;
  cin >> email;
  wFile << firstName << ',' << lastName; // write in the file
  if ((phone[0] > 47) && (phone[0] < 58)) {
  // check if the first char is a digit between 0 and 9
      iPhone = stoi(phone); // convert string to integer
      wFile << "," << iPhone;
  }
  else
      wFile << "," << phone;
  wFile << "," << email << endl;
  wFile.close();// Always close a file when done.
  cin.ignore(); // Remove delimeter at the end
};
void readingFile() { // This function uses getline to read from the file
  // Declare object to read file.
  ifstream rFile("person.csv");// Comma Separated Value file
  // These strings are declared to store the parsed data.
  string firstName, lastName, phone, email;
  int iPhone;
  if (rFile.is_open()) // Open the file{
      // Keep reading until the end of the file.
      while (getline(rFile, firstName, ',')) {
          getline(rFile, lastName, ',');
          getline(rFile, phone, ',');
          getline(rFile, email, '\n');
          cout << firstName << '\t' << lastName;
          if ((phone[0] > 47) && (phone[0] < 58)) {
              // If not header, convert string to integer
              iPhone = stoi(phone);
              cout << "\t" << iPhone;
          }
          else cout << "\t" << phone;
          cout << "\t" << email << endl;
      }
      rFile.close();// Always close a file when done.
  }
```

```
    else cout << "Unable to open file" << endl;
    cin.ignore(); //   Remove delimeter at the end
};
```

3.6.2 Ignore operation in C++

In Chapter 2, we used `fflush(stdin)` to remove the delimiter (a space, a newline, etc.) when we switch from `scanf` to `getc`. The C++ function equivalent to `fflush` is `cin.ignore`.

The function `cin.ignore()` is similar to but more powerful than the C-styled `fflush(stdin)` function that flushes the input buffer to remove the remaining delimiters in the buffer of the standard input file `stdin` after a `scanf` operation. In C++, you must use `cin.ignore()` if you switch from the formatted input function `cin >>` to an unformatted input function such as `cin.get` or `cin.getline`, etc. Similar to `scanf`, `cin >>` will read only up to a space or newline and leave the space or newline in the buffer. The function `cin.ignore()` will remove the space or newline. The function `cin.ignore()` is more powerful than simply removing one character from the input buffer. There are three overloaded functions. (1) `cin.ignore()`: discard one character from the input buffer; (2) `cin.ignore(int n)`: discard `n` characters from the input buffer; and (3) `cin.ignore(int n, char term)`: discard `n` characters or stop when the character in the parameter `term` is encountered. For example,

```
#include <iostream>
using namespace std;
void main(void) {
   char strvar[12];
   cin >> strvar;              // Enter: Hi
   cout << "Please enter the string: Hello world" << endl;
   // cin.ignore();            // option 1
   // cin.ignore(10);          // option 2
   // cin.ignore(10, 'w' );    // option 3
   cin.getline(strvar, 12, '\n'); // Enter: Hello world
   cout << strvar << endl;
}
```

Figure 3.11 illustrates the states of the input variable `strvar` and the input buffer in the execution process of the program above. Notice that the newline character '`\n`' is left in the buffer after the `cin >>` operation is completed and thus the next input string "`Hello world`" is appended to the character. The `cin.getline` function reads the input buffer until it encounters a newline character. Since a newline character already exists in the buffer, the `cin.getline` function will not wait for the user to enter a newline. As a result, no input is needed and an empty string is read into the input variable `strvar`. The print statement `cout <<` will then print nothing. It creates an illusion that the last two statements in the program are skipped.

To solve the problem, we can use the function `cin.ignore()` to discard the newline character (option 1 in the program). To illustrate the effects of the three `cin.ignore` functions, we apply the three options one by one. The outputs of the statement "`cout << strvar << endl;`" in the three executions will be, respectively:

```
Hello world          // correctly printed
ld                   // 10 characters are discarded
```

orld // It stops after the character 'w' is read

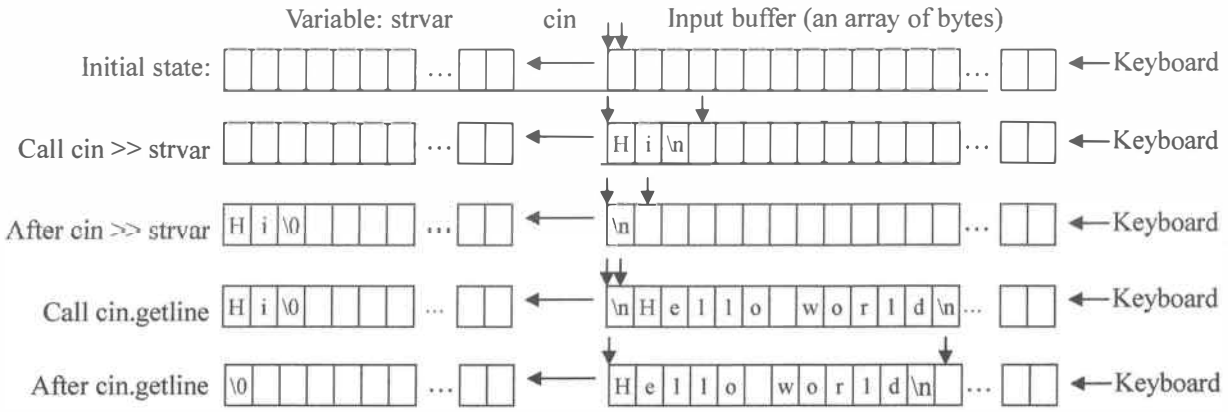


Figure 3.11. The input buffer between the keyboard and the input variable.

Option 1 removes a single character. Option 2 removes 10 characters. Option 3 removes 10 characters or stops when a character 'w' is encountered, whichever comes first, and thus 8 characters are removed. Please notice that `cin.ignore` removes the characters left in the input buffer. If there is no character in the buffer, it will remove the character entered after the `cin.ignore` statement!

Functions `cin.get` and `cin.getline` work like the `cin >>` except that they will read a line with `spaces`, whereas the parameter `strvar` is a string variable that will hold the entered string; parameters `strlength` and `achar` are optional. Parameter `strlength` is an integer that limits the maximum number of characters to be passed to the parameter `strvar`. Parameter `achar` is a character that serves as the terminator or delimiter. The input function stops reading when the terminator character is read. By default, the terminator is the `newline` character '\n'. Both `cin.get` and `cin.getline` will reserve one character for the terminator, that is, you can enter only `strlength-1` characters. The only difference between `cin.get` and `cin.getline` is that the `cin.getline` function will remove the terminator from the `strvar` variable, while `cin.get` will keep the terminator in the `strvar` variable. The following code shows a simple example where the character '@' is used as the terminator:

```
#include <iostream>
using namespace std;
void main() {
  char aline[25];
  cout << "Please enter a line terminated by '@'" << endl;
  cin.getline(aline, 25, '@');
  cout << aline << endl;
}
```

3.7 Exception Handling

An **exception** is a forced deviation caused by a known event, which represents an abnormal situation from the normal execution sequence of the program. There are **internal exceptions** and **external exceptions**.

An internal exception is caused by a message to the CPU seeking attention from a source within the CPU itself, for example, when an operation performs a division by 0, causes an overflow, or executes an illegal (undefined) operation. Internal exceptions are called software-related exceptions.

An external exception is caused by a message to the CPU seeking attention from a source outside the CPU (e.g., out-of-memory, memory access violation, bus error, device busy, etc.). External exceptions are also called **interrupts**.

Exceptions are difficult to handle and are normally handled at all levels of a computer system.

At the hardware level, when an exception occurs, the hardware will identify the exception source, compute the exception handler's entry address, and load the address into the program counter of the CPU. Thus, the CPU starts to execute the exception handler's code. A very limited number of exceptions can be handled at the hardware level. For example, a Motorola 68000 processor can handle up to 256 exceptions and interrupts. At the OS level, more exceptions can be handled. For each exception, a simple exception handler will be provided. At the program language level, most program languages provide exception-handling mechanisms. For example, C++ provides several exception classes to facilitate programmers in handling exceptions more effectively. At the user program level, a programmer can write application-specific exception handlers to handle various semantics-related exceptions. Only the programmers know the semantics of their programs.

At each level, exception handling can make use of the exception handlers below and can add extra exception handlers.

If an exception is not caught by the programmer's handlers, there are two possibilities: (1) The exception is caught by a lower level of exception handler and an error message will normally be shown. The programmer can either terminate the program or enter the debugging state to see what instruction or operation caused the exception. (2) If the exception is not caught by a lower level exception handler, the program will normally crash or freeze.

Visual Studio C++ provides an exception library <stdexcept> that includes the following exception classes:

```
exception Class (root class)
domain_error Class
invalid_argument Class
length_error Class
logic_error Class
out_of_range Class
overflow_error Class
range_error Class
runtime_error Class
underflow_error Class
```

C++ also provides several constructs for programmers to define their exception conditions and exception handlers. The syntax, which is similar to that of Java, is given in BNF notations as follows:

```
<exception-structure> ::= try<code-block><handler-list>
<handler-list> ::= <empty> | <handler> | <handler-list><handler>
<handler> ::= catch (<except-declaration>) <code-block>
<except-declaration> ::= <type-name> | <type-name><identifier> |
                         <type-name> *<identifier>
<throw-statement> ::= throw | throw<expression>
```

The code-block following the `try` keyword is the code that is a part of the code required by the semantics of the program. However, an exception condition may occur in this part of the code. For example, there is a division operation on a variable whose value could be zero, or a memory request is made, which may or may not receive the required memory.

The handler-list may consist of zero, one, or multiple handlers, each of which handles a different type of exception variable. Each handler starts with the keyword `catch` and is followed by declaring an exception variable and a block of code. The variable will be used to receive the "return" value of a `throw` statement in the `try` statement. The code-block in the `catch` statement will handle the exception in a specific way (e.g., print an error message of the value of the exception variable).

The `throw` statement is similar to a `return` statement. It is normally used in the `try` statement to exit (return from) the block and possibly to pass a value to an exception variable. Multiple `throw` statements can be used. If the types of the return values are different, different exception handlers (multiple `catch` statements) must be used. We can also consider that `catch` is a function, and a `throw` statement is a function call to a `catch` function. Since there can be multiple `catch` statements, `catch` is an overloaded function with different parameter types. The value in the `throw` statement will be parameter-passed to the `catch` function.

The following program illustrates the application of the exception statements:

```
#include <iostream>
using namespace std; // It includes <stdexcept> library
int main() {
  int *queue, n;
  cout << "Enter queue-size >= 10: " << '\n';
  cin >> n;
  try {
      if (n < 10)
           throw -1;    // return an integer value
      queue = new int[n];
           if( queue == 0 )
               throw "heap allocation failed!";
  }
  catch( char * se ) {
      cout << "Exception: " << se << '\n'; // return a string
  }
  catch( int ie ) {
      cout << "Exception: " << ie << " too small" << '\n';
  }
  // ...
  return 0;
}
```

In this program, there are two `throw` statements, handling two different exception situations, respectively. One returns an integer value and the other returns a string. Thus, two `catch` statements (two exception handlers) are necessary. The integer return value will be passed to the integer exception variable `ie`, and the string return value will be passed to the string variable `se`.

Although we can use the normal data types to declare exception variables, it is recommended, for the purpose of readability, to define an exception class (type) and use the class to declare exception variables. The following piece of code checks if a date entered is in a valid format and in the valid ranges. A class `DataErr` is defined and used to declare an exception variable `derr`. The variable is assigned values in the `try` statement. In the `catch` statement, another variable `dateerr` of `DataErr` type is declared. When a `throw` statement calls the `catch` function, the values in the variable `derr` are passed to the variable `DataErr`.

```
class DateErr {
  public:
      char *pdate;
      int idate;
} derr;
char birthday[11], atemp[11], *ptemp;
cout << "Enter birthday, e.g., 05/24/1985";
try {  // handling incorrect date input
  cin.getline(birthday, 11);
  strcpy(atemp, birthday); // 05/24/1985
  ptemp = atemp;
  derr.pdate = birthday;
  atemp[2] = '\0'; //extract the month
  // handling incorrect date input
  derr.idate = atoi(ptemp); // extract month
  if ((derr.idate<1)||(derr.idate>12)) {
      throw derr;    // out of month range 1..12
  }
  atemp[5] = '\0';
  ptemp = ptemp + 3;
  derr.idate = atoi(ptemp);  //extract day
  if ((derr.idate<1)||(derr.idate>31))
      throw derr;    // out of day range 1..31
  ptemp = ptemp + 3; //extract the year
  derr.idate = atoi(ptemp);
  if ((derr.idate<1800) || (derr.idate>2050))
      throw derr;
}
catch (DateErr dateerr) {
  cout << "Exception: "
      << dateerr.idate << "incorrect in"
      << dateerr.pdate << '\n';
}
```

3.8 Case Study: Putting All Together

In this section, we give an example that applies many of the data structures and programming techniques learned in this chapter, including

- Class definition: with private members and public members. A friend function is also defined in the Pet class, which allows the changeBreed function outside the class to access this class's members.
- Inheritance and inheritance hierarchy: among the Pet, Cat, Dog, and Fish classes. The Fish class has an additional member that does not exist in the Pet class.
- Containment: between the Container and Pet classes. The Cat, Dog, and Fish classes can be added into the Container class through polymorphic pointer of Pet.
- Type casting: Explicitly convert int type to enumeration type
- Memory management and garbage collection: heap memory is deleted in the remove function.
- Header files and multiple classes and organization of the program: Each class is split into a header file and a source file.
- File operations:

3.8.1 Organization of the program

The program consists of five classes: Pet, Cat, Dog, Fish, and a Container. The Container class contains a pointer to a Pet object and a pointer to the next Container object (self-containment), forming a linked list of Pet, Cat, Dog, and Fish objects. The classes and their relationships among them are shown in Figure 3.12. Each class is split into a header (.h) file and a source code (.cpp) file. The header files are stored in the "Header Files" folder, and the source files are stored in the "Source Files" folder, as shown in the Visual Studio Solution Explorer at the right-hand side of Figure 3.12. The header files are used to store the class definitions, while the source files are used to store the implementation. All the methods in the source files use scope resolution operators to associate the implementation with the class definition. Please also read Section 2.9 on module design of large programs and the scope of variables and functions.

A virtual method is defined in each class to display all the members in each class, which allows the polymorphic to call the display() functions differently implemented in each class. All the data members are defined as private members, and a get-method is defined for each data member.

Linked list manipulation functions are defined, including add, remove, and modify. A save and a load functions are defined to save to linked list data into a text file before the program exits and to load the data into the linked list at program start.

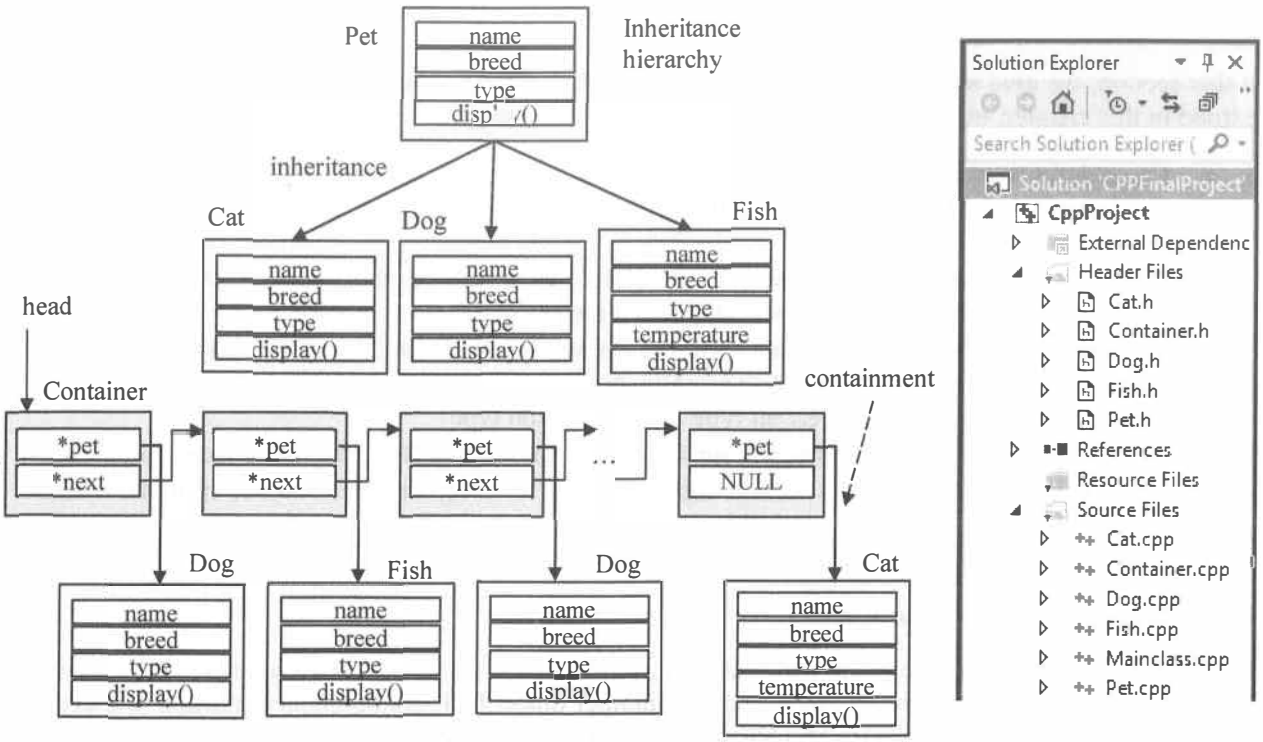


Figure 3.12. Classes and their relationships.

3.8.2 Header files

This section presents all the header files in the program. The definitions are explained through comments. These files should be added into the Header Files folder in the Visual Studio project.

Pet.h

```
#ifndef _PET_H_
#define _PET_H_
// These two preprocessor directives and the one at the end: #endif,
// prevent the header file from being included (linked in) multiple
// times, when it is used multiple times by the user.
#include <string>
using namespace std;
enum PetType { dog = 0, cat, fish };
class Pet {
private:
  string name, breed; // private local variables
  PetType type;
public:
  Pet(string pet_name, string pet_breed, PetType pet_type); // constructor
  string getName();  // accessor methods
  string getBreed();
  PetType getType();
  friend void changeBreed(Pet* pet, string breed);
```

```
  // This friend definition allows changeBreed() to access this class
  virtual void display(){ }
};
#endif // _PET_H_
```

Cat.h

```
#ifndef _CAT_H_
#define _CAT_H_
#include "Pet.h"
class Cat : public Pet {
public:
  Cat(string pet_name, string pet_breed, PetType type) : Pet(pet_name,
pet_breed, type){}
  void display();
};
#endif // _CAT_H_
```

Dog.h

```
#ifndef _DOG_H_
#define _DOG_H_
#include "Pet.h"
class Dog : public Pet {
public:
  Dog(string pet_name, string pet_breed, PetType type) : Pet(pet_name,
pet_breed, type){}
  void display();
};
#endif // _DOG_H_
```

Fish.h

```
#ifndef _FISH_H_
#define _FISH_H_
#include "Pet.h"
class Fish : public Pet {
private:
  int temperature;
public:
  Fish(string pet_name, string pet_breed, PetType type, int temp) :
Pet(pet_name, pet_breed, type) {
      temperature = temp;
  }
  int getTemperature();
  void display();
};
#endif // _Fish_H_
```

Container.h

```
#ifndef _CONTAINER_H_
#define _CONTAINER_H_
#include "Pet.h"
class Container {
public:
  Pet *pet;
  Container *next;
  Container(); // constructor
};
#endif // _CONTAINER_H_
```

3.8.3 Source files

This section presents all the C++ source files in the program. The classes and functions are explained through comments. These files should be added into the Source Files folder in the Visual Studio project.

Mainclass.cpp

```
// This example put together many concepts discussed in this Chapter
#include "Container.h"
#include "Pet.h"
#include "Dog.h"
#include "Cat.h"
#include "Fish.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
// forward declarations, listing all functions defined in this class
void flush();
void branching(char);
void helper(char);
void add_pet(string, string, PetType);
Pet* search_pet(string, string, PetType);
void remove_pet(string, string, PetType);
void remove_all();
void print_all();
void save(string); // Save linked list into a text file before exiting
void load(string); // Load the save data back in the linked list upon
starting
Container* head = NULL; // global list head
int main() {     // main class
  _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
  // Use for checking memory leaks in VS
  load("Pets.txt");  // Load the saved data back into the linked list
  char ch = 'i';
```

```
  do {// Print the selection menu
       cout << "Please enter your selection" << endl;
       cout << "\ta: add a new pet to the list" << endl;
       cout << "\tc: change the breed of a pet" << endl;
       cout << "\tr: remove a pet from the list" << endl;
       cout << "\tp: print all pets on the list" << endl;
       cout << "\tq: quit and save list of pets" << endl;
       cin >> ch;
       flush();
       branching(ch); // Jump to the selected function
  } while (ch != 'q');
  save("Pets.txt");  // Save linked list data into a text file
  remove_all();      // garbage collect all the memory
  head = NULL;
  return 0;
}
void flush() {   // Flush file buffer
  int c;
  do c = getchar(); while (c != '\n' && c != EOF);
}
void branching(char c) { // Jump to the select function
  switch (c) {  // Cases a, c, r, and p are differentiated in helper()
  case 'a': // no break, it falls through
  case 'c':
  case 'r':
  case 'p':
       helper(c);
       break;
  case 'q':
       break;
  default:
       printf("\nInvalid input!\n\n");
  }
}
// The helper function is used to determine what data is needed and which
function to send that data to.
// It uses pointers and values that are returned from some functions to
produce the correct output.
// Study this function and know how it works.
// It is always helpful to understand how the code works before
implementing new features.
void helper(char c) {
  string name, breed;
  PetType type;
```

```
    int type_check = -1;
    if (c == 'p')
        print_all();  // Print all members' information
    else {
        cout << endl << "Please enter the pet's name: " << endl;
        cin >> name;
        cout << "Please enter the pet's breed: " << endl;
        cin >> breed;
        while (!(type_check == 0 || type_check == 1 || type_check == 2))
        {
            cout << endl << "Please select one of the following: " << endl;
            cout << "0. Dog " << endl;
            cout << "1. Cat" << endl;
            cout << "2. Fish" << endl;
            cin >> type_check; // Enter 0, 1, or 2
        }
        type = (PetType)type_check;   // Cast int to PetType
        Pet* pet_result = search_pet(name, breed, type);
        if (c == 'a') { // Call add_pet function
            if (pet_result == NULL) {
                add_pet(name, breed, type);
                cout << endl << "Pet added." << endl << endl;
            }
            else
                cout << endl << "Pet already on list." << endl << endl;
        }
        else if (c == 'c') // change pet breed
        {
            if (pet_result == NULL) {
                cout << endl << "Pet not found." << endl << endl;
                return;
            }
            cout << endl << "Please enter the new breed for this pet: " <<
endl;
            cin >> breed; flush();
            changeBreed(pet_result, breed);
            cout << endl << "Pet's breed changed." << endl << endl;
        }
        else if (c == 'r') { // call remove_pet function
            if (pet_result == NULL) {
                cout << endl << "Pet not found." << endl << endl;
                return;
            }
            remove_pet(name, breed, type);
```

```
            cout << endl << "Pet removed from the list." << endl << endl;
        }
    }
}
void changeBreed(Pet* pet, string breed){
    pet->breed = breed;
}
// This function will be used to add a new pet to the tail of the global
linked list.
// We use the enum 'type' variable to determine which constructor to use.
// Notice that that search is called before this function to make sure the
new pet is not on the list.
void add_pet(string name, string breed, PetType type) {
    Container* new_container = new Container();
    if (type == dog)
        new_container->pet = new Dog(name, breed, type);
    else if (type == cat)
        new_container->pet = new Cat(name, breed, type);
    else if (type == fish) {
        int temp; // For Fish object, an additional parameter is needed
        cout << "Please enter an integer for the Preferred Temperature of
the fish breed";
        cin >> temp;
        new_container->pet = new Fish(name, breed, type, temp);
    }
    new_container->next = NULL;
    if (head == NULL) {    // If the linked list is empty
        head = new_container;
        return;
    }
    Container* container_traverser = head->next;
    Container* container_follower = head;
    // Find the end of the linked list and add the new object at the end
    while (container_traverser != NULL) {
        container_follower = container_traverser;
        container_traverser = container_traverser->next;
    }
    container_follower->next = new_container;
}
// Search linked list and return the pointer to the object
// that meet the criteria.
Pet* search_pet(string name, string breed, PetType type) {
    Container* container_traverser = head;
    while (container_traverser != NULL) {
```

```
        if (container_traverser->pet->getName() == name
            && container_traverser->pet->getBreed() == breed
            && container_traverser->pet->getType() == type)
            return container_traverser->pet;
        container_traverser = container_traverser->next;
    }
    return NULL;
}
// Remove one pet
void remove_pet(string name, string breed, PetType type) {
    Container* to_be_removed;
    if (head->pet->getName() == name
        && head->pet->getBreed() == breed
        && head->pet->getType() == type) {
        to_be_removed = head;
        head = head->next;
        delete to_be_removed->pet;
        delete to_be_removed;
        return;
    }
    Container* container_traverser = head->next;
    Container* container_follower = head;
    while (container_traverser != NULL) {
        if (container_traverser->pet->getName() == name
            && container_traverser->pet->getBreed() == breed
            && container_traverser->pet->getType() == type) {
            to_be_removed = container_traverser;
            container_traverser = container_traverser->next;
            container_follower->next = container_traverser;
            delete to_be_removed->pet;
            delete to_be_removed;
            return;
        }
        container_follower = container_traverser;
        container_traverser = container_traverser->next;
    }
}
// Remove all nodes in the linked list
void remove_all() {
    while (head != NULL) {
        Container* temp = head;
        head = head->next;
        delete temp->pet;
        delete temp;
```

```
    }
  }
  // This function uses the virtual display() method of the Dog,  Cat and
  Fish classes to print all Pets in an organized format.
  void print_all() {
    Container *container_traverser = head;
    if (head == NULL)
        cout << endl << "List is empty!" << endl << endl;
    while (container_traverser != NULL) {
        container_traverser->pet->display();
        container_traverser = container_traverser->next;
    }
  }
  // Save the linked list of pets to a file using ofstream.
  // We first count the number of nodes in the linked list.
  // We cast the enum 'type' to an int before writing it to the file.
  void save(string fileName) {
    int count = 0;
    Container* container_traverser = head;
    // count number of Containers in linked list
    while (container_traverser != NULL)  {
        container_traverser = container_traverser->next;
        count++;
    }
    ofstream myfile;
    myfile.open(fileName);
    if (myfile.is_open()) {
        container_traverser = head;
        myfile << count;
        while (container_traverser != NULL) {
            myfile << container_traverser->pet->getName() << endl;
            myfile << container_traverser->pet->getBreed() << endl;
            myfile << (int)container_traverser->pet->getType() << endl;
            container_traverser = container_traverser->next;
        }
        myfile.close();
    }
  }
  // Load the linked list of pets from a file using ifstream.
  // We create the linked list in the same order that is was saved to a file.
  // We create a new node for the linked list, then add it to the tail of the
  // list. We cast the int type back to the enum 'type'.
  // We use the 'type' variable read from the file to determine which
  constructor to use.
```

```
void load(string fileName) {
  ifstream myfile;
  myfile.open(fileName);
  if (myfile.is_open()) {
      int type_as_int, count = 0;
      string name, breed;
      PetType type;
      Container* container_traverser = head;
      myfile >> count;
      for (int i = 0; i < count; i++) {
          Container* new_node = new Container();
          myfile >> name;
          myfile >> breed;
          myfile >> type_as_int;
          type = (PetType)type_as_int;
          if (type == dog)
              new_node->pet = new Dog(name, breed, type);
          else
              new_node->pet = new Cat(name, breed, type);
          new_node->next = NULL;
          if (head == NULL) {
              new_node->next = head;
              head = new_node;
          }
          else {
              container_traverser = head;
              while (container_traverser->next != NULL)
                  container_traverser = container_traverser->next;
              container_traverser->next = new_node;
          }
      }
      myfile.close();
  }
}
```

Pet.cpp

```
#include "Pet.h"
Pet::Pet(string pet_name, string pet_breed, PetType pet_type) {
  name = pet_name;
  breed = pet_breed;
  type = pet_type;
}
string Pet::getName() {
  return name;
```

```
}
string Pet::getBreed() {
  return breed;
}
PetType Pet::getType() {
  return type;
}
```

Cat.cpp

```
#include "Pet.h"
#include "Cat.h"
#include <iostream>
void Cat::display() {
  cout << endl << "Name: " << getName() << endl;
  cout << "Breed: " << getBreed() << endl;
  cout << "PetType: Cat" << endl << endl;
}
```

Dog.cpp

```
#include "Pet.h"
#include "Dog.h"
#include <iostream>
void Dog::display() {
  cout << endl << "Name: " << getName() << endl;
  cout << "Breed: " << getBreed() << endl;
  cout << "PetType: Dog" << endl << endl;
}
```

Fish.cpp

```
#include "Pet.h"
#include "Fish.h"
#include <iostream>
int Fish::getTemperature() {
  return temperature;
};
void Fish::display() {
  cout << endl << "Name: " << getName() << endl;
  cout << "Breed: " << getBreed() << endl;
  cout << "PetType: Fish" << endl;
  cout << "Preferred Temperature: " << getTemperature() << endl << endl;
}
```

Container.cpp

```
#include "Container.h"
// Constructor for Container class
Container::Container() {
```

```
    pet = NULL;
    next = NULL;
}
```

*3.9 Parallel computing and multithreading

Most computers today have multiple processors. Writing programs using the parallel computing and multithreading paradigms is a necessity in modern software development.

3.9.1 Basic concepts in parallel computing and multithreading

Multitasking in OS and multithreading in application programs are similar. Both provide the ability to execute different parts of a code simultaneously, while maintaining the correct results of computation.

In the OS case, the parts of the code executed in parallel are called **processes** or **tasks** and are often semantically independent of each other (but can be related). OS performs process scheduling and resource (processors, memory, peripherals, etc.) allocation. OS system calls allow users to create, manage, and synchronize processes. In the case of the application program, the parts of the code executed in parallel are called **threads**. Most often, they are semantically dependent (but they can be independent).

In both cases, programmers must carefully design the OS/application in such a way that all the processes/threads can run at the same time without interfering with each other, and produce the same result in a finite amount of time regardless of the order in which they are executed.

We differentiate a program from a process and a function (method) from a thread. A program or a function (method) is a piece of code written by a programmer that is static, while a process or thread consists of an executing program/function, its current values, state information, and the resources used for supporting its execution. In other words, a process or thread is a dynamic entity, which exists only when a program or a method is being executed.

To execute multiple processes/threads truly in parallel, multiple processors must exist. If a system has only one processor, multiple processes/threads appear to execute simultaneously, but actually execute sequentially in time-sharing mode.

From the same piece of code, multiple processes/threads can be created. The code and data contained within different processes/threads are, by default, separated: each has its own copy of executable code, its own stack for local variables, and its own data area for objects and other data elements.

C++ was not initially designed to handle parallel software development. The language must be extended in its environment to handle such tasks. In this section, we will discuss how Visual Studio extends C++ to facilitate multithreading.

3.9.2 Generic features in C++

Generic types are parameterized types [Source: http://msdn.microsoft.com/en-us/library/c570k3f3.aspx]. They are defined with an unknown type parameter that is specified using the generic keyword:

```
generic <typeName T>
```

A type constructed from a generic type is referred to as a **constructed type**, which is a type not fully specified. For example, List<T> is an open constructed type. Type T will be determined at execution time. On the other hand, a type that is fully specified, such as List<double>, is a **closed constructed type** or a specialized type.

Based on the generic type definition, a **generic function** can be defined, which is a function that contains generic type parameters. When being called, actual types are used instead of the type parameters [http://msdn.microsoft.com/en-us/library/skef48fy.aspx].

```
generic <typeName T1, typeName T2 >
T1 foo<T1 x, T2 y> {
   function-body
}
```

In this example, generic types T1 and T2 are used as the parameter of the function `foo`.

Furthermore, we can define a **generic class**. A generic class is a template that can be instantiated by different actual classes.

It can be defined using generic type parameters and generic functions. It can also extend an interface class [http://msdn.microsoft.com/en-us/library/skef48fy.aspx]. For example,

```
generic <typeName T1, typeName T2 >
ref class gStack {
   private T1 *buffer;
   void push(T2 item);
   T2 pop();
}
```

A generic class can have constraints, which limit the types to be used for the type parameter. For example,

```
interface class IcItem { };
generic <class ItemType> where ItemType : IcItem
ref class gStack {};
```

A full example is given below to illustrate the concepts and the syntax of defining generic class, function, and type [http://msdn.microsoft.com/en-us/library/a174071k.aspx].

```
using namespace System;
interface class IAge {  int Age();  };
ref class Senior : IAge {
   public: virtual int Age() { return 70; }
};
ref class Adult: IAge {
   public: virtual int Age() { return 30; }
};
ref class MyClass {
public:
   generic <class ItemType>
   where ItemType : IAge          // could include a list of constraints
   bool isSenior(ItemType item)   // Because of the constraint,
   {        // the Age method can be called on ItemType.
      if (item->Age() >= 65) return true;
      else return false;
   }
};
```

```
int main() {
   MyClass^ ageGuess = gcnew MyClass(); // g.c. - new
   Adult^ parent = gcnew Adult();
   Senior^ grandfather = gcnew Senior();
   if (ageGuess->isSenior<Adult^>(parent))
      Console::WriteLine("\"parent\" is a senior");
   else
      Console::WriteLine("\"parent\" is not a senior");
   if (ageGuess->isSenior<Senior^>(grandfather))
      Console::WriteLine("\"grandfather\" is a senior");
   else
      Console::WriteLine("\"grandfather\" is not a senior");
}
```

The syntax for using a generic class is different from that for using a normal class. When using a generic class as a type to define a variable, we need to append the symbol ^ to the class name, for example, `MyClass^`, `Adult^`, and `senior^`. For accessing the member of an object, arrow notation -> will be used, for example, `ageGuess->isSenior`. For creating an object, `gcnew` will be used, while `new` is used to create a class instance, for example, `Adult^ parent = gcnew Adult()`.

3.9.3 Case Study: Implementing multithreading in C++

After being introduced to the generic class, we can start to implement a **multithreading** program in C++. We will use a simple problem as our case study, so that we can focus on the creation of a multithreading program. The problem we will solve is to add a large array of numbers. Assume the size of the array is 1,000,000. Without applying multithreading, we would use a for-loop that iterates a million time to add the numbers. Now, we assume that we have p processors, and we can split the array into p arrays, and let one processor add 1/p of the numbers. After all of the processors have completed the addition, we add the p sums to obtain the final sum. Figure 3.13 illustrates the process, which consists of three steps:

- **Map:** Divide the input domain into p parts, where p = 4 in this example, and add all the parts in parallel. Multithreading programming can help in completing the tasks in parallel.
- **Spin synchronization:** Wait for the computation (addition) of all the parts to complete.
- **Reduce:** Combine (add) the results from all the parts to obtain the final result.

This process is called **MapReduce**. The name comes from the higher-order functions `Map` and `Reduce` to be discussed in Chapter 4. MapReduce has been widely used in many areas. For example, the web search engines use MapReduce in this way: The domains to be searched are divided into subdomains. The search engine instances search the domains in parallel. The short lists of results from all search engine instances are merged (reduced) into a single list of results.

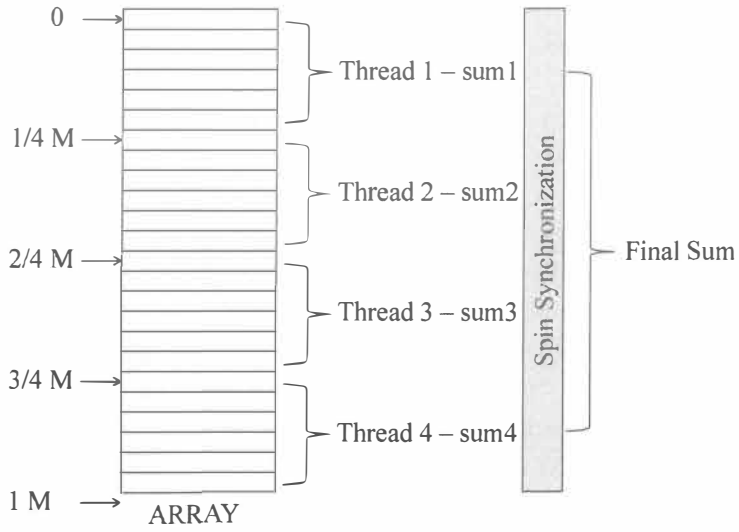


Figure 3.13. Adding a million numbers using MapReduce.

The C++ program below includes two implementations that add a million numbers. One implementation uses a single thread and the second implementation uses multithreading based on the MapReduce process. The execution times of the two implementations are compared at the end.

```
#include "stdafx.h"
#include <iostream>
#include <random>
#include <math.h>
#include <time.h>
#using <System.dll>
using namespace System;
using namespace System::Threading;
using namespace std;
// Create a global variable to store the values
#define ARRAY_SIZE 1000000
double ARRAY[ARRAY_SIZE]; // create a global array
// The ThreadClass class contains the function to be started as a thread.
public ref class ThreadClass // It is a generic class to be instantiated {
private:
  // State information used in the task.
  char* threadName;
  int s_index; // start
  int e_index; // end
public:
  ThreadClass( ) { } // Default constructor
  // The constructor obtains the state information
  ThreadClass(char* name, int start_index, int end_index) {
      threadName = name;  // Parameter passing by name
```

```
        s_index = start_index;
        e_index = end_index;
    }
// The thread function adds numbers in the given range.
    void SumFunc() {
        double sum = 0;
        for (int i = s_index; i < e_index; i++) {
            sum = sum + ARRAY[i] / ARRAY_SIZE; // make small
        }
        printf("%s: - sum =  %f\n", threadName, sum*ARRAY_SIZE);
        switch (threadName[7])  // extract the thread number {
            // store sub sums into the array elements 1, 2, 3, and 4
            case '1': ARRAY[1] = sum; break;
            case '2': ARRAY[2] = sum; break;
            case '3': ARRAY[3] = sum; break;
            case '4': ARRAY[4] = sum; break;
        }
    }
};
// Entry point for MultiThreading, in which threads are created and started
public ref class MainClass {
public:
    static void Main()  {
        clock_t START_TIME, END_TIME;
        // Generate random numbers and store them into a global array
        for(int index = 0; index < ARRAY_SIZE; index++) {
            ARRAY[index] = rand();
        }
        // without multithreading
        START_TIME = clock();
        double sum = 0;
        for (int i = 0; i < ARRAY_SIZE; i++) {
            sum = sum + ARRAY[i]/ARRAY_SIZE;
        }
        printf("Final Sum without threading =  %f\n", sum*ARRAY_SIZE);
        END_TIME = clock();
        printf("Final Sum without threading completed in %0.4f seconds.\n",
(END_TIME - START_TIME) / (float)CLOCKS_PER_SEC);
  // create four thread objects
        ThreadClass^ OBJECT1 = gcnew ThreadClass("THREAD 1", 0,
(ARRAY_SIZE/4) * 1 );
        ThreadClass^ OBJECT2 = gcnew ThreadClass("THREAD 2", (ARRAY_SIZE/4),
(ARRAY_SIZE/4) * 2);
        ThreadClass^ OBJECT3 = gcnew ThreadClass("THREAD 3", (ARRAY_SIZE/4)
* 2, (ARRAY_SIZE/4) * 3);
```

```
        ThreadClass^ OBJECT4 = gcnew ThreadClass("THREAD 4", (ARRAY_SIZE/4)
* 3, (ARRAY_SIZE/4) * 4);
   // Create four threads from the thread objects using library class Thread
        Thread^ THREAD1 = gcnew Thread(gcnew ThreadStart(OBJECT1,
&ThreadClass::SumFunc));
        Thread^ THREAD2 = gcnew Thread(gcnew ThreadStart(OBJECT2,
&ThreadClass::SumFunc));
        Thread^ THREAD3 = gcnew Thread(gcnew ThreadStart(OBJECT3,
&ThreadClass::SumFunc));
        Thread^ THREAD4 = gcnew Thread(gcnew ThreadStart(OBJECT4,
&ThreadClass::SumFunc));
        START_TIME = clock();
// Start running the four threads
        THREAD1->Start();
        THREAD2->Start();
        THREAD3->Start();
        THREAD4->Start();
// Spin Synchronization: Wait for all threads to complete
// before adding the sub-sums to obtain the final sum
        THREAD1->Join();  // wait until the thread terminates
        THREAD2->Join();
        THREAD3->Join();
        THREAD4->Join();
        sum = ARRAY[1]+ARRAY[2]+ARRAY[3]+ARRAY[4];
        printf("Final sum =  %f\n", sum*ARRAY_SIZE);
        END_TIME = clock();
        printf("Entire threads process completed in %0.4f seconds.\n",
(END_TIME - START_TIME) / (float)CLOCKS_PER_SEC);
    }
};
int main() // The main() of C++ must be a global function {
    MainClass::Main();  // call the MainClass's Main() function
}
```

The output of the program is given in Figure 3.14. It shows the execution time (0.046 second) of a simple for-loop implementation without using multithreading, the time of each thread (in clock cycles), and the time (0.172 second) used by the entire threading implementation.

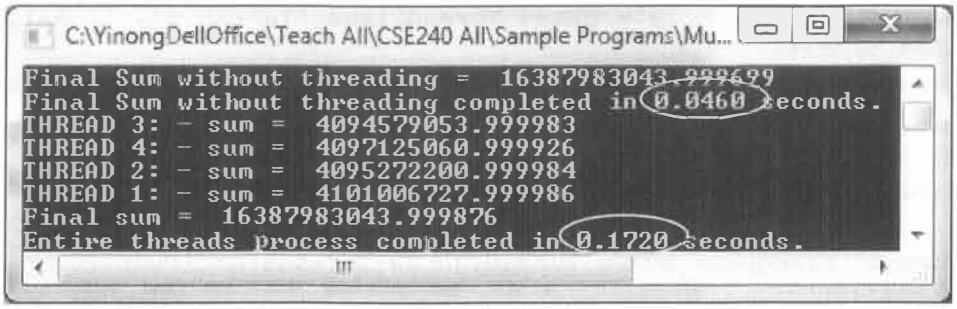


Figure 3.14. Output of the program, showing the execution time.

Why is the multithreading solution slower than the single threading implementation? The reason is that the calculation is too simple. The overhead of creating threads and synchronizing the threads exceeds the benefit of parallel computing. Multithreading will make sense only if the computing tasks are heavy, such as solving equation systems and searching the web.

3.10 Summary

In this chapter, we first briefly reviewed the principles of the object-oriented paradigm. We then discussed class composition and definition, including information hiding through defining public, protected, and private members of a class. A large program example was used to explain the related concepts. Understanding memory management in a programming language is the key to understanding memory allocation and deallocation of global/static, stack, and heap variables. After the discussion of the memory management, we are in a much better position to understand the constructor and destructor of a class. Garbage collection in C++ is done jointly by stack management, destructor and explicit `delete` operations. In Section 3.4, we studied the major features of the object-oriented programming paradigm—including class inheritance, inheritance-based class hierarchy, polymorphic pointer, virtual functions, and late binding—that support polymorphic function calls. Again, we used a large example, the `Personnel` hierarchy, to illustrate the concepts and principles we studied in this section. Section 3.5 presented the overloading features in C++, including function overloading and operator overloading. Section 3.6 discussed C++ standard input and output, as well as the file operations. Section 3.7 introduced the exceptions in computer systems and how to write exception handlers in C++. Finally, Section 3.8 put all the concepts learned in this chapter in a case study. Finally, Section 3.9 briefly studied generic class and, generic functions, and generic types in C++ and parallel computing and multithreading in C++.

3.11 Homework, programming exercises, and projects

1. Multiple Choice. Choose only one answer for each question. Choose the best answer if more than one answer is acceptable.

1.1 In C++, if class B is derived from class A, then without casting,

- ❑ a pointer to a class `A` object can point to a class `B` object.
- ❑ a pointer to a class `B` object can point to a class `A` object.
- ❑ a pointer to a class `A` object can point to a class `B` object, and vice versa.
- ❑ a pointer to a class `A` object can NOT point to a class `B` object, and vice versa.

1.2 A virtual member function in C++

- ❑ is an abstract interface that has no implementation.
- ❑ is an extendable function that allows a programmer to add formal parameters to the function.
- ❑ implies early binding between the function name and the code.
- ❑ implies late binding between the function name and the code.

1.3 Type checking during compilation will prevent a base-class pointer from accessing

- ❑ any members of the derived class object.
- ❑ the inherited members of the derived class object.
- ❑ the additional members of the derived class object.
- ❑ public members in the derived class object.

1.4 What part of memory should be deallocated by the destructor?

- ❑ heap object created in the constructor.
- ❑ stack object created in constructor.
- ❑ heap object created in main() function.
- ❑ static object created in main() function.

1.5 What part of memory must be deallocated by an explicit "delete" operation?

- ❑ heap object created in the constructor
- ❑ stack object created in constructor
- ❑ heap object created in main() function
- ❑ static object created in main() function

1.6 If class B is derived from class A, and x is a protected member of A, in which classes can x be accessed?

- ❑ Only in `A`.
- ❑ Only in `B`.
- ❑ In `A` and `B`.
- ❑ None of them.

1.7 A, B, and C are three independent classes. A wants to allow B to access all its members, but does not allow C to access. What is the best way for class A to implement this access control?

- ❑ Use public.
- ❑ Use protected.
- ❑ Use private.
- ❑ Use friend.

1.8 If class B is derived from class A, and two pointers p and q are declared by

```
A *p; B *q;
```

which operation below is valid according to polymorphism?

- ❑ `p = q;`
- ❑ `q = p;`
- ❑ Both of them.
- ❑ None of them.

1.9 If a member function in a class A is defined as a virtual function, then the member function

- ❑ cannot be defined in class A.
- ❑ cannot be re-defined in a derived class.
- ❑ can be re-defined in a derived class.
- ❑ none of the above.

1.10 Type checking during compilation will prevent a base-class pointer from accessing

- ❑ any members of the derived class object.
- ❑ the inherited members of the derived class object.
- ❑ the new (extended) members of the derived class object.
- ❑ public members in the derived class object.

1.11 If the relationship between two classes can be best described as an "is-a" relation, we should

- ❑ derive one class from the other (use inheritance).
- ❑ contain one class in the other.
- ❑ define them totally independent of each other.
- ❑ none of the above

1.12 The class hierarchy of a C++ program is formed according to the

- ❑ number of data fields in classes.
- ❑ number of member functions in classes.
- ❑ number of public members in classes.
- ❑ inheritance relationship among classes.

1.13 In the C++ exception structure, how many handlers can be defined following each try-block?

- ❑ zero or more.
- ❑ one only.
- ❑ one or more.
- ❑ at most two.

1.14 What is the function of throw statement?

- ❑ exit from a try-block and pass a value to a catch-block
- ❑ same as try
- ❑ exit from a catch-block and pass a value to the try-block
- ❑ none of the above

Given the C++ class definitions below, answer the following three questions:

```
class employee {char *name; long id; char *department;}
class manager1 {employee empl; short rank;}
class manager2: public employee {short rank;}
```

1.15 What class is defined using a containment relationship?

- ❑ `manager1`
- ❑ `manager2`
- ❑ both of them
- ❑ none of them

1.16 How does an object m of manager1 class access the member id?

- ❑ `m.id`
- ❑ `m.empl.id`
- ❑ `m.empl->id`

1.17 How does an object n of manager2 class access the member id?

- ❑ `n.id`
- ❑ `n.empl.id`
- ❑ `n.empl->id`

1.18 What type casting mechanism should be used if you want to cast an integer value to a double value?

- ❑ static_cast
- ❑ const_cast
- ❑ dynamic_cast
- ❑ reinterpret_cast

1.19 What type casting mechanism should be used if you want to change pointer type for pointing to a different object in an inheritance hierarchy?

❑ static_cast ❑ const_cast ❑ dynamic_cast ❑ reinterpret_cast

1.20 What features are supported in C++? Select all that apply.

❑ virtual function ❑ function overloading ❑ operator overloading ❑ virtual operator

1.21 What type of values can a throw-statement throw (return)?

❑ Primitive type ❑ String type ❑ Object types ❑ All of them.

1.22 What is a generic type?

❑ A type that can take different type of values at the same time.

❑ A type that contains all other types.

❑ The type can be determined at run time.

1.23 Can a multithreading program take longer time than a single thread program that solves the same problem?

❑ Yes ❑ No

1.24 Why do we need the spin synchronization at the end of a MapReduce process?

❑ To improve the performance of multithreading.

❑ To make sure that all the threads complete their tasks.

❑ To instantiate a concrete type to a generic type.

1.25 "Map and Reduce" is a concept for

❑ defining generic type. ❑ defining generic class.

❑ defining parallel computing process. ❑ saving memory usage.

2. Given the C++ code below, answer the following questions.

```
class Queue {
  private:
    int queue_size;
  protected:
    int *buffer; int front; int rear;
  public:
    Queue(int n) {
         front = 0; rear = 0; queue_size = n;
         buffer = new int[queue_size];}
    virtual ~Queue(void) {delete buffer; buffer = NULL;}
    void enqueue(int v) {
         if (rear < queue_size) buffer[rear++] = v;
    }
    int dequeue(void){ // return and remove the 1st element
```

```
        if (front < rear)
            return buffer[front++];
    }
}
Queue InitialQueue(100);
void main() {
    Queue *myQueue = new Queue(50);
    myQueue->enqueue(23);
    InitialQueue->enqueue(25);
    delete myQueue;
}
```

2.1 What is the destructor of the `Queue` class?

❑ enqueue(int v) ❑ dequeue(void) ❑ Queue(int n) ❑ ~Queue(void)

2.2 What variable must be destructed (garbage-collected) by the destructor in the `Queue` class?

❑ front ❑ rear ❑ queue_size ❑ *buffer

2.3 Where does the variable (object) `InitialQueue` obtain memory from?

❑ static memory ❑ stack ❑ heap ❑ queue

2.4 Where is the destructor called in the program above?

❑ in the constructor ❑ in the destructor ❑ in the `enqueue` function

❑ in the `dequeue` function ❑ implicitly in `delete myQueue`

2.5 Where does the pointer variable `myQueue` obtain memory from?

❑ static memory ❑ stack ❑ heap ❑ queue

2.6 Where does the object pointed to by the pointer variable `myQueue` obtain memory from?

❑ static memory ❑ stack ❑ heap ❑ queue

3. What are the main differences between the imperative paradigm and the object-oriented paradigm?

4. What is the purpose of a scope resolution operator?

5. What is the purpose of a constructor? Can we have more than one constructor?

6. What is the purpose of a destructor? Can we have more than one destructor? When do we need a destructor?

7. What are the main differences between `Queue r, *s`, where `Queue` is a class? How do we use them? Where do they obtain memory? Which of the following operations are valid?

```
r = s;        s = r;        s = &r;
r = &s;       r = *s;       s = *r;
```

8. In the queue example in Section 3.1, an `enqueue` operation will cause the queue elements to shift to make spaces (compact operation) if the rear pointer points to the end of the queue. This will cause a longer delay than other `enqueue` operations. To smooth the performance, we can organize the queue into a "cyclic queue," as shown in Figure 3.15. In other words, when `rear = n`, we set `rear = 0`, as long as the number of the entries in the queue is less than the queue size (or `front > 0`). Modify the given program to implement the virtual ring. You can add a variable "entries" in the class to keep track of the number of entries in the queue.

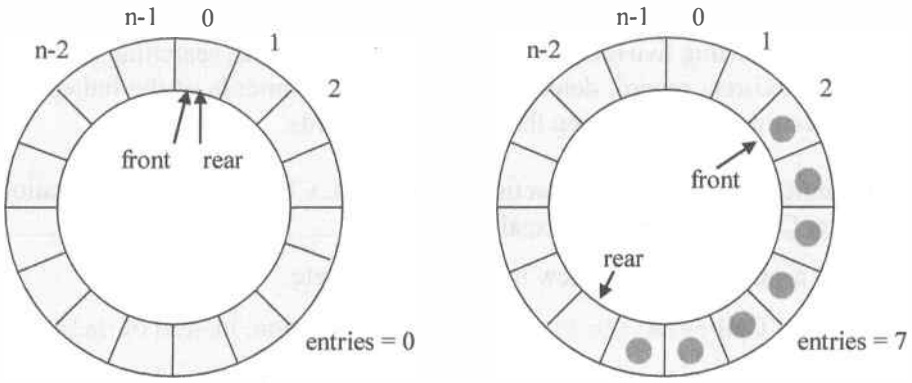


Figure 3.15. A cyclic queue, its initial state, and the state with 7 entries.

9. Memory management.

9.1 What is a static variable? What is the difference between a static local variable and a global variable? Do we need to collect the memory allocated to static variables?

9.2 What is the programming language's stack? What variables obtain memory from the stack? What mechanism is used to collect the memory allocated from the stack?

9.3 What is the programming language's heap? What variables obtain memory from the heap? What mechanism is used to collect the memory allocated from the heap?

9.4 Compare and contrast variables from static memory, stack, and heap. What would happen if we declared a local variable as static local? What would happen if we declared a static local variable as a simple local variable? What would happen if we declared a static local as a global?

10. What is inheritance? What is a "*is-a*" relation? What is a "*has-a*" relation? When do we use inheritance? What are the advantages of using inheritance?

11. Consider the `contactbook.c` program given in the homework, programming exercises, and projects in Chapter 2.

11.1 Read the program and try to understand what the program does. Set command line parameter and execute the programming following the instructions given in the question.

11.2 Save `contactbook.c` as `contactclass.cpp` (`cpp` for C++). The executable is then `contactclass.exe` and the database should be called `contactclass.dbms`.

11.3 Change the structure definition `struct contact` to a class definition `class contact`. All the four data members must be `private`.

11.4 Write `public get-` and `set-` member functions for each of the four data members, so that the data members can be accessed through member functions.

11.5 Write a `constructor`. The `constructor` takes three parameters (`name, phone, email`). The data members `name, phone, email` must be initialed by the parameter values (e.g., `strcpy(this->name, name);`). The pointer `next` must be initialized to `NULL`.

11.6 Change `malloc` to `new` and change `free(p)` to `delete p`. Make minimal changes to the rest of the program to make the program executable.

11.7 Test the program by inserting five records, displaying all the records, searching for an existing record, searching for a nonexistent record, deleting the record in the middle of the linked list, deleting the first and the last records, and displaying the remaining records.

12. Memory management and garbage collection experiment. Close all other applications before you load the following C++ program into Visual Studio C++.

12.1 Build and run the program. It takes a few minutes to complete.

12.2 Modify the program: Call `deletion1();` in the main() function, instead of `deletion2();`

12.3 Modify the program: Call `deletion0();` in the main() function, instead of `deletion2();`

12.4 Explain the differences between `deletion0()`, `deletion1()`, and `deletion2()`.

```
/* Warning: This program illustrates memory leak. The program has been
tested in Visual C++ 6.0 and Visual Studio .Net on Windows 2000 and XP.
Do not run the program on computers with Windows 98 or older. If you
don't see the memory leak problem, you can increase the number of
iterations for i in the main() function. */
#include <iostream>
#include <string.h>
using namespace std;

// forward declaration
int insert(void);
void deletion0(void);
void deletion1(void);
void deletion2(void);
int IDgenerator = 0;   // global variables
class contact {
    private:
        char name[30];
        char phone[20];
        char email[30];
        contact *next;
        int userId;
    public:
        contact() {}
```

```
        contact(char* name, char* phone, char* email){
            strcpy(this->name, name);
            strcpy(this->phone, phone);
            strcpy(this->email, email);
            userId = IDgenerator++;
            this->next = NULL;
        }
        contact* getNext() { return next; }
        void setNext(contact* nx) { next = nx; }
} *head = NULL; // declare head pointer as global
int main() {
    int i = 0, j = 0;
    while (i<10000) {
        for (j = 0 ; j<1000; j++) {
            insert();    // insert 1 000 records
        }
        deletion2();
// What happens if you call deletion0() or deletion1() here?
        i++;
        cout << i << " - " << IDgenerator << endl;
    }
    return 0;
}
int insert() {
    contact *node;
    char sname[30] = "John", sphone[20] = "1234 567", semail[30] =
"john@asu.edu";
    node = new contact(sname, sphone, semail);
        if (node == NULL) {
            cout << "ERROR - Could not allocate memory !" << endl;
            return -1;
        }
        node->setNext(head);
        head = node;
        return 0;
}
void deletion0() { head = NULL; }
void deletion1() { delete head; head = NULL; }
void deletion2() {
    contact *p;
    while (head != NULL) {
        p = head;
        head = head->getNext();
        delete p;
```

```
    }
}
```

13. Complete the `Personnel` hierarchy program in Section 3.4 by adding the following classes and functions into the program.

13.1 Add the classes `Student`, `Grad`, `Undergrad`, and `Staff`.

13.2 Add functions `save_file()`, and `load_file()`, so that the records stored in the linked list can be preserved onto the disk before the program quits and the linked list can be preloaded with the entries stored in the disk file when the program is restarted.

13.3 Add a menu item and related functions to delete all nodes in the `PersonnelNode` linked list. Make sure there is no memory leak for all dynamic memory.

13.4 Use C++ exception constructs to define and handle all possible exceptions in the program, for example, out of memory or file operation error exceptions.

13.5 Use `typeid(*ptr)` to identify the type of the object that the pointer `ptr` is pointing to. For example, the following code can be used to determine if the pointer `ptr` is pointing to an object of `Employee` class.

```
#include<typeinfo>
...
if(typeid(*ptr) == typeid(Employee)) {...}
```

In the Visual Studio environment, you must enable the runtime type info setting in order to use this feature, by following these steps:

```
Menu: Project->YourProjectName Properties... -> Configuration Properties
-> C/C++ -> Language -> Enable Run-Time Type Info: Choose "Yes"
```

Modify the `display()` function in each class to print the class name of the current object using the `typeid` function.

14. Define your own heap to replace the system heap used in the question above.

14.1 Declare a large block of memory (e.g., a global array) called `myheap` and use the block of memory as your own heap. Organize the memory as a linked list whose nodes can have different sizes. In each node, use the first word (e.g., an integer of four bytes) to store the size (number of bytes) of the node. Define two functions

```
void insertion(void *p);
void *deletion(int m);
```

that can insert a new node and remove a node from the linked list, respectively.

14.2 Define a function

```
void *GetMem(int n);
```

that takes n bytes of memory from `myheap` and returns the initial address. The function must call the `deletion` function to obtain the memory.

14.3 Define a function

```
void ReleaseMem(void *p);
```

that returns the memory pointed to by `p` to `myheap`. The function must call the `insertion` function to return the memory.

14.4 Make a copy of the program you wrote. Modify the copy to use `myheap` for the objects of the `Student` class.

14.5 Test your program by adding and deleting `Student` nodes.

Chapter 4

The Functional Programming Language, Scheme

When we moved from the imperative programming paradigm to the object-oriented paradigm, we did not really feel that we had a paradigm shift. Indeed, the object-oriented paradigm is based on the imperative paradigm. Some texts categorize imperative and object-oriented paradigms as the procedural paradigm. However, you will see in this chapter that it is a real paradigm shift when you switch from imperative and object-oriented programming to functional programming.

In this chapter, we will use Scheme as an example to study the main features and programming techniques of functional programming languages. By the end of the chapter, you should

- have a good understanding of the functional programming paradigm and its major differences with the imperative and object-oriented programming paradigms;
- have a good understanding of data types and their operations;
- be able to define procedures and macros;
- understand the relationship between λ-calculus and the functional programming language;
- be able to define and use global and local variables;
- be able to write Scheme programs with multiple procedures;
- have a good understanding of the principles of recursion;
- be able to apply recursive procedures to solve different types of problems;
- understand and use higher-order functions to solve recursive problems.

The chapter is organized as follows. We first discuss the main differences between imperative and functional programming paradigms. In Section 4.2, we briefly review the prefix notation used in Scheme and other functional programming languages. In Section 4.3, we put together the terminology that will be used throughout the chapter. In Section 4.4, we study the data types and the predefined Scheme functions that form the Scheme environment on which we can define our own programs. In Section 4.5, we discuss the basic syntax and semantics of the general computing system λ-calculus. Scheme and many other programming languages can be considered an implementation of λ-calculus. From Section 4.6, we study the important programming constructs of functional programming languages: named procedure, unnamed procedure, let-form that defines local variables, and the conversion between unnamed procedures and let-forms. We also start to write our own Scheme programs with multiple procedures. In Section 4.7, we study the most important programming technique in functional programming languages: recursion. We discuss structural (white-box) and functional (black-box) approaches of understanding a recursive procedure. We outline simple steps that can guide us in writing recursive procedures. In Section 4.8, we continue to study

recursive procedures on different data types. Finally, in Section 4.9, we present the unique feature of functional programming languages: The higher-order functions, where they are used, and how they are implemented. A programming environment that supports the development of Scheme programs, DrRacket, is introduced in Appendix B.

4.1 From imperative programming to functional programming

The idea of **functional programming** is to liberate programming from the von Neumann-style (or stored program concept)-based imperative programming paradigm. Imperative programming languages are more efficient to implement because they match the currently used stored program concept-based computer architecture. However, they force programmers' attention to the detail of storing states and modifying states. The major problems of imperative programming paradigms are:

- the lack of accurate definition of the semantics;
- the referential use of variables and its side effects;
- the need to have a good understanding of computer architecture and memory organization;
- the difficulty of programming parallel-executable components.

The functional programming paradigm tries to address these problems by focusing on what is wanted rather than how it is implemented. An analogy is to compare ordering a pizza in a restaurant to making your own pizza at home. When you order a pizza, you focus on what you want. When you make your own pizza, you not only know what you want, but also understand the underlying hardware, for example, at what temperature, how long the pizza needs to be baked, and how to control the temperature and set the timer of the oven.

The major advantages of functional programming paradigm are as follows:

- Higher level of abstraction that needs less attention to the details of the underlying computer architecture.
- Simpler and more accurate definitions of semantics. It is based on well-founded mathematics and thus is easier to reason about.
- Side-effects-free or referential transparency. For example, in an imperative program, f(x) + f(x) may or may not be equal to 2*f(x), depending on whether the function modifies any global variables or the parameter x. However, in a functional program, it is guaranteed that f(x) + f(x) is always equal to 2*f(x).
- Easier for parallel processing. In functional programs, operations at the same level are independent of each other and can thus be processed in any order or in parallel.
- Powerful higher-order functions. You can pass the operation (not only the return value) of another function as the parameter to a function.

The above differences are mostly at the conceptual level. What are the differences at the programming level? That is, what different approaches do we have to take in writing functional programs? Understand that these differences are the main topics of this chapter. The following list is an outline of these differences:

- Functional programming languages are not based on the stored program concept. You cannot declare a variable (a memory location) to store a value and later modify the value, or store another value in the same memory location. What you can do is to define a name and associate a value with the name (named value). This name then represents the value, not the container of the value, and thus you can never associate the name with another value nor modify the value. This approach is often considered by imperative programmers as a restriction that brings the least convenience in

programming. However, this approach is the milestone of the paradigm shift that gets rid of the stored program concept of stepwise storing and manipulating data.

- In functional programming languages, parameter passing is the major mechanism that passes values into a function. Only call-by-value is allowed. Call-by-alias is not allowed. In fact, parameter passing is used to replace most assignment statements in imperative programs.
- Every function in a functional programming language will return a single value. Since intermediate values cannot be stored in memory and be used later, you must organize your program to use a return value immediately. In other words, a return value must be passed to another function immediately. If you want to return two or more values, you can
  - split the function into multiple functions and each function return one value, or
  - organize multiple values into a pair (or nested pair) structure, or
  - organize multiple values into a list structure.
- Functions are treated as **first-class** objects. In other words, a function can appear in any place where a value is expected. In this case, the function will be first evaluated, and its return value will be used as the value expected. This mechanism effectively supports immediately passing the return value to another function.

Now we will use an example to illustrate these differences. The following imperative program declares two global variables `a` and `b`, and two functions `foo` and `main`.

