C++ is a fast, flexible language but its fundamentals must be understood correctly. These topics are selected to explain the fundamental concepts of the C++ language to the beginners

SELECTED TOPICS IN C++

M.LIYANAGE

Preface

C++ started in the early 80’s and after more than 30 years later, it is still one of the most used programming languages. This is so for a very good reason. It’s not just the speed of C++ that makes it a popular choice. It is a very flexible and predictable language. You can be sure to have a constant performance with a C++ program. And this is achieved by being able to control a lot of aspects of the program. C++ puts the programmer in the front seat. The programmer can take control of how things need to be run. It’s predictable. There are no suprises. Most importantly, C++ let’s you go very close to the low-level of the computer.

You can find a lot of discussions about the speed of C++ vs. Java. It is widely accepted that the speed of Java will match that of C++ in the best case, but not surpass it. But here’s the caveat. To get the best performance of C++ you need to learn how to use it properly. Like I said, C++ puts the programmer in full control and it is our responsibility to control it properly. It is easy to write bad code with C++. But to write good, optimized code, you need to learn the core of it because unless you know what exactly is happening with your code, you cannot control it or optimize it.

It’s often said that C++ is a difficult language to learn. Learning a programming language is not that difficult at all. Same goes for C++. It is not a difficult language. It becomes a difficult language when you don’t know it properly. When you don’t understand its core concepts it becomes difficult. C++ has no garbage collection. You need to collect your own garbage. C++ has the concept of stack memory and heap memory. You need to control where you want to put your objects. It has constructors and destructors. You need to know when and how they are called. Then it has multiple inheritance, something that’s not part of most languages. It has pointers and references. You need to know how and when you should use them. Then you have virtual functions and virtual tables and virtual pointers. These are few of the important fundamentals of C++ that should have a firm grip on.

Purpose of this book

It is not difficult to get up and running with C++. It is like any other language. But once you go through the basics of the language, classes, constructors, inheritance, conditionals, loops, etc., C++ will feel like most other high-level languages. But it can be a challenge to move on from there and to go from being a C++ beginner to an intermediate or advanced programmer. There are many core concepts of C++ and without knowing them you can’t know the language in its fullest. You cannot write fast optimized code without knowing what happens during compilation or runtime.

The aim of this book is to take C++ beginners to the next level. To discuss the fundamental internals of the language so the beginners can fully understand what happens with their code. This book is divided in to 20 different topics which attempt to talk about the most used concepts of C++.

Who this book is for

This book is intended for beginner level C++ programmers. If you know about classes, inheritance, constructors and virtual functions, you are the intended audience of this book.

Who this book is not for

This book is not for programmers starting to learn C++ from the beginning. That is, this is not a “Hello, World” book. This should be the book you read after that.

Table of Contents

Topic 1

The size of an object

Objects are what makes the language Objected-Oriented, so they are at the core of C++. So it's only natural we dedicate our first topic to learning about the size and composition of them. Let's start with the bare minimum, the empty class.

The empty class

What's the definition of an empty class?

```
class emptyClass
{};
```

So how exactly empty is the empty class? Well it's pretty empty, but not nothing. Here's how empty an empty class is:

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

class emptyClass
{};

int main(int argc, char** argv)
{
        emptyClass emptyClassObj;
        cout << "Size of emptyClassObj: " << sizeof(emptyClassObj) << endl;
}
```

```
Size of emptyClassObj: 1
```

The size of a class with nothing in it is 1 byte. But why so? Why is a class object 1 byte when there is nothing in it? Because the standard does not let objects have a size of 0. Again, why? This is because you need to be able to distinguish between two objects of the same class!

Take a look at this:

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

class emptyClass
```

```
{};

int main(int argc, char** argv)
{
        emptyClass emptyClassObj1;
        emptyClass emptyClassObj2;
        cout << “Memory address of emptyClassObj1: “ << &emptyClassObj1 << endl;
        cout << “Memory address of emptyClassObj2: “ << &emptyClassObj2 << endl;
}
```

```
Memory address of emptyClassObj1: 0x7fffffffe43e
Memory address of emptyClassObj2: 0x7fffffffe43f
```

The memory addresses of two different empty classes are different. This way it lets you differentiate two different objects. But there really is nothing in there.

Then how big will an object get if we have an empty class derived from an empty class? 2 bytes, perhaps?

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

class emptyClass
{};

class derivedEmptyClass : public emptyClass
{};

int main(int argc, char** argv)
{
        emptyClass emptyClassObj;
        derivedEmptyClass derivedEmptyClassObj;
        cout << “Size of emptyClassObj: “ << sizeof(emptyClassObj) << endl;
        cout << “Size of derivedEmptyClassObj: “ << sizeof(derivedEmptyClassObj) << endl;
}
```

```
Size of emptyClassObj: 1
Size of derivedEmptyClassObj: 1
```

It doesn’t change. You can keep deriving empty classes but the size will still be 1. Well, the size of an empty class is actually implementation dependent, although it is usually 1, but the fact to keep in mind is that it will always be the same value regardless of how many times you keep deriving.

Classes with member variables

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

class notSoEmptyClass
{
public:
	int value;
};

int main(int argc, char** argv)
{
	notSoEmptyClass notSoEmptyClassObj;
	cout << “Size of notSoEmptyClassObj: “ << sizeof(notSoEmptyClassObj) << endl;
	return 0;
}
```

```
Size of notSoEmptyClassObj: 4
```

No surprises here. The object has to contain the integer member variable value. But perhaps something to note is the size of the object, 4 bytes. Let’s print out a little bit more information:

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

class notSoEmptyClass
{
public:
	int value;
};

int main(int argc, char** argv)
{
	int intVal = 5;
```

```
    notSoEmptyClass notSoEmptyClassObj;
    cout << “Size of notSoEmptyClassObj: “ << sizeof(notSoEmptyClassObj) << endl;
    cout << “Size of integer: “ << sizeof(intVal) << endl;
    return 0;
}
```

```
Size of notSoEmptyClassObj: 4
Size of integer: 4
```

So the size of the object is actually the size of the integer. So when there was nothing in the object, it had a size of 1 byte, but when we put an integer in there, it got as big as the integer itself, which is 4 bytes. This shows that this almost empty object just contains the integer variable, nothing else. Let’s add a bit more stuff and see how things change.

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

class notSoEmptyClass
{
public:
    int value;
};

int main(int argc, char** argv)
{
    int intVal = 5;
    notSoEmptyClass notSoEmptyClassObj;
    cout << “Size of notSoEmptyClassObj: “ << sizeof(notSoEmptyClassObj) << endl;
    cout << “Size of integer: “ << sizeof(intVal) << endl;
    return 0;
}
```

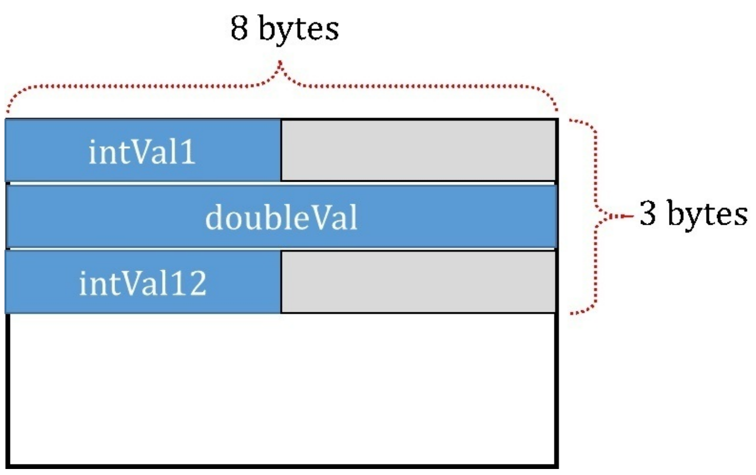
```
Size of notSoEmptyClassObj: 16
Size of integer: 4
Size of double: 8
```

The size of the object is increased, as expected. But why is it 16, and not the summation of the integer and the double? Why not 12 bytes? This is because of *padding*. Variables are padded to certain boundary values. This is mostly implementation/architecture dependent and in this particular case, it looks like boundaries are at 8 bytes. Let’s confirm this by adding another integer.

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

class notSoEmptyClass
{
public:
        int intValue1;
        double doubleVal;
        int intValue2;
};

int main(int argc, char** argv)
{
        notSoEmptyClass notSoEmptyClassObj;
        cout << “Size of notSoEmptyClassObj: “ << sizeof(notSoEmptyClassObj) << endl;
        return 0;
}
```

```
Size of notSoEmptyClassObj: 24
```

Now the size of the object is 24 bytes. Does this add up with our padding to boundary concept? Yes. You see, the layout of the variables has to be in 8 byte boundaries or less. Since the double is 8 bytes long, these variables need to be packed in to 8 byte wide memory locations. You can visualize the above as packed in to memory as follows:

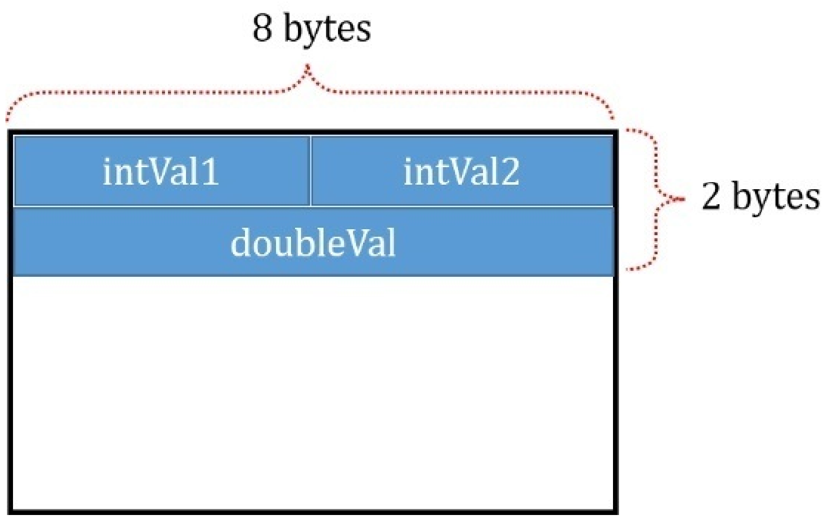


Now let’s see what happens if you rearrange the variables positions to be a little more space efficient.

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

class notSoEmptyClass
{
public:
        int intValue1;
        int intValue2;
        double doubleVal;
};

int main(int argc, char** argv)
{
        notSoEmptyClass notSoEmptyClassObj;
        cout << "Size of notSoEmptyClassObj: " << sizeof(notSoEmptyClassObj) << endl;
        return 0;
}
```

```
Size of notSoEmptyClassObj: 16
```

Just rearranging the positions reduced the object's size by 8 bytes. You can probably guess what is happening here. The variables are packed much more efficiently now.



Why are the variables laid out this way then? Why didn't the compiler make an intelligent choice to re-arrange the variables to be more space efficient? I'm sure the modern compilers are way more intelligent for being able to arrange variables to be memory efficient, but there are other reasons for laying them out as the programmer intended. We will discuss that in another topic. Now before we leave member variables, let's check something else out very quickly.

What would happen when we make one of our integer variables a static?

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

class notSoEmptyClass
{
public:
        int intValue1;
        double doubleVal;
        static int intValue2;
};

int main(int argc, char** argv)
{
        notSoEmptyClass notSoEmptyClassObj;
        cout << "Size of notSoEmptyClassObj: " << sizeof(notSoEmptyClassObj) << endl;
        return 0;
}
```

```
Size of notSoEmptyClassObj: 16
```

We know from our previous example that if *intVal2* was not static, the object size would have been 24 bytes. But now it’s 16 bytes, as if *intVal2* does not exist in the object. And this is indeed the case. Because static variables are not stored in the objects themselves. Objects of a class can be many. So there has to be as many *intVal1*’s and *doubleVal*’s as there are objects. But there can be only one *intVal2*. A common one for all of the objects. So the static variables are not, and practically cannot reside inside one object instance. Therefore they are in the global memory space. Outside of any object. That’s why no matter how many static variables you put into an object, they will not bloat the size of the object.

Classes with member functions

Now we’ll see what happens to the size of an object when you add in functions. Don’t worry, this won’t be long.

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

class notSoEmptyClass
{
```

```
public:
        void foo() {}
};

int main(int argc, char** argv)
{
        notSoEmptyClass notSoEmptyClassObj;
        cout << “Size of notSoEmptyClassObj: “ << sizeof(notSoEmptyClassObj) << endl;
        return 0;
}
```

```
Size of notSoEmptyClassObj: 1
```

What did we do? We removed all the variables it had and put in a single function in the class. And the size? 1 byte. This is the same size of the object when it was completely empty. So functions in a class do not take any space in the object? The answer is yes.

But at that time I said that the 1 byte allocation of an empty class object is to make two objects of the same class to be distinguishable. So you may not be completely convinced, arguing that since now there is something in the class, that may be taking up the 1 byte space. OK, let’s add another function and see.

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

class notSoEmptyClass
{
public:
        void foo() {}
        void bar() {}
};

int main(int argc, char** argv)
{
        notSoEmptyClass notSoEmptyClassObj;
        cout << “Size of notSoEmptyClassObj: “ << sizeof(notSoEmptyClassObj) << endl;
        return 0;
}
```

```
Size of notSoEmptyClassObj: 1
```

It still does not take any space. Now some of you may still be not convinced completely because the functions are empty. So maybe an intelligent compiler is just getting rid of them? It does not, even though it can do it in this case as the functions are empty and they are not called anywhere in the code. Let's wrap up this topic by confirming that functions indeed take no space in the object, empty or not.

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

class notSoEmptyClass
{
public:
    void foo() {}

    int bar(int count) {
        int retVal = 0;

        for (int i = 0; i < count; i++)
        {
            retVal++;
        }
        return retVal;
    }
};

int main(int argc, char** argv)
{
    notSoEmptyClass notSoEmptyClassObj;
    int result = notSoEmptyClassObj.bar(10);
    cout << "Result is: " << result << endl;
    cout << "Size of notSoEmptyClassObj: " << sizeof(notSoEmptyClassObj) << endl;
    return 0;
}
```

```
Result is: 10
Size of notSoEmptyClassObj: 1
```

So I changed the function *bar* to take in an argument, and have a local variable, a functions body and a return type. This is a pretty full function. And you can see the function *bar* is actually doing its calculation in the result. And yet the size of the object is still 1 byte. This is because the functions exist outside of the object. Like a static variable. This makes sense, doesn’t it? A function does the same thing regardless of which object instance calls it. In our case, the function *bar* always does the for loop the number of times passed as the argument. So there is no reason to have a function reside inside the object itself. This is different from a member variable. Because member variables are usually used to differentiate objects. Think about a *Person* class which has a string called *personName,* which has the corresponding person’s name. Now it makes perfect sense for each object to have the *personName* variable inside it. Because each object will have a different value for this variable. Now what about a function, for example, *GetName,* which simply returns the name of the *Person*? All this function would do is return the string variable. And we do not need this function to be inside the object, do we? The function simply has to take the string variable it was called upon and return it.

I suppose this should do it for this topic. I just showed you the basics of how the sizes of objects are determined. It should give you a good foundation to think about other complicated cases.

Topic 2

The Virtual Mechanism

I’m not going to explain what C++ virtual mechanism is. I’m sure you know what a virtual function is. In this topic I’m just going to prove its existence in the object level and show you how it really happens behind the curtains.

Let’s carry on from where we left off in topic 1.

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

class notSoEmptyClass
{
public:
        void foo() {}
};

int main(int argc, char** argv)
{
        notSoEmptyClass notSoEmptyClassObj;
        cout << “Size of notSoEmptyClassObj: “ << sizeof(notSoEmptyClassObj) << endl;
        return 0;
}
```

```
Size of notSoEmptyClassObj: 1
```

We showed that functions do not take any space in an object as they reside out of the object memory space. Let’s then see what would change when we make our only function a virtual.

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

class virtualFuncClass
{
public:
        virtual void foo() {}
};
```

```
int main(int argc, char** argv)
{
        double *doublePrt;
        virtualFuncClass virtualFuncClassObj;
        cout << “Size of virtualFuncClassObj: “ << sizeof(virtualFuncClassObj) << endl;
        cout << “Size of a pointer: “ << sizeof(doublePrt) << endl;
        return 0;
}
```

```
Size of virtualFuncClassObj: 4
Size of a pointer: 4
```

The only difference I made to the class is to make the function *foo* virtual (that and changing class name). The size of the object jumped to 4 bytes. So making the function *foo* a virtual added something to the object. As you’d guess, this is the virtual-table-pointer, commonly known as *vptr*. It is just a pointer. To make it clear I also declared a pointer to a double and showed that its size is 4 bytes. There is no special reason why I chose a pointer to a double. It could be a pointer to anything because pointer sizes don’t change with what they are pointing to. So this makes a slightly convincing argument that making the function virtual added a pointer to the object. But it might as well be an integer variable, right? Because we saw in topic 1 that integer variables take 4 bytes of space in an object. Before we show the actual *vptr* and what it is pointing to, let’s confirm this suspicion.

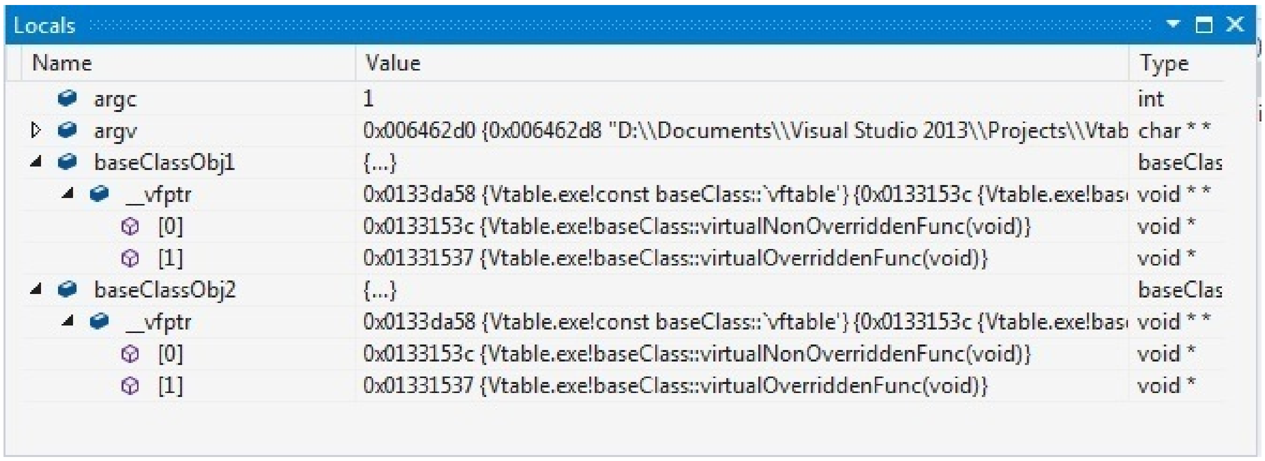
```
#include<iostream>
using namespace std;

class virtualFuncClass
{
public:
        virtual void foo() {}
        virtual void bar() {}
};

int main(int argc, char** argv)
{
        double *doublePrt;
        virtualFuncClass virtualFuncClassObj;
        cout << “Size of virtualFuncClassObj: “ << sizeof(virtualFuncClassObj) << endl;
        cout << “Size of a pointer: “ << sizeof(doublePrt) << endl;
        return 0;
}
```

```
Size of virtualFuncClassObj: 4
Size of a pointer: 4
```

So now there are two virtual functions and yet the size of the object hasn’t changed. Let’s then see what this pointer is pointing to and what is in it. We will use Visual Studio for this illustration as this information is visually available in debug mode.

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

class baseClass
{
public:
	void nonVirtualFunc() {}
	virtual void virtualNonOverriddenFunc() {}
	virtual void virtualOverriddenFunc() {}
};

int main(int argc, char** argv)
{
	baseClass baseClassObj1;
	baseClass baseClassObj2;
	return 0;
}
```

Bear with me for the function names for the time being, they have a purpose. Now if you put a breakpoint at the return statement and look at the *Locals* of the debug view window you will see something similar to this:


Locals

Name	Value	Type
argc	1	int
argv	0x006462d0 {0x006462d8 "D:\\Documents\\Visual Studio 2013\\Projects\\Vtab	char * *
baseClassObj1	{...}	baseClas
__vfptr	0x0133da58 {Vtable.exe!const baseClass::`vftable'} {0x0133153c {Vtable.exe!bas	void * *
[0]	0x0133153c {Vtable.exe!baseClass::virtualNonOverriddenFunc(void)}	void *
[1]	0x01331537 {Vtable.exe!baseClass::virtualOverriddenFunc(void)}	void *
baseClassObj2	{...}	baseClas
__vfptr	0x0133da58 {Vtable.exe!const baseClass::`vftable'} {0x0133153c {Vtable.exe!bas	void * *
[0]	0x0133153c {Vtable.exe!baseClass::virtualNonOverriddenFunc(void)}	void *
[1]	0x01331537 {Vtable.exe!baseClass::virtualOverriddenFunc(void)}	void *

The contents are pretty self-explanatory. We are looking at the contents of the two *baseClass* objects we created. You can see the following facts from this debug view:

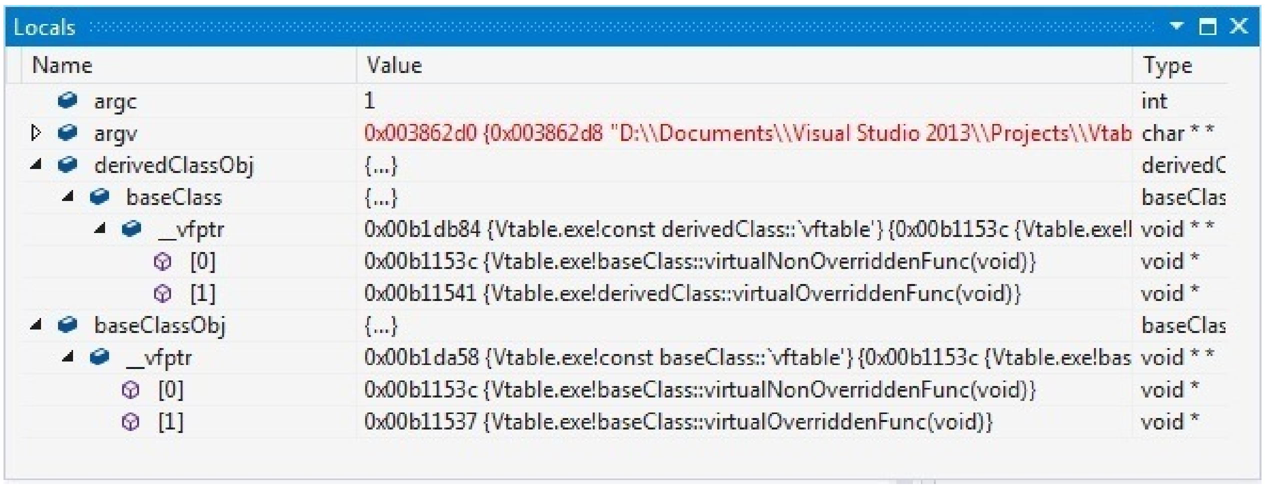
- There is nothing in this object except for the virtual-table-pointer, here named ***__vfptr***
- ***__vfptr*** is a pointer to a pointer of type void. This means that it points to an array of pointers of type void (check last column)
- ***__vfptr*** is pointing to ***__vftable***
- ***__vftable*** has two entries of type ***void****
- ***__vftable*** makes no mention about the non-virtual *nonVirtualFunc()*. This is correct as *nonVirtualFunc()* is only applicable to *baseClass*

Examining the *__vftable* entries, it is obvious what they are pointing to. The first points to function *virtualNonOverriddenFunc* and the second to *virtualOverriddenFunc*. What's more to note is that both of these functions are of type *baseClass*. The significance of this will become apparent soon.

The reason I created two objects of the same type is to validate the fact that there is only one copy of a function. This is evident if you look at the values of the *__vftable* entries. The addresses of the two virtual functions are the same for *baseClassObj1* and *baseClassObj2*, proving again that there is only one copy of a class member function. What is even more interesting is that the *__vftable* address is also the same for both objects. That means there is only one virtual function table for all objects of the same type. Now you need to carefully keep in mind that this is how Visual C++ is doing it. Other compilers could be doing it differently. But there really is no reason why you would have two different *__vftable*s for the same objects because they are both referring to the same functions. (This applies to just this case, as we have two objects of *baseClass* which is not derived of any class. If it is not then this fact would change.)

Now let's see how things change when we derive our *baseClass* and override functions.

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

class baseClass
{
public:
        void nonVirtualFunc() {}
        virtual void virtualNonOverriddenFunc() {}
        virtual void virtualOverriddenFunc() {}
};

class derivedClass : public baseClass
{
public:
        virtual void virtualOverriddenFunc() {}
};

int main(int argc, char** argv)
{
        baseClass baseClassObj;
        derivedClass derivedClassObj;
        return 0;
}
```



```
Locals
Name                    Value                                                                          Type
    argc                1                                                                              int
  ▷ argv                0x003862d0 {0x003862d8 "D:\\Documents\\Visual Studio 2013\\Projects\\Vtab      char * *
  ◢ derivedClassObj     {...}                                                                          derivedC
    ◢ baseClass         {...}                                                                          baseClas
      ◢ __vfptr         0x00b1db84 {Vtable.exe!const derivedClass::`vftable'} {0x00b1153c {Vtable.exe!l void * *
            [0]         0x00b1153c {Vtable.exe!baseClass::virtualNonOverriddenFunc(void)}               void *
            [1]         0x00b11541 {Vtable.exe!derivedClass::virtualOverriddenFunc(void)}              void *
  ◢ baseClassObj        {...}                                                                          baseClas
    ◢ __vfptr           0x00b1da58 {Vtable.exe!const baseClass::`vftable'} {0x00b1153c {Vtable.exe!bas void * *
          [0]           0x00b1153c {Vtable.exe!baseClass::virtualNonOverriddenFunc(void)}               void *
          [1]           0x00b11537 {Vtable.exe!baseClass::virtualOverriddenFunc(void)}                 void *
```

Things are as expected. The *__vftable* for *baseClass* is not changed. Both the functions still point to the *baseClass* versions of the functions. However, the *__vftable* entries of *derivedClass* is different. The second entry in the *__vftable* is of class *derivedClass*. And check the addresses they are pointing to. The first virtual function, *virtualNonOverriddenFunc* points to the same address in both *__ vftables*. This is correct as this function is not overridden. But the *virtualOverriddenFunc* addresses are different in the two *__vftables*. This, again, is expected since now the *derivedClass* object has overridden that function.

Let's wrap this up with a bit of a complete example which will discuss more aspects of the virtual mechanism.

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

class baseClass1
{
public:
        void nonVirtualFunc1()
                { cout << "nonVirtualFunc1" << endl; }
        virtual void virtualNonOverriddenFunc1()
                { cout << "virtualNonOverriddenFunc1" << endl; }
```

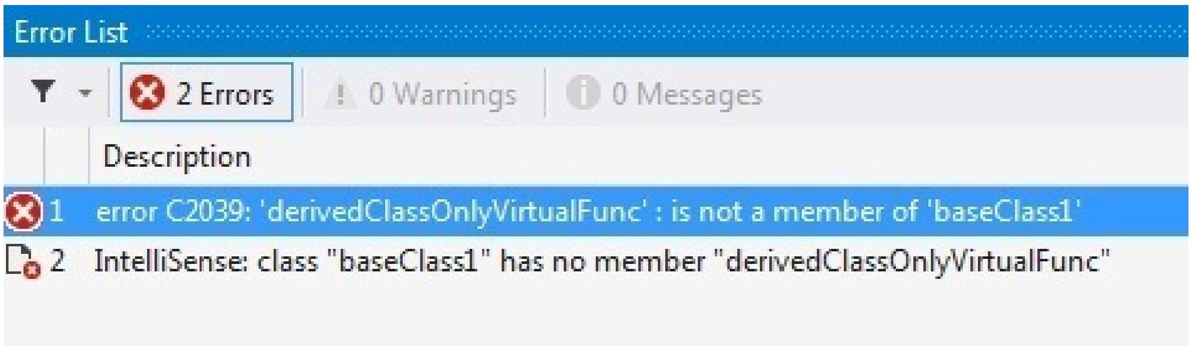
```
    virtual void virtualOverriddenFunc1()
        { cout << “virtualOverriddenFunc1” << endl; }
};

class baseClass2
{
public:
    void nonVirtualFunc2()
        { cout << “nonVirtualFunc2” << endl; }
    virtual void virtualNonOverriddenFunc2()
        { cout << “virtualNonOverriddenFunc2” << endl; }
    virtual void virtualOverriddenFunc2()
        { cout << “virtualOverriddenFunc2” << endl; }
};

class derivedClass : public baseClass1, baseClass2
{
public:
    virtual void virtualOverriddenFunc1()
        { cout << “virtualOverriddenFunc1” << endl; }
    virtual void derivedClassOnlyVirtualFunc()
        { cout << “derivedClassOnlyVirtualFunc” << endl; }
};

int main(int argc, char** argv)
{
    baseClass1 baseClass1Obj;
    baseClass2 baseClass2Obj;
    derivedClass derivedClassObj;

    baseClass1 *bc1Ptr = new derivedClass;
    derivedClass *dcPtr = new derivedClass;
    bc1Ptr->virtualOverriddenFunc1();
    dcPtr->virtualOverriddenFunc1();
    dcPtr->derivedClassOnlyVirtualFunc();

    return 0;
}
```

First off let me explain a little bit about what is happening here:

- There are now two base classes: *baseClass1* and *baseClass2*
- Each of the base classes have their own methods
- *derivedClass* (multiple) inherits from *baseClass1* and *baseClass2*
- *derivedClass* introduces its own new virtual function *derivedClassOnlyVirtualFunc*
- There are two pointers, *bc1Ptr* of type *baseClass1* and *dcPtr* of type *derivedClass*. Both of these points to a *derivedClass* object

Now here are the points I want to make:

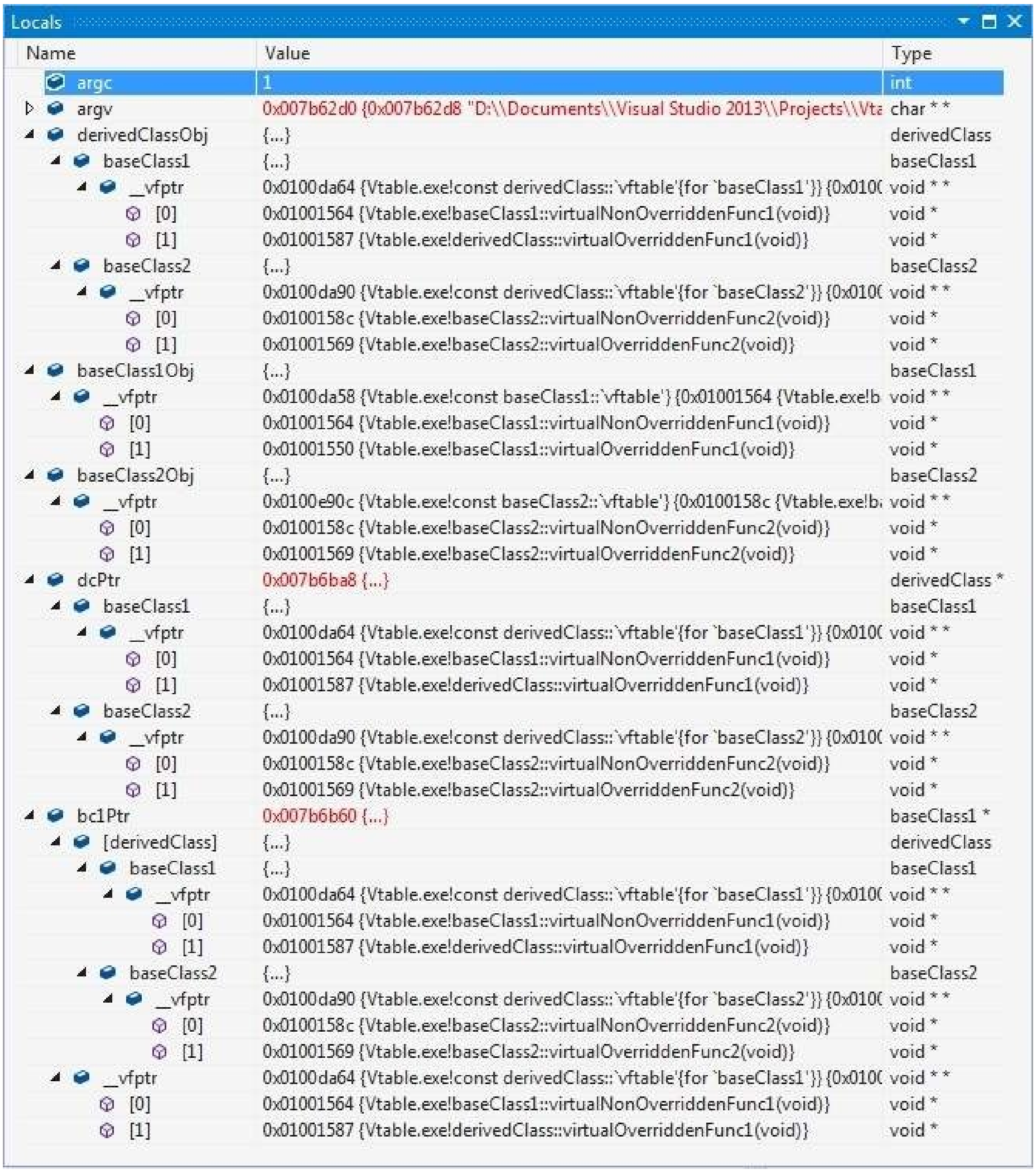
1. This will not compile. There is a compiler error in line *bc1Ptr->derivedClassOnlyVirtualFunc():*



```
1 error C2039: 'derivedClassOnlyVirtualFunc' : is not a member of 'baseClass1'
2 IntelliSense: class "baseClass1" has no member "derivedClassOnlyVirtualFunc"
```

This is important to remember. Polymorphism gives you the ability to do dynamic binding and call an overridden function in a derived class through a base class pointer. But before it does dynamic binding at runtime, the compiler has to know there is such a function at compile time. And for the compiler, it cannot see a *derivedClassOnlyVirtualFunc* in *baseClass1*. Use of __ *vfptr* and __ *vftable* happens at runtime. The compiler has no idea which function is going to be invoked. So for the compiler it looks like you are trying to call a function that is not defined in the class. It has no idea that you are referring to a derived class object and the function is defined there. In fact, it doesn't need to. This is not what polymorphism is supposed to do. So keep this in mind. Now let's comment out the compiler error line and move on.

2. Let's look in the object for __*vfptr* and __*vftable* information:



```
Locals
Name                    Value                                                                                  Type
  argc                  1                                                                                      int
▷ argv                  0x007b62d0 {0x007b62d8 "D:\\Documents\\Visual Studio 2013\\Projects\\Vta               char * *
◢ derivedClassObj       {...}                                                                                  derivedClass
  ◢ baseClass1          {...}                                                                                  baseClass1
    ◢ __vfptr           0x0100da64 {Vtable.exe!const derivedClass::`vftable'{for `baseClass1'}} {0x010(        void * *
        [0]             0x01001564 {Vtable.exe!baseClass1::virtualNonOverriddenFunc1(void)}                    void *
        [1]             0x01001587 {Vtable.exe!derivedClass::virtualOverriddenFunc1(void)}                     void *
  ◢ baseClass2          {...}                                                                                  baseClass2
    ◢ __vfptr           0x0100da90 {Vtable.exe!const derivedClass::`vftable'{for `baseClass2'}} {0x010(        void * *
        [0]             0x0100158c {Vtable.exe!baseClass2::virtualNonOverriddenFunc2(void)}                    void *
        [1]             0x01001569 {Vtable.exe!baseClass2::virtualOverriddenFunc2(void)}                       void *
◢ baseClass1Obj         {...}                                                                                  baseClass1
  ◢ __vfptr             0x0100da58 {Vtable.exe!const baseClass1::`vftable'} {0x01001564 {Vtable.exe!b          void * *
      [0]               0x01001564 {Vtable.exe!baseClass1::virtualNonOverriddenFunc1(void)}                    void *
      [1]               0x01001550 {Vtable.exe!baseClass1::virtualOverriddenFunc1(void)}                       void *
◢ baseClass2Obj         {...}                                                                                  baseClass2
  ◢ __vfptr             0x0100e90c {Vtable.exe!const baseClass2::`vftable'} {0x0100158c {Vtable.exe!b          void * *
      [0]               0x0100158c {Vtable.exe!baseClass2::virtualNonOverriddenFunc2(void)}                    void *
      [1]               0x01001569 {Vtable.exe!baseClass2::virtualOverriddenFunc2(void)}                       void *
◢ dcPtr                 0x007b6ba8 {...}                                                                       derivedClass *
  ◢ baseClass1          {...}                                                                                  baseClass1
    ◢ __vfptr           0x0100da64 {Vtable.exe!const derivedClass::`vftable'{for `baseClass1'}} {0x010(        void * *
        [0]             0x01001564 {Vtable.exe!baseClass1::virtualNonOverriddenFunc1(void)}                    void *
        [1]             0x01001587 {Vtable.exe!derivedClass::virtualOverriddenFunc1(void)}                     void *
  ◢ baseClass2          {...}                                                                                  baseClass2
    ◢ __vfptr           0x0100da90 {Vtable.exe!const derivedClass::`vftable'{for `baseClass2'}} {0x010(        void * *
        [0]             0x0100158c {Vtable.exe!baseClass2::virtualNonOverriddenFunc2(void)}                    void *
        [1]             0x01001569 {Vtable.exe!baseClass2::virtualOverriddenFunc2(void)}                       void *
◢ bc1Ptr                0x007b6b60 {...}                                                                       baseClass1 *
  ◢ [derivedClass]      {...}                                                                                  derivedClass
    ◢ baseClass1        {...}                                                                                  baseClass1
      ◢ __vfptr         0x0100da64 {Vtable.exe!const derivedClass::`vftable'{for `baseClass1'}} {0x010(        void * *
          [0]           0x01001564 {Vtable.exe!baseClass1::virtualNonOverriddenFunc1(void)}                    void *
          [1]           0x01001587 {Vtable.exe!derivedClass::virtualOverriddenFunc1(void)}                     void *
    ◢ baseClass2        {...}                                                                                  baseClass2
      ◢ __vfptr         0x0100da90 {Vtable.exe!const derivedClass::`vftable'{for `baseClass2'}} {0x010(        void * *
          [0]           0x0100158c {Vtable.exe!baseClass2::virtualNonOverriddenFunc2(void)}                    void *
          [1]           0x01001569 {Vtable.exe!baseClass2::virtualOverriddenFunc2(void)}                       void *
  ◢ __vfptr             0x0100da64 {Vtable.exe!const derivedClass::`vftable'{for `baseClass1'}} {0x010(        void * *
      [0]               0x01001564 {Vtable.exe!baseClass1::virtualNonOverriddenFunc1(void)}                    void *
      [1]               0x01001587 {Vtable.exe!derivedClass::virtualOverriddenFunc1(void)}                     void *
```

There are a few interesting points here:

- *baseClass1Obj* and *baseClass2Obj* have their *__vfptr* and *__vftable* as expected. Four different functions with four different addresses. *__ vftables* are different too.
- *derivedClassObj* is a bit interesting. It has two subobjects of type *baseClass1* and *baseClass2*. This is expected as it is multiple inheritance. And these two base class objects have their own *__vfptrs*. And these point to different *__vftables* as *baseClass1Obj* and *baseClassObj2* are pointing at different tables. Also notice the function addresses. They point to the same functions as *baseClass2*, as no virtual functions of *baseClass2* is overridden. For *baseClass1* functions, the overridden function has a different address. Nothing new here. Just proving what we've been discussing earlier.

- Now look at the object pointed to by *dcPtr*. The *__vfptrs* and *__vftables* are exactly the same as *derivedClassObj*. Again, nothing surprising as we are creating and pointing to a *derivedClass.*
- But *bc1Ptr* is a bit different, though. There is a *derivedClass* component and a *__vfptr*. And if you check carefully you can see that the *__vftable* of *baseClass1* inside *derivedClass* is same as the one pointed to by *__vfptr* of *bc1Ptr*. What this means is that *bc1Ptr* has only ONE *__vftable*. You cannot access anything in the *derivedClass* part. It is there, just not accessible through the *baseClass1* pointer. So what this means is that if you have a base class pointer with a derived object, you can only access what is known by the base class.
- So to summarize, *dcPtr* has two *__vftables*, because *derivedClass* has two, but *bc1Ptr* has only one, which corresponds to what *baseClassObj* has, although both points to *derivedClassObj* objects.

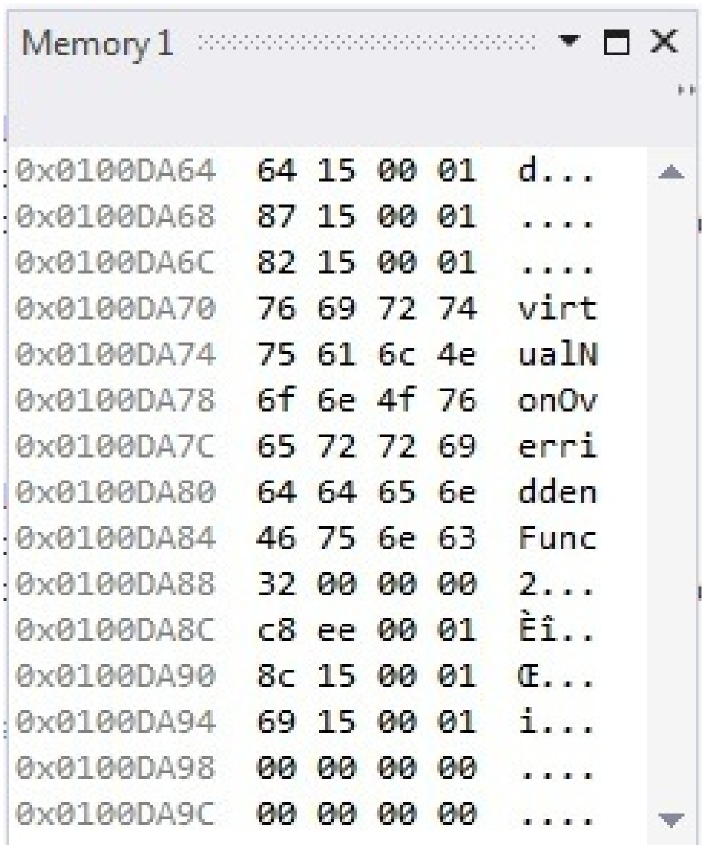
There is a little tidbit I left out in the above discussion. Whatever happened to *derivedClassOnlyVirtualFunction*?

This is a new virtual function in *derivedClass*. So obviously you won't be able to call it from *bc1Ptr*. But since this is a virtual function it must be in the *__vftable*. Go back and look at all the __ *vftable* entries there. You will not find it. There is no secret here. It just doesn't seem to show up there. And honestly, I don't know why, but I will prove to you it is there.

Here's the thing about *__vfptrs* and *__vftables*. This is not a standard. How the dynamic binding is performed is implementation dependent. The compiler is free to choose whatever method to make it work. Although most implementations will use *__vfptrs* and *__vftables*, they don't have to. And even when they use virtual function tables and pointers, they can implement it in different ways.

So, when we added the new virtual function *derivedClassOnlyVirtualFunc*, basically it can be supported by adding a new *__vfptr* and a new *__vftable*. Or it can be added to the existing *__vftable*. It all depends how the compiler and runtime systems looks for it. Then what is happening here? The new function is definitely not in any of the *__vftables*. Is there another *__vftable* that we can't see there? Well I had no idea what was happening and why it was happening. So I tried printing out the size of the *derivedClassObj*. If you remember from the earlier discussion, *__vfptr* has a size of 4 bytes. So I expected my 'invisible' *__vfptr* to make the object weigh 8 bytes. But it was just 4 bytes! So there really is just one *__vfptr*.

Now what I'm saying here is for the Visual Studio 2013 C++ compiler. So if you are using another compiler this could most probably be different. The way I tried to resolve this mystery is to look at the memory contents of *__vftable*. And here is what I saw when I looked at the memory space of the *derivedClassObj __vftable*:



```
0x0100DA64  64 15 00 01  d...
0x0100DA68  87 15 00 01  ....
0x0100DA6C  82 15 00 01  ....
0x0100DA70  76 69 72 74  virt
0x0100DA74  75 61 6c 4e  ualN
0x0100DA78  6f 6e 4f 76  onOv
0x0100DA7C  65 72 72 69  erri
0x0100DA80  64 64 65 6e  dden
0x0100DA84  46 75 6e 63  Func
0x0100DA88  32 00 00 00  2...
0x0100DA8C  c8 ee 00 01  Èî..
0x0100DA90  8c 15 00 01  Œ...
0x0100DA94  69 15 00 01  i...
0x0100DA98  00 00 00 00  ....
0x0100DA9C  00 00 00 00  ....
```

0x0100DA64 is the memory address of the *__vfptr* of *derivedClassObj*. And you can clearly see the first two entries are the addresses of the virtual functions: 0x01001564 and 0x01001587 (separated by bytes and backwards). If you look further you will see that there is a third entry too. Now it is my best guess that this third entry is pointing to *derivedClassOnlyVirtualFunc*. For some reason Visual Studio debugger is not showing this third entry. This is probably due to the *__vfptr* we are looking at is part of the *baseClass1*. (So why did it go to *baseClass1 __vfptr* and not *baseClass2 __vfptr*? I don't know. Like I said this mechanism is totally implementation dependent and I'm sure Visual C++ team had a valid reason for it)

Let me re-iterate. The mechanism of virtual functions and dynamic binding is implementation dependent. Virtual function tables and pointers are a widely known, or at least widely taught method of implementing it. What you saw is specific for Visual C++ compiler. But the fundamentals remain the same. Base class pointers can only know functions defined in it. If there are virtual functions then it can do dynamic binding at runtime and invoke the overridden function, but at compile time the compiler needs to know that function is defined in the base class. As you saw in the case of *bc1Ptr*, although a *derivedClass* object is in there, it is inaccessible through the base class pointer.

You'd understand that I cannot leave this like this. I owe it to you to prove that there indeed is a third entry in the *__vftable* that points to *derivedClassOnlyVirtualFunc*. One way to prove that the third entry in the table is our function is to simply call it with that address. Doing that isn't exactly straight forward but here's how you can do it:

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

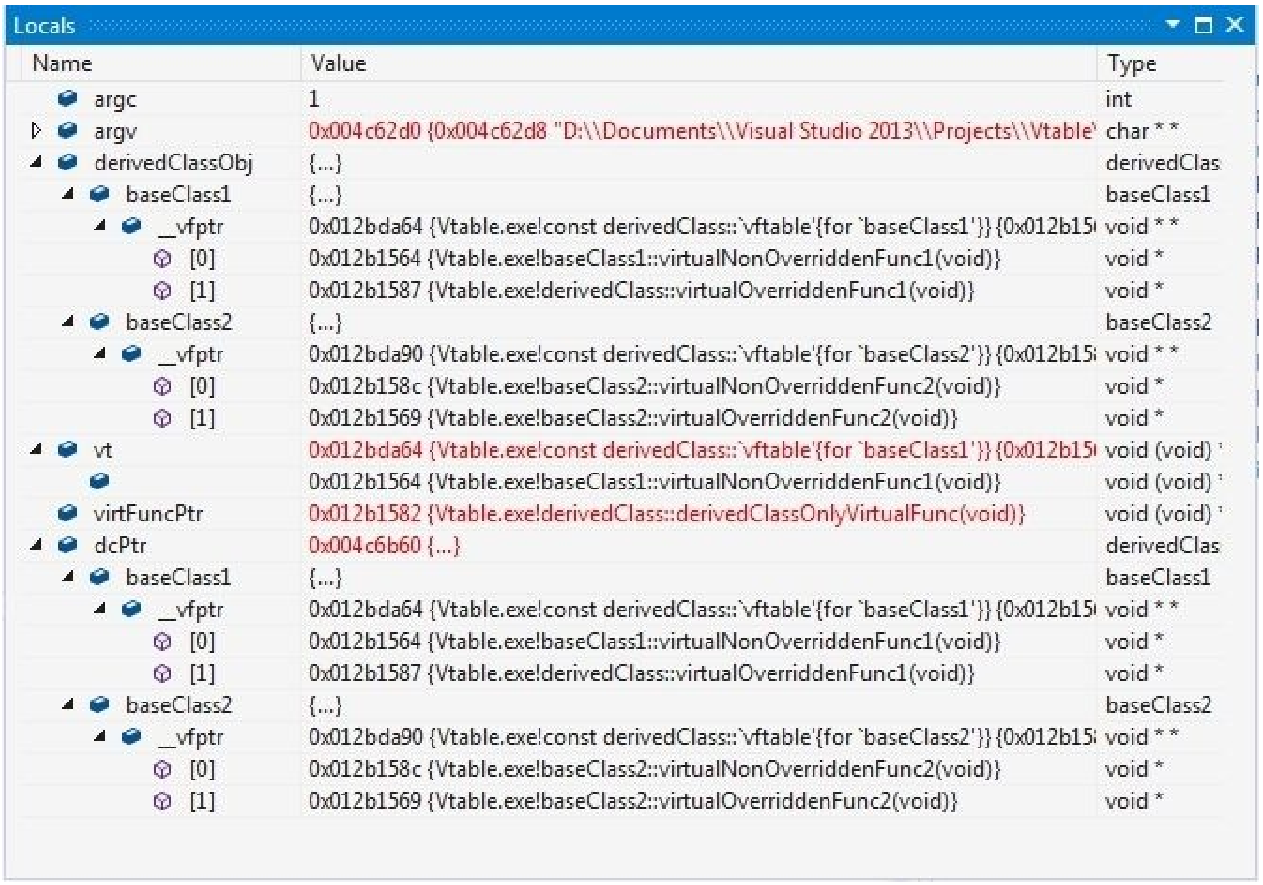
```
class baseClass1
{
public:
	void nonVirtualFunc1()
	{
		cout << “nonVirtualFunc1” << endl;
	}
	virtual void virtualNonOverriddenFunc1()
	{
		cout << “virtualNonOverriddenFunc1” << endl;
	}
	virtual void virtualOverriddenFunc1()
	{
		cout << “virtualOverriddenFunc1” << endl;
	}
};

class baseClass2
{
public:
	void nonVirtualFunc2()
	{
		cout << “nonVirtualFunc2” << endl;
	}
	virtual void virtualNonOverriddenFunc2()
	{
		cout << “virtualNonOverriddenFunc2” << endl;
	}
	virtual void virtualOverriddenFunc2()
	{
		cout << “virtualOverriddenFunc2” << endl;
	}
};

class derivedClass : public baseClass1, public baseClass2
{
public:
	virtual void virtualOverriddenFunc1()
	{
		cout << “virtualDerivedOverriddenFunc1” << endl;
	}
	virtual void derivedClassOnlyVirtualFunc()
	{
		cout << “derivedClassOnlyVirtualFunc” << endl;
	}
};



int main(int argc, char** argv)
{
	derivedClass derivedClassObj;
	derivedClass *dcPtr = new derivedClass;

	cout << “Invoking function through the object pointer…” << endl;
	dcPtr->virtualNonOverriddenFunc1();
	dcPtr->virtualOverriddenFunc1();
	dcPtr->derivedClassOnlyVirtualFunc();
	cout << endl;

	void(**vtPtr)() = *(void(***)())dcPtr; //obtaining __vftable address

	cout << “Printing __vftable…” << endl;
	cout << “__vftable address: “ << vtPtr << endl;
	cout << “__vftable[0] - “ << *vtPtr << endl;
	cout << “__vftable[1] - “ << *(vtPtr + 1) << endl;
	cout << “__vftable[2] - “ << *(vtPtr + 2) << endl;
	cout << endl;

	typedef void func(void);

	cout << “Invoking functions through __vftable…” << endl << endl;

	func* virtFuncPtr = (func*)(*vtPtr);	// pointing to the first virtual func.
	cout << “__vftable[0] - “;
	(virtFuncPtr());

	virtFuncPtr = (func*)(*(vtPtr + 1));	// pointing to the second virtual func.
	cout << “__vftable[1] - “;
	virtFuncPtr();

	virtFuncPtr = (func*)(*(vtPtr + 2));	// pointing to the third virtual func.
	cout << “__vftable[2] - “;
	virtFuncPtr();
```

```
        return 0;
}
```

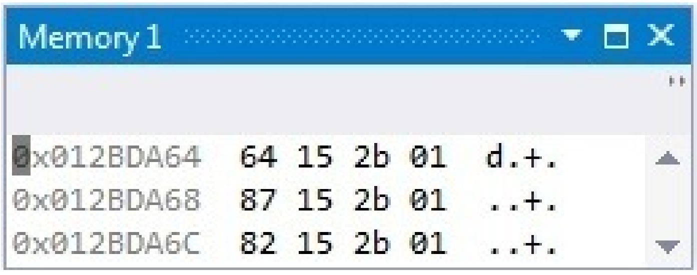
The only thing out of the ordinary here is that I'm obtaining a pointer to the *__vftable*. And then I'm simply going through its contents and invoking the functions pointed to by those entries. Here is the output.

```
Invoking function through the object pointer…
virtualNonOverriddenFunc1
virtualDerivedOverriddenFunc1
derivedClassOnlyVirtualFunc

Printing __vftable…
__vftable address: 012BDA64
__vftable[0] - 012B1564
__vftable[1] - 012B1587
__vftable[2] - 012B1582

Invoking functions through __vftable…

__vftable[0] - virtualNonOverriddenFunc1
__vftable[1] - virtualDerivedOverriddenFunc1
__vftable[2] - derivedClassOnlyVirtualFunc
```

And here are the debug windows.


Locals

Name	Value	Type
argc	1	int
argv	0x004c62d0 {0x004c62d8 "D:\\Documents\\Visual Studio 2013\\Projects\\Vtable'	char * *
derivedClassObj	{...}	derivedClas
baseClass1	{...}	baseClass1
_vfptr	0x012bda64 {Vtable.exe!const derivedClass::`vftable'{for `baseClass1'}} {0x012b15	void * *
[0]	0x012b1564 {Vtable.exe!baseClass1::virtualNonOverriddenFunc1(void)}	void *
[1]	0x012b1587 {Vtable.exe!derivedClass::virtualOverriddenFunc1(void)}	void *
baseClass2	{...}	baseClass2
_vfptr	0x012bda90 {Vtable.exe!const derivedClass::`vftable'{for `baseClass2'}} {0x012b15	void * *
[0]	0x012b158c {Vtable.exe!baseClass2::virtualNonOverriddenFunc2(void)}	void *
[1]	0x012b1569 {Vtable.exe!baseClass2::virtualOverriddenFunc2(void)}	void *
vt	0x012bda64 {Vtable.exe!const derivedClass::`vftable'{for `baseClass1'}} {0x012b15	void (void)
	0x012b1564 {Vtable.exe!baseClass1::virtualNonOverriddenFunc1(void)}	void (void)
virtFuncPtr	0x012b1582 {Vtable.exe!derivedClass::derivedClassOnlyVirtualFunc(void)}	void (void)
dcPtr	0x004c6b60 {...}	derivedClas
baseClass1	{...}	baseClass1
_vfptr	0x012bda64 {Vtable.exe!const derivedClass::`vftable'{for `baseClass1'}} {0x012b15	void * *
[0]	0x012b1564 {Vtable.exe!baseClass1::virtualNonOverriddenFunc1(void)}	void *
[1]	0x012b1587 {Vtable.exe!derivedClass::virtualOverriddenFunc1(void)}	void *
baseClass2	{...}	baseClass2
_vfptr	0x012bda90 {Vtable.exe!const derivedClass::`vftable'{for `baseClass2'}} {0x012b15	void * *
[0]	0x012b158c {Vtable.exe!baseClass2::virtualNonOverriddenFunc2(void)}	void *
[1]	0x012b1569 {Vtable.exe!baseClass2::virtualOverriddenFunc2(void)}	void *


Memory 1

```
0x012BDA64  64 15 2b 01  d.+.
0x012BDA68  87 15 2b 01  ..+.
0x012BDA6C  82 15 2b 01  ..+.
```

Let’s go through the output.

- First, part of the print out is invoking the functions through the pointer. There is nothing out of ordinary here.
- The next part is the *__vftable* address and its contents. Correspond the table address with that of the debug window: 0x012BDA64. So now we know that we have access to the correct *__vftable*.
- Then we are printing out the contents of the *__vftable*. These are the virtual function addresses. You can confirm that the first two are exactly the same between the print out and the debug window *__vftable* contents.
- Our problem is with the third entry, which is not showing up in the debug window. But look at the memory contents and the printout. They match. So now we are certain that we have access to the function in the third entry of the *__vftable*.
- The last part of the printout is calling the functions through the *__vftable* entries. Compare the printous with the first printouts. They are the same. So now we are absolutely certain that there is indeed a third entry in the *__vftable* and it does correspond to the new virtual function *derivedClassOnlyVirtualFunc*.

Don’t worry if you didn’t understand the code completely. I will be explaining this code in the chapter on “Function Pointers”. One point to make clear about this code is that it assumes the *__vfptr* pointer is at the very beginning of the object and that is apparently how Visual C++ implements it.

Construction of virtual tables

I hope you got a good fundamental understanding of the implementation of virtual mechanism (at least how it is done in Visual C++). Before we finish off this topic, let’s quickly look at how *vptrs* are built during object construction.

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

class baseClass
{
public:
        baseClass()
        {
                void(**vtPtr)() = *(void(***)())this;
                cout << “baseClass vptr:” << vtPtr << endl;
                virtualPrintFunc();
        }

        virtual void virtualPrintFunc() { cout << “baseClass::virtualFunc” << endl; }
};


class derivedClass : public baseClass
{
public:
        derivedClass()
        {
                void(**vtPtr)() = *(void(***)())this;
                cout << “derivedClass constructor. vptr:” << vtPtr << endl;
                virtualPrintFunc();
        }

        virtual void virtualPrintFunc() { cout << “derivedClass::virtualFunc” << endl; }
};
```

```
int main(int argc, char** argv)
{
        derivedClass derivedClassObj;
        return 0;
}
```

```
baseClass vptr:002B32C4
baseClass::virtualFunc
derivedClass constructor. vptr:002B32BC
derivedClass::virtualFunc
```

We are doing something pretty simple. We have a base class and a derived class and a virtual function. In the constructors we print out the *vptr* (the same as we did in the previous example) and we call the virtual function. There are two very important points to make here.

- First, you see that when the virtual function is called from each of the constructors, they call the function defined in that class. Why is this? It's pretty simple and goes back to the fundamental mechanism of object construction. Objects are constructed from the most base class. So when we called *derivedClass* cosntructor, it first called the *baseClass* constructor before cosntructing the *derivedClass* part. So when we are in the *baseClass* constructor, the *derivedClass* part does not exist at that time. It's only *baseClass* and its virtual *PrintFunc*. So when it calls the virtual function, it only has one implementation of the function.
- Second, the *vptrs* change during object construction. When in the *baseClass* constructor it had a certain *vptr*, pointing to a *vtable* that has the *virtualPrintFunc* implementation that prints out "*baseClass::virtualFunc*". When the *baseClass* is constructed and the execution moved to *derivedClass* constructor, it defined its own *vptr*, now pointing to a *vtable* that has the *virtualPrintFunc* that prints out "*derivedClass::virtualFunc*". Note that both these constructors use the same *'*this'* object.

So during the construction of an object with a hierarchy, the object goes through different *vptrs* until finally it gets the *vptr* of the most derived class. During construction of each level of the object, the compiler assigns it a *vptr*. And this is generated before the constructor body is entered. So there are two main tasks the compiler needs to do before the constructor body is entered:

- Call the base class constructor (if it's a derived class)
- Define the *vptr* and *vtable*

Let me say this one last time. Implementation of the virtual mechanism is completely compiler dependent. And where the *vptr* is allocated and when it is generated, these are all compiler dependent. But for most part, many compilers use a similar *vptr/vtable* mechanism, so what we discussed here is very important to understand the object construction and virtual dispatch mechanism.

One final word. Virtual mechanism, or polymorphism, can only be invoked through pointers and references. Invoking functions through objects, as we did here, does not do dynamic binding of the functions. But we used objects to show its contents and the composition of the *vptr*.

Topic 3

Structs, Classes and their Inheritance

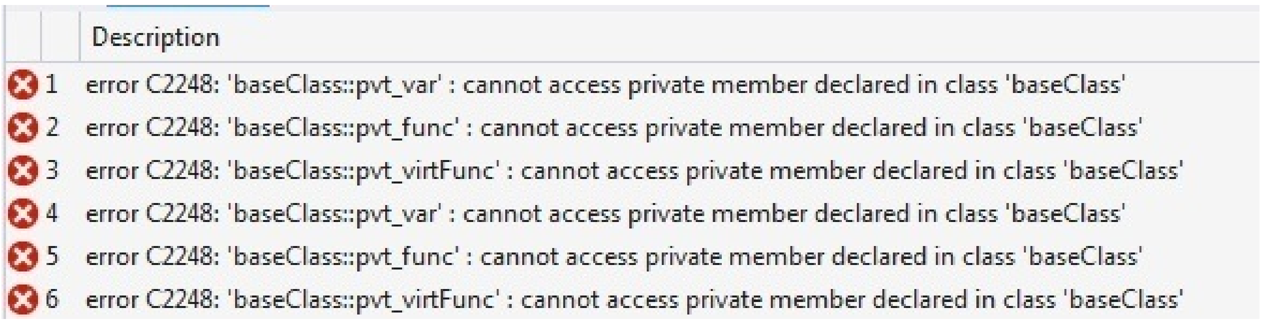
In this topic we will look at some cases of inheritance that we don't see everyday. For example, we will see what inheritance between a struct and a class will look like and also other fundamental inheritance types such as private and virtual inheritance.

Private inheritance

We are all too familiar with public inheritance of a class, but what does it mean to privately inherit? We will be using this simple example to prove our points.

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

class baseClass
{
        int pvt_var = 1;
        void pvt_func() {}
        virtual   void pvt_virtFunc() {}

public:
        int pub_var = 1;
        void pub_func() {}
        virtual void pub_virtFunc() {}
};

class pvtDerivedClass : baseClass
{
        void checkMemberAccess()
        {
                pvt_var = 2;
                pub_var = 2;
                pvt_func();
                pvt_virtFunc();
                pub_func();
                pub_virtFunc();
        }
};

class pubDerivedClass : public baseClass
{
    void checkMemberAccess()
    {
        pvt_var = 2;
        pub_var = 2;
        pvt_func();
        pvt_virtFunc();
        pub_func();
        pub_virtFunc();
    }
};

int main(int argc, char** argv)
{
    baseClass baseClassObj;
    pvtDerivedClass pvtDerivedClassObj;
    pubDerivedClass pubDerivedClassObj;

    return 0;
}
```

Compile and you will get errors similar to this:



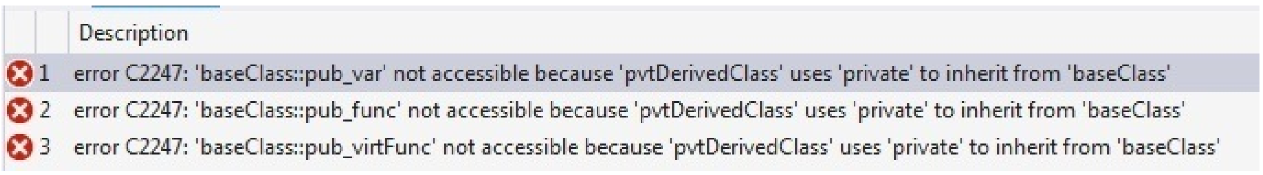
```
Description
1 error C2248: 'baseClass::pvt_var' : cannot access private member declared in class 'baseClass'
2 error C2248: 'baseClass::pvt_func' : cannot access private member declared in class 'baseClass'
3 error C2248: 'baseClass::pvt_virtFunc' : cannot access private member declared in class 'baseClass'
4 error C2248: 'baseClass::pvt_var' : cannot access private member declared in class 'baseClass'
5 error C2248: 'baseClass::pvt_func' : cannot access private member declared in class 'baseClass'
6 error C2248: 'baseClass::pvt_virtFunc' : cannot access private member declared in class 'baseClass'
```

If you follow the error messages you will see that the compiler is complaining about the access of the three private members of *baseClass*. So the point I want to make in this example is, the inheritance type has no influence on what you can access in the class. All the private members of the base class are still private and all the public members are public. This is true regardless of whether the class is derived privately or publicly.

Then what does the inheritance type affect? It affects how the base class objects can be accessed from outside. Look at this example:

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

class baseClass
{
        int pvt_var = 1;
        void pvt_func() {}
        virtual   void pvt_virtFunc() {}

public:
        int pub_var = 1;
        void pub_func() {}
        virtual void pub_virtFunc() {}
};

class pvtDerivedClass : baseClass
{};

class pubDerivedClass : public baseClass
{};

int main(int argc, char** argv)
{
        baseClass baseClassObj;
        pvtDerivedClass pvtDerivedClassObj;
        pubDerivedClass pubDerivedClassObj;

        baseClassObj.pub_var;
        baseClassObj.pub_func();
        baseClassObj.pub_virtFunc();

        pvtDerivedClassObj.pub_var;
        pvtDerivedClassObj.pub_func();
        pvtDerivedClassObj.pub_virtFunc();

        pubDerivedClassObj.pub_var;
        pubDerivedClassObj.pub_func();
        pubDerivedClassObj.pub_virtFunc();

        return 0;
}
```

You will get compiler errors along these lines:



```
Description
1  error C2247: 'baseClass::pub_var' not accessible because 'pvtDerivedClass' uses 'private' to inherit from 'baseClass'
2  error C2247: 'baseClass::pub_func' not accessible because 'pvtDerivedClass' uses 'private' to inherit from 'baseClass'
3  error C2247: 'baseClass::pub_virtFunc' not accessible because 'pvtDerivedClass' uses 'private' to inherit from 'baseClass'
```

The compiler errors are for *pvtDerivedClassObj* only. Both *pvtDerivedClass* and *pubDerivedClass* derive from *baseClass*. And in the example, both objects try to access the public methods of *baseClass*. But *pvtDerivedClassObj* is not allowed to access the public members of *baseClass*. So it is obvious what is happening here. When you derive privately from a class, all inherited base class members act as private members defined in the derived class. The public members are still accessible within the derived class, but from outside, they act as if they are private members in the derived class. A similar effect is happening to public as well as protected inheritance.

It's common knowledge that in the context of C++, structs and classes are almost identical in their functionality and the primary difference is that structs have a default access level of public, while classes are private. So how does this affect during inheritance? Let's change all of our classes to structs and see.

```
struct baseClass
{
private:
        int pvt_var = 1;
        void pvt_func() {}
        virtual   void pvt_virtFunc() {}

public:
        int pub_var = 1;
        void pub_func() {}
        virtual void pub_virtFunc() {}
};

struct pvtDerivedClass : baseClass
{};

struct pubDerivedClass : public baseClass
{};

int main(int argc, char** argv)
{
        baseClass baseClassObj;
        pvtDerivedClass pvtDerivedClassObj;
        pubDerivedClass pubDerivedClassObj;

        baseClassObj.pub_var;
        baseClassObj.pub_func();
        baseClassObj.pub_virtFunc();

        pvtDerivedClassObj.pub_var;
        pvtDerivedClassObj.pub_func();
        pvtDerivedClassObj.pub_virtFunc();

        pubDerivedClassObj.pub_var;
        pubDerivedClassObj.pub_func();
        pubDerivedClassObj.pub_virtFunc();

        return 0;
}
```

Note that I made a small modification to the *baseClass* and put private access modifier, because otherwise all members will be public. Compile this and you will see there are no compilation errors. This is because when you inherit a struct from a struct, the default inheritance is public. Same as default access level for struct members. So *pvtDerivedClass* is actually inheriting public, not private as it is with classes. Just to prove our point we'll explicitly specify the private inheritance for *pvtDerivedClass*.

```
struct baseClass
{
private:
        int pvt_var = 1;
        void pvt_func() {}
        virtual void pvt_virtFunc() {}

public:
        int pub_var = 1;
        void pub_func() {}
        virtual void pub_virtFunc() {}
};

struct pvtDerivedClass : private baseClass
{};

struct pubDerivedClass : public baseClass
{};

int main(int argc, char** argv)
{
        baseClass baseClassObj;
        pvtDerivedClass pvtDerivedClassObj;
        pubDerivedClass pubDerivedClassObj;

        baseClassObj.pub_var;
        baseClassObj.pub_func();
        baseClassObj.pub_virtFunc();

        pvtDerivedClassObj.pub_var;
        pvtDerivedClassObj.pub_func();
        pvtDerivedClassObj.pub_virtFunc();

        pubDerivedClassObj.pub_var;
        pubDerivedClassObj.pub_func();
        pubDerivedClassObj.pub_virtFunc();

        return 0;
}
```

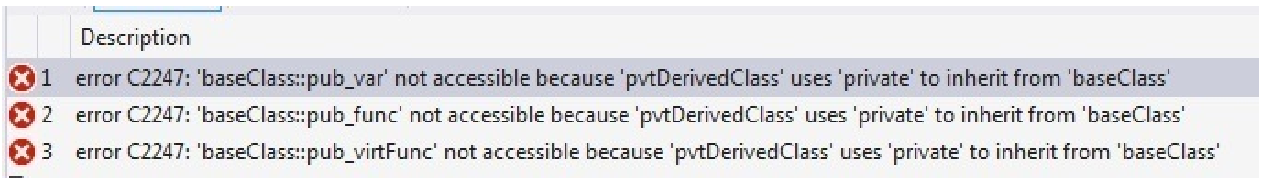
Compile this and you will see that you get the same compilation errors you got with the class version. So that's the difference when you inherit structs.

Inheritance between structs and classes

So naturally you are now intrigued to know what happens when we inherit one from the other, right? Let's find out. First we will derive a class from a struct base and explicitly specify the access level.

```
struct baseClass
{
private:
        int pvt_var = 1;
        void pvt_func() {}
        virtual   void pvt_virtFunc() {}

public:
        int pub_var = 1;
        void pub_func() {}
        virtual void pub_virtFunc() {}
};

class pvtDerivedClass : private baseClass
{};
```

```
class pubDerivedClass : public baseClass
{};

int main(int argc, char** argv)
{
        baseClass baseClassObj;
        pvtDerivedClass pvtDerivedClassObj;
        pubDerivedClass pubDerivedClassObj;

        baseClassObj.pub_var;
        baseClassObj.pub_func();
        baseClassObj.pub_virtFunc();

        pvtDerivedClassObj.pub_var;
        pvtDerivedClassObj.pub_func();
        pvtDerivedClassObj.pub_virtFunc();

        pubDerivedClassObj.pub_var;
        pubDerivedClassObj.pub_func();
        pubDerivedClassObj.pub_virtFunc();

        return 0;
}
```

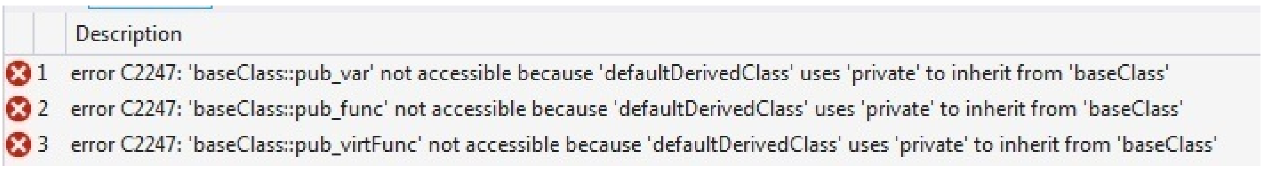


```
Description
1   error C2247: 'baseClass::pub_var' not accessible because 'pvtDerivedClass' uses 'private' to inherit from 'baseClass'
2   error C2247: 'baseClass::pub_func' not accessible because 'pvtDerivedClass' uses 'private' to inherit from 'baseClass'
3   error C2247: 'baseClass::pub_virtFunc' not accessible because 'pvtDerivedClass' uses 'private' to inherit from 'baseClass'
```

You will remember that these are the same errors we got with classes. So with private and public inheritance, there is nothing different in the inheritance of members when deriving a class from a struct. Change the base to a class and the derived one struct and you will see the same compilation errors. So when the inheritance level is specified, classes and structs act the same way. Then, what would be the case if we don't specify? What is the default level?

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

struct baseClass
{
private:
    int pvt_var = 1;
    void pvt_func() {}
    virtual   void pvt_virtFunc() {}

public:
    int pub_var = 1;
    void pub_func() {}
    virtual void pub_virtFunc() {}
};

class defaultDerivedClass : baseClass
{};

int main(int argc, char** argv)
{
    defaultDerivedClass defaultDerivedClassObj;

    defaultDerivedClassObj.pub_var;
    defaultDerivedClassObj.pub_func();
    defaultDerivedClassObj.pub_virtFunc();

    return 0;
}
```

These are the compilation errors you'd get:



```
Description
1  error C2247: 'baseClass::pub_var' not accessible because 'defaultDerivedClass' uses 'private' to inherit from 'baseClass'
2  error C2247: 'baseClass::pub_func' not accessible because 'defaultDerivedClass' uses 'private' to inherit from 'baseClass'
3  error C2247: 'baseClass::pub_virtFunc' not accessible because 'defaultDerivedClass' uses 'private' to inherit from 'baseClass'
```

You will realize that these are the same errors we got with private inheritance. So when we are inheriting from a struct to a derived class, the default access specifier is similar to that of class. What about the other way then?

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

class baseClass
{
private:
    int pvt_var = 1;
    void pvt_func() {}
    virtual   void pvt_virtFunc() {}
public:
	int pub_var = 1;
	void pub_func() {}
	virtual void pub_virtFunc() {}
};

struct defaultDerivedStruct : baseClass
{};

int main(int argc, char** argv)
{
	defaultDerivedStruct defaultDerivedStructObj;

	defaultDerivedStructObj.pub_var;
	defaultDerivedStructObj.pub_func();
	defaultDerivedStructObj.pub_virtFunc();

	return 0;
}
```

You will get no compilation errors. That means when you are inheriting from a class to a derived struct, the default access specifier is that of a struct.

There is nothing special or tricky about this topic but nevertheless it is a topic that is often left out.

Topic 4

Object Construction

In this topic we will look at how class objects are constructed. There are a few ways objects can be constructed. Some are explicit and some implicit. Here is a standard class definition:

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

class standardClass
{
	int objID;

public:

	standardClass(int ID) // constructor
	{
		objID = ID;
		cout << “Constructor: “ << objID << endl;
	}

	~standardClass() // destructor
	{
		cout << “Destructor: “ << objID << endl;
	}

	standardClass(const standardClass &objToCopy)		// copy constructor
	{
		objID = objToCopy.objID;
	cout << “Copy constructor: “ << objID << endl;
}

standardClass & operator=(const standardClass &objToCopy) // assignment operator
{
	cout << “Copy assignment operator: “ << objID << endl;
	objID = objToCopy.objID;
	return *this;
}
};
```

```
int main(int argc, char** argv)
{
        standardClass stdClassObj1(1);                    // Line #1
        cout << "- Line 1 end -" << endl;

        standardClass stdClassObj2(stdClassObj1);         // Line #2
        cout << "- Line 2 end -" << endl;

        standardClass stdClassObj3 = stdClassObj2;        // Line #3
        cout << "- Line 3 end -" << endl;

        stdClassObj1 = stdClassObj2;                      // Line #4
        cout << "- Line 4 end -" << endl;

        stdClassObj2 = standardClass(4);                  // Line #5
        cout << "- Line 5 end -" << endl;

        stdClassObj3 = 5;                                 // Line #6
        cout << "- Line 6 end -" << endl;

        standardClass stdClassObj4 = standardClass(4);    // Line #7
        cout << "- Line 7 end -" << endl;

        standardClass stdClassObj5 = 5;                   // Line #8
        cout << "- Line 8 end -" << endl << endl;

        return 0;
}
```

```
Constructor: 1
- Line 1 end -
Copy constructor: 1
- Line 2 end -
Copy constructor: 1
- Line 3 end -
Copy assignment operator: 1
- Line 4 end -
Constructor: 4
Copy assignment operator: 1
Destructor: 4
- Line 5 end -
```

```
Constructor: 5
Copy assignment operator: 1
Destructor: 5
- Line 6 end -
Constructor: 4
- Line 7 end -
Constructor: 5
- Line 8 end -

Destructor: 5
Destructor: 4
Destructor: 5
Destructor: 4
Destructor: 1
```

Let's quickly go over what is happening here.

- Line 1 creates a new object by calling the constructor.
- Line 2 passes an object to create a new object. This calls the copy constructor. Notice that in the case when the copy constructor is called, the default constructor is not called anywhere.
- Line 3 creates a new object by assigning an existing object. But this calls the copy constructor. NOT the assignment operator. This is because we are creating a new object *stdClassObj3*. So line 3 has the same effect as line 2.
- Line 4 invokes the assignment operator. This is because we are assigning *stdClassObj2* to *stdClassObj1*, which is already created. So keep in mind that the assignment operator is only called when you assign to an existing object.
- Line 5 assigns existing *stdClassObj2* to a newly created object. So this is what is happening in this case:
    - A new temporary object is created by calling the constructor with value 4.
    - This object is assigned to *stdClassObj2* using assignment operator.
    - The destructor is called for the temporary object.
- Line 6 assigns *stdClassObj3* a new object by passing an integer. This is completely fine as we have a constructor that takes an integer parameter. So line 6 is equivalent to "*stdClassObj3 = standardClass(5)*", which is same as line 5. And indeed the output is same as that for line 5.
- Now look at what happens at lines 7 and 8. We are doing the same operation as we did in lines 5 and 6, in that respective order, but now we are creating two new objects. In lines 5 and 6 we did assignments to existing objects. But now we are instantiating new objects. There are no temporary object constructions and calls

to copy constructors. The compiler is smart to understand that it can directly call the constructor as this is a new object instance. So keep in mind how the object creation differs between new objects and assignments to existing ones.

The reason I put line printouts in the middle is to show the boundary between different statements and also to show the point when the implicit object destruction before the exit of the *main* function is happening. After the line 6 printout, the function will return and this invokes the destruction of the stack objects. Everything we created in the main are local to the *main* function and hence they are in the stack. These stack objects will be destroyed by calling their destructors at the end of the function.

- Finally, all of the destructors of the objects are called. You should note that the objects are destroyed in the inverse order they were created. Objects created last are destroyed first. So in our case *stdClassObj3* is destroyed first and then *stdClassObj2* and so on. Note they are destroyed in the order they were “created.” Not the order they are assigned.

Now let’s define some functions and see how this mechanism plays out.

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

class standardClass
{
        int objID;

public:
        standardClass(int ID) // constructor
        {
                objID = ID;
                cout << “Constructor: “ << objID << endl;
        }

        ~standardClass() // destructor
        {
                cout << “Destructor: “ << objID << endl;
        }

        standardClass(const standardClass &objToCopy) // copy constructor
        {
                objID = objToCopy.objID;
                cout << “Copy constructor: “ << objID << endl;
        }
        standardClass & operator=(const standardClass &objToCopy) // assignment operator
        {
                cout << “Copy assignment operator: “ << objID << endl;
                objID = objToCopy.objID;
                return *this;
        }

        void funcCallByVal(standardClass stdClass)
        {
                objID = stdClass.objID;
        }

        void funcCallByRef(standardClass &stdClass)
        {
                objID = stdClass.objID;
        }

        void funcCallByPtr(standardClass *stdClass)
        {
                objID = stdClass->objID;
        }
};

int main(int argc, char** argv)
{
        standardClass stdClassObj1(1);                  // Line #1
        standardClass stdClassObj2(2);                  // Line #2
        cout << “-Line #2 end-” << endl;

        stdClassObj1.funcCallByVal(stdClassObj2);       // Line #3
        cout << “-Line #3 end-” << endl;

        stdClassObj1.funcCallByVal(3);                   // Line #4
        cout << “-Line #4 end-” << endl;

        return 0;
}
```

```
Constructor: 1
Constructor: 2
-Line #2 end-
```

```
Copy constructor: 2
Destructor: 2
-Line #3 end-

Constructor: 3
Destructor: 3
-Line #4 end-

Destructor: 2
Destructor: 3
```

So I defined three functions. The first one takes *standardClass* argument by value, the second by reference and the third as a pointer. For clarity we'll go through the functions separately, and first I am invoking *funcCallByValue*.

Pass by value

I have put printouts of line endings to make it clear what constructors are being called at each statement. Let's go through the results.