```
int a = 5;              // declare a global variable and initialize it to 5
float b = 2.4;          // declare a global variable and initialize it to 2.4
float foo(int x, float y) {  // return value type, function name,
parameters
  int f1;               // declare a local variable and initialize it to 0
  float f2;             // declare a local variable and initialize it to 0
  x = x + 5;            // modify variable value
  y = y + 10.5;         // modify variable value
  f1 = abs(x);          // call a function and assign to return value f1
  f2 = square(y);       // call a function and assign to return value f2
  f2 = max(f1, f2);     // call a function and assign return value to f2
  return f1 + f2;       // return
}
void main() {           // main function does not return any value
  float f;
  f = foo(a, b);        // function call and parameter passing
  printf("%f\n", f);
}
```

As we can see, the program is written in a way in which a typical imperative program is written:

- it stores values in variables;
- it manipulates (modifies) variable values;
- it returns a value.

How do we write a functional program to implement the same function? The following Scheme program does exactly the same job:

```
(define a 5)                    ; name value 5 as a
(define b 2.4)                  ; name value 2.4 as b
```

```
(define (foo x y)                    ; function name followed by parameters
  (+  (abs (+ x 5))                  ; the first operand of + is abs function
      (max (abs (+ x 5))             ; the second operand of + is max function
           (square (+ y 10.5))       ; the second operand of max is square
      )
  )
)
(print (foo a b))                    ; the main function will return a value
```

The syntax of the Scheme program is straightforward. Every operation or expression consists strictly of a list in prefix notation:

```
(operator operand ... operand)
```

where the first element is always the operator (e.g., +, abs, square, max), followed by the list of operands. Each operand can be either a value or a function that returns a value.

Now you are in a good position to understand the Scheme program. A careful comparison between the two programs will confirm the approaches used in the functional program: No memory is available to store the intermediate values; intermediate values must be passed to another function immediately; every function returns a value; a function can be placed in any place where a value is expected. Having understood these differences, your imperative programming experience can now be positively used in writing functional programs.

4.2 Prefix notation

We can represent mathematical operations and expressions in three different notations: **infix, prefix**, and **postfix**. Table 4.1 shows examples of these notations:

Infix	Prefix	Prefix with parentheses	Postfix
3 + 4	+ 3 4	(+ 3 4)	3 4 +
(3 + 4) * 5 + 6	+ * + 3 4 5 6	(+ (* (+ 3 4) 5) 6)	3 4 + 5 * 6 +
45 max 29	max 45 29	(max 45 29)	45 29 max

Table 4.1. Different representations of mathematical expressions.

Infix notation was invented in the 1920s by the Polish mathematician Jan Lukasiewicz and it is thus also known as **Polish notation**. The postfix notation is also called **reverse Polish notation**. Although infix notation is easier for humans to use, Scheme language uses prefix notation with parentheses for all expressions. The reasons for this decision are:

- We can consistently use prefix notation to represent unary, binary, and multi-operand operations, for example, unary: (− 3), binary: (+ 3 4), and multi-operand: (max 3 4 5 6 7), while infix notation can represent only binary operations.
- Prefix notation is a parenthesis-free notation. Although parentheses in the expression help to understand the order of evaluation, they are not necessary for prefix notation to define the order of the evaluation. On the other hand, infix notation has to rely on the parentheses to define the order of evaluation.
- Expressions in prefix notation are easier to execute on a stack-based computer architecture.

A mathematical expression can also be represented as a rooted tree. A **rooted tree** is a directed tree with a unique **root** node, and there is a **path** from the root to any other node. A **directed tree** is a **directed graph** in which there is at most one path between any pair of nodes. The prefix, infix, and postfix notations can be obtained via traversing the tree, which is the process of inspecting the nodes of the tree. There are three common ways of tree traversing: preorder, inorder, and postorder.

Preorder traversing: The root node of a (sub) tree is visited first, its leftmost subtree is then visited, and finally the rightmost tree is visited. Preorder traversing can be applied to any rooted tree.

Inorder traversing: The left subtree is visited first, then the root node of the (sub) tree, and finally, the right subtree is visited. Inorder traversing is defined only for binary trees.

A **binary tree** is a tree in which each node has at most two child nodes.

Postorder traversing: Starting from the leftmost subtree, then the rightmost subtree, and finally, the root node. Postorder traversing can be applied to any rooted tree.

Figure 4.1 shows a binary tree representing a mathematical expression with binary operations.

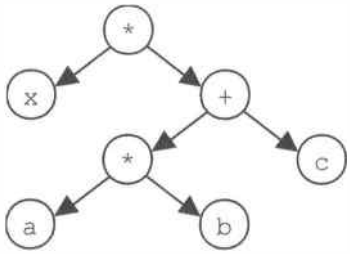


Figure 4.1. Tree representation of a mathematical expression.

If we traverse the tree in the three orders—prefix, infix, and postfix—and print the value of the node when we visit a node, the following expressions will be printed:

Preorder:

```
*x+*abc
```

The printed sequence is the prefix notation of the expression represented by the binary tree.

Inorder:

```
x*a*b + c  {infix notation}
```

The printed sequence is an incorrect infix notation of the expression, because parentheses are necessary in infix notation. We have to apply a more complex algorithm to print the infix notation with correct parentheses. The correct infix notation of the expression represented in the tree should be

```
(x*(a*b + c))
```

Postorder:

```
xab*c+*  {postfix notation}
```

The printed sequence is the postfix notation of the expression represented by the binary tree.

Note that expressions in prefix and postfix notations are free of parentheses: The order of computation is specified by the sequence of symbols only. They can be calculated easily on a stack-based processor. For example, one simple algorithm of calculating an expression in prefix is as follows:

```
(1)   The expression is scanned (read) from right to left;
```

```
(2)   If a parenthesis character "(" or ")" is scanned,
      simply ignore it and continue to scan the next character;
(2)   If a variable or a value is scanned, push it onto the stack;
(3)   If an operator (a symbol representing an operation) is scanned,
      the appropriate number (in the example above, the number is 2) of
      variables is popped out from the stack, and the operation applied to
      them;
(4)   The result of the operation is pushed back onto the stack;
(5)   The scan of the expression continues;
(6)   After the expression has been completely scanned, the result is the
      only item left in the stack.
```

For example, if we evaluate the expression `(- 5 (+ a b) c))` on a stack-based processor, we start from the rightmost parenthesis and stop at the leftmost parenthesis, as shown in Figure 4.2.

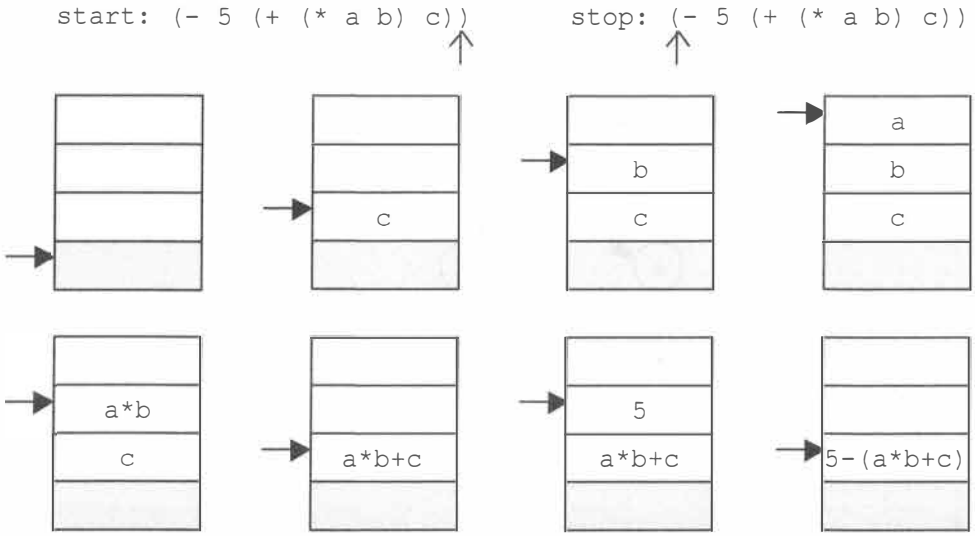


Figure 4.2. Expression processed and the stack maps during processing.

The first stack map is the initial stack status. Following the algorithm above, parentheses are ignored and characters `c`, `b`, and `a` are pushed onto the stack (stack map 2 through 4). The operator * is scanned, `a` and `b` are popped out, multiplied and pushed back onto the stack (stack map 5). Then the operator + is scanned, the values of "`a*b`" and c are popped out, added, and pushed back onto the stack (stack map 6). Next, 5 is pushed onto the stack (stack map 7), and, finally, the operator − is scanned, the values of "`a*b+c`" and 5 are popped out, subtracted, and pushed back onto the stack (stack map 8).

4.3 Basic Scheme terminology

This section introduces basic terminology used in this chapter.

Primitive: A primitive is a predefined operation in Scheme. These operations cannot or do not need to be defined in terms of anything simpler. When we use a Scheme environment, we, in fact, use the set of primitives to build our applications. For example, the following operations are examples of the primitives:

```
(+ 3 6)
```

```
(<= 2 5)
(sqrt 32)
(number? 45)
(symbol? "x")
(append "abc" "123")
```

In the rest of the chapter, we will see many more Scheme primitives.

Form: Anything that you ask Scheme to evaluate is a form. The answer (return value) to a form is the **value** of the form. For example:

```
(+ 3 6)
(+ (* 4 6))
(twoscomplement '(1 0 0 1 1 0))
```

Procedure: Procedure is a user-defined new *primitive operation* using the keyword *define*. A procedure always returns a value when it is evaluated.

Function: In Scheme, there is no function. The term *function* is used only in its general meaning, say, a mathematical function. A *procedure* in Scheme has the same meaning as the *function* in C/C++.

Keyword **define**: It is used to create a named form (named constant or **named procedure**). In other words, it associates a form with a name. A complete definition consists of parts: (1) the keyword *define*, (2) the name to be defined, (3) any form that gives a value. For example:

```
(define size 100)                    ; named constant
(define x (* 5 6))                   ; named operation
(define (add1 x) (+ x 1))            ; named procedure, where x is the
parameter
```

Keyword **define-macro**: It is used in the same way as define is used, except that it introduces a macro, instead of a procedure. For example:

```
(define-macro (add1 x) (+ x 1))   ; define a macro, where x is the
parameter
```

Keyword **lambda**: It is a keyword used to define an **unnamed procedure**. For example:

```
(define (add1 x) (+ x 1))            ; named procedure, where x is the
parameter
((lambda (x) (+ x 1)) 7)             ; unnamed procedure performing x+1 = 7+1
((lambda (x y) (+ x y)) 4 5)         ; unnamed procedure performing x+y = 4+5
```

Since an unnamed procedure is a form, we can associate an unnamed procedure with a name, thus introducing another way of defining a named procedure as shown in the examples below:

```
(define (add1 x) (+ x 1))            ; define named procedure without using
lambda
(define (f x y) (+ x y))             ; define named procedure without using
lambda
(define add1(lambda (x) (+ x 1)))  ; define named procedure using lambda
(define f (lambda (x y) (+ x y))) ; define named procedure using lambda
```

As we can see from these examples, we can define a named procedure using or without using lambda. The two ways of definition are equivalent.

Parameter: Variables used when we define a procedure. Parameters will be replaced by arguments when they are evaluated.

Argument: The input values that a procedure call needs for performing the evaluation. For example, when we call procedures `(add1 x)` and `(f x y)`, we must substitute forms or values for the parameters. For example:

```
(add1 5)          ; 5 is the argument. The procedure call will return 6
(f 6 (* 3 7))     ; 6 and (* 3 7) are arguments. It will return 27.
```

Application: It applies an operation to the results of evaluating the other forms. In other words, a primitive applies a simple operation to values that do not need any further evaluation, while application involves nested evaluation, or complex operations like procedure calls or macro calls. For example:

```
(+ 5 1)           ; is a primitive. It applies addition to simple values.
(+ (+ 2 3) 1)     ; is an application. It involves nested operation.
(add1 5)          ; is an application. It calls a procedure.
```

Figure 4.3 summarizes the terminology introduced in this section and shows the relationships among the components of a Scheme program. Everything we ask Scheme to evaluate is a form. Definitions of a procedure or a macro are not forms, because they do not cause actual evaluation. Only when we use arguments to replace the parameters and call a procedure, does it become a form; that is, procedure call is a form, not the procedure itself. A form can be a simple primitive or a complex application.

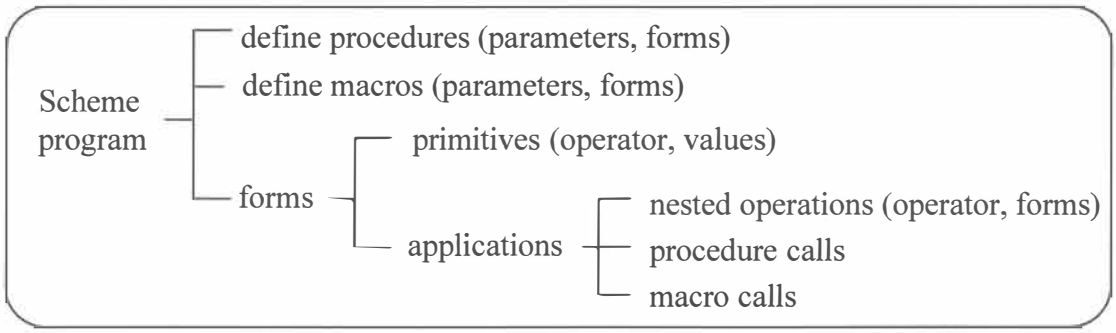


Figure 4.3. Summary of terminology and relationships among components of a Scheme program.

A mathematical expression can be evaluated in different orders. For example, to evaluate `3 + 4 + (15 / 3)`, we can have the following orders:

```
(1) 3 + 4 = 7, 15 / 3 = 5, 7 + 5 = 12, or
(2) 15 / 3 = 5, 4 + 5 = 9, 3 + 9 = 12, or
(3) 15 / 3 = 5, 3 + 4 = 7, 7 + 5 = 12
```

Generally, there are two main methods that are used to determine the **evaluation orders**. Assume that an expression has multiple nested operations.

Eager evaluation: It tries to start from innermost operations first. At the same level, operations can be performed in any order. If there is a function with parameters, it always evaluates all parameters first before it attempts to evaluate the function. Eager evaluation can better support parallel computing than can lazy evaluations, as all the computations can be done before they are needed.

Lazy evaluation: It tries to start from outermost operations first. At the same level, operations are performed from left to right sequentially. If there is a function with parameters, it will evaluate a parameter only if its value is needed.

For example, if the Scheme form to be evaluated is

```
(+ (+ 3 5) (* (+ 4 6) (- 5 3)))
```

then the eager and lazy evaluations will evaluate the form in the following orders, respectively.

```
Eager: (+ 4 6), (- 5 3), (+ 3 5), (* 10 2), (+ 8 20)
Lazy:  (+ 3 5), (+ 4 6), (- 5 3), (* 10 2), (+ 8 20)
```

The orders of the evaluation for the conditional form (if a b c) are

```
Eager: a, b, c, if
Lazy:  a, if, b; or a, if, c  // only one of b and c will be evaluated
```

For imperative languages, the order of evaluation must be strictly predefined, because different orders may produce different results. Most imperative languages, including C, C++, and Java, use lazy evaluation.

For functional programming languages, the order of evaluation does not matter, and thus, the languages do not have to predefine the order. As a result, parallel computing is possible. For example, a multiprocessor computer can evaluate the function (if a b c) in two steps: (1) Evaluate a, b, and c simultaneously; (2) Choose the result from b or c according to the result of a. On the other hand, a multiprocessor computer cannot speed up the evaluation of (if a b c) in an imperative language. It has to evaluate the function in three steps: (1) Evaluate a; (2) Choose to evaluate b or c according to the result of a; (3) Evaluate b or c.

For example, assume that the procedures a, b, and c take one hour each to compute. The eager evaluation can compute a, b, and c in parallel in one hour, while the lazy evaluation cannot evaluate the procedure in parallel. It will take one hour to compute a. Depending on the result of a (true or false), it will take another hour to compute b (if a is true) or c (if a is false).

4.4 Basic Scheme data types and functions

This section introduces basic data types and primitive Scheme functions. The available primitive functions depend on the programming environment and the version of the environment. Our discussion is mainly based on the DrRacket environment. Not all functions listed in this section will be explained here. Some of them will be explained in the later sections when we use these functions to define more complex functions.

A **data type** is defined by the range of values and operations defined in these values. Basic Scheme data types include number, Boolean, character, string, symbol, pair, and list. This subsection will briefly introduce these data types and basic operations defined in these data types.

4.4.1 Number types

There are different types of **numbers** in different programming languages, for example, long, short, integer, float, and double. Scheme has number type that includes all these types. For example, you can use all these numbers 2, −5, 1.03, 2/5, $2.5e^{-3}$ in the number type. Because numbers are represented internally as a list, not stored in a "memory location," there is theoretically no limit on the size of the numbers. For example, there is no overflow when we compute very large numbers like 100! or `(factorial 100)`. Within the number type, we can still differentiate integer and real. An incomplete set of operations defined on number type is listed in Table 4.2.

Accessor functions	Meaning	Predicate functions	Meaning
(+ <num> ... <num>)	addition	(number? <num>)	Is it a number?
(- <num> ... <num>)	subtraction	(integer? <num>)	Is it an integer?
(* <num> ... <num>)	multiplication	(real? <num>)	Is it a real number?
(/ <num> ... <num>)	division	(negative? <num>)	Is it negative?
(- <number>)	negate	(positive? <num>)	Is it positive?
(add1 <num>)	add one	(even? <num>)	Is it even?
(sub1 <num>)	subtract one	(odd? <num>)	Is it odd?
(quotient <int><int>)	quotient	(zero? <num>)	Is it zero?
(remainder <int><int>)	remainder	(= <num> ... <num>)	Are they equal?
(round <num>)	round to nearest integer		
(truncate <num>)	integer part		
(abs <num>)	absolute value		
(sqrt <num>)	square root		
(expt x y)	returns x^y		
(max <num> ... <num>)	maximum		
(min <num> ... <num>)	minimum		
(random <int>)	random number between 0 &<int>		

Table 4.2. Operations defined on number type.

4.4.2 Boolean

Boolean is a simple type with two possible values: `#t` (true) or `#f` (false). An incomplete set of operations defined on Boolean type is given in Table 4.3.

Accessor functions	Meaning	Predicate functions	Meaning
(and <expr> ... <expr>)	logical `and`	(eq? <bool><bool>	Are they equal?
(or <expr> ... <expr>)	logical `or`		Example:
(not <expr>)	logical `not`		(define x #t)
			(define y #f)
			(eq? x y)

Table 4.3. Operations defined on Boolean type.

Boolean type variables can be used in conditional statements like

```
(if c a b)
```

In the conditional form, if the value of `c` is `#t`, the form returns `a`, otherwise, it returns `b`. For example,

```
(if (= x 0) (+ a1 a2) (- b1 b2))
```

the conditional form will evaluate `(+ a1 a2)` if `(= x 0)` is true, otherwise, it will evaluate `(- b1 b2)`.

Another conditional form is the multiple conditional form `cond`:

```
(cond (c1 e1) (c2 e2) ... (cn en) (else ex))
```

In this form, the conditions c1, c2, ..., cn will be evaluated sequentially. When ci, i = 1, 2, ..., n, evaluates to #t, the corresponding expression ei will be evaluated. If all c1, c2, ..., cn evaluate to #f, the expression ex in the else part will be executed. If the else part is missing, no action will be taken.

The following example defines a procedure that converts a numerical grade into a symbolic grade.

```
(define grade (lambda(n)
  (cond ((>= n 90) 'a)
        ((>= n 80) 'b)
        ((>= n 70) 'c)
        ((>= n 60) 'd)
        (else 'f)
  )
))
(grade 89)      ; procedure call, will return b
(grade 55)      ; procedure call, will return f
```

4.4.3 Character

The data values of **character type** are the set of ASCII characters. A complete set of ASCII characters is given in Appendix C. An incomplete set of operations defined on character type is given in Table 4.4. The comparisons between two characters are based on their integer values in the ASCII table.

To differentiate a character from other similar values of other types (e.g., number, symbol, or string), we use #\5 for character 5, #\A for upper case A, #\b for lower case b, and #\space for the character space.

Accessor functions	Meaning	Predicate functions	Meaning
(char->integer <char>)	convert to int	(char? <expr>)	Is it a character?
		(char-alphabetic? <expr>)	Is it an alphabetic?
(integer->char <int>)	convert to char	(char=? <char><char>)	Are they equal?
		(char<? <char><char>)	Is char < char?
		(char>? <char><char>)	Is char > char?
		(char-ci=? <char><char>)	case-insensitive comparison

Table 4.4. Operations defined on character type.

According to the ASCII table, we can find the integral value of a character or find the character for a given integer:

```
(char->integer #\A)  ; will return 65
(char->integer #\C)  ; will return 67
(char->integer #\5)  ; will return 53
(integer->char 97)   ; will return #\a
(integer->char 36)   ; will return #\$
(integer->char 57)   ; will return #\9
```

The comparison between characters is also based on their integral values. For example:

```
(char<? #\A #\a)     ;will return #t because 65 < 97
(char>? #\$ #\9)     ;will return #f because 57 > 36
```

4.4.4 String

A **string** is a sequence of characters in a pair of double quotation marks. For example, "hello world" is a string with length 11 (there are 11 characters in the string). An incomplete set of operations defined on string type is given in Table 4.5.

A string can be indexed and the first character (leftmost) of a string has the index 0. Thus, the following two forms

```
(string-ref "hello world"0)
(string-ref "hello world"6)
```

will return characters `#\h` and `#\w`, respectively.

Accessor functions	Meaning	Predicate functions	Meaning
(string <char>)	convert char to string	(string? <str>)	Is it a string?
(string->symbol <str>)	convert string to symbol	(string=? <str><str>)	Are they equal?
(symbol->string <sym>)	convert symbol to string		
(string->number <str>)	convert string to number		
(number->string <num>)	convert number to string	(string-ci=? <str><str>)	Are they case-insensitively equal?
(string-length <str>)	get string length		
(string-append <str><str>)	append two strings		
(string-ref <str><i>)	get char at position i		
(substring <str><i><j>)	get substring between i and j		

Table 4.5. Operations defined on string type.

4.4.5 Symbol

A **symbol** is a name prefixed with a single quote (e.g., `'James` and `'2t3w` are symbols). Unlike a string, a symbol cannot contain a space. A space marks the end of a symbol. Literal values of number, Boolean, character, and string cannot be symbols. The quote prefixed to them will simply be ignored. For example, `(+ '2 4)` is the same as `(+ 2 4)` and will return 6; `(string-length '"hello")` is the same as `(string-length "hello")` and will return 5. An incomplete set of operations defined on symbol type is given in Table 4.6.

Accessor functions	Meaning	Predicate functions	Meaning
(quote x)	same as `'x`	(symbol? <sym>)	Is it a symbol?
(string->symbol <str>)	convert string to symbol	(eq? <s1><s2>)	Are the two symbols equal?
(symbol->string <sym>)	convert symbol to string		

Table 4.6. Operations defined on symbol type.

Please note that symbols are case-insensitive; that is, `'A` and `'a` are considered to be the same symbol. The form `(eq? 'ABC 'abc)` will thus return true.

4.4.6 Pair

Pair is a structured data type in Scheme. A pair is denoted by `'(x . y)`, where `x` and `y` can be any literal values of any data type.

An incomplete set of operations defined on pair type is given in Table 4.7.

Accessor functions	Meaning	Predicate functions	Meaning
(cons <expr><expr>) (car <pair>) (cdr <pair>)	form a new pair return first element return second element	(pair? <expr>) (equal? <pair><pair>)	Is it a pair? Are they equal?

Table 4.7. Operations defined on pair type.

The acronyms `car` and `cdr` originally meant "Contents of Address portion of Register" and "Contents of Decrement portion of Register," which are the first part and second part of a register in an IBM 704 machine.

The following piece of code illustrates the pair-based functions and their return values:

```
(cons 1 2)                    ; will return a pair (1 . 2)
(cons 4 8)                    ; will return a pair (4 . 8)
(cons 1 (cons 4 8))           ; will return a pair (1 . (4 . 8))
(car '(4 . 8))                ; will return the first element that is 4
(cdr '(4 . 8))                ; will return the second element that is 8
(car (cdr '(1 . (4 . 8))))    ; will return 4
(pair? '((4 . 8) . 9))        ; will return #t (true)
```

In the piece of code, we see nested pairs like `'(1 . (4 . 8))` and `'((4 . 8) . 9)`. In fact, we can nest any level of pairs to produce a very complex structure. For example, the following two forms:

```
(cons (cons 2 (cons 8 7)) (cons 4 8))
(cons 12 (cons 2 (cons 8 (cons 4 (cons 3 7)))) )
```

will produce the following pairs:

```
((2 . (8 . 7)) . (4 . 8))
(12 . (2 . (8 . (4 . (3 . 7)))))
```

To reduce the complex appearances of nested pairs, Scheme allows us to apply the following **pair simplification rule** to simplify the notation of pairs:

A dot and the left parenthesis to the right of the dot can be omitted if the item to the right of the dot is a pair. After the left parenthesis is removed, that corresponding right parenthesis must be removed.

For example, in the following pairs:

```
((2 . (8 . 7)) . (4 . 8))
(12 . (2 . (8 . (4 . (3 . 7)))))
```

The dots and parentheses that can be removed according to the rule are underlined. After the removal, the pairs become:

```
((2 8 . 7) 4 . 8)
(12 2 8 4 3 . 7)
```

DrRacket uses the simplified pair notation for any outputs of the pairs. However, you can use either the simplified pair notation or the complete pair notation when you use pairs in your program.

4.4.7 List

Although pairs are capable of representing collections of data, lists, which are almost special cases of pairs, will be more convenient in many situations.

Using BNF notation, a **list** can be recursively defined as follows:

```
<list> ::= null | '()
<list> ::= (cons x <list>), where x can be any Scheme form.
```

The recursive definition starts with the definition of an empty list, which can be represented as `null` or `'()`. The recursive part defines a new list based on an existing list: `(cons x <list>)` produces a new list, where the operator `cons` is the same operator used to produce a new pair and `x` can be any Scheme form. Since lists, except the empty list `'()`, are constructed by the pair construction operator `cons`, all lists, except the empty list `'()`, are in fact pairs.

Based on the definition of lists, we can create the following lists:

```
'()                          ; is a list. It is the empty list.
(cons 4 '( ))                ; returns list (4 . ())
(cons 7 '(4 . ()))           ; returns list (7 . (4 . ())) = (7 4 . ())
(cons 9 '(7 4 . ()))         ; returns list (9 . (7 4 . ())) = (9 7 4 . ())
```

According to the simplification rule of pair representation, the pair `(7 . (4 . ()))` can be simplified to `(7 4 . ())` and pair `(9 . (7 4 . ()))` can be simplified to `(9 7 4 . '())`. Now the question is can we simplify the pair `(4 . ())` to the pair `(4)`? The answer is "no" according to the pair simplification rule, because the empty list is not at all a pair. However, we can introduce another **list simplification rule** to simplify the list notation:

In a list, if to the right of a dot is an empty list, then the dot and the pair of parentheses representing the empty list can be omitted.

By applying this list simplification rule, the list `(4 . ())` is simplified to `(4)`, `(7 4 . ())` is simplified to `(7 4)`, and `(9 7 4 . ())` is simplified to `(9 7 4)`.

An incomplete set of operations defined on list type is given in Table 4.8.

Please note that the accessor functions `car` and `cdr` in Table 4.8 are the same operations for pairs. They work for a list, if and only if the list is a pair. The only situation where a list is not a pair is when the list is an empty list. For all nonempty lists, `car` and `cdr` will work. When car and cdr are applied to a list, `(car x)` will return the first element of the pair, which is the first element in the list. Similarly, `(cdr x)` will return the second element of the pair, which is the residual list when the first element is taken out.

Accessor functions	Meaning	Predicate functions	Meaning
(cons <expr><lst>)	form a new pair	(lst? <expr>)	Is it a pair?
(car <lst>)	return first element	(null? <lst>)	Is list empty?
(cdr <lst>)	return residual list when the first element is removed	(member x <lst>)	Is x a member of list?
(lst<expr> ... <expr>)	return (<expr> ... <expr>)		
		(equal? <lst><lst>)	Are they equal?
(append <lst><lst>)	append two list into one		
(length <lst>)	return the length of list		
(lst-ref <lst><p>)	return element at position p		
(lst-tail <lst><k>)	return the list after removal of the first k elements		

Table 4.8. Operations defined on list type.

The member function `(member x <list>`will return false if x is not a member if the lst. if x is a member of lst, it will return the sub-list from the first x found in lst. This is similar to C, where 0 is false and any nonzero number is considered true. We can define a member that returns true or false only, as shown in the following code:

```
(define (member? element lst)
  (cond ((null? lst) #f)
        ((equal? element (car lst)) #t)
        (else (member? element (cdr lst)))))
```

4.4.8 Application of Quotes

We have seen that we use quotes in symbols, pairs, and lists, for example:

```
(symbol->string 'James)        ; the symbol must be quoted
(cdr (car '((2 . 4) . 5)))     ; the pair must be quoted at the outermost
level
(cons '(2 (5 . 6) 9) '(3 7)) ; the list must be quoted at the outermost
level
```

It is clear that we need to quote a symbol. But it is not so straightforward to understand why and when we need to quote pairs and lists. As we can see from the example above, we only quote the outermost level of a pair or a list. Can we also quote the pairs or lists inside a pair or a list? First examine the following example:

```
(cdr (car '((2 . 4) . 5)))    ; only quote the outermost level
(cdr (car '('(2 . 4) . 5)))  ; quote the inner pair too
```

What is the difference between these forms? To see the difference, we must understand the way Scheme forms are evaluated. Scheme uses the prefix notation. Every form to be evaluated starts with a left parenthesis. The first element, or the element that immediately follows the left parenthesis, is always the operator. The operator will be compared with the operators stored in the operator table. When a match is found, Scheme will perform the operation defined for the operator.

A list and a pair also start with a left parenthesis. To let the language know that lists and pairs are not forms for evaluation and that their first elements should not be considered an operator, we use a quote prior to the left parenthesis. When Scheme sees a quote and a left parenthesis, it knows that the parenthesis does not

start a form, but instead starts a pair or a list. Scheme will then find the corresponding right parenthesis to identify the end of the pair or the list. Obviously, it will not have any operators within a literal list or pair. Thus, there is no need to quote any list or pair within a list or a pair. Some versions of Scheme will report an error if you quote a list or a pair within a list or a pair. The DrRacket will consider the quote character ' to be a separate element and thus

```
(cdr (car '('(2 . 4) . 5)))
(car (car '('(2 . 4) . 5)))
```

will return rather unexpected results:

```
(2 . 4)
quote
```

According to the rules of orthogonality, if we are allowed to quote literals of a pair and list, we should be allowed to quote other types of literals. Scheme indeed allows you to quote literals of any types. For example, you can write '55 (quote number literal), '#t (quote Boolean literal), '"hello" (quote string literal). In these cases, Scheme will simply ignore the quotes. Note that a literal value of any type cannot be a symbol!

Finally, we consider another example.

```
(member 'jim '(3 5 5 Hi 'jim (5 9) "Hello"))
```

The operation and the arguments of the operation look good. The first argument is a symbol. The second argument is a list containing the symbol same symbol, and thus we expect the membership operation returns `('jim (5 9) "Hello"))` or true. Is it right? However, when we execute the operation, it returns false. Why? The reason is that we should not quote anything inside a list. If we do, the quote symbol will be considered a part of the symbol, instead of the indicator of the symbol. You can try the following forms and see what will be returned.

```
(member ''jim '(3 5 5 Hi 'jim (5 9) "Hello"))
  → ('jim '(3 5 5 Hi 'jim (5 9) "Hello")) or true
(member 'jim '(3 5 5 Hi jim (5 9) "Hello"))
  → ('jim '(3 5 5 Hi jim (5 9) "Hello")) or true
```

Notice in the first line of the code above, "jim are two single quotes (apostrophes), not a double quotation.

In summary, what can or must be quoted? What cannot or should not be quoted?

- You can quote a name to make it a symbol.
- You must quote pair and list literals at the outermost level to differentiate them from forms to be evaluated.
- You must not quote pairs, lists, or anything inside a pair or a list.
- You must not quote forms that you want to evaluate. If you quote a form, the form simply becomes a list. The operator in the first place will be considered the first element of the list.
- You may quote number, string, and Boolean literals, but it makes no difference. Thus, do not quote them.

4.4.9 Definition of procedure and procedure type

A **procedure** is a user-defined function that extends the predefined primitives in the language system. The set of procedures can be considered to form a special data type: The **procedure type**. The data values of the procedure type are all possible procedures. The operations defined on the procedure type are the higher-

order functions. A **higher-order function**, or **higher-order procedure**, is a procedure that can take the operation of another procedure as its parameter. Please note that a procedure that takes the return value of another procedure is not a higher-order procedure. In functional programming languages, functions (procedures) are first class objects. A procedure can be put in any place where a value is expected and the procedure's return value will be used as the value that is expected. Thus, every procedure in Scheme that has a parameter can take the return value of another procedure. The higher-order procedure must take the operation of another procedure as a parameter. Higher-order procedures will be discussed later in the chapter.

The syntax graph of procedure definition is given in Figure 4.4.

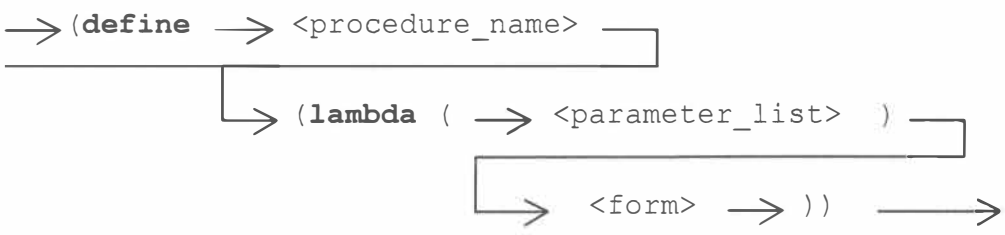


Figure 4.4. Syntax graph of procedure definition.

Using the syntax, we can define a procedure as follows:

```
(define maximum (lambda (x y)
  (if (> x y)
      x
      y
)))
```

The definition uses two keywords: define and lambda. There is a simplified definition that uses only one keyword. The syntax graph of the alternative procedure definition is given in Figure 4.5.

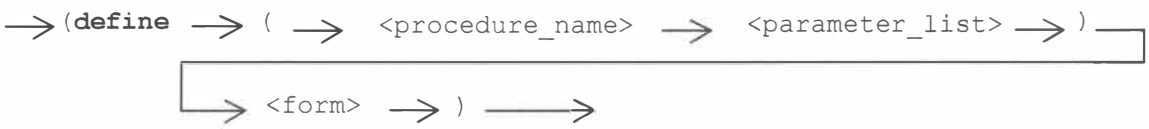


Figure 4.5. Syntax graph of the simplified procedure definition.

Using the simplified definition, the maximum procedure can be defined by

```
(define (maximum x y)
  (if (> x y)
      x
      y
  )
)
```

We will use the two ways of defining procedures interchangeably in the text.

4.4.10 Input/output and nonfunctional features

Scheme has a simple input and output mechanism. The form `(read)` will wait for keyboard input and returns the input value. The inputted value will be interpreted as a symbol. You can convert the symbol to other types as required.

Output can be done by any one of these forms:

```
(display 3)
(write 3)
(print 3)
```

None of these forms prints on a new line. You can use form `(newline)` to change to a new line.

The following Scheme program reads a symbol, converts it to a string, appends a space at the end, reads another symbol, converts it to a string, and appends it to the first string. Finally, the string with a space is displayed.

```
(display
  (string-append
      (string-append (symbol->string (read)) " ")
           (symbol->string (read))
  )
)
```

The following is a more complex example that uses input and output. In this example, we implement the menu function and the branching function we implemented in C/C++.

The menu function takes input from the keyboard, and compares the entered selection with existing options. If a match is found, the program branches to the corresponding function.

```
(define menu (lambda ()  ; no parameter
  (begin
      (newline) (newline)    ; print two newlines
      (display "enter your selection") (newline)
      (display "i: insert a new entry") (newline)
      (display "s: search an entry") (newline)
      (display "d: delete an entry") (newline)
      (display "p: print all entries") (newline)
      (display "q: quit") (newline)
      (let((c (read)))  ; read a symbol from keyboard and assign it to c
           (if (eq? c 'q)
               (display "END.")
               (begin
                    (branching c)   ; call branching below
                    (menu)          ; recursive call to itself
               )
           )
))))

(define branching (lambda (sel)
```

```
    (begin
        (cond
            ((eq? sel 'i) (display "inserted ..."))
            ((eq? sel 's) (display "searched ..."))
            ((eq? sel 'd) (display "deleted ..."))
            ((eq? sel 'p) (display "printed ..."))
            (else (display "invalid input ..."))
)))     )
(menu)
```

There are several forms in the program that we have not discussed:

```
(let ( (name value))
      body
)
```

The let-form assigns the `value` to the `name`, and then executes the `body` part of the code. The let-form will be discussed later in detail.

In the procedure `(menu)`, `(menu)` itself is called. A procedure that calls itself is a **recursive procedure**. Loops are features of imperative languages. Functional programming languages mainly use recursion to implement the functions of loops. Many more recursive procedures will be discussed in the rest of the chapter.

Now we will discuss some of the nonfunctional features in the program. The form

```
(begin
  form1
  ...
  formn
)
```

allows sequential execution of a sequence of independent forms. Sequential execution is really a feature of the imperative programming paradigm. As you can see, Scheme does have some imperative features that make programming easier.

Another imperative feature is that the form `(display x)` does not return a value. For example, if you execute:

```
(+ (display 5) 7)
```

and expect the form to return 12, you will receive an error message instead:

```
Error: addition expects type <number> as 1st argument
```

The reason is that `(display 5)` only prints 5 on the screen, but it does not return any value.

However, we can define a function that performs output and returns a value, as shown in the following procedure definition:

```
(define writeln (lambda(x)
  (begin
      (display x)
      (newline)
```

```
        x
    )
))
```

For example, if we call the procedure:

```
(+ (writeln 5) 4)
```

It will print 5, and then pass the value 5 to the next form that adds it to 4 and returns 9.

Please note that in the implementation of the `writeln` procedure, we used the nonfunctional feature of sequential execution quoted by the `(begin ...)` form.

*4.5 Lambda-calculus

The **λ-calculus** (lambda-calculus) is a formal mathematical system devised by Alonzo Church in 1934 to investigate universal computing models, functions, function application, and recursion. It has influenced many computing systems and programming languages, especially the functional programming languages. Lisp was the first programming language based on λ-calculus. Scheme (a dialect of List), Haskell, Miranda, SML, and ML are more recent functional programming languages based on the mathematical system.

The λ-calculus is analogous to Turing machines. However, λ-calculus is much easier to understand and much simpler to use. Turing machines can be considered the most basic assembly language based on the simplest 1-bit computer architecture, whereas λ-calculus is a super high-level language that can be conveniently learned and applied to solve various kinds of complex problems.

In this section, we briefly discuss the structure (lexical, syntactic, and semantic) of λ-calculus, so that we can better understand the Scheme programming language that is based on λ-calculus. You will shortly see that the Scheme language is strictly based on λ-calculus. If you know λ-calculus, you know how to write Scheme programs.

4.5.1 Lambda-expressions

The structure of the λ-calculus is short and simple. At the lexical level, there are only three lexical units: λ, the parentheses "(" and ")," and an infinite list of variables (names), e.g., a, b, a1, a2, ... , etc.

At the syntactic level, a λ-expression is a finite combination of lexical units and variables. Using BNF notation, a simplified λ-expression can be defined by

```
λ-expression ::= <constant>    | <variable> | <expression>
                               | λ<variable>(< λ-expression>)
                               | (<λ-expression><λ-expression>)
```

In the definition, `constant` is a value of any data type. The definitions of the `variable` (`identifier`) and the `expression` have been discussed in Chapter 1. According to the definitions, the following expressions are λ-expressions:

```
5                ; a constant is a λ-expression
x                ; a variable is a λ-expression
x+y              ; an expression is a λ-expression
λx(x+y)          ; λ<variable>(< λ-expression>) is a λ-expression
```

```
λx(x+y) 5          ; (<λ-expression><λ-expression>) is a λ-expression
```

4.5.2 λ-procedure and parameter scope

One of the λ-expressions, λ<variable><λ-expression>, is called a **λ-procedure**. The variable prefixed by λ is called the **parameter** of the procedure and the λ-expression that follows the parameter is called the **body** of the procedure. The **scope** of the parameter is the body of the λ-procedure. For example, if we have a λ-expression

```
λx(x+y) (x+3)
```

the scope of the parameter x in λx is in and only in the body (x+y). It does not cover the x in (x+3).

An occurrence of a variable x in a λ-expression is **bound** if it is within the scope of a parameter in λx. An occurrence of a variable x in a λ-expression is **free** if it is not within the scope of a parameter in λx. An occurrence of a parameter x binds all free occurrences of x within its scope.

Given the following λ-expression

```
λx(+ (/ λx(* x 2) 8 λx(- x 1) 5) (* λx(+ x 2) 3 x) ) 7
```

How many λ-procedures are contained in the λ-expression? What are the scopes of different parameters?

Each λx corresponds to a λ-procedure, and, thus, the λ-expression contains four λ-procedures. The scope of each parameter is underlined in the following expression:

```
λx(+ (/ λx(* x 2) 8λx(- x 1) 5) (* λx(+ x 2) 3x) ) 7
```

In the next subsection, we will discuss reduction rules that evaluate such complex expressions to a simple value, or the return value of the expression.

4.5.3 Reduction rules

The process of evaluating a λ-expression is called a **reduction**. We will briefly discuss three reduction rules that transform a λ-expression into a simpler λ-expression. They are alpha (α) reduction, beta (β) reduction, and eta (η) reduction. Repeatedly applying these reduction rules transforms a λ-expression to a simple value, which is the return value of the expression. The reduction rules define the semantics of λ-calculus.

(1) The alpha (α) reduction

```
λx(E)  ⇔   λy([y/x]E)
```

The α-reduction rule says that for a λ-expression with a parameter x, we can substitute y for parameter x and all the occurrences of x in its scope.

The α-reduction rule allows us to freely choose and change parameter names for convenience.

For example, in the following expression, different parameters have the same name.

```
λx(+ (/ λx(* x 2) 8 λx(- x 1) 5) (* λx(+ x 2) 3 x) ) 7
```

Although the expression is not ambiguous for the computer that evaluates it, it is simply easier for humans to understand if we choose different names for different parameters. Thus, we can apply α-reduction rule to rename the parameters in the λ-expression as follows:

```
λx1(+ (/ λx2(* x2 2) 8 λx3(- x3 1) 5) (* λx4(+ x4 2) 3 x1) ) 7
```

(2) The beta (β) reduction

```
λx(E1) E2  ⇔  [E2/x]E1
```

The β-reduction rule says that we can remove the parameter `x` for `E1` if we substitute the λ-expression `E2` for all the occurrences of `x` in `E1`. The λ-expression `E2` is called the argument to the λ-procedure `λx(E1)`.

The β-reduction rule defines how to perform parameter passing in a λ-procedure to reduce the complexity of an λ-expression that contains λ-procedures.

Now we can repeatedly apply the β-reduction rule to the following λ-expression:

```
λx1(+ (/ λx2(* x2 2) 8 λx3(- x3 1) 5) (* λx4(+ x4 2) 3 x1) ) 7
```

We assume we use lazy evaluation, that is, we proceed from outermost first (underlined part). Thus, the first step is to substitute argument 7 for `x1`, resulting in the following simplified λ-expression:

```
(+ (/ λx2(* x2 2) 8 λx3(- x3 1) 5) (* λx4(+ x4 2) 3 7) )
```

Then the three λ-procedures are at the same level and we substitute their arguments for their parameters, respectively:

```
   (+ (/ (* 8 2)λx3(- x3 1) 5)) (* λx4(+ x4 2) 3 7))
⇒ (+ (/ (* 8 2) (- 5 1)) (* λx4(+ x4 2) 3 7))
⇒  (+ (/ (* 8 2) (- 5 1)) (* (+ 3 2) 7))
⇒  (+ (/ 16 4) (* 5 7))
⇒  (+ 4 35)
⇒  39
```

Having completed the parameter passing, the λ-expression has become a simple mathematical expression that can be easily evaluated to 39.

(3) Eta (η) reduction

```
λx(E) ⇔ E, if x does not appear in E.
```

For example, assume we have a λ-expression:

```
λx1(* λx2(+ x2 2) 3)
```

If parameter `x1` does not appear in the λ-expression in `x1`'s scope, we can then remove `λx1` according to η-reduction. In words, η-reduction says that, if a parameter is not used in a procedure, it should be removed from the parameter list.

In fact, in beta reduction we have implicitly applied η-reduction: After we substitute the argument for the parameter `x`, there are no longer appearances of `x` in the expression and thus `λx` is removed.

4.6 Define your Scheme procedures and macros

Now we come back to Scheme. In this section, we will discuss the definition of procedures, scope of parameters, and global and local variables. We will also discuss the macro that is related to procedure.

4.6.1 Unnamed procedures

Having studied λ-expressions, it is easy to write Scheme procedures and understand how procedures are evaluated. Consider the λ-expression:

```
λx(+ (/ λx(* x 2) 8 λx(- x 1) 5) (* λx(+ x 2) 3 x)) 7
```

To write a Scheme procedure that is equivalent to the expression, all we need to do is to use the Scheme keyword "lambda" to replace "λ" and add necessary parentheses. Thus, we have

```
lambda x(+(/lambdax(* x 2) 8 lambdax(- x 1) 5)(* lambdax(+ x 2) 3 x))7
```

After adding necessary parentheses, we have a proper Scheme procedure that evaluates to 39.

```
((lambda (x)
   (+   (/    ((lambda (x)(* x 2)) 8)
              ((lambda (x)(- x 1)) 5))
        (*    ((lambda (x)(+ x 2)) 3) x)
   )
 )
 7
)
```

There are four procedures in the code above. None of them is given a name. Such procedures are called **unnamed procedures**.

4.6.2 Named procedures

The problem with the unnamed procedures is that we cannot call the procedure multiple times to obtain the advantage of code reuse. Embedding one procedure in another procedure may compromise the readability of the code. The solution to these problems is to name the procedures and use the names to call the procedures. A **named procedure** is defined by using the keyword "define" to associate an unnamed procedure with a name. Using named procedures, we can rewrite the code as follows:

```
(define foo1 (lambda (x)(* x 2)))
(define foo2 (lambda (x)(- x 1)))
(define foo3 (lambda (x)(+ x 2)))
(define bar (lambda (x)
   (+   (/ (foo1 8) (foo2 5))
        (* (foo3 3) x))
))
(bar 7)
```

4.6.3 Scopes of variables and procedures

In Scheme, any names defined by the keyword "define" are global. For example,

```
(define size 100)
(define foo (lambda (x)(* x 2)))
(define bar (lambda (x) ( … )))
(define writeln (lambda (x) ( … ))
```

A global name can be accessed in the entire program. For example, the value of the global variable `size` can be used in any other procedures. Please note that although we call `size` a variable, it is not a memory location to which we can reassign a value. It is really a named value that we can use but cannot change.

As we have seen, a parameter of a procedure is local. Its scope is only within the body of the procedure. For example, the three procedures `foo`, `bar`, and `writeln` use the same parameter name, but they have different scopes and thus will not cause name conflict. Now the question is can we define a local variable?

Scheme offers a **let-form** to accommodate the needs of local variables. The syntax graph of let-form is given in Figure 4.6.

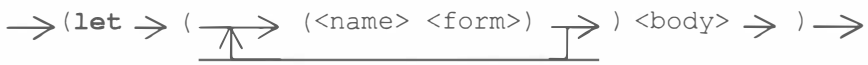


Figure 4.6. Syntax graph of let-form.

A let-form consists of a list of `(<name><form>)` pairs and a body part. In each pair, the `name` is associated with the `form`. The `body` is any form. For example:

```
(let ((a 3) (b 4)) (+ (* a a) (* 2 a b) (* b b)))
```

In this let-form, `a` is associated with 3, and `b` is associated with 4. The `body` is the form:

```
(+ (* a a) (* 2 a b) (* b b))
```

The names defined by a let-form are called **local variables**. The scope of the local variables in the let-form is only within the body part of the form. It is different from imperative languages where the scope of a variable is from the declaration to the end of the block. We can see the difference by the example below.

```
(let   ((x 5)
        (y (* x 4)))
  (+ x y)
)
```

What is the return value of the form? It is not 25, but the following error:

```
reference to undefined identifier: x
```

The problem is in the second local variable where we try to associate `y` with `(* x 4)`. However, the scope of `x` does not start from the declaration (association). Its scope is only in the body part of the let-form. Thus, it is undefined.

To accommodate the expectations of some imperative programmers, some Scheme versions added the let*-form to allow the scope of a local variable to start from its association. Thus,

```
(let*  ((x 5)
        (y (* x 4)))
  (+ x y)
)
```

will return 25 as imperative programmers expected.

To build larger programs, a Scheme program can be divided into modules and further limit the scope of global variables and procedures within a module. Names can be made visible outside a module using the **export** form. The following example shows the definition of a Scheme module and the export form:

```
(module module-name
  (export name1 name3 name5 … namen) ; names visible outside the module
  (define name1 value1)              ; define a global variable
  (define name2 (lambda (x) (…)))    ; define a global procedure
  (define name3 …)
  …
)
```

In the module definition, names name1name3name5, … are accessible outside the module, while names name2name4, … are not accessible outside the module.

Now consider a secure email system that consists of several modules. In the encryption module, only the procedure string-encryption is exported and accessible from the outside.

```
(module encryption
  (export string-encryption)                          ; outside accessible
  (define character-rotation (lambda (ch)
  (define character-encryption (lambda (ch) (...)))
  (define string-encryption (lambda (str) (…)))
  (define encryption-helper (…))
  ( … )
)
```

Then, in another module, say, secure-email, the string-encryption can be called, as shown below:

```
(module secure-email
  (define load_file (lambda (str filename) (...)
   (string-encryption (load_file (str filetext)))
  ...
)
```

4.6.4 Let-form and unnamed procedures

Let-forms and unnamed procedures are, in fact, equivalent: They can be converted from one to the other. The general format of let-forms is

```
(let
  (   (name1  value1)
      (name2  value2)
      . . .
      (namen  valuen)
  )
  body
)
```

It can be mechanically translated into the unnamed procedure:

```
((lambda (name1 name2 . . . namen)
  body)
  value1 value2 ... valuen)
```

For example, the let-form

```
(let
   (    (a 3)
        (b 4)
   )
   (+ (* a a) (* 2 a b) (* b b))
)
```

can be translated into the following unnamed procedure:

```
( (lambda (a b)
   (+ (* a a) (* 2 a b) (* b b))
   )
   3 4
)
```

Now we examine the let-form with incorrect scope

```
(let   ((x 5)
        (y (* x 4)))
   (+ x y)
)
```

If we translate it into unnamed procedure, we have

```
( (lambda (x y)
        (+ x y)
   )
   5 (* x 4)      ; this x is unbound
)
```

From the unnamed procedure, we can see more clearly that the variable `x` in the argument `(* x 4)` is not initialized.

4.6.5 Macros

Macro in Scheme is a pattern-based replacement process. It replaces any code (implementation) that matches a pattern in such a way that an expansion is made. It substitutes the parts of code for the parts in the pattern that they match. A macro in Scheme has similar meaning as a macro in other programming languages like C and C++. A macro introduces a name substitution instead of a control flow change.

The definition of Scheme macros is similar to the definition of procedures. All you need to do is to change the keyword **define** to the keyword **define-macro**.

The following definition defines a macro that computes the cube (the third power) of a number:

```
(define-macro cube (lambda (x) (* x x x)))
```

If we call the macro by `(cube 5)`, it returns `125`. It looks like the macro works exactly like a procedure.

Now consider another use of the same macro in the following program:

```
(define i 5)
(cube i)          ; to be replaced by (* 'i 'i 'i)
```

If `cube` is defined as a procedure, the code should work fine. However, `cube` is defined as a macro. When we execute the code, we have the following error message:

```
expects type <number> as 1st argument, given: i;
```

To understand why this error happens, we need to review the idea of macro: name substitution instead of a control flow change. Before we call `(cube i)`, the call has been replaced by the body of the macro definition; thus, what we really execute is:

```
(* 'i 'i 'i)
```

The system is smart enough to consider the first name as an operator and thus not consider it as a symbol. Thus, what the macro is trying to do is to multiply three symbols.

Why does the call `(cube 5)` work? The call will be replaced by

```
(* '5 '5 '5)
```

Since numbers cannot be symbols, they will be evaluated to numbers. Thus, the form is the same as

```
(* 5 5 5)
```

To make your macro also work for named values, you can use the list function in your definition.

```
(define-macro cube (lambda (x) (list * x x x)))
```

Using this definition, (`cube i`) will be replaced by

```
(list '* 'i 'i 'i)
```

which produces a list `(* i i i)`. Since i has been defined to be 5, `(* i i i)` will evaluate to 125.

Similar to the pattern design in C++ generic class, some Scheme implementations, such as DrRacket Scheme, also allow the design pattern-based macros, where a generic code pattern is defined, and the generic pattern can be replaced by a concrete code. For example, the following Scheme code defines a pattern `(_ x ...)` and the pattern is substituted by the concrete code `(+ x ...)`, where the generic operator is replaced by the addition operator +:

```
(define-syntax addition
    (syntax-rules ()
      ((_ x ...)
        (+ x ...))))
(addition 1 2 3 4)    ; Call the macro-based procedure definition
> 10
```

The same design pattern can be used for a different operation by replacing the generic operator by the multiplication operator *:

```
(define-syntax product
    (syntax-rules ()
      ((_ x ...)
        (* x ...))))
(product 1 2 3 4)       ; Call the macro-based procedure definition
> 24
```

4.6.6 Compare and contrast imperative and functional programming paradigms

Having covered the basic functional programming concepts, we will now compare and contrast the imperative and functional programming paradigms again using an example.

In this example, we design a vehicle's Anti-lock Braking System (ABS). The requirement and specification of the ABS example are as follows:

```
Requirement:
  To obtain the maximum braking effect
Specification:
  Define (or measure) the wheel diameter;
  Measure the wheel rotations per second rps;
  Compute the wheel velocity wv;
  Measure the body velocity bv;
  Error detection and action:
      if (bv > wv), reduce braking force
      else  if (bv < wv), reduce acceleration force
         else "no action"
```

The following code is the C++ implementation using modular design, that is, we try to put related code that performs a coherent job into a module or a function.

```
#include <iostream>
using namespace std;
const float mile_inch = 63360.0;  // inches per mile
const float pi = 3.1416;
float wheel_diameter = 15;        // inches
float wheel_sensor() {
  float rps;
  cout << "get rotations per second: " << endl;
  cin >> rps;
  return rps;
}
float wheel_velocity(float rps) {
  float wv;
  wv = (pi * wheel_diameter * rps * 3600)/mile_inch;
  return wv;
}
float body_velocity() {
  float bv;
  cout << "get miles per hour: " << endl;
  cin >> bv; return bv;
}
void error_detection(float wv, float bv) {
  if  (abs(bv - wv) < 0.01)  // 0.01 is the tolerance
          cout << "no action" << endl;
  else
```

```
        if   (bv > wv)
             cout << "reduce brake force!" << endl;
        else
             cout << "reduce acceleration force!" << endl;
}
void start_engine () {
   float rps, wv, bv;
   rps = wheel_sensor();
   wv = wheel_velocity(rps);
   bv = body_velocity();
   error_detection (wv, bv);
}
void main() {
   start_engine();
}
```

A very similar Scheme program can be constructed using the modular design. For every C++ function, we define a global procedure.

```
(define mile-inch 63360.0)
(define pi 3.1416)
(define wheel-diameter 15)    ; inches
(define wheel-sensor (lambda ()
  (begin    (display "get rotations per second: ")
            (read)) ))
(define wheel-velocity (lambda (rps) ; miles per hour
  (/   (* pi wheel-diameter rps 3600)
       mile-inch)  ))
(define body-velocity (lambda ()
   (begin (display "get miles per hour: ")
       (read)) ))
(define error-detection (lambda(wv bv)
  (if  (<  (abs (- bv wv)) 0.01)  ; 0.01 is the tolerance
       (write "no action")
       (if   (>  bv wv)
             (write "reduce brake force!")
             (write "reduce acceleration force!")))   ))
(define start-engine (lambda ()
  (error-detection (wheel-velocity (wheel-sensor))
       (body-velocity) )))
(define main (lambda ()
  (start-engine)))
(main)
```

Carefully examining and comparing the two implementations, we arrive at the following observations. Both programs consist of functions (procedures). The programs and the functions are organized in the same way as much as possible. However, the two programs are still very different:

- The Scheme program does not allow variables and there is no need to declare names.
- All Scheme procedures must return a value, while C++ functions may or may not return a value.
- The return values of C++ functions have to be stored in temporary variables, while Scheme procedures can be placed in the positions where the return values are needed. This difference can be seen clearly, for example, in the "start engine" functions in these two programs.
- The Scheme program is much more compact than the C++ program.

The Scheme program above used global procedures although those procedures are in fact local. In the following program, the four procedures `wheel-sensor`, `wheel-velocity`, `body-velocity`, and `error-detection` are defined as local procedures in let-forms.

```
(define start-engine (lambda ()
  (let (
  (wheel-sensor (lambda ()
          (begin   (display "get rotations per second: ")
                   (read)) ))
      (wheel-velocity (lambda (rps) ; miles per hour
           (let (    (pi 3.1416)
                     (mile-inch 63360)
                     (wheel-diameter 15))
                (/ (* pi wheel-diameter rps 3600) mile-inch))))
      (body-velocity (lambda ()
           (begin (display "get miles per hour: ")
                   (read)) ))
      (error-detection (lambda(wv bv)
           (if  (<  (abs (- bv wv)) 0.01)
                (write "no action")
                (if  (>  bv wv)
                     (write "reduce brake force!")
                     (write "reduce acceleration force!")))))
       )
  (error-detection (wheel-velocity (wheel-sensor))
      (body-velocity) ))))
(define main (lambda ()
  (start-engine)))
(main)
```

4.7 Recursive procedures

This section continues the discussion on this topic in Chapter 2 and uses more examples to illustrate the fantastic-four abstract approach of understanding and writing recursive procedures:

1. Formulate the size-n problem;

2. Find the stopping condition and the corresponding return value;
3. Select `m` and formulate the size-m problem; and
4. Construct the solution of the size-n problem from the assumed solution of the size-m problems.

Please read Section 2.7 before continuing with this section. The fantastic-four abstract approach is generic. It can be applied in different programming paradigms. In this section, we use the Hanoi Towers problem to illustrate the fantastic-four abstract approach in the functional programming paradigm.

1. Formulate the size-n problem

We can simply formulate the size-n problem as `hanoi(n)` or `(hanoi n)` in prefix notation. We can also formulate the problem as `(hanoitowers n left center right)`, allowing the user to name the pegs. The solution of the return value of this function is to print instructions (steps) that move `n` disk from the `left` peg, using `center` peg as auxiliary, to the `right` peg.

2. Find the stopping condition and the corresponding return value

The stopping condition is `n = 1`. In this case, the size-1 problem is `(hanoitowers 1 left center right)`, and the solution or return value is "move the disk from the `left` peg to the `right` peg."

3. Select m and formulate the size-m problem

Since we can only move one disk at a time, it is obvious that we can only reduce the size by one in one iteration. Since we have multiple parameters in the problem, we have multiple size-(n−1) problems. As explained in Section 2.7, we need the following two size-(n−1) problems:

(1) move `n-1` disks from `left` to `center`, using `right` as auxiliary:

```
hanoitowers(n-1, left, right, center)
```

(2) move `n-1` disks from `center` to `right`, using `left` as auxiliary:

```
hanoitowers(n-1, center, left, right)
```

4. Construct the solution of the size-n problem

Use the solutions for the size-(n−1) problems to construct the solution for the size-n problem.

The text on the left-hand side in Figure 4.7 shows how we construct the solution for the size-n problem based on the solutions for the size-(n−1) and size-1 problems, that is,

```
hanoitowers(n-1, left, right, center)      // move n-1 disks left -> center
hanoitowers(1, left, center, right)     // move 1 disk left -> right
hanoitowers(n-1, center, left, right)      // move n-1 disks left -> center
```

In words, the solution for the size-n problem is: (1) Move `n-1` disks from `left` peg to the `center` peg. We simply assume that we can do it, because it is a size-(n−1) problem. (2) Move the remaining disk from `left` to `right`. (3) Move `n-1` disks from `center` to the `right`.

Based on the discussion, we can directly obtain the Scheme program that solves the Hanoi Towers problem as follows. In the program, we defined a one-parameter procedure `(hanoi n)` as a simpler user interface. The procedure with more parameters is given as a helper procedure.

```
(define hanoi (lambda (n)    ; define a simpler human-interface
  (hanoitowers n "Left" "Center" "Right")
))
```

```
(define hanoitowers (lambda (n source spare destination)
  (if (= n 1)                          ; stopping condition
      (begin
          (display "move top from ")   ; output at stopping condition
          (display source)
          (display " to ")
          (display destination)
          (newline)
      )
      (begin                           ; from size-(n-1) to size-n problem
          (hanoitowers (- n 1) source destination spare)
          (hanoitowers 1 source spare destination)
          (hanoitowers (- n 1) spare source destination)
      )
  )
))
```

If we call the procedure `(hanoi 3)`, we will have the following output describing how to solve the size-3 Hanoi Towers problem.

```
move top from Left to Right
move top from Left to Center
move top from Right to Center
move top from Left to Right
move top from Center to Left
move top from Center to Right
move top from Left to Right
```

As you can see from the example, the most important idea of recursive procedure is that we simply assume that we have the solution for size-(n−1) problem and we do not need to solve it. Why does it work? Because the recursive mechanism will actually solve the problem from size-1 upward; that is, it will solve the size-1 problem, then use the solution of the size-1 problem to construct the solution of the size-2 problem, and so on. Since we have given the solution of the size-1 problem and we have defined how to find the solution from the size-(n−1) to the size-n problem, we basically have given solutions to the problem of all sizes!

4.8 Define recursive procedures on data types

In this section, we continue to study recursion. We discuss recursive procedures that manipulate numbers, characters, strings, pairs, and lists. The purposes of this section are twofold: understand recursion and become familiar with data types in Scheme. In the discussion, we may or may not explicitly mention the four steps of writing recursive procedures. However, if you still do not fully understand the idea of writing recursive procedures, you should carefully go through these examples and try to identify the four steps and application-specific parts of the procedures.

4.8.1 Number manipulations

We will study several recursive procedures that manipulate decimal and binary numbers.

(1) Addition and Multiplication

We start with a few simple procedure definitions. We first define an add procedure that adds two numbers together.

```
(define Add (lambda (x y)
  (+ x y)
))
```

Then, we write a recursive procedure to compute the square using a sequence of additions, based on the mathematical formula $n^2 = 1 + 3 + \ldots + 2n - 1$. The procedure can take both positive and negative numbers as input.

```
(define Square1 (lambda (x)  ; Solution 1
  (if (= x 0)
      0
      (Add (Square1 (- (abs x) 1)) (Add (- (abs x) 1) (abs x))))
))
(define Square2 (lambda (x) ; Solution 2
  (if (= x 0)
      0
      (if (< x 0)
          (Square2 (- x))
          (Add (Square2 (- x 1)) (Add (- x 1) x))))
))
(define square3 (lambda (x) ; Solution 3
    (square-helper x x)
))
 (define square-helper (lambda (x n)
      (if (= n 0)
      0
      (if (< x 0)
          (square-helper (- x) (- n))
          (Add (square-helper x (- n 1)) x)))
))
```

All three solutions use recursion. Please identify the four steps of the fantastic approach used in these three procedures.

(2) Decimal-binary conversion

How do we convert a decimal number to a binary number? One of the algorithms frequently used is shown in Figure 4.7. We divide the decimal number by 2, and keep dividing the quotient by 2, until the quotient becomes 0. The remainder in each step of division forms the binary number that is equivalent to the decimal number.

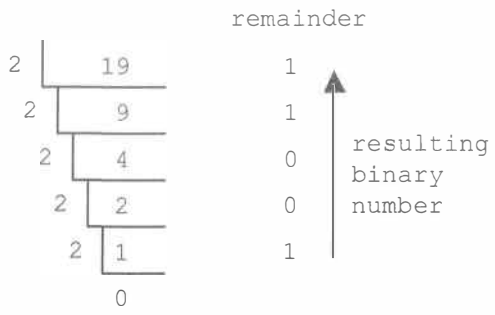


Figure 4.7. Converting a decimal number to a binary number.

Following the same algorithm, we can devise a Scheme program to convert a decimal number in the list format [e.g., the list format of 354 is (3 5 4)], into the equivalent binary number in list format.

```
(define dtob (lambda (n)
  (if (= n 0)                          ; stopping conditions
      (list 0)                         ; return value at stopping
      (append   (dtob (quotient n 2))  ; size-m problem and from
                (list (remainder n 2)) ; size-m to size-n solution
      )
  )
))
```

The output of the procedure call `(dtob 19)` is

```
(0 1 0 0 1 1)
```

The stopping condition is `n = 0` and the corresponding return value is a list `(0)`, because the binary number corresponding to decimal number 0 is also 0. The size-n problem is `(dtob n)` and the size-m problem is `(dtob (quotient n 2))`, where `m = (quotient n 2)`, because the quotient of `n/2` is smaller than `n`, and, thus, the size of the problem will eventually be reduced to the stopping condition.

Constructing the solution of the size-n problem is done by appending the current remainder of `n/2` to the solution of the size-(n−1) problem:

```
(append (dtob (quotient n 2)) (list (remainder n 2)))
```

(3) Binary addition

How do we add two binary numbers arithmetically? The algorithm that we use is shown in Figure 4.8. We add from right to left. A carry will be generated if the result of addition at a position is greater than or equal to 2. The final result may have one more bit than the two numbers to be added.

The following Scheme program mimics the addition process. The program consists of multiple procedures, and all procedures are recursive.

```
(define binaryadd (lambda(L1 L2)
  (let ((len1 (length L1)) (len2 (length L2)))
     (if   (> len1 len2)
           (binaryadd L2 L1)
           (if  (< len1 len2)
```

```
                (binaryadd (append '(0) L1) L2)
                (recursiveAdd (append '(0) L1) (append '(0) L2) 0)
))))    ))
(define recursiveAdd (lambda(L1 L2 carry)
  (if (null? L1)
      '()
      (let ((t (+ (tail L1) (tail L2) carry)))
           (append  (recursiveAdd (rmtail L1)
                                  (rmtail L2)
                                  (quotient t 2))
                    (list (remainder t 2))
))))    )     )
```

```
          1 0 0 1 1
    +     1 1 0 0 1
  ---------------------
carry   1 0 0 1 1 0
result  1 0 1 1 0 0
          <-----------
```

Figure 4.8. Adding two binary numbers.

In the main procedure, the procedure that is called first, we first define two local variables `len1` and `len2` to represent the lengths of the two binary numbers in list format. The size-n problem that the procedure is dealing with is `(binaryadd L1 L2)`, where `n` is the absolute value `|length(L1)-length(L2))|`. The procedure exits the recursive call (stopping condition) when `len1=len2`, or when `n=0`. If `len1>len2`, we recursively call `(binaryadd L2 L1)`, which means we swap the positions of `L1` and `L2`. In other words, we always use the shorter number as the first argument. If `len1<len2`, we recursively call `(binaryadd (append '(0) L1) L2 L1)`, which means we add one 0 to the left of the shorter number, attempting to make the two numbers to be added the same length. We keep adding 0 until the two numbers have same length in their list format.

When `len1=len2`, we exit the main procedure and call the `recursiveAdd` procedure. We add a 0 to the left of both numbers to handle the situation when the addition result takes one extra bit. Again, the `recursiveAdd` procedure is recursive. The size-n problem that `recursiveAdd` is dealing with is `(recursiveAdd L1, L2, carry)`, where `n` is the length of lists `L1` and `L2`. The procedure stops when the lists become empty, or the length becomes 0. The size-(n−1) problem is `(recursiveAdd (rmtail L1) (rmtail L2) (quotient t 2))`, where `(rmtail L)` returns the list without the last element of `L` or having the last element of `L` removed. Thus, the size (length of the lists) of the problem becomes `n-1`. In the program, the addition of three binary bits is done in the definition of the local variable `t`: `(let ((t (+ (tail L1) (tail L2) carry)) ...)`, where, the procedure `(tail L)` returns the last element of the list `L`.

The implementations of `(rmtail L)` and `(tail L)` are not given here. You are asked to complete the recursive procedures as exercises. The procedures will be recursive, and you can follow the four-step abstract approach to implement them.

(4) Two's complement

We can obtain two's complement of a binary by using its one's complement plus one at the end (the least significant bit). The addition is an arithmetic addition and may cause a carry to the higher bit. We can use the `binaryadd` procedure that we just defined for this purpose. The one's complement can be obtained by inverting each bit. Again, we assume the binary numbers we are dealing with are in their list format.

```
(define twoscomplement (lambda (x)
  (binaryadd '(1) (onescomplement x))
))
 (define onescomplement (lambda (x)
  (if (null? x)
      '()
      (if (= (car x) 0)
           (cons 1 (onescomplement (cdr x)))
           (cons 0 (onescomplement (cdr x)))))))
```

The procedure `onescomplement` is recursive. The stopping condition is when the list is empty. The corresponding return value is empty list `'()`. The size-n problem is `(onescomplement x)`, where `x` is a list with `n` elements. The size-(n−1) problem is `(onescomplement (cdr x))`, where `(cdr x)` is a list with `n-1` elements.

4.8.2 Character and string manipulations

In Chapter 2, we wrote a C program to encrypt a string. Now we use Scheme to implement a similar string-encryption program.

Assuming `str` is a string of length `n`, we would like to add an integer `k` to the integer value of each character in the string. The integer value of each character is given in the ASCII table in Appendix C.

The idea of the string-encryption is as follows:

If `str` is an empty string "" (stopping condition), then return empty string. There is no character to encrypt.

The size-n problem is `(string-encryption str)`, where str has `n` characters.

We reduce the size-n problem to a size-(n−1) problem by removing the first character from the string.

We construct the solution of the size-n problem (string-encryption str) by encrypting the first character of `str`, and append the solution of the size-(n−1) problem to the encrypted character.

Before we write the main procedure `(string-encryption str)`, we need to write a few helper procedures:

(character-encryption ch) encrypts a character;

`(string-car s)` returns the first (leftmost) character of string `s`;

`(string-cdr s)` returns the substring of s after removing the first character.

The Scheme program including all necessary procedures is given as follows. The basic operations defined on characters and strings have been given in Tables 4.4 and 4.5.

```
(define string-encryption (lambda (str key)
  (if (string=? str "")      ; stopping condition
```

```
        ""                          ; return empty string
        (string-append
             (character-encryption (string-car str) key)
             (string-encryption (string-cdr str) key)
        )
   )
))
(define character-encryption (lambda (ch k)
   (string (integer->char (+ (char->integer ch) k)))
))
 (define string-car (lambda (s)
   (string-ref s 0)   ;return the element at position 0
))
(define string-cdr (lambda (s)
   (substring s 1 (string-length s)) ;return the element at position 0
))
```

To decrypt the string, we can use the following decryption program. The program calls the encryption program with a negative key value to reverse the encryption.

```
(define string-decryption (lambda (str key)
   (string-encryption str (- key))
))
```

We can add different features to the encryption program. The following procedure generates a random number between 3 and 9, uses it as the `key` to call `(string-encryption str key)`, and appends the `key` to the end of the encrypted string.

```
(define random-encryption (lambda (str)
   (let ((key (+ (random 7) 3)))
        (string-append (string-encryption str key)
                       (number->string key)
        )
   )
))
```

4.8.3 List manipulations

List is the most important data type of the functional programming languages. Since a list consists of a collection of elements, most list manipulations involve repetition or recursion.

The following program computes the sum of a list of numbers:

```
(define list-sum (lambda (lst)
   (if (null? lst)
        (display "Error: the list is empty")
        (list-sum-helper lst)
   )
))
(define list-sum-helper (lambda (lst)
```

```
  (if (null? lst)
      0
      (+ (car lst) (list-sum-helper (cdr lst)))
  )
))
(list-sum '(2 3 4 6))    ; call the procedure. It returns 15.
```

We split the program into two procedures. The first procedure checks if the initial list given is empty. If it is empty, the procedure returns an error message. Note that if we call the second procedure using an empty list, it will return 0. This return value does not differentiate whether the sum is 0 or there is no element in the list.

The second procedure is recursive. The stopping condition is "the list is empty." The corresponding return value is 0. The size-(n−1) problem is `(list-sum-helper (cdr lst))`, because `cdr` returns the list without the first element. Thus, the size of `(cdr lst)` is one smaller than the size of `lst`. The construction of the solution of the size-n problem is by

```
(+ (car lst) (list-sum-helper (cdr lst)))
```

which adds the first element to the sum of the remaining elements.

The next example we discuss is reversing a list. The size-n problem is `(reverse-list lst)`. The stopping condition is empty list and the corresponding return value is empty list. To reduce the size-n problem to the size-(n−1) problem, we take out the first element of the list using `(cdr lst)`. To construct the solution from the size-(n−1) problem to the size-n problem, we append the first element of `lst` to the end of the solution of the size-(n−1) problem. Thus, we obtain the following program.

```
(define reverse-list (lambda (lst)
  (if (null? lst)
      '()
      (append (reverse-list (cdr lst)) (list (car lst)))
  )
))
```

For example, the procedure call (reverse-list `'(1 3 5 7 9)`) will return: `(9 7 5 3 1)`.

We can define a few other list operations. The following procedure returns the union of two lists:

```
(define tail (lambda (lst)
         (car (reverse-list lst))))
(define rmtail (lambda (lst)
         (reverse-list
           (cdr (reverse-list lst)))))
(define union (lambda (x y)
  (cond ((null? x) y)
        ((member (tail x) y)
         (union (rmtail x) y))
        (else (union (rmtail x) (cons (tail x) y))))))) (union '(1 3 5) '(2
3 4 6))
>(1 5 2 3 4 6)
```

We use tail and rmtail procedures to keep the elements in the same order in the merged list. We could use `car` and `cdr`, instead of `tail` and `rmtail`, in the code. However, the output will be `(5 1 2 3 4 6)`, instead of `(1 5 2 3 4 6)`. The order of the first would be reversed in the result list.

The following procedure returns the intersection of two lists:

```
(define intersection (lambda (x y)   ; umbrella to deal with empty list
  (if (null? x) '()
      (intersect-recursive x y))
))
(define intersect-recursive (lambda (x y)
      (if (null? (cdr x))     ; stopping condition
          (if (member (car x) y)
x                ; return value at stopping condition
               '()           ; return value at stopping condition
               )
          (if (member (car x) y)
              (cons (car x) (intersect-recursive (cdr x) y))
              (intersect-recursive (cdr x) y)))
 ))
(intersection '(1 3 5 6 7) '(2 3 4 6))
> (3 6)
```

The procedure is designed following the fantastic-four abstract approach. The steps are:

Step 1: The size-n problem is (intersect-recursive `(x y)`)

Step 2: The stopping condition is when x has one element only, i.e., (`null? (cdr x)`). In this case, if `x` is a member of `y`, then, the intersection is `x`, else, is `'()`.

Step 3: The size-(n-1) problem is (intersect-recursive `(cdr x) y`), where x is replaced by `(cdr x)`, which has one element less than x.

Step 4: Construction of size-n problem's solution from size-(n-1) problem's solution. We assume that the size-(n-1) problem's solution is given. We just need to add `(car x)` into the solution if `(car x)` is a member of y. If `(car x)` is not a member of y, the size-n problem's solution is the same as the size-(n-1) problem's solution.

4.9 Higher-order functions

A **higher-order function** is a function that takes the operation of another function (not the return value) as an argument. All functional programming languages support higher-order functions. There are many different higher-order functions. We will discuss the two most useful higher-order functions defined on lists:

- **Mapping:** Apply the same operation defined by another procedure to all elements of a list;
- **Reduction:** Apply an operation to list and generate a single value as output.
- **Filtering:** Remove elements of a list that do not satisfy a predicate defined by another procedure.

4.9.1 Mapping

The general form of mapping is

```
(map procedure-name list-parameter)
```

where `list-parameter` is a list with the same type of elements and `procedure-name` is the name of any procedure that manipulates a single parameter that has the same type as the element of the `list-parameter`. The map procedure will call the procedure (procedure-name list-element) on each element of the list and return a list that consists of the return values of these procedure calls. For example, if we define a `foo` procedure on an integer and apply the procedure in a map procedure:

```
(define foo (lambda (x) (+ (* x x ) x)))
(map foo '(3 6 9 12 15 18))
```

The map procedure will apply `foo` on each and every element of the list `(3 6 9 12 15 18)`, and, thus, the map procedure will return

```
    (3*3+3  6*6+6  9*9+9  12*12+12  15*15+15  18*18+18)
=   (12 42 90 156 240 342)
```

We can also embed the body of the `foo` into the map procedure to implement the same function:

```
(map (lambda (x) (+ (* x x ) x))'(3 6 9 12 15 18))
```

Now we will use map procedure to reimplement some procedures we wrote before, where the same operation is applied to each element of a list.

First, examine the one's complement. The same operation, the inversion operation, is applied on each element of the list. Thus, we can first define a bit-inversion procedure, and then apply the procedure to the list.

```
(define bitinvert (lambda(x) (if (= x 0) 1 0)))
(define onescomplement (lambda (x)
  (map bitinvert x)
))
```

Next, consider the string encryption, where an integer is added to each character in the string. Since map procedures only work on lists, we need to convert the string to a list. The following is the program:

```
(define character-encryption (lambda (ch)
  (integer->char (+ (char->integer ch) 5))    ; key = 5
))
(define string-encryption (lambda (func str)
  (list->string (map func (string->list str)))
))
```

When we call the second function and use the first function, character-encryption, as a parameter:

> (string-encryption character-encryption "Hello World")

the function will be applied to each element of the list. The return value is: "Mjqqt%\\twqi."

Notice that the string is first converted to a list in order to use the higher-order map function. The returned list is converted back to a string.

As can be seen from these two examples, the previous implementations are recursive, involving repetition of operations. Using the map higher-order function, no recursion is needed. The reason is that recursion is embedded in the map function and is thus transparent to the user of the map higher-order function. If we look at the implementation of map, we will see the recursion.

```
(define map1 (lambda (procedure-name list-parameter)
       (if (null? list-parameter)
            '()
            (cons    (procedure-name (car list-parameter))
                     (map1 procedure-name (cdr list-parameter))
            )
)))
```

As can be seen from this program, `procedure-name` is an ordinary parameter of a `map1` procedure. However, in the body of the program, `procedure-name` is placed in the first place directly following a left parenthesis, where an operator is expected. Thus, the parameter `procedure-name` is considered an operator. This brings, in fact, a new type of parameter passing, known as **call-by-name**. Different from call-by-value or call-by-alias, call-by-name does not pass the value or address of the variable. It passes the name itself.

Now, we extend the string encryption procedure to more complex situations. We will read the key from the keyboard, and we will encrypt alphabetic characters and digits only. We do not encrypt all the other characters, such as space and special characters. We assume the key values will be in the range 1 to 4. We first give the normal recursive version as follows.

```
;Encryption procedure using recursion
(define encrypt (lambda (str)
    (recursive-encrypt str 0 (string-length str) (read))
))
(define recursive-encrypt
  (lambda (str pos len key)
    (if (>= pos len)
        ""
        (string-append
         (encrypt-char (string-ref str pos) key)
         (recursive-encrypt str (+ 1 pos) len key)))))
(define encrypt-char
  (lambda (c key) (encrypt-rotate c key)
))
(define encrypt-rotate
  (lambda (c key)
    (if     (or (char-alphabetic? c)
            (and (> (char->integer c) 47) (< (char->integer c) 58)))
        (string (integer->char (+ (char->integer c) key)))
        (string c))
))
```

In the procedure, we check ASCII code value to determine if a character is a digit. We can also define a predicate (digit? c) for this purpose.

Next, we give the decryption procedure that decrypts the string encrypted using the procedure above. The decryption procedure will have a different data range for decryption, which is the encryption range plus the key value, as the encryption procedure has added a key to the encrypted value.

```
;Decryption procedure using recursion
(define decrypt (lambda (str)
    (recursive-decrypt str 0 (string-length str) (read))
))
(define recursive-decrypt (lambda (str pos len key)
    (if (>= pos len)
        ""
        (string-append
         (decrypt-char (string-ref str pos) key)
         (recursive-decrypt str (+ 1 pos) len key)))
))
(define decrypt-char (lambda (c key)
    (decrypt-rotate c key)
))
(define decrypt-rotate (lambda (c key)
    (if (or (char-alphabetic? (integer->char (- (char->integer c) key)))
        (and (> (char->integer c) (+ 47 key))
             (< (char->integer c) (+ 58 key))))
        (string (integer->char (- (char->integer c) key)))
        (string c))
))
; Testing:
(encrypt "Hello CSE240!") ; Enter key = 4
(decrypt "Lipps GWI684!")
(encrypt "Hello CSE598?")
(decrypt "Lipps GWI9=<?")
```

Now, we present the higher-order function version of the same procedures, where the recursion part is embedded into the map procedure. First we give the encryption part.

```
;Encryption procedure using map
(define key (read))
(define encrypt (lambda (str)
      (list->string (map encrypt-char (string->list str)))
))
(define encrypt-char (lambda (c)
    (if (or (char-alphabetic? c)
        (and (> (char->integer c) 47) (< (char->integer c) 58)))
    (encrypt-rotate c)
    c)
))
(define encrypt-rotate (lambda (c)
      (integer->char (+ (char->integer c) key))
```

```
))
```

The higher-order function version of the decryption procedure is given as follows.

```
;Decryption procedure using map
(define decrypt (lambda (str)
    (list->string (map decrypt-char (string->list str)))
))
(define decrypt-char (lambda (c)
    (if (or (char-alphabetic? (integer->char (- (char->integer c) key)))
        (and (> (char->integer c) (+ 47 key))
             (< (char->integer c) (+ 58 key))))
    (decrypt-rotate c)
    c)
))
(define decrypt-rotate
  (lambda (c)
       (integer->char (- (char->integer c) key))
))
; Testing
; Enter key = 4
(encrypt "Hello CSE240!")
(decrypt "Lipps GWI684!")
(encrypt "Hello CSE598?")
(decrypt "Lipps GWI9=<?")
```

Mapping is an example of higher-order functions. You can design different higher-order functions based on the idea of applying an operation to a list of elements. For example, you can design a (deep-map proc a-list), where the list contains a sublist, such as '(1 3 (4 (2 2) 5) ((6 (7)))). You may also design a map2 function that can add a number list, such as:

```
(map2 + '(3 4 5) '(4 5 6) '(5 6 7)) ; return value: (12 15 18)
```

4.9.2 Reduction

Mapping allows us to apply the same operation to many data points simultaneously. Reduction allows us to combine multiple results from the parallel mapping operations into a single result. Without the reduction function, we would need to write a recursive program to perform such operations, for example,

```
(sum-list '(12 33 564 122 12 1 4))
(define sum-list (lambda (x)
  (if null? x)
      0
      (+ (car x) (sum-list (cdr))))))
```

The reduction function provides a generic way of handling all these functions by computing a function that depends upon all members of a list. Similar to mapping, the idea is to pass an operator (function) as a parameter to the reduction function. The definition of the reduction function (reduce) is given as follows:

```
(define reduce
  (lambda ( op base x) ;passing by name
```

```
    (if (null? x)
        base
        (op (car x) (reduce op base (cdr x)))))
)
```

In the definition above, we used the parameter base, because different functions need different bases (e.g., sum needs 0 and product needs 1 as bases). Having defined the higher-order reduction function, we can apply it to different operators. For example,

Sum: (reduce + 0 '(2 4 6 8 10)) ⇒ 30

Product: (reduce * 1 '(2 4 6 8 10)) ⇒ 3840

Average: (/ (reduce + 0 '(2 4 6 8 10)) (length '(2 4 6 8 10))) ⇒ 5

Mapping and reduction are often used together to solve large problems. For example, Google uses a MapReduce technique in its search engine to perform parallel searches. The web domains to be searched are divided into many subdomains. The same search operation is mapped to each subdomain. The search results from all subdomains are then reduced into a single list as the final search result.

4.9.3 Filtering

A filtering procedure or a filter is a higher-order function similar to a mapping procedure. It applies another procedure to all members of list:

```
(filter procedure-name list-parameter)
```

The difference is that the procedure that is the parameter of filter here is a **predicate** that returns either true or false. If the predicate (procedure-name list-element) returns true, the element will stay in the result list that is to be returned by the filter procedure. If the predicate (procedure-name list-element) returns false, the element will be removed from the result list.

For example,

```
(filter (lambda (x) (> x 200)) '(50 300 500 65 800))
```

will return

```
(300 500 800)
```

which is the sublist of the list in the filter procedure with all elements that are less than or equal to 200 removed. Similar to the map procedure, the predicate procedure in the filter procedure can be defined separately:

```
(define largerthan200? (lambda (x) (> x 200)))
(filter largerthan200? '(50 300 500 65 800))
```

The filter procedure can also be applied to the substructures with a list. For example, if we have a list of (class-name class-size) pairs defined as follows:

```
(define class-list
  '(("CSE100" . 100) ("CSE200" . 80) ("CSE240" . 100) ("CSE310" . 70)
("CSE330" . 75) ("CSE310" . 65) ("CSE420" . 50)))
(define large-class? (lambda (x) (>= (cdr x) 80)))
(define find-large-class (lambda (alist)
  (filter large-class? alist)
```

```
))
```

If we call the procedure that uses the filter procedure, the following sublist will be returned:

```
(("CSE100" . 100) ("CSE200" . 80) ("CSE240" . 100))
```

The filter procedure is not implemented in the current DrRacket version. We can define it as a user's procedure as follows:

```
(define filter (lambda (predicate-name alist)
  (if (null? alist)
      '()
      (if  (predicate-name (car alist))
           (cons (car alist) (filter predicate-name (cdr alist)))
           (filter predicate-name (cdr alist))
      )
)))
```

As we can see, the filter procedure is a recursive procedure. The stopping condition is "if the list is empty" and the return value at the condition is empty list. The size-(n−1) problem is the problem when the first element is removed. To construct the solution of size-n, we use the predicate-procedure to test the current element of the list. If the return value of the predicate procedure is true, we include the element in the return list by performing a `cons` operation on the element and the solution of the size-(n−1) problem. Otherwise, the solution of the size-n problem is simply the solution of the size-(n−1) problem: The current element is simply not included in the construction of the size-n problem.

The map and filter procedures defined above work for plain lists. They do not work for lists with sublists. However, we can modify the definition of map and filter, so that they can dive into sublists and apply the operator to all elements of the sublists. These higher-order functions are called deep-filter and deep-map.

```
(define deep-filter (lambda (pred arg-list)
  (if (null? arg-list)
      '()
      (if (pair? arg-list)
           (if  (not (pair? (car arg-list)))
                (if  (pred (car arg-list))
                     (cons (car arg-list) (deep-filter pred (cdr arg-
list)))
                     (deep-filter pred (cdr arg-list)))
                       (cons    (deep-filter pred (car arg-list))
                       (deep-filter pred (cdr arg-list))))))))
))
```

The key of this procedure is to check if the arg-list is a pair. If it is, we will further check if the first element is a pair. If not, we will check if its value satisfies the predicate. If not, it will not be included into the result list. If the element is a pair, then, we will recursively apply the deep-filter procedure to its first element and the second element of the pair. Notice that all lists, except the empty list, are pairs.

For example, if we test the following call:

```
(deep-filter     (lambda (x) (> x 100))
                 '(200 100 ( 220 50 120) 200 19 300 100 (90 2 900) (20)))
```

The output will be:

```
(200 (220 120) 200 300 (900) ())
```

where the elements of less than or equal to 100 are removed from the result list. Similarly, a deep-map can be defined.

4.9.4 Application of filtering in query languages

Filter is widely used in many situations to remove unwanted data. This is particularly important in data mining, database, and web applications. The C# code example below illustrates a typical query to a database source:

```
class program {
      static void Main() {
          Book[ ] Books = new Book[ ] {
              new Book {bookid = 1, title = "Programming",
                   isbn = "0-7575-0367", price = 69.99},
               new Book {bookid = 3, title = "OS",
                   isbn = "6-5432-123-0", price = 57.77},
  new Book {bookid = 4, title = "Computing",
                   isbn = "0-321-52403-9", price = 94.91},
  new Book {bookid = 5, title = "XML",
                   isbn = "0-201-77168-3", price = 74.21},
          };
          var myQuery =
              from b in Books
              where b.price < 80
              orderby b.title
              select b;
          foreach (Book item in myQuery)
              Console.WriteLine("Title = {0}, Price = {1}",
              item.title, item.price);
      }
  }
}
```

In this example, the query part, "from-where-orderedby-select" is in functional style with filtering. The clause "`where b.price < 80`" defines the filtering predicate and the predicate is applied to each element of the books. Other functional programming features are also used. In the following snippet of code, a simple query is given that returns all the customers that have placed orders between 100 and 1000.

```
static void CalcNoLet() {     // Without using let
  var q =   from c in AllCustomers
            where SumOrders(c) < 1000 && SumOrders(c) > 100
            select c;
            int count = q.Count();
}
```

In the code, the function SumOrders are called twice and executed twice. If we use a let-form to associate the result with a local name expense, there is no need to perform the calculation twice.

```
static void CalcWithLet() {  // Using let
  var q = from c in AllCustomers
          let expense = SumOrders(c)
          where expense < 1000 && expense > 100
      select c;
      int count = q.Count();
}
```

4.10 Summary

In this chapter, we studied the major features of functional programming languages and important techniques of writing functional programs, including

- prefix notation;
- data types and predefined Scheme functions on the data types;
- the syntax and semantics of λ-calculus, and the relationship between λ-calculus and the Scheme programming language;
- important programming constructs of functional programming languages: named procedure, unnamed procedure, let-form global and local variables, and the conversion between unnamed procedures and let-forms;
- writing Scheme programs with multiple procedures;
- the principle of recursion, and the techniques of writing recursive procedures;
- the higher-order functions that can be used to solve recursive problems in a much simpler way.

4.11 Homework, programming exercises, and projects

1. Multiple Choice. Choose only one answer for each question. Choose the best answer if more than one answer is acceptable.

1.1 In Scheme, the primitive `(char? "#\A")` will return

❑ `#t` ❑ `#f` ❑ `A` ❑ error message

1.2 (member '2 '(3 4 2 1)) will return

❑ #f ❑(3 5 2 9) ❑(4 2 1) ❑(2 1)

1.3 (caddr '(2 4 6 8 10)) will return

❑ `(6 8 10)` ❑ `(6)` ❑ `6` ❑ error message

1.4 The most efficient way, in terms of the execution time, to check whether a list `L` is empty is by

❑(NULL? L) ❑(= (length L) 0)

❑(< (length L) 1) ❑(= L 0)

1.5 Which of the following forms is an unnamed procedure?

❑(+ z 3) ❑((lambda (z) (+ z 3)) 4)

❑(define foo (lambda (z) (+ z 3))) ❑(define bar 25)

❑ none of them

1.6 Eager evaluation evaluates

❑ all parameters of a function first.

❑ a parameter of a function only if it is necessary.

❑ no parameters at all.

❑ outermost first.

1.7 Lazy evaluation evaluates

❑ all parameters of a function first.

❑ a parameter of a function only if it is necessary.

❑ no parameters at all.

❑ innermost first.

1.8 In imperative programming languages, different orders of evaluations (eager or lazy)

❑ may produce different results. ❑ always produce different results.

❑ never produce different results. ❑ None of them are correct.

1.9 In functional programming languages, different orders of evaluations (eager or lazy)

❑ may produce different results. ❑ always produce different results.

❑ never produce different results. ❑ None of them are correct.

1.10 Each let-form in Scheme can be converted into

❑ an unnamed procedure. ❑ a named procedure.

❑ a list of local variables. ❑ a list of global variables.

1.11 Assume that you have `(define x '(5))` and `(define y '(8 9))`. What operation will return the list `(5 8 9)`?

❑(cons x y) ❑(list x y) ❑(append x y) ❑ None of them

1.12 Which of the followings is NOT a Scheme pair?

❑ `'()` ❑ `'(x . y)` ❑ `'(x)` ❑ `'(())`

1.13 What is the return value of the following form?

```
(filter (lambda (x) (> x 20)) '(10 30 15 10 80))
```

❑(30 80) ❑(10 15 10) ❑(10 15) ❑(10 10 15)

1.14 A deep-filter can be used in the situation where the list

❑ is a plain list. ❑ contains nonnumerical values.

❑ contains sublists. ❑ is not a pair.

1.15 What is the return value of the following form?

```
(map (lambda (x) (+ x 10)) '(10 30 15))
```

❑ 20 ❑ 40 ❑ 25 ❑(20 40 25)

1.16 In Scheme, an empty list is

❑ a pair. ❑ not a pair. ❑ a string. ❑ 0

1.17 What mechanism cannot be used for passing a value into a Scheme procedure?

❑ Call-by-value ❑ Call-by-alias ❑ Call-by-name ❑ Return value

1.18 How is a procedure name (operator) passed into a procedure?

❑ Call-by-value ❑ Call-by-alias ❑ Call-by-name ❑ Return value

1.19 If you want to return multiple values from a Scheme procedure, which of these methods is invalid?

❑ Use multiple return-statements. ❑ Split the procedure into multiple procedures

❑ Put the values in a pair and return the pair ❑ Put the values in a list and return the list

1.20 Normally, a recursive procedure can be written by following these steps: Define the size-n problem, find the solution for the base case or the stopping condition, and then, find

❑ the solutions of the size-1, size-2, ..., size-n problems.

❑ a loop variable that is incremented in each iteration.

❑ the solutions of the size-n, size-(n−1), size-(n−2), ..., size−1 problems.

❑ the solution of the size-(n−1) problem, and finally find the solution of the size-n problem.

- ❑ the solution of the size-n problem based on the hypothetical solution of the size-(n−1) problem.

2. What is the major difference between the imperative and functional programming paradigms? How does an imperative program typically pass values from one function to another function? How does a functional program pass values from one function to another function?

3. How does a Scheme program pass parameters into a procedure? Does Scheme support call-by-value? Does Scheme support call-by-alias?

4. What is the difference between a Scheme procedure and a Scheme macro? Write a macro that returns the absolute value of a number.

5. What is an unnamed procedure? Why do we need an unnamed procedure? How do we define and call an unnamed procedure?

6. What are bound and free variables in λ-calculus?

7. What are global and local variables in Scheme?

8. What is eager evaluation? What is lazy evaluation? Is the order of evaluation (eager or lazy) important in functional programming language like Scheme? Is the order of evaluation important in imperative programming languages like C/C++? Assume we have a multiprocessor computer that can evaluate 10 independent operations simultaneously and each arithmetic operation takes a unit of time. How many units of time are necessary to evaluate the following form?

```
(+ (+ (- 6 2) (* 5 7)) (* (+ 4 6) (- 5 3)))
```

8.1 if the form is evaluated in an imperative language like C?

8.2 if the form is evaluated in Scheme?

9. According to the BNF definition of λ-expression, if `E1` and `E2` are λ-expressions, `E1 E2` is also a λ-expression. If `E1`, `E2`, and `E3` are λ-expressions, is `E1 E2 E3` also a λ-expression? When do we need a λ-expression of the form `E1 E2 E3`?

10. What is a λ-procedure? What reduction rule evaluates a λ-procedure?

11. What is the relationship between a λ-expression and a Scheme form? How do we convert a λ-expression into a Scheme form?

12. How do we convert a let-form into an unnamed procedure? How do we convert an unnamed procedure into a let-form?

13. Given a λ-expression: λx{+ λx[- x 1] 3 (* λx[+ x 2] 3 x) } 9

13.1 Indicate the scope of each variable by underlining the variable and the expression associated with it.

13.2 Use the α-conversion rule to convert the expression, so that different parameters have different names.

13.3 Use the β- and η-conversion rules to convert (using lazy evaluation) the expression. Show each step of the conversion.

13.4 Give the Scheme unnamed procedure corresponding to the λ-expression.

14. How do we introduce a global variable/procedure? How do we define a local variable/procedure?

15. What kinds of data structures does Scheme support?

16. What is a character type in Scheme? Is a character treated as a string with only one element?

17. What is a pair? How do we represent a pair?

18. What is a list? How do we represent a list? Is a pair a list? Is a list a pair?

19. When do we need a quote and when not?

20. Convert the following expressions into prefix notations and use DrRacket to evaluate them.

20.1 (2 + (4 + (6 + (8 + (10 + 12)))))

20.2 (((((2 + 4) + 6) + 8) + 10) + 12)

20.3 ((2 + 4) + (6 + 8) + (10 + 12))

20.4 (2 + 4 + 6 + 8 + 10 + 12)

20.5 (2 + 4 * 6 + 8 * 10 + 12)

20.6 125^{187}

20.7 Input two integers and add them: `(read) + (read)`

20.8 Print ((2 + 4) + (6 + 8) + (10 + 12))

21. Write Scheme programs/forms to perform the following functions.

21.1 Find the second element of the list, e.g., `'(2 4 6 8 10 12)`. Your form should work for any list containing two or more elements.

21.2 Find the last element of the list `'(2 4 6 8 10 12)`. Your form only needs to work for lists of six elements.

21.3 What would be the return value of the form `(caddddr '(3 1 8 9 2))`? 2 or `'(2)`?

21.4 Merge the two lists `'(1 2 3 4)` and `'(5 7 9)` into a single list `'(1 2 3 4 5 7 9)`.

21.5 Obtain the length of the list `'(a b x y 10 12)`.

21.6 Check whether `'(+ 2 4)` is a symbol.

21.7 Check whether `'+` is a member of the list `'(+ 3 4 6)`.

21.8 Check whether `"+"`, `'(+ 3 5)`, and `"(* 4 6)"` are strings.

21.9 Check whether `(* 3 5)`, `'(/ 3 7)`, `(1 2 3 4)`, `"(+ 2 8)"` and `"( 1 2 3)"` are strings.

22. Show how the form `(/ (+ 5 4) (- 8 (* 2 3)))` is executed on a stack machine, a computer architecture that is based on a stack instead of registers.

23. Given the following Scheme program/procedure:

```
(define myabs
    (lambda (x)
        (if  (negative? x)
             (-  x)
             x
        )
    )
)
```

23.1 What does this program do?

23.2 Find 3 test cases to test the program in DrRacket environment.

24. Given the following Scheme program:

```
(define foo
    (lambda (n)
        (if  (= n 0)
             1
             (* n (foo (- n 1)))
        )
    )
)
```

24.1 What does this program do?

24.2 Test the program with `n = 0`, `n = 5`, `n = 150`, `n = -5`.

24.3 The program does not terminate for some inputs. Find and fix the bug and re-execute the program.

25. Define a Scheme procedure with two parameters.

25.1 Define a procedure `(mymax x y)` that returns the larger value between x and y.

25.2 Test your program with inputs (0 0), (-2 0), (0 -2), (10 0), (0 12), (1000 10), (20 8000).

25.3 Find the largest value among `(48, 6, 120, 35, 12)` by repeatedly calling the procedure `(mymax x y)`.

26. Given the following Scheme procedure:

```
(define dtod (lambda (N)
    (if   (= N 0)
          (list 0)
          (append  (dtod (quotient N 10))
                   (list (remainder N 10)))
    )
))
```

26.1 What does this program do?

26.2 Modify the program to remove the leading zero in the output list.

26.3 Find 3 test cases to test the program under DrRacket.

27. Write a Scheme program `(dtoh N)` that converts a decimal number into the list of its hexadecimal digits, where you must use letter `'a` for `10`, `'b` for `11`, ..., and `'f` for `15`, for example,

```
(dtoh 18)  → (1 2)
(dtoh 26) → (1 a)
(dtoh 225) → (e 1)
```

28. What do the following Scheme procedures do?

```
(define dtoh (lambda (N)
    (if (= N 0)
        '(0)
        (dtoh0 N)
    )
))
(define dtoh0 (lambda (N)
    (if  (= N 0)
        '()
        (append  (dtoh0 (quotient N 16))
                 (list (remainder N 16)))
    )
))
```

29. Write a Scheme program to convert a decimal number into a decimal in list format.

30. Given a λ-expression:

λ(x, y){+ λx[- x 2] 4 (* λx[+ x 2] 3 x) (/ λx[/ x 2] 4) y} 8 λx(+ x 2) 4

30.1 Indicate the scope of each variable by underlining the variable and the expression associated with it.

30.2 Use the α-conversion rule to convert the expression, so that different parameters have different names.

30.3 Use the β- and η-conversion rules to convert (using lazy evaluation) the expression. Show each step of the conversion.

30.4 Give the Scheme program corresponding to the λ-expression. The program should consist of unnamed procedures only.

30.5 Rewrite the Scheme program using let-forms instead of unnamed Scheme procedures.

31. Write a recursive program to implement (tail x): return the last element of list `x`. If `x` is empty, the program should return "error." For example, `(tail '( ))` ⇒ "error," `(tail '(a 3 b))` ⇒ `b` and `(tail '(1 2 w 5 7))` ⇒ 7. You may NOT call `(reverse x)` procedure in your program. Use the following procedure calls as test cases:

```
(tail '())
(tail '(2 3 4 ab 4 5 cd))
(tail '(2 (3 5)))
```

`(tail '(7))`

Hint: To handle the empty list, you can write a separate procedure to check if x is null. If it is, you return "error"; otherwise, you call the procedure that handles a nonempty list.

32. Write a recursive program to implement (rmtail x): remove the last element of list x and return the resulting list. If the initial x is empty, the program must return "error." For example, `(rmtail '())` ⇒ "error," `(rmtail '(a 3 b))` ⇒ `'(a 3)` and `(tail '(1 2 w 5 7))` ⇒ `'(1 2 w 5)`. You must follow the fantastic-four abstract approach to write a recursive program to do the job. You may NOT call `(reverse x)` procedure in your program. Use the following test cases to test your program:

```
(rmtail '())
(rmtail '(2 3 4 ab 4 5 cd))
(rmtail '(2 (3 5)))
(rmtail '(7))
```

*33. Repeat the question above, but use C or C++.

34. Given the following Scheme program:

```
(define mymax (lambda (lst)
    (if  (null? (cdr lst))
         (car lst)
         (let ((m (mymax (cdr lst))))
              (if  (> (car lst) m)
                   (car lst)
                   m)))
))
```

34.1 Add a procedure to handle the case of an empty list, that is, if the given list is empty, the program should return "error: list empty."

34.2 Consider that the fantastic-four abstract approach is used to solve the problem. Describe each step and the solution obtained in each step.

34.3 Compare and contrast the algorithm used in this program and the divide-and-conquer algorithm used in the next question.

34.4 Use C to reimplement the program. Compare and contrast the C and Scheme programs.

35. Use the divide-and-conquer algorithm to implement a procedure `(maxdac lst)` that finds the largest number in the given list `lst`. A divide-and-conquer algorithm divides a size-n problem into two half-sized problems, solves each of them recursively, and combines the solutions of the two half-sized problems.

35.1 Define two procedures to find the first and the second halves of a given list: `(firsthalf lst)` and `(secondhalf lst)`, where their lengths are $\lceil n/2 \rceil$ (ceiling) and $\lfloor n/2 \rfloor$ (floor), respectively.

35.2 Write an umbrella procedure to handle the empty list, that is, if the list is initially empty, the procedure should return "error: list empty."

35.3 Use the divide-and-conquer algorithm and follow the four design steps to devise the solution of the size-`n` problem `(maxdac lst)`. In each step, the list is divided into two sublists of length $\lceil n/2 \rceil$ and

$\lfloor n/2 \rfloor$, respectively. You cannot call any library function that can find the max-value. You can only use <, >, <=, or >= operators to perform comparison operations in your program.

Hint: You can use a let-form to assign the largest number from the first half to `m1` and the largest number from the second half to `m2`, and then choose the larger one between `m1` and `m2`.

35.4 Test the program using the following test cases:

```
(define lst '(5))
(firsthalf lst)
(secondhalf lst)
(maxdac lst)
(define lst '(5 2))
(firsthalf lst)
(secondhalf lst)
(maxdac lst)
(define lst '(2 8 6 5 28 2 9))
(firsthalf lst)
(secondhalf lst)
(maxdac lst)
```

36. A computer system consists of hardware and software. Normally, before we physically make a piece of hardware, we simulate the hardware by a program, so that we can verify its correctness and evaluate its performance. As we know, all complex hardware components can be implemented by the basic gates AND, OR, NOT, and XOR shown in Figure 4.9.

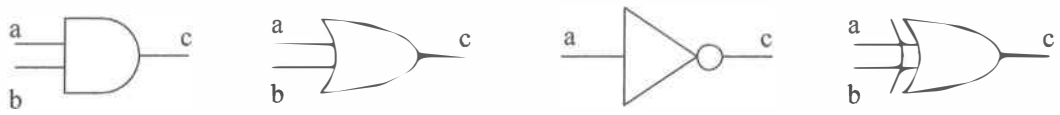


Figure 4.9. Basic gates.

36.1 Write four Scheme procedures to simulate these four gates.

36.2 Define a Scheme procedure `(fulladder x a b)` to simulate the logic in Figure 4.10. The procedure must return a list with two elements `'(s c)`, where `s` is the sum of `a, b, and x`, and `c` is the carry-out. Hint: You can use two procedures to produce the two results, respectively, and then write a main procedure to call the two sub procedures.

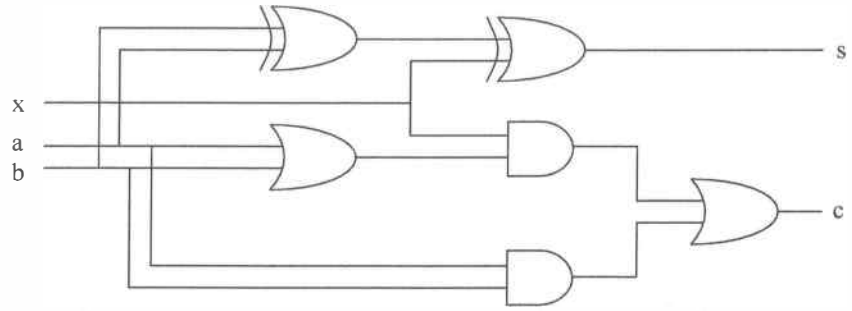


Figure 4.10. The logic of a full adder.

36.3 Verify your procedure by exhaustive testing. Use all valid inputs to test the procedure. There are eight valid inputs:

```
(fulladder 0 0 0)
(fulladder 0 0 1)
(fulladder 0 1 0)
(fulladder 0 1 1)
(fulladder 1 0 0)
(fulladder 1 0 1)
(fulladder 1 1 0)
(fulladder 1 1 1)
```

36.4 Figure 4.11 shows the design of an n-bit (n=32 in the figure) adder using n one-bit adders. The carry-out of bit-i is the carry-in of bit i+1, where carry-in of bit 0 is 0. Write a recursive procedure to implement the n-bit adder, and design a test plan to test the program.

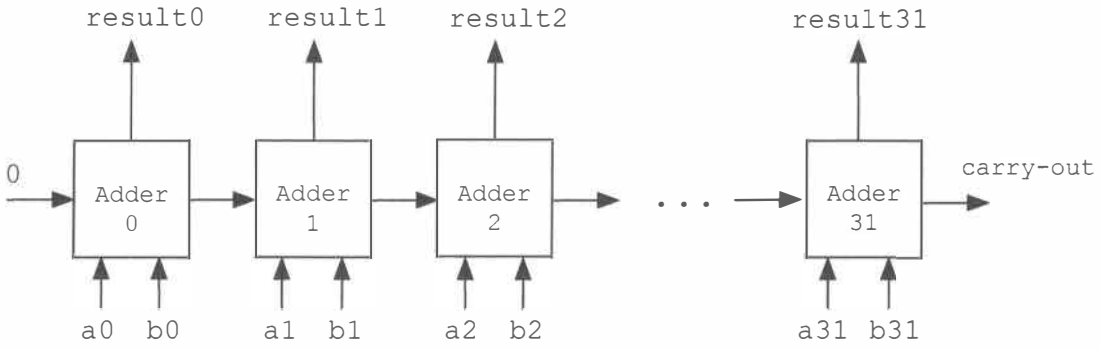


Figure 4.11. Design of a n-bit adder, where n=32.

Chapter 5

The Logic Programming Language, Prolog

Before we move to the logic programming paradigm, we can complete our pizza example discussed in Chapter 4 by including logic programming in this analogy. Imperative programming can be compared with making a pizza from scratch using flour, water, salt, tomato, cheese, and other topping stuff. You write the steps (recipe) and follow the steps to make the pizza. By the end of the steps, you have your pizza. Object-oriented programming uses larger components to simplify the recipe. Using the pizza analogy, it uses premade components, such as a premade pizza base and tomato sauce. Functional programming is comparable with ordering a pizza. Instead of following a recipe to make a pizza, you describe what you want your pizza to have. For example, you like to have a thin base with blue cheese, pepperoni, and bacon. Logic programming requires even less knowledge of the pizza. It allows you to describe a list of requirements, and then the system searches for solutions that satisfy the requirements. In the pizza analogy, you do not even need to know what pizza is. You can describe what you like to eat. For example, you can say that you like pasta, tomato, and mushroom. Then, the waiter or waitress can search the menu and find all the solutions that meet your requirements. Depending on how accurate and restrictive your requirements are, you may obtain many solutions, one solution, or no solution.

In this chapter, we will use Prolog as an example to study the main features and programming techniques of logic programming languages. By the end of the chapter, you should

- have a good understanding of the logic programming paradigm and its major differences with the imperative and functional programming paradigms;
- have a good understanding of variables in Prolog and their differences with variables in imperative and functional programming languages;
- be able to define a database that consists of multiple facts and rules;
- be able to write complex recursive rules;
- be able to write goals (questions) that inquire a database;
- understand the execution model of Prolog facts, rules, and goals;
- be able to use a Prolog programming environment, such as the GNU environment, to edit, debug, and execute Prolog programs.

5.1 Basic concepts of logic programming in Prolog

Logic programming describes what the problem is by a set of conditions and constraints, and leaves the computer to match the problem to the existing knowledge of facts and rules and to find solutions to the problem.

5.1.1 Prolog basics

Prolog uses a simplified version of predicate logic, which was developed to convey logic-based ideas in a written form. It is similar to natural language and thus easy to understand. To convert a natural language sentence into a predicate logic statement, you first eliminate all unnecessary words from your sentences, and then transform the sentence into the prefix notation by placing the relationship first and listing the objects in a pair of parentheses. For example,

- A computer is powerful. → powerful(computer).
- Charlie owns a computer. → own(charlie, computer).
- If a cell phone can perform computing, it must have a processor.
  → if computing(cellphone), contains(cellphone, processor).

A Prolog program consists of a list of facts, rules, and goals. The facts define the known relationships among objects, also called axioms. The rules define functions that generate new relationships based on existing relationships. The collection of all the facts is called the factbase. The collection of all the rules is called the rulebase. The collection of all the facts and rules is called the database. The goals are formulations of problems to be solved by searching and matching the database. The example below shows a Prolog program consisting of three types of statements:

```
% Facts about objects and their relationships
  man(conrad).
  man(obed).
  woman(elaine).
  mother_of(jane, elaine).
  mother_of(jane, mike).
  father_of(mike, andrew).
  father_of(andrew, conrad).
% Rules that extend facts: about objects and their relationships
  parent_of(X, Y) :- mother(X, Y); father(X, Y).          % where ";" = "or"
  grandmother_of(X, Z) :- mother_of(X, Y), parent_of(Y, Z). % "," = "and"
  grandfather_of(X, Z) :- father_of(X, Y), parent_of(Y, Z).
```

Once we have written the facts and rules, we can write goals (questions) about objects and their relationships. Below is a list of goals that query the database:

```
  ?- grandfather_of(mike, conrad).   % Is mike grandfather of conrad
  ?- mother_of(X, mike).             % who is mother of mike?
  ?- father_of(andrew, Y).           % andrew is the father of whom?
```

The syntax of a part of the facts and rules can be defined in BNF notation as follows. BNF notation was discussed in Chapter 1.

```
<digit>     ::= 0|1|2|3|4|5|6|7|8|9
<integer>   ::= <digit>|<number><digit>
<float>     ::= <integer>.<integer>
<number>    ::= <integer>|<float>
<lowercase> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<uppercase> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<underscore>    ::= _
<control>   ::= !|repeat|fail
```

```
<char> ::=  <lowercase>|<uppercase>|<digit>
<identifier>     ::= <lowercase>|<identifier><char>
<literal>   ::= <number>|<identifier>
<literals>  ::= <literal>|<literal>, <literal>
<object>    ::= <literal>|<identifier>|<variable>
<objects>   ::= <object>|<object>, <object>
<variable>  ::= <uppercase>|<underscore>|<variable><char>
<predicate> ::= <identifier>
<assignment>     ::= <variable>is<expression>
<clause>    ::= <control>|<assignment>|<predicate>(<objects>)
<clauses>   ::= <clause>|<clause>, <clause>|<clause>; <clause>
<fact> ::= <predicate>(<literals>).
<rule> ::= <clause>.|<clause>:-. | <clause> :- <clauses>.
<statement> ::= <fact>|<rule>
<statements>     ::= <statement>|<statement><statement>
<goal> ::= <clauses>.
```

A number of terminologies and concepts are exposed through the example and the simplified definition above. A Prolog database consists of a list of statements. A statement is a fact or a rule, which consists of clauses. The clauses are in prefix notation, starting with the predicate, also called relationship, followed by a list of objects or literals in a pair of parentheses, which are called arguments of the clauses. The set of facts or rules with the same predicate and the same **arity** (the number of arguments) can be represented in a notation of predicate/arity. For example, man/1, woman/1, mother_of/2, parent_of/2, and grandfather_of/2.

A name starting with an uppercase letter or the underscore is a **variable**. Numbers and identifiers are literals. Notice that identifiers start with a lowercase character. It cannot start with a number or other characters. It cannot contain special characters. For example, 31st_december and mac&cheese are not valid identifiers or literals. The arguments of a fact are all literals, while the arguments of a rule are normally variables, but can have some literals.

Unlike the variables and names in imperative and functional programming languages, a Prolog variable is a placeholder, not a memory location or a named value. A variable that begins with an uppercase letter is a named variable, while a variable that begins with the underscore character is an **anonymous variable** or unnamed variable. Both input value and return value can be passed to a named variable when calling a function (a **clause**). The anonymous variables are pure placeholders. Neither an input value nor a return value can be passed to an anonymous variable. We use an anonymous variable in a rule or in a goal if we do not need or do not care about the value of the variable.

A **rule** states a general relationship and normally uses variables as arguments. Rules are used to conclude a specific fact or another rule based on given conditions. A rule consists of three parts. The first part is called the conclusion. The second part is a two-character symbol ":-", with the meaning of "if." The third part is called the conditions. A rule states that **if** the **conditions** are true, the **conclusion** is true.

A **goal** is a question that queries the database. The syntax of a goal is the same as a clause or clauses that are executed in the query mode. To simplify the query, a complex goal with multiple clauses can be defined as a rule in the database. When executing the query, the conclusion part of the rule can be entered as the goal.

Next, we will study the structures and programming models of Prolog facts, rules, and goals in detail.

5.1.2 Structures of Prolog facts, rules, and goals

The Prolog examples discussed so far, as well as BNF notations, show simple structures of Prolog clauses. Prolog clauses also allow each object to have the structure of a clause. Let us consider the following book database with a schema of (title, author(first, last), price, date(year, month)).

```
% Facts
  book(c, author(ray, miller), 69.99, data(2009, 6)).
  book(scheme, author(john, smith), 49.99, date(2003, 5)).
  book(prolog, author(mary, lee), 59.99, date(1993, 7)).
  book(compiler, author(elaine, sanders), 109.99, date(2012, 5)).
  book(os, author(elaine, sanders), 129.99, date(1993, 4)).
% Rules
  authorship(Author, Title) :- book(Title, Author, _, _).
  cost(Title, Price) :- book(Title, _, Price, _).
  more_expensive(Title1, Title2) :- book(Title1, _, P1, _),
      book(Title2,_, P2, _), P1 > P2.
  newer(Title1, Title2) :- book(Title1,_, _, date(Y1, M1)),
      book(Title2,_, _, date(Y2, M2)), (Y1 < Y2; (Y1=:=Y2, M1<M2)).
/* Goals to ask
  book(c, Who, _, Published_on).
  authorship(Who, What).
  cost(prolog, Howmuch).
  more_expensive(MoreExpensiveTitle, compiler).
  newer(prolog, NewerBook).
  newer(scheme, NewerBook).
*/
```

In this example, the object author itself is a clause, with the predicate = author and with two objects first and last. The object date is also a clause with year and month as its object.

The last two rules combine the queries with arithmetic operations, where the prices and publication dates are compared. In the rule newer/2, the years are compared first. If the years are equal, then the months are compared.

To query about data using the facts, we need to provide four arguments. For the arguments we do not care about, we can use an anonymous variable (underscore). We can incorporate the "don't care" arguments into a few rules, so that the queries only need to provide the variables needed. Based on this idea, we defined a number of rules with two objects. Below is the query results of executing the goals listed as comments in the example above:

```
?- book(c, Who, _, Published_on).
Published_on = data(2009, 6)
Who = author(ray, miller)
```

This question asks "Who" is the author of the title "c" and what is the date of publication? We do not care about the price and thus an anonymous variable (underscore) is used in its place.

In the next question, the authorship rule is asked with both arguments as variables. In this case, one answer will be printed. If a semicolon is entered, it will print the next possible answer. If a return key is entered, the system will stop searching. In the outputs below, semicolons are entered until all the answers are printed.

```
?- authorship(Who, What).
   →   What = c
       Who = author(ray, miller) ? ;
   →   What = scheme
       Who = author(john, smith) ? ;
   →   What = prolog
       Who = author(mary, lee) ? ;
   →   What = compiler
       Who = author(elaine, sanders) ? ;
   →   What = os
       Who = author(elaine, sanders)
```

In the next question, the cost of the prolog book is asked. It simply returns the price

```
?- cost(prolog, Howmuch).
   →   Howmuch = 59.99
```

In the next question, the more_expensive rule is asked: Which book is more expensive than the compiler book? The query generates one result:

```
?- more_expensive(MoreExpensiveTitle, compiler).
   →   MoreExpensiveTitle = os
```

In the last query, the goal newer/2 is asked. The first question asks which prolog book is newer than which books, and the second question asks if Scheme is newer than which books by publication date. They generate one result and two results, respectively.

```
?- newer(prolog, NewBook).
   →   NewerBook = os
?- newer(scheme, NewBook).
   →   NewerBook = prolog ? ;
   →   NewerBook = os ? ;
   →   no
```

In the last rules of the book example, arithmetic operations are used. In the next section, we will discuss more examples that use arithmetic operations.

5.2 The Prolog execution model

This section explains how a goal is executed by the Prolog runtime. It also explains the unification between two clauses and between variables and literal values.

5.2.1 Unification of a goal

Prolog programs solve problems by asking questions and retrieving information from a database. A question is called a goal. A goal succeeds if there are facts that match or unify the goal. If no facts **unify** a goal, the

search checks the rules. A goal unifies a rule if it unifies all the clauses on the right-hand side of the :- symbol. A goal unifies (matches) a fact or a clause, if:

- Their predicates are the same.
- Their numbers of arguments are the same.
- Their corresponding arguments unify.

Two arguments unify, if they are identical literals, if one of them is a variable, or if both of them are variables. When a literal unifies a named variable, the variable is instantiated with the literal value. When a literal unifies an anonymous variable, the unification succeeds, but no instantiation takes place. If both arguments are variables, they unify into one variable. When one of the them is instantiated with a value, the other is also instantiated with the same value.

Clauses can be connected by "and" operators or "or" operators to form a composite clause. A clause always returns a true value or a false value. It can never return a nonlogical value such as an integer or a string. We must use an argument to store such a value being used outside the clause.

When a match is found for a goal, Prolog runtime prints "yes" if the question is a yes-no question, or prints the variable names and their values if the question uses name variables as the argument. If no more match is found after the entire database is searched, it returns no.

Assume a Prolog database consists of a set of facts and rules, as shown in the right-hand side in Figure 5.1. Assume a goal "?- qst(a_1, …, a_n)." is being executed to query the database. The execution process is illustrated in the C-like pseudo-code program on the left-hand side of Figure 5.1.

First, the predicate of the goal is compared with the predicate of each fact. If a match is found, the parameter list in the goal is further compared with the fact that has the same name. If a match is found, an answer (a solution) is found for the goal. In this case, the user can decide to stop searching or continue to find more solutions by typing the "enter" key or typing the semicolon.

Having compared all the facts sequentially, the search continues into the rules. The name and parameter list of the goal are compared with the name and parameter list of each rule. If a match is found, it does not mean that a solution is found, because the rule is conditional. The conditions of the rule must be further compared one by one based on how the conditions are composed. If they are combined by logic AND operation, a solution is found only if all conditions are true. If they are combined by logic OR operation, a solution is found once a condition is found true.

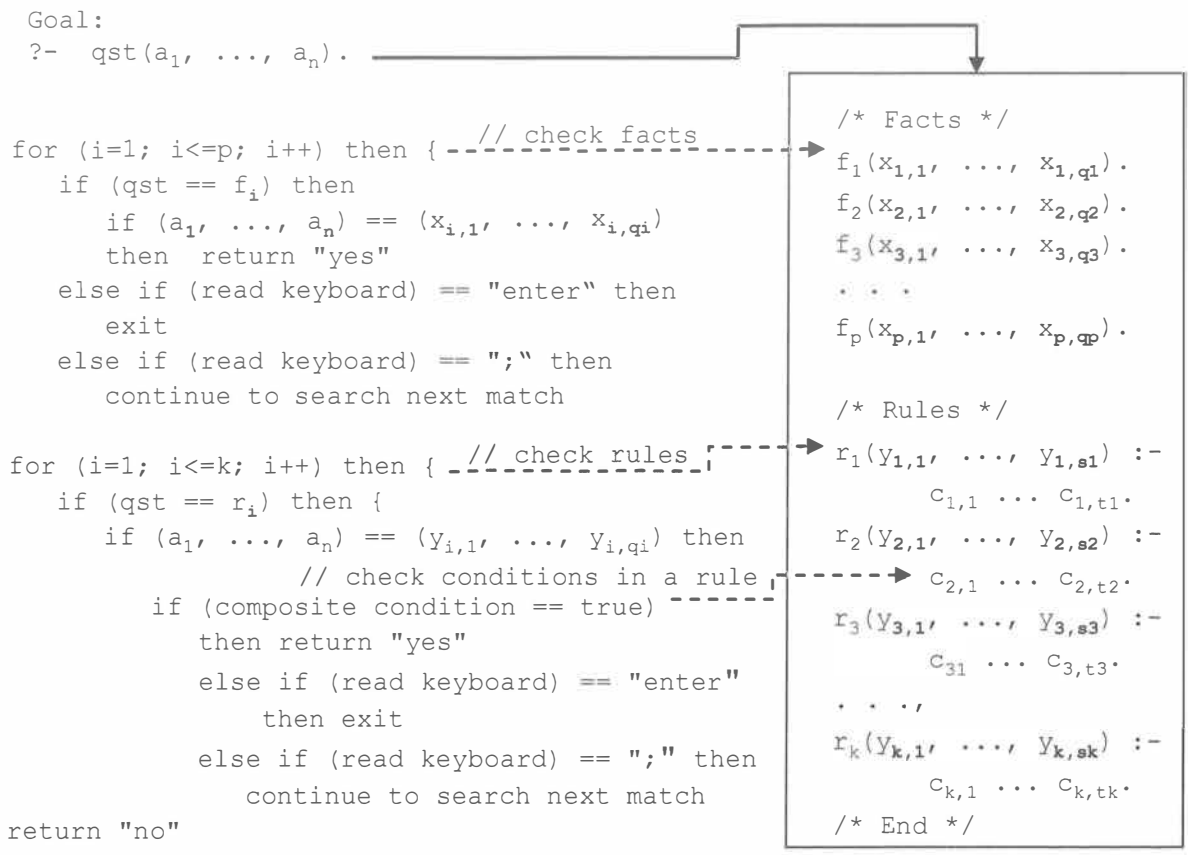


Figure 5.1. Prolog execution model.

5.2.2 Example of searching through a database

Figure 5.2 presents a concrete example demonstrating the entire execution process using the family database in Section 5.1. Given the database consisting of four facts and one rule, a question "?- grandmother_of(jane, conrad)." is asked. The numbered arrows in the figure show the execution steps.

1. The goal "?- grandmother_of(jane, conrad)." will be matched with each item in the database, starting from the beginning.
2. A match is found with the rule.
3. The match instantiates the variables X and Z to jane and conrad, respectively.
4. Since the match is a rule, in order for the rule to be true, the composite condition must be true. The subgoal is to check if mother_of(jane, Y) is true. Y has not yet been instantiated.
5. The subgoal is matched with the database from the beginning.
6. A match is found with the first condition. Notice here that the second fact has not been matched when it moves forward to check the second condition. Thus, a backtrack point needs to be set between the first and the second facts.
7. Variable Y is now instantiated to edith by the match.
8. Now we need to match the second condition mother_of(Y, Z), where Y and Z have been instantiated to jane and conrad, respectively.
9. The subgoal is taken back to the database. No match could be found throughout the entire database and thus the subgoal fails.

10. Since the second and third conditions are connected by an "or" operation, it can give a solution as long as one of the two conditions succeeds. Thus, we need to check father_of(Y, Z), where Y and Z have been instantiated to jane and conrad, respectively. No match can be found for this subgoal and thus the third condition fails. Since both the second and third conditions failed, the goal grandmother_of(jane, conrad) fails.
11. However, the entire database has not been completely searched. The control will return to the backtrack point set at step 6. There is another match for the first condition: mother_of(jane, Y).
12. Variable Y is instantiated to mike and the subgoal becomes mother_of(jane, mike).
13. Now we need to match the second condition mother_of(Y, Z), where Y and Z have been instantiated to mike and conrad, respectively. No match could be found and thus it fails.
14. Now we need to match the third condition father_of(Y, Z), where Y and Z have been instantiated to mike and conrad, respectively. No match could be found and thus it fails.

After the 14 steps of exhaustive comparison, no match could be found for the goal grandmother_of(jane, conrad) and thus the goal failed.

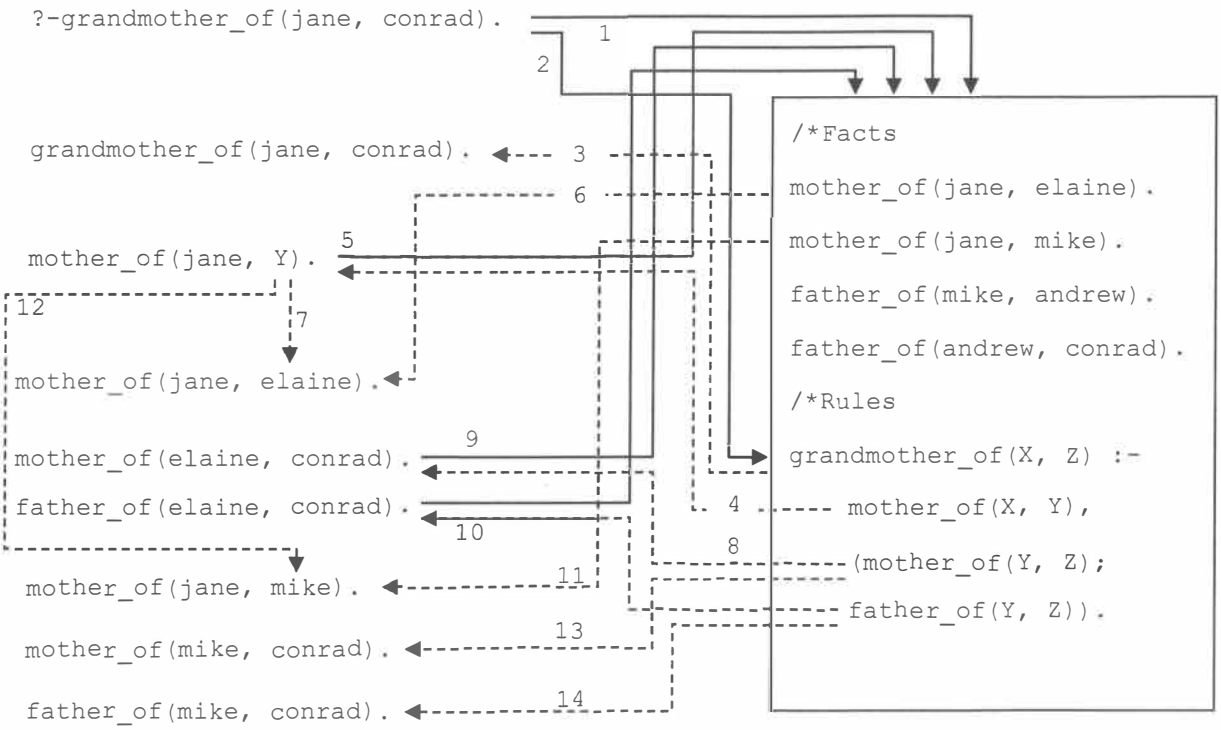


Figure 5.2. Demonstrating the execution model.

5.3 Arithmetic operations and database queries

Prolog is not only a database query language, but also a general programming language that is capable of performing general-purpose computations, with a comprehensive set of operators and built-in functions.

5.3.1 Arithmetic operations and built-in functions

This section presents the frequently used arithmetic operations and functions that are supported in GNU Prolog, as listed in Table 5.1 (Source: http://www.gprolog.org/manual/gprolog.html).

•

Operators and expressions	Meaning or results
pi, e	Constant values 3.145926... and 2.718281...
inc(E)	E+1
E1+E1, E1-E2, E1*E2, E1/E2, E1//E2	Addition, subtraction, multiplication, division, integer division
E1 rem E2, E1 div E2	Remainder and quotient of E1 divided by E2
E1 mod E2	Modular operation
abs(E)	Absolute value of E
min(E1,E2), max(E1,E2)	Minimal and maximal value between E1 and E2
sqrt(E)	Square root of E
sin(E), cos(E), tan(E), atan(E)	Sine, cosine, tangent, arc tangent of E
E1 ^ E2, E1 ** E2	E1 raised to the power of E2
exp(E)	e raised to the power of E
log(E), log10(E)	Natural logarithm and base 10 logarithm of E
X is expression	Instantiate the value of the expression with X
Numeric: E1 =:= E2 Nonnumeric: ==(X1, X2)	E1 =:= E2 succeeds if E1 equals to E2. For example, 2*2 =:=2+2 will succeed, while ==('Apple', 'Orange') will fail.
Numeric: E1 =\= E2 Nonnumeric: \== (X1, X2)	E1 =\= E2 succeeds if E1does not equal to E2. For example, 3*3 =\= 3+3 will succeed, while \==('Apple', 'Apple') will fail.
Numeric: E1 < E2 Nonnumeric: @< (X1, X2)	E1 < E2 succeeds if E1 is less than E2 @< (X1, X2) succeeds if X1 is less than X2
Numeric: E1 =< E2 Nonnumeric: @=< (X1, X2)	E1 =< E2 succeeds if E1 is less than or equal to E2 @=< (X1, X2) succeeds if X1 is less than or equal to X2
Numeric: E1 > E2 Nonnumeric: @> (X1, X2)	E1 < E2 succeeds if E1 is greater than E2 @> (X1, X2) succeeds if X1 is greater than X2
Numeric: E1 >= E2 Nonnumeric: @>= (X1, X2)	E1 >= E2 succeeds if E1 is greater than or equal to E2 @>= (X1, X2) succeeds if X1 is greater than or equal to X2
X is E	Assignment, e.g., X is 3*(5+7). Y is 2**10.

Table 5.1. GNU Prolog arithmetic operators and built-in functions.

Notice that:

- In GNU Prolog, arithmetic expressions can be written in infix notation and in prefix notation. For example, both expressions 2*2 =:= 2+2 and =:=(2*2, 2+2) will succeed and return true. Similarly, both expressions ==(apple, orange) and apple == orange are valid in syntax. As a clause, an arithmetic operation always return true, so that the next clause connected by the "and" operator will be executed.
- Prolog uses >= for greater than or equal to; however, it uses =< for less than or equal to, instead of using <= as is used by C, C++, and Java.

Without writing a program of your own, you can try following built-in arithmetic functions. If you never used a Prolog environment, you can see in the tutorial in Appendix B.4 to get started with using the GNU Prolog programming environment.

```
?- Y is 5*8.                    % 1. return Y = 40, yes
?- Y is 2**10.                  % 2. return Y = 1024, yes
?- Y is X+2*5.                  % 3. What would be the return value?
?- 2*2 =:= 2+2.                 % 4. returns yes.
?- =:=(2*2, 2+2).               % 5. returns yes.
?- ==(apple, orange).           % 6. returns no.
?- apple == apple.              % 7. returns yes.
?- apple @=< orange.            & 8. returns yes.
?- Y is 5*8, 3*3 =:= 3+3.       % 9. no
?- Y is 5*8, 3*3 =:= 3+3+3.     % 10. return Y = 40, yes
```

What does the Prolog runtime print (output) for a query?

If the goal as a clause returns a **false** value, it simply prints "no." See examples 6 and 9 in the code above. Variables and their values will not be printed. One can use "write" clause in the goal to print the names and values one wants to print. If the goal as a clause returns a **true** value, it will print "yes" at the end. If there is a variable in the goal, it also prints variable name = its value.

The runtime also prints an error message if there is an error. An error is neither true nor false. In example 3 above, the goal "?- Y is X+2*5" will return an instantiation error, because X does not have a value, and it cannot appear on the right-hand side of an assignment statement.

5.3.2 Combining database queries with arithmetic operations

As an example, the code below shows a weather database, in which the seasons of the major cities are defined by weather/3 facts. The nearby cities are defined by nearby/2 facts. A hot/1 and a cold/1 rules are used to link the nearby city with the major city. The example also introduces arithmetic operations into database queries.

```
%Facts
weather(phoenix, spring, hot).
weather(phoenix, summer, hot).
weather(phoenix, fall, hot).
weather(phoenix, winter, warm).
weather(charlotte, spring, warm).
weather(charlotte, summer, hot).
weather(charlotte, fall, warm).
weather(charlotte, winter, cold).
weather(minneapolis, winter, cold).
weather(minneapolis, spring, cold).
weather(minneapolis, summer, warm).
weather(minneapolis, fall, cold).
nearby(tempe, phoenix).
nearby(mesa, phoenix).
```

```
nearby(scottsdale, phoenix).
nearby(burnsville, minneapolis).
% Rules
hot(C) :-   (weather(C, spring, hot), weather(C, fall, hot));
            (D\==C, nearby(C, D), hot(D)).
cold(C) :-  (weather(C, spring, cold), weather(C, fall, cold));
            (D\==C, nearby(C, D), cold(D)).
```

By entering the "trace." goal in GNU Prolog, the following executions will be traced, which lists how each unification step is performed. To cancel the trace, enter the goal "notrace." For example, tracing the execution of the goal cold(X) and asking what city is cold, generate the following output.