- Lines 1 and 2 create two objects of *standardClass*. So the constructor is called twice.
- In line 3 we are invoking *funcCallByValue* and passing it to the object we created. Here we see the copy constructor is called first. This is to create the copy of *stdClassObj2*. This is a temporary object that is passed to the function. When the function exits this temporary is destroyed. That is what you see in the next printout. This is similar to what we saw in the previous example in lines 2 and 3 when new objects are created with existing objects.
- In line 4 we are calling *funcCallByVal* with just an integer. This is fine. Since there is a constructor for *standardClass* that takes an integer, the compiler makes line 4 look like *"stdClassObj1.funcCallByVal(standardClass(3))"*. Try it out and you will get the same result. So in this case we are creating a new object and passing it to the function. This is evident by the output where we see the constructor is being called.
- In any case, when calling *funcCallByVal*, a temporary object is created. In line 3 this temporary is created by calling the copy constructor because we are 'copying' an existing object. But in line 4 we are creating the temporary with the constructor because it is passed as an integer argument. This is the exact same mechanism we saw in line 6 of the previous example.

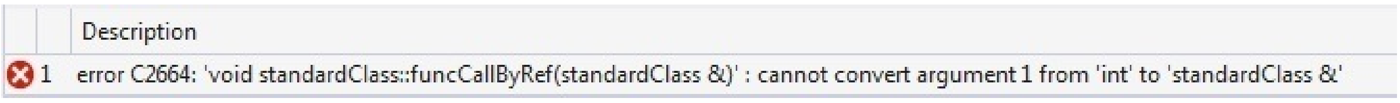
Pass by reference

Now let's move on to the pass by reference. I will only change the function name and keep the arguments as they are.

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

class standardClass
{
        int objID;

public:
        standardClass(int ID) // constructor
        {
                objID = ID;
                cout << “Constructor: “ << objID << endl;
        }

        ~standardClass() // destructor
        {
                cout << “Destructor: “ << objID << endl;
        }

        standardClass(const standardClass &objToCopy) // copy constructor
        {
                objID = objToCopy.objID;
                cout << “Copy constructor: “ << objID << endl;
        }

        standardClass & operator=(const standardClass &objToCopy) // assignment operator
        {
                cout << “Copy assignment operator: “ << objID << endl;
                objID = objToCopy.objID;
                return *this;
        }

        void funcCallByVal(standardClass stdClass)
        {
                objID = stdClass.objID;
        }

        void funcCallByRef(standardClass &stdClass)
        {
                objID = stdClass.objID;
```

```
    }

    void funcCallByPtr(standardClass *stdClass)
    {
        objID = stdClass->objID;
    }
};

int main(int argc, char** argv)
{
    standardClass stdClassObj1(1);              // Line #1
    standardClass stdClassObj2(2);              // Line #2
    cout << "-Line #2 end-" << endl;

    stdClassObj1.funcCallByRef(stdClassObj2);   // Line #3
    cout << "-Line #3 end-" << endl;

    stdClassObj1.funcCallByRef(3);               // Line #4
    cout << "-Line #4 end-" << endl;

    return 0;
}
```

Compile and you will most likely see this error:

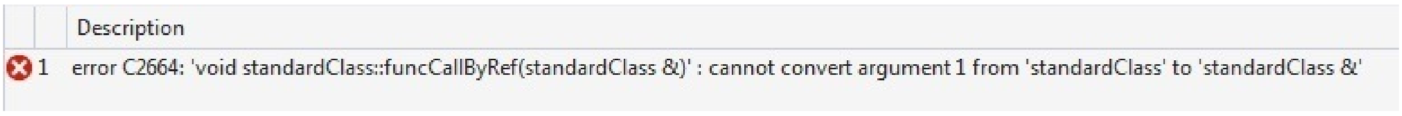


```
Description
1 error C2664: 'void standardClass::funcCallByRef(standardClass &)' : cannot convert argument 1 from 'int' to 'standardClass &'
```

Before I explain why, make this small modification to the function call in line 4. Pass an object argument like this:

```
…
stdClassObj1.funcCallByRef(standardClass(3));  // Line #4
…
```

If you are using Visual Studio chances are that this compiles fine. But truthfully this does not adhere to the standard. Compiler vendors add their own extensions to the standard specifications and this seems to be one of Visual C++'s ones. What you need to do is go to the project properties, choose C/C++ category and select *All Options*. In there you will see "*Disable language extensions*." Select Yes. Now try compiling again. You should see a similar error as before:



```
Description
1  error C2664: 'void standardClass::funcCallByRef(standardClass &)' : cannot convert argument 1 from 'standardClass' to 'standardClass &'
```

Now that we have a consistent error case, why is this happening? Why is it saying that it cannot pass a *standardClass* object to a reference? After all, aren't we doing the same thing in line 3 by passing *stdClassObj2*? So what is the difference?

Temporaries and const references

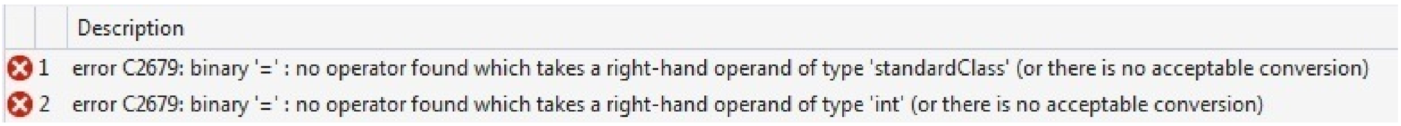
Before looking in to that, let me ask you, did you wonder why the copy constructor and copy assignment operator take "*const*" arguments? Why do the parameters need to be references to const?

Let us redo that example again, this time without the const references. Also if you are using Visual Studio make sure you have the language extensions disabled.

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

class standardClass
{
        int objID;

public:
        standardClass(int ID) // constructor
        {
                objID = ID;
                cout << "Constructor: " << objID << endl;
        }

        ~standardClass() // destructor
        {
                cout << "Destructor: " << objID << endl;
        }

        standardClass(standardClass &objToCopy) // copy constructor
        {
                objID = objToCopy.objID;
                cout << "Copy constructor: " << objID << endl;
        }

        standardClass & operator=(standardClass &objToCopy) // assignment operator
        {
                cout << "Copy assignment operator: " << objID << endl;
                objID = objToCopy.objID;
```

```
            return *this;
        }
};

int main(int argc, char** argv)
{
        standardClass stdClassObj1(1);                   // Line #1
        cout << "- Line 1 end -" << endl;

        standardClass stdClassObj2(stdClassObj1);      // Line #2
        cout << "- Line 2 end -" << endl;

        standardClass stdClassObj3 = stdClassObj2;    // Line #3
        cout << "- Line 3 end -" << endl;

        stdClassObj1 = stdClassObj2;                    // Line #4
        cout << "- Line 4 end -" << endl;

        stdClassObj2 = standardClass(4);               // Line #5
        cout << "- Line 5 end -" << endl;

        stdClassObj3 = 5;                               // Line #6
        cout << "- Line 6 end -" << endl;

        return 0;
}
```

You should get the following compilation errors:



```
Description
1  error C2679: binary '=' : no operator found which takes a right-hand operand of type 'standardClass' (or there is no acceptable conversion)
2  error C2679: binary '=' : no operator found which takes a right-hand operand of type 'int' (or there is no acceptable conversion)
```

So we are having compilation errors in lines 5 and 6. But notice that line 4 is almost the same as what we do in lines 5 and 6. In line 4 we are calling the copy assignment operator with an existing object, and in lines 5 and 6 we are passing temporary objects. So what is the problem here? Here is the output result we got in the previous case when we had the const specifier.

```
Constructor: 1
- Line 1 end -
Copy constructor: 1
```

```
- Line 2 end -
Copy constructor: 1
- Line 3 end -
Copy assignment operator: 1
- Line 4 end -
Constructor: 4
Copy assignment operator: 1
Destructor: 4
- Line 5 end -
Constructor: 5
Copy assignment operator: 1
Destructor: 5
- Line 6 end -
Destructor: 5
Destructor: 4
Destructor: 1
```

What does line 5 do? It first constructs an object, and then passes that object to the copy assignment operator of *stdClassObj2* and then calls the destructor of the created object. The object we created to pass to the copy assignment operator is a temporary. Herein lies the problem. Because the temporary we created in line 5 by calling *standardClass(4)* is an *rvalue*. There is no storage allocated for it. It is not assigned to any variable. That makes it an rvalue. And this is the problem. Because the standard says that you cannot bind an rvalue to a non-const reference. To say it in another way, only lvalues can be bounded to non-const references

So that is our problem. We are creating a temporary rvalue and passing it to be bound to *objToCopy* non-const reference. This is in violation of the standard and hence the compiler error. Think for a moment why it would be a problem if we could bind a temporary to a reference. A reference is basically an alias for the object. So unlike passing by value, which creates a new object, pass by reference does not. That is why it is efficient. Then think what would happen if we could bind a temporary to a reference. What would happen when the temporary gets destroyed? Because temporaries go out of scope soon. You will be left with a dangling reference. Then what happens when we make the reference const? In that case the standard specifically states that when a temporary is bound to a const reference, the lifetime of it is extended until the reference goes out of scope.

That is the reason why everything works when you put the const specifier. So then, you wouldn't need the const if you are not going to invoke them through statements like lines 5 and 6? If you comment out lines 5 and 6 and remove the const specifier you will see that you get no compiler errors. The compiler wouldn't complain to you that the copy constructor and copy assignment operator are taking non-const parameters. You are free to define those functions like that. But keep in mind that copy constructors and copy assignment operators must not modify the reference being passed. The task of those functions is to copy the object being passed. Not modifying anything in it. Making the parameter const makes sure that your function does not do anything undesirable to the object that is being passed to it.

Now go ahead and change the argument to *funcCallByRef* to a const reference. This time you should get no compiler errors for line 4. Because the temporary created by the passed argument, *standardClass* object with integer 3, can now bind to the const reference. So no rules are broken.

Let’s go ahead and finish the pass by pointer and wrap up this topic.

Passing the pointer

Just to be clear, there is no concept as pass by pointer. There are only pass by value and pass by reference. Pass by pointer is essentially pass by value. We are passing a copy of the arguments’ address as the value.

This is the best you could do with this code:

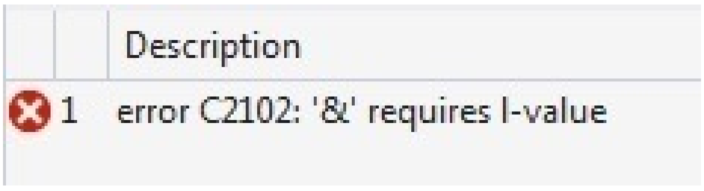
```
#include<iostream>
using namespace std;

class standardClass
{
	int objID;

	public:
	standardClass(int ID) // constructor
	{
		objID = ID;
		cout << “Constructor: “ << objID << endl;
	}

	~standardClass() // destructor
	{
		cout << “Destructor: “ << objID << endl;
	}

	standardClass(const standardClass &objToCopy) // copy constructor
	{
		objID = objToCopy.objID;
		cout << “Copy constructor: “ << objID << endl;
	}

	standardClass & operator=(const standardClass &objToCopy) // assignment operator
```

```
        {
                cout << “Copy assignment operator: “ << objID << endl;
                objID = objToCopy.objID;
                return *this;
        }

        void funcCallByVal(standardClass stdClass)
        {
                objID = stdClass.objID;
        }

        void funcCallByRef(const standardClass &stdClass)
        {
                objID = stdClass.objID;
        }

        void funcCallByPtr(const standardClass *stdClass)
        {
                objID = stdClass->objID;
        }
};

int main(int argc, char** argv)
{
        standardClass stdClassObj1(1);                  // Line #1
        standardClass stdClassObj2(2);                  // Line #2

        cout << “-Line #2 end-” << endl;

        stdClassObj1.funcCallByPtr(&stdClassObj2);      // Line #3

        cout << “-Line #3 end-” << endl;

        //stdClassObj1.funcCallByPtr(3);                // Line #4

        cout << “-Line #4 end-” << endl;

        return 0;
}
```

Since a pointer is a variable containing an address of what it points to, when we do not have a pointer itself, we need to pass the address of the object we need to pass. Therefore in line 3 we pass the address of the object. And as for line 4, there is no way we can pass an integer and make the compiler implicitly call the constructor. Why can't we do this?

```
…
stdClassObj1.funcCallByPtr(3);                    // Line #4
```

You will get an compiler error along the lines of:



		Description
✖	1	error C2102: '&' requires l-value

What we are trying to do is to get the address of a temporary. This temporary is not an lvalue and we are not allowed to take the address of non-lvalues. Hence the error.

Explicit constructor

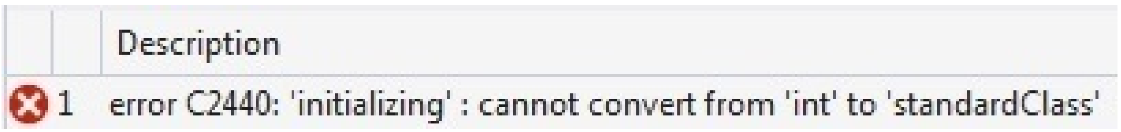
Let's finish this topic with a discussion on the '*explicit*' keyword, which was added since C++11. Before we discuss 'explicit' let's revisit our first example (with a bit of trimming).

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

class standardClass
{
        int objID;

public:

        standardClass(int ID)
        {
                objID = ID;
                cout << "Constructor: " << objID << endl;
        }

};

int main(int argc, char** argv)
{
        standardClass stdClassObj1(1);                // Line #1
        standardClass stdClassObj2 = 2;               // Line #2

        return 0;
}
```

```
Constructor: 1
Constructor: 2
```

These are the two cases where the constructor is directly called. A constructor like this is called a *converting constructor*. A converting constructor lets the compiler use that constructor to convert a parameter to the class type. For example, in line 2 above, what we should really assign is an object of type *standardClass*. Instead we are assigning an int. But the compiler is able to implicitly convert this int to a *standardClass* instance because there is a converting constructor. What 'explicit' does it take to make the constructor a non-converting one. This will restrict the compiler from implicitly converting the int to an object. Let's do that and see.

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

class standardClass
{
        int objID;
public:
        explicit standardClass(int ID)                  // now explicit
        {
                objID = ID;
                cout << "Constructor: " << objID << endl;
        }

};

int main(int argc, char** argv)
{
        standardClass stdClassObj1(1);                  // Line #1
        standardClass stdClassObj2 = 2;                  // Line #2
        return 0;
}
```



		Description
⊗	1	error C2440: 'initializing' : cannot convert from 'int' to 'standardClass'

The compiler now cannot convert the int to an object of *standardClass*. It is not allowed to implicitly call the constructor. So what do we need to do then? We need to call the constructor explicitly.

```
…
…
int main(int argc, char** argv)
{
        standardClass stdClassObj1(1);                              // Line #1
        standardClass stdClassObj2 = standardClass(2);    // Line #2
        return 0;
}
```

```
Constructor: 1
Constructor: 2
```

There really is no reason why we would want this constructor to be explicit in this example. But there are cases where you don't want the compiler calling the constructor implicitly and perhaps making a wrong argument conversion. Although you can make constructors that take multiple arguments to be explicit, there really is not much benefit in that. The only effect would be when using list type initialization of C++11.

This ends our topic on the fundamentals involved in C++ object construction. Apart from the four methods we discussed in this topic, C++11 standard adds a few more methods in there. The basics you learned in this topic should serve well in understanding the mechanism of those as well.

Topic 5

Pointers

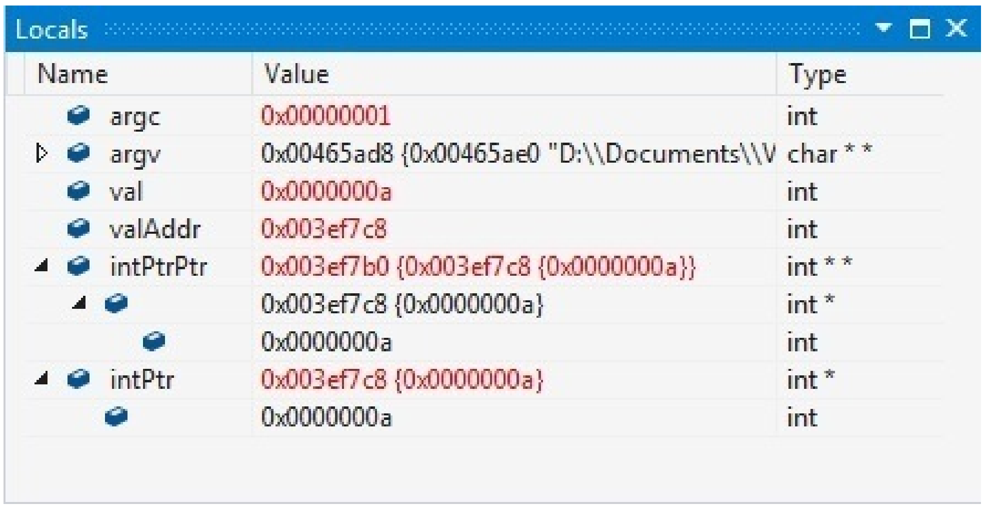
No discussion on C++ would be complete without talking about pointers. In this topic we will look into a few different aspects of pointer use.

As usual let’s start with a basic example to get things going.

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

int main(int argc, char** argv)
{
        int val = 10;
        int * intPtr = &val;
        int **intPtrPtr = &intPtr;

        cout << “val: “ << val << endl;
        cout << “*intPtr: “ << *intPtr << endl;
        cout << “**intPtrPtr: “ << **intPtrPtr << endl;

        return 0;
}
```

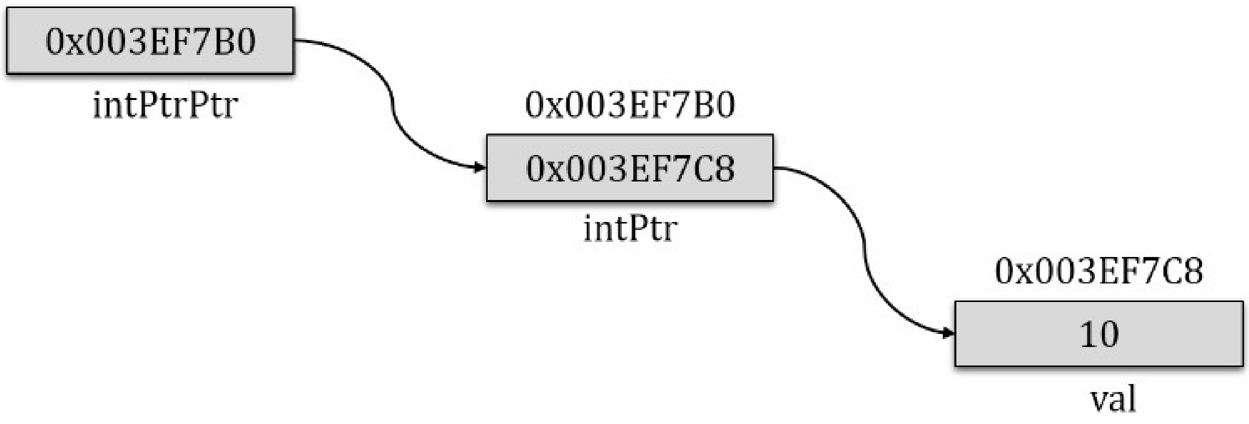
```
val: 10
*intPtr: 10
**intPtrPtr: 10
```

Here we have an integer, a point to the integer and a pointer to the pointer to the integer. You can see what is happening in the debug view:



Name	Value	Type
argc	0x00000001	int
argv	0x00465ad8 {0x00465ae0 "D:\\Documents\\V	char * *
val	0x0000000a	int
valAddr	0x003ef7c8	int
intPtrPtr	0x003ef7b0 {0x003ef7c8 {0x0000000a}}	int * *
	0x003ef7c8 {0x0000000a}	int *
	0x0000000a	int
intPtr	0x003ef7c8 {0x0000000a}	int *
	0x0000000a	int

Depicting it pictorially:



Passing pointers to functions

Now let’s try passing those pointers to functions.

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

void funcIntPtr(int * intPtr)
{
    cout << “Value: “ << *intPtr << endl;
}

void funcIntPtrPtr(int ** intPtrPtr)
{
    cout << “Value: “ << **intPtrPtr << endl;
}

int main(int argc, char** argv)
{
    int val = 10;
    int * intPtr = &val;
```

```
        int **intPtrPtr = &intPtr;

        funcIntPtr(intPtr);
        funcIntPtrPtr(intPtrPtr);

        funcIntPtr(&val);
        funcIntPtrPtr(&intPtr);

        return 0;
}
```

```
Value: 10
Value: 10
Value: 10
Value: 10
```

Pretty basic stuff. I’m just showing here how to call functions taking pointers and pointers to pointers. Now let’s see how things change when arrays come in to play.

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

void funcIntPtr(int * intPtr)
{
        cout << “Value: “ << *intPtr << endl;
}

void funcIntPtrPtr(int ** intPtrPtr)
{
        cout << “Value: “ << **intPtrPtr << endl;
}

void funcIntArr(int intArr[])
{
        cout << “Value: “ << intArr[0] << endl;
}

int main(int argc, char** argv)
{
        int val = 10;
```

```
    int * intPtr = &val;
    int **intPtrPtr = &intPtr;

    int intArr[] = { 1, 2, 3 };          // Line 1
    intPtr = intArr;                     // Line 2

    funcIntPtr(intArr);                  // Line 3
    funcIntArr(intArr);                  // Line 4
    funcIntPtr(intPtr);                  // Line 5
    funcIntArr(intPtr);                  // Line 6
    funcIntPtrPtr(intPtrPtr);            // Line 7
    return 0;
}
```

```
Value: 1
Value: 1
Value: 1
Value: 1
Value: 1
```

We defined a new integer array and also a function that takes an integer array parameter. Let’s go through the lines one by one to see what we are doing here.

- Line 1 defines an integer array with 3 elements and initializes it with values.
- In line 2 we assign the *intArr* to our previously defined pointer to int. Note the difference between making *intPtr* point to an integer and an array. First, *intPtr* could point to both an integer and also an integer array. Second, when we want *intPtr* to point to the integer *val*, we would write:

```
    *intPtr = val;
```

When it is pointing to an array we write:

```
    intPtr = intArr;
```

Note the asterisk. What is happening here, then?

intArr is actually a pointer to an integer. That is why we could assign it to *intPtr* as we did. Without the asterisk. So what is it pointing to? Keep reading.

- In line 3 we are passing the array to the function, which takes an integer pointer as a parameter. And you will see, it prints out the first element of the array.

What this means is that, the array is passed as a pointer. And when we dereferenced that pointer it was printing the first element of the array. That means the array name is a pointer to its first element.

In fact this is how C/C++ passes arrays to functions. They are not copied as in pass by value. When an array is passed to a function as an argument, it is passed as a pointer to its first element.

- Line 4 is passing the array to the function, which is expecting an integer array. Nothing exciting here.
- In line 5 we are passing the *intPtr* to the function, expecting a pointer to an integer, which is exactly what it is pointing to.
- Line 6 is a bit interesting. We are passing the pointer to integer to the function, which is expecting an integer array as the argument. But as the code compiles, it seems the compiler is happy to accomodate a pointer to an integer as an integer array argument.

  Why is this? As I said before, this is because the compiler treats an array argument as a pointer to its first element. So as far as the compiler is concerned, *funcIntPtr* and *funcIntArr* are both exactly the same function, having a point to integer parameter.
- In line 7 then, we are passing a pointer to pointer to an integer. The same as we did in the previous example.

Now that we got through that, did you wonder how the compiler was calling the subscript operator on a pointer in line 7? Let's find out.

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

int main(int argc, char** argv)
{
    int intArr[] = { 1, 2, 3 };
    int *intPtr = intArr;

    cout << *intPtr << endl;            // Line 1
    cout << intPtr[0] << endl;          // Line 2
    cout << intPtr[1] << endl;          // Line 3
    cout << *(intPtr + 1) << endl;      // Line 4
    cout << *(intPtr + 2) << endl;      // Line 5
    cout << 2[intPtr] << endl;          // Line 6

    return 0;
}
```

```
1
1
2
2
3
3
```

- Line 1 dereferences the pointer and this prints out 1. This means that *intPtr* is indeed pointing to the first element of *intArr*.
- Line 2 looks a bit weird. We are using the subscript operator on a pointer. But it works. This will again print out the first element of *intArr*.
- Line 3 accesses the second element of *intArr* and it correctly prints out 2. So this shows that you can indeed dereference a pointer with the subscript operator as you would with a normal array. But how is this possible?
- Line 4 is the answer to why lines 2 and 3 work. It is because all subscript operators on a pointer are converted to an expression, as in line 4. The pointer is advanced by the amount added to it and then dereferenced. But how does the compiler know exactly how much to advance? It knows it by the type of the pointer. Since *intPtr* is an integer pointer, compiler knows it should advance by 4 bytes (we found in topic 1 that an int is 4 bytes in this platform).
- Line 5 is the same as line 4 but now accessing the 3rd element of the array.
- Line 6 looks strange too. You'd probably never want to use this notation but it shows what we said about the compiler converting a subscript operator in to an addition and then dereferencing. So for the compiler line 6 just looks like:

```
cout << *(2 + intPtr) << endl;
```

In this example we assigned *intPtr* to *intArr*. I mentioned that *intPtr* is a pointer to its first element. In that sense we should be able to assign *intPtr* a pointer to the second element of the array like this:

```
int *intPtr = intArr[1];
```

Do that and you will get an error saying it cannot convert from 'int' to 'int *'. But why?

Because *intArr* by itself is a pointer to its first element. But *intArr[1]* is just an integer variable. If that is the case then we should be able to assign its address like we did for *val* in the previous example, right? Indeed we can.

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

int main(int argc, char** argv)
{
	int intArr[] = { 1, 2, 3 };
	int *intPtr = &(intArr[1]);

	cout << *intPtr << endl;
	cout << intPtr[0] << endl;
	cout << intPtr[1] << endl;
	cout << *(intPtr + 1) << endl;
	cout << *(intPtr + 2) << endl;
	cout << 2[intPtr] << endl;

	return 0;
}
```

```
2
2
3
3
-858993460
-858993460
```

You can easily decode the output. One thing to note is the final two outputs. You can guess why they are garbage values. Because *intPtr* is now pointing to the second element of *intArr* and lines 5 and 6 are trying to access the fourth element of the array, when there isn't one. So it is reading out of the array bounds and getting garbage values. Note that you can use the subscript operator or the pointer addition to a pointer that was pointing to a integer variable as well. That is, *intPtr* does not need to be pointing to an array. You could do the same operations when *intPtr* was pointing to *val* in the previous example. You'd just get garbage values.

Dealing with chars and its pointers can be a little confusing sometimes. In this next part we will look at the basic uses of char and it's pointers.

Char and its arrays

Let's change our original code to incorporate a char and a char array. Except for the function names, you can see that everything else is pretty much the same as in the integer example.

```
include<iostream>
using namespace std;

void funcCharPtr(char * charPtr)
{
        cout << “Value: “ << *charPtr << endl;
}

void funcCharPtrPtr(char ** charPtrPtr)
{
        cout << “Value: “ << **charPtrPtr << endl;
}

void funcCharArr(char charArr[])
{
        cout << “Value: “ << charArr[0] << endl;
}

int main(int argc, char** argv)
{
        char val = ‘a’;
        char * charPtr = &val;
        char **charPtrPtr = &charPtr;

        funcCharPtr(charPtr);                    // Line 1
        funcCharArr(charPtr);                    // Line 2
        funcCharPtrPtr(charPtrPtr);              // Line 3

        char charArr[] = { ‘x’, ‘y’, ‘z’ };       // Line 4
        charPtr = charArr;                       // Line 5

        funcCharPtr(charArr);                    // Line 6
        funcCharArr(charArr);                    // Line 7

        funcCharPtr(charPtr);                    // Line 8
        funcCharArr(charPtr);                    // Line 9

        funcCharPtrPtr(charPtrPtr);              // Line 10

        return 0;
}
```

```
Value: a
Value: a
Value: a
Value: x
Value: x
Value: x
Value: x
Value: x
```

- Here we start with a char val and assign it the single character ‘a’. Then we assign that char to a pointer to char and that to a pointer to pointer to char.
- Lines 1 and 2 pass the char pointer to the functions taking a pointer to char and a char array. They work the same way they did in the integer example. And line 3 as well. So you can see that a char is nothing different from an int in the way the compiler handles it.

  There is the notion of passing chars as integers but that is a different story. As far as the pointers are concerned, there is no relationship and they behave the same way. For example you cannot pass *charPtr* to a function with a pointer to integer parameter.
- Line 4 defines a char array with 3 elements and then in line 5 assigns it to the pointer to char. This, again, is exactly what we did with the *intArr* and *intPtr* in our previous example. So as we discussed then, a char array variable is a pointer. It is a pointer to its first element, as it was with *intArr*.
- The rest of the lines call the functions passing *charPtr* and *charArr*. Nothing is different here. The *charPtr* and *charArr* behave exactly the same way as *intPtr* and *intArr* did. As it did with the integer array, the compiler will convert the subscript operator to a pointer plus offset so you can use the array name and the pointer interchangeably.

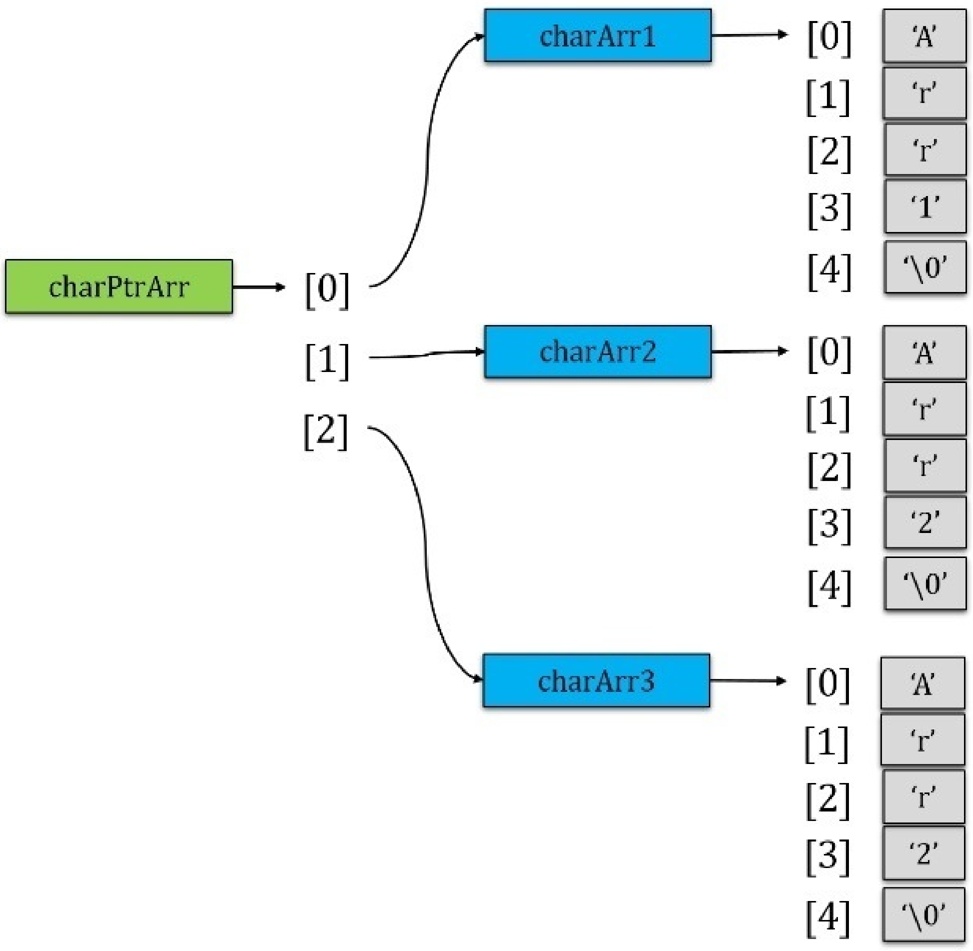
Now that we found that a char and a char array are nothing much different from an integer and an integer array in terms of pointer handling, let’s look at an array of char array.

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

void funcCharPtrArr(char *charPtrArr[])
{
        cout << “Value: “ << charPtrArr[0] << endl;
}
```

```
void funcCharPtr1(char * charPtr)
{
        cout << “Value: “ << charPtr << endl;
}

void funcCharPtr2(char * charPtr)
{
        cout << “Value: “ << *charPtr << endl;
        cout << “Value: “ << charPtr[3] << endl;
}

int main(int argc, char** argv)
{
        char charArr1[] = { ‘A’, ‘r’, ‘r’, ‘1’, ‘\0’ };          // Line 1
        char charArr2[] = { ‘A’, ‘r’, ‘r’, ‘2’, ‘\0’ };          // Line 2
        char charArr3[] = { ‘A’, ‘r’, ‘r’, ‘3’, ‘\0’ };          // Line 3

        char *charPtrArr[] = { charArr1, charArr2, charArr3 };

        funcCharPtrArr(charPtrArr);                          // Line 4
        funcCharPtr1(*charPtrArr);                           // Line 5
        funcCharPtr1(charArr1);                              // Line 6
        funcCharPtr2(*charPtrArr);                           // Line 7

        return 0;
}
```

```
Value: Arr1
Value: Arr1
Value: Arr1
Value: A
Value: 3
```

Before we discuss the results let’s illustrate what is happening here.



The figure above shows the relationship between the pointers and arrays. Note the arrows. These arrows show the binding of each element to the variable name. Remember that the array name is a pointer to its first element. That is what the arrows depict. Each array name is a pointer to its first element: *charPtrArr* is a pointer to *charArr1*, and *charArr1* is a pointer to char 'A', and so on.

Now it should be pretty easy to understand the results of the above code. Note that I included the terminating character at the end of each array. Including this has no special meaning to the code other than that the printout will be cleaner. If it there was no terminating character the output would contain garbage values when we print out the char arrays. Let's go through the code now.

- *charPtrArr* is an array of pointers to char. Remember that an array name is a pointer to it first element. So we can initialize *charPtrArr* with the three arrays we defined at the beginning because those are pointers to char (to their first char).
- *funcCharPtrArr* has an array of pointers to char parameters. This is the same type as *charPtrArr* so we should have no problem passing it. It prints out 'Arr1' and you know why.

  As we learned before, *charPtrArr[0]* for the compiler is **charPtrArr*. And this points to *charArr1*. So we are passing *charArr1* to the *cout* and it obliges by printing out the array contents. This is why we needed the terminating character for the cout to know where the sequence ends.
- *funcCharPtr1* takes a pointer to char parameter and prints it out. In line 5 we are

passing **charPtrArr*. You could easily understand what we are actually passing here. We are passing *charArr1* and it prints out *Arr1* as we'd expect. Note the similarity with line 4's printout and understand that we are passing the same thing to cout in both cases.

- Just to solidify your understanding I'm calling *funcCharPtr1* with *charArr1* to show that it is exactly the same as passing **charPtrArr*.
- *funcCharPtr2* takes the same type of parameters as *funcCharPtr1*, but the cout statement is a bit different. I want to show the relation of variable name and pointer again. Also note the passing argument is a bit different here but you could easily understand what we are passing. **(charPtrArr+2)* is just *charArr3*.

  Then in *funcCharPtr2* we are passing **charArr3* to the cout. So what are we passing here then? You know *charArr3* is a pointer to the first element 'A' so dereferencing it simply prints out 'A'. And finally for good measure we are also printing out the last character in the next statement. This also confirms that we indeed passed *charArr3*.

If we replaced the char arrays with integer arrays and with an array of pointers to integers and changed the types of the function parameters, eveything will work, with one exception. When we passed the char array to the cout, it printed out the whole array. But if we do that with the integer array, cout wouldn't print out all of the integer elements. Instead it would just print out its pointer value. That is the address of the first element. This is not because of anything special in a char pointer but rather how cout handles different types.

Let's look at one more observation. We discussed earlier that arrays are passed to the functions as pointers by the compiler. So function parameters **char* and *char[]* are identical to the compiler. Then we could do the following:

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

void funcCharPtrArr(char *charPtrArr[])
{
        cout << "Value: " << charPtrArr[0] << endl;
}

void funcCharPtrArr2(char **charPtrArr)
{
        cout << "Value: " << charPtrArr[0] << endl;
}

int main(int argc, char** argv)
{
        char charArr1[] = { 'A', 'r', 'r', '1', '\0' };
```

```
	char charArr2[] = { ‘A’, ‘r’, ‘r’, ‘2’, ‘\0’ };
	char charArr3[] = { ‘A’, ‘r’, ‘r’, ‘3’, ‘\0’ };

	char *charPtrArr[] = { charArr1, charArr2, charArr3 };

	funcCharPtrArr(charPtrArr);
	funcCharPtrArr2(charPtrArr);

	return 0;
}
```

```
Value: Arr1
Value: Arr1
```

funcCharPtrArr is the same as before and *funcCharPtrArr2* has a slightly different parameter type. It takes a pointer to pointer to char. But you by now know that both of these functions are the same. Because an array name is essentially a pointer, we can write **charPtrArr[]* as **(*charPtrArr)*. So an array of pointers is a pointer to a pointer.

Before we finish off this topic let’s look at one more example to demonstrate what a pointer means to a compiler. As I mentioned before, a pointer is just an address to a block of memory. There is no difference between a pointer to an object and a pointer to an int, or pointer to anything else. All of the pointers are of the same size and all of them have a memory address. The type of the pointer is what defines it. The type of the pointer tells the compiler what is in that block of memory it points to. Because you see, the pointer only points to the start of the memory block. The compiler has no idea how big that particular memory block is. It deduces that information from the type of the pointer. It’s a simple matter of *sizeof(typeOfPointer)* to determine the size of the memory block. That is why you can never dereference a *void** pointer because the compiler doesn’t know how to work with that memory block. This example should clarify this point.

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

class baseClass
{
public:
	int baseClassVar;

	baseClass(int baseVal) : baseClassVar(baseVal)
	{}
```

```
    void printVals()
    {
        cout << “baseClass::baseClassVar- “ << baseClassVar << endl;
    }
};


class derivedClass : public baseClass
{
public:
    int derivedClassVar;

    derivedClass(int baseVal, int derivedVal) : baseClass(baseVal), derivedClassVar(derivedVal)
    {}

    void printVals()
    {
        baseClass::printVals();
        cout << “derivedClass::derivedClassVar- “ << derivedClassVar << endl;
    }
};


int main(int argc, char** argv)
{
    baseClass* baseClassPtr = new baseClass(1);
    static_cast<derivedClass*>(baseClassPtr)->printVals();
    return 0;
}
```

```
baseClass::baseClassVar- 1
derivedClass::derivedClassVar- 7536751
```

What we did was to have a *baseClass* pointer with a *baseClass* instance. Then we static cast it to a *derivedClass*. Since *derivedClass* is of the same hierarchy the compiler is fine with that and it trusts us that it is indeed pointing to a *derivedClass* instance. Then we invoke the *derivedClass’s printVals* function. The *derivedClass printVal* function accesses *derivedClassVar*. The compiler assumes that the memory block pointed to by *baseClassPtr* contains a complete *derivedClass* instance of size *sizeof(derivedClass)*, but in fact it only contains a *baseClass* instance, which is smaller than a *derivedClass* one. So when the compiler accesses *derivedClassVar*, it is accessing a location that does not belong to it, and hence prints out a garbage value. For the compiler, it has no way to do any boundary checking. The boundary is set by the pointer type. We will learn more on this when we discuss class member offsets. But until then keep in mind that the compiler's only way of knowing an object type and size is through the pointer type.

So now you should understand the second argument of the main function that you've been writing all along. *char** argv* is basically *char *argv[]*. It is an array of pointers to char, exactly the same as *charPtrArr* in our example, and *argv*, as you know, contains a list of parameters passed to the *main* function.

Topic 6

Non-Constructible, Non-Copyable Class

In the topic "Object Construction" we discussed how objects are created, copied and assigned. There are three principal functions that every class should have: *constructor*, *copy constructor* and *assignment operator*. There are additional functions added in C++11 but we will limit our focus to these three primary functions. In this topic we will discuss how to make a class non constructible or copyable and investigate these functions in more detail.