```
| ?- trace.
The debugger will first creep -- showing everything (trace)
| ?- cold(X).
      1    1  Call: cold(_16) ?
      2    2  Call: weather(_16,spring,cold) ?
      2    2  Exit: weather(minneapolis,spring,cold) ?
      3    2  Call: weather(minneapolis,fall,cold) ?
      3    2  Exit: weather(minneapolis,fall,cold) ?
      1    1  Exit: cold(minneapolis) ?
X = minneapolis ? ;
      1    1  Redo: cold(minneapolis) ?
      2    2  Redo: weather(minneapolis,spring,cold) ?
      2    2  Fail: weather(_16,spring,cold) ?
      2    2  Call: _84\==_16 ?
      2    2  Exit: _86\==_16 ?
      3    2  Call: nearby(_16,_109) ?
      3    2  Exit: nearby(tempe,phoenix) ?
      4    2  Call: cold(phoenix) ?
      5    3  Call: weather(phoenix,spring,cold) ?
      5    3  Fail: weather(phoenix,spring,cold) ?
      5    3  Call: _157\==phoenix ?
      5    3  Exit: _159\==phoenix ?
      6    3  Call: nearby(phoenix,_182) ?
      6    3  Fail: nearby(phoenix,_170) ?
      4    2  Fail: cold(phoenix) ?
      3    2  Redo: nearby(tempe,phoenix) ?
      3    2  Exit: nearby(burnsville,minneapolis) ?
      4    2  Call: cold(minneapolis) ?
      5    3  Call: weather(minneapolis,spring,cold) ?
      5    3  Exit: weather(minneapolis,spring,cold) ?
      6    3  Call: weather(minneapolis,fall,cold) ?
      6    3  Exit: weather(minneapolis,fall,cold) ?
      4    2  Exit: cold(minneapolis) ?
      1    1  Exit: cold(burnsville) ?
```

```
X = burnsville ? ;
      1    1  Redo: cold(burnsville) ?
      4    2  Redo: cold(minneapolis) ?
      5    3  Redo: weather(minneapolis,spring,cold) ?
      5    3  Fail: weather(minneapolis,spring,cold) ?
      5    3  Call: _157\==minneapolis ?
      5    3  Exit: _159\==minneapolis ?
      6    3  Call: nearby(minneapolis,_182) ?
      6    3  Fail: nearby(minneapolis,_170) ?
      4    2  Fail: cold(minneapolis) ?
      1    1  Fail: cold(_16) ?
No
```

The execution is done in three phases. In the first phase, a match cold(minneapolis) is found from the weather/3 facts. After a semicolon is entered, the search continues, and the nearby facts are compared, where a match cold(burnsville) is found. After another semicolon is entered, the search continues, but no match is found and a "No" output is printed.

In the following sections, we will use arithmetic operations and functions in many different situations, particularly in recursive functions.

5.4 Prolog functions and recursive rules

We have discussed the fantastic-four abstract approach of writing recursive functions in C/C++ (Section 2.7) and in Scheme (Section 4.7). The same approach can be applied to writing Prolog recursive rules.

5.4.1 Parameter passing in Prolog

Prolog clauses that perform arithmetic functions are often called Prolog functions. Unlike in C/C++ and Scheme functions, a Prolog function cannot have a return value, except a Boolean value, and thus an argument must be used for passing the return value to the outside of the function. Table 5.2 lists the parameter-passing mechanisms supported by C/C++, Java, Scheme, and Prolog. For the Prolog column, the items are explained below:

- **Call-by-value and call-by-alias:** These are the main parameter-passing mechanisms. All the parameters (arguments) can be used for both mechanisms: They can take values as input and can pass results as output of the function.
- **Return value:** Prolog functions or clauses return a Boolean (logical) type value (true or false), and they never return other types of values. Thus, if a nonlogical return value is needed, an argument must be defined for that.
- Function as the first class object is a feature of a functional programming language. Scheme fully supports this feature. C/C++ and Java partly support this feature. For example, we can place an expression, such as X+5, in a print statement. However, you cannot use write(X+5) in Prolog. You must use Y is X+5, write(Y). Thus, Prolog does not support this feature at all.

We will see more examples that illustrate the features in this table in later discussions.

Parameter-passing mechanism	C/C++	Java	Scheme	Prolog
Call-by-value	Always	Primitive type	Always	Always
Call-by-alias	Always	Object type	Never	Always
Return value	Return value or void	Return value or void	Always	True or false, never other types
Function as the first class object	Not always	Not always	Always	Never

Table 5.2. Parameter-passing mechanisms of different languages.

5.4.2 Factorial example

A Prolog rule is recursive if a clause in the condition part has the same predicate and arity as the conclusion. We begin with a simple example of calculating the factorial(N) = N*(N−1)* … *2*1.

To define the factorial function, we need two arguments, one for passing in the input N and one for passing the result F. Below is the recursive rule that implements the factorial function:

```
factorial(0, 1).
factorial(N, F) :- N>0, N1 is N-1, factorial(N1, F1), F is N * F1.
```

The Prolog recursive function consists of two rules. The first rule is the stopping condition. If N is 0, then F will unify with value 1 and thus the return value of the function will be F = 1. As this rule does not have a condition, the stopping condition is a fact.

The second rule is the main part of the recursive function that implements the other three steps of the fantastic-four abstract approach:

Size-N problem: factorial(N, F).

Size–(N−1) problem: factorial(N1, F1). Notice that we cannot use factorial(N−1,F1) in Prolog, because Prolog does not support the feature of a function as the first-class object, and thus, the function N−1 cannot be placed in the place where the result of N−1 is expected. We have to first calculate the value of N−1, and then place the value into the function.

Constructing the size-N problem's solution from the assumed size-(N−1) problem's solution: F is N * F1.

The order of Prolog rules is important. Normally, the rule representing the stopping condition must be placed before the other rules of the recursive function. We can call the rule using different inputs:

```
?- factorial(3, 5).      % return: no
?- factorial(3, 6).      % return: yes
?- factorial(4, F).      % return: F = 24
?- factorial(N, 6).      % What does this goal return?
```

In most cases, a Prolog argument can be used for both input and output. However, when arithmetic operations and assignments are involved, such as N > 0 and F is N * F1, all the variables on the right-hand side must have been instantiated. In the example above, factorial(N, 6), an instantiation error will occur, because N does not have a value and N > 0 needs to be performed.

5.4.3 Fibonacci numbers example

As a slightly more complex example, we implement the Fibonacci numbers function, which is defined as follows:

$$fib(N) = \begin{cases} 0 & if\ N = 0 \\ 1, & if\ N = 1 \\ fib(N-1) + fib(N-2), & if N \geq 2 \end{cases}$$

The recursive rules that implement the Fibonacci numbers function are as follows:

```
fib(F, N) :- N =:= 0, F is 0.     % Stopping condition 1 and return value
fib(F, N) :- N =:= 1, F is 1.     % Stopping condition 2 and return value
fib(F, N) :- N > 1,
      M1 is N-1,
      M2 is N-2,
      fib(F1, M1),                % Size N-1 problem
      fib(F2, M2),                % Size N-2 problem
      F is F1 + F2.               % Constructing size-N problem solution
```

The example is implemented by directly following the abstract approach of recursive function design. As defined in the Fibonacci formula, there are two stopping conditions and there are two size-M problems, with M1=N−1 and M2=N−2. We assume that fib(F1, M1) and fib(F2, M2) will solve the two size-M problems and place the results in the calculation in F1 and F2. Then, the size-N problem is solved by F1+F2.

5.4.4 Hanoi Towers

As we have discussed in C/C++ (Section 2.7) and in Scheme (Section 4.7), the Hanoi Towers game is a good example for showing recursive function design. Please read Section 2.7 for the rules for playing the game. Using Prolog, the puzzle can be simply solved by the following recursive rules:

```
hanoi(N) :- move(N, source, center, destination). %Helper rule
move(1, S, _, D) :-   % stopping condition
  write('Move top from '), write(S), write(' to '), write(D), %Output
  nl.           % nl = newline
move(N, S, C, D) :-  % Size-N problem
  N>1,
  M is N-1,
  move(M, S, D, C),   % Size-(N-1) problem: move N-1 disks from S to C
  move(1, S, _, D),   % move remaining 1 from S to D
  move(M, C, S, D).   % Size-(N-1) problem: move N-1 disks from C to D
```

The first line of code introduces a helper to simplify the call to the rule. A user can call the rule by using one argument, instead of four arguments.

The recursive rules have four arguments, defining the size N and the three pegs on which the disks can be placed. The rule is defined following the abstract approach. The stopping condition is the case when there is one disk in the game. In this case, we simply move the disk from S (source) to D (destination). The C (center) peg is not necessary in this case, and thus we used an anonymous variable in the place of center.

The size-(N−1) problem is the case when we have N−1 disks. These N−1 disks can be on the source peg or on the center peg. The key to the solution of the size-N problem is to assume that we can move these N-disks together. Notice that this assumption does not violate the rules for playing the game. The size-(N−1) problem is solved step by step, by following the playing rules of recursive mechanism.

If we call the rule using the goal Hanoi(3), we have the following outputs. Examining the steps, we can see that none of the steps moved more than one disk or placed a larger disk on a smaller disk.

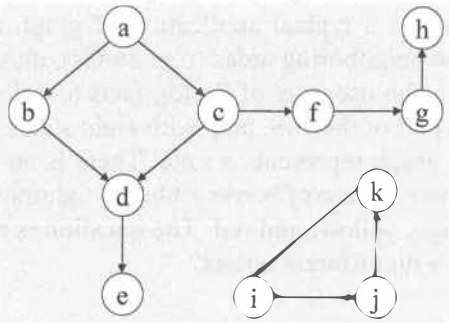
```
Move top from source to destination
Move top from source to center
Move top from destination to center
Move top from source to destination
Move top from center to source
Move top from center to destination
Move top from source to destination
```

5.4.5 Graph model and processing

Before we start to define Prolog rules on graph, we first learn the basics of graph. A graph is a mathematical model and a data structure that is widely used for representing related objects. A graph consists of a set of nodes and a set of edges between the nodes. A graph is a **directed graph** if a direction is defined for each edge, and a graph is an **undirected graph** if there is no direction defined for any edge. Assuming that the direction of the edge is the driving direction of streets, a directed graph allows the flow following the edge directions. An undirected graph allows the flow in both directions of the edge. A **path** from Node `N1` to Node `N2` is a sequence of edges connecting `N1` to `N2`, for example, `(N1, X1), (X1, X2), (X2, X3), ..., (Xk, N2)`. We also say node N1 is **connected** to node `N2` if there is a path from `N1` to `N2`.

In this section, we represent a directed graph in a factbase and define recursive rules to search the graph. We use a set of edges to represent a graph, where each edge consists of a pair of nodes attached to the edge. An example is given below.

```
edge(a,b).
edge(a,c).
edge(b,d).
edge(c,d).
edge(c,f).
edge(d,e).
edge(f,g).
edge(g,h).
edge(i,j).
edge(j,k).
edge(k,i).
```



Now, we can define rules to search the factbase and check if any given nodes are connected through a path. Consider the following definition of connected rules:

```
connected(Node1, Node2) :- edge(Node1, Node2).
connected(Node1, Node2) :- connected(Node1, X), connected(X, Node2).
```

This set of rules are recursive rules, where the first rule is the stopping condition, while the second rule constructs the size-N problem's solution from the solutions of two smaller problems. In words, it says that Node1 is connected to Node2, if Node1 is connected to node X, and X is connected to N2. However, the definition uses two recursive calls in the second rule, which can make the search unnecessarily complex. The two recursive rules are not based on the stopping condition, which can lead to an infinite loop. A better way of defining the rules is to use the stopping condition and use one recursive call only:

```
connected(Node1, Node2) :- edge(Node1, Node2).
```

```
connected(Node1, Node2) :- edge(Node1, X), connected(X, Node2).
```

Now let us consider how we can convert the directed graph into an undirected graph. As an undirected graph allows the flow to go in both directions on each edge, we can simply add a mirror edge(b, a) into the factbase if there exists edge(a, b). This approach works, but can cause the data inconsistency problem, as the mirror edges provide redundant facts. A better way is to define a rule to describe the fact that each edge is bidirectional. Consider adding the following mirror rule into the graph factbase:

```
edge(X, Y) :- edge(Y, X).
```

Will this simple rule turn the directed graph into an undirected graph? The answer is no! Carefully examining this rule, we can see it is a recursive rule, as the condition part of the rule calls its conclusion part. Once we have identified that the rule is a recursive rule, we know it is incorrect, because it does not follow the four-step abstract approach of designing recursive rules. There is no stopping condition, and thus the rule can enter into an infinite loop. For example, the goal

```
?- edge(a, d)
```

will cause an infinite loop. Of course, not all goals will land into an infinite loop.

To address such a problem, we can define a helper rule: adjacent(X, Y) :- edge(X, Y); edge(Y, X). Then, we use the helper rule to define the connected rule without following the directions of the edges.

```
adjacent(X, Y) :- edge(X, Y); edge(Y, X).
connected(Node1, Node2) :-adjacent(Node1, Node2).
connected(Node1, Node2) :- adjacent(Node1, X), connected(X, Node2).
```

5.4.6 Map representation and coloring

Map coloring is a typical application of graph theory and artificial intelligence. To make a map more readable, the neighboring areas (e.g., states), must be marked in different colors. Taking a U.S. map as an example, we can use a set of Prolog facts to define the states and colors marked on each state. Figure 5.3 illustrates a part of the U.S. map with eight states on the west, and the graph model of the eight states. Each node in the graph represents a state. There is an undirected edge between two nodes if the two states are adjacent (share a piece of border). On the rightmost part of the figure, the graph is colored by three different colors: orange, yellow, and red. The question is are the colors correctly applied so that neighboring states are marked with different colors?

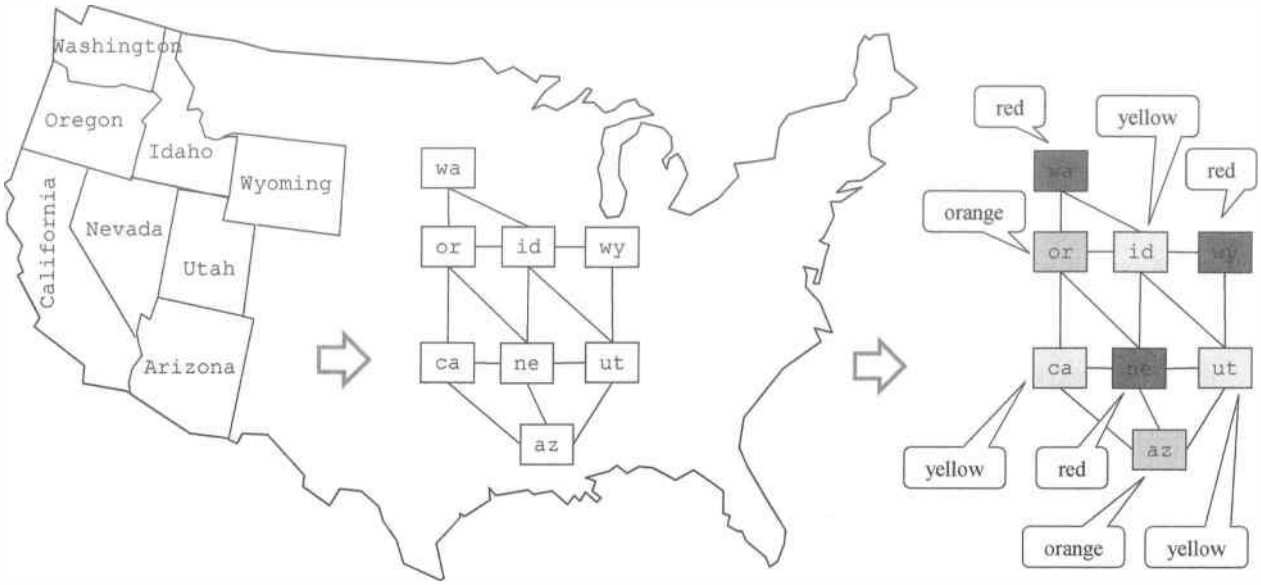


Figure 5.3. Map, its graph model, and coloring.

It may be easy enough to answer the question by desk-checking the colors if we have only seven states in the graph. However, after we have added all the states into the graph, the problem is no longer trivial. To solve the problem in general, we can use a set of Prolog facts and rules:

```
edge(az, ca).          color(az, orange).
edge(az, ne).          color(ca, yellow).
edge(az, ut).          color(ne, red).
edge(ca, ne).          color(ut, yellow).
edge(ca, or).          color(or, orange).
edge(ne, ut).          color(id, yellow).
edge(ne, or).          color(wy, red).
edge(ne, id).          color(wa, red).
edge(ut, id).
edge(ut, wy).
edge(id, wy).
edge(id, wa).
edge(or, id).
edge(or, wa).
adjacent(X, Y) :- edge(X, Y); edge(Y, X).
miscolor(S1, S2, Color) :-
   adjacent(S1, S2), color(S1, Color), color(S2, Color).
```

The facts edge/2 and color/2 are straightforward. The adjacent rule is the same rule we discussed in the previous graph example. The rule miscolor/3 has three conditions, which check if the two states are adjacent and if the two states have the same color.

Now, we can test the program by asking the question with three variables: miscolor(S1, S2, C). The question will generate the following output:

```
C = yellow
```

```
S1 = ut
S2 = id ? ;
C = yellow
S1 = id
S2 = ut ? ;
no
```

From the output we can see that ut (Utah) and id (Idaho) are adjacent, yet both are colored in yellow. A coloring error is detected. To fix the error, we can use another color for ut or id, for example, green. After fixing the error and testing the program again, no error is detected:

```
| ?- miscolor(S1, S2, C).
no
```

5.5 List and list manipulation

Similar to what we discussed in Scheme, Prolog uses pairs to define the lists. All the concepts, rules, and operations of pairs and lists discussed in Scheme can be applied to Prolog, except that the syntax is different.

5.5.1 Definition of pairs and lists

Pair is a structured data type in Prolog. A pair consists of two items in a pair of brackets and the two items are separated by a vertical bar: `[X | Y]`. Using BNF notation, a Prolog pair is defined as

<pair> ::= [<A> | <B>]

where both A and B can be a variable, value, or an item of any data type. Using the terminology of Scheme, A is the value of (car pair) and B is the value of (cdr pair). Below are a few examples of pairs:

```
[1 | 2]
[1 | [2 | 3]]
[1 | [2 | [3 | []]]]
[[2 | [8 | 7]] | [4 | 8]]
[12 | [2 | [8 | [4 | [3 | 7]]]]]
```

Similar to a Scheme list, a Prolog list can be defined by pairs recursively:

```
  <list> ::= [] (empty list)
  <list> ::= [<H> | <list>]
where
  <H> is a variable, a value, or an item of any type;
```

According to the definition, all lists are lists, except that the empty list is not a pair. However, there are many pairs that are not lists.

The execution (search) model of Prolog does not really differentiate between a list and a pair. The rules defined on lists can be applied on pairs, even if the pairs are not lists. However, results can be surprising when a pair is not a list. For example, the membership rule can be applied to pairs:

```
?- member(1, [1 | [2 | 3]]). % It returns "true"
?- member(2, [1 | [2 | 3]]). % It returns "true"
?- member(3, [1 | [2 | 3]]). % It returns "false"
```

The pair [1 | [2 | 3]] is not a list. However, the first two elements 1 and 2 are in the same positions as list members, and thus the search can find them as members, and element 3 is in a position that the membership rule cannot find and thus it returns "false." We will discuss this in detail later when we discuss the definition of the membership rule, and the way the member rule executes.

5.5.2 Pair simplification rules

To reduce the complex appearances of nested pairs, Prolog allows us to apply the following **pair simplification rule** to simplify the notation of pairs:

- A (vertical) bar and the left bracket to the right of the bar can be replaced by a comma, if the item to the right of the bar is a pair. After the left bracket is removed, the corresponding right bracket must be removed.
- If a (vertical) bar is followed by an empty list, the bar and empty list can be removed.

We can apply these simplification rules to the pairs below by underlining the bars and the adjacent pairs or empty list:

```
[1 | 2]                          → cannot be simplified
[1 | [2 | 3]]                    → [1, 2 | 3]
[1 | [2 | [3 | []]]])            → [1, 2, 3])
[[2 | [8 | 7]] | [4 | 8]]        →[[2, 8 | 7], 4 | 8]]
[9 | [2 | [8 | [4 | [3 | 7]]]]]  →[9, 2, 8, 4, 3 | 7]
```

After replacement and removal, the simplified pairs are given in the list above. Prolog supports both the full notation and the simplified notation of pairs.

The pair representation and simplification rules are similar to those of Scheme. When reading this section, please compare and contrast with the Scheme pairs in Section 4.4.6. Table 5.3 lists a few examples of Scheme and Prolog pairs, and their full notations and simplified notations.

Scheme pairs: full and simplified notations		Prolog pairs: full and simplified notations	
(1 . 2)	(1 . 2)	[1 | 2]	[1 | 2]
(1 . (2 . 3))	(1 2 . 3)	[1 | [2 | 3]]	[1, 2 | 3]]
(1 . (2 . (3 . ())))	(1 2 3)	[1 | [2 | [3 | []]]]	[1, 2, 3]
((2 . (8 . 7)) . (4 . 8))	((2 8 . 7) 4 . 8)	[[2 | [8 | 7]] | [4 | 8]]	[[2, 8 | 7], 4 | 8]]
(9 . (2 . (8 . (4 . (3 . 7)))))	(9 2 8 4 3 . 7)	[9 | [2 | [8 | [4 | [3 | 7]]]]]	[9, 2, 8, 4, 3 | 7]

Table 5.3. Comparison between Scheme and Prolog pairs.

Representing a nonempty list as a pair [H | T] is very useful when writing recursive rules, where T is in most cases the argument to the size-(N−1) problem. It is also helpful when learning Prolog to reread the Scheme chapter on solving recursive problems. For example, when writing recursive rules to reverse a list, we can consider the following cases:

- **Define the size-n problem:** reverse(L, RL), where L is the list to be reversed and RL will hold the result (i.e., the reversed list).
- **Define the stopping condition and its return value:** If the list is empty (stopping condition), then the reversed list is an empty list (return value).

- **Identify the size-(n−1) problem and assume the problem is solved:** Since the list is not empty, it can be represented as a pair [H | T]. T is a list of size n−1 and thus, the size-(n−1) problem is reverse(T, RT). We assume that we have obtained RT, which is the reversed list of T.
- **Construct the solution to the size-n problem based on the hypothetic solution in the last step:** Since RT is reversed from T, and H is a first element of list L, if we insert H at the end of RT, we have the complete list reversed.

The complete recursive function (rules) for reversing a list is shown below:

```
reverse([ ],[ ]).                /* Stopping condition */
reverse([X | T],RL) :-           /* size-n problem */
  reverse(H, RT),                /* size-(n-1) problem */
  append(RT, [H], RL).           /* construct size-n problem's solution */
```

We will discuss many more list-related recursive examples in the following sections.

5.5.3 List membership and operations

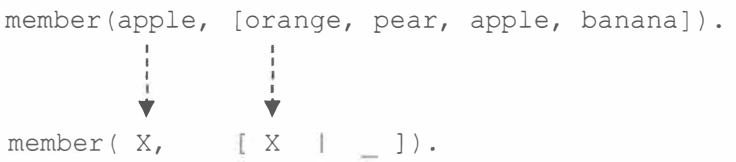
Prolog offers powerful operations to manipulate lists and list members. Many of the operations are related to the membership rules. The membership rules can be defined as

```
member(X, [X | _ ]).
member(X, [ _ | T ]) :- member(X, T).
```

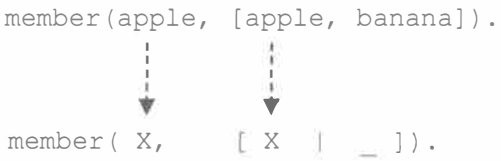
In words, the two rules state:

- X is a member of a list whose first element is X. An anonymous variable is used in the tail part of the list, because we do not care what it is if we have found that the element matches the first element of the list.
- X is a member of a list whose tail is T if X is a member of T. An anonymous variable is used in the head part of the list, as the head has been checked in the first rule.

To see how the member rules work, let us examine a query ?- member(apple, [orange, pear, apple, banana]). The query will be unified with the first rule member(X, [X | _]), where the first X unifies with apple, while the second X unifies with orange, as shown in the diagram below. The unification fails.



Then, the query tries the second rule member(X, [_ | T]) :- member(X, T). This rule removes the head element orange, resulting in a query ?- member(apple, [pear, apple, banana]). A similar situation will occur, where the first X unifies with apple, while the second X unifies with pear. The unification fails again. As the rules are recursive, it continues to remove the head element, and it then enters the query ?- member(apple, [apple, banana]). Now, the first X unifies with apple, and the second X also unifies with apple, as shown in the diagram below. The goal succeeds.



What would happen if we asked the question ?- member(apple, [apple | orange])? The query would successfully unify with the first member rule and the goal would succeed.

What would happen if we asked the question ?- member(orange, [apple | orange])? Obviously, the orange does not unify with apple and the first rule fails. When matching with the second rule, the tail part of the pair [apple | orange] would be used as the second argument, resulting in the query ?- member(orange, orange). Would this goal succeed? No. The second argument of member rule is not a list. The element range can be a member of a list [orange], but it cannot be a member of orange.

Similar to the Scheme list functions `car` and `cdr`, we can define the rules to extract the first element and the remaining list:

```
car(X, [X |_].
cdr(X, [_ |X].
```

We can also define rules to extract the last element of a list:

```
last(X, [X]).                    % stopping condition
last(X, [ _ | Tail]) :- last(X, Tail).   % recursive
```

The second rule removes the head repeatedly until there is only one element left. Then, the stopping condition will return the element as output.

Another frequently used list operation is `append`. The rule has three arguments, where the first two arguments are input lists, and the third argument is the resulting list:

```
append([ ], X, X).                % stopping condition
append([X | Y], Z, [X | W]) :-    % size-n problem & construction
  append(Y, Z, W).                % size-(n-1) problem
% goal example
?- append(H, [d, f, g], [x, a, b, c, d, f, g])
  H = [x, a, b, c]
```

The append/3 rule can be used to define the list reverse rules:

```
reverse([ ],[ ]).              % Stopping condition
reverse([X | L],Rev) :-        % size-n problem
  reverse(L, RL),              % size-(n-1) problem
append(RL, [X], Rev).          % construction
```

To count the number of members, we can define the count rules:

```
count([ ], 0).                 % Stopping condition
count([_ | Tail], S) :-        % Size-N problem
  count(Tail, S1),             % Size-(N-1) problem
  S is S1+1.                   % Construct the solution to Size-N problem
```

The idea is to assume that the recursive clause has found that the count of the tail is S1, and the total count with the head is simply S1+1. The accrual process is to repeatedly remove the head and add 1 to the sum

till the list becomes empty. Notice that there is a built-in rule in GNU Prolog that performs the same function: length(List, Len).

If the list consists of numbers only, we can revise the count/2 rule to perform summation. The only difference to add the element, instead of adding 1, in each step of the recursion:

```
sum_list([ ], 0).                    % stopping condition
sum_list([Head|Tail], Sum) :-
  sum_list(Tail, Sum1),
  Sum is Head+Sum1.                  % Adding the value of Head
```

As an example, consider an industry application of the membership rules. In a computer assembly line, workers are supposed to put the following components in each carton box. The weight of the carton box is 3 lb, and the weights of the other components are

- a chassis, with weight 27 lb;
- a monitor, with weight 7 lb;
- a keyboard, with weight 5 lb;
- a set of speakers, with weight 4 lb.

The correct weight of each completed box should be 27+7+5+4+3 = 46 lb. However, human errors can be made. They may miss a component or put an additional component in the box. Assume that the maximum weight that the box can possibly hold is 50 lb, excluding the carton box's weight. To detect errors, the last step of the assembly line is to weigh the completed box. If the total weight is not equal to 46 lb, an error must have occurred.

Based on the requirement, we can write a weight-check rule to detect how many components of each type is placed in the box. Based on the total weight that the box can possibly hold, the numbers of components that the box can hold are

- 0 or 1 chassis
- 0, 1, 2, 3, 4, 5, 6, or 7 monitors
- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, or 10 keyboards
- 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, or 12 sets of speakers

Using the member rules, we can implement the weight-check in the following Prolog program.

```
weightcheck (C, M, K, S, W) :-
      member(C, [0, 1]),
      member(M, [0, 1, 2, 3, 4, 5, 6, 7]),
      member(K, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
      member(S, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]),
      W = 27*C + 7*M + 5*K + 4*S + 3,
      W =:= 46.
```

We can ask different questions to check if the weight is correct, and how many components are placed in the carton box. For example,

```
?- weightcheck(1, 1, 0, 2, W).
```

will return the result: W = 49.

5.5.4 Knapsack problem

The example of the membership rules can be further applied to solve optimization problems, such as the Knapsack problem. Knapsack problem is a problem in combinatorial optimization. Given a set of items, each with a weight and a value, both have integer values, we need to write a program to find the number of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is at its maximum (http://en.wikipedia.org/wiki/Knapsack_problem). Knapsack problem's algorithm is complex in execution time, as the algorithm must try all the combinations.

There are many applications of the Knapsack problem. A typical example showing this problem is the thief's backpack problem. A thief breaks into a store and wants to fill the backpack with as much value as possible, under the condition that the backpack has a limit in weight to carry. Here we will show a variation of the problem, search for the minimum value, instead of the maximum value.

In a buffet restaurant, one can eat unlimited food. For simplicity, we assume only these items are available: pizza slices, salad bowls, meatballs, and bags of fries. These items have the following weights and calories:

- **Pizza slice:** The weight of each slice is 5 ounce and the calories are 150;
- **Salad bowl:** The weight of each bowl of salad is 7 ounce and the calories are 50;
- **Meatball:** The weight of each meatball is 3 ounce and the calories are 100;
- **Fries:** The weight of each bag of fries is 4 ounce and the calories are 200.

Assume a person needs to eat certain amount of food exceeding a given weight, 20 ounce, in order to obtain sufficient energy for the day. On the other hand, the person wants to consume the minimum amount of calories to keep healthy.

You want to write a program to help the customers to find the food that meets the weight requirement and have the minimum amount of the calories. We can write a rule to find how many items of each kind can be included for a given total weight, based on the total weight and the weight of each item. Then, based on the number of items, we can calculate the total calories of the food. We assume that the person will not eat any single type of items whose total weight exceeds the minimum weight (20) needed.

```
caloriesByWeight(P, S, M, F, Weight, Calories) :-
  member(P, [0, 1, 2, 3, 4]),
  member(S, [0, 1, 2, 3]),
  member(M, [0, 1, 2, 3, 4, 5, 6, 7]),
  member(F, [0, 1, 2, 3, 4, 5]),
  Weight is 5*P + 7*S + 3*M + 4*F,
  Weight >= 20,
  Calories is 150*P + 50*S + 100*M + 200*F.
```

To compare if a particular collection of food has less calories than another collection with the same weight, we can write a rule to make the comparison:

```
moreCalories(W, MoreCal) :-
  caloriesByWeight(_, _, _, _, W, Cal),
  MoreCal > Cal.
```

This rule will return true if MoreCal has a large value than the value Cal calculated from the rule `caloriesByWeight(_, _, _, _, W, Cal)`.

Now, we can find the particular collection of food that has the least amount of calories among all other collections of food with the same weight.

```
leastCalories(P, S, M, F, Weight, MinCalories) :-
  caloriesByWeight(P, S, M, F, Weight, MinCalories),
  not(moreCalories(Weight, MinCalories)).
  % \+ moreCalories(Weight, MinCalories). This is an alternative to not().
```

In this rule, we calculate the calories of a given collection of food using `caloriesByWeight(P, S, M, F, Weight, MinCalories)`. Then, use the next rule to make sure that the calculated calories is NOT more than any other collections. We use the not(X) rule to make sure that all the combinations will be checked. If not rule is not predefined in your programming environment, you can define your own not rule.

```
not(X) :- X, !, fail.    % Define not rule
not(_).
```

Other uses of the not rule will be further discussed in Section 5.6.2.

The use of the "not" rule in Prolog can be problematic, as Prolog will try to prove that the proposition is not true. The only way to prove something is not true in Prolog is an exhaustive search of all possibilities and find no answer. It may take a long time. In the example above, however, searching all possibilities to find the minimum value is exactly what we want.

5.5.5 Quick sort

There exist many sorting algorithms, such as bubble sort, selection sort, merge sort, and quick sort. Quick sort is the fast sorting algorithm among all the sorting algorithms.

Assume the numbers to be sorted are in list List1 and the sorted list is List2. The idea of quick sort is to pick any element, for example, the first element, as the pivot value. Then, quick sort will put the numbers smaller than the pivot into a sublist L1, and put the numbers that are greater than or equal to the pivot into the list L2. We repeat the process for L1 and L2 until all the numbers are sorted.

Figure 5.4 illustrates the process of sorting people by their ages. First, we have all the people lined up in a row. We take the first (leftmost) person as the pivot. Then, all the people who are younger than the pivot person move to the left of that person, and all the people who are not younger than the pivot person move to the right of that person. Then, we recursively repeat the process for the left-side people and the right-side people until there are no more people in the sublists.

The Prolog rules that implement the quick sort process are given as follows:

```
qsort([],[]).                          % empty list is already sorted
qsort([Pivot|Tail],Sorted):-           % Take first number as pivot
  split(Pivot, Tail, L1, L2),          % Call split/4 rule.
  qsort(L1,Sorted1),                   % sort first part
  qsort(L2,Sorted2),                   % sort second part
  append(Sorted1,[Pivot|Sorted2], Sorted).

split(_,[],[],[]).                     % stopping condition
split(Pivot,[X|T],[X|Le],Gt):-         % take first from Tail
  X=<Pivot, split(Pivot,T,Le,Gt).      % and put it into Le
split(Pivot,[X|T],Le,[X|Gt]):-         % take first from Tail
  X > Pivot, split(Pivot,T,Le,Gt).     % and put it into Gt
```

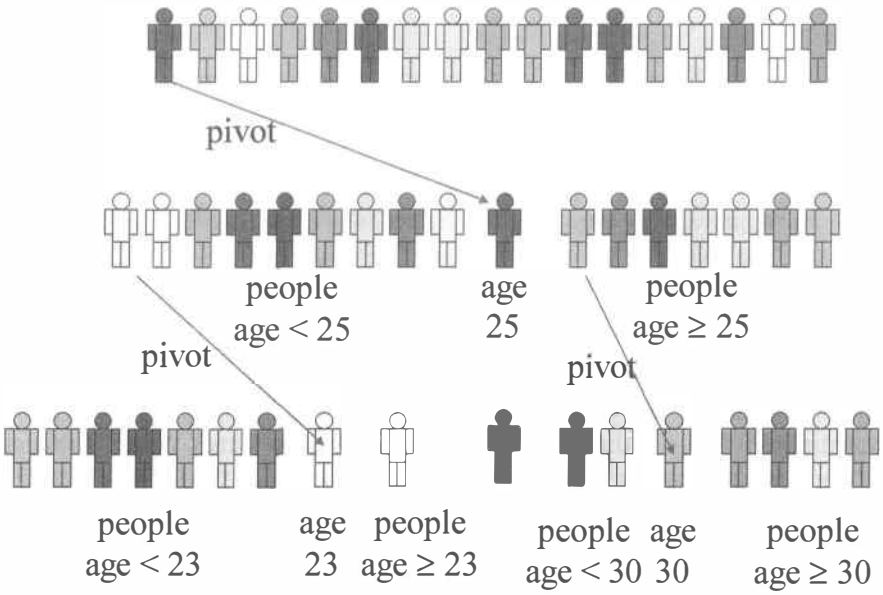


Figure 5.4. Illustration of quick sort of people by age.

The sorting rules consist of two recursive functions. The qsort/2 rule assumes that the split/4 function will split the list into two sublists L1 and L2 around the given pivot value. Then, we recursive call qsort on lists L1 and L2. Assuming that the two size-M problems, where M < N, will be solved (sorted), the last clause of the recursive rule will append the two lists, with the pivot value inserted into the second list before appending.

The second set of recursive rule split/4 compares each element with the pivot value and puts them into the Le list if the element is less than the pivot; otherwise, it puts them into the Gt list.

Calling qsort as a goal, the given list is sorted:

```
| ?- qsort([8, 3, 4, 12, 25, 4, 6, 1, 9, 22, 6], Sorted).
Sorted = [1,3,4,4,6,6,8,9,12,22,25] ?
yes
```

5.6 Flow control structures

So far, we have been focusing on understanding the automated searching process of the Prolog runtime, which exhaustively searches all the possible answers. Now, we will study the ways of changing the search options by removing and adding the backtrack points. A **backtracking** point is a point from which the Prolog runtime will restart its search if the current search fails, or if the current search succeeds but a semicolon is entered thereafter.

There are several built-in clauses that can be used for changing the order of searching the Prolog database. We will discuss three important flow control clauses in this section: ! (`cut`) that removes the backtrack points, `repeat` that adds backtrack points, and `fail` that enables the search to continue from the last backtrack point.

5.6.1 Cut

A `cut` (!) is a special control facility in Prolog that enables programmers to restrict the backtracking options. A `cut` will succeed when it is met (executed). It will remove all existing backtracking points, but new backtracking points may be added thereafter. Notice that a `cut` may cut off valid options and thus the search may not find all the answers even if the semicolon key is typed. Thus, you should use `cut` only if you are sure that there are no more answers or you do not want to have all possible answers. For example, in the factorial rule, we are sure that when `N = 0`, there cannot be any more answers and thus we can use `cut`:

```
factorial(0, 1) :- !.
```

Figure 5.5 shows the search structure (the solid lines) of a Prolog database, where a branch means that there are two possible search branches at the point. The dotted lines show the actual search paths and the circled numbers are backtrack points. The search starts from the beginning of the database. When the first branch is encountered, it continues with, say, the left branch and makes a backtrack point ① to mark that the right branch has not yet been searched. When the second and third branches on the left are encountered, backtrack points ② and ③ are added, respectively. Then the search goes to the end at the leftmost branch and it returns to the latest backtrack point, which is ③. The search removes ③, goes to the end, and then returns to the latest backtrack point, which is now ②. The research process continues until a match is found, or the entire database has been searched.

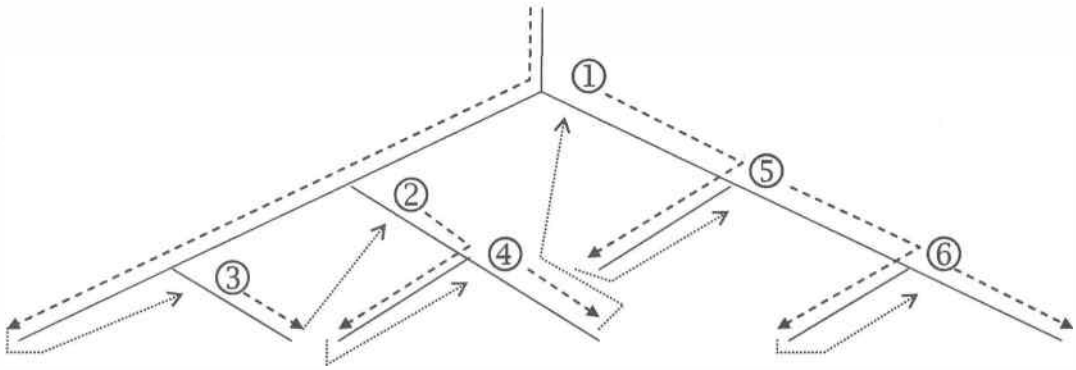


Figure 5.5. Adding backtrack points.

Now examine what happens if a `cut` (!) is used in the rules of the database. As shown in Figure 5.6 (a), after backtrack points ① and ② are created and then a `cut` is executed, all existing backtrack points, in this example, ① and ②, will be removed, as shown in Figure 5.6 (b). A `cut` will succeed when it is executed and thus the search continues as normal. When the next branch is encountered, a new backtrack point ③ will be added. After the search goes to the left end, it returns to the latest backtrack point, which is ③ and continues from ③. Now when the search goes to the point "e," the search terminates and the remaining structure will not be searched, because there are no more backtrack points pointing to indicate what parts have not been searched.

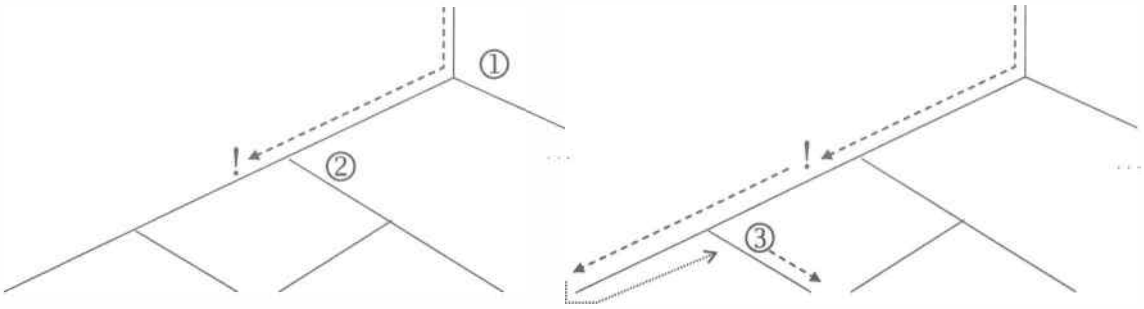


Figure 5.6. (a) A `cut` is executed after creating two backtrack points.
(b) The existing backtrack points are removed and search continues.

The concept is simple. However, it is not trivial to understand what backtrack points are and where backtrack points are added and when they are removed. Consider the following simple database. To understand the effect of `cut`, we define four programming language rules: `plg0` (no `cut`), `plg1` (a `cut` as the first condition), `plg2` (a `cut` as the second condition), and `plg3` (a `cut` as the third condition).

```
language(english).
language(scheme).
language(prolog).
computer(scheme).
computer(prolog).
plg0(P) :- language(P), computer(P).     // no cut in the rule
plg1(P) :- !, language(P), computer(P). // a cut as the first condition
plg2(P) :- language(P), !, computer(P). // a cut as the second condition
plg3(P) :- language(P), computer(P), !. // a cut as the third condition
```

Figure 5.7 illustrates the search process when the question `?- plg0(P)` is asked. The first backtrack point is created when a match is found for the first condition "`language(X)`" and the search proceeds to matching the second condition. After the second condition fails, the research resumes from the backtrack point ①.

Thus, for the question `?- plg0(X)`, where no `cut` is applied in the rule `plg0(X)`, the question will find two answers if a semicolon is typed after the first answer:

```
X = scheme
X = prolog
```

For the question `?- plg1(X)`, where a `cut` is put as the first condition of the rule `plg1(X)`, the `cut` is executed before any backtrack points are created. Thus, no backtrack points will be removed and the answer to this question is the same as the answer to the question `?- plg0(X)`.

For the question `?- plg2(X)`, the `cut` will be executed after backtrack point ① is created and before an answer is found. The execution of `cut` then removes the backtrack point ①. As the result, after the search failed (see the bottom right of Figure 5.7), there is no backtrack point to return to and, thus, not a single answer is found for the question.

Figure 5.7. Adding backtrack points.

For the question `?- plg3(X)`, the `cut` will be executed after backtrack point ① is created and after the first answer `X = scheme` is found. The execution will remove the backtrack point ①. As a result, the second answer will not be found, even if a semicolon is typed.

Not all the search-returning points in the Prolog database are backtracking points that can be removed by `cut`. There are other kinds of search-returning points: **composite condition returning points** and **recursive exit points**. These two kinds of returning points may not be removed by the `cut`.

For example, there are two conditions in the rule `plg3(P)` in the programming language example above. While searching for a match for the first condition, a search-returning point must be marked so that the search can continue from this point to find a match for the second condition after a match is found for the first condition. Obviously, this search-returning point may not be removed from a semantics point of view. If such a search-returning point could be removed, a solution that only meets the first condition would be accepted as a "true" solution of the rule, which is incorrect.

For a recursive rule, there is a similar situation where a searching returning point must be marked. As shown in Figure 5.8, when a recursive call is made, the execution reenters the rule, leaving the condition behind the recursive call not searched and, thus, a research-returning point must be added every time a recursive

call is made. Similar to the composite condition returning point, the recursive exit point may not be removed by the `cut`.

```
factorial(0, 1) :-
                !.
factorial(N, F) :-
                N> 0,
                N1 is N- 1,
                factorial(N1, F1),
                /* Recursive exit point */
                F is N * F1.
```

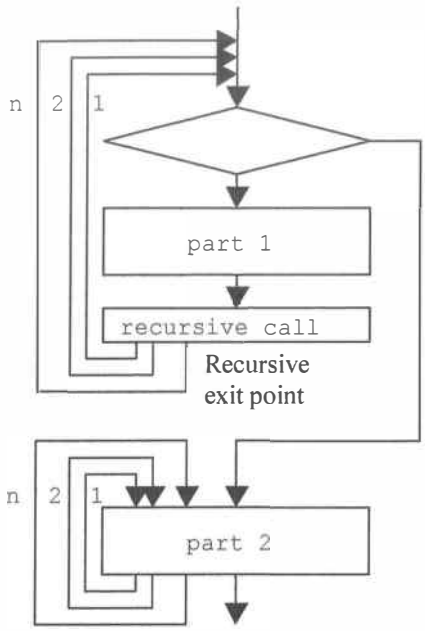


Figure 5.8. Recursive rule and recursive exit point.

5.6.2 Fail

The built-in function **fail** simply returns a false value. In Prolog runtime, if a true value is returned, meaning that a solution is found, the search will pause, waiting for a manual input of a semicolon to continue searching, or an enter for stopping. If you want to ask the runtime to continue searching, you can add a `fail` clause after the clause where a true value can return. We will use the definition of the not rules to illustrate the use of `fail`.

```
not(X) :- X, !, fail.    % what happen if ! is not used?
not(_).
```

As the name suggests, the rule `not(X)` should return false if `X` is true, and return true if `X` is false. In the definition, if `X` is true, the `cut` (!) clause will be executed, which returns true too, and then the `fail` clause will be executed, which will fail. Why do we need the `cut` before the `fail`? With the `cut`, the search will continue after a failed clause. The `cut` will remove all the backtracking points, so that the search will not continue after a failed clause.

On the other hand, if `X` is false, the `cut` and the `fail` will not be executed, and thus, the next `not(X)` clause will be executed. As it has no condition, it always succeeds and returns true. An anonymous variable is used in the second not rule to prevent a "singleton" variable warning.

Now we use an example to illustrate the use of the not rules, where the not rule is highlighted.

```
% facts
female(elaine).
female(jane).
female(sarah).
male(conrad).
```

```
male(joe).
married(luke).
married(mike).
father_of(andrew, conrad).
father_of(luke, mike).
father_of(mike, andrew).
mother_of(elaine, sarah).
mother_of(jane, edith).
mother_of(jane, mike).
% Rules
not(P) :- P, !, fail.
not(_).
is_male(X) :- father_of(X, _); male(X).
is_female(X) :- mother_of(X, _); female(X).
bachelor(X) :- is_male(X), not(married(X)).
grandmother_of(X, Z) :- mother_of(X, Y),
  (mother_of(Y, Z); father_of(Y, Z)).
familyquestions :-
  grandmother_of(X, andrew),
  write('The grandmother of andrew is '),
  write(X), nl,
  father_of(Y, mike),
  write(Y), write(' is the father of mike'), nl,
      bachelor(Z), write(Z), write(' is a bachelor'),nl.
```

Using the family question as the goal ?- familyquestions, the following output is generated:

```
?- familyquestions.
The grandmother of andrew is jane
luke is the father of mike
andrew is a bachelor
true ? ;
conrad is a bachelor
true ? ;
joe is a bachelor
```

5.6.3 Repeat

Now we examine the next control structure **repeat** that always succeeds and always adds a backtrack whenever it is executed. Figure 5.9 shows a simple application of `repeat` that creates an infinite loop: When the `repeat` is executed, a backtrack point is created and `repeat` succeeds. Then, the control enters the body of the loop to "do something." At the end, the built-in predicate `fail` is executed. The `fail` predicate does nothing but `fail` (return false). Since the rule fails, it automatically returns to the latest backtrack point. The backtrack point is removed when the control returns to it. However, when the `repeat` is executed, a new backtrack point is created. The loop thus repeats forever.

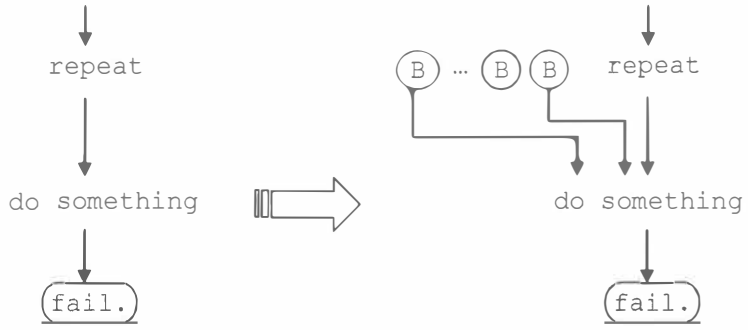


Figure 5.9. The `repeat` adds a backtrack point every time it is executed.

The following program gives an example of this application. The program repeatedly gets a printable character from the keyboard and writes the character on the screen.

```
get_forever :-
      repeat,          /*  add a backtracking point */
      get(X),          /*  enter a printable char */
      write(X),
      nl,
      fail.            /*  to return to last backtracking point.  */
```

Without the `fail` predicate at the end, the rule will succeed and not automatically return to `repeat`. It returns to the `repeat` only after a semicolon is entered.

Now examine another application of `repeat` that creates a loop and the loop exits when a certain condition is met.

```
get_digit(X) :-
      repeat,              /* add a backtracking point */
      write('enter a digit'),
      nl,
      get0(X1),            /* get any character */
      X1 > 47,             /* will fail if not digit */
      X1 < 58,
      X is X1 - 48,
      !.                 /* remove backtracking points */
```

When the rule `get_digit(X)` is executed, the first condition is a `repeat`, which succeeds and adds a backtrack point. Predicates `write` and `nl` always succeed. Then a character is entered from the keyboard and instantiated to `X1`. If `X1` is a digit, its ASCII value must be between 48 and 57. If these two boundary conditions are met, the ASCII value is converted to the value it represents. Then the `cut` is executed, which removes the backtrack point created by `repeat`. Thus, the loop exits. On the other hand, if one of the boundary conditions is false (i.e., a nondigit character is entered), the composite condition fails and it returns to the backtrack point and another iteration of the loop starts.

*5.7 Prolog application in semantic Web

The web was originally built for human users to retrieve information. Although a large part of the web is machine-readable, the data are not machine-understandable. Because of the volume of information on the web today, it is no longer possible to manage and retrieve information manually. The solution is to use metadata to describe the data contained on the web.

As defined by W3C (WWW Consortium), the **semantic web** is a vision for the future of web information, where the information available on the web is given explicit meaning to better support automatic processing and integration of web information. If we consider that the current web is a decentralized platform for distributed presentations, the semantic web is a decentralized platform for distributed knowledge. Ontology is the key technology for implementing the semantic web.

The word *ontology* comes from the field of philosophy, where it means a systematic explanation of being. In computer science, **ontology** is a formal specification of the terms/objects in a domain and the relationships among them. Technically, an ontology defines a common vocabulary of elements in which the meanings of the elements are described in terms of their relations, properties, and attributes (e.g., synonym and antonym), and the structure of the statements using the vocabulary.

A shared representation is essential for the common understanding of data and communication. Ontology defines the structure of knowledge representation and conceptualization of a domain. It describes domain knowledge in a generic way and provides an agreed-upon understanding of a domain.

Ontology includes machine-understandable definitions of basic concepts in a specific domain for the purposes of

- sharing a common understanding of the structure of information among people or software agents that use the information;
- making domain assumptions explicit, and enabling reuse of domain knowledge;
- separating domain knowledge from operational knowledge;
- enabling domain knowledge analysis and reasoning.

The languages currently used for implementing ontologies include **RDF** (Resource Description Framework), **OWL** (Web Ontology Language), and Prolog. RDF and OWL have the capacity to represent knowledge, but they do not have the computing power to process the knowledge. Prolog has the capacity for representing knowledge as well as for processing knowledge.

Prolog is a well-established high-level logic and declarative programming language. It is designed for artificial intelligence and is a natural choice for implementing ontologies (semantic web). A Prolog program has a built-in database and the capacity for data processing and reasoning. The statements in RDF and in OWL can be easily translated into Prolog statements. Prolog has been used to implement RDF and OWL parsers, for example,

- **SWI-Prolog RDF parser:** http://www.swi-prolog.org/
- **Thea:** A Web Ontology Language - OWL Parser for [SWI] Prolog http://www.semanticweb.gr/TheaOWLParser/

Figure 5.10 shows an example of representing knowledge in RDF. Its XML representation is given below, which is a list of triples.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:cs="http://venus.eas.asu.edu/WSRepository/xml/Courses.rdf">
```

```
<rdf:Description rdf:about="professors">
<cs:subsume rdf:resource= "faculty"/>
<cs:includes rdf:resource= "Mary"/>
  </rdf:Description>

<rdf:Description rdf:about="SOC">
      <cs:taughtOn>Mon and Wed</cs:taughtOn>
      <cs:taughtIn>Room220</cs:taughtIn>
<cs:member rdf:resource= "seniorCourses"/>
  </rdf:Description>
<rdf:Description rdf:about="seniorCourses">
<cs:subsume rdf:resource= "courses"/>
  </rdf:Description>
  ...
</rdf:RDF>
```

As can be seen in Figure 5.10, the RDF representation can be easily represented in Prolog facts. The processing of the data can be written in Prolog rules.

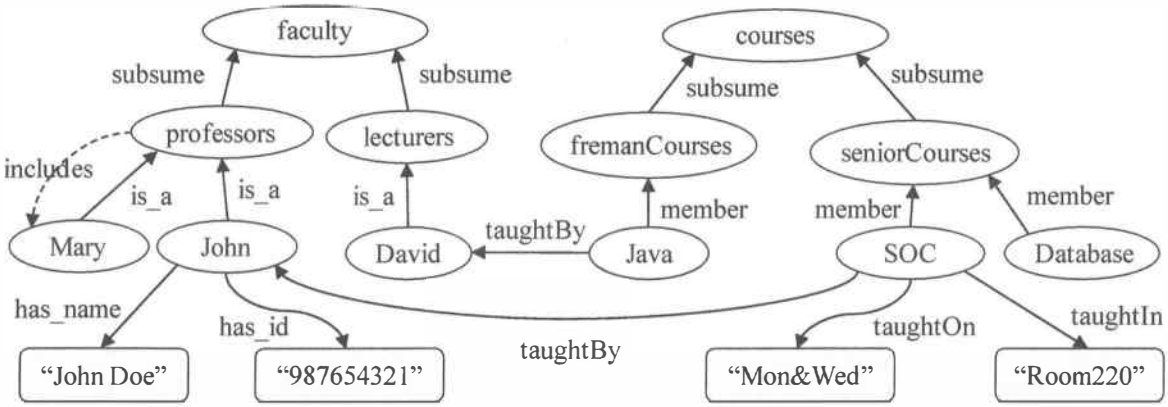


Figure 5.10. Graphic representation of RDF statements.

5.8 Summary

This chapter described the fundamental concepts of logic programming and basic programming techniques. It covered:

- The basic concepts and execution model of Prolog is described in a C-like pseudo language, which helps in understanding the Prolog rules and the problem-solving process.
- The structure of Prolog programs. At the lexical level: The composition of Prolog variables; at the syntactic level: The syntax of constructing facts, rules, and goals; at the level of the contextual structure: The instantiation and scope of variables; and at the semantic level, there are two different programming models: The database-base unification model and the arithmetic model.
- **Recursive rules and application of recursion:** The fantastic-four abstract approach discussed in the C and Scheme chapters can be applied to Prolog recursive programming.

- **Lists and list operations:** This section shares much common ground with the list operations in the Scheme chapter. Prolog lists and Scheme lists are compared and the pair type defined in Scheme is used to explain the different list representations used in Prolog.
- The flow control structures of Prolog are explained in detail, including the application of cut, fail, and repeat.
- Finally, Prolog application in the semantic web.

Logic programming language is becoming more and more important in the context of ontology, semantic web, and Big Data analysis.

5.9 Homework, programming exercises, and projects

1. Multiple Choice. Choose only one answer for each question. Choose the best answer if more than one answer is acceptable.

1.1 Which of the following programming languages most closely follows the stored program concept?

❑ C ❑ Lisp ❑ Scheme ❑ Prolog

1.2 Prolog is based on

❑ Lambda calculus ❑ Predicate logic ❑ Turing machine ❑ Boolean logic

1.3 There are three kinds of clauses in a Prolog program: facts, rules, and goals.

❑ A fact can be considered a special case of a rule.
❑ A fact can be considered a special case of a goal.
❑ A rule can be considered a special case of a fact.
❑ A rule can be considered a special case of a goal.

1.4 What mechanism cannot be used for passing values between clauses within a Prolog rule?

❑ Call-by-value ❑ Call-by-alias ❑ Return value ❑ All of them

1.5 If you want to pass multiple values out of a Prolog rule, which of the following methods is valid?

❑ Use multiple return statements in the predicate.
❑ Use a single return statement to return a list that contains the multiple values required.
❑ Use multiple named variables to hold the values.
❑ Use multiple unnamed variables to hold the values.

1.6 A goal clause and a fact unify, if

❑ their predicates are the same. ❑ their arities are the same.
❑ their corresponding arguments match. ❑ all of the above are true.

1.7 The arity of a predicate is

❑ the head of the predicate. ❑ the neck of the predicate.
❑ the body of the predicate. ❑ the number of arguments of the predicate.

1.8 The scope of a Prolog variable is within

❑ a single rule. ❑ a single clause in a rule.
❑ the fact/rule base. ❑ the body of the next rule.

1.9 A circular definition of a Prolog rule

❑ is an imperative feature that should be discouraged.
❑ will cause a dead loop for every goal.
❑ will cause a dead loop when no match can be found.
❑ will never cause a dead loop.

1.10 An anonymous variable in Prolog is a

❑ constant. ❑ placeholder. ❑ predicate. ❑ question.

1.11 What is the output when the following Prolog goal is executed?

```
?-member(apple, [orange, apple, pear]).
```

❑ true? ❑ X = apple ❑ [apple, pear] ❑None of them

1.12 Assume we have the following fact in a Prolog factbase:

```
child_of(mary, [amy, david, conrad]).
```

What is the output when the following Prolog goal is executed?

```
?- child_of(mary, [amy | T]).
```

❑ T = [amy, david, conrad] ❑ T = [david, conrad]

❑ T = david, conrad ❑ T = conrad

1.13 Assume that we have the following fact in a Prolog factbase:

```
child_of(mary, [amy, david, conrad]).
```

What is the output when the following goal is executed?

```
?- child_of(mary, [amy | [H | [conrad]]]).
```

❑ H = [amy] ❑ H = [david] ❑ H = david ❑ H = [amy, conrad]

1.14 What is the output when the following Prolog goal is executed?

```
?- member(X,[81, 25, 9, 29]), Y is X*X, Y<400.
```

❑ X = [81, 25, 9, 29] ❑ X = 9 Y =29 ❑ X = 9 Y = 81 ❑ X = 81 Y =400

1.15 Given the following recursive rules in Prolog:

```
foo(X, [X]).
foo(X, [ _ | T]) :- foo(X, T).
```

The rules find

❑ the last element of a list. ❑ the length of a list.

❑ whether an element is a member of a list. ❑ the sum of all members in a list.

1.16 When Prolog is searching for possible matches and a cut "!" is encountered,

❑ all existing backtrack points will be removed.

❑ all existing recursive exit points will be removed.

❑ all existing backtrack points and recursive exit points will be removed.

❑ none of the existing backtrack points and recursive exit points will be removed.

1.17 What does the built-in predicate **cut** (!) do?

❑ Stop searching immediately.

❑ Remove all existing backtracking points.

- ❑ Stop search after the first match is found.
- ❑ Remove all backtracking and recursive exit points.

1.18 What does the built-in predicate **fail** do?

- ❑ Jump to the previous repeat predicate.
- ❑ Remove one backtracking point.
- ❑ Return false.
- ❑ Stop searching immediately.

1.19 The built-in Prolog predicate `repeat` always

- ❑ fails immediately.
- ❑ succeeds and adds a backtracking point.
- ❑ fails at the end of the database.
- ❑ fails when it is visited (executed) for the second time.

1.20 What does the following code do?

```
go :- repeat, get(X), write(X), nl, fail.
```

- ❑ It takes a single character from the keyboard and prints it.
- ❑ It repeatedly takes a character from the keyboard and prints it.
- ❑ It takes a character from the keyboard, prints it, and exits if the character is a digit.
- ❑ It takes a character from the keyboard, prints it, and exits if the character is NOT printable.

1.21 What is the key difference between the semantic web and the syntactic web?

- ❑ Semantic web performs semantic check of pages submitted to web.
- ❑ Semantic web enables keyword-based search.
- ❑ Semantic web better supports automatic processing and integration of web information.
- ❑ Semantic web is based on Prolog.

1.22 How is Prolog related to the semantic web?

- ❑ Prolog is frequently used to describe the semantic web as a markup language.
- ❑ Prolog is frequently used for writing the parsers of the semantic description language.
- ❑ Prolog is a service-oriented programming language.
- ❑ All Prolog programs are semantic web.

2. There are several data passing mechanisms between the calling function (caller) and the called function (callee): call-by-values, call-by-alias, return value, and global variable (global name). What data passing mechanisms are supported by imperative C, functional Scheme, and logic Prolog? Draw a table to summarize the supported mechanisms by these three languages.

3. What are named and unnamed (anonymous) variables in Prolog? What are their differences?

4. What is the scope rule of Prolog?

5. What are the differences between the variables in imperative, functional, and logic programming languages?

6. How is a Prolog program executed? What is the definition of a match?

7. What is a circular definition? What is the consequence of a circular definition? In a family database, will a problem occur if we define the following rules? Put these rules in a family database and test them.

```
sister(X, Y) :- sister(Y, X).
brother(X, Y) :- brother(Y, X).
parent_of(X, Y) :- child_of(Y, X).
child_of(X, Y) :- parent_of (Y, X).
```

8. What is the difference between a circularly defined rule and a properly defined recursive rule? Can a recursive rule cause a dead loop? How can you avoid a dead loop in the definition of recursive rules?

9. Does the order of the clauses in a rule matter? Does the order of the facts and rules in a fact/rule base matter?

10. How do you design a recursive rule? Is the following design process correct?

(1) Formulate the size-N problem: Choose the predicate name and appropriate arguments for holding the return value and the input values to the rule.

(2) Design the stopping condition (normally, N = 0 or N = 1) and its solution.

(3) Assume that you have already found the solution of the same problem with size N1, where N1 < N.

(4) Use the solution of the size-N1 problem to define the solution of the size-N problem.

(5) Verify the above design process using the "Hanoi Towers" problem discussed in Chapter 4 as an example.

11. Using BNF notation, a Prolog list can be recursively defined as follows:

```
<list> ::= [], where [] is an empty list
<list> ::= [<X> | <Y>], where X is a variable or value, and Y is a list.
```

In the definition, `[<X> | <Y>]` can be considered to be a pair as defined in Scheme in Chapter 4.

11.1 Describe simplification rules that can be used to simplify the Prolog lists.

11.2 Apply the simplification rules to simplify `[1 | [2 | [3 | []]]]` into `[1 2 3]`.

11.3 Is this structure `[[1 | 2] | [3 | [4 | 5]]]` a valid Prolog list? Explain your answer.

12. Define a rule to return the common members of two lists. Trace the execution manually and by the trace routine. Compare the two traces. The format of the rule is

```
common(X, List1, List2)
```

13. Define a rule to return the change (number of quarters, dimes, nickels, and pennies) of a given amount *S*, where 0 <= *S*<= 100. The format of the rule is

```
change(S, Q, D, N, P)
```

14. You are given the following Prolog factbase. The `familyquestions` rule is, in fact, a compound question. It will cause a number of goals (questions) to be called. You can consider the question as the "main" program that addresses the problem you want to solve. However, since the individual questions are connected by an "and" relationship, the compound question will stop if a "no" answer is given to any individual question. You could use the "or" relationship to connect some questions. In this case, the compound question will stop if a "yes" answer is given to any individual question.

```
/* Factbase for family. It consists of facts and rules. */

/* Facts */
male(luke).
male(mike).
female(sarah).
mother_of(jane, elaine).
mother_of(jane, mike).
father_of(mike, andrew).
father_of(andrew, conrad).
father_of(luke, mike).

/* Rules */
grandmother_of(X, Z) :-
        mother_of(X, Y),
        (mother_of(Y, Z); father_of(Y, Z)).

familyquestions :-
    grandmother_of(X, andrew),
    write('The grandmother of Andrew is '), write(X), nl,
    father_of(Y, mike),
    write(Y), write(' is the father of mike'), nl, nl.
```

14.1 Enter the program using a text editor under the Unix operation system and save the file as `family.pl`. You can enter the program on your PC and upload the program into the Unix server.

You may also enter the program on your PC and upload the program into your prolog directory in the Unix server.

14.2 Compile the program using the Prolog command:

```
> gplc family.pl
```

14.3 Enter GNU Prolog by executing the Unix command `gprolog`.

14.4 Execute the program `family` by typing GNU Prolog command

```
|?- [family].
```

14.5 Ask a few questions, for example,

```
|?- male(luke).
|?- male(X).
|?- grandmother_of(jane, mike).
```

```
|?- grandmother_of(jane, X).
|?- grandmother_of(X, mike).
|?- familyquestions.  /* This will call all questions in the rule. */
```

14.6 Change the "and" operator, "," after the first "`nl`" statement (newline) to the "or" operator ";" in the `familyquestions` program and observe what differences are made in the output.

14.7 Manually trace the execution of the following goal. List all statements in their execution order involved in the trace.

```
|?- grandmother_of(jane, andrew).
```

14.8 Add the following set of rules that extend the relationships among the members: brother, sister, grandfather, grandparent, greatgrandfather.

14.9 Add 30 facts into the factbase showing the relationships between family members. These facts must ensure each rule defined on the facts can return a yes value for at least one set of parameters.

14.10 Compile the extended program and ask at least 10 different questions using GNU Prolog commands, also use variables in these questions.

14.11 Add a rule called `morequestions` at the end of factbase to include the 10 questions that you have tested. Include sufficient write statements so that the answers to individual questions are printed. Make sure that the compound question can terminate (will not cause a "dead loop").

14.12 Recompile the program and call the following goal.

```
|?- morequestions.
```

15. Define recursive rules to compute mathematical functions.

15.1 Compute `addexp(X, Y, N) = (X + Y)^N`, where `X`, `Y`, and `N` are nonnegative integers, and `X`, `Y` and `N` cannot be 0 at the same time, because 0^0 is undefined. You must add an argument to store the return value.

15.2 Test your program using different input sets (test cases) and verify (e.g., using your calculator) the correctness of the answer. Give five sets of inputs-outputs that you tested.

15.3 Compute `fe(X, Y, N) = ((X + Y)^N)!`. The function must call the function that you designed in question 15.1. You must include all functions that are used.

15.4 Test your program in question 15.3 using different input sets (test cases) and verify (e.g., using your calculator) the correctness of the answer. Give two sets of inputs-outputs that you tested.

16. The Ackermann function is defined recursively for nonnegative integers m and n as follows:

$$A(s, t) = \begin{cases} t + 1, & if\ s = 0 \\ A(s - 1, 1), & if\ s > 0\ and\ t = 0 \\ A\big(s - 1, A(s, t - 1)\big), & if\ s > 0\ and\ t > 0 \end{cases}$$

Note: The Ackerman function is a very rapidly growing function. Even values of 4 for m and n will yield an extremely large number. Thus, use small values of m and n, like 1, 2, or 3, when you test your program.

Follow the fantastic-four abstract approach to implement the function in Prolog rules.

16.1 Define the size-n problem.

16.2 Define the stopping conditions and the return values.

16.3 Define the size-m problems.

16.4 Construct the size-n solution from the size-m solutions.

16.5 Give the complete rules that can be used for computing the Ackermann function for any given integer $N \geq 0$.

17. You are asked to create a family tree according to the given tree structure in Figure 5.11. In the family tree, the horizontal edges represent the spouse relationship. The left one is female and the right one is male. The top-down edges represent the parent-child relationship.

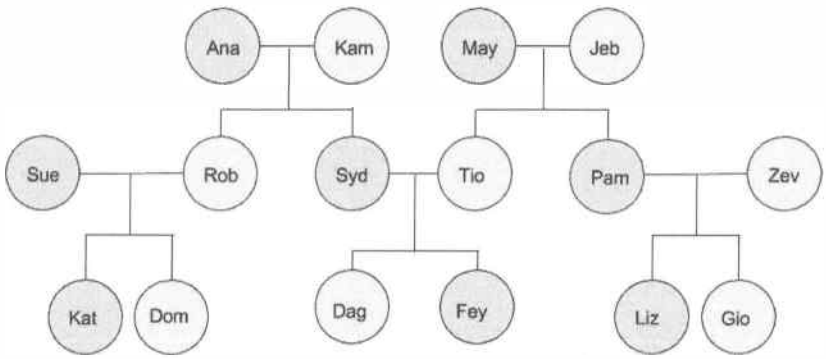


Figure 5.11. The structure of a family tree.

17.1 Define the factbase according to the given family tree structure by using the facts *father_of*(*a*, *b*) and *mother_of*(*a*, *b*) You can add more facts, and the names added in the tree must be different.

17.2 Define a rule `sibling(X, Y)` which returns `yes` if `X` and `Y` have the same parent, where `X` and `Y` may not be the same (e.g., `sibling(mike, mike)` must return `no`).

17.3 Define a recursive rule `depth(D, X)` where `D` is the number of levels from the root to `X`. The depth of the root is 0.

17.4 Based on the depth relationship, define a rule `cousin1(X, Y)` which returns "yes" if `X` and `Y` have the same depth. Note, this cousin relationship is slightly different from what the word cousin means. In this definition, siblings are cousins too.

17.5 Define a recursive rule `cousin2(X, Y)` without using the depth relationship. Hint: Use the parent rule in the question.

17.6 Define a rule called `treequestions` that will call all rules that you have defined once. Make sure that all calls in the rule `treequestions` will be executed.

Chapter 6

Fundamentals of the Service-Oriented Computing Paradigm

Having discussed four major programming paradigms, this chapter introduces the fundamentals of the emerging service-oriented computing (SOC) paradigm, including the basic concepts, principles, programming, and the software development processes based on this paradigm. As a preparation, we first give a short introduction to C#. As C# is closely related to C++ and Java, it should be straightforward to learn this language. The focus of the chapter is not on the language, but on the paradigm that suggests a new way of software development.

6.1 Introduction to C#

C# is an object-oriented programming language, and it is one of the major languages that is used for writing web services (WS) today. In fact, the constituent services (components) of a service-oriented application are object oriented. The object-oriented classes are wrapped with open standard interfaces to become services. A service-oriented application is composed of services from different service providers.

6.1.1 Getting started with C# and Visual Studio

To get started with C# and Visual Studio, enter the following program that prints the string "Hello, World!":

```
using System;
public class MyFirstClass {
   public static void Main() {
      Console.WriteLine("Hello, World!");
} }
```

To execute this C# program on Visual Studio, you can follow the following simple steps:

1. Start Visual Studio from the Windows "Start" menu.
2. Choose Visual Studio menu "File - New - Project...": A "New Project" dialog box will pop up, in which you can choose different languages, including C/C++, J# (Java), and C#.
3. Once you have selected C# in the box on the left-hand side, you can further choose a template to facilitate the application you want to develop. For example:
    a. Choose "Console Application" to start a text and command line-based programming template.

b. Choose "Windows Application" to start a forms-based application template, which allows you to define graphic user interfaces.

4. At the bottom of the same dialog box, you can choose a Name for your project, choose a Location (directory) where you want to save your project, and choose a Solution Name. You can put multiple projects in the same Solution.
5. Click OK.

A project template with appropriate libraries (depending on the template that you select) will be created. If you have selected C#, you can type your C# program in the file with the extension .cs that is created when you start a new project.

The following program shows a more complex example. The program manages the scores/weights of a weightlifting competition. The user is asked to enter the names and weights lifted by four players, and the program prints the winner's name and weight.

```
using System;
class Tournament {
   static void FindWinner(string[] N, Int32[] W) {
      if (W[0] > W[1])
          if (W[0] > W[2])
              if (W[0] > W[3])
                  Console.WriteLine(N[0]+" wins with weight = "+W[0]);
              else
                  Console.WriteLine(N[3]+" wins with weight = "+W[3]);
          else
              if (W[2] > W[3])
                  Console.WriteLine(N[2]+" wins with weight = "+W[2]);
              else
                  Console.WriteLine(N[3]+" wins with weight = "+W[3]);
      else
          if (W[1] > W[2])
              if (W[1] > W[3])
                  Console.WriteLine(N[1]+" wins with weight = "+W[1]);
              else
                  Console.WriteLine(N[3]+" wins with weight = "+W[3]);
          else
              if (W[2] > W[3])
                  Console.WriteLine(N[2]+" wins with weight = "+W[2]);
              else
                  Console.WriteLine(N[3]+" wins with weight = "+W[3]);
   }
   static void Main() {            // Declare variables (memory spaces)
      Int32 i, N = 4;   // N is the number of players
      Int32[] Weights;   // Declare a reference to an array of int
      string Num, Weight;
      string[] Names;   // Declare a reference to an array of string
      Names = new string[N]; // Create an array object
      Weights = new Int32[N];    // Create an array object
```

```
        for (i=0; i<N; i++) {
            Console.WriteLine("Enter the name and weight of Player {0}",
i);
            Names[i] = Console.ReadLine(); // read the name
            Weight = Console.ReadLine();   // read the weight in a string
            Weights[i] = Convert.ToInt32(Weight); // Convert string to int
        }
        FindWinner(Names, Weights);
    }
}
```

This program illustrates many important issues of a typical program, including input; output; converting string to integer; declaring integer, string, reference to array of integer, and reference to array of string; creating an object of array; loop, defining static function; and parameter passing and function call. The comments in the program give more details of explanation.

C# inherits most of its syntax from the C/C++ family of languages and supports most features that Java supports. It is strongly typed with automatic garbage collection and a rich functionality set that empowers developers of both object- and service-oriented software. This section introduces, from a C++ programmer's point of view, a small subset of this extensive language.

6.1.2 Comparison between C++ and C#

Table 6.1 compares and contrasts the main features of C++ and C#. As can be seen, C# moves toward automatic management like Java, while trying to keep the C++ features where possible.

6.1.3 Namespaces and the using directives

A namespace is used to group a set of classes, and a "using namespace" is also used to quote the classes in the namespace as library functions to be used in a program. The following code segment shows the very basic code template for a C# program:

```
using <namespace>        // using existing namespace as library
namespace myNamespace1   // define my own namespace
class myclass1 {
  public static void Main() {
  ...
  }
class myclass2 {
  public double PiValue() {
  ...
  }
}
```

Header files in C/C++ do not exist in C#. Instead, namespaces are used to reference groups of libraries and classes. Programmers can define namespaces in order to prevent class naming conflicts, as well as to reference namespaces, which define the .Net Framework SDK. For example:

```
namespace VirtualStore {
  namespace Customer {
     //define customer classes
     class ShoppingCartOrder( ) { ... }
```

```
    }
    namespace Admin {
        //define administration classes
        class ReportGenerator( ) { ... }
    }
}
```

Feature	C++	C#
main()	Global function	public static function in a class
Use of library functions	Header files (#include directives) and the **using** directive can be used.	Header files may not be used. The **using** directive is used to reference types in other namespaces.
Preprocessor directives	Preprocessor directives and macros are allowed.	Preprocessor directives are allowed, but cannot create macros. Directives can be used for conditional compilation.
Global functions or variables	Allowed	Not allowed. They must be contained within a type declaration (such as class or struct).
Inheritance	Multiple inheritance is allowed.	A class can inherit implementation from one base class only. However, a class or an interface can implement multiple interfaces.
Override	Declaring override functions does not require the **override** keyword.	Declaring override methods requires the **override** keyword.
Garbage and destructor	No automatic garbage collector. Destructors are called automatically, but a programmer can call destructors.	There is an automatic garbage collector. Programmer cannot call the destructors.
long type	32 bits	64 bits
Array declaration	The brackets "[]" appear following the array variable, e.g., `int myArray[] = {1, 2, 7};`	The brackets "[]" appear following the array type, e.g., int[] myArray = {1, 2, 7};
String	An array of characters with a terminator.	A string type is defined. One can use == and != to compare two string objects.
Pointer	Allowed	Pointers are allowed only in unsafe mode.
switch statement	Supports fall through from one case label to another. Use **break** to exit.	Does not support fall through from one case label to another.
foreach statement	Not allowed	Used to iterate through arrays and collections.

Table 6.1. Comparing and contrasting C++ and C# features.

The `using` directive tells the compiler where to search for definitions for namespace class's member methods that are used in the application. For example:

```
using VirtualStore;
using System.Windows.Forms;
```

An alternative to `using` directives is to fully qualify each single reference like this:

```
private System.Windows.Forms.Button Button1;
```

6.1.4 The queue example in C#

To see concrete differences between C++ and C# and to get started with writing a C# program, we present the C# version of the Queue example given in Section 3.1. From this example, you can see that it is easy to get started with C# after you have learned C++.

```
using System;
namespace ConsoleApplication1
{ // class is defined within curly bracket
  class Queue {
      private int[] buffer;
      private int queue_size;
      protected int front;
      protected int rear;
      public Queue() {  // constructor
          front = 0; rear = 0; queue_size = 10;
          buffer = new int[queue_size];
      }
      public Queue(int n) {  // constructor
          front = 0; rear = 0; queue_size = n;
          buffer = new int[queue_size];
      }
      public void enqueue(int v) {
          if(rear < queue_size) buffer[rear++] = v;
          else if(compact()) buffer[rear++] = v;
      }
      public int dequeue() {
          if(front < rear) return buffer[front++];
          else {
              Console.WriteLine("Error: Queue empty");
              return -1;
          }
      }
      private bool compact() {
          if(front == 0) {
              Console.WriteLine("Error: Queue overflow");
              return false;
          }
          else {
              for(int i = 0;i < rear-front; i++)
                  buffer[i] = buffer[i+front];
              rear = rear - front; front = 0;
```

```
                return true;
            }
        }
        static void Main( ) {
            Queue Q2 = new Queue(4);
            Q2.enqueue(12); Q2.enqueue(18);
            int x = Q2.dequeue(); int y = Q2.dequeue();
            Console.WriteLine("X = {0}  Y = {1} ", x, y);
            Console.ReadLine();
        }
    }
}
```

If you compare this program with the C++ program in Section 3.1, you can see that the two programs are very similar and C# is not difficult to learn.

6.1.5 Class and object in C#

Like Java, all C# applications require a unique program entry point, or `Main` method, implemented as a member method within a class. This differs from C++, where `Main` is a function that must be located outside any class. In C# programs, `Main`'s location is determined by the compiler, and it does not matter which class defines the `Main`. `Main` is required to be defined as `static`, and may optionally receive arguments or return a value. An optional `public` access modifier notifies the C# compiler that anyone can call this member method. The required `static` keyword means that `Main` is called without requiring an object instance.

A class is a user-defined type and a blueprint of functionality for variables of that type. An object is a named reference to that class with memory allocated. Instantiating a class with the `new` function creates an instance or an object with information about member methods and other members allocated on the heap. For both C++ and C#, a variable of a reference type takes values of the heap addresses required to locate those class members. Class members include anything defined inside a class, such as variables, constants, and functions.

In C#, accessing the members of a static object is accomplished, like Java, using class name and the "." dot operator

```
<className>.<memberName>
Console.WriteLine("Hello World!");
```

where `memberName` is a method call or variable name, respectively.

Accessing the members of a reference object is also accomplished like Java, with the "." dot operator

```
<referenceName>.<memberName>
time.printStandard( );
```

where `memberName` is a method call or variable name, respectively.

C++ offers a second way to define objects, not as reference types on the heap, but as local types on the stack. This is done by simply instantiating without the `new` function. In C#, the `new` keyword is the only way to create an object instance.

The class syntax can be described using the following syntax diagram, where the item in square brackets is optional.

```
[attributes][modifiers] class <className> [: baseClassName]
{
  [class-body]
}[;]
```

Attributes can be thought of as inline notes and declarative statements that the programmer can attach to a class, members, parameters, or other code elements. Through a library called reflection, this extra information can be retrieved and used by other codes at run time. Attributes provide a generic means of associating information with declarations, a powerful tool in numerous scenarios.

The access **modifiers** public, `protected`, and `private` have equivalent semantics in C# and C++. Both C# and C++ will default to `private` if no access modifier is explicitly defined.

Other possible method modifiers include `sealed`, `override`, and `virtual`, as well as the class modifier `abstract` that deals with class inheritance functionality and scope.

In C++, programmers have the choice of defining class members inside the class declaration, or outside the class declaration with the use of the scope-resolution operator. In C#, programmers must define all class members inside the curly brackets of that class. The simple idea of grouping related objects inside the same class is designed to create more modular bundles of code.

A key feature of object-oriented programming is to decompose the application into multiple classes. One of the classes will contain the Main() method, while other classes will contain reusable members and methods.

Let us consider a program that helps a person to prepare for travel, including the computation of the amount of U.S. dollars and local currency needed and local temperature converted into Fahrenheit. The program is decomposed into four classes, as shown in Figure 6.1.

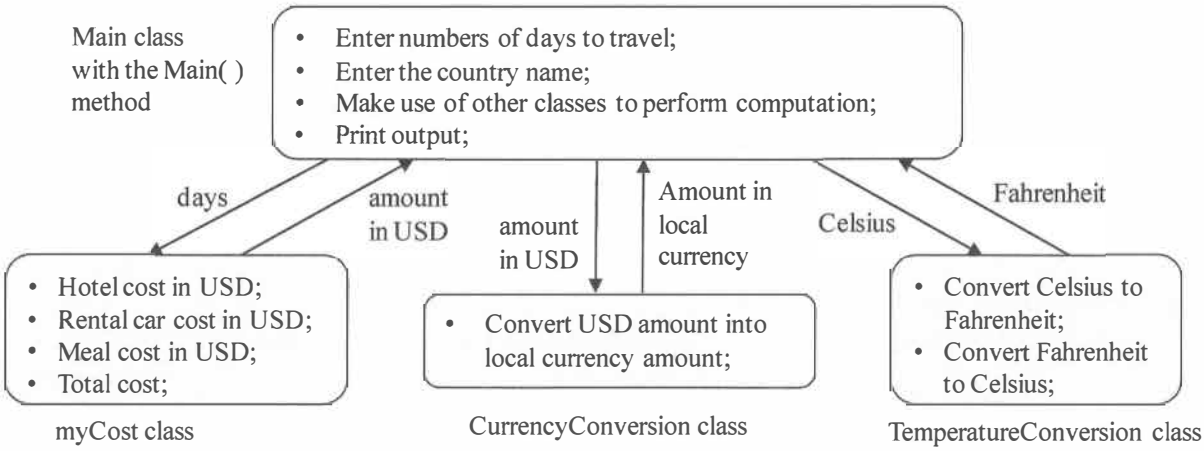


Figure 6.1. Problem decomposition into multiple classes.

The sample code implementing the travel preparation is given as follows. A constructor with a parameter is given in the class myCost. Since the parameter is used by multiple member functions in the class, it is more productive to pass the parameter through the constructor. The parameter value will be passed to the object when the object is instantiated by the new function. On the other hand, we do not need to have a constructor for the other two classes: CurrencyConversion and TemperatureConversion. In these classes, the parameters are used by one member function only, and thus, we can pass the parameters directly to the member functions, instead of creating data members to hold the parameter values.

```
using System;
```

```
class TravelPreparation {     // Main Class
  static void Main(string[] args) {  // The main method
    Console.WriteLine("Please enter the number of days you will travel");
    String str = Console.ReadLine();     // read a string of characters
    Int32 daysToStay= Convert.ToInt32(str); // Convert string to integer
    myCost usdObject = new myCost(daysToStay);    // Create an object
    int usdCash = usdObject.total(); // Call a method in the object
    Console.WriteLine("Please enter the country name you will travel to");
    String country = Console.ReadLine();
    CurrencyConversion exchange = new CurrencyConversion();
    Double AmountLocal = exchange.usdToLocalCurrency(country, usdCash);
    Console.WriteLine("The amount of local currency is: " + AmountLocal);
    Console.WriteLine("Please enter the temperature in Celsius");
    str = Console.ReadLine();
    Int32 celsius = Convert.ToInt32(str);
    TemperatureConversion c2f = new TemperatureConversion();
    Int32 fahrenheit = c2f.getFahrenheit(celsius);
    Console.WriteLine("Local temperature in Fahrenheit is: "+fahrenheit);
    }
}
class myCost {
  private Int32 days;              // Data member
  public myCost(Int32 daysToStay) { // Parameter passed into the class
    days = daysToStay;              // through the constructor, which is
  }                                    // used to initialize the data member
  private Int32 hotel() {
    return 100 * days;        // Parameter value used in all methods
  }
  private Int32 rentalCar() {
    return 30 * days;         // Parameter value used in all methods
  }
  private Int32 meals() {
    return 20 * days;         // Parameter value used in all methods
  }
  public Int32 total() {
    return hotel() + rentalCar() + meals();
  }
}
class CurrencyConversion  {
  public Double usdToLocalCurrency(String country, Int32 usdAmount)  {
    switch(country)  {
      case "Japan": return usdAmount * 117;
      case "EU": return usdAmount * 0.71;
      case "Hong Kong": return usdAmount * 7.7;
      case "UK": return usdAmount * 0.49;
```

```
      case "South Africa": return usdAmount * 6.8;
      default: return -1;
    }
  }
}
class TemperatureConversion {
  public Int32 getFahrenheit(Int32 c) {
    Double f = c * 9 / 5 + 32; return Convert.ToInt32(f);
  }
  public Int32 getCelsius(Int32 f) {
    Double c = (f - 32) * 5 / 9; return Convert.ToInt32(c);
  }
}
```

The classes used in this program are synthetic. When SOC is studied in the later sections, we will show that we can access remote objects over the Internet, called WS, which provide real-time services, such as obtaining the temperatures of given locations and actual currency exchange rates.

6.1.6 Parameters: passing by reference with `ref&out`

In C#, parameter passing by reference, or giving the receiving method access to permanently changing the value, is done with the **`ref`** keyword, as seen in the example below:

```
using System;
class Point {
  public Point(int x) {
      this.x = x;
  }
  public void GetPoint(ref int x) {
      x = this.x;        // this.x refers to the class member
  }
  int x;
}
class Test {
  public static void Main() {
      Point myPoint = new Point(10);
      int x = 0;
      myPoint.GetPoint(ref x);      //x = 10
  }
}
```

C# offers a second way to pass parameters by reference, with the **`out`** keyword. The `out` keyword makes it possible to pass an uninitialized parameter by reference.

```
using System;
class Point {
  public Point(int x) {
      this.x = x;
  }
```

```
    public void GetPoint(out int x) {
        x = this.x;
    }
    int x;
}
class Test {
    public static void Main() {
        Point myPoint = new Point(10);
        int x;
        myPoint.GetPoint(out x);          //x = 10
    }
}
```

6.1.7 Base classes and constructors

C# takes after the C++ model for defining a parent class in the class header. Classes may inherit from one base class at most. The C# syntax for defining a base class and for calling the base class constructor might look like this:

```
class CalculatorStack: stack {
    public CalculatorStack(int n) :stack(n) {
    ...
    }
}
```

6.1.8 Constructor, destructor, and garbage collection

Like C++, if the programmer does not define a constructor, C# creates a default constructor for each class. This ensures that the member variables of the class are set to appropriate default values, rather than pointing to random garbage. Multiple constructors can be overloaded for a class. Constructor header syntax includes the public modifier and the class name with zero or more parameters. Constructors are called automatically when the class object is instantiated and do not return a value.

In general, destructors release a reference, an object holds to other objects. In C++, the programmer is responsible for implementing a class destructor to deallocate heap memory after the object is no longer referenced. Without manual clean up, memory leaks may ultimately crash the system. C# avoids this potential problem with automatic object clean up and tracking of all memory allocation by the .Net Garbage Collector (GC). The GC is nondeterministic. It does not take up processor time by running constantly, and it only runs when heap memory is low. There are some cases when the C# programmer wants to release resources manually; for example, when working with nonobject resources like a database connection or window handle. To ensure deterministic finalization, the `Object.Finalize` method can be overridden. C# does not have a delete function. Other than that small difference, overriding `Object.Finalize` method has the same syntax and effect as using a C++ destructor, as shown in the code below:

```
public class DestructorExample{
        public DestructorExample( ){
        Console.WriteLine('Woohoo, object instantiated!');
    }
    ~DestructorExample( ){
        Console.WriteLine('Yay, destructor called!');
    }
```

```
}
```

6.1.9 Pointers in C#

C# supports the following pointer operations, which will appear familiar to C++ programmers:

`&` The address-of operator returns the memory address of the variable.

`*` The primary pointer operator is used in two scenarios:

1. to declare a pointer variable;
2. to dereference or access the value in the memory location to which the pointer points.

`->` The dereferencing and member access operator first gets the object the pointer points to, and then accesses a given member of that object. * can accomplish the same operation. These expressions equally access a member x of an object pointed to by pointer p.

```
(*p).x;
p->x;
```

Semantics for C/C++ pointers, as well as syntax for referencing and dereferencing their values, are upheld in C#. The main difference in C# is that any code using pointers needs to be marked as unsafe. The **unsafe** keyword is used as a modifier in the declaration of an unsafe method, and to mark blocks of code that call unsafe methods. Code written in the unsafe context is not explicitly unsafe—it simply allows the programmer to work with raw memory and sidesteps compiler type checking. Unsafe code should not be confused with unmanaged code; the objects in unsafe code are still managed by the runtime and GC.

Pointers in C# can point to either value types (basic data types) or reference types. However, you can only retrieve the address of a value type. Another thing to note is, if you are working with the Visual Studio IDE, the code needs to be compiled with the */unsafe* compiler option.

This example illustrates pointers in C#:

```
public class MyPointerTest {
   unsafe public static void Swap( int *xVal, int *yVal) {
        int temp = *xVal;
        *xVal = *yVal;
        *yVal = temp;
   }
public static void Main(string[] args) {
   int x = 5;
   int y = 6;
   Console.WriteLine("Original Value: x = {0}, y = {1}"", x, y);
   unsafe {
        Swap(&x, &y);
   }
        Console.WriteLine("New Value: x = {0}, y = {1}", x, y);
   }
}
```

The console outputs are:

```
Original Value: x = 5, y = 6
New Value: x = 6, y = 5
```

6.1.10 C# unified type system

C# uses a **unified type system** that makes the value of every data type an object. Reference types (complex types) and value types share the same roots through the base class `System.Object`. Value types have a minimum set of abilities inherited through this hierarchy. The following are all valid C# code examples:

```
5.ToString( )                  //Retrieves the name of an object
b.Equals( ) == c.Equals( )     //Compares two object references at runtime
w.GetHashCode( )               //Gets the hash code for an object
4.GetType( )                   //Gets the type of an object
```

Because all types inherit from objects, it is possible to use the dot (.) operator on value types without first wrapping the value inside a separate wrapper class. This solves some of the inefficient code that object-oriented programmers must write in C++ (and in Java) to wrap value types before using them like reference types.

In C++, if you want to create a method with a parameter that accepts any type, you have to write a wrapper class with overloaded constructors for each value type you want to support. For example:

```
class AllTypes {
public:
  AllTypes(int w);
  AllTypes(double x);
  AllTypes(char y);
  AllTypes(short z);
  //a constructor must be overloaded for each desired type
  //retrieving a value from this class would require overloaded functions
};
class CTypesExample
{
  public Example(AllTypes& myType) {
  }
};
```

In C#, whenever a value type is used where an object type is required, the compiler will automatically box the value type into a heap-allocated wrapper. **Boxing** is the compiler process that converts a value type to a reference type. **Unboxing** is explicitly casting the reference type back to a value type.

Boxing and unboxing example 1:

```
int v = 55;
Console.WriteLine ("Value is: {v}", v);     //Console.Writeline only accepts
objects
//The compiler wraps value types for you
int v2 = (int) v;                            //Unboxing is like Java casting
```

Boxing and unboxing example 2:

```
int v = 55;
object x = v;                  //box int value type v into reference type x
Console.WriteLine ("Value is: {0}", x);                  // Console.Writeline
int v2 = (int) x;                                       // only accepts objects
```

A unified type system makes cross-language interoperability possible. Other benefits of the type system include guaranteed type safety, a security enhancement where each type in the system is tracked by the runtime. The overall effect is a safer, more robust code, with mainstream functionality creating a conceptually simpler programming model.

6.1.11 Further topics in C#

The purpose of this section is to extend the object-oriented features discussed in C++. C# is a powerful programming language to which many books and websites have been dedicated. Key topics that are integral in C#, which are not covered extensively here, include: generics, indexers, properties, event handling, delegates, attributes, and a long list of capabilities within the namespaces of the .Net framework.

Section B.2.3 in Appendix B introduces how to use Visual Studio to compile and run C++ and C# programs.

6.2 Service-oriented computing paradigm

The evolutional and technological shift from products to services means that the value of a technology is moving from the resulting product itself to how the technology is being received by its users. The value (in terms of increased productivity of using the product, total cost of ownership, improved efficiency and effectiveness, return on investment formula or benchmark, project completion time, and increased revenue) must be explicit and reflected from the technology's investment. For software products, the emerging SOC, including service-oriented architecture (SOA) and WS technologies, reflects the shift from the product-oriented paradigm to the service-oriented paradigm. This paradigm shift is completely changing the way we develop and use computer software. In the near future, computer users may no longer need to buy hardware or software. All they will need to do is to sign up for a service contract with a service provider or broker. Hardware with necessary software can be provided for free. Users are charged for the services they use, similar to the models used by cable TV or cellular phone operators. The services can be provided online through the web technology. Computer companies such as IBM, Microsoft, Micro Sun Systems, Oracle, and SAP have started to implement the new paradigm. For example, WebSphere (IBM) and Visual Studio .Net (Microsoft) are platforms that support the development and applications of WS. The software licensing model is also being changed from the previous model, which is a step toward this service-oriented direction: the users have to register the software or it stops functioning within a certain period of time. Furthermore, the entire software is no longer stored on CDs. They are partially stored on the vendor's web servers with appropriate access control.

In the traditional software development process, the developer takes the requirements, converts them into the specification, uses a programming language to code the specification, and then uses a compiler to translate the code into the executable. The SOC paradigm evolves from the object-oriented computing (OOC) and component-based computing paradigms by splitting the developers into three independent but collaborative parties: the application builders (also called service requesters), the service brokers (or service publishers), and the service providers. The responsibility of the service providers is to write program components, such as classes, and then, to wrap them into services with the open standard interfaces. The service brokers publish and market the available services. The application builders find the available services through service brokers and use the services to compose new applications. The application development is done via discovery and composition, rather than traditional design and coding.

6.2.1 Basic concepts and terminologies

Technically, a **service** is the interface between the producer and the consumer. From the producer's point of view, a service is a function module that is well defined, self-contained, and does not depend on the context or state of other functions. In this sense, a service is often referred to as a service agent or simply an agent. These services can be newly developed modules or just wrapped around existing legacy programs

to give them new interfaces. From the application builder or user's point of view, a service is a unit of work done by a service provider to achieve desired end results. A service normally does not have the human user's interface. Instead, it provides a programming interface so that a service can be called (invoked) by other services. For human users to use the services, a user interface needs to be added. For example, the airline's services always have two sets of interfaces. The programming interfaces are used by programs for automated search (e.g., one can write a program to find the lowest fare across airlines from city A to city B with a given number of stops). The human user interfaces allow human users to manually browse through the airline's web pages to find the tickets they want.

Service-Oriented Architecture is a software system consisting of a collection of loosely coupled services (components) that communicate with each other through standard interfaces and via standard message-exchanging protocols. These services are autonomous and platform independent. They can reside on different computers and make use of each other's services to achieve their own desired goals and end results. A new service can be composed at runtime based on locally or remotely available services. Remote services can be searched and discovered through service brokers that publish services for public access.

Web services implement a web-based SOA and a set of enabling technologies, including XML (eXtensible Markup Language), the standard for data representation. Simple Object Access Protocol (SOAP) enables remote invocation of services across network and platforms. Web Services Description Language (WSDL) and Web Ontology Language for Services (OWL-S) are XML-based languages for service description. Universal Description Discovery and Integration (UDDI) allows WS to be published and listed in its WS registry for searching and discovering. WS have three technical aspects:

- Services are functional building blocks. Multiple services can form a composite service and the composite service becomes a new building block. Service composition can be done at runtime when such a building block is needed.
- Services are software modules that can be identified by a Uniform Resource Locator (URL) and whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. WS are often described by WSDL or OWL-S, accessed by the protocol SOAP, and published by UDDI. With an added human user interface, a single service or a composite service can form a web application. WS are normally accessed by computer programs while web applications are accessed by human users.
- Services support direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols; for example, hypertext transfer protocol (HTTP) and file transfer protocol (FTP) are independent of platforms and programming languages.

A **Service-Oriented Enterprise** (SOE) is an enterprise that implements and exposes its business processes through an SOA, and it provides frameworks for managing its business processes across an SOA landscape.

A **Service-Oriented Infrastructure** (SOI) better supports the operation of software developed in SOE. Intel proposed the SOI concept. The idea is to develop computing components, memory components, and networking components as virtual services, so that they seamlessly interoperate with software services. Another implication of SOI is that the hardware can be constructed as recomposable services which allow hardware components to be replaced or upgraded without stopping the operation of the system.

Service-oriented computing refers to the paradigm that represents computation in service-orientation concepts and principles. SOC is also used as an umbrella term for SOA, WS, SOE, SOI, etc.

Service-oriented system engineering (SOSE) is a combination of system engineering, software engineering, and SOC. It suggests developing service-oriented software and hardware under system engineering principles, including requirement, modeling, specification, verification, design, implementation, testing (validation), operation, and maintenance.

6.2.2 Web services development

Under the SOC paradigm, individual services are developed independently based on standard interfaces. They are submitted to service brokers. The application builders or service requesters search, find, bind, test, verify, and execute services in their applications dynamically at runtime. Such SOA gives the application builders the maximum flexibility to choose the best service brokers and the best services. Figure 6.2 shows a typical WS architecture, its components, and the process of registering and requesting a service.

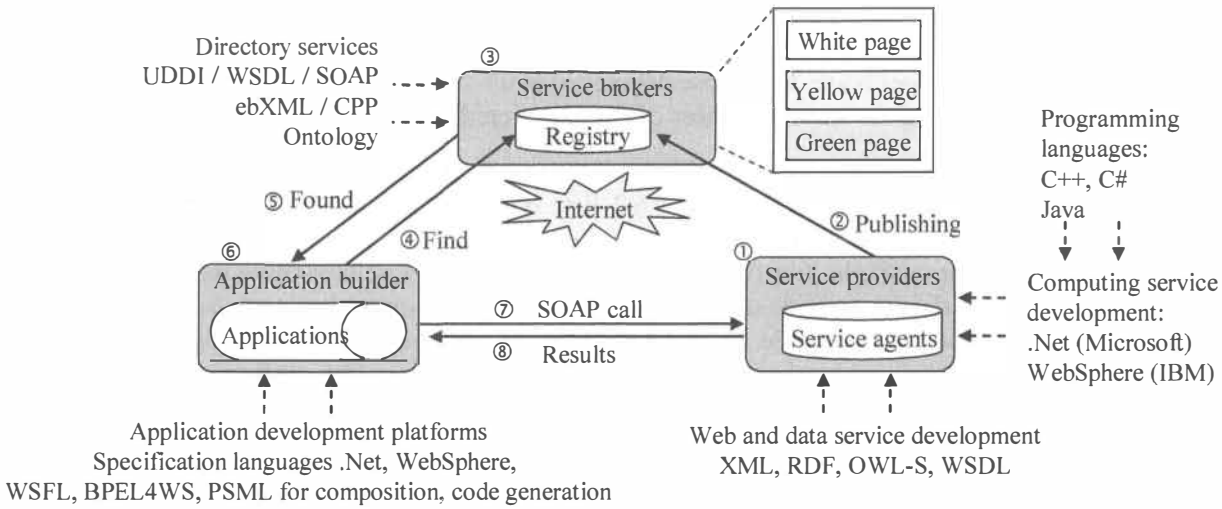


Figure 6.2. A typical web service architecture.

The components and steps shown in the diagram are explained as follows:

1. The service providers develop software components (agents) to provide different services using ordinary programming languages such as C++, C#, or Java. SOC development platforms like Microsoft Visual Studio and IBM WebSphere support the generation of standard interfaces in XML-based WSDL, which can be then registered to a service registry, such as UDDI.
2. The service providers register (publish) the services to a service broker and the services are published in the registry.
3. Current service brokers are UDDI or ebXML-based, which provide a set of standard service interfaces for publishing and discovering WS. For UDDI, the information needed for publishing a service includes: (1) White page information: Service provider's name, identification (e.g., the company's DUNS number), and contact information. (2) Yellow page information (business category): industry type, product type, service type, and geographic location. (3) Green page information: technical detail on how other WS can access (invocate) the services, such as APIs (application programming interfaces). UDDI's white and yellow pages are an analogy to the telephone white and yellow pages.
4. An application builder looks up, through the Internet, the broker's service registry, seeking desired services and instructions on how to use the services. The ontology and standard taxonomy will help automatic matching between the requested and registered services.
5. Once the service broker finds a service in its registry, it returns the service's details (service provider's binding address and parameters for calling the service) to the application builder.
6. The application builder uses the available services to compose the required application. This is higher-level programming using service modules to construct larger applications. In this way, an application builder does not have to know low-level programming. With the help of an application

development platform, the application code can be automatically generated based on the composition logic and the constituent services.

7. The code of certain services found through a broker could reside in a remote site (e.g., in the service provider's site). SOAP calls can be used for accessing the services remotely.
8. The service agents in the service provider directly communicate with the application and deliver service results.

6.2.3 Service-oriented system engineering

Service-oriented computing uses services discovered over the Internet to compose applications. The trustworthiness of the discovered services, as well as the availability of remote servers and the connectivity to the Internet, are much bigger issues than those developed in the same house. Thus, SOSE techniques will play a bigger role in ensuring the trustworthiness of the discovered services.

Because SOC software is developed by three independent yet collaborative parties, SOSE will be different from the traditional software engineering. Table 6.2 lists typical SOSE techniques in the development phases of SOC software. Because the development process is collaborative, many of the SOSE techniques are collaborative too. For example, test cases may be contributed in a collaborative manner by all three parties. The service provider can provide sample unit test cases for the service broker and service requestors to reuse. The service broker can provide its own test cases via a specification-based test case generation tool, and the broker may even make the tool available for all the parties. The application builder can use the sample test cases provided by the service broker, and also contribute its own application test cases.

While the basic engineering principles remain the same, the way they are applied will be different in the SOC paradigm. Specifically, most engineering tasks will be done on the fly at runtime in a collaborative manner. Because systems will be composed at runtime using existing services, many engineering tasks need to be performed without complete information and with significant information available just in time before application. In this way, SOSE in some ways may be drastically different from traditional engineering where engineers have complete information about the system requirements and thorough analyses can be performed even before system design is started.

Development phase	SOSE techniques
Collaborative specification & modeling	Service specification languages, model-driving architecture, ontology engineering, and policy specification
Collaborative verification	Dynamic completeness and consistency checking, dynamic model checking, and dynamic simulation
Collaborative design	Ontology engineering, dynamic reconfiguration, dynamic composition and recomposition, dynamic dependability (reliability, security, vulnerability, safety) design
Collaborative implementation	Automatic system composition and code generation
Collaborative validation	Dynamic specification-based test generation, group testing, remote testing, monitoring, and dynamic policy enforcement
Collaborative run-time evaluation	Dynamic data collection and profiling, data mining, reasoning, dependability (reliability, security, vulnerability, etc.) evaluation
Collaborative operation and maintenance	Dynamic reconfiguration and recomposition, dynamic reverification and revalidation

Table 6.2. Different SOSE techniques.

6.2.4 Web services and enabling technologies

Web services and the enabling technologies form a perfect instance of the SOC paradigm, where available resources on the web make full SOC possible and attractive. One part of WS is the **semantic web**, which represents resources, and the other part is the service agents, which make use of available resources on the web to provide services requested by the application builders and service requesters.

The semantic web is a vision for the future of the web in which information is given explicit meaning, making it easier for WS to process and integrate information available on the web in an automatic manner. A specific domain of the semantic web is called **ontology**, which defines a vocabulary of terms (words), their meanings (semantics), their interconnections (e.g., synonym and antonym), and rules of inference. The enabling technologies for WS and the semantic web include the following:

Extensible markup language **(XML)** is a universal meta language used to define other WS standards, protocols, interfaces, documents, data, etc. Like Backus Naur Form (BNF), XML is a context-free language that only defines the syntax of the language. The context and semantics of data and programs need to be defined in a detailed language or protocol based on XML syntax, such as SOAP, RDF, OWL, or WSDL, to be discussed in this chapter.

Resource Description Framework **(RDF)** can be considered a language that describes individual resources. An RDF document consists of a collection of statements. Each statement is a triple of (1) subject; (2) predicate, property, or relationship; and (3) object or value. The subject and the predicate are each a resource and the object can be a resource or a value. A statement makes an assertion that the subject is related to the object in the way specified by the predicate. An RDF document can be represented as a directed and labeled graph if we use a node to represent the subject or an object, and use an edge to represent the predicate. The subject is the source, the object is the target, and the predicate is the label of the edge.

RDF Schema (RDFS) extends RDF with frame-based primitives to specify class hierarchy and property hierarchy, with domain and range definitions of the properties, which can be used to define a simple ontology like a dictionary.

Web Ontology Language (**OWL**). In terms of the expressivity of describing the ontology or semantics, RDF and RDFS are very limited. OWL is built on RDF and RDFS to provide the description logic, including the classes and properties, as well as the constraints on the properties and their combinations using logic operators.

Web Services Description Language (**WSDL**) is an XML-based specification language for describing the WS interfaces. It can be used to define data types, input and output data formats, the operations (methods) provided by WS, network addresses, and protocol bindings.

Simple Object Access Protocol (**SOAP**) enables remote invocation of services across network and platforms. A SOAP packet is written in XML format and is normally embedded in an HTTP (hypertext transfer protocol) packet to be sent to the destination using HTTP protocol. Once the receiver receives the SOAP packet, it calls the requested service and returns the result to the requester.

Universal Description Discovery and Integration (**UDDI**) [http://www.uddi.org] provides a service directory service. The goal of the UDDI initiative is to build a standard, open, global, and platform-independent framework to share a global business registry, let business entities find each other, and define how they access each other's services over the Internet. UDDI is implemented as a WS that allows service providers to publish their WS and for service requesters to search services by name, by business type, or by geographic region. When submitting a service to UDDI, the service provider must provide the provider's identity, contact information, geographic location, service type, and service APIs for the programs of the application builders to call the services. UDDI uses WSDL for service description and uses SOAP for service invocation.

electronic business XML (**ebXML**) [http://www.ebxml.org]. Similar to UDDI, ebXML also provides a service directory, but it is more complex than UDDI. The goal of ebXML is more challenging, as it is to build a global e-business infrastructure, with the specific goal of letting small companies and companies in developing nations participate (with a very low entry cost). In ebXML, a central role is played by business processes. Every company publishes one or more Collaboration Protocol Profile (CPP), a description of the processes and the technical service interfaces to interact with them. The role of CPP is comparable to that of WSDL in UDDI, but CPP is more powerful, which allows users to define the semantics. ebXML also provides a messaging service to play the role of SOAP in UDDI.

Figure 6.3 shows the relationships among these techniques and their roles in creating SOC systems, where XML, XML Schema, DRF, RDF Schema, and OWL are used to describe resources, data, and semantics of resources and data, which form global and local ontologies. Programming languages such as Java, C++, and C# can be used to write the service agents. The IDE such as Visual Studio and WebSphere can be used to wrap the program into a standard service interface written in WSDL to form service agents, which make use of the resources, data, and ontologies to provide required services. UDDI and ebXML are used to publish services. SOAP is used for remote service invocation between services.

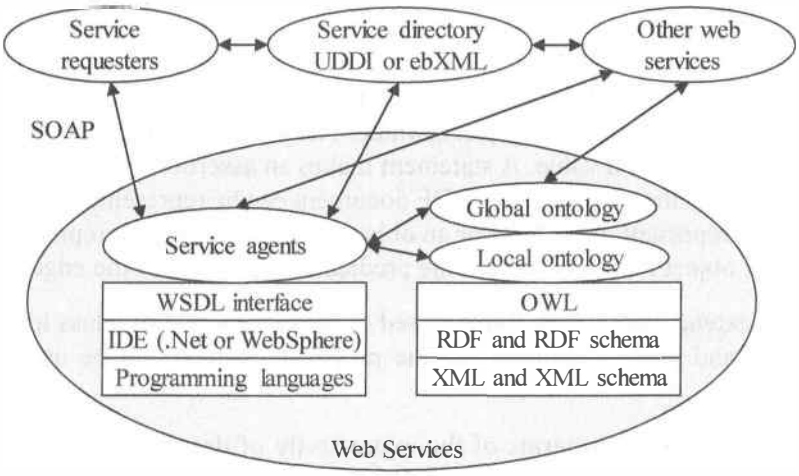


Figure 6.3. Overview of the techniques to be discussed in this chapter.

In the remainder of the chapter, we will study creating WS, registering WS, and using WS to build applications.

6.3 *Service providers: programming web services in C#

Web services are platform independent. However, the class that actually performs the computation tasks will still be written in a specific programming language. In other words, a WS is an interface that converts the service call (remote invocation) in an open standard language (e.g., XML/SOAP/WSDL) into the function-call of the programming language (e.g., C# or Java), in which the service agent is written, as shown in Figure 6.4.

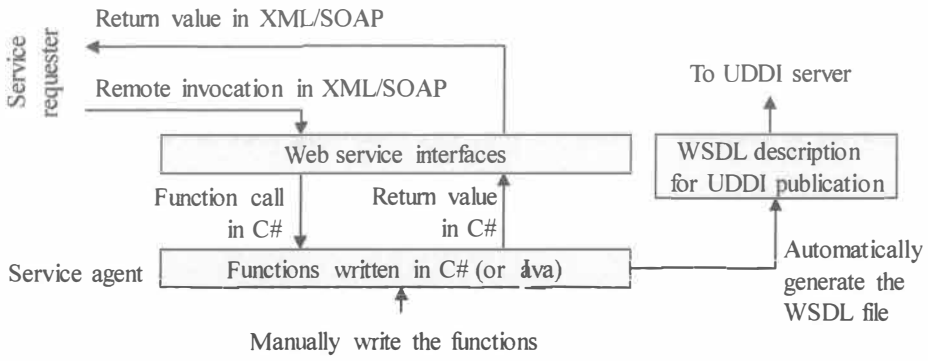


Figure 6.4. Service agent and web service interfaces.

In the previous section, we mentioned the standard interfaces in XML, SOAP, and WSDL. We did not study them in detail. In fact, we do not need to write the interfaces manually for the service agent. There are tools that can automatically generate the interface file. This is analogous to writing a web page. You can either directly use the html language to write a web page, or use a web authoring tool to write a web page.

There are different languages and development environments that can be used to develop WS. In this section, we will use C# and Visual Studio as examples to show how to develop and access WS. We will discuss:

- How a service provider creates a WS project, writes WS consisting of a set of C# functions, and automatically generates the WSDL interface file;
- How a service provider puts the WS and its WSDL interface file online so that the service requesters (clients) can remotely invocate the services;
- How a service requester (client) accesses the WS by calling the functions remotely either in a web application or in another WS.

6.3.1 Creating a web service project

ASP.NET is the built-in programming template in the Visual Studio.Net framework's Common Language Runtime (CLR) that is used to create individual WS on the service provider's server. It can also be used to create web applications on the client site to consume the WS. ASP.Net WS tutorials for programmers and a downloadable version can be found in its official site at [http://www.asp.net/Tutorials/quickstart.aspx].

Similar to creating a C++ or C# programming project, you can create a C# and ASP.Net web service template as follows:

- In the "File" menu of Visual Studio, choose "New" and "Web Site ...";
- A dialog window will be open. Choose the template "`ASP.Net Web Service.`" Choose a location and a name for your WS, for example, enter a name for the project (e.g., `c:\inetpub\wwwroot\WebStrar`). Then a WS project `WebStrar` with a stack of folders and files will be created;
- In the project stack, the file with the name "`Service.cs`" is the file in which your C# code should be incorporated. Double click on the `Service.cs` file, and you will see the template of the service agent code. You can simply add your C# functions into the template. See the next subsection.

Once you have added your code into the template, you can use the tools in Visual Studio to compile your C# code, build the project, debug the code, and execute the code.

6.3.2 Writing the service class

You still have to write the code of your service class manually to perform the required tasks (services), for example, perform addition, sorting, etc. However, ASP.NET has provided a template with defined namespaces and classes to facilitate writing of WS. The following code shows the template in the file `Service.cs`.

```
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService {
    public Service () {
        //Uncomment the following line if using designed components
        //InitializeComponent();
    }
    [WebMethod] // Make the following function accessible over Web
    public string HelloWorld() {
         return ("Hello World");
    }
    [WebMethod] // Make the following function accessible over Web
    public double PiValue() {
         double pi = System.Math.PI; // call lib function PI
         return (pi);
    }
    [WebMethod] // Make the following function accessible over Web
    public int abs(int x) {
         if (x >= 0) return (x);
         else return (-x);
    }
}
```

The program template starts with listing namespaces that could be used in the WS. Not all of the namespaces listed are necessary for this small example. Since the project name is called `WebStrar`, the new namespace created for the project is called `WebStrar`. Then, a public class called `Service` is defined, which inherits the built-in class `WebService` in the namespace `System.Web.Services`. In the class definition, it starts with the constructor `Service()`, followed by a list of functions. A list of required system functions are omitted in the given program above. Three web accessible functions are defined in the class:

The stringHelloWorld()function simply returns a string "Hello World."

The double `PiValue(int x)` function returns a double value (e.g., 3.14159265358979).

The `abs()` function takes an integer value as input and returns an integer value, the absolute value of `x`: If x is nonnegative, it returns `x`, otherwise, it returns `-x`.

Before each function, an instruction `[WebMethods]` is used, which makes the following function web accessible.

6.3.3 Launch and access your web services

Once you have incorporated all your service functions into the C# program template `Service.cs`, you can build (compile) and execute the program.

When the program is executed using the option: "debug" – "start without debugging," it will immediately launch your WS on your local host. If you are working on a Windows XP machine, the service will be launched as a local web page located at:
`http://localhost:1262/WebStrar/Service.asmx`.

Figure 6.5 shows the web page that contains the three services `abs`, `PiValue`, and `HelloWorld`, defined in `Service.asmx`.

Click on a service, and the function behind the service will be remotely invocated. For example, if `abs` is clicked, the web page shown in Figure 6.6 will pop up. Since a parameter value is needed in the WS, the web page provides an input panel to take the parameter value. After a value (e.g., `-1802`) is entered and the bottom "invoke" is clicked, the WS `abs` is remotely invoked. The return value will be shown in another web page containing the following information:

```
<?xml version="1.0" encoding="utf-8" ?>
  <int xmlns="http://tempuri.org/">1802</int>
```

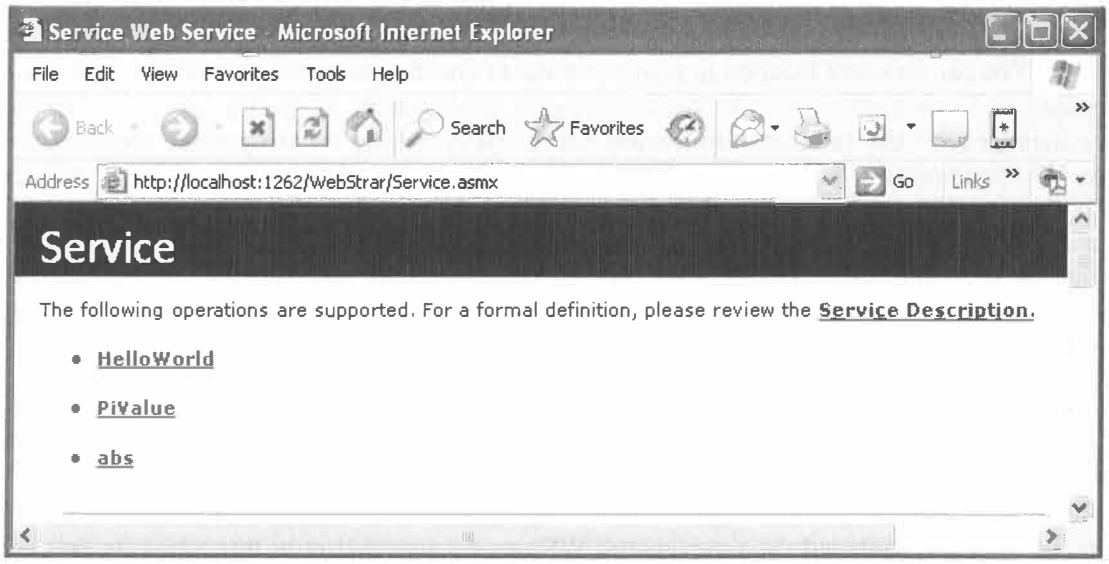


Figure 6.5. The web service is launched on a service-hosting server.

The return value `1802` is wrapped in the XML format.

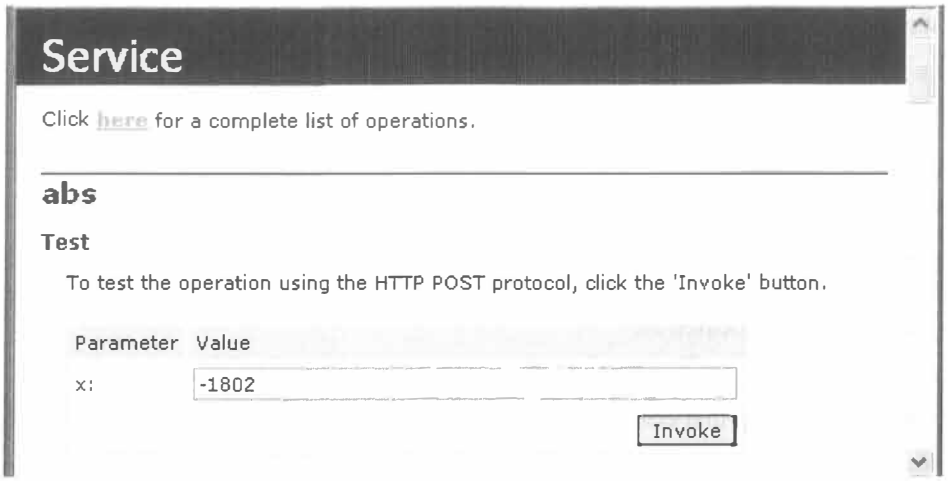


Figure 6.6. Enter the parameter value from a web browser.

The above process is in fact in the test mode. It is possible only when the services are launched on a local host. In this mode, the services cannot be remotely accessed by other application builders over the Internet. They must be deployed on an Internet-accessible server and be given a URL in order for them to be accessed by other WS or applications on the Internet. If you want to make the services available on the Internet, you have two different ways to do it.

First, you can publish the services on your Windows XP computer by following these steps. In your Visual Studio WS project, you can choose menu item "`Build`" – "`Publish Web Site.`" Then a dialog window will be open. You can choose a location in your computer to host the services; for example, you can choose the location: `C:\Inetpub\wwwroot\WebStrarServices`, where `Inetpub\wwwroot` is the web directory managed by the Internet Information Server (IIS), which is an optional component of the Windows XP operating system. You need to install IIS in order to run WS from your Window XP machine. Once the website is published, a web page with the address `http://localhost\WebStrarServices/Service.asmx` will be open, which will display the same services shown in Figure 6.5. Replacing the word "`localhost`" with the IP address of the local machine, we obtain the URL of the WS (i.e., `http://149.169.177.107/WebStrarServices/ Service.asmx` will be the URL of the services).

Second, you can publish the services if you have access to a web server (e.g., a machine running Windows Server and IIS). The service discussed here has been deployed on the server at the following address:

http://venus.eas.asu.edu/WSRepository/Services/BasicThree/Service.asmx

Once the services are deployed on a server, the WS can be accessed only through their programming interfaces. The following code shows an example of writing a SOAP/HTTP request to call the WS `abs`, and the response from the WS to be sent back to the service requester.

The following code shows the SOAP request, where the `length` needs to be replaced with the actual value.

```
POST /WebStrar/Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/abs"
<?xml version="1.0" encoding="utf-8"?>
```

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<abs xmlns="http://tempuri.org/">
<x>int</x>
</abs>
</soap:Body>
</soap:Envelope>
```

The SOAP response, where the `length` needs to be replaced with the actual value:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<absResponse xmlns="http://tempuri.org/">
<absResult>int</absResult>
</absResponse>
</soap:Body>
</soap:Envelope>
```

Furthermore, in order to publicize the WS, their URL and WSDL file must be registered to a service directory to make the services Internet-searchable.

6.3.4 Automatically generating a WSDL file

When we execute the Service1.asmx file on Visual Studio, its WSDL file is automatically generated by the ASP.Net environment and linked to the web page in Figure 6.5. When you click on the link "**Service Description,**" a web page containing the service's WSDL file will be opened. Below is a part of the WSDL file for the service.

```
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tns="http://tempuri.org/" xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://tempuri.org/">
<wsdl:types>
<s:schema elementFormDefault="qualified"
targetNamespace="http://tempuri.org/">
<s:element name="HelloWorld">
<s:complexType/>
</s:element>
<s:element name="HelloWorldResponse">
```

```
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="HelloWorldResult"
type="s:string"/>
</s:sequence>
</s:complexType>
</s:element>
<s:element name="PiValue">
<s:complexType/>
</s:element>
<s:element name="PiValueResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1" name="PiValueResult"
type="s:double"/>
</s:sequence>
</s:complexType>
</s:element>
<s:element name="abs">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1" name="x" type="s:int"/>
</s:sequence>
</s:complexType>
</s:element>
<s:element name="absResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1" name="absResult" type="s:int"/>
</s:sequence>
</s:complexType>
</s:element>
<s:element name="string" nillable="true" type="s:string"/>
<s:element name="double" type="s:double"/>
<s:element name="int" type="s:int"/>
</s:schema>
</wsdl:types>
```

This WSDL file contains the information for the UDDI server to match the service requests from the clients with the services provided by the WS.

In the next section, we discuss the service directory and, in Section 6.5, we discuss how to use the .Net framework to build applications based on the WS running on service providers' servers.

6.4 Publishing and searching web services using UDDI

There are three types of service brokers: Directory services (e.g., UDDI), repository services (e.g., ebXML), and ad hoc service listings.

6.4.1 UDDI file

The Universal Description, Discovery, and Integration (UDDI) is an OASIS standard that is used to represent, model, and publish WS (http://www.uddi.org). UDDI was initiated by IBM, Ariba, and Microsoft. Today, over 300 companies participate in the organization, including HP, Intel, Novell, and SAP. UDDI is based on existing standards, including XML, SOAP, and WSDL. UDDI's main function is a service registry, and its registry information is roughly organized in three groups:

- **White Pages** include the service provider's name, identity (e.g., the DUNS number), and contact information.
- **Yellow Pages** include the industry type, product and service type, and geographic location.
- **Green Pages** include binding information associated with services, references to the technical models that those services implement, and pointers to various file and URL-based discovery mechanisms. The information can be searched and interpreted by programs.

This section uses Microsoft UDDI Business Registry as an example to elaborate the service publication and searching processes.

All major computing companies have their UDDI implementation, including IBM, Microsoft, Oracle, and SAP. As an example, Microsoft UDDI interface is shown in Figure 6.7. The functions include:

1. **Create a new account:** This is a manual process using a graphical user interface (GUI), as shown in Figure 6.7.
2. **Publish a service provider:** This is a manual process using a GUI, as shown in Figure 6.8.
3. **Publish your services:** This can be a manual process using a GUI or an automatic process using an API, as shown in Figure 6.9.
4. **Search for services:** This can be a manual process using a GUI or an automatic process using an API, as shown in Figure 6.10.

Figure 6.7. Open an account at Microsoft UDDI registration page.

Once an account is opened, the user can register as a service provider, as shown in Figure 6.8. A unique identity number will be returned to the service provider, which can be used by the service provider for publishing WS.

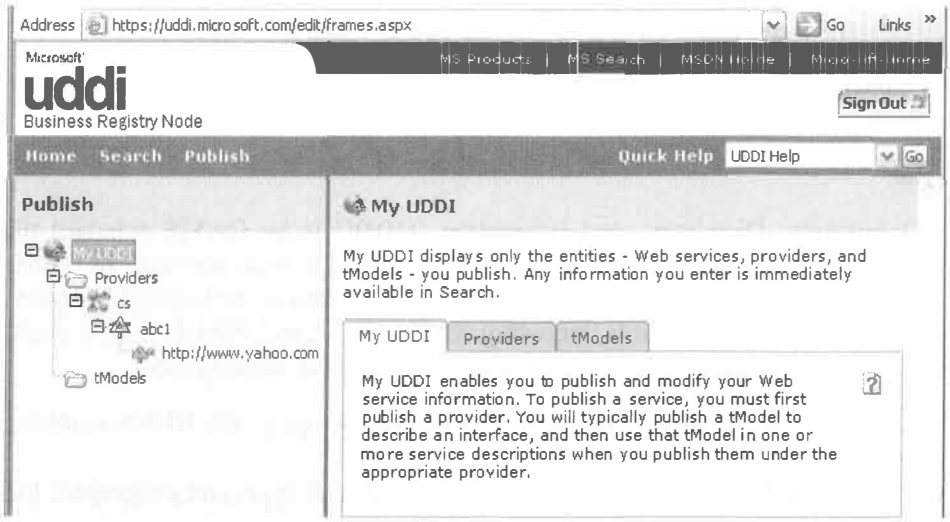


Figure 6.8. Register as a service provider.

Once a WS is published by a service provider, a tModel key will be returned to the service provider, which can uniquely identify the WS, as shown in Figure 6.9.

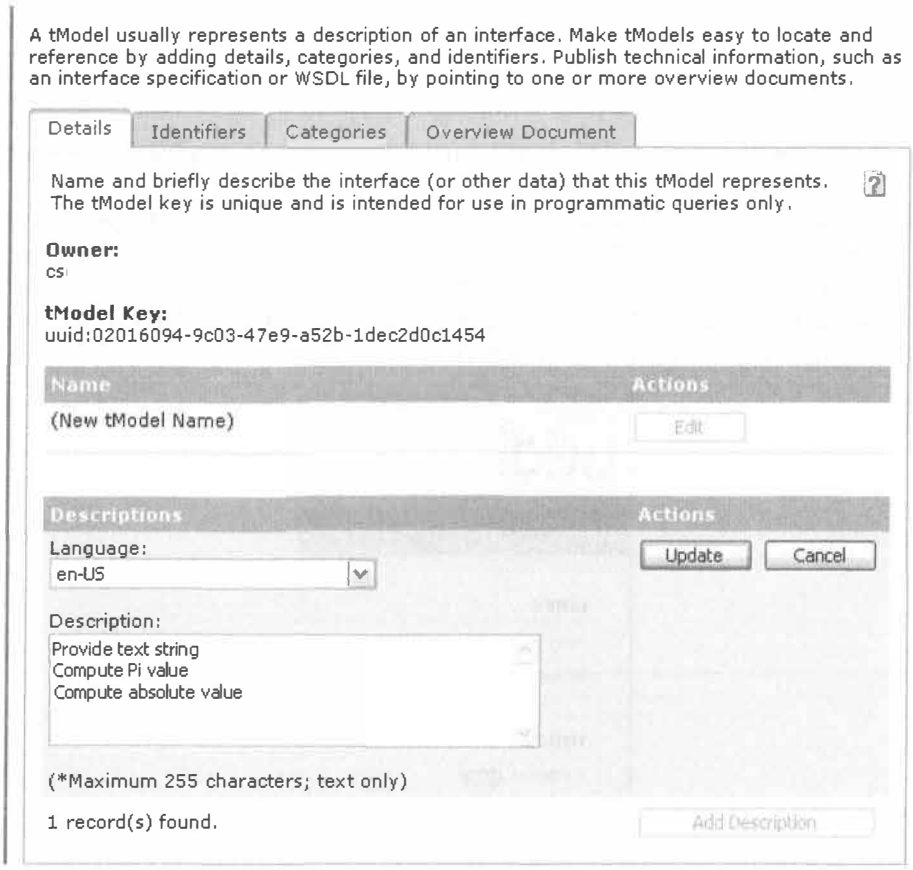


Figure 6.9. Publish a web service.

The WS published in UDDI are organized in a category tree, as shown in Figure 6.10, which can be searched manually or through a programming interface.

6.4.2 ebXML

Electronic Business using eXtensible Markup Language (eBXML) is a modular suite of specifications/standards that enables enterprises of any size and in any geographic location to conduct business over the Internet. Using ebXML, companies now have a standard method to exchange business messages, conduct trading relationships, communicate data in common terms, and define and register business processes (http://ebxml.org/).

From the service broker's point of view, ebXML defines more features than does UDDI. The most significant difference is that ebXML offers a **service repository** in addition to service registry. Since large computing companies such as Microsoft, IBM, and Oracle have resources to host their own services, they prefer the UDDI. On the other hand, ebXML is preferred by medium and small IT companies, companies whose core business is not in IT, and governmental organizations. ebXML was started in 1999 as an initiative of OASIS and the United Nations/ECE agency CEFACT. The original project envisioned and delivered five layers of substantive data specifications:

- ebXML Business Process Specification Schema
- Core Components
- Collaboration Protocol Profiles and Agreements (CPPA)
- Message Service
- Registry and Repository

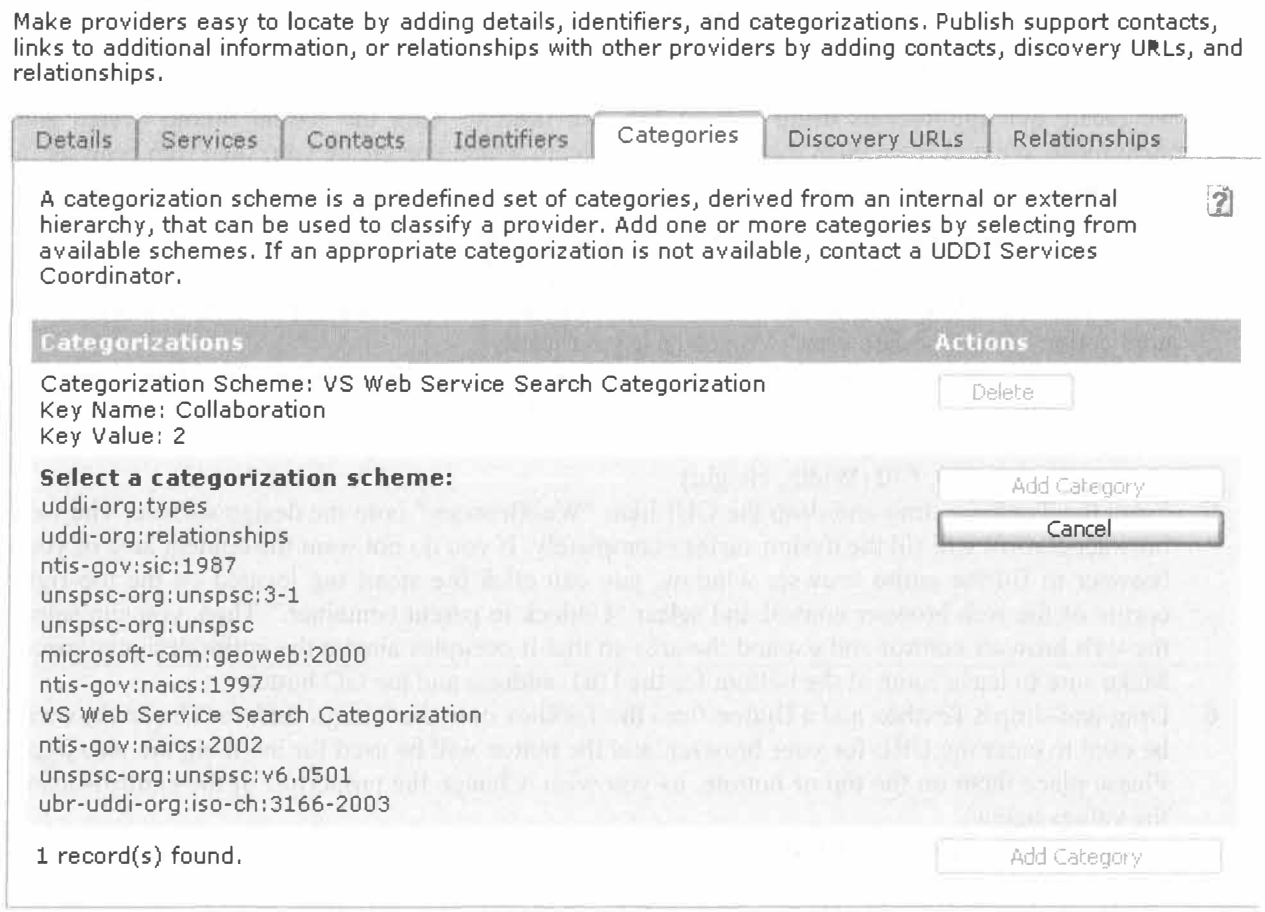


Figure 6.10. Categorization of web services in UDDI directory.

ebXML aims to provide a migration path of technologies for Small-Medium Enterprises (SME) and government organizations to an integrated eBusiness platform. The heterogeneous nature of eBusiness transactions requires a flexible infrastructure/framework that supports simple service calls and complex document exchange. For eBusiness, the key integration patterns realize SOA benefits in a pragmatic iterative manner.