Non-constructible class

What does it mean to construct, or instantiate, an object? It simply means calling one of the class constructors. So how do you then make a class non-constructible? You just make it so that the class constructor cannot be called. And there are two ways you can do that:

- Make the class abstract
- Make the constructors private

Abstract class

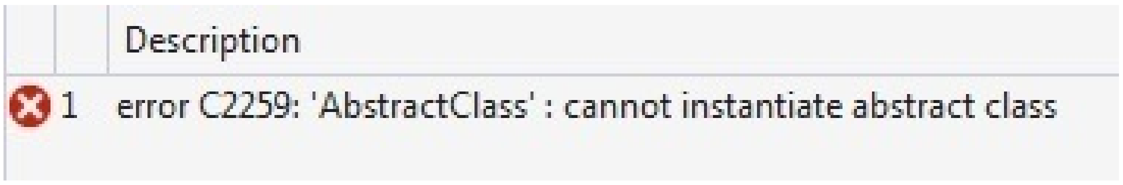
Unlike Java, C++ doesn't have an 'abstract' keyword to make a class abstract. The way to make a C++ class abstract is to have a pure virtual function. How do you make a virtual function pure? Assign the function a value of zero.

```
class AbstractClass {
public:
        AbstractClass();
        virtual void pureVirtFunc() = 0;  // <— this makes this class abstract.
}
```

So what does it mean for a class to be abstract?

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

class AbstractClass {
public:
        AbstractClass();
        virtual void pureVirtFunc() = 0; // <— this makes this class abstract.
};
```

```
int main(int argc, char** argv)
{
	AbstractClass absClass;
	return 0;
};
```

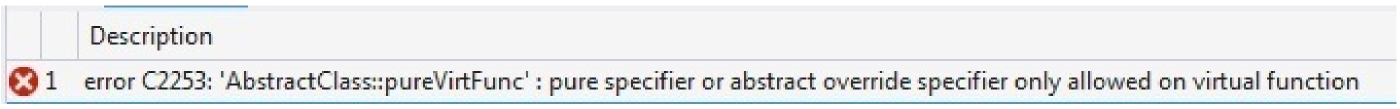


```
Description
1	error C2259: 'AbstractClass' : cannot instantiate abstract class
```

It means you cannot instantiate an object by calling the constructor. And you know that only way you can do it is by deriving this class and implementing the pure virtual function.

By the way, what does it mean for a non-virtual function to be abstract? It makes no sense. The point of making a function 'pure' is for a deriving class to define it. Then you must have the function as virtual. But what happens if you try to make a non-virtual function pure?

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

class AbstractClass {
public:
	AbstractClass();
	void pureVirtFunc() = 0; // <— (^_^)
};

int main(int argc, char** argv)
{
	cout << "Can we make it pure???" << endl;
	return 0;
};
```



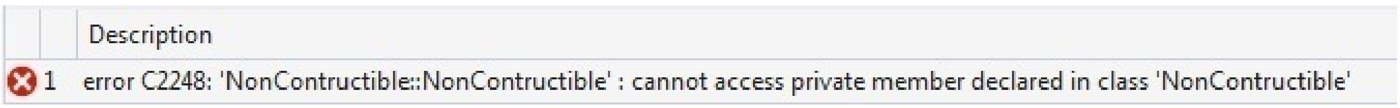
```
Description
1	error C2253: 'AbstractClass::pureVirtFunc' : pure specifier or abstract override specifier only allowed on virtual function
```

The compiler apparently is smart.

So what does making a class abstract do? It makes the class non-instantiable by itself. But any class that derives (and defines the pure virtual function) can. Then, the real motivation in making a class abstract is to make that class a base class. To make sure whoever wants to use that class derives it and implements the pure virtual function. We will look into this concept in a different topic, but what you need to understand here is the difference in non-constructability of an abstract. Abstract class objects are non-constructible because they are not supposed to be constructed as they are. Not because they shouldn't be. There is a difference as we'll see next.

An abstract class can still be constructed through derivation. The constructors of an abstract class are public or protected. The motive of making a class abstract is to make it a base class. Base classes must be able to be constructed. But what if you want your class to be completely non-constructible? You remove the only way it can be constructed. You know a constructor is the only way a class can be instantiated. You make it so that the constructor cannot be called from outside the class. How do you do it? Make the constructor private.

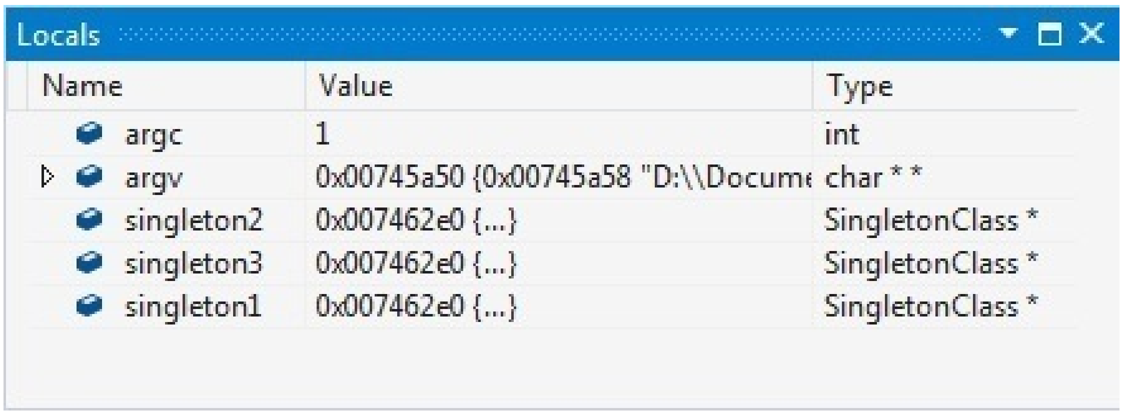
```
#include<iostream>
using namespace std;

class NonContructible {
private:
        NonContructible() { cout << "Haha. You can't call this." << endl; }
};

int main(int argc, char** argv)
{
        NonContructible nonCons;

        return 0;
};
```



```
Description
1   error C2248: 'NonContructible::NonContructible' : cannot access private member declared in class 'NonContructible'
```

The compiler is not happy because it cannot call the private method. So what do you do with a class like this? Like this, you can't do much. There is no way you can instantiate this class as it is. One use of making a constructor private is in the *Singleton* pattern, where you only want one instance of the class in existence. So you will have something like this:

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

class SingletonClass {
private:
        static SingletonClass * theSingleton;
        SingletonClass() { cout << "This will be called only once." << endl; }
public:
        static SingletonClass * getSingleton()
        {
                if (!theSingleton)
                        SingletonClass::theSingleton = new SingletonClass;

                return theSingleton;
        }

};
SingletonClass* SingletonClass::theSingleton = NULL;



int main(int argc, char** argv)
{
        SingletonClass *singleton1 = SingletonClass::getSingleton();
        SingletonClass *singleton2 = SingletonClass::getSingleton();
        SingletonClass *singleton3 = SingletonClass::getSingleton();
        return 0;
}
```



```
Locals
Name          Value                                     Type
  argc        1                                         int
▷ argv        0x00745a50 {0x00745a58 "D:\\Docume        char * *
  singleton2  0x007462e0 {...}                          SingletonClass *
  singleton3  0x007462e0 {...}                          SingletonClass *
  singleton1  0x007462e0 {...}                          SingletonClass *
```

This will be called only once.

You will see in this case the *SingletonClass* constructor was called only once and from the locals window you see that all three *SingletonClass* pointers point to the same object. This is one use of making the constructor private to restrict the object construction, but still provide a utility function to receive an object.

So far what we looked at was making a class non-constructible. Next we will look at making it non-copyable.

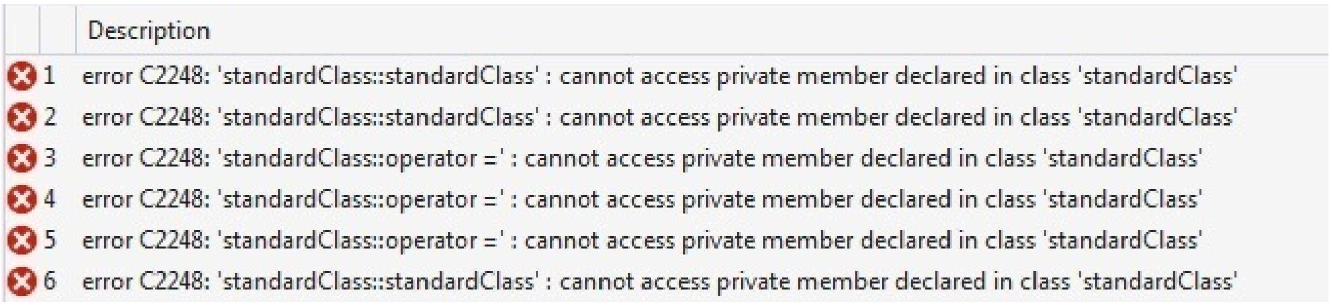
Non-copyable class

Let's refresh copy constructors from topic 4.

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

class standardClass
{
        int objID;
public:
        standardClass(int ID) // constructor
        {
                objID = ID;
        }

        ~standardClass()        // destructor
        {           }

        standardClass(const standardClass &objToCopy)     // copy constructor
        {
                objID = objToCopy.objID;
        }

        standardClass & operator=(const standardClass &objToCopy) // assignment operator
        {
                objID = objToCopy.objID;
                return *this;
        }
};

int main(int argc, char** argv)
{
        standardClass stdClassObj1(1);
        standardClass stdClassObj2(stdClassObj1);           // Line #1
        standardClass stdClassObj3 = stdClassObj2;          // Line #2
        stdClassObj1 = stdClassObj2;                          // Line #3
        stdClassObj2 = standardClass(4);                     // Line #4
        stdClassObj3 = 5;                                       // Line #5
        return 0;
}
```

Lines 1 and 2 call the copy constructor and lines 3-5 call the assignment operator (constructor is also called in lines 4 and 5). Making a class non-copyable means to not let the user do operations like in the above code. So it means that you restrict making one object from another object. Stop copying. And now you know how we could do that. We just make the functions that do the copying private. We simply make the copy constructor and the copy assignment operator private. Let’s do that.

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

class nonCopyable
{
        int objID;
public:
        nonCopyable(int ID) // constructor
        {
                objID = ID;
        }

        ~nonCopyable()        // destructor
        {}

private:
        nonCopyable(const nonCopyable &objToCopy)       // copy constructor
        {
                objID = objToCopy.objID;
        }

        nonCopyable & operator=(const nonCopyable &objToCopy) // assignment operator
        {
                objID = objToCopy.objID;
                return *this;
        }
};

int main(int argc, char** argv)
{
        nonCopyable ncClassObj1(1);
        nonCopyable ncClassObj2(ncClassObj1);      // Line #1
        nonCopyable ncClassObj3 = ncClassObj2;     // Line #2
     ncClassObj1 = ncClassObj2;                    // Line #3
     ncClassObj2 = nonCopyable(4);                 // Line #4
     ncClassObj3 = 5;                              // Line #5
        return 0;
}
```



```
Description
1  error C2248: 'standardClass::standardClass' : cannot access private member declared in class 'standardClass'
2  error C2248: 'standardClass::standardClass' : cannot access private member declared in class 'standardClass'
3  error C2248: 'standardClass::operator =' : cannot access private member declared in class 'standardClass'
4  error C2248: 'standardClass::operator =' : cannot access private member declared in class 'standardClass'
5  error C2248: 'standardClass::operator =' : cannot access private member declared in class 'standardClass'
6  error C2248: 'standardClass::standardClass' : cannot access private member declared in class 'standardClass'
```

The compiler throws more than a few errors and you can easily understand what they mean. It cannot access the copy constructor and the assignment operator to do the copying.

Note in the above code that we have made the constructor public. Of course it needs to be because we need to instantiate an object and there is no utility function to otherwise. Now what if we want to make a class non-constructible and non-copyable? Is it sufficient to make only the constructor private? After all, how can you copy if you can't construct, right? Let's look at an obvious example.

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

class nonCopyable
{
	int objID;

	class nonCopyable(int ID) // constructor
	{
		objID = ID;
	}

public:
	nonCopyable(const nonCopyable &objToCopy) // copy constructor
	{
		objID = objToCopy.objID;
	}

	nonCopyable & operator=(const nonCopyable &objToCopy) // assignment operator
	{
		objID = objToCopy.objID;
		return *this;
	}
```

```
        ~nonCopyable() // destructor
        {           }

        static nonCopyable * getObject()
        {
                return new nonCopyable(1);
        }
};

int main(int argc, char** argv)
{
        nonCopyable *ncClassObjPtr = nonCopyable::getObject();
        nonCopyable ncClassObj2(*ncClassObjPtr);
        nonCopyable ncClassObj3 = ncClassObj2;
        delete ncClassObjPtr;

        return 0;
}
```

In the above code we have made the constructor private and provided a utility function to receive a new object (similar to the Singleton pattern but we are not restricting to one object). Then we have made the copy constructor and the assignment operator public. This code compiles and works, for obvious reasons. I just wanted to show you that if you want to make a class non-constructible and non-copyable, it is not sufficient to make only the constructor private. But what if we don't define the copy constructor and assignment operator at all? Let's remove those two functions and see.

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

class nonCopyable
{
        int objID;
        class nonCopyable(int ID) // constructor
        {
                objID = ID;
        }
public:
        static nonCopyable * getObject()
        {
                return new nonCopyable(1);
        }
};

int main(int argc, char** argv)
{
        nonCopyable *ncClassObjPtr = nonCopyable::getObject();
        nonCopyable ncClassObj2(*ncClassObjPtr);
        nonCopyable ncClassObj3 = ncClassObj2;
        delete ncClassObjPtr;
        return 0;
}
```

This code compiles happily and it will copy *ncClassObjPtr* with no issues. But why? Because, since we did not define the copy constructor and the assignment operator, the compiler obliged by making them for us. There are a few rules governing whether the compiler will generate default copy constructors and assignment operators. But what you need to keep in mind is that these compiler generated ones are *public*. In an almost empty class like this the compiler will opt to do bit-wise copying and not define explicit functions. We'll look into that in a later topic. But what needs to be kept in mind is that if you don't define them, the compiler will do that for you. So in this case the compiler is happy to do the copying for us. Therefore, if you don't want your class to be copied, define those functions privately.

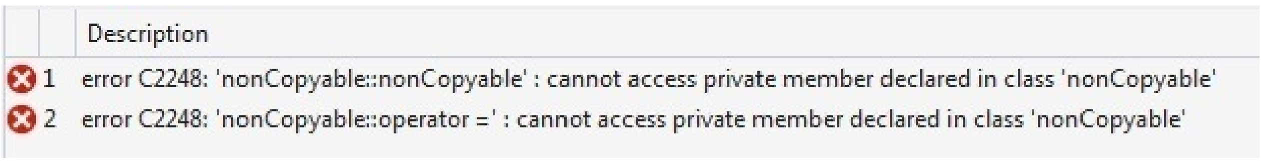
I should mention that there is a notion of not defining the copy constructor and the assignment operator, in addition to making them private. Like this:

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

class nonCopyable
{
        int objID;

public:

        nonCopyable(int ID) // constructor
        {
                objID = ID;
        }

        ~nonCopyable()            // destructor
        {}

private:
        nonCopyable(const nonCopyable &objToCopy);                       // Not defined
        nonCopyable & operator=(const nonCopyable &objToCopy);     // Not defined
};
```

The idea here is that if there is a friend class, then even that class would not be able to copy, because otherwise friends are allowed to call private functions. When these functions are not defined, the code will compile fine but there will be a linker error as the linker cannot find the implementation for these functions. But you might have a need to have an implementation for the copy constructor and the assignment operator, depending on your application. Maybe for an internal function use. Like we had an implementation for the constructor in the Singleton class. Whatever way you choose, keep in mind that if you want these functions to be private, define them private. Otherwise the compiler will define them public.

Before we leave non-copyable classes, what would be the fate of a derived class of a non-copyable class?

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

class nonCopyable
{
        int objID;
public:
        nonCopyable(int ID) // constructor
        {
                objID = ID;
        }

        ~nonCopyable()              // destructor
        {}

private:
        nonCopyable(const nonCopyable &objToCopy) // copy constructor
        {
                objID = objToCopy.objID;
        }

        nonCopyable & operator=(const nonCopyable &objToCopy) // assignment operator
        {
                objID = objToCopy.objID;
                return *this;
        }
};

class derivedNonCopyable : public nonCopyable
{
public:
        derivedNonCopyable(int ID) : nonCopyable(ID)
        {}
};

int main(int argc, char** argv)
{
        derivedNonCopyable dncClassObj1(2);
        derivedNonCopyable dncClassObj2(dncClassObj1);    // Line #1
        derivedNonCopyable dncClassObj3 = dncClassObj2;  // Line #2
        dncClassObj1 = dncClassObj2;                      // Line #3
        dncClassObj2 = derivedNonCopyable(4);            // Line #4
        dncClassObj3 = 5;                                 // Line #5
        return 0;
}
```



```
Description
1  error C2248: 'nonCopyable::nonCopyable' : cannot access private member declared in class 'nonCopyable'
2  error C2248: 'nonCopyable::operator =' : cannot access private member declared in class 'nonCopyable'
```

The compiler isn't happy. It is complaining of *nonCopyable* class's copy constructor and assignment operator being private. As you've seen many times, lines 1 and 2 above make calls to the copy constructor while lines 3-5 call the assignment operator. If you commented out lines 1-5, you will see that there are no compiler errors. *dncClassObj1* is constructible with no problems as the constructors are public. But if you then uncomment only line 1, the compiler will throw an error saying the copy constructor of *nonCopyable* is private. And if you uncomment any of lines 3-5 it will complain about the assignment operator being private. What is happening is that as long the program is not calling for copy constructor or assignment operator, the compiler will not bother with them. But when there is a statement that calls for the copy constructor or the assignment operator, if the class hasn't defined them, the compiler will generate them for you, which will contain a call to the base class copy constructor and the assignment operator. But it cannot do that if the *nonCopyable* has them private, so you get the errors.

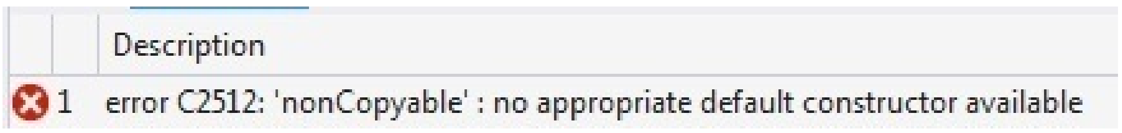
Let me make a small detour. Let's say all the *nonCopyable* functions were public (it's not non-copyable anymore). How would you implement the copy-constructor for the derived class?

This is how you'd write. Note that *nonCopyable* is copyable now.

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

class nonCopyable
{
        int objID;

public:
        nonCopyable(int ID) // constructor
        {
                objID = ID;
        }

        ~nonCopyable()              // destructor
        {             }

public:
        nonCopyable(const nonCopyable &objToCopy) // copy constructor
        {
                objID = objToCopy.objID;
        }

        nonCopyable & operator=(const nonCopyable &objToCopy) // assignment operator
        {
                objID = objToCopy.objID;
                return *this;
        }
};

class derivedNonCopyable : public nonCopyable
{
public:
        derivedNonCopyable(int ID) : nonCopyable(ID)
        {}

        derivedNonCopyable(const derivedNonCopyable &objToCopy) : nonCopyable(objToCopy)
        {}
};

int main(int argc, char** argv)
{
	derivedNonCopyable dncClassObj1(2);
	derivedNonCopyable dncClassObj2(dncClassObj1);		// Line #1
	derivedNonCopyable dncClassObj3 = dncClassObj2;		// Line #2
	return 0;
}
```

We invoke the *nonCopyable* copy constructor in the initializer list. But did you know you must call it in the initializer list? What if not?

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

class nonCopyable
{
	int objID;
public:
	nonCopyable(int ID) // constructor
	{
		objID = ID;
	}

	~nonCopyable() // destructor
	{		}

public:
	nonCopyable(const nonCopyable &objToCopy) // copy constructor
	{
		objID = objToCopy.objID;
	}
};

class derivedNonCopyable : public nonCopyable
{
public:
	derivedNonCopyable(int ID) : nonCopyable(ID)
	{}

	derivedNonCopyable(const derivedNonCopyable &objToCopy)
	{
		nonCopyable::nonCopyable(objToCopy);
	}
};

int main(int argc, char** argv)
{
        derivedNonCopyable dncClassObj1(2);
        derivedNonCopyable dncClassObj2(dncClassObj1);        // Line #1
        derivedNonCopyable dncClassObj3 = dncClassObj2;       // Line #2
        return 0;
}
```



```
   Description
1  error C2512: 'nonCopyable' : no appropriate default constructor available
```

Why are you getting errors about no default constructor? Why can't we call the base class copy constructor in the method body?

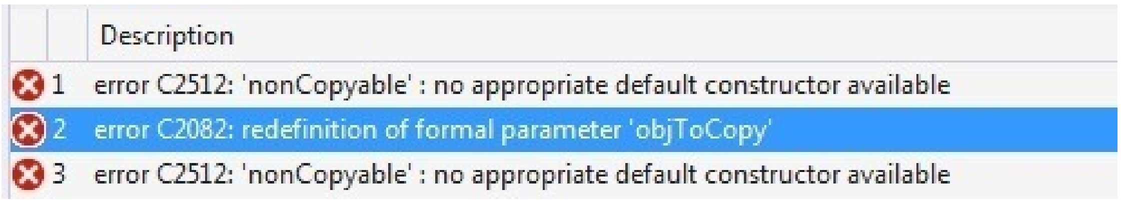
Because, remember that when we are invoking the copy constructor, unlike when the assignment operator is invoked, there is no object in existence. The copy constructor is going to instantiate a new object. So when the derived class copy constructor gets invoked to create a new derived class object, it must first have a base class object. Because at this point there is no base class object in existence. This is similar to calling a derived class constructor. Before the derived class constructor does its work, it first calls the base class constructor to construct its part. The copy constructor does the same thing. It needs the base class to construct itself first. So what does the derived class copy constructor do? It calls the base class constructor. That is why we are getting this error. Because *nonCopyable* does not have a constructor that takes no arguments (if it did, we wouldn't have this error). So how does making the copy constructor call in initializer list change this? Same reason as we'd initialize a const variable or a reference in the initializer list of the constructor. We initialize it before the body of the method. Because the compiler needs to initialize them before it reaches the method body. Same case with the copy constructor. It tries to initialize the base class object by calling the default constructor if there is no call for the base class constructor or the copy constructor to initialize the base object. So instead of calling the copy-constructor in the initializer list, you could very well call the constructor. (Not in our case though as the variable is private we can't pass it to the constructor.)

Notice that I called the *nonCopyable* copy constructor with class qualifier? Had we called like this:

```
…
class derivedNonCopyable : public nonCopyable
{
public:
        derivedNonCopyable(int ID) : nonCopyable(ID)
```

```
    {}

    derivedNonCopyable(const derivedNonCopyable &objToCopy)
    {
        nonCopyable(objToCopy); // Line 1
    }
};
…
```

you'd get a new additional error:



		Description
✖	1	error C2512: 'nonCopyable' : no appropriate default constructor available
✖	2	error C2082: redefinition of formal parameter 'objToCopy'
✖	3	error C2512: 'nonCopyable' : no appropriate default constructor available

What does it mean "*redefinition of formal parameter 'objToCopy'*?"

Because you see, since you are never allowed to call constructors directly, what line 1 above does is define a new variable called *objToCopy* of type *nonCopyable*. It is like defining an integer like *int(5)*. Since there is already a variable by that name, the compiler is complaining about redefinition. That is why you need to qualify this call with the classname to let compiler know exactly that you are calling the copy constructor.

Apologies for that rather long detour. I thought it's an interesting bit to mention.

Now that we know how to make a class non-constructible and non-copyable, let's see how we can control 'where' they are instantiated.

Stack and heap allocation

There are two entities where an object can reside in C++: the 'stack' and the 'heap'. Stack is where you have things such as procedure records and local variables, while heap is where you do the dynamic allocations. Here's a simple example.

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

class someClass
{
    int objID;

public:
    someClass(int ID)
    {
        objID = ID;
```

```
                cout << “Constructor for ID: “ << objID << endl;
        }

        ~someClass()
        {
                cout << “Destructor for ID: “ << objID << endl;
        }

};

int main(int argc, char** argv)
{
        someClass someClassObj(1);                                 //Line 1
        someClass *someClassPtr = new someClass(2);                //Line 2
        delete someClassPtr;                                       //Line 3
        return 0;
}
```

```
Constructor for ID: 1
Constructor for ID: 2
Destructor for ID: 2
Destructor for ID: 1
```

- Line 1 creates the *someClassObj* in the stack. This is the first constructor printout in the output.
- Line 2 creates a *someClass* object in the heap and returns its address to *someClassPtr*. The second constructor printout corresponds to this instantiation.
- Line 3 deletes *someClassPtr* by explicitly calling the destructor. We must deallocate each heap memory we allocated. This is the third printout in the output.
- Then finally we see the *someClassObj* destructor being called.

A couple of things to note here:

- All heap allocated objects must be deallocated explicitly. The compiler will not do this for us.
- Stack allocated objects are automatically deallocated when they go out of scope. *someClassObj* was created in the scope of *main* and when *main* returns, *someClassObj* goes out of scope, so its destructor is called automatically.

This is one of the important things to note about C++. Unlike Java/C# where they have automatic garbage collection, in C++, you need to do your own housekeeping whenever you use the heap. Stack is a different story. Compiler will take care to clean it up. So all of the auto variables will be taken care of by the compiler when they go out of scope. Let’s make a small addition to our code.

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

class someClass
{
	int objID;
public:
	someClass(int ID)
	{
		objID = ID;
		cout << “Constructor for ID: “ << objID << endl;
	}

	~someClass()
	{
		cout << “Destructor for ID: “ << objID << endl;
	}

};

int main(int argc, char** argv)
{
	someClass someClassObj(1);				// Line1
	{
		someClass someClassObj2(3);		// Line 2
	}
	someClass *someClassPtr = new someClass(2); // Line 3
	delete someClassPtr;					// Line 4

	return 0;
}
```

```
Constructor for ID: 1
Constructor for ID: 3
Destructor for ID: 3
Constructor for ID: 2
Destructor for ID: 2
Destructor for ID: 1
```

Note that we added a scoped object *someClassObj2*. Because it is within a pair of curly braces, the scope of *someClassObj2* is confined to within those braces. As soon as the execution leaves right curly brace *someClassObj2* goes out of scope, and the compiler calls its destructor. That is why you see the constructor and the destructor being called in succession.

Things will become much more convincing if you take a look at the disassembly for the above code.

```
int main(int argc, char** argv)
{
    009D8BA0 push ebp
        009D8BA1 mov ebp, esp
        …
        …
        someClass someClassObj(1);
    009D8BDD push 1
        009D8BDF lea ecx, [someClassObj]
        009D8BE2 call someClass::someClass(09D14E7h) <—(1)
        009D8BE7 mov dword ptr[ebp - 4], 0

    {
        someClass someClassObj(3);
        009D8BEE push 3
            009D8BF0 lea ecx, [ebp - 20h]
            009D8BF3 call someClass::someClass(09D14E7h) <—(2)
    }
    009D8BF8 lea ecx, [ebp - 20h]
        009D8BFB call someClass::~someClass(09D150Fh)   <—(3)

        someClass *someClassPtr = new someClass(2);
    009D8C00 push 4
        009D8C02 call operator new (09D13DEh)            <—(4)
        009D8C07 add esp, 4
        009D8C0A mov dword ptr[ebp - 11Ch], eax
        …
        …
        009D8C25 call someClass::someClass(09D14E7h)   <—(5)
        …
        …
        009D8C4C mov ecx, dword ptr[ebp - 128h]
        009D8C52 mov dword ptr[someClassPtr], ecx

        delete someClassPtr;
    009D8C55 mov eax, dword ptr[someClassPtr]
        …
        …
        009D8C75 mov ecx, dword ptr[ebp - 110h]
        009D8C7B call someClass::`scalar deleting destructor' (09D1500h) <—(6)
        …
        …

        return 0;

    009D8C92 mov dword ptr[ebp - 0F8h], 0
        009D8C9C mov dword ptr[ebp - 4], 0FFFFFFFFh
        009D8CA3 lea ecx, [someClassObj]
        009D8CA6 call someClass::~someClass(09D150Fh) <—(7)
        009D8CAB mov eax, dword ptr[ebp - 0F8h]
}
```

The code statements are clearly printed out in the disassembly so you can easily see what the disassembly is for that particular statement. Note that I have removed a lot of assembly statements in between.

- (1) is the call to the constructor for instantiating the first object (Line 1).
- (2) is the constructor call to the explicitly scoped object we instantiate within the curly braces.
- Right after the right curly brace we see (3) calling the destructor for *someClassObj2*. So you see that this call is right after the object goes out of scope.
- (4) calls the operator new to allocate memory in the heap. And after some operations it calls the constructor in (5).
- (6) is our call to delete the heap object.
- Then finally after the return statement you can see the compiler calling the destructor for *someClassObj* in (7).

Looking at the disassembly makes it pretty clear what is actually going on behind the curtains. The destructor calls at (3) and (7) are automatically added by the compiler. So you can see how stack objects are automatically deleted when they go out of scope.

So why are we discussing this? Because this is what we are going to use to put restrictions for stack and heap allocations.

Restricting stack allocation

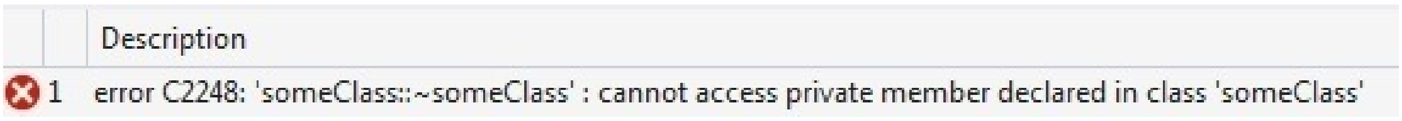
Let's first look at restricting allocation on stack. How would you program to restrict your class being instantiated on the stack? For example you want to restrict statements like line 1 in the above program.

What happens when we create an object on stack by calling a statement like line 1? As we also saw in the disassembly, two things happen:

- The ompiler first puts a call to the constructor of the class
- Then after the object goes out of scope, it calls the destructor

Although the call to the destructor is something we don't explicitly do, the compiler adds the call for us. We saw this in point (7) of the disassembly. Then, how would we restrict stack allocation of a class? We can certainly make the constructor private, but that makes the class non-constructible altogether. What we want is to make it non-constructible on the stack. How about we make the destructor private?

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

class someClass
{
        int objID;
public:
        someClass(int ID)
        {
                objID = ID;
                cout << "Constructor for ID: " << objID << endl;
        }

private:
        ~someClass()
        {
                cout << "Destructor for ID: " << objID << endl;
        }
};

int main(int argc, char** argv)
{
        someClass someClassObj(1);                          // Line 1
        someClass *someClassPtr = new someClass(2);   // Line 2
        return 0;
}
```



```
    Description
 1  error C2248: 'someClass::~someClass' : cannot access private member declared in class 'someClass'
```

It works. The compiler throws an error for line 1 where we are going to instantiate *someClassObj* on the stack. But notice that the compiler has no problem with line 2 where the object will be on the heap. So you see this is one way you can restrict stack allocation of a class. But did you notice something was missing in the above code? Yes, we are not deleting *someClassPtr*. It is because we cannot. If you put a delete statement you will get exactly the same compiler error as we got here. It is because the *delete* statement itself is a call to the destructor. So how do we handle this then? We restrict stack allocation, forcing the objects to be allocated only on the heap, but we cannot delete them. Well, in this case we need to provide an utility function to delete the object. The same way we did for private constructors in the Singleton class.

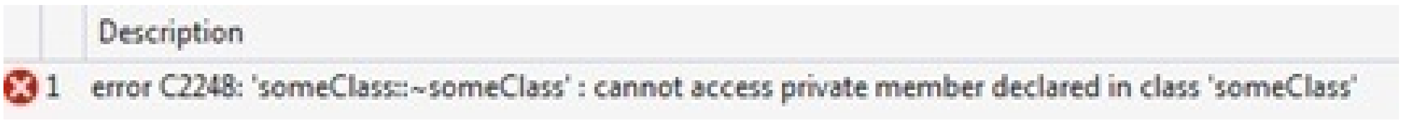
There are two cases of this private destructor that makes me curious.

- What if we derive this class?
- What if we have this class as a member object?

Let's find out.

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

class someClass
{
	int objID;
public:
	someClass(int ID)
	{
		objID = ID;
		cout << “Constructor for ID: “ << objID << endl;
	}

private:
	~someClass()
	{
		cout << “Destructor for ID: “ << objID << endl;
	}

};

class derivedSomeClass : public someClass
```

```
{
public:
        derivedSomeClass(int ID) : someClass(ID)
        {}

        ~derivedSomeClass() {}
};

int main(int argc, char** argv)
{
        derivedSomeClass derivedSomeClassObj(1);
        return 0;
}
```



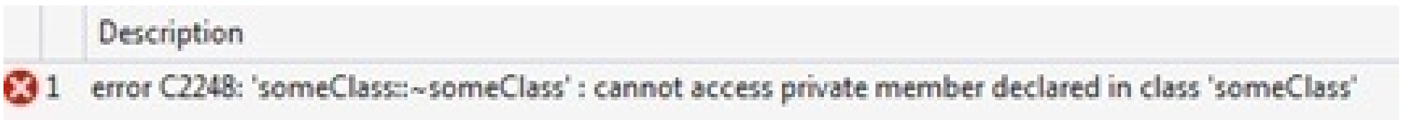
```
Description
1   error C2248: 'someClass::~someClass' : cannot access private member declared in class 'someClass'
```

You get the same compiler error as before. *derivedSomeClass* destructor is trying to call the *someClass* destructor and it is private. So making the destructor private makes even the derived classes unable to be on the stack. So what if you want the derived classes to be able to be on the stack? You make the destructor in *someClass* protected. Try it and you will see the code above works fine. But you will not be able to instantiate the *someClass* object on the stack, only the derived classes.

Then what about when *someClass* is a member object?

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

class someClass
{
        int objID;
public:
        someClass(int ID)
        {
                objID = ID;
                cout << "Constructor for ID: " << objID << endl;
        }

private:
        ~someClass()
        {
                cout << "Destructor for ID: " << objID << endl;
```

```
	}

};

class someOtherClass
{
	someClass someClassObj;
public:
	someOtherClass(int ID) : someClassObj(ID)
	{}

	~someOtherClass() {}
};

int main(int argc, char** argv)
{
	someOtherClass someOtherClassObj(1);
	return 0;
}
```



```
Description
1	error C2248: 'someClass::~someClass' : cannot access private member declared in class 'someClass'
```

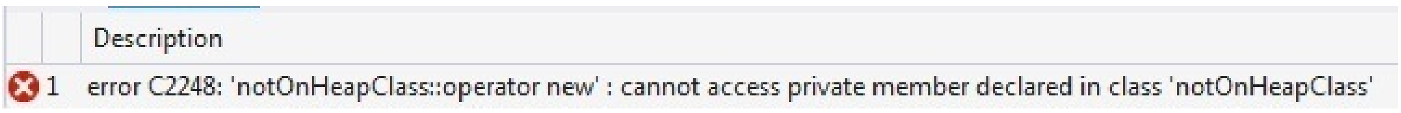
You get the same error. The compiler still needs to call the destructor for the member objects and it cannot do so with the private destructor. And in this case making the destructor protected wouldn't help either.

So you see, that by making the destructor private, you can completely restrict the stack allocation of a class. But also keep in mind that this makes you not able to call *delete* on heap allocated objects. You need some other utility function to take care of that.

Restricting heap allocation

Now let's look at how we restrict heap allocation. Do you remember the disassembly we saw a while back? The heap allocation had a call to *the operator new* in the disassembly. In C++, the way to allocate objects in the heap is to use operator *new*. So then, how do we control heap allocation? We do something similar to what we did for stack allocation. We make *operator new* private.

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

class notOnHeapClass
{
    int objID;
public:
    notOnHeapClass(int ID)
    {
        objID = ID;
        cout << “Constructor for ID: “ << objID << endl;
    }

    ~notOnHeapClass()
    {
        cout << “Destructor for ID: “ << objID << endl;
    }

private:
    void *operator new(size_t);
};

int main(int argc, char** argv)
{
    notOnHeapClass notOnHeapObj(1);                             // Line 1
    notOnHeapClass *notOnHeapPtr = new notOnHeapClass(2);       // Line 2
    delete notOnHeapPtr;
    return 0;
}
```

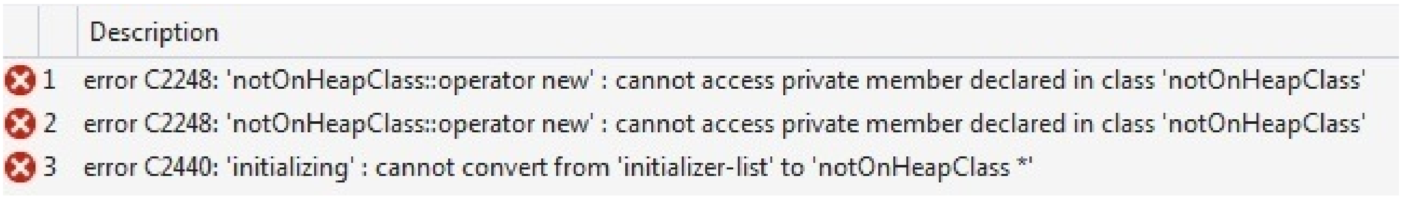


```
Description
1   error C2248: 'notOnHeapClass::operator new' : cannot access private member declared in class 'notOnHeapClass'
```

As expected, the compiler throws an error saying that it cannot access the operator new. That’s it. That is how you restrict heap allocation. We will look in to operator new in more detail when we talk about *placement new* in another topic.

Ideally, when we restrict heap allocation by making the *operator new* private, we usually make all of the following four operators private too.

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

class notOnHeapClass
{
    int objID;

public:
    notOnHeapClass()
    {
        objID = 0;
    }
    notOnHeapClass(int ID)
    {
        objID = ID;
        cout << “Constructor for ID: “ << objID << endl;
    }

    ~notOnHeapClass()
    {
        cout << “Destructor for ID: “ << objID << endl;
    }

private:
    void *operator new(size_t);
    void *operator new[](size_t);
    void operator delete(void*);
    void operator delete[](void*);
};

int main(int argc, char** argv)
{
    notOnHeapClass notOnHeapObj(1);
    notOnHeapClass *notOnHeapPtr = new notOnHeapClass(2);    // Line 1
    notOnHeapClass *notOnHeapArr = new notOnHeapClass[5];    // Line 2
    delete notOnHeapPtr;                                      // Line 3
    delete[] notOnHeapArr;                                    // Line 4
    return 0;
}
```



```
    Description
1   error C2248: 'notOnHeapClass::operator new' : cannot access private member declared in class 'notOnHeapClass'
2   error C2248: 'notOnHeapClass::operator new' : cannot access private member declared in class 'notOnHeapClass'
3   error C2440: 'initializing' : cannot convert from 'initializer-list' to 'notOnHeapClass *'
```

Without making *new[], delete and delete[]* private, lines 2, 3 and 4, respectively, would compile without any errors. Although you would not *delete* without being able to *new*, it is always better to make them all private, if that is your intention. Note that operator new is different from *operator new[]*. So if you need to restrict heap allocation for arrays of objects, make sure you make array *operator new* private too.