6.4.3 Ad hoc registry lists

Less formally, many organizations offer a simple list of services or a manually organized service registry. Users can instantly register a new service and manually search a service by browsing through the list. Some of the useful service lists include:

- ASU Services and Application Repository (http://venus.eas.asu.edu/WSRepository/repository/)
- Remote Methods (http://www.remotemethods.com/)
- Web Service X (http://www.webservicex.net/)
- Xmethods (www.xmethods.net)

6.5 Building applications using ASP.Net

In Section 6.3, we discussed in detail how to use C# and ASP.Net to develop individual WS. This section uses the same development tool to construct applications using the remote WS available online.

6.5.1 Creating your own web browser

Before we create our applications using remote WS, we need to learn the Visual Studio design and development tools. We will learn them through an example, in which you create your own web browser in a few simple steps.

1. Start Visual Studio by clicking Start → All Programs → Microsoft Visual Studio.
2. Create a new project by clicking on File → New → Project.
3. From the New Project window, create a new Visual C# Windows application and name the application "JohnDoesBrowser." You can use your name.
4. On the newly created project, select the "Form1" and modify the following properties using the values below:
    a. Text → John Doe's Browser
    b. Size → 720, 640 (Width, Height)
5. From the Toolbox, drag-and-drop the GUI item "WebBrowser" onto the design surface. The web browser control will fill the design surface completely. If you do not want the content area of your browser to fill the entire browser window, you can click the smart tag located on the top-right corner of the web browser control and select "Undock in parent container." Then, you can select the web browser control and expand the area so that it occupies almost the entire designer space. Make sure to leave room at the bottom for the URL address and the GO button
6. Drag-and-drop a Textbox and a Button from the Toolbox onto the Design surface. The textbox will be used to enter the URL for your browser, and the button will be used for invoking the web page. Please place them on the top or bottom, as you wish. Change the properties of the controls using the values below:
    a. Textbox: (Name) → txtURL
    b. Textbox: Text: http://
    c. Size the Textbox wide enough that it can accommodate most URLs
    d. Button: Text → Go, and (name) → btnGo
7. Now, you can link the code behind the button "Go" by double clicking on the button, and it will take you to the code area. Add one line of code (highlighted) in the prototype, as shown below.

```
        private void btnGo_Click(object sender, EventArgs e) {
webBrowser1.Navigate(txtURL.Text); // Add this line of code
        }
```

8. Compile and execute your application by pressing Ctrl+F5 (or use the menu command). Your own web browser is ready to take you to any URL you enter, as shown in Figure 6.11.

Figure 6.11. John Doe's web browser GUI.

Choose "Build" → "Batch build," and check the "release" box to generate the .exe file to run on different computers. Send your browser to your friends for testing.

You can add many other features to your browser. For example, you can build a simple calculator in your browser, which allows you to do calculations while reading a web page. You can also add encryption and decryption services in your browser, so that you can encrypt your data before sending them to your friends.

6.5.2 Creating a Windows application project in ASP.Net

There are two types of applications: the applications that run on your PC and applications that run in a web browser. The web browser application we developed is running on a PC. In this section, we will develop an application running on a PC, and in the next section, we will develop an application running in a web browser.

Similar to creating a C# and ASP.Net project as discussed in Section 6.2, you can create a C# and ASP.Net Windows application template as follows:

1. In the "File" menu of Visual Studio, choose New → Project…
2. Choose C# as the project type and "Windows Forms Application" as the template. Enter a name for the project, for example, WindowsFormsApplication1 and a location. You can create a new solution for the project, or add the project in an existing solution. If you want to use the ASP.Net development server to test the WS that you created in Section 3.1 and the Windows application you are creating, you should add this project into the same solution that you created for the WS project, called WebStrar. Then a solution with two projects will be created, WebStrar and WindowsFormsApplication1, with a stack of other folders and files.
3. Once the project is created, a form called `Form1.cs` will be generated for creating the GUI. Before we design the GUI, we first link the remote service into the project.
4. To add remote WS into the application (creating proxies of services), in the Solution Explorer, mouse right-click the "References" folder in the project stack, and then choose "Add Web Reference," as shown in Figure 6.12(a). If you do not see "Add Web Reference", choose "Add Service Reference." Then, in the dialog box, click the "Advanced… ." You will then find the "Add Web Reference" option. "Add Web reference" is used for accessing services in .asmx format, while

"Add Service Reference" is used for accessing .svc services developed in Windows Communication Foundation.

5. A dialog window will be opened for "Add Web Reference." You can search the services that you want to use in the application. In this example, we will use the basic service developed in Section 6.3.1. Use the URL (http://localhost:49187/WebStrar/Service.asmx) on the local host if you developed the service, or use the deployed service at:

http://venus.eas.asu.edu/WSRepository/Services/BasicThree/Service.asmx

Copy and paste the URL into the URL textbox of the Window and click on "Go." The three services will be found and linked into the application, as shown in Figure 6.1 3. You can choose a name for the proxy class created. In the example, we choose "myFirstServices." Click on the button "Add Reference;" the services in the proxy "myFirstServices" will be added into the application stack. After the service is added, it will show up in the Solution Explorer, as shown in Figure 6.12(b).

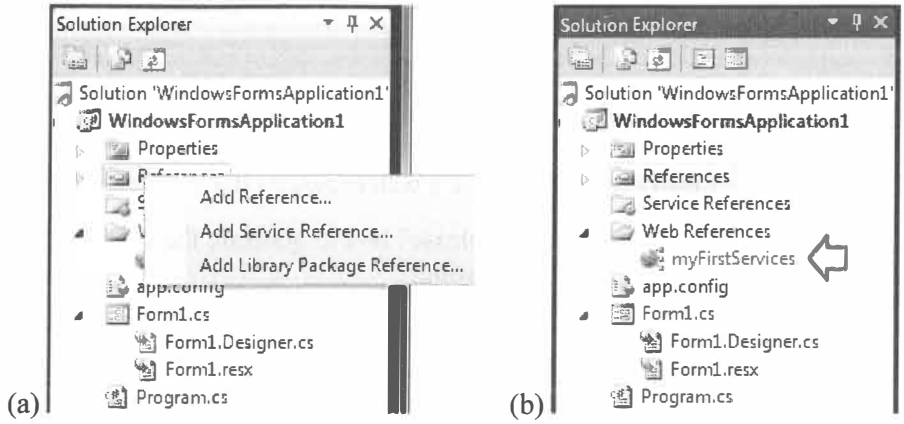


Figure 6.12. Solution Explorer before and after the web services are linked to the application.

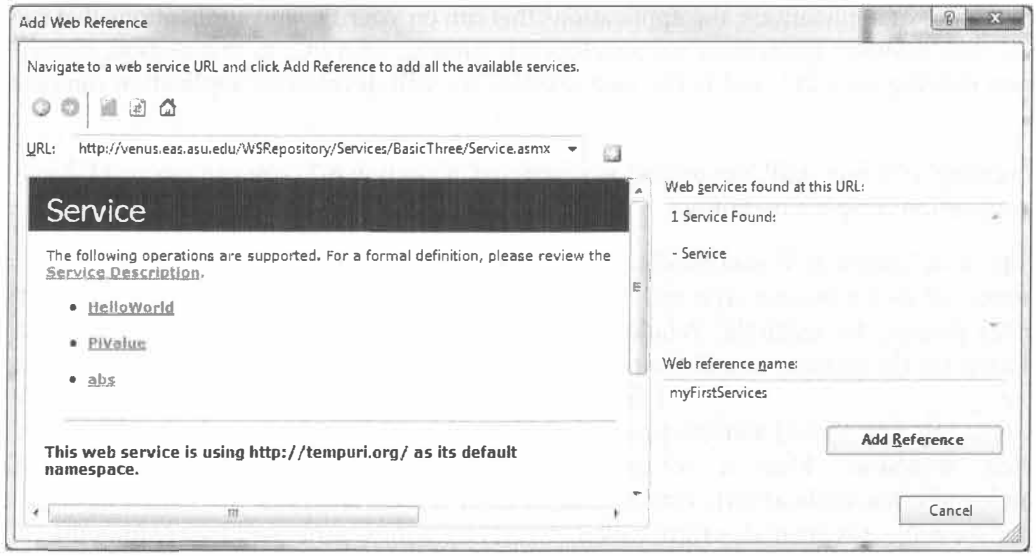


Figure 6.13. Add a web reference in a windows application.

Please notice that in step 2 above, you can also choose "`New`" and "`Web Site,`" and choose `ASP.Net Web Site` as the template. In this case, the application created will run on a server and the users can use the application over the Internet.

Both web applications and WS need to reside on a web server. However, they are different. The former refers to an application with a human user interface. Users can use a web browser to access the functions of the application. Examples of web applications include online banking, online shopping, web gaming, and online examination. The latter refers to a piece of executable code that is accessed through a programming interface. Any WS equipped with a human interface is considered a web application. A web application may use all local functions, all WS, or both as its components.

After linking the remote WS into the Windows Forms Application, we can now design the GUI. Figure 6.14 shows the GUI defined in the `Form1.cs`. We can drag and drop the GUI controls (components) in the system toolbox (button, label, textbox, etc.) on the left-hand side of the diagram into the blank area of `Form1` and add the names for them. In this example, we added four buttons and named them, respectively:

1. **Invoke String Service:** This button will be linked to the WS function HelloWorld() and the returned string will be displayed in the Label area named "Print String Value Here."
2. **Get Pi Value:** This button will be linked to the WS function PiValue() that returns the pi value and the returned value will be displayed in the Label area named "Print Pi Value Here."
3. **Get Absolute Value:** This button will be linked to the WS function abs() that returns the absolute value. The returned value will be displayed in the Label area named "Print Return Value Here."
4. **Add Pi and Abs Value:** This button will be linked to the WS functions that are composed of PiValue() plus abs(). The sum will be displaced in the Label area named "Print Result Here."

Each function above can be considered an independent application. The first three applications use one web operation each, while the fourth application uses two web operations.

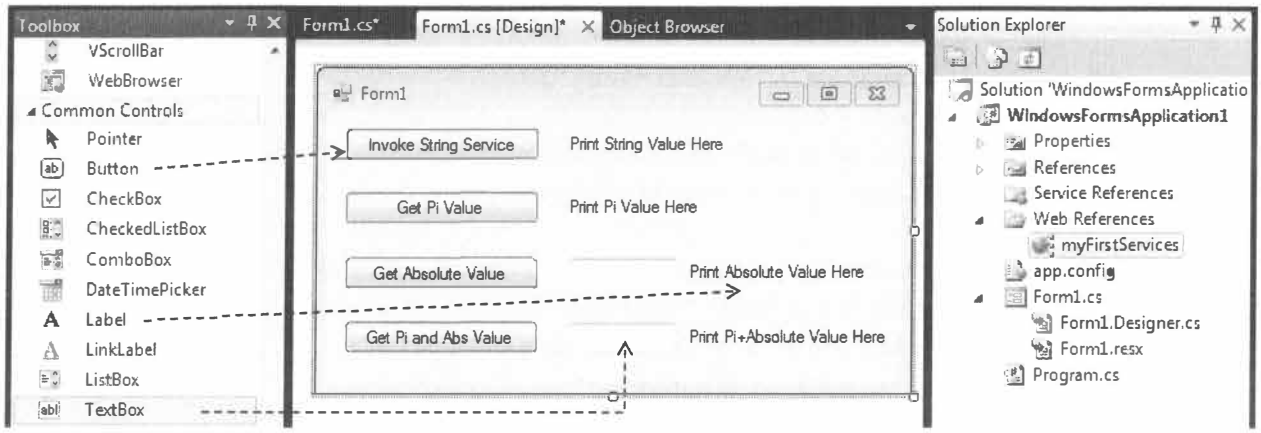


Figure 6.14. Using the toolbox to design the GUI.

While adding the GUI controls (button, label, textbox) into the form, the default names are button1, button2, label1, label2, textbox1, textbox2, etc. To make the code more readable, we have renamed buttons to btnString, btnPi, btnAbs, and btnPiAbs. We have renamed the labels to lblString, lblPi, lblAbs, and lblPiAbs. We leave textbox1 and textbox2 unchanged. Notice that renaming is done in the Property list, and it must be done before we click the button and add the code for the button.

Once the graphic interface is designed, the code that draws the graphic items such as buttons, labels, and textboxes is automatically generated from the library functions, so that we can focus on the part of the code that performs the functions we want to perform. In the code below, we have highlighted the part of the code we added into the template in boldface text.

To add your code, double click each button in the form. After each click, a method template (an empty method) will be created. All you need to do is to add code in the template to perform the task the button is supposed to perform. The C# code below shows the completed code after all the buttons are programmed.

Notice that you must add the code button by button. You cannot copy the code all together. If you copy the code all at once, the link between the button and the code will not be created.

```
using System;
using System.Windows.Forms;
using WindowsFormsApplication1.myFirstServices;// add this line
namespace WindowsFormsApplication1{
    public partial class Form1 : Form {
        public Form1() {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e){  }
        private void btnString_Click(object sender, EventArgs e) {
            Service hw = new Service();
            this.lblString.Text = hw.HelloWorld();
        }
        private void btnPi_Click(object sender, EventArgs e){
            Service pivar = new Service();
            this.lblPi.Text = pivar.PiValue().ToString();
        }
        private void btnAbs_Click(object sender, EventArgs e){
            Service absvar = new Service();
            int number = Convert.ToInt32(this.textBox1.Text);
            int result = absvar.abs(number);
            this.lblAbs.Text = result.ToString();
        }
        private void btnPiAbs_Click(object sender, EventArgs e){
            int number = Convert.ToInt32(this.textBox2.Text);
            Service absvar = new Service();
            number = absvar.abs(number);
            double result = number + absvar.PiValue();
            this.lblPiAbs.Text = result.ToString();
        }
    }
}
```

A part of the code is generated when we double click the buttons. The code that we add is highlighter. First, we added the namespace, where the service name myFirstServices is chosen in Figure 6.13.

```
using WindowsFormsApplication1.myFirstServices;
```

The namespace points to the remote WS Service(). Without using this namespace, we have to add the path to each use of Service().

Four functions are added behind the four buttons, respectively, each of which defines the action when one of the buttons is clicked by the user:

1. **BtnString_Click():** The function first creates an object of the WS Service(), and then it calls the method HelloWorld(). The returned string value is assigned to the lblString area.
2. **BtnPi_Click():** The function first creates an object of Service(), and then it calls the method PiValue(). The returned double value is converted to string and then assigned to the lblPi area.
3. **BtnAbs_Click():** Different from the first two functions, this function will take the input from textbox1, convert it into integer type, and then call the abs function in Service(). The returned value is displayed in lblAbs.
4. **BtnPiAbs_Click():** The function will call two functions in Service(). It first calls the abs() function, taking the input from textbox2. Then it calls the PiValue(), adds the two numbers, and displays the result in lblPiAbs.

When you compile and execute the program, the application GUI will be generated. After you click on the buttons, with proper input values if required, the results are displayed in the label areas, as shown in Figure 6.15.

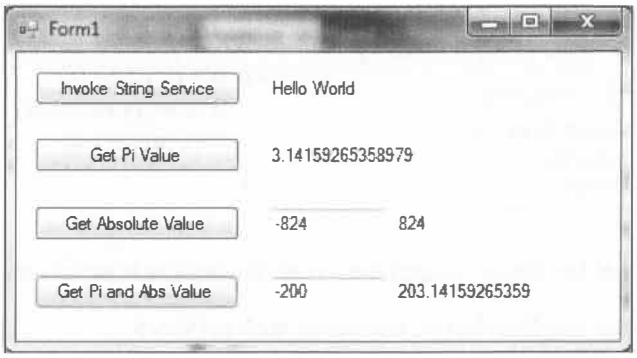


Figure 6.15. Graphic interface of the web application based on remote web services.

Note, since this application is a web application, it needs to be deployed to a web server. On the other hand, the application does not have to be a web application. It can be a Windows application. For example, if we develop a game based on WS, the game can be either a web application itself or a Windows application. In the latter case, the application can be downloaded to a Windows computer. However, the computer must have Internet access when playing the game, because the game will contact the WS at runtime.

In the code above, a remote service "Service()" is used as a class to instantiate an object and the methods in the object are used to perform the required function:

```
Service hw = new Service();
```

However, the object linked to the reference hw is a "virtual object" or a proxy, which does not contain the code for the methods. It creates a channel to each method in the remote service, as shown in Figure 6.16.

Please notice that, since this application is a forms application, it does not need to be deployed to a web server. For example, if we develop a game based on WS, the game can be a Windows Forms Application. In this case, the application can be downloaded to a Windows computer to play. However, the computer must have Internet access while playing the game because the game will contact the WS at runtime. On the other hand, website applications must be deployed to a web server. If we develop a game as a website application, one can play the game in a web browser without downloading. In the next example, we will go through the development of website application.

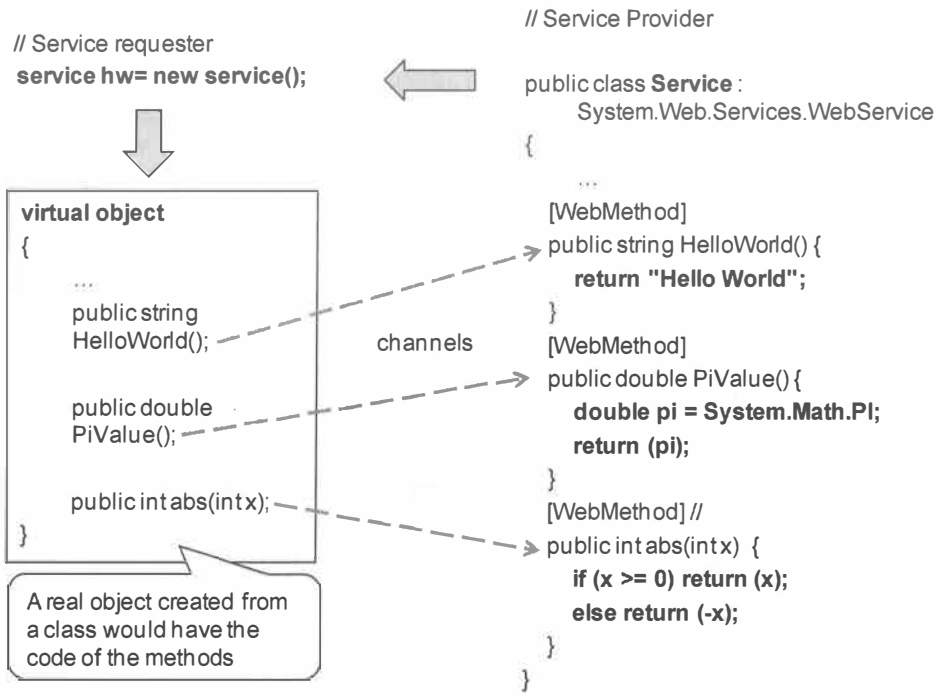


Figure 6.16. Proxy in application accessing the remote services.

6.5.3 Developing a website application to consume web services

Similar to creating a Forms application that runs on a Windows machine as discussed in the previous section, you can create a website application template in the following steps.

Step 1. Start a Website Project

In the "File" menu of Visual Studio, choose New → ASP.Net website..., and name the project TestClient, as shown in Figure 6.17. Then, a Solution with one project will be created. In the Solution Explorer, a file named `Default.aspx` will be created, which is the platform for drawing the web GUI. This form is equivalent to the `Form1.cs` in the Windows Forms Application template.

Step 2. Add Service Reference

Now, we can add the service address into our service test client. We could add the same service that we discussed in the previous example: http://venus.eas.asu.edu/WSRepository/Services/BasicThree/Service.asmx. However, we will use a real service. The service is an encryption/decryption service that can secure our WS and web applications. This service is at the address:

http://venus.eas.asu.edu/WSRepository/Services/EncryptionWcf/Service.svc

The service also has a different format (.svc file) and is developed using Windows Communication Foundation. To add this service into your application, right-click the References folder in the "TestClient" Solution Explorer, and choose "Add Service Reference..." Notice that WCF services use Add Service Reference, while ASP.Net Services use Add Web Reference.

Type the service's WSDL address

http://venus.eas.asu.edu/WSRepository/Services/EncryptionWcf/Service.svc?wsdl

into the Add Service Reference dialog window, as shown in Figure 6.18. We name the service reference "AspProxyToWcf."

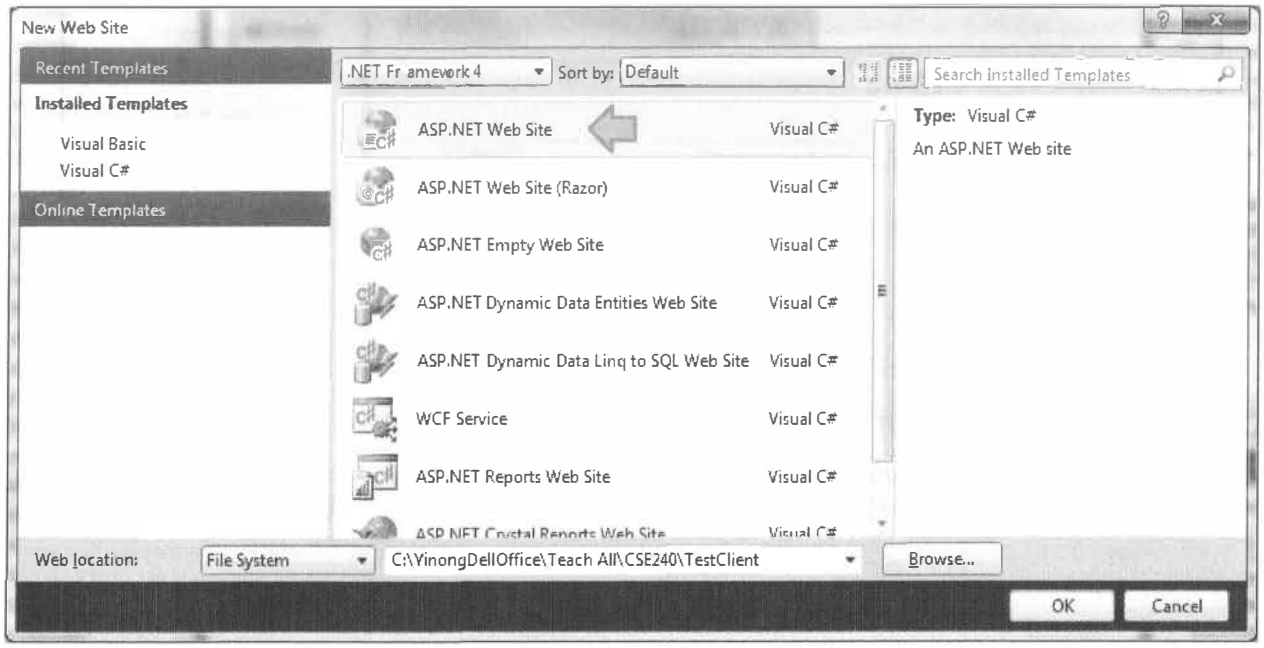


Figure 6.17. Choosing ASP.Net website as a development template.

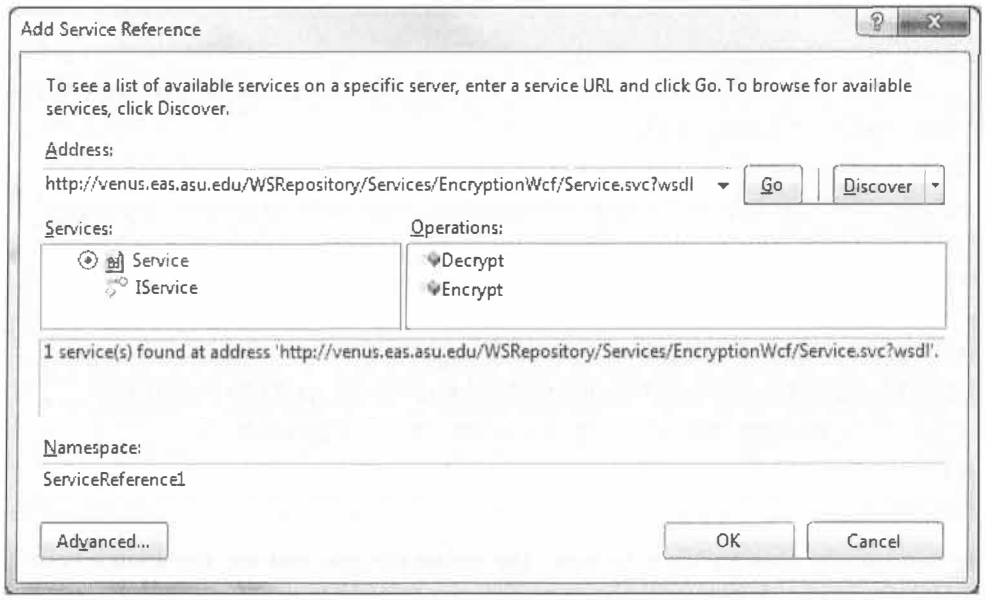


Figure 6.18. Add service reference to ASP.Net website

Step 3. Design GUI to Access WCF Service

Open the Default.aspx page. There are two ways to create the GUI. Using the design of the Default.aspx page, you can use the controls in the toolbox to draw (drag-and-drop). The GUI design is shown in Figure 6.19.

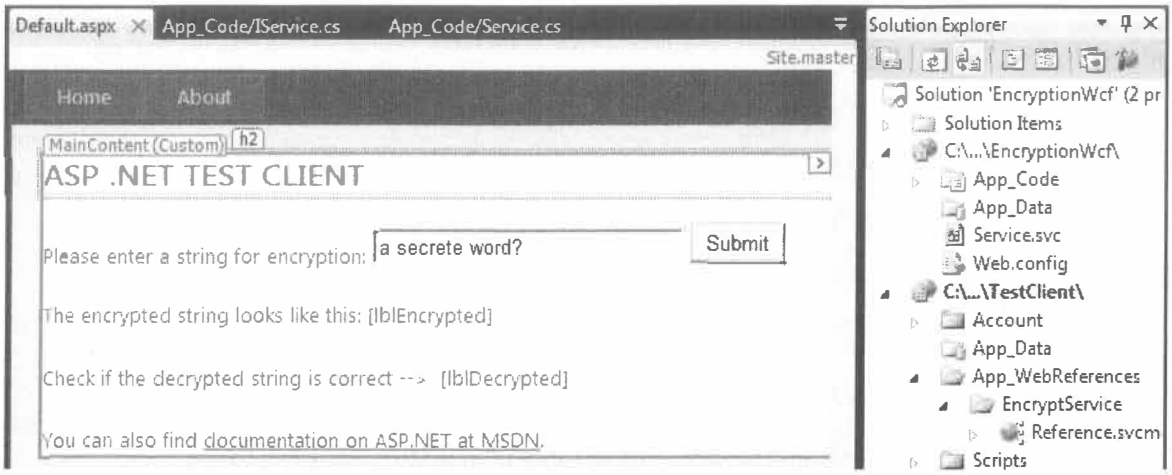


Figure 6.19. Design view of the Default.aspx page.

The other method for GUI design is to enter the Source View of the Default.aspx page and type the source code to create the GUI. The source code of the GUI in Figure 6.19 is given as follows:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.master"
AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="_Default" %>
<asp:Content ID="HeaderContent" runat="server"
ContentPlaceHolderID="HeadContent">
</asp:Content>
<asp:Content ID="BodyContent" runat="server"
                                    ContentPlaceHolderID="MainContent">
<h2>ASP.Net Test Client</h2>
<p>Please enter a string for encryption:
<asp:TextBox ID="txtInput" runat="server" Width="200px">a secrete
word?</asp:TextBox>
 <asp:Button ID="btnSubmit" runat="server" onclick="btnSubmit_Click"
          Text="Submit" /></p>
<p>The encrypted string looks like this:
<asp:Label ID="lblEncrypted" runat="server"></asp:Label></p>
<p>Check if the decrypted string is correct --&gt; 
<asp:Label ID="lblDecrypted" runat="server"></asp:Label></p>
</asp:Content>
```

Typically, you can use the Design View to make the initial design, and use the source code to refine the design. If you already have a design, you can copy the source code to make another design.

Step 4. Write Client Code to Consume WCF Service

The C# code behind the submit button in the Default.aspx page is given below:

```
using System;
public partial class _Default : System.Web.UI.Page {
    protected void btnSubmit_Click(object sender, EventArgs e){
        EncryptService.ServiceClient myClient = new
                                        EncryptService.ServiceClient();
```

```
        try{lblEncrypted.Text = myClient.Encrypt(txtInput.Text);}
        catch (Exception ec){lblEncrypted.Text = ec.Message.ToString();}
        try{lblDecrypted.Text = myClient.Decrypt(lblEncrypted.Text);}
        catch (Exception dc){lblDecrypted.Text = dc.Message.ToString();}
        }
}
```

Step 6. Test the ASP.Net Client

To build the page and start the page, the test window will be opened in a web browser, as shown in Figure 6.20. Enter a string and click the "Submit" button. The encrypted string should be displayed.

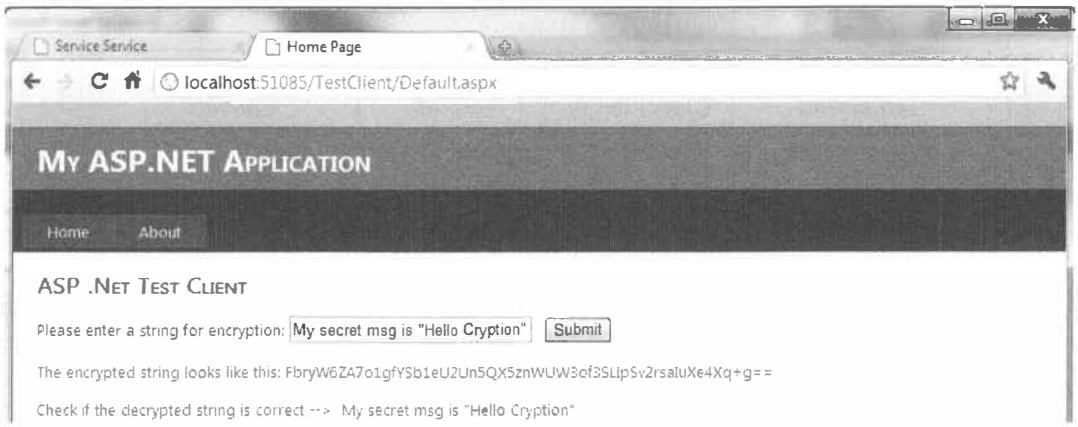


Figure 6.20. Accessing WCF service using ASP.Net website.

Now, you can extend the application to include the decryption of the encrypted text. To test your decryption service, you can copy and paste the encrypted string into the decryption textbox and see if the decrypted text is the same as the text before the encryption.

6.6 Silverlight and Phone Applications Development

ASP.Net Applications discussed in the previous section allows us to develop web applications with convenient graphic users interface. Silverlight further enables us to develop graphic users interface with animation. Silverlight can be used to develop both web applications and phone applications.

As smartphones also support ordinary web browsers, any website application can be executed on smartphones. However, the phone apps are different from web apps in several aspects. The phones have a smaller screen, and screen orientation can change when the phone is rotated. The phones have their sensory devices such as vibration sensor, GPS sensor, camera, and touch screen. The apps development can take advantage of the sensors. The phones have limited computing and storage capacity, but can be backed up by backend servers.

6.6.1 Silverlight Applications

Silverlight is a general website application development platform, which is a compact version of the Windows Presentation Foundation (WPF). Silverlight extends ASP.Net's GUI capacity for developing better presentation layer; particularly it adds the animation functions for developing applications that are more dynamical. Figure 6.21 gives an overview of the development packages on Visual Studio environment.

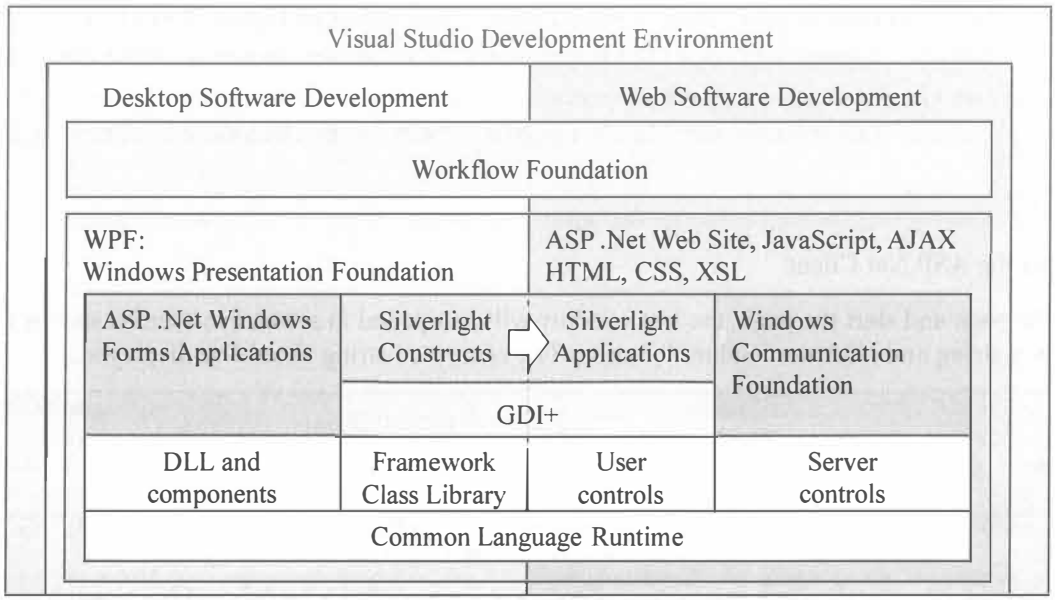


Figure 6.21. Development packages on Visual Studio environment.

In addition to the server controls offered in ASP.Net, Silverlight includes a set of additional and enhanced controls to facilitate the graphics and animation design, including:

- Common controls:
  - Textbox, Checkbox, RadioButton, ComboBox, …
  - TabControl, ScrollViewer, ProgressBar, SpeedBar, …
- Extensible control base classes
- Layout controls:
  - Grid, StackPanel
- Data controls:
  - DataGrid, TexBlock, and so on.

Silverlight supports a number of animation classes, which enable the developers to create different games and movies. The classes include:

- Linear interpretation: The object moves smoothly from point A to point B
  - DoubleAnimation
  - PointAnimation
  - ColorAnimation
- Key-frame animation: Object jumps from point A to point B
  - DoubleAnimationKeyFrame
  - PointAnimationKeyFrame
  - ColorAnimationUsingKeyFrame
  - ObjectAnimationUsingKeyFrame

To start a Silverlight project, we choose in the Visual Studio File → New → Project…, and then choose C# → Silverlight application. A project containing a MainPage.xaml file will be created, which is the main page where we design our GUI, as shown in Figure 6.22. We can start to draw the GUI items, as we did in ASP.Net, where a Default.aspx page serves the same purpose. A Silverlight GUI page is represented in XAML code, while an ASPX page is coded in XHTML. XAML has not only date representation capacity like XHTM, but also programming capacity.

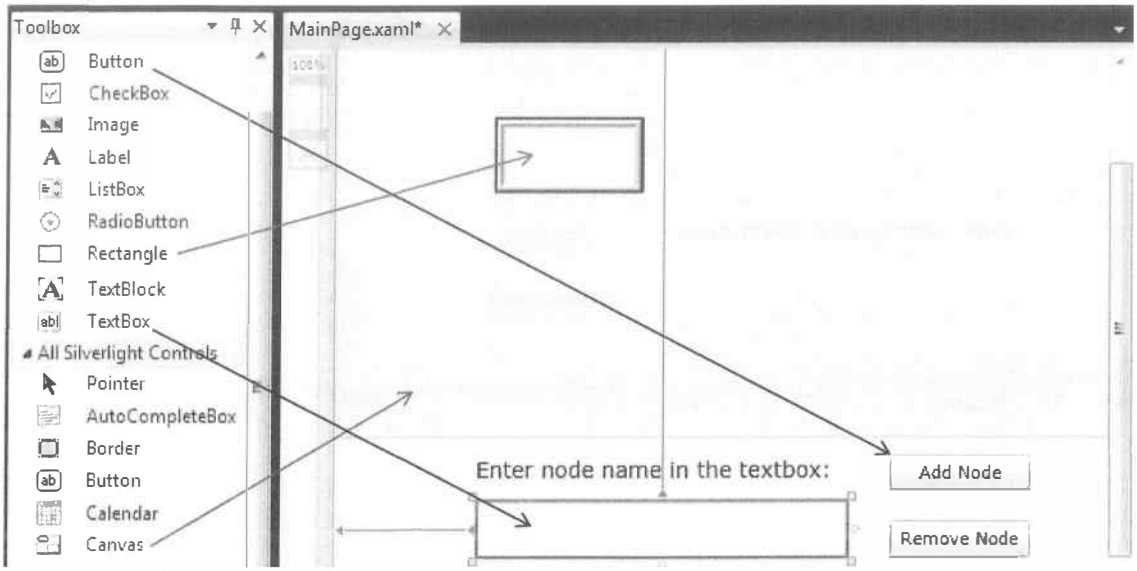


Figure 6.22. MainPage.xaml of a Silverlight project.

We can put C# code behind the .xaml page, in the same way we did in .aspx page design and coding. The following is an example where we created an animated linked list data structure with adding and removing node functions.

```
namespace LinkedList {
  public partial class MainPage : UserControl  {
      private MasterList list;
      public MainPage() {
          InitializeComponent(); list = new MasterList(mainSpace);
      }
      private void addButton_Click(object sender, RoutedEventArgs e) {
          list.addLast(textBox.Text); textBox.Text = "";
      }
      private void removeButton_Click(object sender, RoutedEventArgs e) {
          list.remove(textBox.Text); textBox.Text = "";
      }
  }
}
```

Figure 6.23 shows the Silverlight project and its files. There are two groups of files. The folder LinkedList contains all the source files, including both the GUI design files in XAML and the C# code. The folder LinkedList.Web contains all the files generated for web deployment, where the LinkList.xap file is a zip file that contains all the executable code in MSIL (MS Intermediate Language) language. The code in the xap file will be executed in the out-of-browser mode in a sandbox on the client machine. The html file LinkListTestPage.html will be the access point from a browser to execute the Silverlight application. Notice that the files can be deployed on a file server without IIS hosting. If we do have an IIS server, we can also access the LinkListTestPage.aspx file, which is also created in the LinkedList.Web folder.

Silverlight is used for building both website and phone applications. In this section, we briefly discussed using Silverlight to build website applications. We will show more Silverlight applications in the next section in conjunction with Windows Phone apps development. The Silverlight components discussed in phone applications development can be applied to web application development as well.

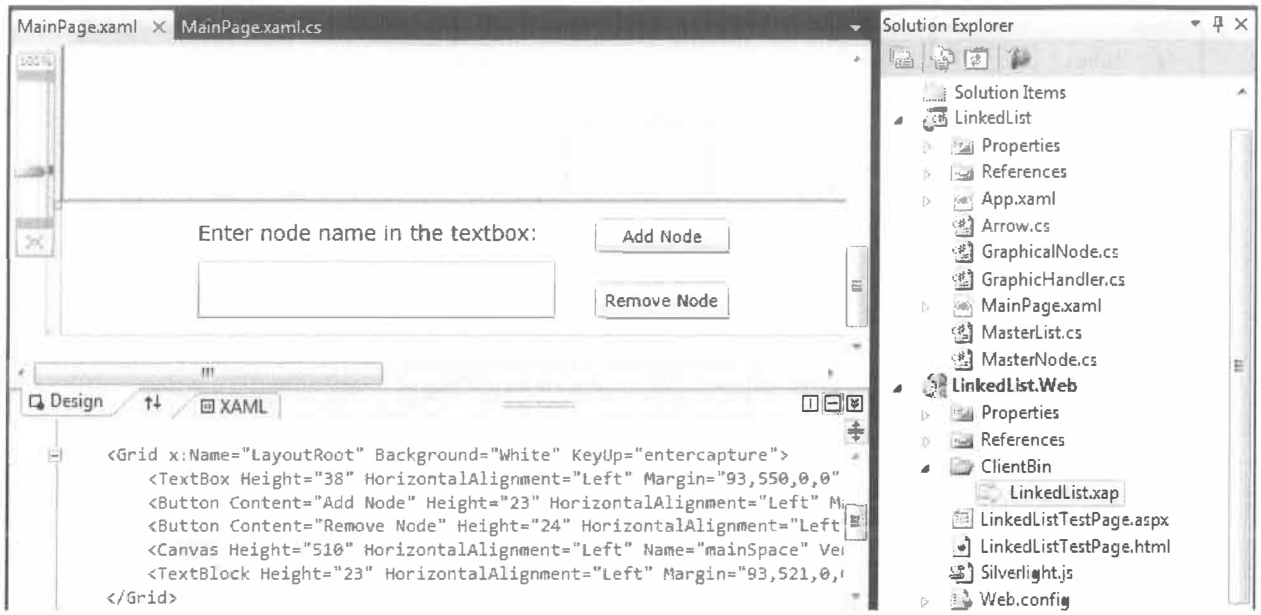


Figure 6.23. Silverlight project deployment files and its out-of-browser execution file.

6.6.2 Developing Windows Phone Apps Using Silverlight

There are a few major phone development platforms, including Apple iPhone, Android phone, Black Berry phone, and Microsoft Windows phone. There are many different development environments available for each of these phone platforms. In this section, we will briefly discuss using Visual Studio Silverlight for developing Windows phone apps and App Inventor for developing Android phone apps.

Windows Phone operating system operates on a micro version of the Windows operating system and the .NET framework. It incorporates Silverlight and XAML into its GUI design. Any .NET common language can drive a Windows Phone behind GUI, including C#, F#, and Visual Basic. C# will be used in this section to implement our service-oriented development approach. Although we are using the phone platform to host Silverlight and XAML, many of the mechanisms and techniques studied here can be applied for developing website applications as well.

To get started, we download the Windows Phone SDK from the official Windows phone development site: http://dev.windowsphone.com/en-us/home, where we can also read tutorials and download sample apps with source code. Once we have installed the SDK, we will see the Windows Phone Application template in the Visual Studio templates, as shown in Figure 6.24.

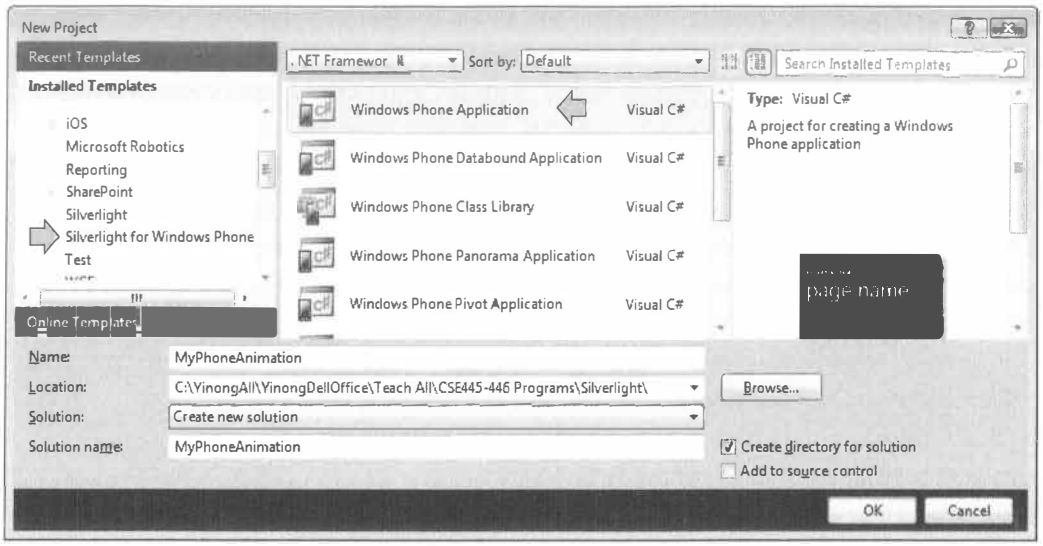


Figure 6.24. Starting a Windows Phone Application project.

Using the Windows Phone Application template, a project with a stack of files will be generated. The MainPage.xaml will be the main GUI page. We can use the Toolbox controls to draw the user interface, as shown in Figure 6.25, where we used the Ellipse, Rectangle, and Button to draw the GUI.

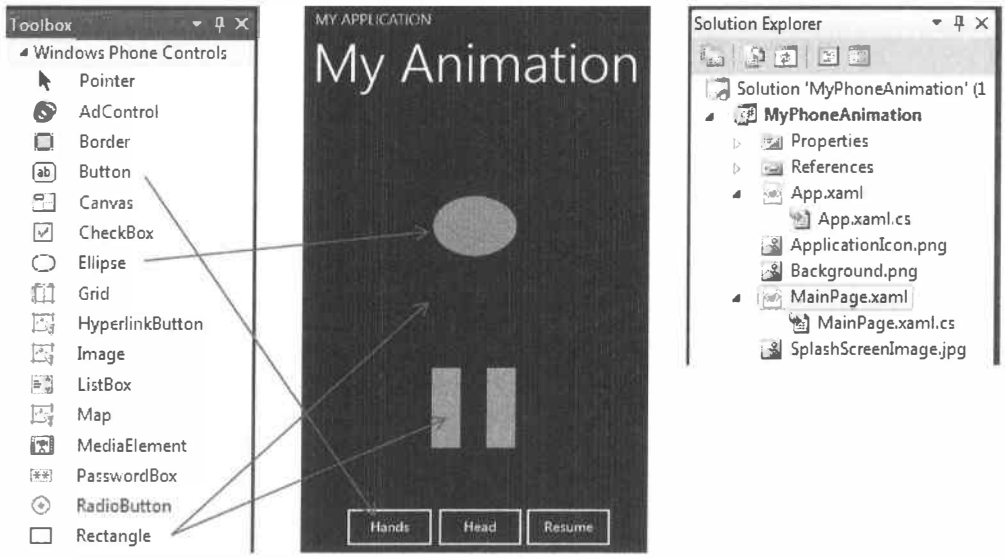


Figure 6.25.Adding ellipse and rectangle objects to the MainPage.xaml file.

Once the GUI is drawn, the .xaml file will be modified to include the objects. Now, we will open the xaml source file to add animation to the objects. The XAML code below is a part of the MainPage.xaml. The code is generated when the objects are placed on the design surface. Now, we will use XAML's programming capacity to program the animation. The highlighted code is the code that is added to define the animation of the objects.

```
<phone:PhoneApplicationPage
    x:Class="MyPhoneAnimation.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
xmlns:phone="clr-
namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignWidth="480" d:DesignHeight="768"
    FontFamily="{StaticResource PhoneFontFamilyNormal}"
    FontSize="{StaticResource PhoneFontSizeNormal}"
    Foreground="{StaticResource PhoneForegroundBrush}"
    SupportedOrientations="PortraitOrLandscape" Orientation="Portrait"
shell:SystemTray.IsVisible="True">
<phone:PhoneApplicationPage.Resources>
<Storyboard x:Name="myStoryboard">
<DoubleAnimation Storyboard.TargetName="headMove"
                 Storyboard.TargetProperty="X"
                 From="-10" To="10" Duration="0:0:0.50"
                 AutoReverse="True"
                 RepeatBehavior="Forever" />
<DoubleAnimation Storyboard.TargetName="legMove"
                 Storyboard.TargetProperty="Y"
                 From="-5" To="5" Duration="0:0:0.50"
                 AutoReverse="True"
                 RepeatBehavior="Forever" />
</Storyboard>
</phone:PhoneApplicationPage.Resources>
<!-- LayoutRoot is the root grid where all page content is placed-->
<Grid x:Name="LayoutRoot" Background="Transparent">
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition Height="*"/>
</Grid.RowDefinitions>
<!--TitlePanel contains the name of the application and page title-->
<StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
<TextBlock x:Name="ApplicationTitle" Text="MY APPLICATION"
Style="{StaticResource PhoneTextNormalStyle}"/>
<TextBlock x:Name="PageTitle" Text="My Animation" Margin="9,-7,0,0"
Style="{StaticResource PhoneTextTitle1Style}"/>
</StackPanel>
<!--ContentPanel - place additional content here-->
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
<Ellipse Name="ellipse1" Height="82" HorizontalAlignment="Left"
Margin="169,104,0,0" Stroke="Black" StrokeThickness="1"
VerticalAlignment="Top" Width="113" Fill="#FFB17F7F" >
<Ellipse.RenderTransform>
<TranslateTransform x:Name="headMove" />
</Ellipse.RenderTransform>
```

```
</Ellipse>
<Rectangle Height="134" HorizontalAlignment="Left" Margin="169,192,0,0"
Name="rectangle2" Stroke="Black" StrokeThickness="1"
VerticalAlignment="Top" Width="113" Fill="#FF5C3232" />
<Rectangle Fill="#FF6C4040" Height="108" HorizontalAlignment="Left"
Margin="148,206,0,0" Name="rectangle3"
Stroke="Black" StrokeThickness="1" VerticalAlignment="Top" Width="40" >
<Rectangle.RenderTransform>
<RotateTransform x:Name="rotate1" />
</Rectangle.RenderTransform>
</Rectangle>
<Rectangle Fill="#FF6C4040" Height="108" HorizontalAlignment="Left"
Margin="288,192,0,0" Name="rectangle4" Stroke="Black" StrokeThickness="1"
VerticalAlignment="Top" Width="40" >
<Rectangle.RenderTransform>
<RotateTransform x:Name="rotate2" />
</Rectangle.RenderTransform>
</Rectangle>
<Rectangle  Fill="#FF529F09" Height="108" HorizontalAlignment="Left"
Margin="242,332,0,0" Name="rectangle6" Stroke="Black" StrokeThickness="1"
VerticalAlignment="Top" Width="40" >
<Rectangle.RenderTransform>
<TranslateTransform x:Name="legMove" />
</Rectangle.RenderTransform>
</Rectangle>
<Rectangle Fill="#FF529F09" Height="108" HorizontalAlignment="Left"
Margin="242,332,0,0" Name="rectangle6" Stroke="Black" StrokeThickness="1"
VerticalAlignment="Top" Width="40" />
<Button Content="Resume" Height="72" HorizontalAlignment="Left"
Margin="285,508,0,0" Name="button1" VerticalAlignment="Top" Width="134"
Click="button1_Click" FontSize="18" />
<Button Content="Hands" Height="72" HorizontalAlignment="Left"
Margin="48,508,0,0" Name="button2" VerticalAlignment="Top" Width="135"
Click="button2_Click" FontSize="18" />
<Button Content="Head" Height="72" HorizontalAlignment="Right"
Margin="0,508,154,0" Name="button3" VerticalAlignment="Top" Width="133"
FontSize="18" Click="button3_Click" />
</Grid>
</Grid>
</phone:PhoneApplicationPage>
```

We use Storyboard to define the animation. Storyboard is an element in XAML file. It can contain different child elements that associate different types of timelines and animations with different objects, for example, buttons, textbox, shapes that you draw, etc., to make the objects move by the timelines and animations. We can apply timelines and animations in many different situations. For example:

- We can make an object move automatically up and down or sideways by defined timelines.
- We can change its color when the user moves the mouse over a button.
- We can make it grow when the user selects the button.

- We can shrink away and then grow back to its original size when we click the button.
- We can fade a button when it is disabled or becomes unavailable.

In the foregoing code, the first piece of the highlighted code defines two animations using Storyboard:

- The headMove defines the head of the person to move left 10 points and right 10 points along the X-axis.
- The legMove defines the left leg of the person to move up 5 points and down 5 points along the Y-axis.

The storyboard defined still needs to be associated with an object. To do so, we need to find the "Ellipse" object, modify and add the following code into the definition of the Ellipse element.

```
<Ellipse.RenderTransform>
<TranslateTransform x:Name="headMove" />
</Ellipse.RenderTransform>
            </Ellipse>
```

Notice: when we add this line of the code, we need to change the empty node <Ellipse/> into a nonempty node, and thus, we need to remove the slash character "/" at the end of the empty node. Then, we add `<Ellipse.RenderTransform>`element and then close the `<Ellipse>` node by adding `</Ellipse>`.

Similarly, we need to associate the storyboard with the "Rectangle" object. We modify and add the following code into the definition of the Rectangle element.

```
<Rectangle.RenderTransform>
<TranslateTransform x:Name="legMove" />
</Rectangle.RenderTransform>
            </Rectangle>
```

Before we can test your animation code, we still need to modify the code behind the MainPage.xaml.cs. In the Solution Explore in your project, double click the file MainPage.xaml.cs, and add the code shown below in the page.

```
        public MainPage() {
            myStoryboard.Begin();  // add this line of code
        }
```

Now, we can test our phone app to see the head's moving. Use menu command: Debug → Start Without Debugging to test your program. Now, we should see the head is moving left and right and one of the legs is moving up and down.

Next, we will define the animation for the two arm (rectangle) objects. We add the following two pieces of code into the MainPage.xaml page:

```
<Rectangle.RenderTransform>
<RotateTransform x:Name="rotate1" />
</Rectangle.RenderTransform>
</Rectangle>
```

and then

```
<Rectangle.RenderTransform>
<RotateTransform x:Name="rotate2" />
</Rectangle.RenderTransform>
            </Rectangle>
```

If we test our phone app now, we will not be able to see any movement of the arms. The reason is that we have not defined the arms movement variables rotation1 and rotation2 in the storyboard.

There are two different ways to program animations in Silverlight. We showed how to do it in xaml code in the previous steps. We can also program the animation in C#.

We will now add more code into the C# file MainPage.xaml.cs. First, we program the right arm. In the Solution Explore in our project, we double click the file MainPage.xaml.cs. Then, we add the following code in the MainPage( ) constructor. Do not delete any code. Just add the code into the existing template.

```
public partial class MainPage : PhoneApplicationPage {
    DateTime startTime;
    int resume = 1;
    // Constructor
    public MainPage() {
        InitializeComponent();
        startTime = DateTime.Now;
        CompositionTarget.Rendering += OnCompositionTargetRendering;
        myStoryboard.Begin();
    }
    void OnCompositionTargetRendering(object sender, EventArgs args) {
        // Angle directly controlled rotation for rotation1
        rotate1.Angle = (rotate1.Angle + resume * 0.5) % 360;
        // Time-controlled rotation for rotation2
        TimeSpan elapsedTime = DateTime.Now - startTime;
        rotate2.Angle = resume * elapsedTime.TotalMinutes * 1000;
    }
}
```

In the aforesaid code, we used two different ways to control the rotations. The rotation1 uses the degrees of angle, while rotation2 uses the elapsed time. Now, we can test our phone to see the arms moving.

Next, we will further program the three buttons to stop and resume the movements of head, leg, and arms. In order to add the event handler behind each button, we double click each button to create the templates in the event handlers, and then we add the code in the event handlers.

First, double click the Hand button. It will take us to the MainPage.xaml.cs and create an event handler template for the button. We can add one line of code: `resume = 0;` this line of code will freeze the movement of the arms.

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    resume = 0; // Add this line code. It will freeze the arms
}
```

Return to the MainPage.xaml page, and double click the Head button. It will take us to the MainPage.xaml.cs and create a code template for the button. Again, we add one line of code: `myStoryboard.Pause();`

```
private void button3_Click(object sender, RoutedEventArgs e)
{
    myStoryboard.Pause();  // This line of code will freeze head.
```

```
}
```

This line of code will pause the storyboard, and thus stop both the head and leg movements.

Finally, return to the MainPage.xaml page and double click the Resume button. It will take us to the MainPage.xaml.cs, and create a code template for the button. We add two lines of code:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    resume = -1;  // Make arms rotate in the opposite direction
    myStoryboard.Begin();
}
```

The first line of the code (`resume = -1;`) will make the arms rotate in the opposite direction, and the second line of code starts the storyboard again and makes the head and leg to move again.

Now, we have completed all parts of the animation, and we can test our phone app using the Visual Studio menu command: Debug → Start Without Debugging to test the program. Figure 6.26 shows a few screenshots of the animation.

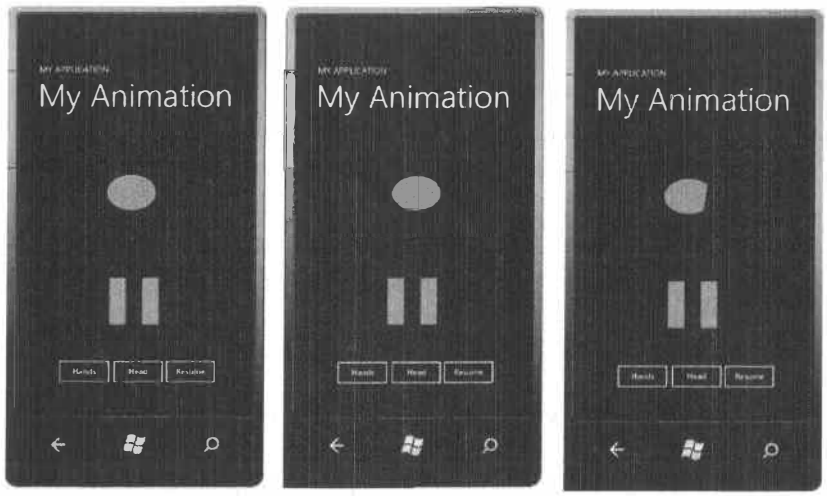


Figure 6.26. Screenshots of the phone app with multiple objects animation.

Windows phone supports both SOAP and RESTful services, and we can easily reimplement our web applications on the Windows Phone template. Figure 6.27 shows a secure messenger Phone app that calls our encryption and decryption service developed in Chapter 6, Section 6.3 and deployed in the ASU Repository of Web Services and Web Applications.

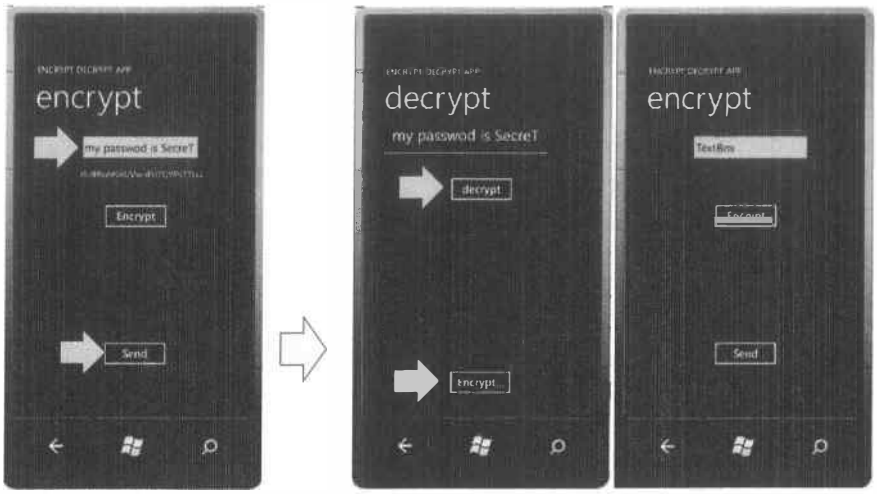


Figure 6.27. Screenshots of a secure messenger app using the encryption and decryption service.

The C# code behind the MainPage.xaml.cs of the application, which includes the encryption function, is given as follows:

```
namespace EncryptDecryptApp {
    public partial class MainPage : PhoneApplicationPage {
        public string msgEncrypted;
        public MainPage() // Constructor  { InitializeComponent(); }
        protected override void

OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e) {
            base.OnNavigatedTo(e);  // try to receive msg from another page
            string msg = "";
            // Retrieve the string that is passed in the navigation URI
            if (NavigationContext.QueryString.TryGetValue("msg", out msg))
                textBlock1.Text = msg;     // put the received data TextBox
        }
  private void Encrypt_Click(object sender, RoutedEventArgs e) {
            string msg1 = textBox1.Text;
            encryption.ServiceClient prxyEncrypt = new
encryption.ServiceClient();
            prxyEncrypt.EncryptCompleted += new
EventHandler<EncryptCompletedEventArgs>(prxyEncrypt_EncryptAsync);
            prxyEncrypt.EncryptAsync(msg1);  // call encryption service
EventHandler<System.ComponentModel.AsyncCompletedEventArgs>
(prxyEncrypt_CloseCompleted);
      }
      private void prxyEncrypt_EncryptAsync(object sender,

EncryptCompletedEventArgs e) {
            textBlock1.Text = e.Result;
        }
        private void send_Click(object sender, RoutedEventArgs e) {
```

```
            this.NavigationService.Navigate(new Uri("/decrypt.xaml?msg=" +
textBlock1.Text, UriKind.Relative));
}    }   }
```

The C# code behind the decrpt.xaml.cs is given as follows:

```
namespace EncryptDecryptApp  {
    public partial class decrypt : PhoneApplicationPage {
        public decrypt()  { InitializeComponent();  }
        protected override void

OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e) {
  base.OnNavigatedTo(e); // try to receive msg from another page
  string msg = "";
// Retrieve the query string values that were passed in the navigation URI
  if (NavigationContext.QueryString.TryGetValue("msg", out msg))
      textBlock1.Text = msg; // put the received data TextBox
  }
  private void decrypt_Click(object sender, RoutedEventArgs e) {
      string msgEncrypted = textBlock1.Text;
      encryption.ServiceClient prxyEncrypt = new
encryption.ServiceClient();
      prxyEncrypt.DecryptCompleted += new
EventHandler<DecryptCompletedEventArgs>(prxyEncrypt_DecryptAsync);
      prxyEncrypt.DecryptAsync(msgEncrypted);
  }
  private void prxyEncrypt_DecryptAsync(object sender,
DecryptCompletedEventArgs e) {
  textBlock1.Text = e.Result;
  }
      private void Encrypt_Click(object sender, RoutedEventArgs e) {
           string msg = "";
           this.NavigationService.Navigate
              (new Uri("/MainPage.xaml?msg=" + msg, UriKind.Relative));
} }   }
```

The secure messenger application does not involve animation, and the development process has little difference with that of a website application. Figure 6.28 shows the phone app that implements a simulated maze navigation application. As the maze can be modified and the robot can move in the maze, extensive animation coding is required.

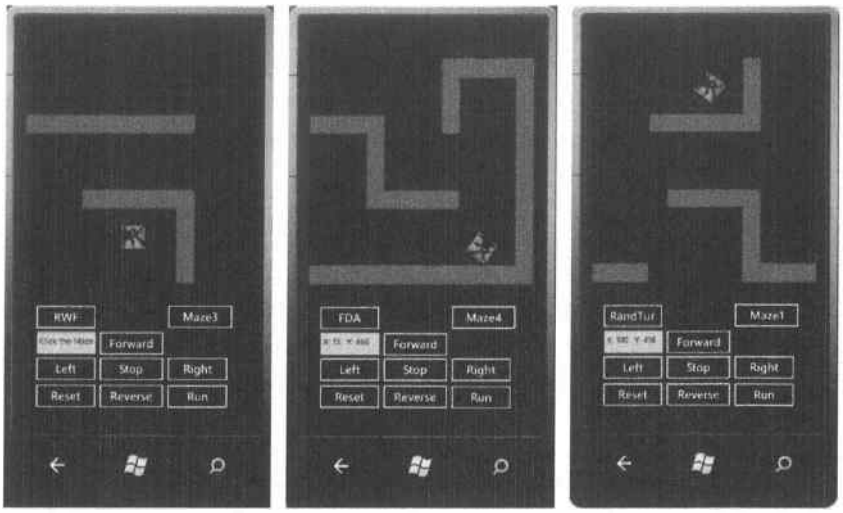


Figure 6.28. Screenshots of a maze navigation app.

6.7 Cloud computing and big data processing

Cloud computing and big data have received significant attention recently. Cloud computing offers a new computing infrastructure that enables rapid delivery of computing resources as a utility in a dynamic, scalable, and virtualized manner. The advantages of cloud computing over traditional desktop-based computing include agility, lower entry cost, device independency, location independency, and scalability. Recently, IT giants have developed their cloud computing environments, including Amazon Elastic Compute Cloud, Google App Engine (GAE), Microsoft Azure, Oracle Exalogic Elastic Cloud, and Saleforce.com cloud. The capacity of cloud computing makes it possible to process extreme big data sets to data mine valuable information.

6.7.1 Cloud computing

Cloud computing provides a large-scale delivery of services to clients and allows for flexible payment in a "pay-as-go" model without the need for clients to own the IT infrastructure or software applications. Here, the clients can be the software applications or the end users of the software applications, as SaaS refers to both the service as a software component and the application available to the end users.

Cloud computing often has several key components, including:

- **Top level:** This level hosts Software-as-a-Service (SaaS) for users to use in applications;
- **2nd level:** This level provides Platform-as-a-Service (PaaS);
- **3rd level:** This level provides basic support as Infrastructure-as-a-Service (IaaS); and
- **4th level:** This is the lowest level that provides a data center.

Most of the current clouds are built on top of modern data centers. They incorporate IaaS, PaaS, and SaaS, and provide these services like utilities, so that the clients are billed by how much they use.

Data Centers: This level provides the hardware for the cloud. Data centers are usually built in less populated areas with a low energy rate and a low probability of natural disasters. Modern data centers usually consist of thousands of interconnected servers with lots of disk storage and caches connected by high-speed networks.

Infrastructure-as-a-Service: Built on top of the data centers tier, the IaaS tier virtualizes the computing power, storage, and network connectivity of the data centers, and it is offered as provisioned services to consumers. Users can scale up and down these computing resources on demand. Typically, multiple tenants coexist on the same infrastructure resources. Examples of this tier include Amazon EC2 and Microsoft Azure Platform.

Platform-as-a-Service: Often referred to as cloudware, PaaS provides a development platform with a set of services to assist application design, development, testing, deployment, monitoring, and hosting on the cloud. It usually requires no software download or installation, and supports geographically distributed teams to work on projects collaboratively. Google App Engine, Microsoft Azure, and Amazon Map Reduce/Simple Storage Service are among the examples of this tier.

Software-as-a-Service: The SaaS is the application software presented to end users as services on demand, usually in a browser. It saves users from the troubles of software deployment and maintenance. The software is automatically updated from the clouds, and no additional license needs to be purchased typically. Features can be requested on demand, and are rolled out more frequently. Also, as a SaaS application is often a service-oriented program, it can often be easily integrated with other mashup applications. An example of SaaS is Google Map, and it can participate in various mashup applications across the web. Other examples include Salesforce.com and Zoho productivity and collaboration suites.

The dividing lines for the four tiers are not distinctive. Components and features of one tier can also be considered to be in another tier. For example, data storage services may be considered as IaaS or PaaS.

In a cloud environment, everything can be implemented and treated as a service. SaaS runs on top of a cloud environment, and it delivers web-based services to clients, and shifts IT responsibilities of application functionality, deployment, maintenance to the service providers. A client does not own the software but pays for the services on the web provided by the software. Often a user uses an API to access the services on the web. Even though it is called Software as a Service, devices and Internet of Things, such as Robot as a Service (RaaS) are included in the SaaS level to extend cloud computing into the physical world.

PaaS runs in the middle of a cloud environment, and it provides the computing environment for the SaaS to execute by using the computing, communication, and storage resources provided by IaaS. These functionalities are traditionally provided by an OS; however, PaaS may have more resources than traditional computing systems, and need to handle millions of users in real time. A common technique used by PaaS is virtualization and it provides many virtual systems to either end users or SaaS.

In a typical cloud environment, IaaS may have thousands or even hundreds of thousands of processors with a large storage capability.

Note that a SaaS application may also be considered a traditional service as both of them are based on SOC and may use common service-oriented techniques such as publication, search, discovery, ontology, composition, and policy enforcement. However, a SaaS application is indeed distinct from a service application in the following ways:

- A service application may reside locally on a computer (such as a laptop or desktop), or it can be made available as a WS supported by a server. In contrast, a SaaS application must be a web application and must run on a top of a server with large computing and storage resources.
- A service application can be shared by people, but each user will see the same service with identical functionalities; in contrast, a SaaS application can be shared by people but each user may see the

same SaaS application differently with different user interfaces and functionality. For example, a user may customize the user interface and request specific new items to display as priority items. Gmail is a typical SaaS application where a user may specify user interfaces and request specific features. This feature is not commonly available in a service application. This is the configurability of SaaS.

- A SaaS application often uses a multi-tenancy architecture where the same software is used to serve multiple clients with different features. However, a typical service application does not have this multi-tenancy architecture.

SOA and cloud computing are related. Specifically, SOA is an architectural pattern that guides business solutions to create, organize, and reuse its computing components, whereas cloud computing is a set of enabling technologies that service a bigger, more flexible platform for enterprises to build their SOA solutions. In other words, SOA and cloud computing will coexist, complement, and support each other.

Note that SaaS participants include at least the following people:

- **End users:** They consume services provided and pay for those services. They may also form Communities of Interest (COI) to share their own services and experience.
- **Business providers:** A business provider engages, influences, deploys, and supports the clients' usage of services.
- **Service providers:** A service provider creates, migrates, and composes services for users. Note that a service provider does not need to provide the platform to run these services; service providers can run their software using a public, private, or hybrid cloud environment.
- **Platform operators:** They offer a platform for managing the entire lifecycle of services from creation to deployment, operation, and delisting.

A cloud can be a public cloud where everyone can access it, a **private cloud** where only the people within a specific organization can access it, or a hybrid cloud where parts of resources are available to the public but other parts are available to selected people only. In general, cloud computing has the following features:

- **Service-oriented computing**: Most of the cloud environments support SOC and SaaS. Thus, a cloud environment often supports publishing, discovery, and composition of services including application services as well as supporting services such as information services, storage services, and communication services.
- **Web-based operations**: People will use software offered on the web, and often the software will be available as SaaS running on PaaS and IaaS. This feature provides *device and location independence*.
- **Scalable computing via dynamic provisioning**: A typical cloud environment will automatically provide sufficient resources to complete a requested task. This may involve automated workload detection, automated resource allocation, load balancing, intelligent scheduling, and parallel processing on a cluster of processors.

- **Multi-tenancy architecture with automated configuration and customization**: Instead of having a customer using individualized software, multiple customers may share the same software. In this way, *the cost of software development can be reduced as only one version of the software needs to be developed.* However, each client may still feel that the software is custom made for them.

- **Reliability and availability**: A cloud environment often has redundant resources so that if some parts of the system fail, the rest of the system can recover from the failures automatically. Furthermore, this will be done without the knowledge of users as the recovery will be autonomously done by the system.

- **Security via isolation and policy enforcement**: As a cloud environment often provides a centralized administration of distributed resources, data security is critical. Furthermore, in multi-tenancy architecture, because both the software and the databases may be shared among different clients who do not know each other, users will demand high security assurance. Various security policies may be enforced at runtime to ensure system security.

- **Automated system maintenance and upgrades**: A cloud environment often maintains its resource automatically including software and machine upgrades. Software upgrades may be handled via service updates, software configuration, and database design; and system updates will include system management and hardware replacement. Furthermore, these updates may be performed while users are using various cloud services at the same time.

6.7.2 Big data

Big data is the term for a collection of data sets, which are so large and complex that it becomes difficult to process using on-hand database management tools or traditional data processing applications [http://en.wikipedia.org/wiki/Big_data]. Sources of big data are mainly from humans through social networking and from devices in IoT (Internet of Things). The challenges in big data processing lie not only in the volume, but also in the types of data and the velocity of new data generated. There are three types of data stored in computer systems:

- **Structured Data:** Tables of data in traditional relational databases. SQL is the typical query language.

- **Semi-Structured Data:** XML files stored in XML databases, as discussed in Chapter 4 and this chapter.

- **Unstructured Data:** Data that are not structured or semi-structured, mainly streamed data like voice, photos, and video files.

A big data system typically includes all the types, and big data processing will deal with all the types simultaneously. Thus, the data type in big data systems is called poly-structured. A big data system can be characterized by several Vs, including:

- **Value:** Big data is considered the next big thing after the Internet (communication) and Cloud Computing (computation). It can bring tremendous value to the society and the economy.

- **Volume:** A moving target from petabyte (10^{15} bytes), exabyte (10^{18}), zetabyte (10^{21}), to more.

- **Velocity:** Real-time data require real-time responses.
- **Variety:** Data from different sources with different semantics are integrated into different applications.
- Variability in data structures: Poly-structured data.
- **Veracity:** Noise elimination and fault tolerance are required to process big data.
- **Volatile:** Not all data can be stored, and some will be permanently deleted, and thus, big data processing systems are required to selectively store and organize the data to maximize its value.

A big data system requires technologies from different domains. The key technologies supporting include infrastructure, management, and analytic techniques, where:

- **Infrastructure:** Parallel computing, cloud computing, storage, and database facilities. Scalability is the key issue here, including scale up and more importantly, scale out.
- **Management:** Organizing data and facilities, including data representation and management, NO-SQL (Not Only SQL) movement, Key-value store for unstructured data, CAP (Consistency and data integrity, Availability and reliability, Partition and distribution) Theory for optimization and compromise, MapReduce, and Hadoop.
- **Analytic techniques:** Specifically developed for processing big data in specific applications, which include aggregation and statistics, for example, data warehouse, data centers and OLAP (On-Line Analytical Processing); indexing, searching, and querying: keyword search & Pattern matching (XML/RDF); and knowledge discovery using data mining and statistical modeling.

The MapReduce concept is based on the higher-order functions Map and Reduce that we have discussed in Chapter 4, where:

- **Map:** Divide a list of data into n subsets and each processor performs the same operations to compute one subset.
- **Reduce:** Merge the results from subsets to one result.

Assume the job is partitioned into n pieces. For each piece i, where $i = 0, 1, 2, \ldots, n\text{-}1$.

- One Map operation takes a key-value pair $<k_{i1}, v_{i1}>$ and generates a new pair $<k_{i2}, v_{i2}>$. All Map operations will generate a list $<k_{i2}, v_{i2}>$, where $i = 0, 1, 2, \ldots, n\text{-}1$.

Map: list $< k_{i1},\ v_{i1} > \xrightarrow{\text{yields}}$ list $< k_{i2},\ v_{i2} >$

- One Reduce operation takes all n outputs from all Maps and generates a single element $<k4, \text{v3}>$.

Reduce: $< k3,$ list $v_{i2} > \xrightarrow{\text{yields}} < k4, \text{v3}>$

Note: Map and Reduce do not take list operation names in the example, because the operation is implied.

Big data systems have been applied in many domains. For example:

- **Health care:** A big data system can link all patients' data, doctors' decisions, and outcomes, require an ontology.
- **National security:** Utah Data Center is a big data system for Comprehensive National Cybersecurity Initiative. The mission is classified.
- **Tax:** IRS collects all data from all organizations and individuals to detect any tax evasion.
- **Credit scores:** U.S. companies collect many types of finance-related activities of every person with a social security number.
- **Retailers:** Not only online retailers like Amazon and eBay, but also traditional retailers like Walmart and Target, have million transactions/hour to process.
- **Recommender system:** A subclass of information filtering system that predicts the rating, ranking, and preference that a user would give to an item, or the probability that a user would buy the item.
- **Real estate:** Collect GPS signals to help home buyers to determine their drive times to work throughout at different times.

6.8 Summary

This chapter introduced the fundamentals of C# programming and the emerging SOA and the enabled programming paradigm. We started with comparing and contrasting C++ and C# through features and examples. Then, we discussed the SOC paradigm, including the basic concepts, the three-party model of service providers, service brokers, and application builders. We discussed the WS and the enabling technologies. The chapter provided examples of developing WS as a service provider, publishing and discovering WS in service directories, and composing Forms applications and website applications using existing WS.

Service-oriented architecture emerged as a new programming paradigm, which has demonstrated its strength in becoming a dominating programming paradigm. All major computing companies, including HP, IBM, Intel, Microsoft, Oracle, SAP, and Sun Microsystems, have moved into this new paradigm and are using the new paradigm to produce software and even hardware systems. The need for skill in SOA programming increases as the deployment of SOA applications increases. This new SOA paradigm is not only important in the practice of programming, but it also contributes to the concepts and principles of programming theory. In fact, SOA programming applies a higher level of abstraction, which requires fewer technical details for building large software applications. We, the authors of this book, are leading researchers and educators in SOA programming.

Service-oriented architecture has been used in web application and smartphone application development. This chapter discussed the ASP.Net development environment for web GUI application development as well as Silverlight for GUI and animation application development.

6.9 Homework, programming exercises, and projects

1. Multiple Choice. Choose only one answer for each question. Choose the best answer if more than one answer is acceptable.

1.1 What programming paradigm supports the highest level of abstraction?

❑ imperative ❑ object-oriented ❑ functional ❑ service-oriented

1.2 What language supports automatic garbage collection?

❑ C ❑ C++ ❑ C# ❑ None of them

1.3 Who will need to have a detailed knowledge of programming languages in the SOC paradigm?

❑ service providers ❑ service brokers ❑ application builders ❑ None of them

1.4 What party in the SOC paradigm needs to use an object-oriented programming language?

❑ service providers ❑ service brokers ❑ application builders ❑ None of them

1.5 What feature does C# not support?

❑ preprocessor directives ❑ macros
❑ overriding ❑ switch statement

1.6 What feature does C# not support?

❑ multiple inheritance ❑ array
❑ pointer ❑ for each statement

1.7 What do the C# **using** namespace directives replace?

❑ declaration in C++ ❑ printf statement
❑ pointer ❑ header files

1.8 What language is used for both object- and service-oriented computing?

❑ C ❑ C++ ❑ C# ❑ Scheme

1.9 What is WSDL used to describe?

❑ control flow of web services ❑ interface of web services
❑ syntax of web services ❑ semantics of web services

1.10 What is UDDI used for?

❑ describing the interface of web services ❑ composing SOA applications
❑ publishing web services ❑ calling the remote web services

1.11 What are the green pages in UDDI used for?

❑ describing contact information of service providers
❑ describing the service types of the web services

❑ describing technical detail for remote invocation of web services

❑ describing testing results on the reliability and trustworthiness of web services

1.12 What is SOAP used for?

❑ describing the interface of web service ❑ composing SOA applications

❑ publishing web services ❑ calling remote web services

1.13 What is the difference between a web service and a web application?

❑ A web service is intended for being accessed by a computer program.

❑ A web service is intended for being accessed by a human user.

❑ A web application is intended for being accessed by a computer program.

❑ Web application is a synonym of web service.

1.14 Where can we program animations in a Windows phone app? Select all that apply.

❑ in HTML file ❑ in XAMIL file ❑ in C# file ❑ in Web.config file

1.15 What animation class should be used if you want your object to jump from one point to another point?

❑ DoubleAnimation ❑ PointAnimation

❑ DoubleAnimationKeyFrame ❑ SingleAnimation

1.16 What features does SaaS have?

❑ It is identical to SOA software, and there is no difference.

❑ SaaS does not use SOA technology at all.

❑ SaaS is similar to SOA software; however, it is often hosted on a cloud environment.

❑ SaaS is the same as a web service.

1.17 What does PaaS offer?

❑ Software components and services ❑ Application development environment

❑ Computing capacity ❑ Memory and disk space

1.18 What type of data does a big data system process?

❑ Structured data ❑ Semi-structured data

❑ Unstructured data ❑ All of these types

1.19 What does veracity mean among big data characteristics?

❑ Noise elimination and fault tolerance ❑ Poly-structured data

❑ Real-time data processing ❑ Volatility of data

1.20 Where is MapReduce used in a big data system?

❑ Infrastructure ❑ Management

❑ Analytic techniques ❑ Data types

2. Add a column in table 6.1 to compare Java with C++ and C#.

3. Compare and contrast C++ and C#.

3.1 Can a scope resolution operator be used in C#?

3.2 Does a C# program need a constructor? Does a C# program need a destructor?

3.3 Are there any global functions (functions that are outside any class) in C#?

3.4 Does C# support stack objects (objects that obtain their memory from the language stack)?

4. A part of the queue and priority queue example in C++ (the `Queue` class) in Section 3.1 has been rewritten in C# in Section 6.1. Complete the priority queue class in C#.

5. Compare and contrast the object-oriented programming paradigm and the service-oriented programming paradigm.

6. What are the major differences between object-oriented software development and service-oriented software development?

7. Sorting an array or a list of numbers is a frequently used service.

7.1 Use an efficient algorithm (e.g., merge sort) and C# to write a sorting program for float numbers; wrap it as a web service; and deploy it as a web service.

7.2 Register the web service to a free UDDI server.

7.3 Build a Windows-based application that needs a sorting service and use the web service that you placed on the web server to perform the required sorting tasks.

8. In this assignment, you will write a "Rescue Turtles" game in C#. As shown in Figure 6.29, your program should display a random pattern with turtles, where an "M" represents a turtle on its feet and a "W" represents a turtle on its back. The goal of the game is to have all turtles on their feet using the minimum time and minimum number of operations. The player can choose one of the following game actions:

   - Select a column (e.g., c2): All turtles on the column will be inversed (M → W and W → M).
   - Select a row number (e.g., r3): All turtles on the row will be inversed.
   - Select a column AND a row number (e.g., c2r3): The turtle at that position will be inversed.
   - Enter a q to exit the game.

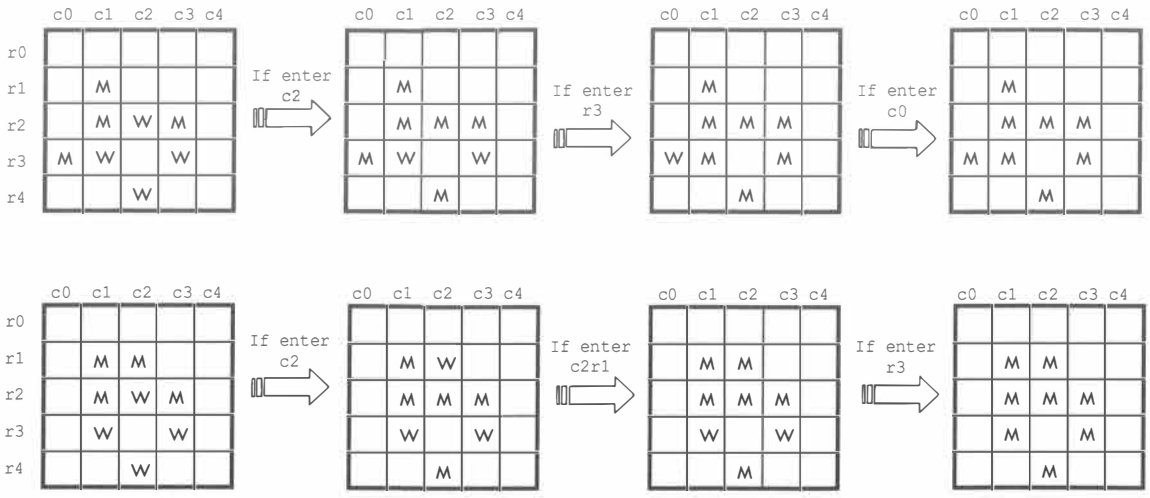


Figure 6.29. Two patterns and possible solutions.

The game should offer multiple levels of different size and complexity, for example, different sizes (3 x 3), (5 x 5), (7 x 7), and (9 x 9) and different ways of placing turtles. The game starts with a player-selected level. After a certain number of consecutive wins, the game proceeds to the next level, if there is a next level. At the highest level, the same level of the game will be continuously displayed until a "q" is entered.

The score at the end of each match (pattern) and total score will be computed by the following formulas:

Match score: MS = (200 + L)/t - 5* Nm - 10* Ni. If MS < 0, then set MS = 0.

Total score: TS = TS + MS after each match, the initial value of TS = 0.

where

L is the level of the GameLevel type and can take the values of easy, fair, tough, and extreme.

t is the time interval (in seconds) from the time point the pattern is displayed to the time point the player wins a match (all turtles are on their feet).

Nm is the number of invert actions when a column or a row number is entered.

Ni is the number of individual invert actions when both column and row numbers are entered.

8.1 Use C# to implement the Rescue Turtle game.

8.2 Convert the reusable functions into web services and put them into a web server. The game must use the web services, instead of local functions.

8.3 Use the reusable functions to build another game; you may need to add some additional functions (web services).

*8.4 Save the game in the Pocket PC or PDA format, so that the game can be played on a Pocket PC or PDA.

9. This is a group project. Figure 6.30 shows a Teaching Assistant Ontology (TAO) system that can be used to assist the instructors in keeping their test questions, as well as for students to ask questions and obtain answers.

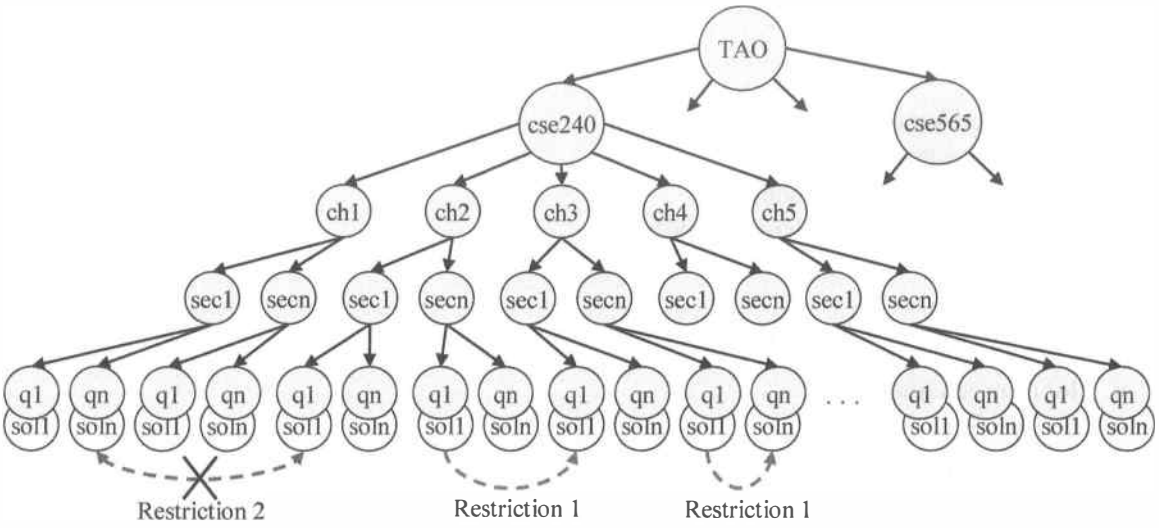


Figure 6.30. A Teaching Assistant Ontology.

The system allows the instructor to add a new course (e.g., cse565), into the system, and also add chapters, sections, and questions and solutions into the system. It also allows the instructor to specify certain relations (restrictions) among the data in the ontology, for example, restriction 1: the two questions must be in the given order if they appear in the same test paper and restriction 2: the two questions may not appear in the same test paper.

9.1 Implement the following functions as web services and save the services on a web server.

1. addTreeNode(subRoot, name);
2. removeTreeNode(nodeName);
3. addTreeLeave(subRoot, name);
4. removeTreeLeave(leaveName);
5. selectLeave(leaveName);
6. takeTest(testName);
7. gradeTest(testName, grade);
8. enterGrade(roster, testName, grade);
9. sort(roster);
10. display(roster, range);
11. login(userName, pwd);
12. logout();

9.2 Register these web services to a free UDDI server.

9.3 Make use of the above web services to build the following applications.

1. testPaper(subRoot, name);

- login(userName, pwd);
- selectLeave(name1) … selectLeave(namen);
- buildTest(testName);
- logout();

2. testGiving(testName);

- login(userName, pwd);
- takeTest(testName);
- gradeTest(testName, grade);
- enterGrade(roster, testName, grade);
- logout();

3. reportGrade(roster, key1, key2, key3);

- login(userName, pwd);
- sort(roster);
- display(roster, range);
- logout();

10. Figure 6.31 shows the component architecture model of an online bookstore.

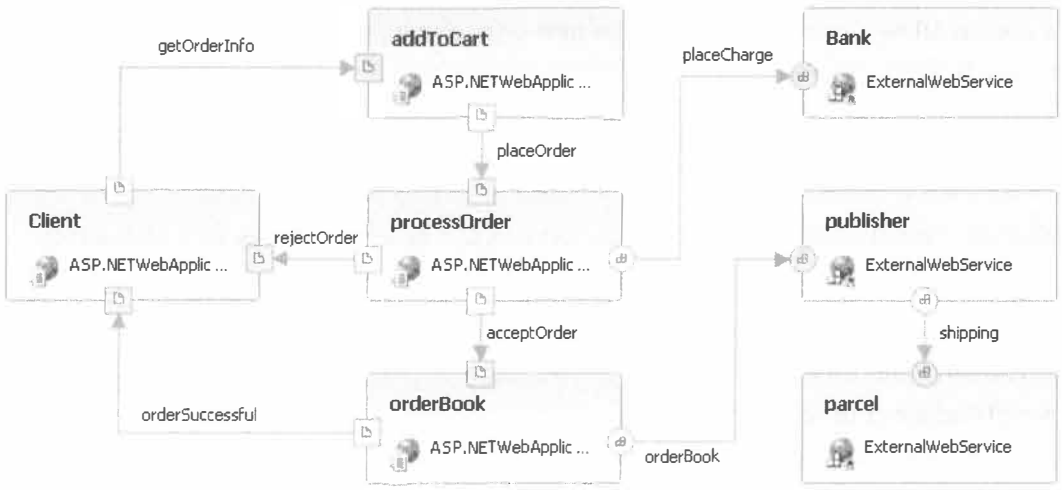


Figure 6.31. Components of an online bookstore.

10.1 Use a process specification language, such as BPEL4WS, PSML-S, WSFL, or C# to define the flow of the bookstore.

10.2 Find existing services on the Internet, where possible, and bind the services into your process model to perform required functions.

*10.3 Use a verification tool to check the properties of your process model, such as completeness, consistency, reachability, deadlock, etc.

*10.4 Use a test case generation tool to generate test cases and apply the test cases to test your program.

*10.5 Apply a reliability model to evaluate the reliability of your program based on the test data.

Appendix A

Basic Computer Architectures and Assembly Language Programming

In this appendix, we first introduce different computer architectures, and discuss how the architectures impact the way we write programs at the assembly language level. We then briefly examine how local variables are allocated on the language stack.

A.1 Basic computer components and computer architectures

At the highest level, a computer system can be abstracted as consisting of five components: control, datapath, memory, input, and output.

Control tells the other components, data path, memory, input, and output devices, what to do according to the instructions of the program. In other words, the control component of a computer decodes instructions and sends the control signals to other components to perform the desired operations.

Datapath consists of one instruction register (IR), several data registers, and an arithmetic logic unit (ALU). The data registers buffer the data fetched from memory and the ALU performs basic arithmetic and logic operations on the data stored in registers and possibly in memory. Since all high-level operations are decomposed into basic arithmetic and logic operations, datapath is the component that manipulates data in the required way. Datapath and control are also called the **processor** because they are closely related and are usually implemented on a single chip.

Memory stores the instructions (machine language code) and data translated from the programs.

Input and **output components** are the interface between the user and the computer. The input component writes the user's input to memory and the output component reads data from memory and sends them to the user. Keyboard, mouse, and scanner are typical input devices, and screen, printer, and speaker are typical output devices of a computer system.

Figure A.1 shows the five components and their interactions. A typical process is as follows. A computer program is entered through a keyboard and stored in memory. The program is compiled into machine code and stored into the memory. When this program is executed, a single instruction is fetched into the datapath at a time.

For example, assume that an addition instruction

```
add R1, R2, R3   // R1 = R2 + R3
```

is fetched into the IR in the datapath. The control reads the instruction from IR, decodes the instruction, and finds that the instruction is to add the content of register `R2` and `R3` and to store the result `R2+R3` back into register `R1`. The control then sends proper control signals to the datapath to complete the required operation.

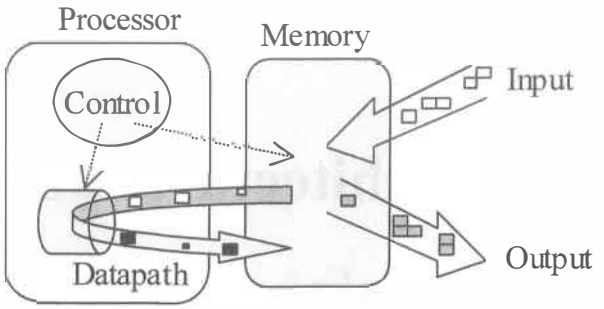


Figure A.1. Five-component model of a computer system.

In the following discussion, we will focus on the datapath, memory, and the organization of the two components, because they are directly related to imperative programming.

Figure A.2 shows four different architectures of computer systems. The **memory-memory architecture** in Figure A.2(a) has no data registers in the datapath, and, thus, the ALU has to take operands directly from memory. Since memory access (MA) is very slow, this architecture is simple but extremely slow. The **accumulator architecture** in Figure A.2(b) has one data register called the accumulator. One of the operands is always in the accumulator and the data has to be fetched into the accumulator before any ALU operation. The result is written back into the accumulator too. In the **stack architecture** in Figure A.2(c), the data registers are organized as a last-in first-out stack. The ALU can only take operands from the top of the stack. The result is pushed onto the top of the stack. The diagram in Figure A.2(d) is called **load-store architecture**. It is a generalization of the stack architecture in which the ALU can take operands from any registers and write back the result into any register.

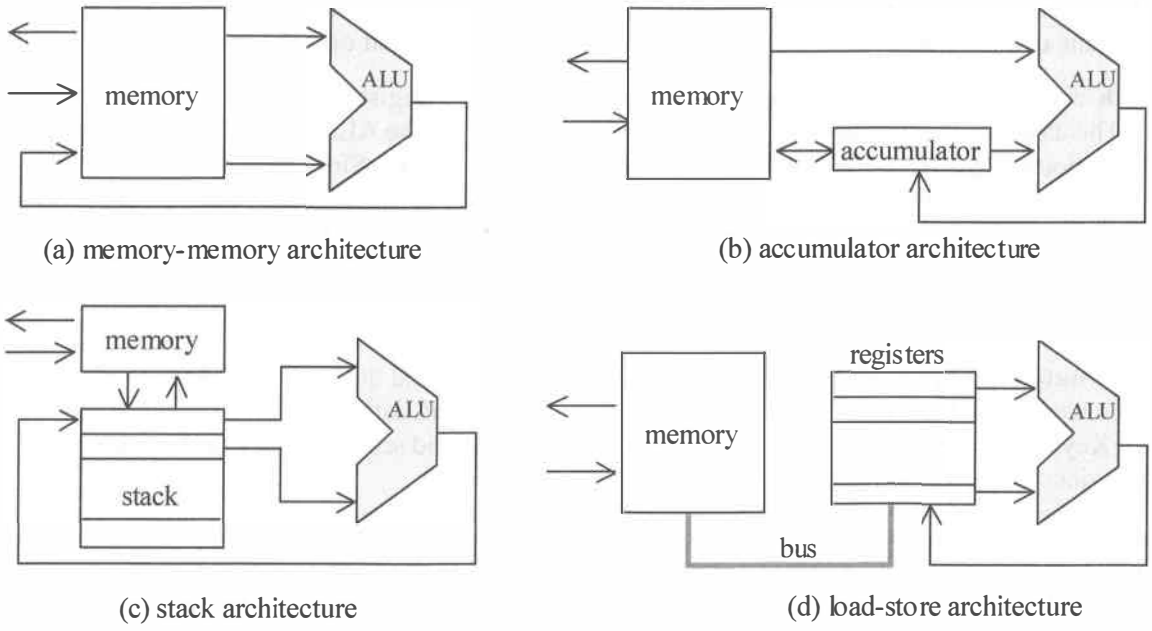


Figure A.2. Four major computer architectures.

A.2 Computer architectures and assembly programming

To see how the architectures impact the way we write programs, we show in Table A.1 the assembly language programs on the four architectures that solve the same computation problem:

```
y = x1 * x2 + x3 * x4
```

where x1, x2, x3, x4, and y correspond to memory locations.

We now compare the instruction count (IC) and the MA of the programs on the four architectures. Register or accumulator accesses are much faster than memory accesses and thus can be ignored in the analysis.

On the memory-memory architecture, the three instructions perform the following operations:

```
y1 = x1*x2;
y2 = x3*x4;
y = y1+y2;
```

Since each instruction involves reading two operands from memory and writing the result back to the memory, the total number of memory accesses is 9.

Memory-memory		Accumulator		Stack		Load-store	

mult y1, x1, x2
mult y2, x3, x4
add y, y1, y2
		Load x1
		mult x2
		Store y1
		Load x3
		mult x4
		Add y1
		Store y
				Push x1 (Load x1)
				Push x2 (Load x2)
				mult
				Push x3 (Load x3)
				Push x4 (Load x4)
				mult
				Add
				Pop y (Store y)
						Load R1 x1
						Load R2 x2
						Load R3 x3
						Load R4 x4
						mult R1 R1 R2
						mult R3 R3 R4
						Add R1 R1 R3
						Store R1 y	

IC	MA	IC	MA	IC	MA	IC	MA
3	9	7	7	8	5	8	5

Table A.1. Assembly language programs on the four architectures.

On the accumulator architecture, one of the operands of all arithmetic operations is by default in the only register (accumulator) and the result is always written back into the accumulator. In this example, in order to perform x1*x2, x1 is first loaded into the accumulator, the multiplication instruction multiplies the accumulator content with x2, and the result is written back into the accumulator. Then the content of the accumulator is stored in the memory location y1, making the accumulator free for the next multiplication. The program has seven instructions and each instruction has exactly one MA, resulting in seven memory accesses.

On the stack architecture, operands of all operations are assumed to be on the top of the stack and the results are written back on the top. A **stack** consists of a set of registers or a block of memory in which data can be stored and two operations can be performed on the data: push x and pop x. The former loads the data from memory location x and puts it on top of the stack and the latter takes (and removes) the data on the stack top and stores it into the memory location x. Figure A.3 shows the execution process of the stack-based program in Table A.1. You can imagine that the stack is a storage compartment (or magazine) that has only one access (push and pop) point and a spring is used to hold the available item to the access point. When a new item is pushed onto the stack, all items already in the stack are pushed down one place. When the item on the stack top is removed, all the items in the stack move one place up.

For example in Table A.1, the values of x1, x2, x3, and x4 are stored in memory. To compute:

y = x1 * x2 + x3 * x4

the value of `x1` is pushed (copied) onto the stack by the operation `Push x1`. Then `x2` is pushed onto the stack. The operation `mult` will pop the two values on the stack top (one after another) and perform the multiplication. The result of `x1*x2`, assumed to be (`x12`), is pushed back onto the stack. Similarly, `x3` and `x4` are pushed onto the stack, the result of `x3*x4`, assumed to be (`x34`), is pushed back onto the stack. The operation `add` will pop the two values on the stack top (one after another) and perform the addition. The result (`x12+x34`) is pushed back onto the stack. Finally, the operation `pop y` will move the data on the stack top onto memory location `y`. The stack state returns to the state before it starts the operation.

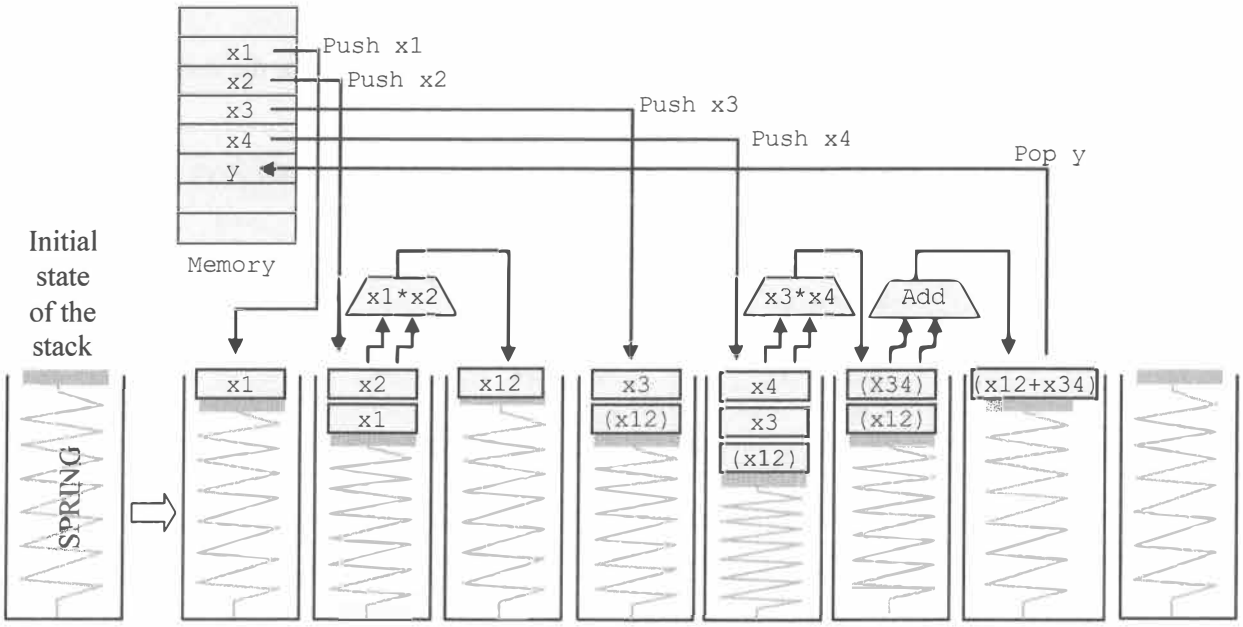


Figure A.3. Using a stack to compute `y = x1 * x2 + x3 * x4.`

In the real implementation, a stack has the same structure as a memory or an array. We use a pointer or an index variable "top" to hold the position of the stack top. Instead of moving all data in the stack up and down when pop and push operations are executed, we simply move the value of the variable top. Figure A.4 shows stack states by moving the position of the top variable.

There are eight instructions in this program; however, only push and pop instructions involve memory accesses. Thus there are only five memory accesses.

The last architecture is a generalization of the stack architecture in which the registers can be accessed in an arbitrary way, that is, the ALU can take inputs from any two registers, instead of only from the two registers below the `top` pointer. This extra flexibility does not reduce the numbers of IC and MA in this example, but in general, it can reduce IC and MA. It also makes programming easier. Load-store architecture, also called Reduced Instruction Set Computer (RISC) architecture, is a mainstream architecture used in today's computer systems. In the following example, we will explain how we write programs on the load-store architecture and what are the roles of memory, registers, and ALU.

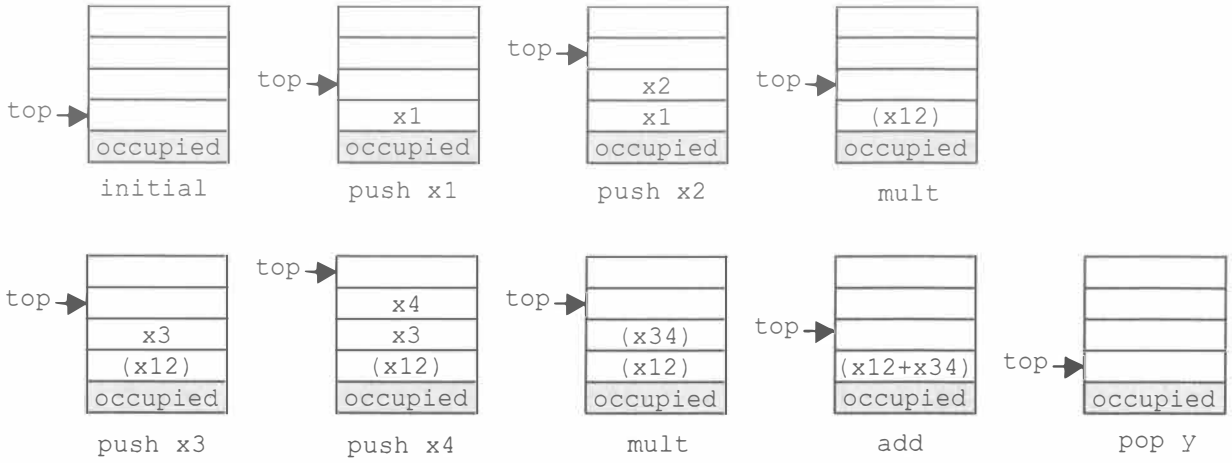


Figure A.4. Stack states during the execution.

Assume we want to organize a game involving a large number of teams and matches. In this example, we consider eight teams `t0`, `t1`, `t2`, ..., `t7`. We assume `t0` plays `t1` and `t2` plays `t3`, and so on. In the first round, the winners in round 1 will play in round 2, and the winners in round 2 will play in round 3, as shown in Figure A.5.

Now we will see how we organize the matches in an efficient way. We put the teams in a hotel where we have a large number of rooms to accommodate a large number of teams. However, the hotel is not close to the competition venue, and teams need to be transported, say by a bus, to the competition venue. To save time and eliminate unforeseeable traffic situations, teams will be transported (loaded) into waiting rooms close to the competition venue. The organization of the game facility is shown in Figure A.6. The game facility is analogous to a load-store architecture. The hotel corresponds to the memory, the waiting rooms correspond to the registers, the competition venue corresponds to the ALU, and the bus route corresponds to the data bus connecting memory and the registers.

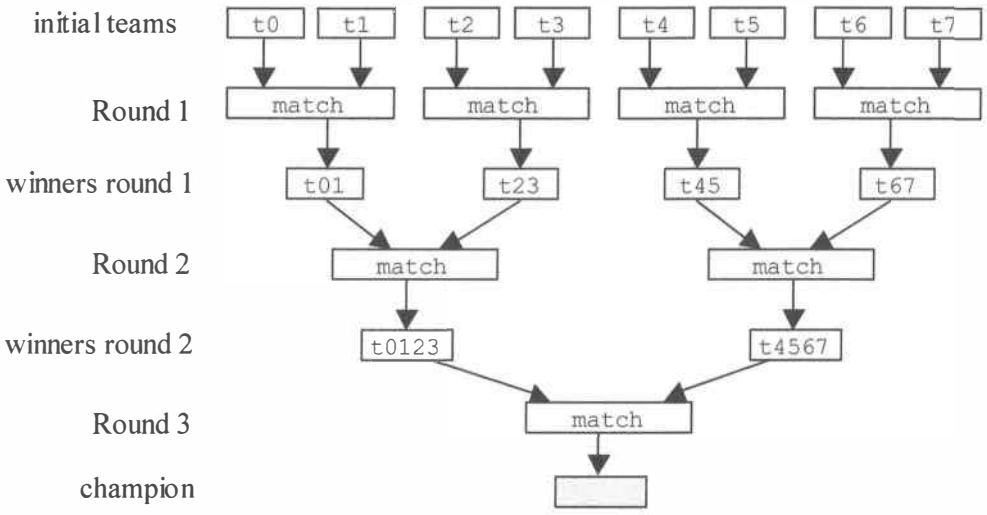


Figure A.5. The scheme of matches.

The organization of matches in Figure A.5 can be implemented by the following pseudo assembly language program. The comments after the double slash // explain what each instruction does. The instruction `match(Z, X, Y)` is, in fact, a procedure call and the definition of the procedure is given at the end of the program. The procedure simply chooses the larger value between `X` and `Y` and puts it in `Z`.

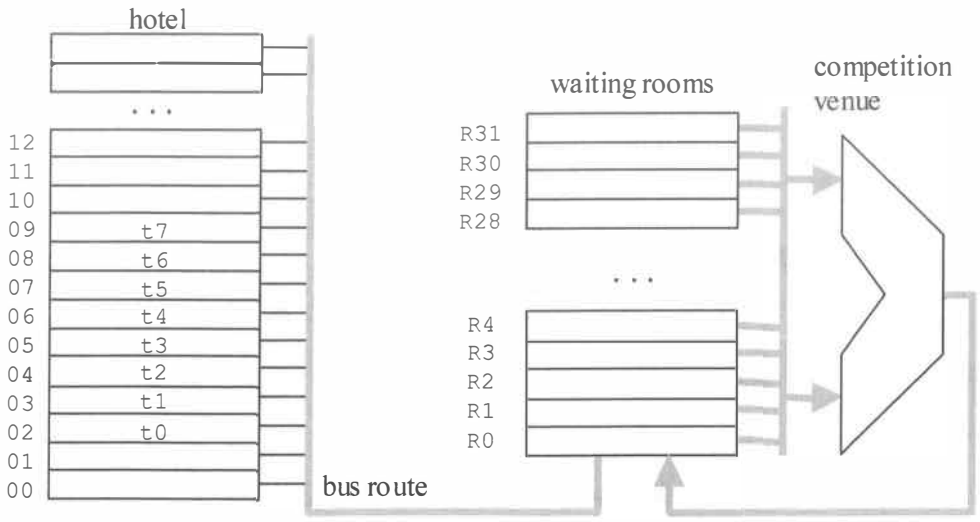


Figure A.6. The organization of the game facility.

```
// This program implements the organization in figure A.5. The teams (data)
// are preloaded in the memory locations 21, 22, 23, 24, 25, 26, 27, 28.
//round 1, four matches
load   R1, mem[21]    // load the content in memory location 21 into R1
load   R2, mem[22]    // load the content in memory location 22 into R2
match  R1, R1, R2     // call procedure match. The winner is put back into
R1

load   R3, mem[23]    // load the content in memory location 23 into R3
load   R4, mem[24]    // load the content in memory location 24 into R4
match  R3, R3, R4     // call procedure match. The winner is put back into
R3

load   R5, mem[25]    // load the content in memory location 25 into R5
load   R6, mem[26]    // load the content in memory location 26 into R6
match  R5, R5, R6     // call procedure match. The winner is put back into
R5

load   R7, mem[27]    // load the content in memory location 27 into R7
load   R8, mem[28]    // load the content in memory location 28 into R8
match  R7, R7, R8     // call procedure match. The winner is put back into
R7

// round 2, two matches
match  R1, R1, R3     // two winners in round 1 play and put winner in R1
match  R5, R5, R7     // other two winners in round 1 play and put winner in
R5

// round 3, one match, and store the final result
match  R1, R1, R5     // two winners in round 2 play and put winner in R1
store  R1, mem[20]    // store the champion in memory location 20

// the "match" instruction is implemented as a procedure
```

```
match(Z, X, Y):      // match procedure needs 3 parameters
if X > Y branch G    // if X > Y, then jump to label G below
move Z Y             // Y is greater than X, put Y in Z
branch L             // unconditionally jump to L
G: move Z X          // X is greater than Y, put X in Z
L: return Z          // The larger value is put in Z
```

A.3 Subroutines and local variables on stack

Most operating systems today are capable of multitasking. A subroutine (procedure or function) may be interrupted before its completion and be called by a second caller (reentrance). A subroutine may also call itself (recursion). In both cases (reentrant) and (recursive), the subroutine needs to offer a separate workspace (local variables) for each occasion of its execution. In other words, each time a subroutine is called, a new workspace must be created. In fact, it makes no difference at the assembly language level whether the call is a reentrant or recursive call.

At the assembly language level, local variables are implemented by a **stack frame** within the stack and are accessed through the **frame pointer**. A stack is usually a block of memory. As shown in Figure A.7, the stack is accessed through a stack pointer `sp`. The stack pointer is usually stored in a register. When a subroutine is called, the return address, the address of the instruction next to the subroutine call, will be stored onto the stack and `sp` incremented. Then a stack frame (a block of memory) will be created and a register is used as the frame pointer `fp` to access the memory locations in the frame. The codes below illustrate the process of a subroutine call:

```
stack[sp] = PC; // store return address onto stack. PC: program counter.
sp++;           // increment stack pointer
fp = sp;        // set frame pointer
sp = frame_size; // create frame of frame_size
```

Then we have memory space between `stack[fp]` and `stack[fp+frame_size]` for local variables, as shown in Figure A.7. When the subroutine completes and returns to the caller program, the following operation will be executed to return the stack to the state before the subroutine call:

```
sp = fp;             // deallocate the stack frame
PC = stack[--sp]     // restore the PC and thus
                     // the control will return to the caller
```

To support reentrant and recursive subroutine calls, the frame creation and deletion described above can be executed repeated and, thus, create nested frames on the stack, as shown in Figure A.8.

In fact, the value of the frame pointer associated with each reentrance (recursion) will be stored in its stack frame, so that we only need one register for the frame pointers, no matter how many times the subroutine is called.

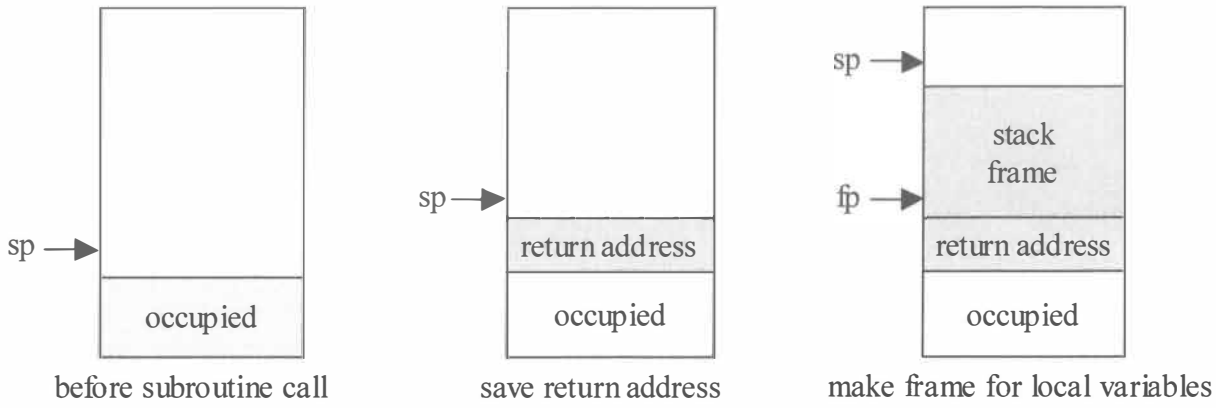


Figure A.7. A stack frame is created for local variable in a subroutine.

In summary, we discussed the following issues in this appendix:

- The major components of a computer system and the basic computer architectures.
- How to write assembly language programs on different architectures and how to write imperative programs: store data in memory, move data into registers, manipulate the data in ALU, put the result back into the register, and finally store data back into the memory.
- An example of an assembly language program on a load-store architecture.
- The process of subroutine calls and how stack frames are created for local variables in reentrant and recursive occasions of the subroutine.

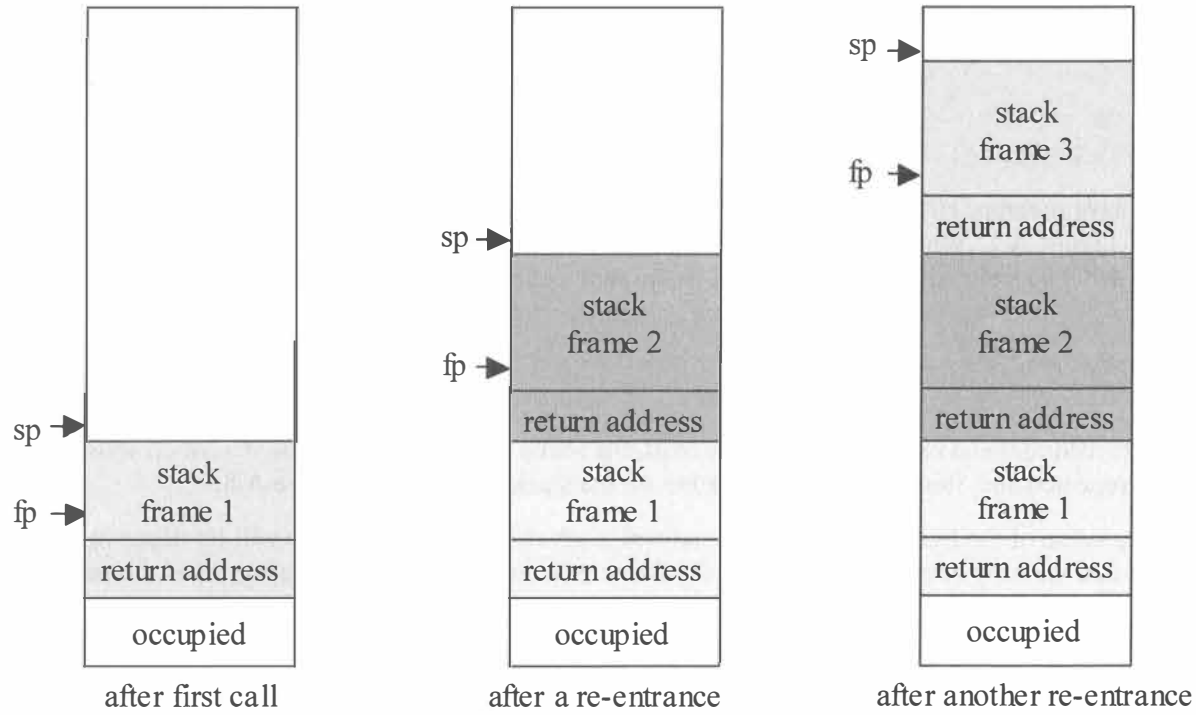


Figure A.8. Multiple reentrances of a subroutine.

Appendix B

Programming Environments Supporting C, C++, Scheme, and Prolog

Operating system is the interface between human and computer. It oversees operations of a computer, including storing and retrieving files (file management system), scheduling programs for execution, and coordinating the execution of programs. It also supports the development environments of computer programs by running different compilers, linkers, and runtimes. In this appendix, we will discuss the Unix/Linux and Windows operating systems, and the programming environments for C, C++, C#, Scheme, and Prolog that are discussed in this book.

B.1 Introduction to operating systems

A computer system without software is useless. Software can be roughly divided into two kinds: the system programs, which control the operation of the computer itself, and the application programs which solve problems for their users.

The most fundamental system program is the operating system, which manages all system resources and provides the base upon which the application programs can be written and executed. The goal of an operating system is to provide an environment in which users can execute programs in a convenient and efficient manner.

Initially the operating system is defined as a **monolithic resource manager**. Under this definition, a computer is viewed as a set of resources such as processors, memory, devices, and files that have to be multiplexed among competing tasks or jobs. In this context, the main problem solved by the operating system is the orderly sharing and protection of resources: offering computing tasks to share common machine resources while guaranteeing the necessary protection of tasks from one another. Control of sharing and protection is based on the maintenance of detailed status information about resource usage. The status information must be constantly updated by the operating system, which keeps track of all resources within one large monolithic program.

As computer capacity and application domains grew and computer resources required more sophisticated management, the maintenance and synchronization of resources usage and status information became more and more complex. Operating systems designed as monolithic resource managers became less and less tractable, hard to understand, hard to maintain, and hard to test.

The newer operating systems are considered an **extension to hardware**, providing the users with a **virtual machine**. A computer system is treated as structured classes of resources, where each class of resources is managed by a separate set of programs.

Generalizing the concept that the operating system can be viewed as a collection of software modules which turn real resources into virtual ones, one can view the entire operating system as one piece of software that

turns the real computer, including all resources, into one virtual machine composed of virtual resources, as shown in Figure B.1(a). In this perspective, the operating system is nothing more than a software extension to the computer hardware that makes the hardware more amenable to user programming.

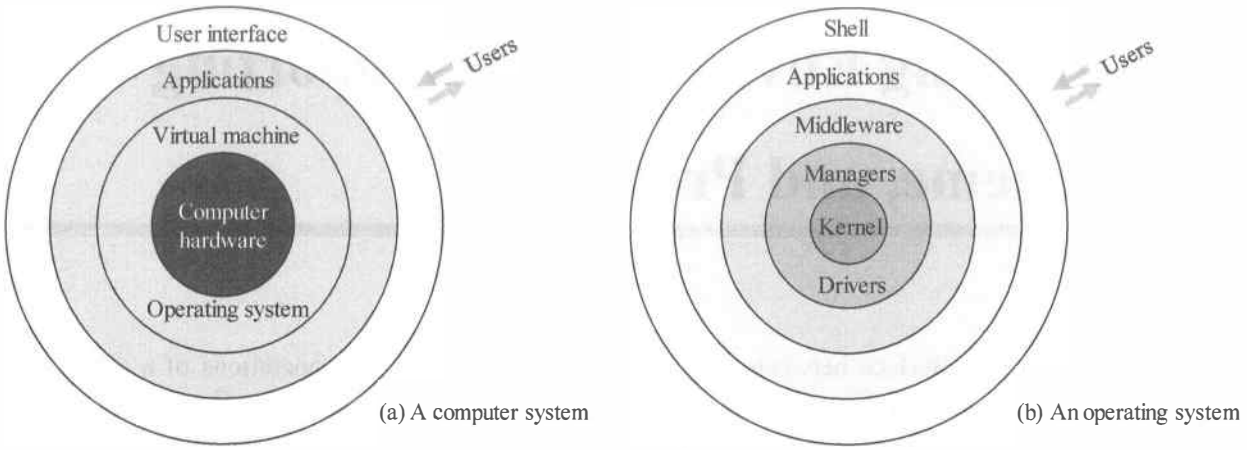


Figure B.1. Computer system and operating system.

In Figure B.1(b), the layers of the operating system are elaborated. The core of an operating system is its kernel. Different device managers and drivers are built on the kernel. Different middlewares can be built on the managers and drivers to facilitate the communication and execution of different types of applications. The operating system shell facilitates the human user interface.

- Kernel: Performs basic required functions, including disk file management, memory management, and processor management for task schedule and dispatch. The kernel of an operating system is not replaceable.
    - File manager: It manages the directories (or folders) in the secondary storage, typically, disks. It allows users to create, remove, change directories and access files in the directories. A file is accessed through a directory path: A sequence of directories within directories. For example, DOS (Disk Operating System) is basically a file manager, as the shell and other kernel mangers are very simple. DOS runs one program at a time, and thus memory manager is simple.
    - Memory manager: It allocates space for each program in the main memory. It may create the illusion that the computer has more memory than it actually does through virtual memory. It shifts blocks of data (pages) back and forth between the main memory and secondary storage (disk). Memory manger is complex in multitasking and multiprocessor systems. To improve memory access speed, a computer system also uses cache to buffer the frequently accessed data.
    - Task scheduler and dispatcher: Figure B.2 shows an example of task states and state transitions in an operating system. The task scheduler adds new tasks to the task queue and removes completed tasks from the queue. The dispatcher controls the allocation of time slices to the tasks in the task table (the tasks in ready state). The other state transitions can be controlled in the task programs.
- **Device managers and Drivers** are responsible for the peripherals attached to the computer systems, such as printer, monitor, mouse, keyboard, etc. Peripherals and their drivers can be installed and uninstalled by users.
- **Middlewares**: are software packages installed to map a general operating system to a system for more specific purposes. An embedded system is often implemented by installing a middleware to map the embedded applications to its sensors and actuators. For example, a VIPLE middleware is installed on a Linux operating system to facilitate communication and data interpretation between

ASU VIPLE software and the devices (sensors and motors) attached to the hardware (http://neptune.fulton.ad.asu.edu/VIPLE/).

- **Applications**: A special purpose operating system has a set of preinstalled applications to fulfill the given missions. A general purpose operating system allows users to develop and to install applications.
- **Shell**: The human user interface responsible for communicating with users. There are text-based shells, such as DOS and Linux, and graphical user interface (GUI), such as Windows, iOS, and Android.

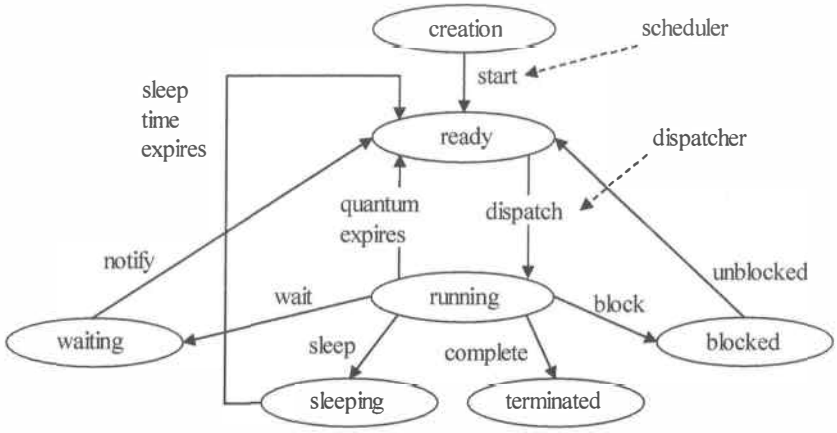


Figure B.2. States and transitions between the states in an operating system.

There are different types of operating systems, developed at different times and serving different types of applications.

Batch operating systems draw their name from the way in which work is submitted to them in batches of cards. A batch of cards defines a job. A job consists of a related set of tasks such as the compilation, loading, and execution of a program. Typically, a batch of cards includes control cards and program cards. Control cards are commands to the operating system that tell it what to do about the program cards that follow. For example, compile only; or load, execute, and punch the results; or pass as input to some application program residing in the file system. **Remote batch systems** are distinguished from batch systems by the simple fact that the jobs are submitted not at the site of computer, but at a remote card reader that is connected via a network to the batch system. A remote card reader is usually installed together with a printer, so the users can obtain their results at the site where the jobs are submitted. Although off-line preparation of jobs continued for some time, it was quickly replaced in most systems. Disk systems became widely available and greatly improved on off-line operation.

Disk operating systems. A computer can, while processing one job, simultaneously devote a very small amount of its CPU to (or use an I/O processor for) reading in one or more jobs and writing them onto the disk. Thus, when the current job is completed, the computer can rapidly retrieve another batch of work from its disk (reading from a disk is much quicker than reading a batch of cards). Similarly, when a job requires the printer to output a line, the computer first writes the line to the disk. The line is printed later when the printer becomes available. This form of overlap of CPU and I/O operations is called **spooling** (Simultaneous Peripheral Operation On-Line). Obviously, spooling which keeps both CUP and I/O devices working at higher rates has a direct beneficial effect on the performance of the system.

Unlike spooling, buffering overlaps the I/O of a job with its own computation. The advantage of spooling over buffering is that spooling overlaps the I/O of one job with the computation of the other jobs.

Time-sharing operating systems. Among the online systems, one must distinguish between shared multiuser systems and private single-user systems on the one hand, and between user-programmable and

non-programmable systems on the other hand. The terms user-programmable and non-programmable mean that the users of the operating system are allowed, and not allowed, to write user programs under the operating system environment, respectively. In the first category, shared and programmable systems are called time-sharing or interactive systems. The term **time-sharing** means that the processor time is shared among all users and the switches occur so frequently that the users can interact with each program while it is running. Thus time-sharing systems are also called **interactive systems**. The term time-sharing could be used for any system that serves several users at a time. Even batch systems may be time-shared. However, the rate at which a modern interactive system jumps from one user to another is perhaps 1000 times higher than that of a batch system.

Web operating system. The concept was developed at the University of California at Berkeley around 1999. It starts to challenge the desktop operating when service-oriented computing paradigm and web-based computing model become an alternative to the object-oriented computing paradigm and desktop-based computing model in recent years. The main idea is to connect to the web server through a browser, and all the computation and resources are on the server and thus the operating system is running on the server. The key technologies related to web operating systems include:

- Web-based computing concepts;
- Service-oriented architecture and service-oriented computing;
- Web applications in all domains, including business, finance, and society
- Web 2.0: web as computing platform and Web 3.0: semantic web
- Cloud computing, including, Software as a Service (SaaS), Infrastructure as a Service (IaaS), and, Platform as a Service (PaaS);
- Big data processing through parallel computing and parallel data management.

B.2 Introduction to Unix and C/C++ programming environments

This section introduces Unix operating system and GNU C and C++ programming environment under Unix and Linux operating systems.

B.2.1 Unix and Linux operating systems

Unix is a multitasking and time-sharing operating system. The design principles include:

- Simplicity: keep the operating system simple and have it support only a minimum number of functions
- Generality and orthogonality: A single method should serve a variety of purposes, for example:
    - the same system calls are used to read (or write) files and devices;
    - the same naming, aliasing, and access protection mechanisms apply to data files, directories, and devices;
    - the same mechanism is used to trap interrupts and processor traps.
- Portability: UNIX portability means at least three things:
    - Portability of UNIX itself from one computer model to another;
    - Portability of C programs from one computer model to another, where both computers are running the same version of UNIX;
    - Portability of C programs from one version of UNIX to another.
- File system: There are three kinds of UNIX files: ordinary files, directories, and special files. The files are organized as a multiple tree. An ordinary file is a sequence of bytes. A special file is either some type of device (e.g., tape and communication line) or a FIFO (first-in-first-out queue), which is a mechanism used to pass data between processes. A directory is a file which contains information on how to find other files.

- Hierarchy: UNIX can be viewed as being layered, as shown in Figure B.3, where user interface and programmer interface will be studied as operating system example in the remaining sections of this chapter.

	Human users			
Shell	User interface			
System programs	Commands, compilers, interpreters, and system libraries			
	Programmer's interface (system calls to the kernel)			
Kernel and managers	Signals Terminal handling I/O system Terminal drivers	File system Swapping Block I/O system Device drivers	Page replacement Demand paging Virtual memory	CPU scheduler Dispatcher
	Kernel interface to hardware			
Hardware	Terminal controllers Terminals	Device controllers Disk, printer, etc.	Memory controllers Cache, memory	Processors

Figure B.3. Unix operating system hierarchy.

Unix is an operating system standard used and supported by many companies, including Solaris, Intel, HP etc., as workstation, mainframes, and server operating system. These are typically bigger computer systems.

Linux is a free and open source version of Unix. It is typically used in smaller systems, such as PC and embedded systems. Many different commercial products have been developed on Linux, including:

- Android (Google, U.S.). It has been used in mobile applications.
- Redhat (Red Hat, U.S.). It has been widely used as general purpose operating systems.
- Ubuntu (Canonical Ltd, U.K.). It has been widely used as general purpose and embedded operating systems.
- Yocto (https://www.yoctoproject.org/). It is used in many embedded applications, such as Internet of Things (IoT) and robots.

B.2.2 Unix shell and commands

In Unix, the built-in system programs, as well as the user-written programs, are usually executed by a command line interpreter, which is called **shell**, as it surrounds the operating system. The shell is not permanently resident in the main memory like the kernel and is executed as a user process. Users can write their own shells. There are several shells that are widely used, e.g., the Bourne shell, the Korn shell, and the C shell.

A shell command line consists of a **command name** (the name of any executable file), followed by options and a list of arguments separated by blanks. During execution, the shell loads the file specified in the command, and makes the command arguments accessible to it. Then a child process is created to execute the command while the parent process (the shell) waits for the child to terminate. Upon termination of the child process, the shell types its prompt to the terminal to indicate that the user may type the next command. For example, commands

rm temp will remove (delete) the file temp;

ps will list all processes which are running;

kill PID will terminate the running process with process identifier PID;

test -f temp will return a True, if file temp exists and is not a directory;

test -w temp will return a True, if file temp is writable.

A UNIX shell can be used as a programming language. The execution of a command is analogous to a subroutine call. A file of commands, called a **shell script**, can be executed like any other simple command. Current shell command languages include shell variables and the usual high-level programming-language control structures like if-then-else, case, while, for, etc. For example, the executable file

myScript

```
if      test -f temp
then    test -r temp
else    rm      temp
fi

ps

kill 27491
```

can be executed as if myScript is a simple command.

In this section, we will briefly explain the basic Unix commands that we may need to run our C and C++ programs. We first explain a few commands that we will use immediately to get started. Then most of the commands that you may need are listed in Table B.1 at the end of the section.

Current directory

To find what directory you are currently in, you simply type

```
pwd
```

pwd stands for "print working directory." The output shows the path from the root directory to your current working directory.

Creating a new directory

To create a new directory within the current directory, type command

```
mkdir newdirectoryname
```

Deleting a directory

To remove (delete) a nonempty directory in the current directory, type command

```
rmdir directoryname
```

Note that the directory to be removed must be empty. If the directory is not empty, you can enter the `directory` and then use the commands `rm */*`, `rm *`, `rmdir */*`, `rm *`, etc., to remove all the files and directories contained in the directory before removing the directory.

List directory and files

To list all the subdirectories and files in the current directory, type