Before we finish this topic, keep in mind that making operator new private will not help you restrict heap allocation in a case like this:

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

class notOnHeapClass
{
        int objID;
public:
        notOnHeapClass()
        {
                objID = 0;
        }
        notOnHeapClass(int ID)
        {
                objID = ID;
                cout << "Constructor for ID: " << objID << endl;
        }
        ~notOnHeapClass()
        {
                cout << "Destructor for ID: " << objID << endl;
        }

private:
        void *operator new(size_t){}
        void *operator new[](size_t){}
        void operator delete(void*){}
        void operator delete[](void*){}
};

int main(int argc, char** argv)
{
        std::vector<notOnHeapClass> notOnHeapClassVec;
      notOnHeapClassVec.push_back(notOnHeapClass(1));
        return 0;
}
```

This program works fine. You'd think this should work because we are not allocating anything on the heap as *notOnHeapClassVec* is on the stack. Actually, this is not the case.

STLs like *Vector* allocate the vector itself, that is, the vector related header data on the stack, but the actual vector elements, that is *notOnHeapClass* objects, are actually always allocated on the heap. So when we pushback a *notOnHeapClass* object to the vector, that object is actually instantiated on the heap. But then why is there no error when the operator new is private? Because the compiler does not call operator new. See the disassembly of this code shown below:

```
int main(int argc, char** argv)
{
    00B38280 push ebp
    00B38281 mov ebp,esp
    …
    …

    std::vector<notOnHeapClass> notOnHeapClassVec;
    00B382BD lea ecx,[notOnHeapClassVec]
    00B382C0 call std::vector<notOnHeapClass,std::allocator<notOnHeapClass>
    >::vector<notOnHeapClass,std::allocator<notOnHeapClass> > (0B3134Dh)
    00B382C5 mov dword ptr [ebp-4],0

    notOnHeapClassVec.push_back(notOnHeapClass(1));
    00B382CC push 1
    00B382CE lea ecx,[ebp-0F8h]
    00B382D4 call notOnHeapClass::notOnHeapClass (0B31005h) <—(1)
    00B382D9 mov dword ptr [ebp-100h],eax
    …
    00B382F6 lea ecx,[notOnHeapClassVec]
    00B382F9 call std::vector<notOnHeapClass,std::allocator<notOnHeapClass> >::push_back (0B31334h)
    00B382FE mov byte ptr [ebp-4],0
    00B38302 lea ecx,[ebp-0F8h]
    00B38308 call notOnHeapClass::~notOnHeapClass (0B3150Ah) <—(2)

    return 0;
    00B3830D mov dword ptr [ebp-0ECh],0
    00B38317 mov dword ptr [ebp-4],0FFFFFFFFh
    00B3831E lea ecx,[notOnHeapClassVec]
    00B38321 call std::vector<notOnHeapClass,std::allocator<notOnHeapClass>
    >::~vector<notOnHeapClass,std::allocator<notOnHeapClass> > (0B31064h)
    00B38326 mov eax,dword ptr [ebp-0ECh]
}
```

You see the vector never calls the *operator new* even though the object is on the heap.

Why? Because it doesn’t need to. *Operator new* is one way the compiler helps us allocate memory and call the constructor for the object. We can certainly do this in the C-style by manually calling ‘malloc’ and then invoking the constructor. Operator new is sort of a convenient way to do both memory allocation and constructor invocation in one call. So the *Vector* does not need to do it. It has its own way of allocating heap memory. But you can be sure that it calls the constructor when we call *pushback*, as evident from point 1. And then it calls the destructor (point 2) when the vector goes out of scope.

So this is one case where making the operators private will not restrict the heap allocation. We can restrict heap allocation this way only when *operator new* is called to do the allocation

I hope this section gave you a solid understanding on the basics of stack and heap allocation and also how to manipulate object construction and copying. Everything is handled through a few basic fundamental functions and we can manipulate those functions to achieve the implementation we need.

Topic 7

Understanding new

new is C++‘s version of C’s *malloc* and the variants. But new actually does more than *malloc* does. There is very little reason to manually allocate memory using *malloc* (or its variants) in C++ because *new* will do all that for you. We’ve used *new* in many occasions in other topics but let’s take some time here to formally get to know it.

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

class simpleClass
{
	int objID;
public:
	simpleClass(int ID) // constructor
	{
		objID = ID;
		cout << “Constructing object with ID: “ << objID << endl;
	}
};

int main(int argc, char** argv)
{
	simpleClass *simpleClassPtr = new simpleClass(1);
	return 0;
}
```

```
Constructing object with ID: 1
```

To get to know a little bit more of what is happening behind the scenes, let’s look at the disassembly (I have removed most parts).

```
int main(int argc, char** argv)
{
…
…
    simpleClass *simpleClassPtr = new simpleClass(2015);
0010591D push 4
0010591F call operator new (0101514h)              <—(1)
…
…
0010593D push 7DFh                                 <—(2)
00105942 mov ecx,dword ptr [ebp-0ECh]
00105948 call simpleClass::simpleClass (010162Ch)  <—(3)
…
…
delete simpleClassPtr;
…
…
0010598B call operator delete (01011B3h)           <—(4)
00105990 add esp,4
return 0;
}
```

There are a few interesting points here and I believe they provide very good insight into what is happening at the compiler level.

- When we call *new*, notice that the compiler actually calls the *operator new* at (1). As we will investigate later, *operator new* is what allocates memory. So the first task of *new* is to allocate memory using *operator new*.
- Notice how I passed 2015 as the object initializer? There is nothing special about 2015. I just wanted to pass something more unique than a number like ‘1’ and show how this number is used in the disassembly. In (2) you see 2015 is being pushed to a register to be used in the object construction.
- In (3) we see the constructor for the object is being called. This is the second task of *new*. First it allocated memory by calling *operator new*, and now it calls the constructor.
- (4) calls the *operator delete* on the object.

So what we see is that *new* does two main operations:

- Calls *operator new* to allocate memory
- Calls the class constructor

Let’s then take a closer look at operator new.

Operator new and placement new

There are two main variants of *operator new*:

```
    void* operator new (std::size_t size);
    void* operator new (std::size_t size, void* ptr);
```

And these two operator versions do slightly different things.

The first syntax:

```
        void* operator new (std::size_t size);
```

Takes as an argument the size in bytes, and allocates that amount of storage and returns a *void* pointer to the first byte of that memory allocation. This is generally called the *operator new*.

The second syntax:

```
        void* operator new (std::size_t size, void* ptr);
```

This does something different. It simply returns the passed *ptr* argument. It might not make much sense but it will become clear as you see its purpose. This syntax is called *placement new*. It is used when we need to construct the desired object at the location specified by *ptr*.

Let’s do an example.

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

class simpleClass
{
        int objID;

public:
        simpleClass(int ID)          // constructor
        {
                objID = ID;
                cout << “Constructing object with ID: “ << objID << endl;
        }

        ~simpleClass()
        {
                cout << “Destructing object with ID: “ << objID << endl;
        }
};

int main(int argc, char** argv)
```

```
{
    void *ptrToMem = operator new(sizeof(simpleClass));               // Line 1
    simpleClass *simpleClassPtr = new (ptrToMem)simpleClass(26);       // Line 2
    simpleClassPtr->~simpleClass();                                    // Line 3
    operator delete(ptrToMem);                                         // Line 4
    //delete simpleClassPtr;                                           // Line 5
    return 0;
}
```

```
Constructing object with ID: 26
Destructing object with ID: 26
```

- Line 1 calls the *operator new* (syntax 1), or the allocation function, with the size. It simply allocates a chunk of memory and returns a pointer to that memory location.
- Line 2 calls the *placement new* (syntax 2), and passes it the pointer to the allocated memory, and also passes the argument for the constructor. This line places a *simpleClass* object at *ptrToMem* and calls the constructor on that.
- What actually happens in line 2 is that *new* will call *placement new* (because of the calling syntax). Then *placement new* will simply return back *ptr*; this is the allocated memory where we want the object constructed. After that, *new* calls the class constructor, the same second step when we used *new*.
- Line 3 calls the destructor explicitly. We will look at this more in the topic on destructors but destructors can be called like this. But if we are going to call the destructor like this, then we need to call *operator delete* explicitly as in like 4, too. Call to *operator delete* will deallocate the memory.
- You do not need to call destructor and *operator delete*. You can use *delete* as in line 5 and it will call both the destructor and the *operator delete* for you. It does the reverse of *new*. But notice that I have commented out line 5. Because if you are going to deallocate the memory using *operator delete* (line 4) you must not do *delete* again. Deallocating an already deallocated memory is an undefined operation in C++.

Placement new

This is probably a good place to explain *placement new* a bit more as its functionality is a little peculiar. It simply returns the *void* pointer passed to in the arguments*. The compiler decides whether it should call *operator new* or *placement new* by looking at the syntax. The syntaxes are clearly different for the two. If we are simply calling *new* with no additional parameters, the compiler will:

- Call *operator new* and receive the pointer to the allocated memory.

- Call class constructor to construct the object at the memory it received.

If we are calling *new* with additional parameters, the compiler will:

- Call *placement new* and receive the pointer to the allocated memory.
- Call class constructor to construct the object at the memory it received.

So you see, the same functionality happens whether *operator new* is called or *placement new* is called, as far as *new* is concerned. The only difference is which operator (*new* or *placement*) it calls to get the memory block. In the case of *operator new* it allocates memory dynamically and passes the pointer. And with *placement new*, the purpose is NOT to allocate memory dynamically but instead use the memory block address passed to it as an argument. So *placement new* simply needs to pass back that memory address to *new*, because that's what *new* needs from it. In both cases *new* calls the class constructor.

To show that there is nothing fancy about *operator new*, here is the same code, replaced with your trusty *malloc*:

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

class simpleClass
{
        int objID;
public:
        simpleClass(int ID) // constructor
        {
                objID = ID;
                cout << "Constructing object with ID: " << objID << endl;
        }
};

int main(int argc, char** argv)
{
        void *ptrToMem = malloc(sizeof(simpleClass));
        simpleClass *simpleClassPtr = new (ptrToMem)simpleClass(26);
        simpleClassPtr->~simpleClass();
        free(ptrToMem);
        return 0;
}
```

So you see that *operator new* isn't doing anything special. It simply allocates memory as *malloc* does. Also note that we are calling *free* to deallocate the memory.

So far you've seen what *new* does. It calls *operator new* to allocate memory and then calls the constructor to create an object on the allocated memory. As you saw above, we can do what *new* does manually by calling *operator new* and then *placement new*.

If *new* does both of these for us, why would we want to do them manually by calling them explicitly? In most cases, we don't. We need not call *operator new* and *placement new*. But there are certain situations where you would want to do it. And C++ provides the flexibility to do it.

- *Restrict heap allocation*: We discussed this in the topic on object construction. By defining *operator new* in the class and making it private, we can restrict the class being allocated on the heap.
- *Memory pools*: There are situations where we don't want to do dynamic memory allocation. It takes time and if there isn't enough memory it throws exceptions. So we can allocate memory beforehand and keep it in a pool. And then when we want to create new objects, we call *placement new* on the already allocated memory.

OK, so we've looked at how *operator new* and *placement new* work. These are global scope operators. They have defined behavior. *Operator new* allocates memory and *placement new* constructs objects on already allocated memory. What if we want to change this behavior? We definitely can and all we need to do is define our own *operator new* in our class. A custom class specific operator. Let's see a simple example.

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

class simpleClass
{
        int objID;
public:
        simpleClass(int ID) // constructor
        {
                objID = ID;
                cout << “Constructing object with ID: “ << objID << endl;
        }

        void* operator new(std::size_t size)
        {
                void * ptr = malloc(size);
                cout << “Custom operator new. Allocating “ << size << ” bytes at “ << ptr << endl;
                return ptr;
        }

        void operator delete(void* ptr)
        {
                cout << “Custom operator delete” << endl;
                std::free(ptr);
        }

        void* operator new (std::size_t count, void* ptr)
        {
                cout << “Custom placement new.” << endl;
        }
};

int main(int argc, char** argv)
{
        simpleClass *simpleClassPtr = new simpleClass(10);
        cout << “simpleClassPtr at “ << simpleClassPtr << endl;
        delete simpleClassPtr;
        return 0;
}
```

```
Custom operator new. Allocating 4 bytes at 003A95B8
Constructing object with ID: 10
simpleClassPtr at 003A95B8
Custom operator delete
```

What did we do here?

- We defined a custom class specific operator new by overriding it. In here we simply allocate the required number of bytes using *malloc* and return the pointer returned from *malloc*. We already saw that we can use *malloc* as *operator new* replacement.
- We also override *operator delete*. Since we are allocating memory with *malloc* we must take care to deallocate it with *free*.
- In the result output, we see *new* is calling our custom operator new, which allocates 4 bytes and returns the pointer to the memory.
- Then we see the constructor is being called.
- As confirmation, we see that address of the memory allocated from our custom operator new is the same as where the object was constructed.
- Finally we call *delete*, which calls our overloaded version.
- Also note how we defined a class specific *placement new*. It does nothing but a printout and not even returning a *void**. But see that it was not called by *new*. So *new* doesn’t use placement new to construct the object.

So you see how we can easily customize the memory allocation routine by having class specific *operator new*. What if you simply want to log, or do some other kind of housekeeping in your custom *operator new* and not mess with the global *operator new*. Just change your custom function as follows:

```
void* operator new(std::size_t size)
{
        void * ptr = ::operator new(size); // calling global operator new
        cout << “Custom operator new. Allocating “ << size << ” bytes at “ << ptr << endl;
        return ptr;
}
```

Note that we called *::operator new*, with global-scope specifier. This is important. Otherwise our operator new would be calling itself recursively. And also in this case the *operate delete* overload should call *::operate delete*.

One more thing; how are we calling our class specific *operator new* without an instantiation? We are calling it as if it were static function but we didn’t define it as *static*. It’s because these *allocation functions* are indeed static. They are actually special operators and are handled in a special way (we’ll learn more on this in the operator overloading topic).

So let’s recap what we have done so far:

- We found out what the *new* operator does. It calls *operator new* and then calls the constructor to create the object.
- We saw how *operator new* is simply allocating memory and returning a *void**. And that we could do the same with *malloc*.
- We used *placement new* to construct a new object on already allocated memory.
- We defined custom class specific *operator new* and saw how calling *new* calls our custom operator new to allocate memory, and then calls the constructor as it normally would.

We’ve covered a lot of ground on *new*. Let’s finish off this topic by looking at how we can further customize the *new* behavior.

Let’s assume you want to do some housekeeping tasks before you construct new objects. You don’t want to put these into the class constructor. You can simply overload *placement new* with additional parameters. As an example:

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

class simpleClass
{
        int objID;
public:
        simpleClass(int ID) // constructor
        {
                objID = ID;
                cout << “Constructing object with ID: “ << objID << endl;
        }

        void* operator new(std::size_t size)
        {
                void * ptr = ::operator new(size); // calling global operator new
                cout << “Custom operator new.” << endl;
                return ptr;
        }

        void* operator new(std::size_t size, bool memoryFull, int objCount)
        {
                cout << “Custom placement new: Obj. count: “ << objCount << endl;

                if (memoryFull)
                {
                        cout << “Memory full!” << endl;
                        // call function to delete objects
                }
                //LogObjectCount(objCount);   // do logging
                return ::operator new(size);
        }

        void* operator new (std::size_t count, void* ptr)
        {
                cout << “Custom placement new.” << endl;
        }

        void operator delete(void* ptr)
        {
                cout << “Custom placement delete.” << endl;
      ::operator delete(ptr);
        }
};


int main(int argc, char** argv)
{
        simpleClass *simpleClassPtr1 = new simpleClass(10);                    // Line 1
```

```
    simpleClass *simpleClassPtr2 = new (true, 1) simpleClass(20);    // Line 2
    delete simpleClassPtr1;
    delete simpleClassPtr2;
    return 0;
}
```

```
Custom operator new.
Constructing object with ID: 10
Custom placement new: Obj. count: 1
Memory full!
Constructing object with ID: 20
Custom placement delete.
Custom placement delete.
```

Not a very meaningful example but it can clarify some points.

- Line 1 calls *new* with no additional parameters. This calls the overloaded *operator new* in the class. Then the class constructor is also called implicitly by *new*.
- Line 2 passes a *bool* and an *int* to *new*. This matches with the custom overloaded *placement new* we defined in the class. As there is a matching function, *new* first calls this function to obtain a pointer to the memory. You can see the *bool* and the *int* is properly passed to the *placement new* function.
- After calling the custom *placement new* and getting the memory, *new* then calls the class constructor.
- Note how our regular placement new was never called. This shows again that regular placement new will only be called if we pass a *void** to allocated memory.
- Finally we see the custom *operator delete* is called.

Last thing to note is that *placement new* does not always needs to be passed a pointer. *Placement new* is any overloaded version of *operator new.*

I hope this topic provided you with a solid understanding of what goes on behind *new* operator and how you can overload the *operator new* allocation functions to get the flexibility you need.

Topic 8

Understanding Constructors

Constructors. We all know what they are so let’s get right to it.

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

class baseClass
{
public:
        int objID;

        baseClass()
        {
                cout << “(1) Default constructor” << objID << endl;
        }

        baseClass(int ID) // constructor
        {
                objID = ID;
                cout << “(2) Constructing base object with ID: “ << objID << endl;
        }
};

class derivedClass : public baseClass
{
public:
        derivedClass()
        {
                cout << “(3) Constructing derived object with default ID: “ << objID << endl;
        }

        derivedClass(int ID) : baseClass(ID)
        {
                cout << “(4) Constructing derived object with ID: “ << objID << endl;
        }

        derivedClass(float ID)
        {
            cout << "(5)C onstructing derived object with ID: " << objID << endl;
        }
};


int main(int argc, char** argv)
{
        derivedClass dcObj1();                          // Line 1
        cout << "— Line 1 —" << endl;
        derivedClass dcObj2;                            // Line 2
        cout << "— Line 2 —" << endl;
        derivedClass dcObj3(1);                         // Line 3
        cout << "— Line 3 —" << endl;
        derivedClass dcObj4(2.0f);                      // Line 4
        return 0;
}
```

```
— Line 1 —
(1) Default constructor-858993460
(3) Constructing derived object with default ID: -858993460
— Line 2 —
(2) Constructing base object with ID: 1
(4) Constructing derived object with ID: 1
— Line 3 —
(1) Default constructor-858993460
(5)C onstructing derived object with ID: -858993460
```

Let's go through the steps quickly:

- The first output is "— Line 1 —". This means line 1 of the code has not called the constructor. Why is that? Well it doesn't, because line 1 is not defining an object of type *derivedClass*. For the compiler, line 1 is a function prototype. A function named *dcObj1*, that returns a *derivedClass* object. So it does nothing but declare a function prototype. So make sure you omit the parentheses!
- Line 2 instantiates an object by calling the default constructor. See, no parentheses. This is how you call the default constructor. Remember that the default constructor is the one with no arguments. There's one more point to note here. The variable *objID* is not initialized. We will discuss this in more detail later, but just keep in mind that the compiler does not automatically initialize your class variables.

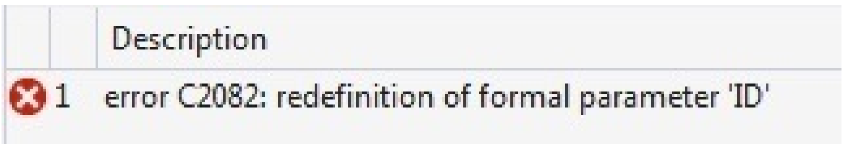
- *dcObj2* also shows how the *baseClass* constructor is called first and then the derived class constructor. This is the most basic thing to know about constructors and inheritance. Base class constructors are always called before the derived class one.
- Also see that the *derivedClass* default constructor is not calling the base class constructor explicitly. The compiler does this for you. It implicitly adds the call to the base class default constructor before the function body.
- Line 3 instantiates an object with an int argument. And this constructor is explicitly calling the *baseClass* constructor with the argument. You can see the *objID* is now correctly initialized by the *baseClass* constructor.
- Finally in line 4 we call with the float argument. The thing to note here is that the default constructor of the *baseClass* is explicitly called.
- The take aways here are: base class constructor is always called before the derived class one, and if the derived class doesn't explicitly call a base class constructor in the initializer list, the compiler will always implicitly call the default constructor. We will see why it is essential that we call the base class constructor in the initializer list.

Calling base class contrsuctor

Let's look at another example.

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

class baseClass
{
public:
    int objID;

    baseClass()
    {
        cout << “(1) Default constructor” << objID << endl;
    }

    baseClass(int ID)
    {
        objID = ID;
        cout << “(2) Constructing base object with ID: “ << objID << endl;
    }
};

class derivedClass : public baseClass
```

```
{
public:
        derivedClass(int ID)
        {
                baseClass(ID);
                cout << “(4) Constructing derived object with ID: “ << objID << endl;
        }
};


int main(int argc, char** argv)
{
        derivedClass dcObj(1);
        return 0;
}
```



```
     Description
1    error C2082: redefinition of formal parameter 'ID'
```

Note how I changed the call to the *baseClass* constructor in the *derivedClass* constructor? Before we had it in the initializer list. Here I removed it from there and put it in the function body. So why this error?

Because what is happening when we call *baseClass(ID)* inside the function is, the compiler interprets it as a declaration of a *baseClass* type called *ID*. So with the argument named *ID* in the parameter list, the compiler is complaining that we are redefining the parameter *ID*. This happens because you are not allowed to call constructors directly, that is, unless you do it in the initializer list.

Now just for fun, what do you think would happen if we had the derived class like this in the example above?

```
…
class derivedClass : public baseClass
{
public:
        derivedClass(int ID)
        {
                baseClass(1); // calling with an int
                cout << “(4) Constructing derived object with ID: “ << objID << endl;
        }
};
…
```

```
(1) Default constructor-858993460
(2) Constructing base object with ID: 1
(4) Constructing derived object with ID: -858993460
```

So instead of passing parameter *ID*, we pass an int explicitly. You will see that it works. This will call *baseClass* constructor. But why? Why did we get the error when we pass *ID*, but not when we pass in an int? After all, *ID* is an integer.

The same thing as before happens. First, the compiler will try to interpret the call as a declaration. In the previous example, *baseClass(ID)* is interpreted as “*baseClass ID*”. So the compiler in that case attempted to make a *baseClass* type *ID* and found the redefinition. But when we pass the integer it is interpreted as ”*baseClass 1*”. And this clearly cannot be a declaration, so the compiler constructs an unnamed *baseClass* object, which will be destructed immediately.

Can you explain the output?

- Since there is no call to the base class constructor in the initializer list of derived class constructor, the compiler adds a call to the base class default constructor. Since no argument is passed *objID* is not initialized; that is why the garbage value.
- The second output line corresponds to “*baseClass(1)*”, which, as we discussed, creates a temporary *baseClass* object with value 1.
- The third output is the final cout statement in the *derivedClass* constructor.

OK, so we now know that we cannot call a base class constructor in the method body of a derived class constructor. But how about calling a base class constructor in one of its own constructors. Let’s do one.

In the simple example below we are calling the default constructor from the constructor that takes an int. This isn’t doing anything meaningful but I just want to show you something.

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

class baseClass
{
public:
    int objID;

    baseClass()
    {
        cout << “(1) Default constructor” << objID << endl;
        }

        baseClass(int ID)
        {
                baseClass();
                objID = ID;
                cout << “(2) Constructing base object with ID: “ << objID << endl;
        }
};

int main(int argc, char** argv)
{
        baseClass bcObj(1);
        return 0;
}
```

```
(1) Default constructor-858993460
(2) Constructing base object with ID: 1
```

Things look fine here. We are calling the constructor with the int argument, which first calls the default constructor. The default constructor prints out *objID*, which is garbage, but it is expected as we haven’t set the value yet; then the constructor sets the value of *objID* and prints it out, which correctly prints out the expected value 1. Everything seems OK so far. But let’s change the order of calling the default constructor. Let us first set the *objID* value and call the default constructor, which I think makes more sense.

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

class baseClass
{
public:
        int objID;

        baseClass()
        {
                cout << “(1) Default constructor” << objID << endl;
        }

        baseClass(int ID)
        {
```

```
        objID = ID;
        baseClass();
        cout << “(2) Constructing base object with ID: “ << objID << endl;
    }
};

int main(int argc, char** argv)
{
    baseClass bcObj(1);
    return 0;
}
```

```
(1) Default constructor-858993460
(2) Constructing base object with ID: 1
```

What is going on here? Why isn’t the default constructor seeing the value we set for *objID*? Is it because we need to initialize *objID* in the initializer list, and not in the method body? Nope. Although we should be initializing it in the initializer list. Let’s define the destructor and things will become a lot clearer.

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

class baseClass
{
public:
    int objID;

    baseClass()
    {
        cout << “(1) Default constructor” << objID << endl;
    }

    baseClass(int ID) // constructor
    {
        objID = ID;
        baseClass();
        cout << “(2) Constructing base object with ID: “ << objID << endl;
    }
```

```
        ~baseClass()
        {
                cout << “(3) Default destructor” << objID << endl;
        }
};


int main(int argc, char** argv)
{
        baseClass bcObj(1);
        return 0;
}
```

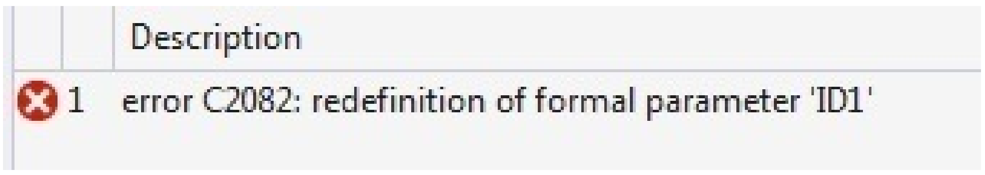
```
(1) Default constructor-858993460
(3) Default desstructor-858993460
(2) Constructing base object with ID: 1
(3) Default destructor1
```

Do you see what is happening here?

First, the default constructor is called and immediately after that, the destructor is called. What?

What is happening here is that the call “*baseClass()*” is constructing an unnamed *baseClass* object and then the compiler destroys it immediately. And then the compiler goes on to execute the rest of the statements in the constructor. So we are not calling the default constructor of the *this* object. We are calling it on another object, an unnamed temporary one, that gets destroyed in the very next line, without serving any purpose. So is this any different from what we saw in the case of the derived class constructor calling the base class constructor? No. It is the same thing. You simple cannot call a constructor in the method body. Why then, did we get an error in the previous case and not in this one? Because we were calling the default constructor, no arguments. The following example should make it clear.

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

class baseClass
{
public:
        int objID;
        baseClass()
        {
                cout << "(1) Default constructor" << objID << endl;
        }

        baseClass(int ID) // constructor
        {
                objID = ID;
                cout << "(2) Constructing base object with ID: " << objID << endl;
        }

        baseClass(int ID1, int ID2) // constructor
        {
                baseClass(ID1);
                cout << "(3) Constructing base object with ID: " << objID << endl;
        }
};

int main(int argc, char** argv)
{
        baseClass bcObj(12);
        return 0;
}
```



```
Description
1    error C2082: redefinition of formal parameter 'ID1'
```

I'm sure you don't need any further explanation.

So the bottom line is, you cannot call a constructor directly, ever. The only way you can do it is in the initializer list. Like this.

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

class baseClass
{
public:
        int objID;

        baseClass()
        {
            cout << "(1) Default constructor" << objID << endl;
        }

        baseClass(int ID) : baseClass()
        {
            objID = ID;
            cout << "(2) Constructing base object with ID: " << objID << endl;
        }

        baseClass(int ID1, int ID2) : baseClass(ID1)
        {
            cout << "(3) Constructing base object with ID: " << objID << endl;
        }
};


int main(int argc, char** argv)
{
        baseClass bcObj(1, 2);
        return 0;
}
```

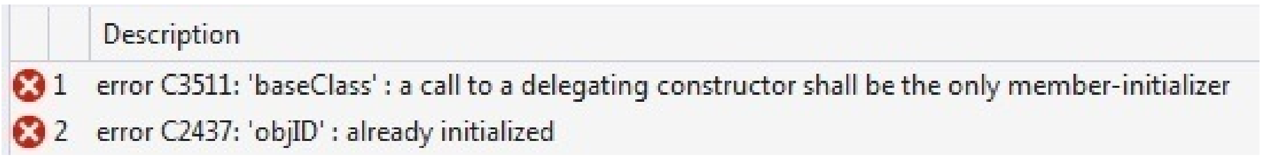
```
(1) Default constructor-858993460
(2) Constructing base object with ID: 1
(3) Constructing base object with ID: 1
```

Constructor delegation

The type of constructors that call other constructors (in the initializer list, of course), are called *delegating constructors*, for obvious reasons. But there is a limitation in delegating constructors. You cannot do any other initialization in the initializer list. That is, you cannot do something like this:

```
    baseClass(int ID) : baseClass(), objID(ID)
        {
            cout << "(2) Constructing base object with ID: " << objID << endl;
        }
```

The compiler will dutifully let you know this fact.



```
    Description
1   error C3511: 'baseClass' : a call to a delegating constructor shall be the only member-initializer
2   error C2437: 'objID' : already initialized
```

Delegating constructors can be very efficient in eliminating code repeat in similar constructors, but with a small restriction, of course.

Member initialization

In the examples above we saw how the member variable *objID* has to be initialized in the constructor explicitly and the compiler will not do that for us. So how exactly are member variables initialized by the compiler? Let’s start with a very simple one.

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

class baseClass
{
public:
        int objID;

};

int main(int argc, char** argv)
{
        baseClass bcObj;
        cout << “bcObj.objID: “ << bcObj.objID << endl;
        return 0;
}
```

```
bcObj.objID: 4114384
```

Here we are not defining any constructor. We let the compiler generate one for us. But in reality even the compiler wouldn’t provide one, actually. Because we only have one integer member variable and the compiler doesn’t need to do anything about it. So here we see that the integer is indeed left alone. No initialization is done. Make sure you initialize variables before using; uninitialized variables have undefined behavior.

Let’s add a bit more context.

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

```
class baseClass
{
public:
        int baseClassObjID;
};

class anotherClass
{
public:
        int anotherClassObjID;
        baseClass baseClassObj;
};

int main(int argc, char** argv)
{
        anotherClass acObj;
        cout << "anotherClassObjID: " << acObj.anotherClassObjID << endl;
        cout << "baseClassObjID: " << acObj.baseClassObj.baseClassObjID << endl;
        return 0;
}
```

```
anotherClassObjID: -858993460
baseClassObjID: -858993460
```

Here too, as expected, the integer member variables are not being initialized. Now what if the *baseClass* has a default constructor defined?

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

class baseClass
{
public:
        int baseClassObjID;

        baseClass()
        {
                baseClassObjID = 1;
                cout << "baseClass constructor" << endl;
        }
};

class anotherClass
{
public:
        int anotherClassObjID;
        baseClass baseClassObj;
};

int main(int argc, char** argv)
{
        anotherClass acObj;
        cout << “anotherClassObjID: “ << acObj.anotherClassObjID << endl;
        cout << “baseClassObjID: “ << acObj.baseClassObj.baseClassObjID << endl;
        return 0;
}
```

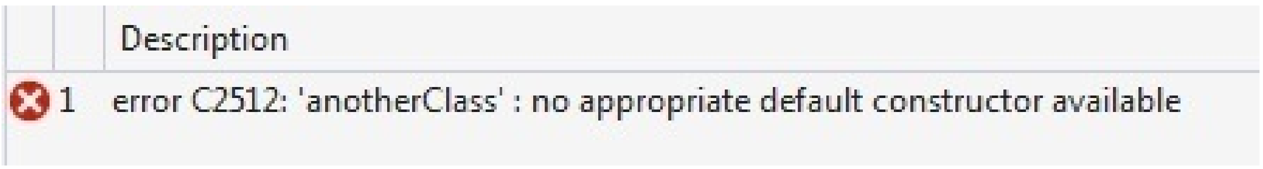
```
baseClass constructor
anotherClassObjID: -858993460
baseClassObjID: 1
```

Here we see that the *baseClass* default constructor is being called implicitly by the compiler. Why? Remember I said earlier that the compiler isn’t even generating a default constructor when we had only the integer member variable? But in this case, the compiler definitely generates a default constructor for *anotherClass*. This is because it has a member variable that has a default constructor defined. So the compiler is obliged to call it. This is the difference between a class type member variable and a POD like the integer. For a POD, the compiler is not going to do any initialization. But for a class type, if there is a default constructor defined, the compiler must call it to initialize.

So here we had a default constructor defined for *baseClass*. What if it had a constructor with parameters?

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

class baseClass
{
public:
        int baseClassObjID;
```

```
    baseClass(int ID)
    {
        baseClassObjID = ID;
        cout << “baseClass constructor” << endl;
    }
};

class anotherClass
{
public:
    int anotherClassObjID;
    baseClass baseClassObj;
};

int main(int argc, char** argv)
{
    anotherClass acObj;
    cout << “anotherClassObjID: “ << acObj.anotherClassObjID << endl;
    cout << “baseClassObjID: “ << acObj.baseClassObj.baseClassObjID << endl;
    return 0;
}
```



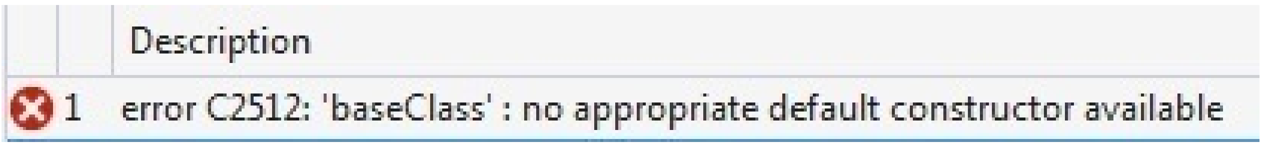
```
Description
1   error C2512: 'anotherClass' : no appropriate default constructor available
```

Compiler complains about the lack of a default constructor for *anotherClass*. What’s happening is that the compiler sees that there is a constructor defined for *baseClass*. It is not a default constructor, so it must be called explicitly. Compilers will never call non-default constructors (well, it doesn’t know what the arguments are to call anyway). But the compiler sees that there is no constructor in *anotherClass* that calls the *baseClass* constructor and emits the error. So if your member variable has a non-default constructor, your enclosing class must have a constructor that initializes the member class object.

What if you defined a constructor for *anotherClass*, but it doesn’t call the *baseClass* constructor? Something like this:

```
class anotherClass
{
public:
    int anotherClassObjID;
    baseClass baseClassObj;
```

```
	anotherClass()
	{
		anotherClassObjID = 2;
	}
};
```



```
Description
1	error C2512: 'baseClass' : no appropriate default constructor available
```

It’s the same problem. Because this is the same error we saw when we had a derived class with a base class that had no default constructor. We saw in the case of derived class constructors that we always need to call the base class constructor from the initializer list. And if we don’t initialize the base class, the compiler will try to do that before entering the constructor body. But the compiler can only call the default constructor. And if there is no default constructor, it complains.

If there is a class-type member variable with a constructor, the compiler needs to make sure it is called. If it is not called explicitly in the program, it tries to invoke the default constructor. If there is no default constructor, it will not compile.

Another thing to note here. We saw before that the compiler will generate a default constructor if one is required (for example when there is a class member variable with a constructor). But you need to remember that if there is a constructor defined by the user, the compiler will never generate a default constructor. That is why the compiler is complaining that *baseClass* doesn’t have a default constructor. Because there is a non-default constructor. So the compiler will not generate one for us.

Now how would you get around a situation like this? You have your *baseClass*, that has a constructor that takes an integer. This is how you intend to initialize your *baseClass*. How can you avoid errors for other classes which want to have *baseClass* as a member? Like *anotherClass*. Of course it becomes *anotherClass’s* responsibility to initialize *baseClass* as it is intended. But if your implementation can allow it, you can simply provide default values to your constructor parameters and the problem will go away.

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

class baseClass
{
public:
	int baseClassObjID;

	baseClass(int ID = 1)
	{
```

```
                baseClassObjID = ID;
                cout << “baseClass constructor” << endl;
        }
};

class anotherClass
{
public:
        int anotherClassObjID;
        baseClass baseClassObj;

        anotherClass()
        {
                anotherClassObjID = 2;
        }
};

int main(int argc, char** argv)
{
        anotherClass acObj;
        cout << “anotherClassObjID: “ << acObj.anotherClassObjID << endl;
        cout << “baseClassObjID: “ << acObj.baseClassObj.baseClassObjID << endl;
        return 0;
}
```

```
baseClass constructor
anotherClassObjID: 2
baseClassObjID: 1
```

See, no problem. When we gave a default value, the compiler can call it as it would a default constructor.

Let’s look at two things quickly.

- Can you have a base class instance as a member variable in the derived class?

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

class derivedClass;
class baseClass
{
```

```
public:
	int objID;

	baseClass()
	{
		cout << “(1) Default constructor” << objID << endl;
	}

	baseClass(int ID)
	{
		objID = ID;
		cout << “(2) Constructing base object with ID: “ << objID << endl;
	}
};

class derivedClass : public baseClass
{
public:
	baseClass baseClassObj;
	derivedClass() : baseClass(), baseClassObj()
	{}

	derivedClass(int ID) : baseClass(ID), baseClassObj(ID)
	{
		cout << “(3) Constructing derived object with ID: “ << objID << endl;
	}
};

int main(int argc, char** argv)
{
	derivedClass dcObj(9);
	return 0;
}
```

```
(2) Constructing base object with ID: 9
(2) Constructing base object with ID: 9
(3) Constructing derived object with ID: 9
```

So yes, you can. No problem.

- What about a derived class instance in the base class then?

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

class derivedClass;
class baseClass
{
public:
        int objID;
        derivedClass *derivedClassObj;

        baseClass();

        baseClass(int ID)
        {
                objID = ID;
                cout << “(2) Constructing base object with ID: “ << objID << endl;
        }
};

class derivedClass : public baseClass
{
public:
        derivedClass()
        {}

        derivedClass(int ID) : baseClass(ID)
        {
                cout << “(3) Constructing derived object with ID: “ << objID << endl;
        }
};


baseClass::baseClass()
{
        cout << “(1) Default constructor” << objID << endl;
        derivedClassObj = new derivedClass();
}

int main(int argc, char** argv)
{
	derivedClass dcObj;
	return 0;
}
```

Note that the derived class is a pointer, not an instance as we had with the base class in the previous example. This is because we cannot define an instance in the *baseClass*, because at that point *derivedClass* is not defined yet. We can use a pointer, with a forward declaration at the top, and move the definition of the constructor after the definition of the *derivedClass*. We will discuss more on this in another topic.

This will compile and run. But what happens? It will be stuck in an infinite loop. *derivedClass* is inherited from *baseClass*, so it has an instance of *baseClass* in itself. And this *baseClass* instance will have a *derivedClass* instance in, and so on. So you cannot have a derived class instance in the base class.

So what have we learned so far:

- If no default constructor (constructor with no arguments) is defined, the compiler will generate one (if required).
- If a constructor is defined, default or otherwise, the compiler will not generate any constructor (even if required).
- The compiler will not initialize any member variable that is a POD. They must be initialized explicitly by a constructor we define.
- If there are class-type member variables that have constructors of their own, they will be initialized by the compiler by calling the default constructors. This is important. If there is no default constructor for the class, the compiler will generate one, and this compiler generated constructor will call default constructors of the member classes. If there is a constructor defined, and this constructor is not initializing the member classes by calling their constructors, then the compiler will implicitly call the default constructors. Remember, the compiler can only call default constructors by itself. If the member classes have no default constructors, they need to be initialized explicitly, otherwise there will be compiler errors.
- Base class constructors and member class constructors must be called in the initializer list. Because the initializer list is the only place you can call a constructor.

Member initialization overhead

So is there any situation where we can initialize a class-type member in the constructor body and not in the initializer list? Yes, but you may not want to do it. Then what are the other options? Consider this:

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

```
class baseClass
{
public:
        int baseClassObjID;

        baseClass()
        {
                baseClassObjID = 99;
                cout << “(1) baseClass constructor:” << baseClassObjID << endl;
        }

        baseClass(int ID)
        {
                cout << “(2) baseClass constructor: “ << baseClassObjID << endl;
                baseClassObjID = ID;
                cout << “(3) baseClass constructor: “ << baseClassObjID << endl;
        }

        baseClass & operator =(const baseClass & baseClassObj)
        {
                baseClassObjID = baseClassObj.baseClassObjID;
                cout << “(4) baseClass assignment operator: “ << baseClassObjID << endl;
                return *this;
        }

        ~baseClass()
        {
                cout << “Destructor for object: “ << baseClassObjID << endl;
        }
};

class anotherClass
{
public:
        int anotherClassObjID;
        baseClass baseClassObj;

        anotherClass(int ID)
        {
                baseClassObj = ID;
                anotherClassObjID = 2;
        }
};

int main(int argc, char** argv)
{
        anotherClass acObj(5);
        return 0;
}
```

```
(1) baseClass constructor:99
(2) baseClass constructor: -858993460
(3) baseClass assignment operator: 5
(4) Destructor for object: 5
(4) Destructor for object: 5
```

What are we doing here? We are passing an int argument to the *anotherClass* constructor and this argument is then assigned to the *baseClass* member. Let's go through the output:

- Line 1 of the output is when the default constructor for *baseClass* is called. Where is this called? This is called in the *anotherClass* constructor. And this is called implicitly by the compiler. Why? Because as I said before, if there is a member class that has a default cosntructor defined, the compiler is obliged to call it, if and only if, there is no call to the constructor in the initialize list. We are not doing any initializing in the initializer list of *anotherClass* constructor, so the compiler is calling the default constructor for *baseClassObj*.
- Now we are inside the *anotherClass* constructor body and executing the assignment. We saw in the topic on Constructors that in this case, what the compiler does is create a temporary class with the argument *ID* and then pass this temporary object to the *baseClassObj* assignment operator. That is what you see in line 2 of the output. It is creating a temporary object by calling the constructor with the int argument. I did two printouts to show that we are dealing with different objects. So in output line 2, *baseClassObjID* is not initialized, proving that this is not the *baseClassObj* member, but a new temporary.
- Line 3 of the output is the assignment operator. The new temporary that was created before is passed to the assignment operator of *baseClassObj*. Now *baseClassObj* is set with value 5. This is what we intended to do.
- Then there are two invocations of the destructor. The first one for the temporary we created to pass to the assignment operator, and the other when *acObj* went out of scope.

Now compare that with what happens if we do this in the initializer list:

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

class baseClass
{
public:
        int baseClassObjID;

        baseClass()
        {
                baseClassObjID = 99;
                cout << “(1) baseClass constructor:” << baseClassObjID << endl;
        }

        baseClass(int ID)
        {
                cout << “(2) baseClass constructor: “ << baseClassObjID << endl;
                baseClassObjID = ID;
        }

        baseClass & operator =(const baseClass & baseClassObj)
        {
                baseClassObjID = baseClassObj.baseClassObjID;
                cout << “(3) baseClass assignment operator: “ << baseClassObjID << endl;
                return *this;
        }

        ~baseClass()
        {
                  cout << “(4) Destructor for object: “ << baseClassObjID << endl;
        }
};

class anotherClass
{
public:
        int anotherClassObjID;
        baseClass baseClassObj;

        anotherClass(int ID) : baseClassObj(5) // <— In the initializer list
        {
```

```
		anotherClassObjID = 2;
	}
};

int main(int argc, char** argv)
{
	anotherClass acObj(5);
	return 0;
}
```

```
(2) baseClass constructor: -858993460
(4) Destructor for object: 5
```

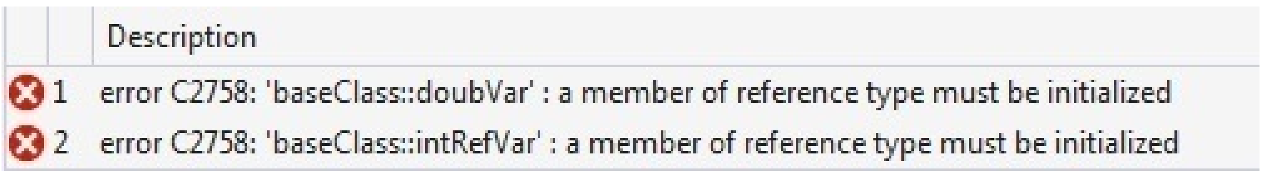
See the difference? When we do this in the initializer list it directly calls the constructor with the argument. We save a lot of steps here. First, there is no call to the default constructor and then there is no construction and destruction of the temporary object. So always try to initialize in the initializer list. It is much more efficient.