```
ls     ; only list directory or file names
ls -l        ; will list details on directory and file, e.g., access permission
```

Change directory

To change into a different directory within the current directory, type

```
cd directoryName      ; enter the directory
cd ..                 ; return to the parent directory
```

Type `cd`, a space, and two full stops (periods) to move one level up in the directory structure or back into the parent directory.

Unix online manual

To read the description of a command, type

```
man commandName
```

For example, `man cd`. If you do not know the exact command, you may try `apropos keyword` and a list of possible commands relating to the keyword will be printed. A full help description of commands listed above can then be displayed using `man`.

Table B.1 below lists most commands that you may need. You can use the online manual to check the options available to each command and the full description of each command. The commands are alphabetically sorted.

Unix command	Description
`cat` [options] `file`	concatenate and print on standard output
`cd` [directory]	change directory
chgrp [options] group file	change the group of the file
`chmod` [options] `file`	change file or directory access permission mode
chown [options] owner file	change the ownership of a file
cmp [options] file1 file2	compare two files and list where differences occur (text or binary files)
compress [options] file	compress the file and save it as `file.z`
cp [options] file1 file2	copy `file1` into `file2`
`date` [options]	report the current date and time
diff [options] file1 file2	compare the two files and display the differences (text files only)
`df` [options] [resource]	report the summary of disk blocks free and in use
`du` [options] [`directory` or `file`]	report amount of disk space in use
`echo` [text string]	echo the text string to stdout
`ed` or `ex` [options] `file`	Unix line-oriented text editor
`emacs` [options] `file`	full-screen editor
`file` [options] `file`	report the file type
`find directory` [options] [actions]	find files matching a type or pattern
finger [options] username@hostname	report information about users on local and remote machines
ftp [options] hostname	file transfer using file transfer protocol

gcc [options] file g++ [options] file	gcc is used to compile C and C++ program g++ is used to compile C and C++ program, but C file will be treated as C++ program
gplc [options] file	GNU Prolog compiler
grep [options] 'search string' argument	search a file for a pattern
gzip [options] file gunzip [options] file	compress or uncompress a file. Compressed files are stored with a .gz ending
head [-number] file	display the first 10 or the given number of lines of a file
hostname	display the name of the current machine
javac [options] file	Java compiler
kill [options] [pid#] [%job]	the process with the process id number (pid#)
ln [options] source_file target	link the source_file to the target
login logout, or exit	sign on exit
lpq [options]	show the status of print jobs
lpr [options] file	print to defined printer
lprm [options] cancel [options]	remove a print job from the print queue
ls [options] [directory or file]	list directory contents or file permissions
man [options] command	show the manual (man) page for a command
mkdir [options] directory	make a directory
mv [options] file1 file2	change file name from file1 into file2
passwd [options]	change your login password
pico [options] file	text editor
ps [options]	show status of active processes
pwd	print working (current) directory
rcp [options] hostname	remotely copy files from this machine to another machine
rlogin [options] hostname	login remotely to another machine
rm [options] file	remove a file
rmdir [options] directory	remove an empty directory
sort [options] file	sort the lines of the file according to the options chosen
tail [options] file	display the last few lines (or parts) of a file
telnet [host [port]]	communicate with another host using telnet protocol
tr [options] string1 string2	translate the characters in string1 from stdin into those in string2 in stdout
uncompressfile.Z	uncompress file.Z and save it as a file
uniq [options] file	remove repeated lines in a file

uudecode [file]	decode a uuencoded file, recreating the original file
uuencode [file] new_name	encode binary file to 7-bit ASCII, useful when sending via email, to be decoded as new_name at destination
`vi` [options] `file`	visual, screen-oriented text editor
`wc` [options] [`file`(s)]	display word (or character or line) count for `file`(s)
who or w	report who is logged in and what processes are running

Table B.1. Unix command table.

B.2.3 Unix system calls

A system call in an operating system provides public library functions for the users to access the resources managed by the operating system, including accessing the file system, controlling the processes, and implementing interprocess communication. The system calls are running in the kernel mode. All Unix system calls are included as C library and can be called just like calling C library functions. Table B.2 list the common Unix system calls.

Category	Function description	System calls
File management	Creating file and file buffer	create(); open(); close()
	File stream read and write	read(); write();
	File random access	lseek();
	File linking and unlinking	link(); unlink
	File status	stat(); fstat()
	Security and access control	access(); chmod(); chown; umask;
	Device control	ioctl();
Process management	Process creation and termination	exec(); fork(); kill(); exit();
	Process ownership and grouping	getuid(); geteuid();getegid();
	Process ID access	getid(); getppid();
	Change process working directory	chdir();
Interprocess communication	Pipelining	pipe();
	Messaging	msgget(); msgsnd();
	Process synchronization	Wait(); signal(); alarm();
	Semaphores	semget(); semop();
	Shared memory communication	shmget();shmat();shmdt()

Table B.2. Unix system calls.

All the input and output functions, as well as the file operations we use in C and C++ are calling the operating system calls to complete their jobs, as only system calls can access the file system, devices, and processes in the system, as discussed in Chapters 2 and 3.

Like all modern operating systems, Unix supports multitasking programming. We will use system calls to illustrate the implementation of parallel computing in Unix. We start to use the process creation system call:

int fork( );

When fork() is executed in a process, it creates a new (child) process, which is basically a copy of its parent process: The same program code, including the fork() statement, status, user-data, and system-data segments, will be copied. The only difference is that the two processes (parent and child) will receive different return values from the system call fork().

The child receives a 0 return value, whereas the parent receives the process-ID of the child. If the parent receives "-1" value, an error has occurred in creating a child process. fork() has no arguments, the caller could not have done anything wrong. The only cause of an error is resource exhaustion (e.g., out of memory). In the exception handling, the parent process may want to wait a while (with a sleep call) and try later again.

Another Unix system that usually works with fork() in tandem is the exec(parameters). Usually, the child process executes an exec(parameters) system call after the return of fork(), whereas the parent either waits for the child to terminate or goes off to do something else. Figure 2.12 shows a typical use of fork() and exec(parameters).

The exec(parameters) system call reinitializes the child process from the given program and data files in the parameters. The steps of the creation and reinitialization of the new process are marked on the lower part of the diagram in Figure B.4.

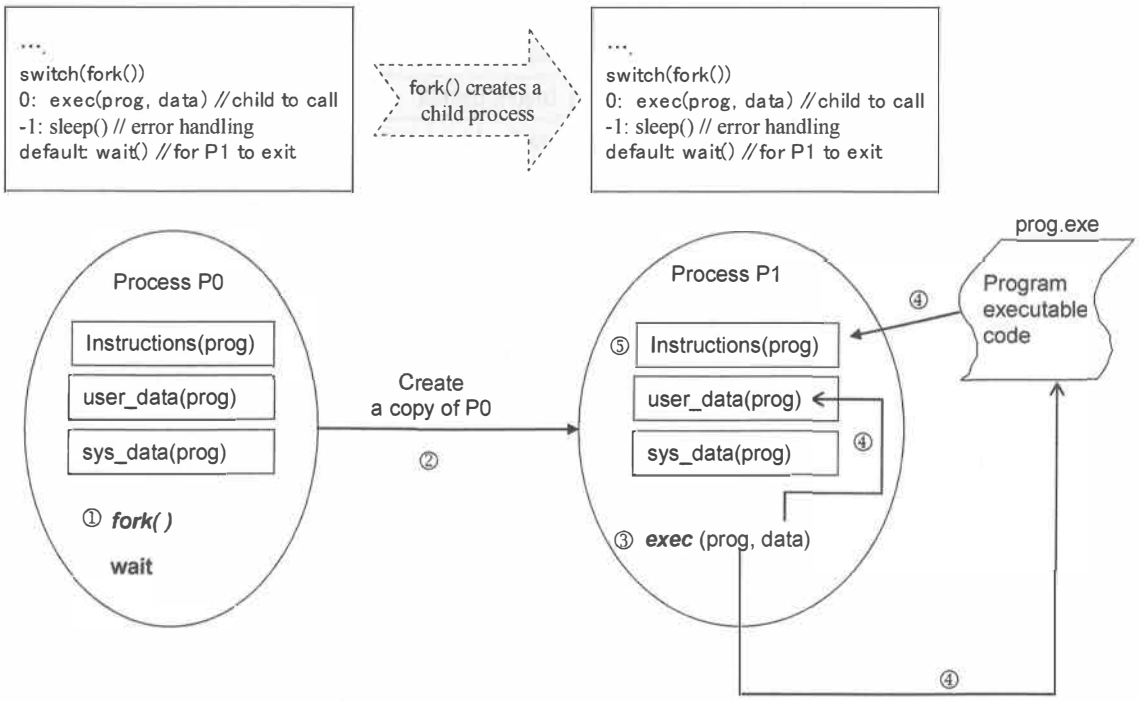


Figure B.4. A typical application of system calls fork() and exec()

Different versions of the system call are used to facilitate different parameter formats. Below are C specifications of two versions of the system call:

```
// Version 1: list all parameters
int execl(path, arg0, arg1, ..., argn, null )
  char *path;    // path (location) of program file
char   *arg0;    // first argument (program file)
char   *arg1;    // second argument
. . .
```

```
char   *argn;    // last argument
char   *null;    // null indicates end of arguments
// Version 2: Use a file name and an array as the parameters
int execvp(file, argv )
  char*file;    //program file name
char   *argv[ ]; // pointer to the array of arguments
```

The following C program shows a simplified design of a command line interpreter (CLI), which waits for a command to be entered, and starts the program associated with the command as a child process of itself:

```
#include <fcntl.h> // Command Line Interpreter
static void main (int argc, char *argv[ ]) {
  while (TRUE) { // read, execute a command and wait for termination
      read_command(argv);
      // read command name in argv[0] and data in argv[1] ... argv[argc-1]
      switch(fork()) {
          case -1:
            printf("Cannot create new process \n");
            break;
          case 0:
            execvp (argv[0], argv);
            // The execvp function should never return. If it returns
            printf("Cannot execute \n"); // an error must have occurred
            break;
          default:   // CLI process itself will come to this case
            if (wait(NULL) == -1)
            printf("Cannot execute wait system call \n");
            // Parent process receives the PID of child process
            // and then waits for the termination of child
    }
  }
}
```

B.2.4 Getting started with GNU GCC under the Unix operating system

GNU GCC is a C/C++ development environment under the Unix operating system. We assume that you know how to write a C/C++ program and have read the previous section on Unix commands.

First, you need to write your C/C++ program. You can write the program on your desktop computer and SHH or PuTTy software to transfer the program to the Unix operating system. You can also write the program in Unix using different text editors available on Unix, such as pico, nano, vi, and vim.

A short introduction to Unix and basic Unix commands are given in Section B.1. If you use the SSH Secure Shell, you can start a console window by choosing the menu item "`Window`" and then choosing "`New Terminal`." To enter and edit your program, you can use any Unix text editors (e.g., `pico` or `vi`). You enter, for example,

pico hello.c

A new text window will be opened and you can start to enter your program.

```
// My first program, file name hello.c
```

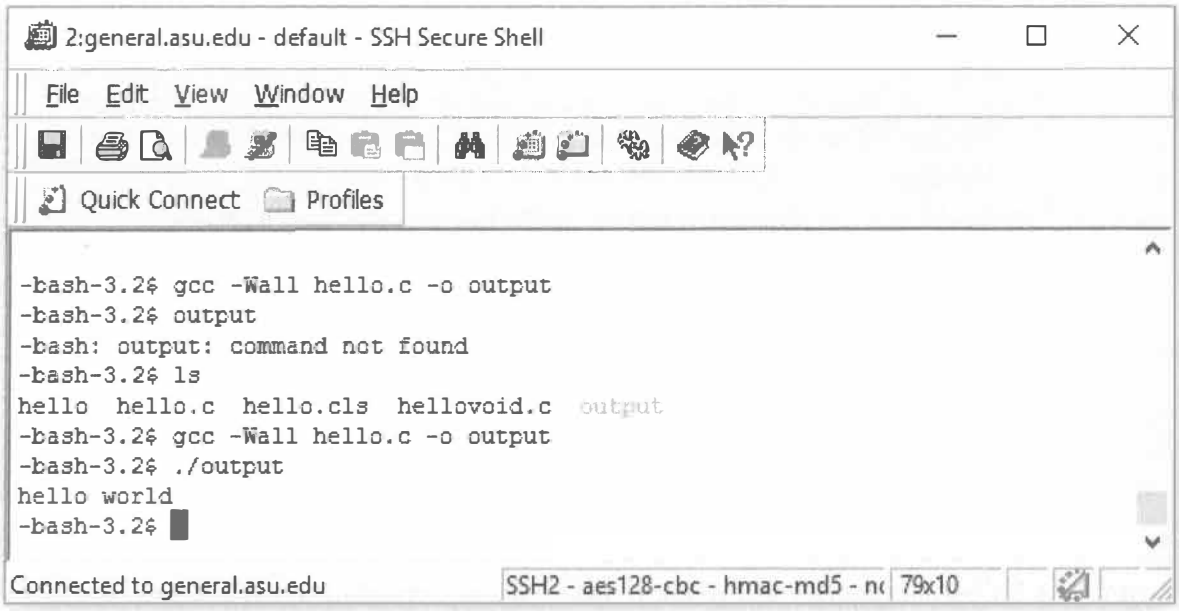
```
#include <stdio.h>   // the library functions standard I/O will be used
int main( ) {
   printf("hello, world\n");
   return 0;
}
```

After you have entered the program, type `ctrl-X` to exit the `pico` editor. Then you can use the GCC compiler to compile your program.

You can execute the command

```
gcc -Wall hello.c -o output
```

which will take your 'hello.c' file and output an executable called "output." It is a good idea to add the **–Wall** flag, which enables the compiler's warning messages, to learn to generate better code. The **–o** flag is also optional, so if you choose not to specify an output file name, it will name the file **a.out** by default. We run **gcc –Wall hello.c –o output** in the example below, and we see that the compiler provides several warnings messages, but no errors, so the code is successfully compiled.

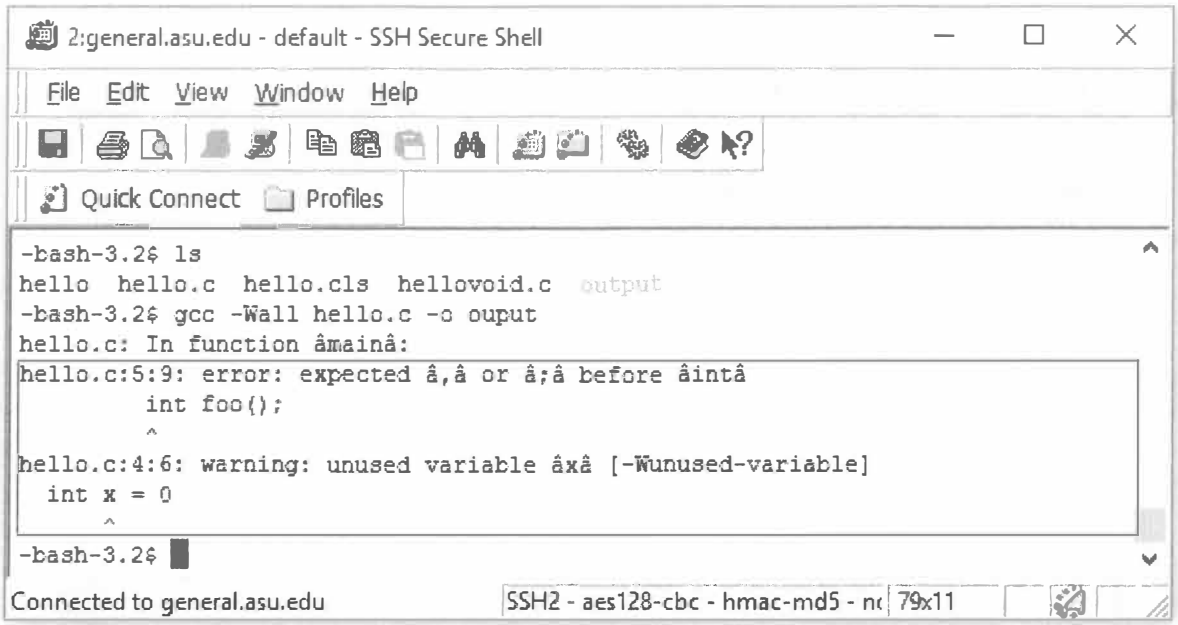


Now, to run the executable file ''output,'' you simply type the following command:

```
./output
```

It will execute the program and print the output.

If your program has a syntax error, such as a missing semicolon, you will receive an error message, as seen below:



Note, you can also use C++ compiler g++ to compile your C program, for example:

```
g++ hello.c -o output
```

If no compilation errors are found, the command will create an object code (machine code) and store the code in the file `output`. To run the program, you type

```
./output
```

The program will be executed and the output printed on the screen.

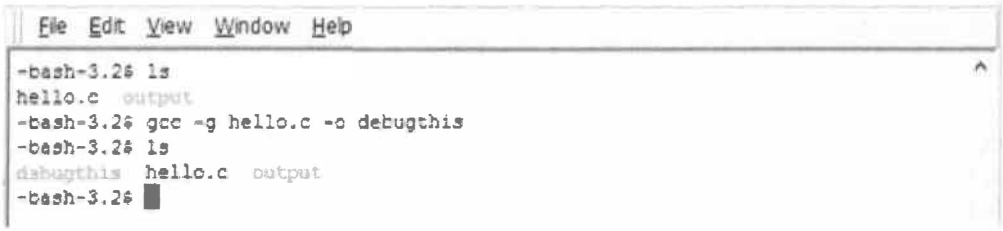
The compiler g++ can compile both C and C++ programs. The name of a C program must have the extension `.c` while the name of a C++ program must have the extension `.cpp`.

Note, gcc can also be used to compile both C and C++ programs. The difference is that g++ will treat C program also as C++ program, while gcc will not treat C program as C++ program.

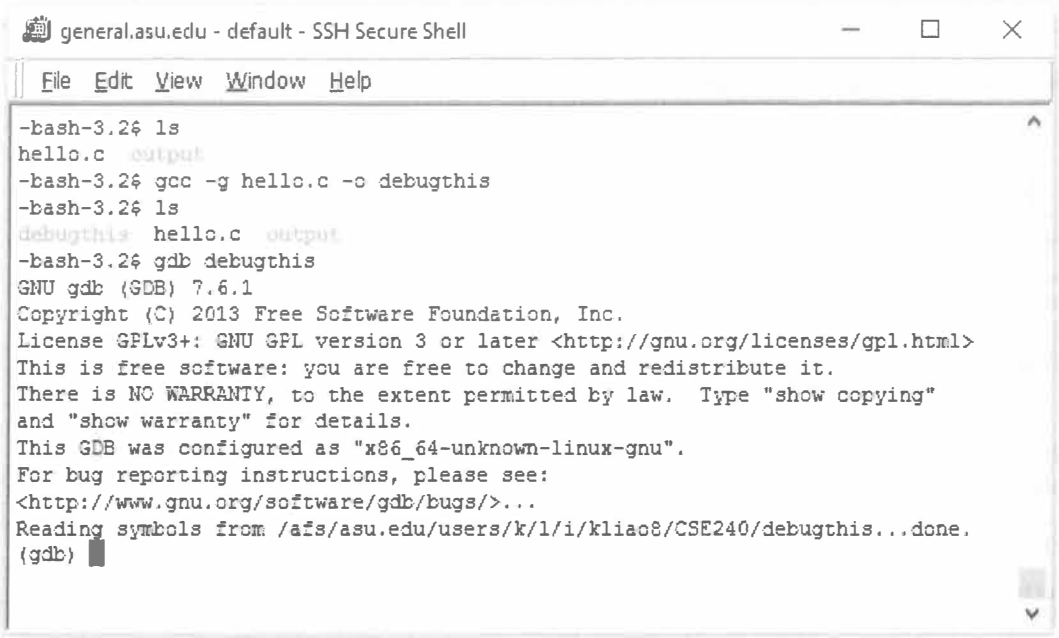
B.2.5 Debugging your C/C++ programs in GNC GCC

Now, we will go over how to perform a basic debugging in GNU GCC.

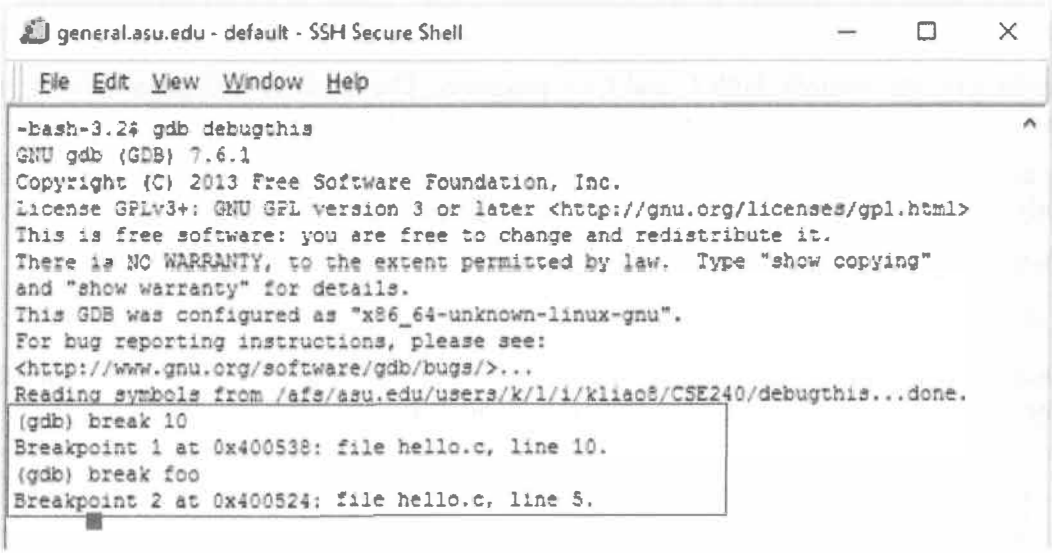
1. In order to debug our program, we first compile our code with debugging information enabled using the **–g** flag. In other words, **gcc –g hello.c –o debugthis**.



2. Now, we can run the debugger using **gdb debugthis**.



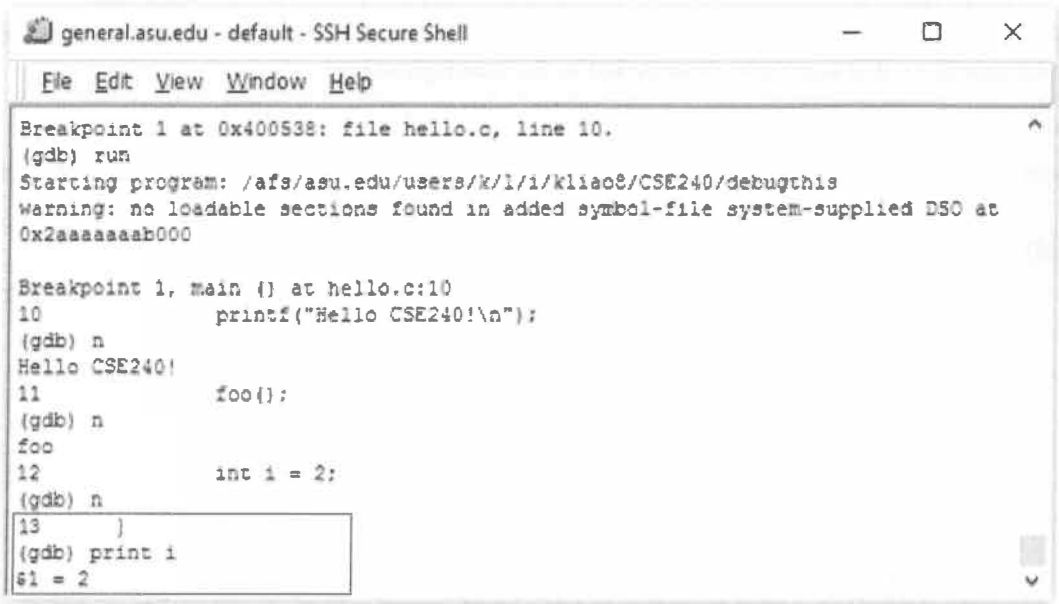
3. Notice that we can open multiple SSH/PuTTy terminals at once, so it might be a good idea to have the code open in another terminal and the debugging process open in another terminal. Once we are in **gdb**, we can add breakpoints into our program, which stipulate where the program should stop executing. We can add breakpoints for line numbers and also function calls. Below, we add a breakpoint to line 10 and the function call foo() using the commands **break 10** and **break foo**, respectively.



4. Once our breakpoints are set, we can run the program using the command **run**, and it should stop once it hits a breakpoint. At this point, we can begin executing the program line by line. One way to do this is "step-into," which will execute a program, including all function calls, line by line. We can step into by using the **s** command, as shown below where we step into the foo() function.

5. Another option is to use "step-over" or **n**, which will skip over function calls when executing line by line.

6. Once a line is executed, we can also print variable values using **print i**, for example.



7. Finally, to quit the debugger, we can use **quit** and then enter **y** if the program is still running.

Memory leak detection tools are also in GNU GCC. For example, Valgrind is a GNU GCC/G++ memory leak detection tool. Its details can be found at:

http://www.valgrind.org/docs/

The tool is installed in GNU GCC/G++ in the ASU general server. The following commands can be used to include the memory leak detection tool

```
g++ -o myProg -g myProg.cc // Compile
// The use of -g allows exact line numbers in error messages
valgrind --leak-check=full --tool=memcheck ./myProg
```

B.2.6 Frequently used GCC compiler options

GCC has many available options. We list a few commonly used options for convenience in Table B.3. Multiple options can be applied in one command.

Command	Compilation option description
gcc hello.c	No option. It generates default executable code: a.out
gcc hello.c -o hello	-o generates executable code: hello
gcc -Wall hello.c -o hello	-Wall generates warning
gcc -E hello.c > hello.i	-E generates the preprocessor (macro) output and redirects it to hello.i
gcc -S hello.c > hello.s	-S generates the assembly code output and redirects it to hello.s
gcc -C hello.c	-C generates object file without linking and save it in hello.o
gcc -Wall -v hello.c -o hello	-V prints the steps of the compilation steps

Table B.3. Frequently used GCC compilation options.

B.2.7 C/C++ operators

Table B.4 lists the C and C++ operators, their precedence, description, and associativity. The table also indicates the operators that cannot be overloaded in the description part.

[Reference: https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B].

Precedence	Operator	Description	Associativity
1 (highest)	::	Scope resolution (C++ only). No overloading.	None
2	++	Postfix increment	Left-to-right
	--	Postfix decrement	
	( )	Function call	
	[ ]	Array subscripting	
	.	Element selection through value semantics. No overloading.	
	->	Element selection through pointer	
	typeid()	Run-time type information (C++ only)	
	const_cast	Type cast (C++ only). No overloading.	
	dynamic_cast	Type cast (C++ only). No overloading.	
	reinterpret_cast	Type cast (C++ only). No overloading.	
	static_cast	Type cast (C++ only). No overloading.	
3	++	Prefix increment	Right-to-left
	--	Prefix decrement	

	+	Unary plus (positive sign)	
	-	Unary minus (negative sign)	
	!	Logical NOT	
	~	Bitwise NOT (One's Complement)	
	(*type*)	Type cast.	
	*	Indirection (dereference)	
	&	Address-of (dereference)	
	sizeof	Size of objects. No overloading	
	new, new[]	Dynamic memory allocation (C++ only)	
	delete, delete[]	Dynamic memory deallocation (C++ only)	
4	.*	Member of object Contact selected by pointer-to-member name. No overloading.	Left-to-right
	->*	Pointer to member (C++ only).	
5	*	Multiplication	Left-to-right
	/	Division	
	%	Modulo that returns remainder	
6	+	Addition	Left-to-right
	-	Subtraction	
7	<<	Bitwise left shift	Left-to-right
	>>	Bitwise right shift	
8	<	Less than	Left-to-right
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
9	==	Equal to	Left-to-right
	!=	Not equal to	
10	&	Bitwise AND	Left-to-right
11	^	Bitwise XOR (exclusive or)	Left-to-right
12	|	Bitwise OR (inclusive or)	Left-to-right
13	&&	Logical AND	Left-to-right
14	||	Logical OR	Left-to-right
15	?:	Ternary conditional. No overloading.	Right-to-left
16	=	Direct assignment	Right-to-left
	+=	Assignment by sum	
	-=	Assignment by difference/subtraction	
	*=	Assignment by product	
	/=	Assignment by quotient	
	%=	Assignment by remainder	
	<<=	Assignment by bitwise left shift	
	>>=	Assignment by bitwise right shift	

	&= ^= |=	Assignment by bitwise AND Assignment by bitwise XOR Assignment by bitwise OR	
17	throw	Exceptions throw operator (C++ only)	Right-to-left
18 (lowest)	,	Comma	Left-to-right

Table B.4. C/C++ operators, their features, and overloading abilities.

B.2.8 Download programming development environments and tutorials

You can download a desktop copy of GNU GCC from the GNU home page:

http://www.gnu.org/

GNU GCC page:

`http://www.gnu.org/software/gcc/gcc.html`

Tutorials and references related to this section can be found at:

1. The Beginner's Guide to Nano, the Linux Command-Line Text Editor
   http://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/
2. Interactive Vim tutorial
   http://www.openvim.com/
3. Vim Tutorial
   https://linuxconfig.org/vim-tutorial
4. The ultimate Vim configuration: vimrc
   https://github.com/amix/vimrc
5. Compiling C and C++ Programs
   http://pages.cs.wisc.edu/~beechung/ref/gcc-intro.html
6. Tutorial of gcc and gdb
   http://cseweb.ucsd.edu/classes/fa09/cse141/tutorial_gcc_gdb.html
7. GDB Tutorial – A Walkthrough with Examples
   https://www.cs.umd.edu/~srhuang/teaching/cmsc212/gdb-tutorial-handout.pdf
8. Introduction to GDB a tutorial (video)
   https://www.youtube.com/watch?v=sCtY--xRUyI

B.3 Getting started with Visual Studio programming environment

In this subsection, we introduce the Visual Studio environments that support the development of C/C++, C#, and SOA programs.

You can download a free copy of Visual Studio Professional from Microsoft DreamSpark (https://www.dreamspark.com). You must be a registered student in order to download from this site. You can use Visual Studio for C, C++, and C# program development.

B.3.1 Creating a C/C++ project in Visual Studio

Visual Studio supports C, C++, and C#. C# and SOA programming in Visual Studio are discussed in Chapter 6. To start a C or C++ project, you can follow the following steps to set up your system properly.

- Start Visual Studio.
- Choose Visual Studio menu "`file`" – "`new`" – "`project...`": A "`New Project`" dialog box will pop up.
- Choose "`Visual C++ project`" in the window on the left-hand side, and then choose "`Win32 Project`" in the window on the right-hand side. Enter the project name. Assume your project name is "MyCprogram," in the text box below the two windows. Click OK.
- Under "`Project type,`" select "`Win32.`"
- Under "Templates," select "Console Application."
- Click next.
- A new window pops up. Click on "`application type`" and choose "`Empty Project`," as shown in Figure B.5.
- Click "`Finish`."

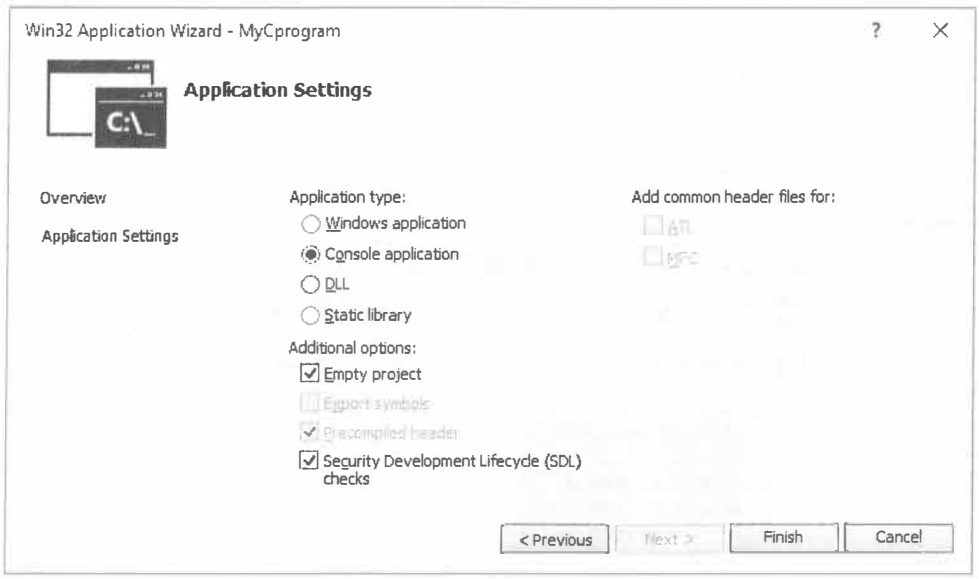


Figure B.5. Creating a new C/C++ project.

Now you have created an empty project that can be used to run your C/C++ programs. There are two possibilities for you to add your C/C++ program into the project: (1) You want to type your program in Visual Studio. (2) You have already saved your program in a .c file or a .cpp file.

Option 1: type your program in Visual Studio

Step 1. In your Solution Explorer, right click the folder "Source Files," choose Add → New Item..., as shown in Figure B.6:

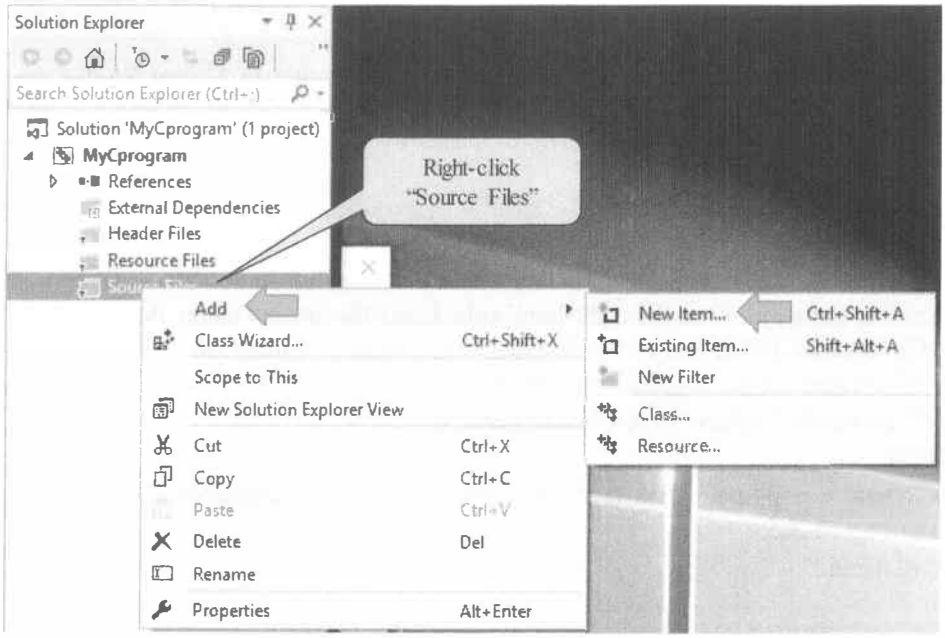


Figure B.6. Adding a source file into the project.

Step 2. As shown in Figure B.7, choose C++ File template. When you enter the program name, make sure you enter an extension .c if you want to write a C program. By default, it creates a C++ program.

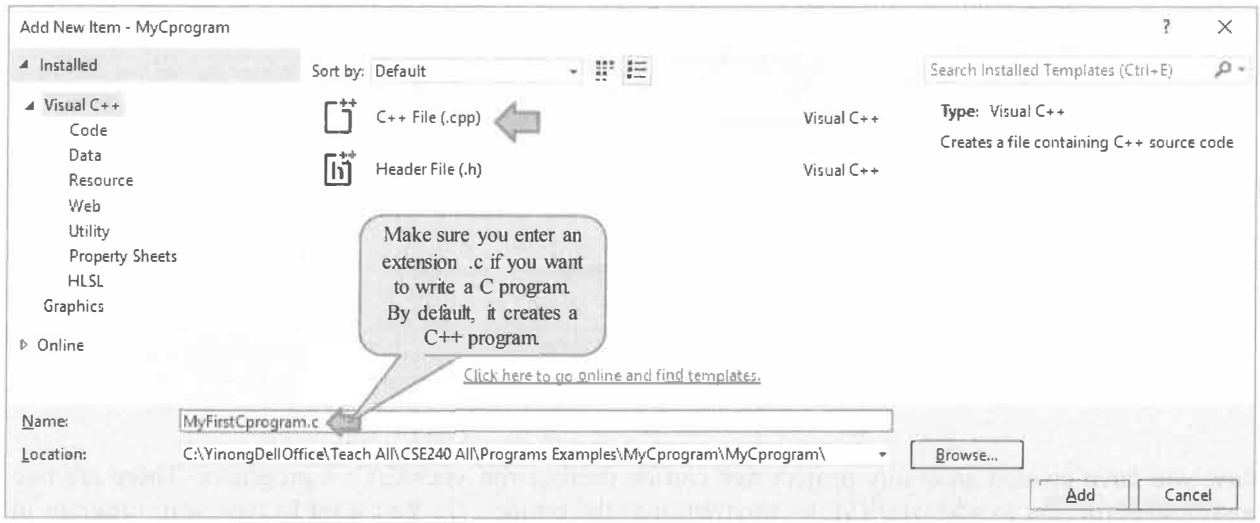


Figure B.7. Adding a C file by changing the extension to .c.

Step 3. Click Add. An empty C program file is created. You can start to write your program in the file, as shown in Figure B.8.

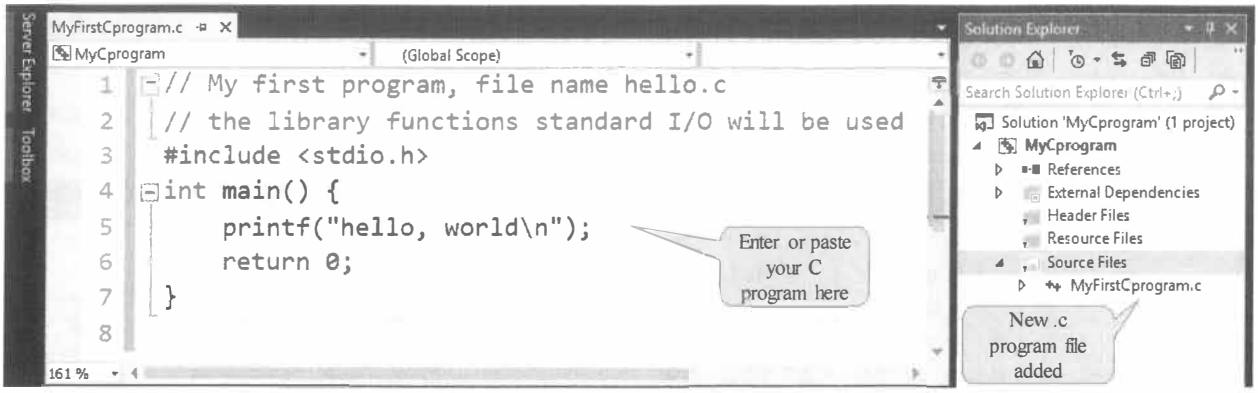


Figure B.8. Typing your C program in the new file.

Option 2: Open an existing C/CPP program

Assume that you have created a program called `MyFirstCProgram.c` or `MyFirstCPPProgram1.cpp` and now you can add your program into the project following these steps:

Step 1. In your Solution Explorer, right click the folder "Source Files," choose Add → Existing Item..., instead of Add → New Item.

Step 2: Browse to the location where you have saved your .c or .cpp file and open the file.

Using Visual Studio, you can easily implement GUI (graphic user interface) in C++ and in C#: When you create a new C++ (or C#) project, instead of selecting an empty project, you can select "Windows Forms Application" as the project type. Then, the GUI tools and library will be included in your project stack. You can use the provided tools and library functions, such as buttons (for mouse click inputs), textboxes (for text inputs), and labels (for outputs), to implement different kinds of GUI. See Chapter 6 for more details.

B.3.2 Debugging your C/C++ programs in Visual Studio

Visual Studio has the most powerful debugging capacity. As shown in Figure B.9, you can simply click the left edge of the program panel to add a break point. You can choose Debug → Start Debugging to start running the program. The program will stop at the first break point. You can click "Continue" to the next break point. You can also click the single-step execution to move the next statement. There are three single-step execution methods:

- **Step into**: The execution will enter into the function if the current statement is a function call;
- **Step over**: The execution will move to the next statement, no matter the current statement is a simple statement or a function call;
- **Step out**: If you enter a function, and you want the execution to exit the function and return to the next statement after the function call.

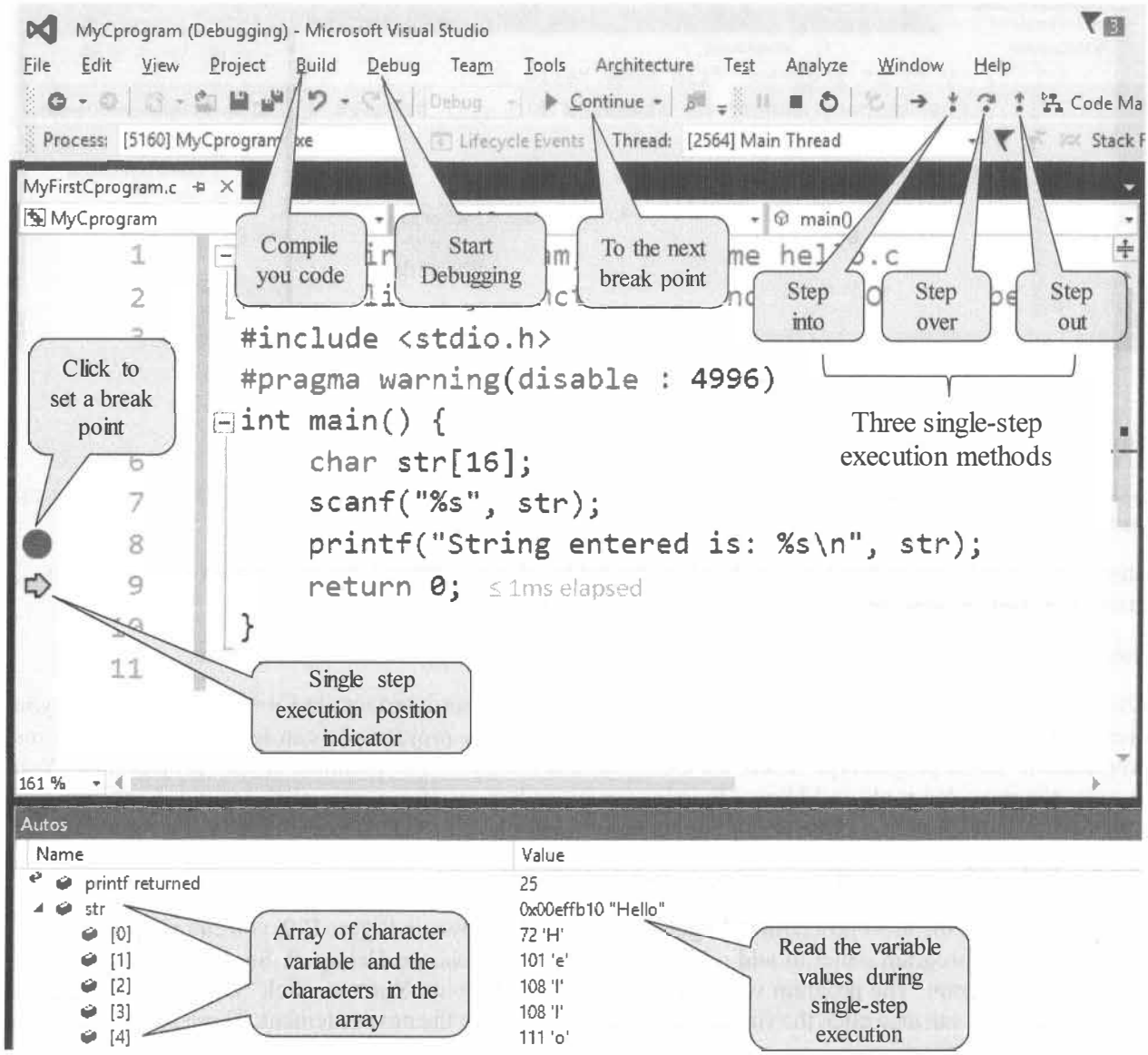


Figure B.9. Debugging your C program.

B.4 Programming environments supporting Scheme programming

In this section, we first introduce how can you start to write simple Scheme programs, and then explain how you use the DrRacket programming environment to run your Scheme programs.

B.4.1 Getting started with DrRacket

A simple Scheme program is extremely simple. Most mathematical expressions in prefix notations are simple Scheme programs. For example,

(+ 1 2)

is a simple Scheme program that can be executed. To print "hello world," all we need to write is:

(print "hello world")

In the rest of the section, we will introduce DrRacket, a free Scheme programming environment for educational purposes. DrRacket can be downloaded from `http://racket-lang.org/download/`

After you have downloaded and installed DrRacket, you can follow the following steps to set up and then use the programming environment to run your Scheme programs. We are referring to DrRacket version R5Rd:

1. Start the program DrRacket. As shown in Figure B.10, a window with two sections will be opened. In the upper section window, you can enter your program, and in the lower section, evaluation results and possible error message will be displayed.

2. To have the full functionality, you need to choose the right language. In this text, we used R5RS (professional edition). You can also use the "Advanced Student" version. However, some features are not supported in the student version, such as the pair data structure and its operations.

   By default, the system is set to the "Beginning Student" version that works only for basic functions.

3. Enter your program in the upper window. For example,

```
(print "hello world")
(newline)                ; start a newline for next print
(print (+ (* 3 8) 10))
(- 20 5)
```

4 Click on "Check Syntax" to check the possible syntax errors. Click on "Run" to execute your program. The single step execution option is only available when you choose the "Beginning Student" version. Click on "Stop" to abort the execution.

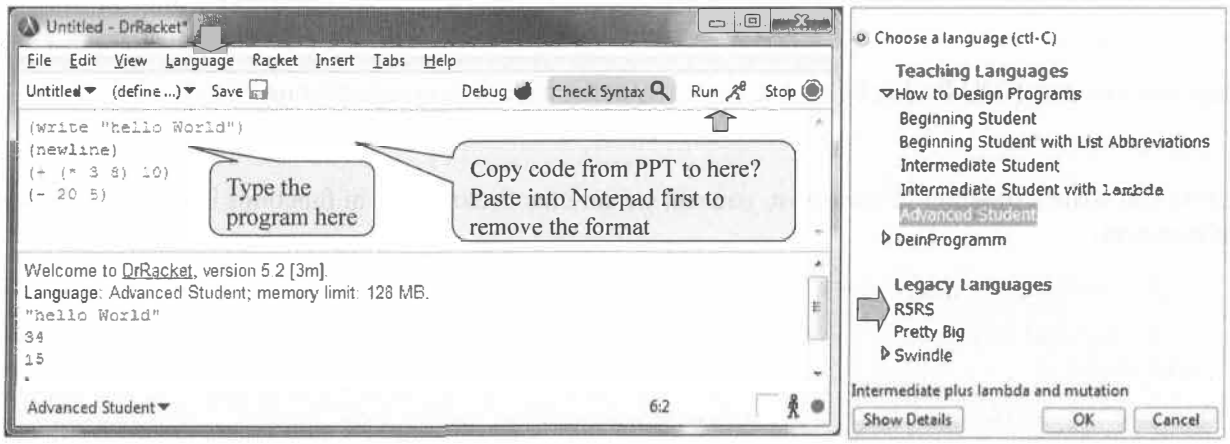


Figure B.10. DrRacket programming environment and language selection.

B.4.2 Download DrRacket programming environment

DrRacket home page:

`http://racket-lang.org/`

DrRacket download page:

`http://racket-lang.org/download/`

B.5 Programming environments supporting Prolog programming

In this section, we will explain how to use the GNU Prolog environment under the Unix operating system to edit, compile, and execute Prolog programs. A short introduction to Unix and basic Unix commands is given in Appendix B.1.

B.5.1 Getting started with the GNU Prolog environment

After you have logged onto a Unix server with GNU Prolog installed, you can create a new directory using the command:

```
mkdir Prologfiles
```

Then you can put all your Prolog files and programs in this directory. You may create subdirectories in this directory.

Enter the directory by using the Change Directory command:

```
cd Prologfiles
```

To write a GNU Prolog program, you can either use a Unix editor, for example, `pico` or `vi` (`pico` is more convenient) or upload (e.g., using an FTP client software) a pre-edited file into the directory `Prologfiles`. The name of a Prolog program should have an extension `.pl`.

To compile a Prolog program, say `myprologprog.pl`, type the command

host> gplc myprologprog.pl

at the operating system prompt `host>`, where gplc is the GNU Prolog compiler.

The compiler will generate the machine code program, sometimes called executable, stored under the name `myprologprog`.

Then you can start GNU Prolog by typing `gprolog` at the operating system prompt `host>`

```
host> gprolog
```

Before you write a program of your own, you can execute the Prolog built-in functions in the GNU Prolog environment.

```
| ?- write(hello).      % hello is a constant
| ?- write(Hello).      % Hello is a variable. Its address will be
displayed*/
| ?- write('hello world').  % a string is printed
| ?- read(Y), write('The variable entered is '), write(Y), nl.
% nl prints a newline. Type a period and an enter at the end of the input
| ?- X is 2+2.
| ?- Y is 5*8.
| ?- Y is 2**10.
| ?- 2*2 =:= 2+2.   % returns yes.
| ?- =:=(2*2, 2+2). % returns yes.
```

```
| ?- ==('Apple', 'Orange'). % returns no.
| ?- 'Apple' == 'Apple'.    % returns yes.
| ?- length([a, b, x, y, 2, 45, z], L).
| ?- append([a, b, c, d], [4, 6, 8], LL).
| ?- append(X,Y,[a,b,c]). Then, type ";" to obtain all possible answers.
| ?- X is [1 | [2 | [3 | []]]], write(X). Explain the output.
```

If you have written your own Prolog program, you can execute the executable created by the `gplc` compiler. For example, you can type

```
|?- [myprologprog].
```

to execute your program, where, `|?-` is the GNU Prolog prompt. Then you can enter a goal (question) to search the database of `myprologprog`.

To exit from the GNU Prolog system, type the end-of-file character at the Prolog prompt `^d`, that is,

```
|?- ^d
```

You can turn on and turn off the debugging tool (trace) by executing the goals, respectively:

```
|?- trace.       // turn on trace tool
|?- notrace.     // turn off trace tool
```

When dealing with big sources, it is not very practical to trace every single step of the execution. You can define a set of spy-points on interesting predicates, and you will be prompted when the debugger reaches one of these predicates. Spy-points can be added using spy/1.

You can set a set of spy-points using spy(PredSpec) sets, which set a spy-point on all the predicates given by PredSpec, where PredSpec defines one or several predicates.

For more Prolog commands, please check the GNU online manual at:

http://www.gprolog.org/manual/gprolog.html

B.5.2 Getting started with Prolog programming

To write a GNU Prolog program in Unix, you can either use a Unix editor, for example, pico or vi (pico is more convenient) or upload (using an SSH) a pre-edited file into your Unix environment. The name of a GNU Prolog program should have an extension .pl. To start with, you can enter the following program using pico editor (or other editor) and save the file as weather.pl;

```
/* Database for weather. It consists of facts rules. In this example, the
rule will cause a number of goals to be called */

% Facts

weather(phoenix, spring, hot).

weather(phoenix, summer, hot).

weather(phoenix, fall, hot).

weather(phoenix, winter, warm).

weather(wellington, spring, warm).
```

```
weather(wellington, summer, warm).

weather(wellington, fall, hot).

weather(wellington, winter, cold).

weather(toronto, spring, cold).

weather(toronto, summer, hot).

weather(toronto, fall, cold).

weather(toronto, winter, cold).

% Rules

warmer(C1, C2) :- weather(C1, spring, hot), weather(C2, spring, warm).

warmer(C1, C2) :- weather(C1, spring, hot), weather(C2, spring, cold).

/* The following rule is a compound question. It will cause a number of goals
(questions) to be called. It can be considered as the "main" program. Please
note, since the questions are connected by "and" relationship, it stops if a
"no" answer is given to any single question. You could use the "or"
relationship to connect questions. The compound question will stop if a "yes"
answer                                  is                                found.
*/

weatherquestions :-

  warmer(phoenix, X),

  write('Phoenix is warmer than '), write(X), nl,

  weather(City1, fall , hot),

  write('City1 = '), write(City1), nl,

  weather(City2, _ , hot),

  write('City2 = '), write(City2), nl,

  weather(_, Season , warm),

  write('Season = '), write(Season), nl,

  weather(C1, summer , hot),

  weather(C1, fall , hot),

  write('C1 = '), write(C1), nl,

  weather(C2, spring, warm),

  weather(C2, fall , warm),

  write('C2 = '), write(C2), nl, nl.
```

To compile a Prolog program, say myprologprog.pl, type gplc weather.pl at the operating system prompt

```
host> gplc weather.pl
```

The compiler will generate the machine code program stored under the name weather. To start GNU Prolog, type "gprolog" at the operating system prompt

```
host> gprolog
```

Once you are in gprolog mode, you can try a few simple program statements:

To execute a program that has been compiled (machine code of a Prolog program), for example, weather, type

```
|?- [weather].
```

where, |?- is the GNU Prolog prompt and you do not enter it. DO NOT forget the dot at the end. Now, you can ask the following questions about the weather program:

```
|  ?- warmer(phoenix, X).
|  ?- weather(City, fall , hot).
|  ?- weather(City, _ , hot).
% To see more answers, enter ";" after each answer.
|  ?- weather(_, Season , warm).
|  ?- weatherquestions.  % this will call all questions in the rule.
```

To exit from GNU Prolog, type your end-of-file character at the main Prolog prompt ^d (Ctrl-d).

You can find a complete set of GNU Prolog commands at http://www.gprolog.org/manual/gprolog.html.

B.5.3 Download Prolog programming development tools

GNU Prolog is running in the Unix operating system. To connect to a Unix server, you can use a telnet client software, or a secure telnet client software like SSH if your server requires a secure connection.

You can download a personal version of SSH program from, for example, `www.openssh.com`.

The full GNU Prolog manual is available at:

http://www.gprolog.org/manual/gprolog.html

Windows and Mac versions running as an Independent Development Environment can be downloaded from the GNU Prolog download page (Unix, Linux and MS Windows versions):

http://www.gprolog.org/#download

You can also execute Prolog programs using SWI-Prolog and Quintus Prolog, which can be downloaded at the following sites, respectively.

SWI-Prolog: `http://www.swi-prolog.org/`

Quintus Prolog: `http://www.sics.se/isl/quintuswww/site/`

Appendix C

ASCII Character Table

ASCII stands for American Standard Code for Information Interchange. The appendix lists the decimal, hexadecimal, binary, and the corresponding character values.

Deci -mal	Hex	Binary	Char	Comment
000	00	000 0000	NUL	Null character
001	01	000 0001	SOH	Start of Header
002	02	000 0010	STX	Start of Text
003	03	000 0011	ETX	End of Text
004	04	000 0100	EOT	End of Trans.
005	05	000 0101	ENQ	Enquiry
006	06	000 0110	ACK	Acknowledge
007	07	000 0111	BEL	Bell
008	08	000 1000	BS	Backspace
009	09	000 1001	HT	Horizontal Tab
010	0A	000 1010	LF	Line Feed
011	0B	000 1011	VT	Vertical Tab
012	0C	000 1100	FF	Form Feed
013	0D	000 1101	CR	Carriage Return
014	0E	000 1110	SO	Shift Out
015	0F	000 1111	SI	Shift In
016	10	001 0000	DLE	Data Link Escape
017	11	001 0001	DC1	Device Control 1
018	12	001 0010	DC2	Device Control 2
019	13	001 0011	DC3	Device Control 3
020	14	001 0100	DC4	Device Control 4
021	15	001 0101	NAK	Negative Ack.
022	16	001 0110	SYN	Synchronous Idle
023	17	001 0111	ETB	End Trans. Block
024	18	001 1000	CAN	Cancel
025	19	001 1001	EM	End of Medium
026	1A	001 1010	SUB	Substitute
027	1B	001 1011	ESC	Escape
028	1C	001 1100	FS	File Separator
029	1D	001 1101	GS	Group Separator
030	1E	001 1110	RS	Record Separator
031	1F	001 1111	US	Unit Separator
032	20	0010 0000	SP	Space
033	21	010 0001	!	Exclamation mark
034	22	010 0010	"	Double quote
035	23	010 0011	#	Number sign
036	24	010 0100	$	Dollar sign
037	25	010 0101	%	Percent
038	26	010 0110	&	Ampersand
039	27	010 0111	'	Single quote
040	28	010 1000	(	Parenthesis
041	29	010 1001	)	Parenthesis
042	2A	010 1010	*	Asterisk
043	2B	010 1011	+	Plus
044	2C	010 1100	,	Comma
045	2D	010 1101	-	Dash
046	2E	010 1110	.	Dot
047	2F	010 1111	/	Slash
048	30	011 0000	0	
049	31	011 0001	1	
050	32	011 0010	2	
051	33	011 0011	3	
052	34	011 0100	4	
053	35	011 0101	5	
054	36	011 0110	6	

055	37	011 0111	7	
056	38	011 1000	8	
057	39	011 1001	9	
058	3A	011 1010	:	Colon
059	3B	011 1011	;	Semicolon
060	3C	011 1100	<	Less than
061	3D	011 1101	=	Equal
062	3E	011 1110	>	Greater than
063	3F	011 1111	?	Question mark
064	40	100 0000	@	At symbol
065	41	100 0001	A	
066	42	100 0010	B	
067	43	100 0011	C	
068	44	100 0100	D	
069	45	100 0101	E	
070	46	100 0110	F	
071	47	100 0111	G	
072	48	100 1000	H	
073	49	100 1001	I	
074	4A	100 1010	J	
075	4B	100 1011	K	
076	4C	100 1100	L	
077	4D	100 1101	M	
078	4E	100 1110	N	
079	4F	100 1111	O	
080	50	101 0000	P	
081	51	101 0001	Q	
082	52	101 0010	R	
083	53	101 0011	S	
084	54	101 0100	T	
085	55	101 0101	U	
086	56	101 0110	V	
087	57	101 0111	W	
088	58	101 1000	X	
089	59	101 1001	Y	
090	5A	101 1010	Z	

091	5B	101 1011	[	Bracket
092	5C	101 1100	\	Back slash
093	5D	101 1101	]	Bracket
094	5E	101 1110	^	Circumflex
095	5F	101 1111	_	Underscore
096	60	110 0000	`	
097	61	110 0001	a	
098	62	110 0010	b	
099	63	110 0011	c	
100	64	110 0100	d	
101	65	110 0101	e	
102	66	110 0110	f	
103	67	110 0111	g	
104	68	110 1000	h	
105	69	110 1001	i	
106	6A	110 1010	j	
107	6B	110 1011	k	
108	6C	110 1100	l	
109	6D	110 1101	m	
110	6E	110 1110	n	
111	6F	110 1111	o	
112	70	111 0000	p	
113	71	111 0001	q	
114	72	111 0010	r	
115	73	111 0011	s	
116	74	111 0100	t	
117	75	111 0101	u	
118	76	111 0110	v	
119	77	111 0111	w	
120	78	111 1000	x	
121	79	111 1001	y	
122	7A	111 1010	z	
123	7B	111 1011	{	Brace
124	7C	111 1100	|	Vertical bar
125	7D	111 1101	}	Brace
126	7E	111 1110	~	Tilde
127	7F	111 1111	DEL	Delete

Bibliography

Appleby, D., and J. VandeKopple. Programming Languages: Paradigm and Practice. 2nd ed. New York: McGraw-Hill, 1997.

Bal, H., and D. Grune. *Programming Languages Essentials.* Boston: Addison Wesley, 1994.

Brickley, D., and R. V. Guha. "RDF Vocabulary Description Language 1.0: RDF Schema." *W3C Recommendation.* 2004. http://www.w3.org/TR/rdf-schema/

Chen, Y. Testing and Evaluating Fault-Tolerant Protocols by Deterministic Fault Injection. Düsseldorf: VDI-Verlag GmbH, 1993.

Chen, Y. "Analyzing and visual programming internet of things and autonomous decentralized systems," Simulation Modelling Practice and Theory Volume 65, June 2016, pp. 1-10.

Chen, Y., G. De Luca, "VIPLE: Visual IoT/Robotics Programming Language Environment for Computer Science Education," IPDPS Workshops 2016: 963-971.

Chen, Y. and H. Hu. "Internet of Intelligent Things and Robot as a Service." *Simulation Modelling Practice and Theory* (2012). http://dx.doi.org/10.1016/j.simpat.2012.03.006.

Chen, Y., H. Huang, and W. T. Tsai. "Scheduling Simulation in a Distributed Wireless Embedded System." *SIMULATION: Transactions of the Society for Modeling and Simulation International* 81, no. 6 (June 2005):425–36.

Chen, Y., and Y. Kakuda. "Autonomous Decentralised Systems in Web Computing Environment." *International Journal of Critical Computer-Based Systems* 2, no. 1 (2011):1–5.

Chen, Y. and W. T. Tsai. *Service-Oriented Computing and Web Software Integration.* 5th ed. Dubuque: Kendall Hunt, 2015.

Chen, Y. and W. T. Tsai. "Towards Dependable Service-Orientated Computing Systems." *Simulation Modelling Practice and Theory* 17, no. 8 (September 2009):1361–66.

Clements, A. *68000 Family Assembly Language.* Boston: Brooks/Cole, 1994.

Cormen, T., C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* 2nd ed. Cambridge: MIT, 2001.

Deitel, H., and P. Deitel. *C++: How to Program.* 3rd ed. Murray Hill: Prentice Hall, 2001.

Diaz, D. "GNU Prolog." *A Native Prolog Compiler with Constraint Solving over Finite Domains.* http://gnu-prolog.inria.fr/manual/index.html.

Gunnerson, E. *A Programmer's Introduction to C#.* Berkeley: Apress, 2001.

Hankins, T., and T. Luce. Programming Languages: Paradigm and Practice—Prolog Mini-Manual. New York: McGraw-Hill, 1991.

Hennessy, J. L., and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* 2nd ed. Burlington: Morgan Kaufmann, 1995.

Hull, R. G. Programming Languages: Paradigm and Practice—PC Scheme Mini-Manual. New York: McGraw-Hill, 1991.

Kernighan, B. *Programming in C: A Tutorial.* http://www.lysator.liu.se/c/bwk-tutor.html.

Kernighan, B., and D. Ritchie. *The C Programming Language.* Murray Hill: Prentice Hall, 1978.

Kernighan, B., and R. Pike. *The UNIX Programming Environment.* Prentice Hall, 1984.

Lister, A. M., and R. D. Eager. *Fundamentals of Operating Systems.* 5th ed. Hampshire: Macmillan, 1993.

Manis, V., and J. Little. *The Schematics of Computation.* Murray Hill: Prentice Hall, 1995.

Manola, F., and E. Miller. "RDF Primer". *W3C Recommendation.* 2004. http://www.w3c.org/TR/REC-rdf-syntax/.

Muldner, T. C++ Programming with Design Patterns Revealed. Boston: Addison Wesley, 2002.

Myers, G. J. *The Art of Software Testing.* New York: Wiley, 1979.

Oualline, S. *Practical C Programming.* 3rd ed. Sebastopol: O'Reilly & Associates, 1997.

Patterson, D., and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* 2nd ed. Burlington: Morgan Kaufmann, 1998.

Paul, R., W. T. Tsai, Y. Chen, C. Fan, Z. Cao, and H. Huang. "E2E Testing and Evaluation of High Assurance Systems." In *Springer Handbook of Engineering Statistics*, edited by Hoang Pham. London: Springer-Verlag, 2006.

Powell, R., and R. Weeks. *C# and the .Net Frameworks: The C++ Perspectives.* Indianapolis: Sams, 2002.

Siewiorek, D., and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation.* 3rd ed. Natick: A. K. Peters, 1998.

Singh, M. P., and M. N. Huhns. *Service-Oriented Computing.* New York: Wiley, 2005.

Stroustrup, B. *The C++ Programming Language.* Murray Hill: Addison Wesley, 1997.

Tanenbaum, A. *Modern Operating Systems.* 2nd ed. Upper Saddle River: Prentice Hall, 2001.

Tsai, W. T. "Service-Oriented System Engineering: A New Paradigm." *IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, Beijing, October 2005, 3–8.

Tsai, W. T., X. Bai, and Y. Chen. *On Service-Oriented Software Engineering.* Beijing: Tsinghua University Press, 2008.

Tsai, W. T., Y. Chen, C. Cheng, X. Sun, G. Bitter, and M. White. "An Introductory Course on Service-Oriented Computing for High Schools." *Journal of Information Technology Education* 7, (2008):323–46.

Tsai, W. T., C. Fan, Y. Chen, and R. Paul. "DDSOS, Distributed Service-Oriented Simulation." *Proceedings of 39th Annual Simulation Symposium (ANSS)*, Huntsville, AL, April 2006.

Tsai, W. T., X. Liu, Y. Chen, and R. Paul. "Dynamic Simulation Verification and Validation by Policy Enforcement." *38th Annual Simulation Symposium*, April 2005, 91–98.

Tsai, W. T., R. A. Paul, B. Xiao, Z. Cao, and Y. Chen. "PSML-S: A Process Specification and Modeling Language for Service Oriented Computing." *9th IASTED International Conference on Software Engineering and Applications (SEA)*, Phoenix, November 2005, 160–67.

Tsai, W. T., W. Song, R. Paul, Z. Cao, and H. Huang. "Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing." *Proceedings of IEEE COMPSAC 2004*, 554–59.

Tsai, W. T., X. Wei, and Y. Chen. "A Robust Testing Framework for Verifying Web Services by Completeness and Consistency Analysis." *IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, Beijing, October 2005, 151–58.

Tsai, W. T., X. Wei, Y. Chen, B. Xiao, R. Paul, and H. Huang. "Developing and Assuring Trustworthy Web Services." In *Proceedings of 7th International Symposium on Autonomous Decentralized Systems*, Chengdu, China, April 4–8, 2005, 43–50.

Tsai, W. T., L. Wu, J. Elston, and Y. Chen. "Collaborative Learning Using Wiki Web Sites for Computer Science Undergraduate Education: A Case Study." *IEEE Transactions on Education* 54, no. 1 (February 2011):114–24.

Index

CPSIA information can be obtained
at www.ICGtesting.com
Printed in the USA
LVOW05s2012040218
564542LV00004BA/12/P