We looked at many different scenarios of constructors and I believe now you have a solid grasp on constructors. Just a few more things to talk about.

Const and reference members

We saw earlier that member variables such as integers are not initialized and also that base classes must be initialized in the initializer list. These add a restriction to const variables and references. You see, if you have consts or references as members, you must initialize them in the initializer list. You cannot omit the initialization as you would for a class-type variable and expect the compiler to do it, nor can you initialize them in the constructor body.

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

class baseClass
{
public:
	int intVar;
	const float floatVar;
	int &intRefVar;

	baseClass()
	{
		cout << "(1) Default constructor" << endl;
	}
};

int main(int argc, char** argv)
{
	baseClass bcObj;
	cout << “objID: “ << bcObj.intVar << endl;
	return 0;
}
```



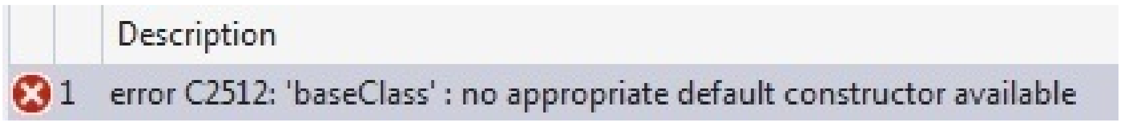
```
Description
1   error C2758: 'baseClass::doubVar' : a member of reference type must be initialized
2   error C2758: 'baseClass::intRefVar' : a member of reference type must be initialized
```

You get a compiler error that you need to initialize the const and the reference. And you cannot do this either.

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

class baseClass
{
public:
	int intVar;
	const float floatVar;
	int &intRefVar;

	baseClass()
	{
		intVar = 1;
		floatVar = 2.0f;
		intRefVar = intVar;
		cout << “(1) Default constructor” << endl;
	}
};

int main(int argc, char** argv)
{
	baseClass bcObj;
	cout << “objID: “ << bcObj.intVar << endl;
	return 0;
}
```

The const and the reference must be ‘explicitly’ initialized in the initializer list. And what if you don’t define any constructor?

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

class baseClass
{
public:
	int intVar;
	const float floatVar;
	int &intRefVar;
};


int main(int argc, char** argv)
{
	baseClass bcObj;
	return 0;
}
```



```
Description
1	error C2512: 'baseClass' : no appropriate default constructor available
```

No again. When you have a const or a reference you must define a constructor, and you must initialize them and you must do it in the initializer list. So this is how you need to write this.

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

class baseClass
{
public:
	int intVar;
	const float floatVar;
	int &intRefVar;

	baseClass() : intVar(1), floatVar(2.0f), intRefVar(intVar)
	{
```

```
                cout << “Default constructor” << endl;
        }
};

int main(int argc, char** argv)
{
        baseClass bcObj;
        cout << “floatVar: “ << bcObj.floatVar << endl;
        cout << “intRefVar: “ << bcObj.intRefVar << endl;
        return 0;
}
```

```
Default constructor
floatVar: 2
intRefVar: 1
```

So that’s about all there is to it with consts and references. Just remember to initialize them.

Member initialization order

The other little tidbit I need to discuss about the initializer list is the order of initialization. A simple example is sufficient.

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

class baseClass
{
public:
        int baseClassObjID;

        baseClass(int ID)
        {
                baseClassObjID = ID;
                cout << “baseClass constructor: “ << baseClassObjID << endl;
        }
};

class anotherClass
{
public:
	baseClass baseClassObj1;
	baseClass baseClassObj3;
	baseClass baseClassObj2;

	anotherClass(int var1, int var2, int var3) : baseClassObj1(var1), baseClassObj2(var2), baseClassObj3(var3)
	{}
};

int main(int argc, char** argv)
{
	anotherClass acObj(1, 2, 3);
	return 0;
}
```

```
baseClass constructor: 1
baseClass constructor: 3
baseClass constructor: 2
```

Did you notice something in the output? It's constructing *baseClassObj1*, then *baseClassObj3*, and then *baseClassObj2*. But in the initializer list we initialize *baseClassObj1* first, *baseClassObj2* second and finally *baseClassObj3*. This is because the compiler always initializes member variables in the order they were defined. Not in the order they are initialized in the initializer list. See how we have defined *baseClassObj1*, then *baseClassObj3* and then *baseClassObj2*. So the constructor initializes them in that order. And then the destructor destructs them in the reverse order. This fact might not have any effect on an initialization like this, but if you have member variables which are initialized with relation to another variable, then this fact becomes very crucial.

One final thing about constructors and initialization: "value initialization". Since C++11 you can initialize your class members without explicitly doing it in the constructor. Like this:

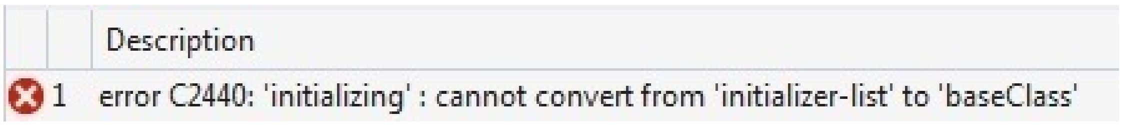
```
#include<iostream>
using namespace std;

class baseClass
{
public:
	int var1;
	int var2;
	int var3;
};


int main(int argc, char** argv)
{
	baseClass bcObj = { 10, 20, 30 };
	cout << “var1: “ << bcObj.var1 << endl;
	cout << “var2: “ << bcObj.var2 << endl;
	cout << “var3: “ << bcObj.var3 << endl;
	return 0;
}
```

```
var1: 10
var2: 20
var3: 30
```

Simple. You just define the values you want your member variables to be initialized as inside braces and the compiler does it for you. Pretty neat. So that means we don’t need to initialize them in the initializer list, right? Not so fast.

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

class baseClass
{
public:
	int var1;
	int var2;
	int var3;

	baseClass()
	{
		cout << “Constructor” << endl;
	}
};

int main(int argc, char** argv)
{
	baseClass bcObj = { 10, 20, 30 };
	cout << “var1: “ << bcObj.var1 << endl;
```

```
        cout << “var2: “ << bcObj.var2 << endl;
        cout << “var3: “ << bcObj.var3 << endl;
        return 0;
}
```



```
Description
1    error C2440: 'initializing' : cannot convert from 'initializer-list' to 'baseClass'
```

You see, if you define a constructor then the compiler is not going to do the initialization for you. It makes sense, right? If you are defining a constructor you do that to initialize the object. So you should initialize them. So when you implement your constructor, the compiler backs off and doesn’t involve in your business of initialization.

That was pretty lengthy but we covered a lot of ground here. I believe you gained quite a bit of insight as to how constructors are working.

Topic 9

Forward Declarations, Compiling and Linking

Sometimes we take things for granted. If you are using an IDE, like Visual Studio or Eclipse, you build your project and run it, never thinking about compiling and linking. But these two steps happen all the time and it's useful to understand its basics.

Function forward declaration

Let's start simple.

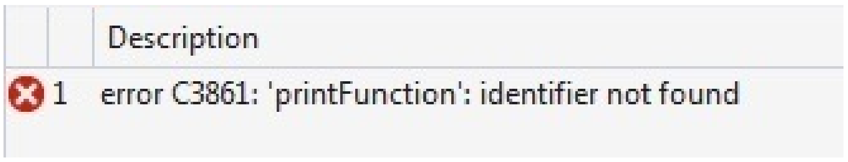
```
#include<iostream>
using namespace std;

void printFunction(int var); // forward declaration

class simpleClass
{
public:
	int var;

	simpleClass(int value)
	{
		var = value;
		printFunction(var);
	}
};

void printFunction(int var)
{
	cout << “Passed value is: “ << var << endl;
}

int main(int argc, char** argv)
{
	simpleClass(10);
	return 0;
}
```

```
Passed value is: 10
```

This program isn't doing anything meaningful. We instantiate an object by passing it an integer, and the constructor passes this value to *printFunction* to print it, or do whatever it does.

These are the starting points:

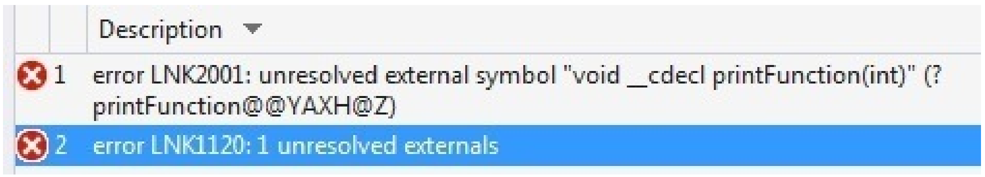
- Before the start of the class definition we have forward declared the function. This is the *function prototype*.
- The *simpleClass* constructor calls *printFunction* and passes it the integer. It's important to understand that at this point, the compiler does not know the definition of the *printFunction*. It simply knows that there is a function by that name and it takes an integer argument and returns nothing. Comment out the forward declaration at the top and you will get this:

  ```
  Description
  1   error C3861: 'printFunction': identifier not found
  ```

  So you see, you get a compiler error, even though we have defined the function (after the class definition), the compiler isn't aware of it. This is because the compiler is going through the code sequentially. So when it is in the constructor it has no idea that the function definition is right below it. Go ahead and uncomment the forward declaration now.

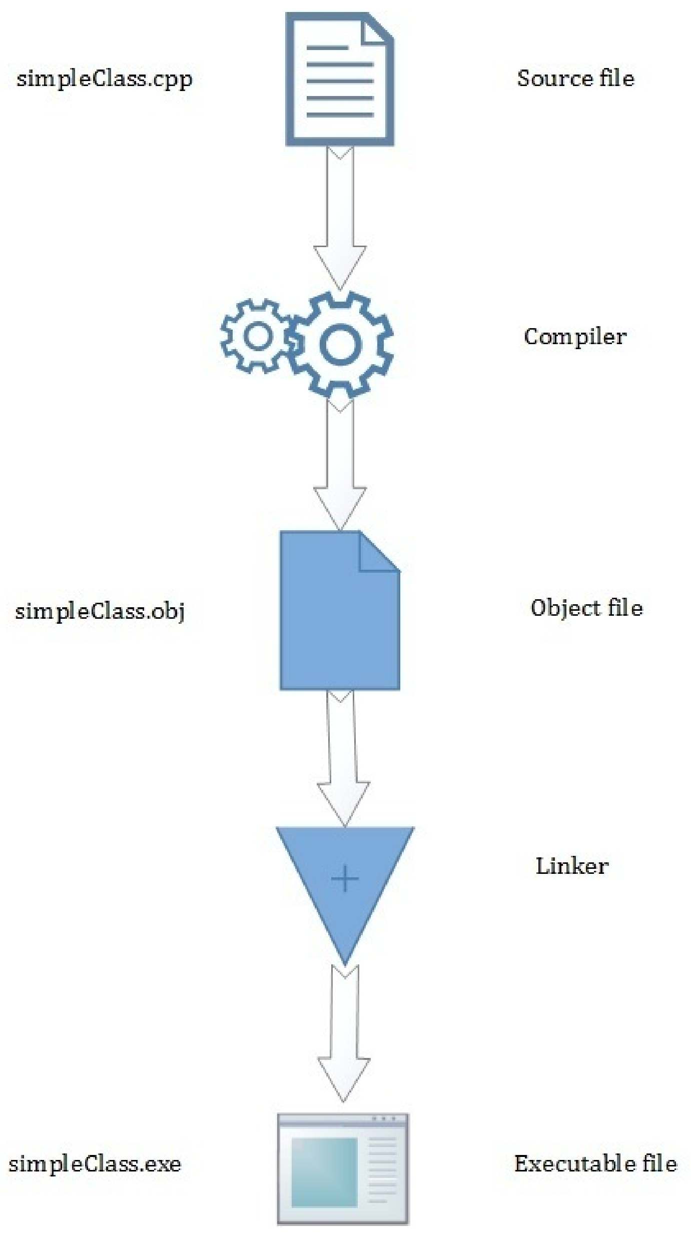
- Now keep the forward declaration and comment out the entire function definition. You should get en error like this:

  ```
  Description
  1   error LNK2001: unresolved external symbol "void __cdecl printFunction(int)" (?
      printFunction@@YAXH@Z)
  2   error LNK1120: 1 unresolved externals
  ```

  This is a linker error. Earlier it was a compiler error. It is important to note the difference. Linking is something that happens after compiling. So here we know that our code compiled fine. But it couldn't link. The linker errors aren't as friendly as compiler errors. But you can make out that it is complaining about the *printFunction*.

Before we go any further it's important to understand the compiling and linking a little bit. The figure below shows a very basic mechanism of compiling and linking.



Your source file with the code is first analyzed by the compiler. This is where errors such as syntaxes, definitions, etc. are found. The compiler then generates an object file. If you are using a Unix environment it will be an *.o* file, or if you are using Visual Studio it would be an *.obj* file. This object file is then sent to the linker, which outputs the executable file. In the example we did so far we had only one source file. But note that we are making use of the <*iostream*> library. This is where the linker does its work. You see, the object file contains placeholders. These placeholders are for variables or methods which are not defined in the source file itself. What the linker does is take this object file and fill the placeholders with other object files. Let’s modify our example to see this in action.

We will have our main file as follows:

```
//main.cpp
#include<iostream>
using namespace std;

void printFunction(int var); // forward declaration
```

```
class simpleClass
{
public:
	int var;
	simpleClass(int value)
	{
		var = value;
		printFunction(var);
	}
};

int main(int argc, char** argv)
{
	simpleClass(10);
	return 0;
}
```

Note that we have the forward declaration for ‘printFunction’ but the definition is not there. Now compile this. If you are using Visual Studio you can do “Ctrl+F7”, or you can do it through Visual Studio command prompt. You can compile by

```
CL /c main.cpp
```

This will create a *main.obj* file. If you are using an Unix environment you can do

```
g++ main.cpp main.o
```

You will see that you get no errors. If you try to build this program, however, you will get the linker error we encountered before. So now you know that the compiler is happy with this program, but the linker is not.

So why does the compiler have no complaints, but the linker is throwing an error? Because you see, the compiler is perfectly happy with the forward declaration. The compiler is confident that there is a function named *printFunction* defined somewhere and this function takes an int argument and returns void. The compiler trusts you when you forward declared the function. From the compiler’s point of view it does not need to know anything more. It simply puts a placeholder where we call the *printFunction* that says something like “*add the printFunction address here*” and creates the object file.

Now it becomes the linker's job to find where the definition of *printFunction* is and fill that placeholder. But you see we have not defined the function anywhere. So the linker naturally complains.

Now let's go ahead and define it. But this time, let's do it in a separate file. Call it printFunction.cpp. It should just have the definition we had before in our main.cpp file.

```
//printFunction.cpp
#include<iostream>
using namespace std;

void printFunction(int var)
{
        cout << "Passed value is: " << var << endl;
}
```

Now if you build the project you will see that it compiles and links and runs without any issues. So you know that the linker found it and 'linked'. But let's take it step by step.

What we want to do now is to compile the printFunction.cpp. Do what we did before.

```
CL /c printFunction.cpp
```

This should create a printFunction.obj or an .o file, depending on your environment. Again, you see that the compiler has no issues. What we want to do now is to link our two object files. Let's first pass only main.obj to the compiler and see what it thinks.

```
LINK main.obj
```

You should see the same linker error we got before. So now you have an idea of what happens when you build a project in Visual Studio. It does compiling and linking both in one step for you. So you see the linker is not happy with the information it has. Or the lack of it.

Let's give the linker more information to work with.

```
LINK main.obj printFunction.obj
```

You will see that the linker has no complaints now. It creates the executable file. Because now we passed the *printFunction* object file the linker can now fill the *printFunction* placeholder in the *main.obj* file with the function definition in *printFunction.obj*.

What we discussed here is the very basic functionality of the compiler and linker but it should give you an understanding of what is happening behind the scenes.

Forward declaring class-types

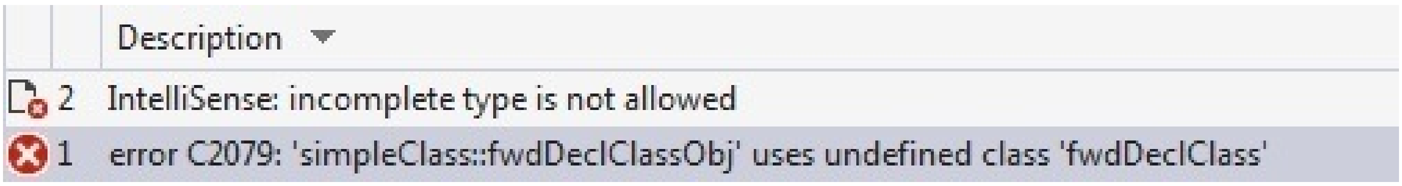
Now let's look at this slightly differently. Instead of a function, we will define a class. Our main file will be like this:

```
// main.cpp
#include<iostream>
using namespace std;

class fwdDeclClass; // forward declaration

class simpleClass
{
public:
	fwdDeclClass fwdDeclClassObj;

	simpleClass() : fwdDeclClassObj()
	{}
};

int main(int argc, char** argv)
{
	simpleClass simpleClassObj;
	return 0;
}
```

And we will define our fwdDeclClass like this.

```
// fwdDeclClass.cpp
#include<iostream>
using namespace std;

class fwdDeclClass
{
public:
	fwdDeclClass()
	{
        cout << “fwdDeclClass cosntructor” << endl;
    }
};
```

Now try to compile your main.cpp. You will get an error similar to:



```
    Description
2   IntelliSense: incomplete type is not allowed
1   error C2079: 'simpleClass::fwdDeclClassObj' uses undefined class 'fwdDeclClass'
```

The compiler is basically complaining about using an undefined class. But why? Let’s go ahead and try to compile our fwdDeclClass.cpp. You should see no errors. It should compile fine. So then, why is our main.cpp refusing to compile? Why is it complaining that *fwdDeclClass* is undefined when we have forward declared it? OK, let’s change our *simpleClass* so that instead of a *fwdDeclClass* instance it has a pointer to a *fwdDeclClass*.

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

class fwdDeclClass; // forward declaration

class simpleClass
{
public:
    fwdDeclClass *fwdDeclClassPtr;

    simpleClass()
    {}
};

int main(int argc, char** argv)
{
    simpleClass simpleClassObj;
    return 0;
}
```

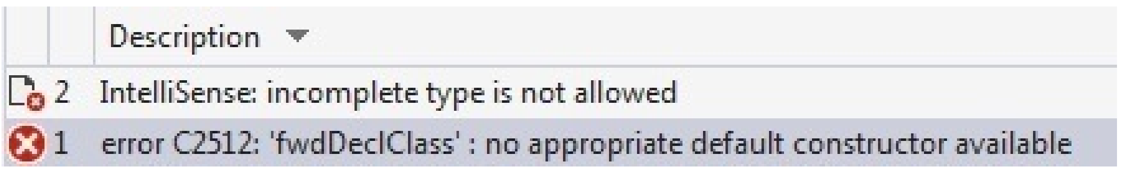
This should compile fine. In fact, you can build and run. The question is why the compiler couldn’t work with the forward declaration when we had a member class object, but it is fine with a pointer to that class?

The reason is, the compiler doesn’t need to know anything about the class to have a pointer. A pointer occupies the same memory space regardless of its type. A pointer to *fwdDeclClass* and a pointer to an integer are both of same size. So the compiler can allocate memory to the *fwdDeclClass* pointer without having to know its implementation. This is the same case when we forward declared a function. The compiler only needed to know the function signature to check the syntax and put a placeholder for the linker. That's all the compiler had to do with the function call, and the forward declaration provided the necessary information.

But this is not the case when we have a class object. When we have a class object as a member, the compiler must know about the class composition because it needs to allocate memory for it. That object is going to be part of the class itself. So the compiler needs to know about *fwdDeclClass* to instantiate it inside *simpleClass*. And the compiler cannot do this with just the forward declaration. It contains no information other than that there is a class by the name *fwdDeclClass*.

This will become clearer when you modify the *simpleClass* constructor to initialize the *fwdDeclClassPtr*.

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

class fwdDeclClass; // forward declaration

class simpleClass
{
public:
        fwdDeclClass *fwdDeclClassPtr;

        simpleClass()
        {
                fwdDeclClassPtr = new fwdDeclClass();
        }
};

int main(int argc, char** argv)
{
        simpleClass simpleClassObj;
        return 0;
}
```



```
      Description
2   IntelliSense: incomplete type is not allowed
1   error C2512: 'fwdDeclClass' : no appropriate default constructor available
```

Can you see why now? We are trying to initialize *fwdDeclClassPtr* with an instance of *fwdDeclClass* and you can see why this is going to be a problem for the compiler. The compiler knows nothing about *fwdDeclClass*. It doesn't know whether fwdDeclClass has a default constructor or not. Now if you just include "fwdDeclClass.cpp" to main.cpp the compiler errors will be gone (usually you would have *fwdDeclClass* in a .h header file).

```
#include<iostream>
using namespace std;
#include "fwdDeclClass.cpp"

class simpleClass
{
public:
        fwdDeclClass *fwdDeclClassPtr;
        fwdDeclClass fwdDeclClassObj;

        simpleClass(int value) : fwdDeclClassObj()
        {
                fwdDeclClassPtr = new fwdDeclClass();
        }

        ~simpleClass()
        {
                delete fwdDeclClassPtr;
        }
};

int main(int argc, char** argv)
{
        simpleClass simpleClassObj;
        return 0;
}
```

```
fwdDeclClass cosntructor
fwdDeclClass cosntructor
```

Before we finish this topic off let me ask this. What if we have an undefined function in the class? We saw earlier that if we call this function, then we'd have a linker error. But what if we don't call this function in the code? Would there still be errors? Let's find out quickly.

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

class simpleClass
{
public:
        int var;

        simpleClass(int value)
        {
                var = value;
                memberFunc(var); // <— Line A
        }

        void memberFunc(int var);
};

int main(int argc, char** argv)
{
        simpleClass(10);
        return 0;
}
```

Try compiling and then running the code above. You'd see that it compiles fine but does not link. That is because we have declared the *memberFunc* in the class so the compiler is aware of it. But there is a linker error and we know why. Because there is no definition for the function. Now what would happen if we comment out line "A"? That is, we don't call our *memberFunc* anywhere in the code. You will see that neither the compiler nor the linker has any problems. It compiles and runs. So you see, if we are not calling a function we don't need to provide its definition. It is when we call them that the linker starts looking for the implementation.

Did you notice something odd about the code above? Did you notice that the declaration of *memberFunc* is after the call to the function in the constructor? Shouldn't we get an "identifier not found" compiler error, because the compiler is not aware of *memberFunc* when in the constructor? Well, in C++, classes are compiled differently. The compiler actually goes through the class twice. The first time it will look for all of the declarations (including defined ones) available, and then in the next round it will perform the compilation.

So those are the very fundamentals of forward declaration and linking. You see forward declaration doesn't always work. It only works when the declaration provides the compiler enough information to compile.

Topic 10

Copy Constructor and Object Cloning

We discussed copy constructors before but in this topic we’ll look at their functionality in more detail and see how we can clone objects.

Class member copy

Let’s start simple.

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

class simpleClass
{
public:
	int val;

	simpleClass(int value)
	{
		val = value;
		cout << “SimpleClass constructor: “ << val << endl;
	}
};


class anotherClass
{
public:
	int var;
	int* varPtr;
	simpleClass simpleObj;

	anotherClass(int value) : var(value), varPtr(&var), simpleObj(value)
	{
		cout << “anotherClass constructor: “ << var << endl;
	}
};

int main(int argc, char** argv)
{
	anotherClass anotherObj1(10);
```

```
    anotherClass anotherObj2(anotherObj1);

    cout << endl;
    cout << “anotherObject2.simpleObj.val: “ << anotherObj2.simpleObj.val << endl;
    cout << “anotherObject2.var: “ << anotherObj2.var << endl;
    cout << “*anotherObject2.varPtr: “ << *anotherObj2.varPtr << endl;
    return 0;
}
```

```
SimpleClass constructor: 10
anotherClass constructor: 10

anotherObject2.simpleObj.var1: 10
anotherObject2.var: 10
*anotherObject2.varPtr: 10
```

We aren’t doing much here. We have two classes, *simpleClass* and *anotherClass*. *simpleClass* is a member of *anotherClass*. There is an int pointer in *anotherClass* that points to its own *var*. We are instantiating *anotherObj1* and then use that instance to create *anotherObj2*. We print out the values of *anotherObj2* and we see that it has copied the correct values for not only *anotherClass* member but also the *simpleClass*::*val*. Everything looks nice and dandy. The compiler generated copy constructor seems to have done a superb job, or has it?

Actually, it is an absolute disaster. Let’s just print out the pointers of the two objects and you will see.

```
…
…
int main(int argc, char** argv)
{
    anotherClass anotherObj1(10);
    anotherClass anotherObj2(anotherObj1);

    cout << endl;
    cout << “anotherObject1.varPtr: “ << anotherObj1.varPtr << endl;
    cout << “anotherObject2.varPtr: “ << anotherObj2.varPtr << endl;
    return 0;
}
```

```
SimpleClass constructor: 10
anotherClass constructor: 10

anotherObject1.varPtr: 0029FD90
anotherObject2.varPtr: 0029FD90
```

Here we are printing out the pointer addresses, not what it points to as we did before. And you see they are the same. What does that mean? Yes, it means they are both pointing to the same variable! Can you see how this is absolutely not what we'd want? What we wanted was just to copy the values of *anotherObj1*, but we wanted an independent object. Not one that is tied to the original. Can you see how this can go wrong in so many ways?

Let's see some examples. (I have omitted the class definitions. They do not change.)

- Disaster #1:

```
int main(int argc, char** argv)
{
        anotherClass anotherObj1(10);
        anotherClass anotherObj2(anotherObj1);

        cout << endl;
        cout << "anotherObject1.var: " << anotherObj1.var << endl;
        cout << "*anotherObject1.varPtr: " << *anotherObj1.varPtr << endl;
        cout << "anotherObject2.var: " << anotherObj2.var << endl;
        cout << "*anotherObject2.varPtr: " << *anotherObj2.varPtr << endl << endl;

        anotherObj1.var = 20;

        cout << "anotherObject1.var: " << anotherObj1.var << endl;
        cout << "*anotherObject1.varPtr: " << *anotherObj1.varPtr << endl;
        cout << "anotherObject2.var: " << anotherObj2.var << endl;
        cout << "*anotherObject2.varPtr: " << *anotherObj2.varPtr << endl << endl;
        return 0;
}
```

```
anotherObject1.var: 10
*anotherObject1.varPtr: 10
anotherObject2.var: 10
*anotherObject2.varPtr: 10

anotherObject1.var: 20
```

```
*anotherObject1.varPtr: 20
anotherObject2.var: 10
*anotherObject2.varPtr: 20

anotherObject1.var: 30
*anotherObject1.varPtr: 30
anotherObject2.var: 10
*anotherObject2.varPtrPtr: 30
```

In the first part of the printout we see that *var* of both objects have a value of 10, which is what we expect, and the two dereferenced pointers are also 10s. So far things look right. Then we see how things go horribly wrong in the second part of the printout. What we do is to change var of *anotherObj1* to 20. What we would expect is *varPtr* of *anotherObj1* to have a value of 20, and we expect absolutely no change in *anotherObj2*. But that is not what we are seeing. Then in the third part, we dereference *varPtr* of *anotherObj2* and change the value. And this changed the var of *anotherObj1*. This is madness!

So these two objects are acting completely against our expectations. *varPtr* is not pointing to *var* of *this* object. It is pointing to var of *anotherObj1*. That means every time we change *varPtr* of one object, it affects the other too. Why is this happening then?

It is because the compiler generated copy constructor has done a *'bit-wise'* copy. That means the copy constructor has copied the *anotherObj1* to *anotherObj2*, bit-to-bit. Whatever the bits were in *anotherObj1.var* was copied to *anotherObj2.var2* memory space. And the same for *varPtr*. It has basically done a memcopy. The compiler generated copy constructor could very well be of the form:

```
	anotherClass(const anotherClass& objToCopy)
	{
		std::memcpy(this, &objToCopy, sizeof(anotherClass));
	}
```

Another clue should be the constructor printouts. See, in the first example there was only one call to each constructor of the classes. That means when we copied the object, the constructor was never called. There are cases where this is fine. But this is absolutely not right when we have a pointer. You see the pointer is the root cause of all this madness. Both *varPtr*'s point to *var* of *anotherObj1*. Here's another way that this can go wrong:

- Disaster #2:

```
…
…
int main(int argc, char** argv)
{
        anotherClass* anotherPtr1 = new anotherClass(10);
        anotherClass* anotherPtr2 = new anotherClass(*anotherPtr1);

        cout << endl;
        cout << "anotherPtr1->var: " << anotherPtr1->var << endl;
        cout << "*anotherPtr1->varPtr: " << *anotherPtr1->varPtr << endl;
        cout << "anotherPtr2->var: " << anotherPtr2->var << endl;
        cout << "*anotherPtr2->varPtr: " << *anotherPtr2->varPtr << endl << endl;

        delete anotherPtr1;

        cout << "anotherPtr2->var: " << anotherPtr2->var << endl;
        cout << "*anotherPtr2->varPtr: " << *anotherPtr2->varPtr << endl << endl;
        return 0;
}
```

```
anotherPtr1->var2: 10
*anotherPtr1->var2Ptr: 10
anotherPtr2->var2: 10
*anotherPtr2->var2Ptr: 10

anotherPtr2->var2: 10
*anotherPtr2->var2Ptr: -572662307
```

We are using pointers here. It’s easy to see what’s going wrong here. *varPtr* of *anotherPtr2* is pointing to var of *anotherPtr1* object. So when we delete *anotherPtr1*, the variable *anotherPtr->varPtr* was pointing to was deleted. We are dereferencing a deleted pointer. This has undefined behavior.

So you see why it is very important to pay special attention to the copy constructor, and of course the assignment operator, which does a similar thing. Never rely on compiler generated functions to do what you are expecting them to do. Let’s fix this example.

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

class simpleClass
{
public:
        int val;
        simpleClass(int value) : val(value)
        {
                cout << “SimpleClass constructor: “ << val << endl;
        }

        simpleClass(const simpleClass& objToCopy) : val(objToCopy.val)
        {
                cout << “SimpleClass copy constructor: “ << endl;
        }
};

class anotherClass
{
public:
        int var;
        int* varPtr;
        simpleClass simpleObj;

        anotherClass(int value) : var(value), varPtr(&var), simpleObj(value)
        {
                cout << “anotherClass constructor: “ << var << endl;
        }

        anotherClass(const anotherClass& objToCopy) : var(objToCopy.var), varPtr(&var),
        simpleObj(objToCopy.simpleObj)
        {
                cout << “anotherClass copy constructor: “ << endl;
        }

};

int main(int argc, char** argv)
{
        anotherClass anotherObj1(10);
        anotherClass anotherObj2(anotherObj1);

        cout << endl;
        cout << “anotherObject1.var: “ << anotherObj1.var << endl;
        cout << “*anotherObject1.varPtr: “ << *anotherObj1.varPtr << endl;
        cout << “anotherObject2.var: “ << anotherObj2.var << endl;
        cout << “*anotherObject2.varPtr: “ << *anotherObj2.varPtr << endl << endl;
```

```
	anotherObj1.var = 20;

	cout << “anotherObject1.var: “ << anotherObj1.var << endl;
	cout << “*anotherObject1.varPtr: “ << *anotherObj1.varPtr << endl;
	cout << “anotherObject2.var: “ << anotherObj2.var << endl;
	cout << “*anotherObject2.varPtr: “ << *anotherObj2.varPtr << endl << endl;

	return 0;
}
```

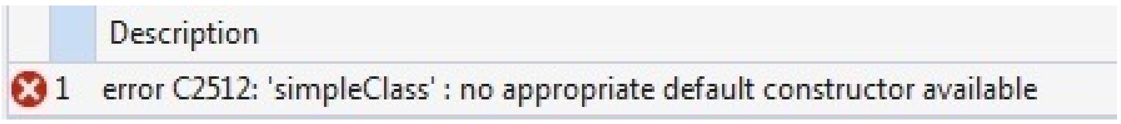
```
SimpleClass constructor: 10
anotherClass constructor: 10
SimpleClass copy constructor:
anotherClass copy constructor:

anotherObject1.var: 10
*anotherObject1.varPtr: 10
anotherObject2.var: 10
*anotherObject2.varPtr: 10

anotherObject1.var: 20
*anotherObject1.varPtr: 20
anotherObject2.var: 10
*anotherObject2.varPtr: 10
```

Everything is working as expected. There is no link between *anotherObj1* and *anotherObj2*. They are two independent objects now. See how the copy constructor is called for both *simpleClass* and *anotherClass*? This shows that the defined copy constructor is used by the compiler to do the copying. If you haven’t defined it then the compiler will do the bit-wise copy itself.

It’s time we can introduce the words *‘shallow’* copy and *‘deep’* copy. Shallow copy is what the compiler generated copy constructor would do. It is simply just copying member to member, bit-by-bit. This is what we said is bit-wise copy. But in a situation like when we have pointers or dynamically allocated memory, we need to do ‘deep’ copy. What we did above is sort of deep copy. Do not expect the compiler to do any deep copying for you.

Let’s take some time now to explore the behavior of the copy constructor. See how I called the *simpleObj* copy constructor in the initializer list? You see this must be called in the initializer list. The copy constructor cannot be called within the body. Sounds familiar? Remember we saw the same thing with calling base class constructors or member class constructors. Just remember the copy constructor is another kind of constructor. There isn't much difference. The copy constructor constructs the object by copying another object. But both types of constructors do the same thing; construct the object. So the rules are the same. Now what if I didn't call the copy constructor of *simpleObj* at all?

```
…
…
class anotherClass
{
public:
        int var;
        int* varPtr;
        simpleClass simpleObj;

        anotherClass(int value) : var(value), varPtr(&var), simpleObj(value)
        {}

        anotherClass(const anotherClass& objToCopy)
        {
                var = objToCopy.var;
                varPtr = &var;
        }
};
…
…
```


Description
1 error C2512: 'simpleClass' : no appropriate default constructor available

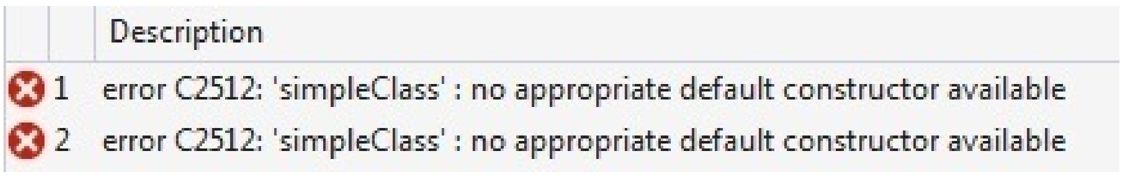
Why is the compiler complaining there is no default constructor? This is what we discussed earlier, that the copy constructor is another constructor. The compiler sees that the *simpleClass* has a constructor and so the compiler is now obliged to call that constructor. And it is complaining that there is no default constructor. Isn't this the same thing we saw when calling derived class constructors or classes that have a member class with a constructor? It's the same thing. The compiler is trying to construct the *anotherClass* object and there is a *simpleClass* object in it which has a constructor defined. And now the compiler needs to call the constructor and it cannot because there is no default one. Remember what I said before, the compiler can only call default constructors. Let's look at two different scenarios now.

1. *simpleClass* with no constructor:

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

```
class simpleClass
{
public:
	int val;
};

class anotherClass
{
public:
	int var;
	int* varPtr;
	simpleClass simpleObj;

	anotherClass(int value) : var(value), varPtr(&var)
	{}

	anotherClass(const anotherClass& objToCopy)
	{
		var = objToCopy.var;
		varPtr = &var;
	}

};

int main(int argc, char** argv)
{
	anotherClass anotherObj1(10);
	anotherClass anotherObj2(anotherObj1);
	return 0;
}
```

This compiles and runs fine. But note how I removed both the constructor and the copy constructor from *simpleClass*? Because had I left the copy constructor there, I would've gotten:



```
Description
1   error C2512: 'simpleClass' : no appropriate default constructor available
2   error C2512: 'simpleClass' : no appropriate default constructor available
```

Why two errors? One for the constructor and the other for the copy constructor. So this should convince you that for the compiler, the copy constructor is a constructor that takes const reference argument. Nothing more than that. This brings us to another point: if you ever implement the copy constructor for a class you must implement the constructor too, because the compiler will not do that for you now.

2. *simpleClass* with default constructor:

Now you should already know the answer to this one.

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

class simpleClass
{
public:
	int val;

	simpleClass(int value = 99) : val(value)
	{
		cout << “SimpleClass constructor: “ << val << endl;
	}

	simpleClass(const simpleClass& objToCopy)
	{
		val = objToCopy.val;

		cout << “SimpleClass copy constructor: “ << endl;
	}
};


class anotherClass
{
public:
	int var;
	int* varPtr;
	simpleClass simpleObj;

	anotherClass(int value) : var(value), varPtr(&var), simpleObj(value)
	{}

	anotherClass(const anotherClass& objToCopy)
	{
		var = objToCopy.var;
		varPtr = &var;
	}

};

int main(int argc, char** argv)
{
	anotherClass anotherObj1(10);
	anotherClass anotherObj2(anotherObj1);
	return 0;
}
```

```
SimpleClass constructor: 10
SimpleClass constructor: 99
```

It’s easy to see what is going on here. Construction of *anotherObj1* calls the constructor with the argument 10. The copy constructor of *anotherObj2* calls the constructor with the default argument, 99. As we’ve seen before, a constructor with a default argument is a default constructor.

OK, so we have defined a copy constructor that does what we want. When we didn’t define the copy constructor the compiler did a bit-wise copy and landed us in all sorts of trouble. What if we define a constructor that does nothing? It should be pretty intuitive but let’s see for ourselves anyway.

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

class simpleClass
{
public:
	int val;

	simpleClass(int value = 99) : val(value)
	{
		cout << “SimpleClass constructor: “ << val << endl;
	}

	simpleClass(const simpleClass& objToCopy)
	{
		val = objToCopy.val;
```

```
                cout << “SimpleClass copy constructor: “ << endl;
        }
};


class anotherClass
{
public:
        int var;
        int* varPtr;
        simpleClass simpleObj;

        anotherClass(int value) : var(value), varPtr(&var), simpleObj(value)
        {}

        anotherClass(const anotherClass& objToCopy)
        {
                cout << “anotherClass copy constructor” << endl;
        }

};

int main(int argc, char** argv)
{
        anotherClass anotherObj1(10);
        anotherClass anotherObj2(anotherObj1);

        cout << endl;
        cout << “anotherObject1.var: “ << anotherObj1.var << endl;
        cout << “*anotherObject1.varPtr: “ << *anotherObj1.varPtr << endl;
        cout << “anotherObject2.var: “ << anotherObj2.var << endl;
        cout << “*anotherObject2.varPtr: “ << *anotherObj2.varPtr << endl << endl;
        return 0;
}
```

```
SimpleClass constructor: 10
SimpleClass constructor: 99
anotherClass copy constructor

anotherObject1.var: 10
*anotherObject1.varPtr: 10
```

```
anotherObject2.var: -858993460
```

There is no telling what will actually happen. One thing for sure is that this code will not run properly. Depending on the environment you'd probably be getting a runtime exception at least. You see, when you didn't implement any copying in the copy constructor, the compiler didn't do anything. So the *var* and *varPtr* were uninitialized in *anotherObj2*. And we tried to use those variables, the behavior of which is undefined. We are being reminded again how the copy constructor is another constructor. If you define it, implement the full functionality. Because once you define it the compiler completely takes its hands off and will give you no assistance.

Now let's look at how the copy constructor should work in a derived class. It's not that much different from what we've seen so far so let's go through it quickly.

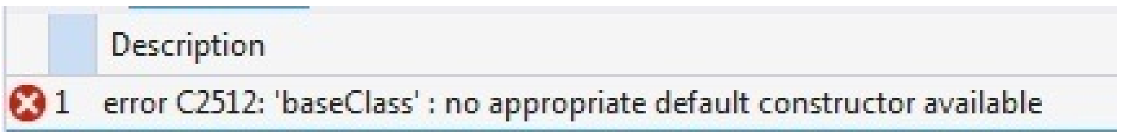
```
#include<iostream>
using namespace std;

class baseClass
{
public:
        int var;
        int* varPtr;

        baseClass(int value) : var(value), varPtr(&var)
        {}
};

class derivedClass : public baseClass
{
public:
        derivedClass(int value) : baseClass(value)
        {}
};


int main(int argc, char** argv)
{
        derivedClass derivedObj1(5);
        derivedClass derivedObj2(derivedObj1);

        cout << "redivedObj1.var: " << derivedObj1.var << endl;
        cout << "redivedObj1.varPtr: " << derivedObj1.varPtr << endl;
        cout << "redivedObj2.var: " << derivedObj2.var << endl;
        cout << “redivedObj2.varPtr: “ << derivedObj2.varPtr << endl;
        return 0;
}
```

```
redivedObj1.var: 5
redivedObj1.varPtr: 002DFD1C
redivedObj2.var: 5
redivedObj2.varPtr: 002DFD1C
```

Here we have a base class and have derived a class from it. The base class has an int variable and a pointer variable. Same as our examples earlier. No copy constructor is defined. So the compiler is doing the copying. Looking at the output, we are certain that the compiler is doing shallow copy. It is bitwise copy. No different from the copying we saw earlier.

Let’s define a copy constructor for the *derivedClass*. Since it has nothing to copy in it we’ll leave it blank.

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

class baseClass
{
public:
        int var;
        int* varPtr;

        baseClass(int value) : var(value), varPtr(&var)
        {
                cout << “baseClass constructor” << endl;
        }
};

class derivedClass : public baseClass
{
public:
        derivedClass(int value) : baseClass(value)
        {}

        derivedClass(const derivedClass& objToCopy)
        {}
};


int main(int argc, char** argv)
{
	derivedClass derivedObj1(5);
	derivedClass derivedObj2(derivedObj1);

	cout << “redivedObj1.var: “ << derivedObj1.var << endl;
	cout << “redivedObj1.varPtr: “ << derivedObj1.varPtr << endl;
	cout << “redivedObj2.var: “ << derivedObj2.var << endl;
	cout << “redivedObj2.varPtr: “ << derivedObj2.varPtr << endl;
	return 0;
}
```



```
Description
1   error C2512: 'baseClass' : no appropriate default constructor available
```

By now you should’ve anticipated this error. You know what is happening. We are defining a copy constructor for the derived class but not calling any copy constructor in the base class. The compiler needs to construct the base class so it tries to call the default constructor. Remember the compiler can only call the default constructor by itself. It finds no default constructor defined and hence the error. Let’s fix this code and see how this should be written.

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

class baseClass
{
public:
	int var;
	int* varPtr;

	baseClass(int value) : var(value), varPtr(&var)
	{
		cout << “baseClass constructor” << endl;
	}

	baseClass(const baseClass& objToCopy) : var(objToCopy.var), varPtr(&var)
	{
		cout << “baseClass copy constructor” << endl;
```

```
        }
};

class derivedClass : public baseClass
{
public:
        derivedClass(int value) : baseClass(value)
        {
                cout << “derivedClass constructor” << endl;
        }

        derivedClass(const derivedClass& objToCopy) : baseClass(objToCopy)
        {
                cout << “derivedClass copy constructor” << endl;
        }
};

int main(int argc, char** argv)
{
        derivedClass derivedObj1(5);
        derivedClass derivedObj2(derivedObj1);

        cout << endl;
        cout << “redivedObj1.var: “ << derivedObj1.var << endl;
        cout << “redivedObj1.varPtr: “ << derivedObj1.varPtr << endl;
        cout << “redivedObj2.var: “ << derivedObj2.var << endl;
        cout << “redivedObj2.varPtr: “ << derivedObj2.varPtr << endl;
        return 0;
}
```

```
baseClass constructor
derivedClass constructor
baseClass copy constructor
derivedClass copy constructor

redivedObj1.var: 5
redivedObj1.varPtr: 0017FE88
redivedObj2.var: 5
redivedObj2.varPtr: 0017FE78
```

A copy constructor of a derived class is no different from a derived class constructor. You need to call the base class copy constructor in the initializer list. Never in the body. If you tried to do this:

```
derivedClass(const derivedClass& objToCopy)
        {
                baseClass(objToCopy);
                cout << "derivedClass copy constructor" << endl;
        }
```

The compiler will complain that there is no default constructor for *baseClass*. As you already know, this is because the compiler now needs to call a constructor to create the object and it cannot find one. What if we had a default constructor for *baseClass*? As we discussed in the topic on object construction, in that case the compiler treats "*baseClass(objToCopy)*" as a declaration of *objToCopy* of type *baseClass* and complains about the redefinition of '*objToCopy*'.

Copying consts and references

In the topic on constructors we discovered that when we have a const or a reference member, variables behave a little differently. In that case we must provide a constructor and we must initialize those variables in the initializer list. What about the copy constructor?

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

class baseClass
{
public:
        int var;
        int* varPtr;
        const float floatVar;
        int &intRefVar;

        baseClass(int value) : var(value), varPtr(&var), floatVar(value), intRefVar(var)
        {
                cout << "baseClass constructor" << endl;
        }
};

class derivedClass : public baseClass
{
```

```
public:
        derivedClass(int value) : baseClass(value)
        {
                cout << “derivedClass constructor” << endl;
        }

        derivedClass(const derivedClass& objToCopy) : baseClass(objToCopy)
        {
                cout << “derivedClass copy constructor” << endl;
        }
};

int main(int argc, char** argv)
{
        derivedClass derivedObj1(5);
        derivedClass derivedObj2(derivedObj1);

        cout << endl;
        cout << “derivedObj1.var: “ << derivedObj1.var << endl;
        cout << “derivedObj1.varPtr: “ << derivedObj1.varPtr << endl;
        cout << “derivedObj1.intRefVar: “ << derivedObj1.intRefVar << endl;
        cout << “derivedObj2.var: “ << derivedObj2.var << endl;
        cout << “derivedObj2.varPtr: “ << derivedObj2.varPtr << endl;
        cout << “derivedObj2.intRefVar: “ << derivedObj2.intRefVar << endl;

        derivedObj1.intRefVar = 99;
        cout << “redivedObj2.intRefVar: “ << derivedObj2.intRefVar << endl << endl;
        return 0;
}
```

```
baseClass constructor
derivedClass constructor
derivedClass copy constructor

redivedObj1.var: 5
redivedObj1.varPtr: 0043FEB0
redivedObj1.intRefVar: 5
redivedObj2.var: 5
redivedObj2.varPtr: 0043FEB0
redivedObj2.intRefVar: 5
```

```
redivedObj2.intRefVar: 99
```

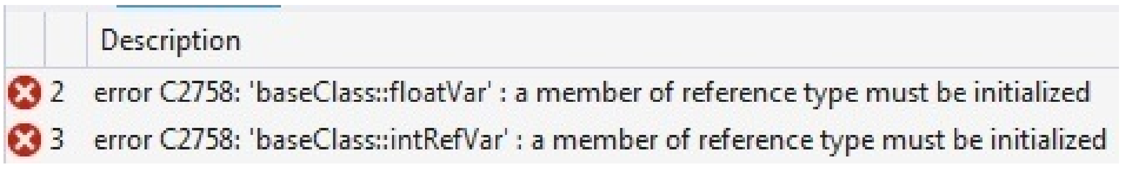
So apparently the compiler has no problem with the fact that we are not defining a copy constructor to explicitly initialize the const and the reference. You can see from the output that the compiler has done a shallow copy. And you know the consequences of that. The references of the two objects are tied. Changing one affects the other object. This is the same issue we saw earlier with the pointer. So there is a subtle difference here with the copy constructor and the constructor. The compiler is happy to just copy the values of the object passed to it to initialize the const and reference.

Default copy constructor call

OK, before we leave this section, did something about the previous code feel odd to you? Did you notice how we are explicitly calling the copy constructor of the *baseClass* from *derivedClass*, but we haven't defined any copy constructor in *baseClass*? Yet the compiler had no complaints. Although when we did not call any *baseClass* constructors, the compiler was complaining about no default constructor. What's going on here with the compiler? Well, this is just how it is. When we do not explicitly call the copy constructor, the compiler will always attempt to call the default constructor. So now we have established the fact that unless told otherwise, the compiler will always try to call the default constructor. But a copy constructor is actually generated by the compiler. Since we did not define a copy constructor, the compiler did that for us. A copy constructor that does bitwise copy. But the compiler just does not bother to call it. If we don't ask for the copy constructor (in the initializer list), it will just call the default constructor. But what happens when we call the copy constructor explicitly is the compiler will call its generated copy constructor. We need to tell the compiler to use the one it generated. But the same rules apply when we define the copy constructor. If we define it, we need to initialize the const and the reference. Otherwise if we just do this:

```
class baseClass
{
public:
        int var;
        int* varPtr;
        const float floatVar;
        int &intRefVar;

        baseClass(int value) : var(value), varPtr(&var), floatVar(value), intRefVar(var)
        {
                cout << “baseClass constructor” << endl;
        }

        baseClass(const baseClass& objToCopy) : var(objToCopy.var), varPtr(&var)
        {
            cout << “derivedClass copy constructor” << endl;
        }
};
```

we get our all too familiar error:



```
Description
2   error C2758: 'baseClass::floatVar' : a member of reference type must be initialized
3   error C2758: 'baseClass::intRefVar' : a member of reference type must be initialized
```

Now we are on to the last part. First we talked about copying when we have a class type member. Then we discussed about copying a class hierarchy. Now let’s see how to handle when we have a member class which has a hierarchy.

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

class baseClass
{
public:
        int var;
        int* varPtr;
        const float floatVar;
        int &intRefVar;

        baseClass(int value) : var(value), varPtr(&var), floatVar(value), intRefVar(var)
        {
            cout << “baseClass constructor” << endl;
        }

        baseClass(const baseClass& objToCopy) : var(objToCopy.var), varPtr(&var), floatVar(objToCopy.floatVar),
        intRefVar(var)
        {
            cout << “baseClass copy constructor” << endl;
        }
};

class derivedClass : public baseClass
{
public:
        derivedClass(int value) : baseClass(value)
        {
            cout << “derivedClass constructor” << endl;
        }

        derivedClass(const derivedClass& objToCopy) : baseClass(objToCopy)
        {
                cout << “derivedClass copy constructor” << endl;
        }
};

class theOtherClass
{
public:
        derivedClass derivedClassObj;

        theOtherClass(int value) : derivedClassObj(value)
        {
                cout << “theOtherClass constructor” << endl;
        }
};

int main(int argc, char** argv)
{
        cout << “–- Object construction –-” << endl;
        theOtherClass theOtherClassObj1(5);
        cout << endl;
        cout << “–- Object copy –-” << endl;
        theOtherClass theOtherClassObj2(theOtherClassObj1);
        return 0;
}
```

```
–- Object construction –-
baseClass constructor
derivedClass constructor
theOtherClass constructor

–- Object copy –-
baseClass copy constructor
derivedClass copy constructor
```

You can clearly see in the output the construction of the object and the copying of the object. When we copied the object the compiler properly called the copy constructors of *baseClass* and *derivedClass*. So you see, when we have the copy constructors properly defined, we don't need to take any extra care when we have a member object with a hierarchy. Note that we didn't define any copy constructor for *theOtherClass*. The compiler generated one for us and it correctly called the copy constructor defined in *derivedClass*. It did not do a shallow copy. What if *derivedClass* didn't have a copy constructor defined?

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

class baseClass
{
public:
        int var;
        int* varPtr;
        const float floatVar;
        int &intRefVar;

        baseClass(int value) : var(value), varPtr(&var), floatVar(value), intRefVar(var)
        {
                cout << "baseClass constructor" << endl;
        }
};

class derivedClass : public baseClass
{
public:
        derivedClass(int value) : baseClass(value)
        {
                cout << "derivedClass constructor" << endl;
        }
};


class theOtherClass
{
public:
        derivedClass derivedClassObj;

        theOtherClass(int value) : derivedClassObj(value)
        {
                cout << "theOtherClass constructor" << endl;
        }
};

int main(int argc, char** argv)
{
        cout << "–- Object construction –-" << endl;
        theOtherClass theOtherClassObj1(5);

        cout << endl;
        cout << "–- Object copy –-" << endl;
        theOtherClass theOtherClassObj2(theOtherClassObj1);

        cout << endl;
        cout << "–- Print varPtr –-" << endl;
        cout << "theOtherClassObj1.derivedClassObj.varPtr: " << theOtherClassObj1.derivedClassObj.varPtr << endl;
        cout << "theOtherClassObj2.derivedClassObj.varPtr: " << theOtherClassObj2.derivedClassObj.varPtr << endl;

        return 0;
}
```

```
–- Object construction –-
baseClass constructor
derivedClass constructor
theOtherClass constructor

–- Object copy –-

–- Print varPtr –-
theOtherClassObj1.derivedClassObj.varPtr: 003AF740
theOtherClassObj2.derivedClassObj.varPtr: 003AF740
```

You see, the rules don't change. The compiler generated copy constructor sees that *derivedClass* has not provided any copy constructor, so it does a shallow copy. We see that from the two *varPtr*'s. They are the same.

We looked at a lot of different scenarios. Let's summarize the compiler copy mechanism.

- If a copy constructor is not defined the compiler will generate one. This copy constructor will do a bit-wise copy (we will see when this is not the case).
- If a copy constructor is explicitly defined the compiler will not generate any type of constructor. Not even the default constructor.

- But if a constructor is defined, but not a copy constructor, the compiler will generate a copy constructor.
- If a copy constructor is defined, the member class or base class copy constructor must be called in the initializer list.
- If the copy constructor does not call the member class or base class copy constructor in the initializer list, the compiler will try to call the default constructor of the member class or base class.
- If a copy constructor is defined and calls the member class or base class copy constructor in the initializer list, the compiler will call the copy constructor. In this case, if a copy constructor is not defined in the member class or base class, the compiler generated copy constructor will be called.
- The compiler generated copy constructor will always call the copy constructor of the member class or base class.
- Remember, a copy constructor is always generated by the compiler (if not defined). Always. But whether the compiler calls the copy constructor depends on how and if it is called.

So I assume that list made some sense. There aren't a lot of rules. The easiest way to deal with this is to implement your own copy constructor.

Object cloning

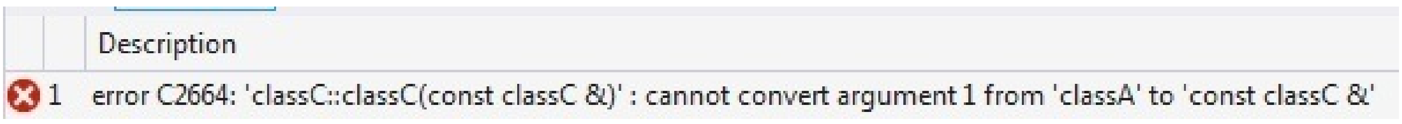
So far we saw how to copy objects. Whether they are derived classes or member classes. But the situation is a bit complicated when we are dealing with pointers. Look at this class hierarchy.

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

class classA
{
public:
        int classAVar;

        classA(int value) : classAVar(value)
        {
                cout << “classAVar constructor” << endl;
        }
};

class classB : public classA
{
public:
        int classBVar;
        classB(int value1, int value2) : classA(value1), classBVar(value2)
        {
                cout << “classB constructor” << endl;
        }
};

class classC : public classB
{
public:
        int classCVar;

        classC(int value1, int value2, int value3) : classB(value1, value2), classCVar(value3)
        {
                cout << “classC constructor” << endl;
        }
};


int main(int argc, char** argv)
{
        classA* classAptr1 = new classC(1, 2, 3);
        return 0;
}
```

No copy constructor is defined for these classes. That is fine. The compiler generated ones do the job. Note how our pointer is of type *classA* but it has a *classC* object. This is usually what you would see when handling pointers and class hierarchies. Pointers let us use polymorphism. Now the problem is, how can we copy *classAptr* to another pointer? You could try this:

```
int main(int argc, char** argv)
{
        classA* classAptr1 = new classC(1, 2, 3);
        classA* classAptr2 = new classC(*classAptr1);
        return 0;
}
```

But this wont work.



```
Description
1    error C2664: 'classC::classC(const classC &)' : cannot convert argument 1 from 'classA' to 'const classC &'
```

The compiler cannot convert a *classA* type to a *classC* reference. The only way we can make the compiler happy is if we instantiate a new *classA* type. Let’s do that and see what happens.

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

class classA
{
public:
        int classAVar;

        classA(int value) : classAVar(value)
        {}

        virtual void printVars()
        {
                cout << “classAVar: “ << classAVar << endl;
        }
};

class classB : public classA
{
public:
        int classBVar;

        classB(int value1, int value2) : classA(value1), classBVar(value2)
        {}

        virtual void printVars()
        {
                cout << “classAVar: “ << classAVar << endl;
                cout << “classBVar: “ << classBVar << endl;
        }
};

class classC : public classB
{
public:
        int classCVar;
```

```
	classC(int value1, int value2, int value3) : classB(value1, value2), classCVar(value3)
	{}

	virtual void printVars()
	{
		cout << “classAVar: “ << classAVar << endl;
		cout << “classBVar: “ << classBVar << endl;
		cout << “classCVar: “ << classCVar << endl;
	}
};

int main(int argc, char** argv)
{
	classA* classAptr1 = new classC(1, 2, 3);
	cout << “classAptr1->printVars()” << endl;
	classAptr1->printVars(); cout << endl;

	classA* classAptr2 = new classA(*classAptr1);
	cout << “classAptr2->printVars()” << endl;
	classAptr2->printVars();
	return 0;
}
```

```
classAptr1->printVars()
classAVar: 1
classBVar: 2
classCVar: 3

classAptr2->printVars()
classAVar: 1
```

Obviously this is not what we want. *classAptr2* is just a *classA* object. The called copy constructor is of *classA* and the copy constructor only receives the *classA* part of the object. So it only can copy that part. How are we going to fix this? For starters, we need something other than copy constructors. Because we now need to ‘clone’ the object.

So we now understand that we need a new way to do the copying, or the cloning. What would this method need to do? Let’s call this function *clone*.

This is what *clone* needs to be:

- This function needs to support a cloning of a hierarchy of a class, so it needs to be a virtual method.
- This function should return a pointer to a new cloned object.
- This function needs to copy its values to the new cloned object.

This is how simple this function is:

```
        virtual classType* clone()
        {
                return new classType(*this);
        }
```

Think about it. All we need to do is to return a new object that is a copy of itself. And what function does that? Yes, the copy constructor. So what we are doing here is calling the copy constructor to create a new object and returning it. The trick is we need to implement this in each derived class.

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

class classA
{
public:
        int classAVar;

        classA(int value) : classAVar(value)
        {}

        virtual void printVars()
        {
                cout << “classAVar: “ << classAVar << endl;
        }

        virtual classA* clone()
        {
                return new classA(*this);
        }
};

class classB : public classA
{
public:
        int classBVar;

        classB(int value1, int value2) : classA(value1), classBVar(value2)
        {}

        virtual void printVars()
        {
                cout << “classAVar: “ << classAVar << endl;
                cout << “classBVar: “ << classBVar << endl;
        }

        virtual classB* clone()
        {
                return new classB(*this);
        }
};

class classC : public classB
{
public:
        int classCVar;

        classC(int value1, int value2, int value3) : classB(value1, value2), classCVar(value3)
        {}

        virtual void printVars()
        {
                cout << “classAVar: “ << classAVar << endl;
                cout << “classBVar: “ << classBVar << endl;
                cout << “classCVar: “ << classCVar << endl;
        }

        virtual classC* clone()
        {
                return new classC(*this);
        }
};


int main(int argc, char** argv)
{
        classA* classAptr1 = new classC(1, 2, 3);
```

```
        cout << “classAptr1->printVars()” << endl;
        classAptr1->printVars(); cout << endl;

        classA* classAptr2 = classAptr1->clone();
        cout << “classAptr2->printVars()” << endl;
        classAptr2->printVars();
        return 0;
}
```

```
classAptr1->printVars()
classAVar: 1
classBVar: 2
classCVar: 3

classAptr2->printVars()
classAVar: 1
classBVar: 2
classCVar: 3
```

Because *clone* is virtual, when called through the pointer, it calls the clone of the correct object type. And it calls the copy constructor of itself, which copies the entire object, including the base classes. Remember we discussed how copy constructors of derived classes work? They always need to copy the base class part. By the way, in case you are wondering about the difference in a clone’s return type across classes, it is allowed as long as the they are of the same hierarchy.

This cloning mechanism is called *‘virtual copy pattern’* and it is an example of deep copying.

Trivial/ non-trivial copy constructors

Before we leave this section I need to mention about the trivial and non-trivial copy constructors. The compiler generated copy constructors we saw for the most part of this discussion are ‘trivial’ copy constructors. They don’t do much more than a member-wise copy. But the compiler can’t always do this. Especially in a program like the last one where we had the virtual clone function. There are a few other cases when a copy constructor cannot be trivial and having a virtual function is one of the main reasons.

You see, when a class has a virtual function, it has a virtual table pointer (vptr) and an associated virtual table (*vtable*). We discuss this on the ‘virtual mechanism’ topic. And when there is an associated *vptr* for the object, it cannot simply be copied to the newly copy constructed class. The *vtable* and *vptr* need to be generated for each object depending on its type. Now you might wonder why the *vptr* cannot be the same as the copying object? Well, if the object being copied and object being constructed are both of the same inheritance level, then you can. But this is not always the case. For example, in our last example, a *classB* copy constructor has the following form:

```
class classB : public classA
{
        …
        classB(const classB& objToCopy)
        {
                // copy object
        }
};
```

We need to consider two possibilities here.

First one, both objects are *classB*:

```
int main(int argc, char** argv)
{
        classB classBObj1 = classBObj(1, 2);
        classB classBObj2(classBObj1);
        return 0;
}
```

In this case, yes, you can have the same *vptr* and *vtable* for both the objects. They are of same type.

But what about this case?

```
int main(int argc, char** argv)
{
        classC classCObj = classCObj(1, 2, 3);
        classB classBObj(classCObj);
        return 0;
}
```

Pass the *objToCopy* as a reference; it gets “sliced” to a *classB*. So a direct bit-wise copy will not work at all. This is why when there is a virtual function the copy constructor is non-trivial. It needs to do more work than just simply copying bits. Having a virtual function is just one case. There are a few other cases that require a non-trivial copy constructor.

One last bit. Although I talked about *vptr*s and *vtable*s, I need to tell you that this concept of *vptr*s and *vtable*s are not specified in the standard. The compiler is free to implement the virtual mechanism in any way it wants (but *vptr*s and *vtable*s are the most used way). So having a *vptr* is not a definite reason for a non-trivial copy constructor, although in most compilers it would be.

So if the class needs a non-trivial copy constructor, the compiler will generate that for you. If you have defined a copy constructor, the compiler will add the non-trivial part before the copy constructor body, such that properties like *vptr*s are correctly defined so you don’t need to worry about it.

Topic 11

Class Member Access

We will discuss a minor topic here. It is not complex but could be a little confusing at times so it is better to look at some examples.

Let’s get things going with an example.

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

class baseClass
{
public:
        int baseClassVar;
        int commonVar;

        baseClass(int baseVal, int comVal) : baseClassVar(baseVal), commonVar(comVal)
        {}

        void printVals()
        {
                cout << “baseClass::printVals()—>” << endl;
                cout << “baseClass::baseClassVar- “ << baseClassVar << endl;
                cout << “baseClass::commonVar- “ << commonVar << endl;
        }
};


class derivedClass : public baseClass
{
public:
        int derivedClassVar;
        int commonVar;

        derivedClass(int baseVal, int baseComVal, int derivedVal, int derivedComVal) : baseClass(baseVal, baseComVal),
        derivedClassVar(derivedVal), commonVar(derivedComVal)
        {}

        void printVals()
        {
```

```
        cout << “derivedClass::printVals()” << endl;
        baseClass::printVals();
        cout << “derivedClass::derivedClassVar- “ << derivedClassVar << endl;
        cout << “derivedClass::commonVar- “ << commonVar << endl << endl;
    }

    void printAllVals()
    {
        cout << “derivedClass::printAllVals()” << endl;
        cout << “baseClass::baseClassVar- “ << baseClassVar << endl;
        cout << “baseClass::commonVar- “ << baseClass::commonVar << endl;
        cout << “derivedClass::derivedClassVar- “ << derivedClassVar << endl;
        cout << “derivedClass::commonVar- “ << commonVar << endl;
    }
};


int main(int argc, char** argv)
{
    derivedClass derivedClassObj(1, 2, 3, 4);
    derivedClassObj.printVals();
    derivedClassObj.printAllVals();
    return 0;
}
```

```
derivedClass::printVals()—>
baseClass::printVals
baseClass::baseClassVar- 1
baseClass::commonVar- 2
derivedClass::derivedClassVar- 3
derivedClass::commonVar- 4

derivedClass::printAllVals()—>
baseClass::baseClassVar- 1
baseClass::commonVar- 2
derivedClass::derivedClassVar- 3
derivedClass::commonVar- 4
```

Here we have a base class and a derived class. Each class has its own unique integer variable and another integer variable which has the same name. Each also have their own non-virtual *printVals* function. These are the take aways:

- It is possible for the derived class to have its own member variable with the same name as a base class member. When accessed, there is no ambiguity. The compiler chooses the one in the enclosing class.
- The derived class can access the base class member variables and functions by specifying the base class name with the scope resolution operator.

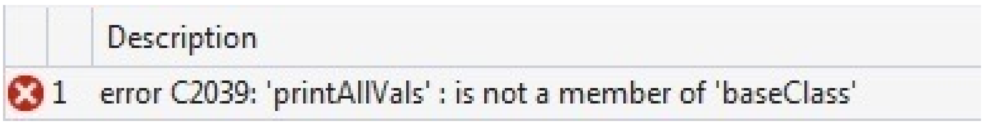
Pretty straightforward. Nothing complex here. Now let's get the pointers in!

```
…
…
int main(int argc, char** argv)
{
        baseClass* baseClassPtr = new baseClass(1, 2);
        baseClassPtr->printVals(); cout << endl;

        derivedClass* derivedClassPtr = new derivedClass(3, 4, 5, 6);
        derivedClassPtr->printVals();
        return 0;
}
```

```
baseClass::printVals()—>
baseClass::baseClassVar- 1
baseClass::commonVar- 2

derivedClass::printVals()—>
baseClass::printVals()—>
baseClass::baseClassVar- 3
baseClass::commonVar- 4
derivedClass::derivedClassVar- 5
derivedClass::commonVar- 6
```

Again, nothing special here. We have a *baseClass* pointer with a *baseClass* instance and a *derivedClass* pointer with a *derivedClass* instance. Nothing much interesting going on here. Now let's see how things behave when we have different pointer and instance types.

```
…
…
int main(int argc, char** argv)
{
    baseClass* baseClassPtr = new derivedClass(1, 2, 3, 4);
    baseClassPtr->printAllVals();
    return 0;
}
```



```
Description
1   error C2039: 'printAllVals' : is not a member of 'baseClass'
```

This shouldn't come as a surprise but it's good to validate this fact. It doesn't matter which type the instance is. The compiler does not care what type of object the pointer is pointing to. It must not. Because the *baseClassPtr* can point to different instances throughout the program. So the compiler does not attempt to look into the type of the instance. This is one of the principles of polymorphism and dynamic binding. For the compiler, it is pointing to a *baseClass* instance. And baseClass does not have a *printAllVals* function defined.

But we know it is pointing to a *derivedClass* instance. So we can cast it like this:

```
…
…
int main(int argc, char** argv)
{
    baseClass* baseClassPtr = new derivedClass(1, 2, 3, 4);
    static_cast<derivedClass*>(baseClassPtr)->printAllVals();
    return 0;
}
```

```
derivedClass::printAllVals()—>
baseClass::baseClassVar- 1
baseClass::commonVar- 2
derivedClass::derivedClassVar- 3
derivedClass::commonVar- 4
```

Here we are casting the *baseClassPtr* to a *derivedClass* type. We are telling the compiler to take our word that *baseClassPtr* is indeed pointing to a *derivedClass* instance. And the compiler obliges. You see, there is no difference between a pointer to a *derivedClass* and a pointer to a plain old integer. They are both memory addresses with the same amount of bits. The pointer contains just a memory address. What is in this memory address is determined by the pointer type. So here we are telling the compiler that *baseClassPtr* is pointing to a memory block that has a *derivedClass* object. And indeed it does, so we have no issues. This will become clear when we do this:

```
…
…
int main(int argc, char** argv)
{
        baseClass* baseClassPtr = new baseClass(1, 2);
        static_cast<derivedClass*>(baseClassPtr)->printAllVals();
        return 0;
}
```

```
derivedClass::printAllVals()—>
baseClass::baseClassVar- 1
baseClass::commonVar- 2
derivedClass::derivedClassVar- 1359395851
derivedClass::commonVar- -2013265804
```

Here we only have a *baseClass* instance but we are lying to the compiler that it is pointing to a *derivedClass* object. The compiler does no validation. It takes our word and tries to read the member variables and gets garbage because those variables don't exist.

Now let's see an example of accessing member variables through pointers.

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

class baseClass
{
public:
        int commonVar;

        baseClass(int comVal) : commonVar(comVal)
        {}
};

class derivedClass : public baseClass
{
public:
        int commonVar;

        derivedClass(int baseComVal, int derivedComVal) : baseClass(baseComVal), commonVar(derivedComVal)
        {}
};
```

```
int main(int argc, char** argv)
{
        baseClass* baseClassPtr = new derivedClass(1, 2);
        cout << "baseClassPtr->commonVar: " << baseClassPtr->commonVar << endl;

        derivedClass* derivedClassPtr = new derivedClass(3, 4);
        cout << "derivedClassPtr->commonVar: " << derivedClassPtr->commonVar << endl;
        cout << "derivedClassPtr->baseClass::commonVar: " << derivedClassPtr->baseClass::commonVar << endl;
        return 0;
}
```

```
baseClassPtr->commonVar: 1
derivedClassPtr->commonVar: 4
derivedClassPtr->baseClass::commonVar: 3
```

The accessed variable, then, depends on the type of the pointer, not the type of the object it points to. This is because member variables are accessed through offsets. You will learn more on this in the next chapter. The offset value for *baseClass::commonVar* is different from *derivedClass::commonVar*. In fact, we can print them out and see.

```
…
….
int main(int argc, char** argv)
{
        cout << offsetof(baseClass, commonVar) << endl;
        cout << offsetof(derivedClass, commonVar) << endl;
        return 0;
}
```

```
0
4
```

As you can see, the offset values for *commonVar* are different. That is how the compiler is able to access the correct variable depending on the type.

There are no suprises here. But I wanted to show you the access of member variables and functions in the class hierarchy when the same name is shared and how it works with object types and pointer types.

Topic 12

Class member offsets

In this short topic we will discuss class member offsets. When you define member variables in a class they need to be located at certain places within the class. What I am going to discuss here is not part of the standard. As with virtual mechanism, the compiler developers are free to use their own methods.

Member placement within an object

Class member offsets define the offset, usually in bytes, that a particular member has within the object. When you create an object of a class, it occupies a block of memory and the compiler knows where this memory block starts. And it knows how long the block spans, which can be found by using the *sizeof(classType)*. But how does the compiler find a member variable within that object? It is through the offset. Here’s an example.

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

class baseClass
{
public:
        int var1;
        int var2;
        int var3;

        baseClass(int val1, int val2, int val3) : var1(val1), var2(val2), var3(val3)
        {}
};

int main(int argc, char** argv)
{
        cout << “Offset of baseClass::var1: “ << offsetof(baseClass, var1) << endl;
        cout << “Offset of baseClass::var2: “ << offsetof(baseClass, var2) << endl;
        cout << “Offset of baseClass::var3: “ << offsetof(baseClass, var3) << endl;
        return 0;
}
```

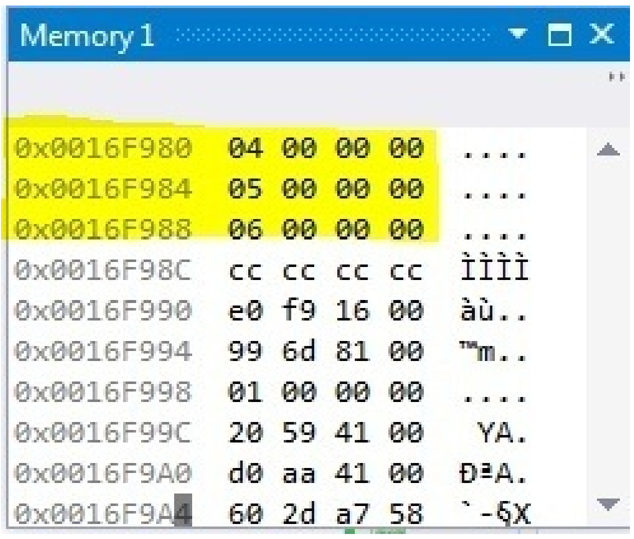
```
Offset of baseClass::var1: 0
Offset of baseClass::var2: 4
Offset of baseClass::var3: 8
```

We have a simple class with three integer variables. And note what we are printing out. We are looking at the offset. This *offsetof* macro gives the offset the variable has from the beginning of the object. Note how we're not making any objects. Just the class itself. So we see that our first variable, *var1*, is at an offset 0. This means *var1* is at the start of the object memory space. And *var2* after 4 bytes and *var3* after 8 bytes. This makes sense. An integer is 4 bytes long. So these variables are offset one after the other. Let's really make sure this is indeed the case by looking at the memory.

```
int main(int argc, char** argv)
{
        cout << "Offset of baseClass::var1: " << offsetof(baseClass, var1) << endl;
        cout << "Offset of baseClass::var2: " << offsetof(baseClass, var2) << endl;
        cout << "Offset of baseClass::var3: " << offsetof(baseClass, var3) << endl;

        baseClass baseClassObj(4, 5, 6);
        cout << "&baseClassObj: " << &baseClassObj << endl;
        cout << "&baseClassObj.var1: " << &baseClassObj.var1 << endl;
        cout << "&baseClassObj.var2: " << &baseClassObj.var2 << endl;
        cout << "&baseClassObj.var3: " << &baseClassObj.var3 << endl;
        return 0;
}
```

```
Offset of baseClass::var1: 0
Offset of baseClass::var2: 4
Offset of baseClass::var3: 8

&baseClassObj: 0016F980
&baseClassObj.var1: 0016F980
&baseClassObj.var2: 0016F984
&baseClassObj.var3: 0016F988
```

Here we create an object and print out the actual memory addresses of the three variables. And you can clearly see from the memory addresses that the variables are laid out 4 bytes apart in memory. Here's a look at the memory space.



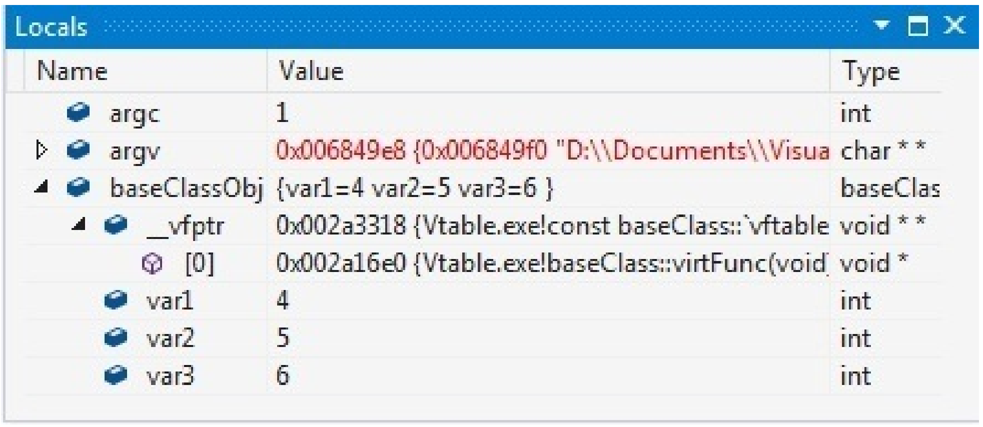
```
0x0016F980  04 00 00 00  ....
0x0016F984  05 00 00 00  ....
0x0016F988  06 00 00 00  ....
0x0016F98C  cc cc cc cc  ÌÌÌÌ
0x0016F990  e0 f9 16 00  àù..
0x0016F994  99 6d 81 00  ™m..
0x0016F998  01 00 00 00  ....
0x0016F99C  20 59 41 00   YA.
0x0016F9A0  d0 aa 41 00  ÐªA.
0x0016F9A4  60 2d a7 58  `-§X
```

So this is how the compiler is able to find the member variables within a class. Each object of the same type is laid with the same offsets. Now let's look at a case where this offset changes. Let's define a simple virtual function. This makes the class objects to include a *vptr* and a corresponding *vtable*.

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

class baseClass
{
public:
        int var1;
        int var2;
        int var3;

        baseClass(int val1, int val2, int val3) : var1(val1), var2(val2), var3(val3)
        {}

        virtual void virtFunc()
        {
                cout << "virtual function" << endl;
        }
};


int main(int argc, char** argv)
{
        baseClass baseClassObj(4, 5, 6);

        cout << "&baseClass::var1: " << offsetof(baseClass, var1) << endl;
        cout << "&baseClass::var2: " << offsetof(baseClass, var2) << endl;
        cout << "&baseClass::var3: " << offsetof(baseClass, var3) << endl << endl;

        cout << "&baseClassObj: " << &baseClassObj << endl;
        cout << "&baseClassObj.var1: " << &baseClassObj.var1 << endl;
        cout << "&baseClassObj.var2: " << &baseClassObj.var2 << endl;
        cout << "&baseClassObj.var3: " << &baseClassObj.var3 << endl;
        return 0;
}
```

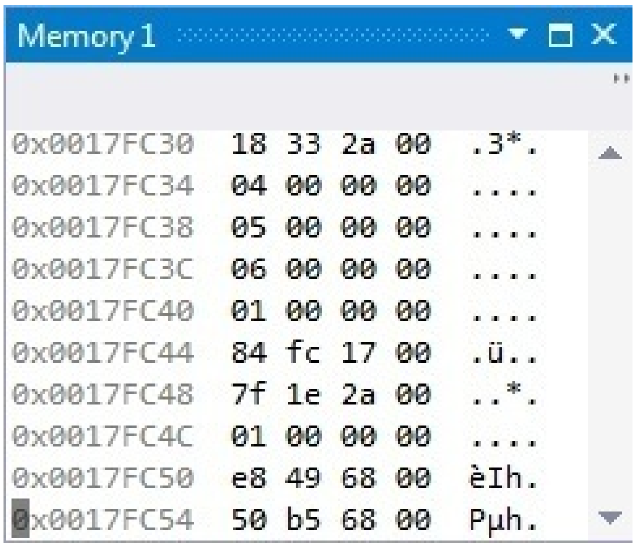
```
offsetof(baseClass, var1): 4
offsetof(baseClass, var2): 8
offsetof(baseClass, var3): 12

&baseClassObj: 0017FC30
&baseClassObj.var1: 0017FC34
&baseClassObj.var2: 0017FC38
&baseClassObj.var3: 0017FC3C
```

Did you see how the offsets of the variables changed by 4 bytes when we add the virtual function? Because the compiler has put the *vptr* at offset 0. *vptr* is like any other pointer and it is taking 4 bytes, hence the variables are shifted by 4 bytes. It is important to keep in mind that this is not the standard to put the *vptr* at the beginning of the class, or even to use a *vptr*. This is just how the Visual C++ compiler is doing it. But this insight gives us a peek at how the compiler is handling this member access. Let's look at the object locals and the corresponding memory block.



```
Name                Value                                          Type
   argc             1                                              int
 ▷ argv             0x006849e8 {0x006849f0 "D:\\Documents\\Visua   char * *
 ◢ baseClassObj     {var1=4 var2=5 var3=6 }                        baseClas
   ◢ __vfptr        0x002a3318 {Vtable.exe!const baseClass::`vftable void * *
        [0]         0x002a16e0 {Vtable.exe!baseClass::virtFunc(void void *
     var1           4                                              int
     var2           5                                              int
     var3           6                                              int
```

The object locals show the variable values and also the *vptr* contents. It shows that the *vptr* is pointing to memory address '0x002A3318'. This means the *vtable* is at this address. So how are we sure that the variables got offset by 4 bytes because of *vptr*? Let's look at the memory block of the object at 0x0017FC30:



```
0x0017FC30  18 33 2a 00  .3*.
0x0017FC34  04 00 00 00  ....
0x0017FC38  05 00 00 00  ....
0x0017FC3C  06 00 00 00  ....
0x0017FC40  01 00 00 00  ....
0x0017FC44  84 fc 17 00  .ü..
0x0017FC48  7f 1e 2a 00  ..*.
0x0017FC4C  01 00 00 00  ....
0x0017FC50  e8 49 68 00  èIh.
0x0017FC54  50 b5 68 00  Pµh.
```

See the contents at the offset 0 of the object start address (which we printed out earlier)? It is the same as the address of the *vptr* that we saw in the locals view. This confirms that the (Visual C++) compiler puts the *vptr* at the zero offset of the object.

I hope you understood a little bit about how the compiler is accessing objects. Although the implementation is compiler dependent, having memebr offsets are an efficient way of accessing objects and their contents.

Pointer to class member

OK, so we know how the compiler is doing things, but is there any benefit we can take from this?

```
int main(int argc, char** argv)
{
        baseClass baseClassObj(4, 5, 6);
        cout << "baseClassObj.var1: " << baseClassObj.var1 << endl;

        int baseClass::*intVarPtr;                    // Line 1
        intVarPtr = &baseClass::var1;                 // Line 2
        baseClassObj.*intVarPtr = 10;                 // Line 3
        cout << "baseClassObj.var1: " << baseClassObj.var1 << endl;
        return 0;
}
```

```
baseClassObj.var1: 4
baseClassObj.var1: 10
```

Let's quickly go through what we are doing here:

- In line 1 we are creating a pointer *intVarPtr* that will point to an integer variable of class *baseClass*. This pointer can point to any integer of *baseClass*.

- In line 2 we assign *intVarPtr* to point to *var1* of *baseClass*. Note how we are taking the address of a *baseClass*, not the *baseClassObj* instance. When we use 'address of' operator on a class member, unless that member is static, it gives the offset of that variable (If the member is static then the actual memory address is obtained). Now *intVarPtr* is locked to point to *var1* of *baseClass* instances only.
- In line 3 we are using *intVarPtr* to dereference and access *var1* member of instance *baseClassObj*.

What this means is that you can have a single pointer that is able to point to a particular member of a class. Can you see how this can be useful?

```
int main(int argc, char** argv)
{
        baseClass baseClassObj1(1, 2, 3);
        baseClass baseClassObj2(4, 5, 6);
        cout << "baseClassObj1.var1: " << baseClassObj1.var1 << endl;
        cout << "baseClassObj2.var1: " << baseClassObj2.var1 << endl << endl;

        int baseClass::*intVarPtr;
        intVarPtr = &baseClass::var1;

        baseClassObj1.*intVarPtr = 10;
        cout << "baseClassObj1.var1: " << baseClassObj1.*intVarPtr << endl;
        baseClassObj2.*intVarPtr = 20;
        cout << "baseClassObj2.var1: " << baseClassObj2.*intVarPtr << endl;
        return 0;
}
```

```
baseClassObj1.var1: 1
baseClassObj2.var1: 4

baseClassObj1.var1: 10
baseClassObj2.var1: 20
```

See how we can have just one pointer to access a particular variable of any object? This is another very convenient way of utilizing polymorphism. Look at this example.

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

```
class baseClass
{
public:
        int var1;
        int var2;
        int var3;

        baseClass(int val1, int val2, int val3) : var1(val1), var2(val2), var3(val3)
        {}

        virtual void virtFunc()
        {
                cout << “virtual function” << endl;
        }
};

class derivedClass : public baseClass
{
public:
        int var4;

        derivedClass(int val1, int val2, int val3, int val4) : baseClass(val1, val2, val3), var4(val4)
        {}
};

void memberVarPrintFunc(baseClass object, int baseClass::*varPtr)
{
        cout << object.*varPtr << endl;
}

int main(int argc, char** argv)
{
        baseClass baseClassObj(1, 2, 3);
        derivedClass derivedClassObj(4, 5, 6, 7);

        int baseClass::*intVarPtr;

        intVarPtr = &baseClass::var1;
        cout << “baseClassObj.var1: “;
        memberVarPrintFunc(baseClassObj, intVarPtr);
        cout << “derivedClassObj.var1: “;
        memberVarPrintFunc(derivedClassObj, intVarPtr);
```

```
    intVarPtr = &baseClass::var3;
    cout << “baseClassObj.var3: “;
    memberVarPrintFunc(baseClassObj, intVarPtr);
    cout << “derivedClassObj.var3: “;
    memberVarPrintFunc(derivedClassObj, intVarPtr);
    return 0;
}
```

```
baseClassObj.var1: 1
derivedClassObj.var1: 4
baseClassObj.var3: 3
derivedClassObj.var3: 6
```

For this example I introduced a derived class. This is just to demonstrate to you that you can use this concept with inheritance and how to make use of it. The function *memberVarPrintFunc* takes as argument an object of *baseClass* and an integer pointer to a member of *baseClass*. Now see how conveniently we can utilize *intVarPtr* to access any int variable of *baseClass* or any derived class of it? We have one pointer and we can make it point to any integer member of *baseClass*. And *memberVarPrintFunc* does not need to know anything about the *baseClass* variables. You can change the variable names in *baseClass* and it wouldn’t have any effect on the function. You can make the function access any of the integer members. See how beneficial this offset mechanism can be?

Topic 13

Function Pointers

This topic will discuss the mechanism of function pointers. They are not much different than a normal pointer but the syntax can be a little confusing. Function pointer syntax can be extremely confusing and complex if you want to make it be so, but I'm only going to discuss the basics of it which will help you in deciphering complex notation if you ever need to.

Let's start with a simple example.

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

void simpleFunc(int val)
{
        cout << "simpleFunc with val:" << val << endl;
}

int main(int argc, char** argv)
{
        void(*funcPtr)(int) = &simpleFunc;
        (*funcPtr)(10);
        return 0;
}
```

```
simpleFunc with val:10
```

What are we doing here?

First we define a function *simpleFunc*, which takes an integer parameter and returns void. Then we create our function pointer *funcPtr* which we want to refer to *simpleFunc*. The thing with function pointers is that they cannot be general. You cannot have one function pointer that can point to a function with any signature. The return type and argument types need to match. So we need to make *funcPtr* return void and take an integer argument.

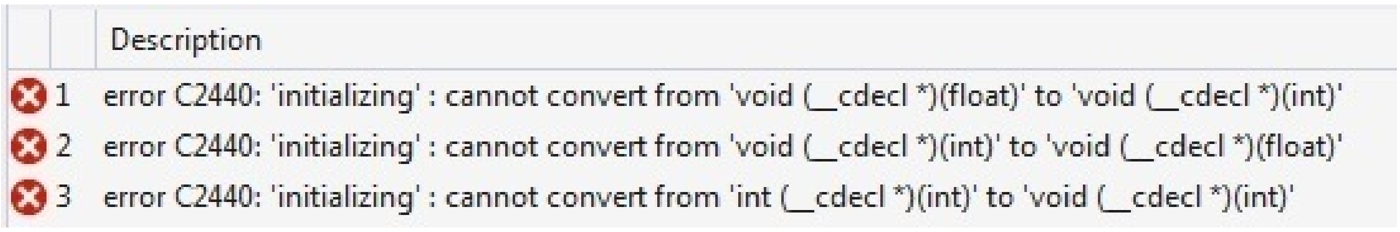
Look at the syntax. The parantheses are important. *(*funcPtr)* declares a pointer to a function. The parantheses to the right of it declares the arguments, which in our case is just an integer. To the left is the return type. That's it. Not that complicated right? But remember that the parentheses are important. Note the difference between these two statements:

```
        void(*funcPtr)(int);
        void *funcPtr(int);
```

The first statement is a declaration for a function pointer that takes in an integer parameter and returns void. The second is a function prototype that takes an integer argument and returns a *void**. So be mindful of the parentheses.

Now let’s look at a few different caveats of using function pointers.

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

void intArgFunc(int val)
{
        cout << “intArgFunc with val:” << val << endl;
}

void floatArgFunc(float val)
{
        cout << “floatArgFunc with val:” << val << endl;
}

int intReturnFunc(int val)
{
        cout << “Returning val:” << val << endl;
        return val;
}


int main(int argc, char** argv)
{
        void(*intArgFuncPtr1)(int) = &intArgFunc;                    // Line 1
        void(*intArgFuncPtr2)(int) = &floatArgFunc;                  // Line 2
        void(*floatArgFuncPtr1)(float) = &intArgFunc;                // Line 3
        void(*floatArgFuncPtr2)(float) = &floatArgFunc;              // Line 4
        void(*intRetFuncPtr1)(int) = &intReturnFunc;                 // Line 5
        int(*intRetFuncPtr2)(int) = &intReturnFunc;                  // Line 6
        return 0;
}
```



```
Description
1  error C2440: 'initializing' : cannot convert from 'void (__cdecl *)(float)' to 'void (__cdecl *)(int)'
2  error C2440: 'initializing' : cannot convert from 'void (__cdecl *)(int)' to 'void (__cdecl *)(float)'
3  error C2440: 'initializing' : cannot convert from 'int (__cdecl *)(int)' to 'void (__cdecl *)(int)'
```

We have compiler errors in lines 2, 3 and 5. So we see that the function parameter types and return types need to match. There is no argument promotion possible here. We can call *floatArgFunc* with an integer argument and the compiler is fine with that. But not when we are dealing with function pointers. The function signature needs to match in this case.

A minor piece of information about using function pointers: you do not need to use the address of operator to assign the function to the pointer, nor do you need '*' to dereference it. In the case of function pointers the compiler is happy to do that for you. So you can write the first example as:

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

void simpleFunc(int val)
{
    cout << "simpleFunc with val:" << val << endl;
}

int main(int argc, char** argv)
{
    void(*funcPtr)(int) = simpleFunc;
    funcPtr(10);
    return 0;
}
```

The compiler can figure out for itself what you want to do.

Member function pointers

Things aren't much different if you want a function pointer to a class member function.

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

class simpleClass
{
public:
    void simpleClassFunc1()
    {
        cout << "simpleClassFunc1" << endl;
```

```
	}

	void simpleClassFunc2()
	{
		cout << “simpleClassFunc2” << endl;
	}
};

int main(int argc, char** argv)
{
	simpleClass simpleClassObj;
	void(simpleClass::*simpleClassFuncPtr)() = &simpleClass::simpleClassFunc1;
	(simpleClassObj.*simpleClassFuncPtr)();

	simpleClassFuncPtr = &simpleClass::simpleClassFunc2;
	(simpleClassObj.*simpleClassFuncPtr)();
	return 0;
}
```

```
simpleClassFunc1
simpleClassFunc2
```

So you can see how it is possible to invoke class member functions through pointers and also how you can re-assign them to different functions (with the same signature, of course).

There are a number of uses of function pointers. The most obvious one is as a callback mechanism. Take a look at this example.

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

class baseClass
{
public:
	int objID;
	baseClass(int id) : objID(id)
	{}

	virtual void callbackFunc()
	{
```

```
                cout << “callbackFunc: baseClass: “ << objID << endl;
        }
};

class derivedClass : public baseClass
{
public:
        derivedClass(int id) : baseClass(id)
        {}

        virtual void callbackFunc()
        {
                cout << “callbackFunc: derivedClass: “ << objID << endl;
        }
};

void callbackFunc(baseClass *obj, void (baseClass::*callbackFuncPtr)())
{
        (obj->*callbackFuncPtr)();
}

int main(int argc, char** argv)
{
        baseClass* baseClassPtr = new baseClass(1);
        derivedClass* derivedClassPtr = new derivedClass(2);

        void(baseClass::*callbackFuncPtr)() = &baseClass::callbackFunc;

        callbackFunc(baseClassPtr, callbackFuncPtr);
        callbackFunc(derivedClassPtr, callbackFuncPtr);
        return 0;
}
```

```
callbackFunc: baseClass: 1
callbackFunc: derivedClass: 2
```

I included a derived class and a virtual function to show you that this mechanism works fine with inheritance and dynamic binding. Can you see how this concept of function pointers can be easily used for the callback mechanism? It is important that we used a pointer to *baseClass* in the *callbackFunc* argument. This is essential if we need the virtual functions to work. Dynamic binding only works with pointers (and references). What would happen if we didn’t use pointers?

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

class baseClass
{
public:
        int objID;
        baseClass(int id) : objID(id)
        {}

        virtual void callbackFunc()
        {
                cout << “callbackFunc: baseClass: “ << objID << endl;
        }
};

class derivedClass : public baseClass
{
public:
        derivedClass(int id) : baseClass(id)
        {}

        virtual void callbackFunc()
        {
                cout << “callbackFunc: derivedClass: “ << objID << endl;
        }
};

void callbackFunc(baseClass obj, void (baseClass::*callbackFuncPtr)())
{
        (obj.*callbackFuncPtr)();
}

int main(int argc, char** argv)
{
        baseClass baseClassObj(1);
        derivedClass derivedClassObj(2);

        void(baseClass::*callbackFuncPtr)() = &baseClass::callbackFunc;
```

```
    callbackFunc(baseClassObj, callbackFuncPtr);
    callbackFunc(derivedClassObj, callbackFuncPtr);
    return 0;
}
```

```
callbackFunc: baseClass: 1
callbackFunc: baseClass: 2
```

The derived class function will not be called. Why? Because when we pass a *derivedClass* object to a function that takes in a *baseClass* as pass by value, the *derivedClass* object is 'sliced' to a *baseClass* object. The *derivedClass* implementation is no longer in that object. That is why we must be careful to use pointers (or references) whenever we need dynamic binding.

Virtual pointer example

Let's finish this topic after discussing something we conveniently avoided explaining in the 'Virtual Mechanism' topic. Here's the program.

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

class baseClass1
{
public:
    void nonVirtualFunc1()
    {
        cout << "nonVirtualFunc1" << endl;
    }
    virtual void virtualNonOverriddenFunc1()
    {
        cout << "virtualNonOverriddenFunc1" << endl;
    }
    virtual void virtualOverriddenFunc1()
    {
        cout << "virtualOverriddenFunc1" << endl;
    }
};

class baseClass2
{
```

```
public:
        void nonVirtualFunc2()
        {
                cout << “nonVirtualFunc2” << endl;
        }
        virtual void virtualNonOverriddenFunc2()
        {
                cout << “virtualNonOverriddenFunc2” << endl;
        }
        virtual void virtualOverriddenFunc2()
        {
                cout << “virtualOverriddenFunc2” << endl;
        }
};

class derivedClass : public baseClass1, public baseClass2
{
public:
        virtual void virtualOverriddenFunc1()
        {
                cout << “virtualDerivedOverriddenFunc1” << endl;
        }
        virtual void derivedClassOnlyVirtualFunc()
        {
                cout << “derivedClassOnlyVirtualFunc” << endl;
        }
};

int main(int argc, char** argv)
{
        derivedClass derivedClassObj;
        derivedClass *dcPtr = new derivedClass;

        cout << “Invoking function through the object pointer…” << endl;
        dcPtr->virtualNonOverriddenFunc1();
        dcPtr->virtualOverriddenFunc1();
        dcPtr->derivedClassOnlyVirtualFunc();
        cout << endl;

        void(**vtPtr)() = *(void(***)())dcPtr;   //obtaining __vftable address

        cout << “Printing __vftable…” << endl;
        cout << "__vftable address: " << vtPtr << endl;
        cout << "__vftable[0] - " << *vtPtr << endl;
        cout << "__vftable[1] - " << *(vtPtr + 1) << endl;
        cout << "__vftable[0] - " << *(vtPtr + 2) << endl;
        cout << endl;

        typedef void func(void);

        cout << "Invoking functions through __vftable…" << endl << endl;
        func* virtFuncPtr = (func*)(*vtPtr);        // pointing to the first virtual func.
        cout << "__vftable[0] - ";
        (virtFuncPtr());

        virtFuncPtr = (func*)(*(vtPtr + 1));        // pointing to the second virtual func.
        cout << "__vftable[1] - ";
        virtFuncPtr();

        virtFuncPtr = (func*)(*(vtPtr + 2));       // pointing to the third virtual func.
        cout << "__vftable[2] - ";
        virtFuncPtr();

        return 0;
}
```

```
Invoking function through the object pointer…
virtualNonOverriddenFunc1
virtualDerivedOverriddenFunc1
derivedClassOnlyVirtualFunc

Printing __vftable…
__vftable address: 009133B0
__vftable[0] - 009116E0
__vftable[1] - 00911740
__vftable[0] - 00911760

Invoking functions through __vftable…

__vftable[0] - virtualNonOverriddenFunc1
__vftable[1] - virtualDerivedOverriddenFunc1
__vftable[2] - derivedClassOnlyVirtualFunc
```

This program is about *vptr* and invoking the virtual functions through the *vtable*. But here we will look at how it was done. Here's the code segment we are concerned about:

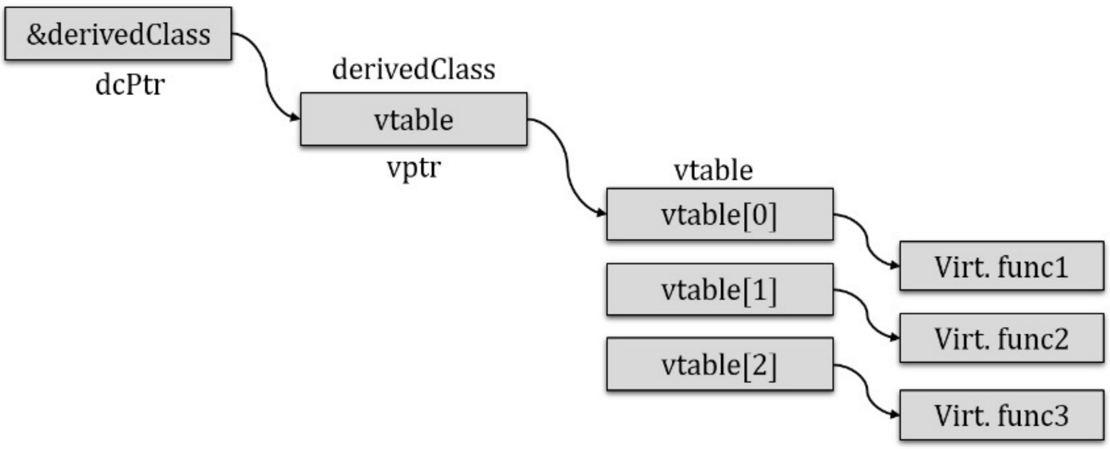
```
derivedClass *dcPtr = new derivedClass;
void(**vtPtr)() = *(void(***)())dcPtr;   //obtaining __vftable address
```

Here we have a pointer *dcPtr* that points to an object of *derivedClass*. This mechanism depends on one fact: the *vptr* of the class is at zero offset. That is, *dcPtr* is actually pointing to the *vptr*. We have confirmed that this is indeed the case with at least the Visual C++ compiler.

Let's dissect the second statement:

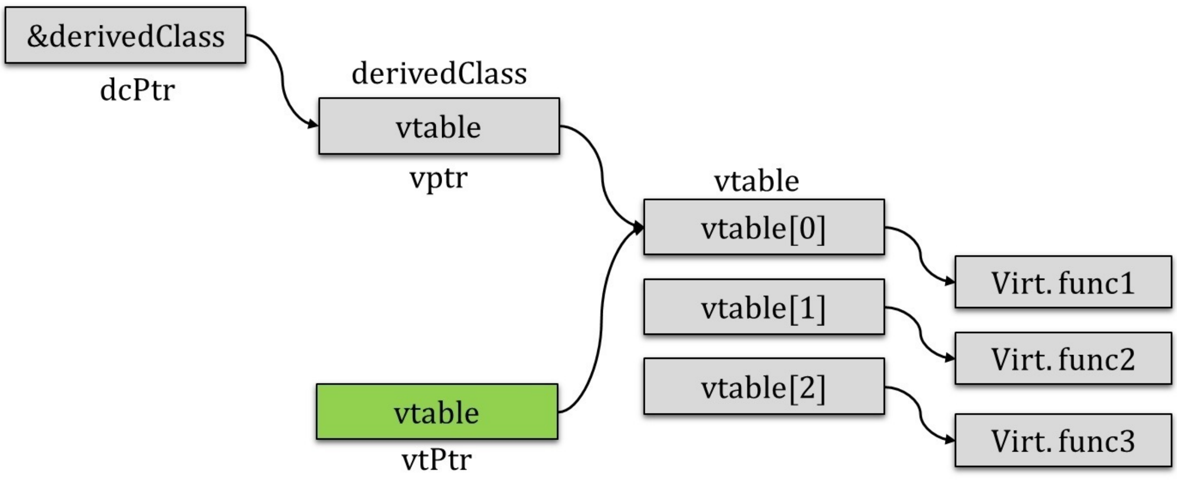
```
void(**vtPtr)() = *(void(***)())dcPtr;   //obtaining __vftable address
```

We know that *"void (*vtPtr)()"* is a declaration of a pointer to a function that has no arguments and no return value. Therefore, *"void (**vtPtr)()"* is a *'pointer to a pointer to a function with no arguments and returns void'*. But why do we have this structure? Remember what a *vtable* is? A *vtable* is an array of function pointers. Perhaps an illustration of what we are dealing with will help.



Remember that an array can be represented by a pointer. The array name itself is a pointer to its first element. Since *vtable* is an array of function pointers, by pointing to *vtable*, *vptr* is a "pointer to pointer to a function pointer". So what *dcPtr* ultimately is a 'pointer-to-pointer-pointer-to-function'. Trace the figure from left and this will become clear.

So where does *vtPtr* fit in here?



Remember we discussed that *vtPtr* is a *'pointer-to-pointer-function'*. So *vtPtr* is pointing to the *vtable* array. Now we know what *vtPtr* needs to point to. It needs to point to the *vtable*. How do we get that, then? We know that the virtual table pointer (*vptr*) is at the beginning of the object, so *dcPtr* is pointing to it.

But what are we exactly doing in this part of the code?

```
*(void(***)())dcPtr
```

'*(void(***)())*' is nothing more than a cast. A cast of type what? A '*pointer-to-pointer-to-pointer-to-functio*n'. Right? This is what *dcPtr* is. What we are doing here, is we are telling the compiler that *dcPtr* is a '*pointer-to-pointer-to-pointer-to-function*'. That's it. Then we dereference it once. Once only. And what do you get when you dereference *dcPtr* once? Check the figure. If we dereference *dcPtr* once we get *'vtable'*. Which is what? Yes, a '*pointer-to-pointer-to-function*'. And this is exactly what we need to assign to our *vtPtr*.

The syntax looks rather complex but if you layer it out, it's nothing much. All we do is cast *dcPtr* to a '*pointer-to-pointer-pointer-to-function*', and then dereference it to get to the object's '*vtable*', which is what we want. Now let's look at calling the virtual functions through the *vtable* entries.

In the rest of the code we are invoking the virtual functions through the *vtable*. We use a typedef in the code to make the call look like more of a regular function call but let's rewrite that part without typedef. We need to understand clearly how this function pointer is working.

So what is *vtPtr* again? It is a *'pointer-to-pointer-to-function'*. Currently it is pointing to the first element of the *vtable*. So to call the function pointer at *vtable[0]*, we just need to dereference *vtPtr*. And do you recall how we called a function pointer? You should not forget the parentheses. We can simply call the first virtual function like this:

```
(*(*vtPtr))();
```

The inner dereference is to get to *vtable[0]*. Since *vtable[0]* is a *pointer-to-function* we have the outer dereference. But remember we saw earlier that in the case of function pointers we actually don’t need to dereference the function pointer itself. So you can simply do the following:

```
        (*vtPtr)();
```

To get to other pointers you just need to add one and two. Since *vtPtr* is a pointer, it does the correct pointer arithmetic to refer to *vtable[1]* and *vtable[2]*. Let’s rewrite the code and confirm that we are correct.

```
…
…
int main(int argc, char** argv)
{
        derivedClass derivedClassObj;
        derivedClass *dcPtr = new derivedClass;

        cout << “Invoking function through the object pointer…” << endl;
        dcPtr->virtualNonOverriddenFunc1();
        dcPtr->virtualOverriddenFunc1();
        dcPtr->derivedClassOnlyVirtualFunc();
        cout << endl;

        void(**vtPtr)() = *(void(***)())dcPtr;

        cout << “Invoking functions through vtPtr…” << endl;

        cout << “(* vtPtr)() - “;
        (*vtPtr)();

        cout << “(*( vtPtr +1))() - “;
        (*(vtPtr + 1))();

        cout << “(*( vtPtr + 2))() - “;
        (*(vtPtr + 2))();
        return 0;
}
```

```
Invoking function through the object pointer…
virtualNonOverriddenFunc1
virtualDerivedOverriddenFunc1
```

```
derivedClassOnlyVirtualFunc

Invoking functions through vtPtr…
(* vtPtr)() - virtualNonOverriddenFunc1
(*( vtPtr +1))() - virtualDerivedOverriddenFunc1
(*( vtPtr + 2))() - derivedClassOnlyVirtualFunc
```

It’s working as it should. So you see function pointers might seem complex in certain cases but if you break it down they are pretty simple. I hope you’ve got a good fundamental grasp of them.

Topic 14

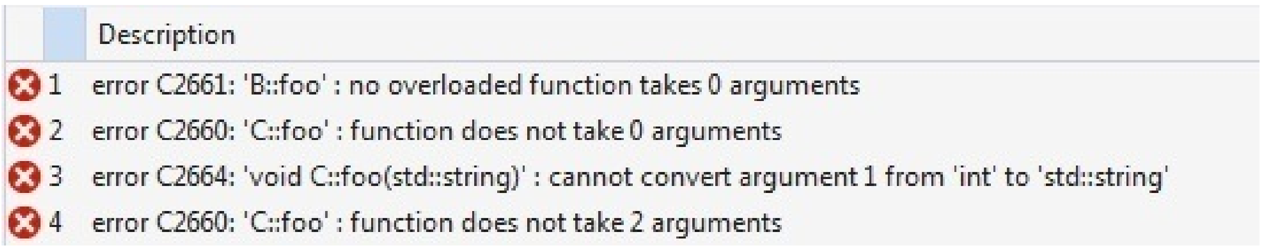
Function Shadowing

This is not a big topic but nonetheless an important one to know about in C++. As usual, let's start off with an example.

We have three very simple classes with each implementing function *foo*. Each class overloads function *foo* so each of these functions are different.

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

class A
{
public:
        void foo()
        {
                cout << "Function foo in class A" << endl;
        }
};

class B : public A
{
public:
        void foo(int val)
        {
                cout << "Function foo(int) in class B" << endl;
        }

        void foo(int val1, int val2)
        {
                cout << "Function foo(int, int) in class B" << endl;
        }
};

class C : public B
{
public:
        void foo(string str)
        {
                cout << "Function foo(string) in class C" << endl;
```

```
    }
};

int main(int argc, char** argv)
{
    A classAobj;
    B classBobj;
    C classCobj;

    classAobj.foo();                // Line 1

    classBobj.foo();                // Line 2
    classBobj.foo(1);               // Line 3
    classBobj.foo(1, 2);            // Line 4

    classCobj.foo();                // Line 5
    classCobj.foo(1);               // Line 6
    classCobj.foo(1, 2);            // Line 7
    classCobj.foo("foo");           // Line 8

    return 0;
}
```



```
  Description
1 error C2661: 'B::foo' : no overloaded function takes 0 arguments
2 error C2660: 'C::foo' : function does not take 0 arguments
3 error C2664: 'void C::foo(std::string)' : cannot convert argument 1 from 'int' to 'std::string'
4 error C2660: 'C::foo' : function does not take 2 arguments
```

Intuitively, we would assume class B to be able to access A's methods and class C to be able to access A's and B's methods. After all, the methods are public and the inheritance is public. But we are getting quite a bit of compiler errors. The problem is not with access. B has access to A's method and C has access to A's and B's. The problem is the compiler just can't see those functions in the base classes. Because the functions are *shadowed*.

Take a look at the compiler errors. The first error is for line 2 about class B having no zero argument function. Class B has two overloaded functions, one taking an integer argument and the other taking two integer arguments. The second error is for line 5 and it is the same as the first error, but for class C this time. The third error is for line 6, and it says that the integer argument we passed in line 6 cannot be converted to a string argument. Because class C has only one *foo* which takes in a string argument. The fourth error for line 7 is about having no function that takes two arguments.

What do all of these errors say? They are saying that the compiler simply cannot find the *foo* functions that we want it to. But we know that these functions are properly inherited and accessible. They are just shadowed. They are shadowed by the overloaded functions of the same name. Whenever we overload an inherited function, all the base class functions with the same function name are shadowed. But, because something is in the shadows doesn't mean it's not there. The functions are there. We just need to tell the compiler that those functions exist. It's pretty simple to do that. Let's make this right.

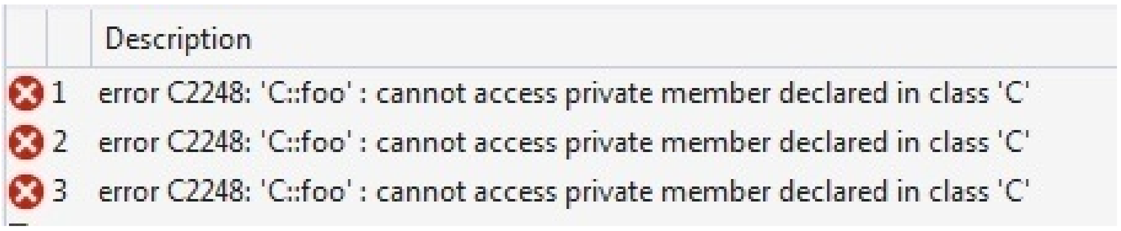
```
#include<iostream>
using namespace std;

class A
{
public:
        void foo()
        {
                cout << “Function foo in class A” << endl;
        }
};

class B : public A
{
public:
        using A::foo;
        void foo(int val)
        {
                cout << “Function foo(int) in class B” << endl;
        }

        void foo(int val1, int val2)
        {
                cout << “Function foo(int, int) in class B” << endl;
        }
};

class C : public B
{
public:
        using B::foo;
        void foo(string str)
        {
                cout << “Function foo(string) in class C” << endl;
        }
};

int main(int argc, char** argv)
{
        A classAobj;
        B classBobj;
        C classCobj;

        classAobj.foo();

        classBobj.foo();
        classBobj.foo(1);
        classBobj.foo(1, 2);

        classCobj.foo();
        classCobj.foo(1);
        classCobj.foo(1, 2);
        classCobj.foo(“foo”);
        return 0;
}
```

```
Function foo in class A
Function foo in class A
Function foo(int) in class B
Function foo(int, int) in class B
Function foo in class A
Function foo(int) in class B
Function foo(int, int) in class B
Function foo(string) in class C
```

We just need to use the *using* directive to bring the function we need in to the scope. This is the same thing we do when we say *‘using namespace std*‘. We just bring those definitions in to the scope where we need them. So when we said *‘using A::foo*‘, we brought all the *foo* functions defined in class A in to B. And we do the same thing in class C. Note how we only brought class *B::foo* in to class C scope, although we are using the class *A::foo*. We don’t need to because class *A::foo* is already in the scope of B, so *B::foo* already has *A::foo* in its scope.

Now here’s an important point. The place where you put the using directive matters. Let’s change class C like this so that the using directive is called before the public declaration.

```
…
…
class C : public B
{
        using B::foo;       // before declaring public members
public:
        void foo(string str)
        {
                cout << “Function foo(string) in class C” << endl;
        }
};
…
…
```



```
Description
1   error C2248: 'C::foo' : cannot access private member declared in class 'C'
2   error C2248: 'C::foo' : cannot access private member declared in class 'C'
3   error C2248: 'C::foo' : cannot access private member declared in class 'C'
```

Now the compiler is complaining about trying to access private methods. The compiler is now fully aware of the functions in classes A and B, but unfortunately these functions are brought in to the scope as private. This is all because our using directive is before the public declaration. So you see, using directive is just about bringing functions in to that scope, and it matters where we bring these functions in to. Since the functions were brought in before we declared public, for the compiler it is as if these *foo* functions of A and B were defined in the private section of class C. The functions are there, the compiler knows it, but we just don’t have the access rights to them.

That’s just about it function shadowing. Nothing much to it but it is important to realize what is happening when you overload a base class function. Before we leave this topic, would you think anything would’ve changed if function *foo* was virtual? No it wouldn’t. Because virtual functions are about dynamic dispatching. It’s a runtime mechanism, not compile time. Before we get to the runtime we need to pass the compilation. There is no relation between function shadowing and virtualness. Overloaded functions are different functions, so *foo()* is a different function than *foo(int)*. *foo()* being virtual has no affect on *foo(int, int)*.

Topic 15

Understanding the Destructor

We know what a destructor is and what it does but let’s take some time in this topic to really understand what happens in the destruction phase of an object. Here we go.

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

class A
{
public:
        A()
        {
                cout << “A constructor” << endl;
        }

        ~A()
        {
                cout << “A destructor” << endl;
        }
};

class B : public A
{
public:
        B()
        {
                cout << “B constructor” << endl;
        }

        ~B()
        {
                cout << “B destructor” << endl;
        }
};


int main(int argc, char** argv)
{
        B Bobj;
```

```
    return 0;
}
```

```
A constructor
B constructor
B destructor
A destructor
```

We are all too familiar with this. B’s constructor first calls A’s constructor and then B’s. When the object goes out of scope the destructor is called by the compiler automatically. B’s destructor is called first and then A’s. We know this.

Virtual destructor

Let’s get the pointers in.

```
…
…
int main(int argc, char** argv)
{
    A* Aptr = new B;
    delete Aptr;
    return 0;
}
```

```
A constructor
B constructor
A destructor
```

We have an issue now, don’t we? We are calling both A and B constructors but only A’s destructor is called. This is important to note. B’s destructor is not called because the destructor is not virtual. When we created the object we called the B’s constructor directly. But when we deleted it, we called the delete on a pointer of class type A. So the compiler called the destructor of class A. Let’s fix this.

```
class A
{
public:
    A()
    {
        cout << “A constructor” << endl;
    }

    virtual ~A()
    {
        cout << “A destructor” << endl;
    }
};
…
…
```

```
A constructor
B constructor
B destructor
A destructor
```

So make sure you make the destructor virtual if your class is designed to be inherited. Because it is common for derived classes to be associated with base class pointers, we need to make sure when those base class pointers are deleted and the required destructors are called.

Object size through destruction

It'll be interesting to see how an object's size is changed during construction and destruction.

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

class A
{
    int var1;
    int var2;
public:
    A()
    {
        cout << “A constructor. Size: “ << sizeof(*this) << endl;
    }

    virtual ~A()
    {
        cout << “A destructor. Size: “ << sizeof(*this) << endl;
        }
};

class B : public A
{
        int var3;
        int var4;
public:
        B()
        {
                cout << "B constructor. Size: " << sizeof(*this) << endl;
        }

        ~B()
        {
                cout << "B destructor. Size: " << sizeof(*this) << endl;
        }
};


int main(int argc, char** argv)
{
        B Bobj;
        return 0;
}
```

```
A constructor. Size: 12
B constructor. Size: 20
B destructor. Size: 20
A destructor. Size: 12
```

Size of object A is 12 bytes. Know why? There are two integers, each with 4 bytes and then we have the *vptr*, which is another 4 bytes. Object B has two integers so it adds 8 more bytes. See how the size of the object changes from construction and destruction?

Calling destructor for class members

When you have pointer allocations in your class it is important that you deallocate them in your destructor by yourself. The compiler is only going to call the base class destructor. It is not going to do any pointer deleting.

```
#include<iostream>
```

```
using namespace std;

class A
{
public:
        A()
        {
                cout << “A constructor” << endl;
        }

        virtual ~A()
        {
                cout << “A destructor” << endl;
        }
};

class B
{
        A* Aptr;
public:
        B()
        {
                Aptr = new A;
                cout << “B constructor” << endl;
        }

        ~B()
        {
                cout << “B destructor” << endl;
        }
};


int main(int argc, char** argv)
{
        B Bobj;
        return 0;
}
```

```
A constructor
B constructor
```

```
B destructor
```

Note that the A’s destructor is not called. So remember that if you have pointer allocations, deallocate them yourself. This is not the case if you have an automatic variable.

```
…
…
class B
{
        A Aobj;

public:
        B()
        {
                cout << “B constructor” << endl;
        }

        ~B()
        {
                cout << “B destructor” << endl;
        }
};
```

```
A constructor
B constructor
B destructor
A destructor
```

When *Bobj* goes out of scope the destructor of B is called and this destructor in turn calls the member variable’s destructors. This is why it is always a good idea to use smart pointers and use the concept of Resource-Aquisition-Is-Initialization (RAII).

Delete *this*

Let’s end this topic with this: what exactly does ‘*delete this*’ do? Try this out.

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

class A
{
```

```
public:
        A()
        {
                cout << “A constructor” << endl;
        }

        ~A()
        {
                cout << “A destructor” << endl;
        }

        void callDeleteThis()
        {
                delete this;
        }
};

int main(int argc, char** argv)
{
        A Aobj;
        aobj.callDeleteThis();
        return 0;
}
```

```
A constructor
A destructor
```

This shouldn’t work. The result depends on the runtime system but you must get some type of runtime failure. If you put a breakpoint at the return statement though, you could see the output shown above.

We are trying to do an undefined operation here. What does *‘delete this’* do then? It simply calls the destructor. That’s it. Why then is this giving a runtime error? Because we are trying to call the destructor on an object that’s already destructed. The destructor is first called when we call *callDeleteThis*. And when the object goes out of scope the compiler implicitly calls the destructor on *Aobj*. But *Aobj* is already destructed. This is undefined behavior.

Then how about calling the destructor explicitly? Can you do that? You sure can.

```
…
…
```

```
int main(int argc, char** argv)
{
        A Aobj;
        aobj.~A();
        return 0;
}
```

Did you notice something different now? This code don’t have any runtime failures. It executes the fine. But we are doing the same thing as before when we did ‘*delete this*’, right? Actually, no.

Now here’s the caveat. Destructor does not deallocate the object. The destructor is just another function that is supposed to do some housekeeping before the object is deallocated. But the destructor itself does not deallocate the memory. After the destructor is called, the object memory is still there intact. The object still technically exists. This is the difference between calling the destructor explicitly and calling it through *operator delete*. Note how we got a runtime error before when did ‘*delete this*’, but we had no issues when we called the destructor explicitly. This is because *delete* is the one that actually does the memory deallocation. So operator delete has two operations:

- First calls the destructor, and lets it do the necessary housekeeping.
- Deallocates the memory of the object.

Does the functionality of *operator delete* sound familiar? This is the exact opposite of *operator new*. Operator new allocates memory and then constructs the object, while operator delete destructs the object and deallocates the memory. Here’s an example.

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

class A
{
public:
        int var;
        A() : var(10)
        {
                cout << “A constructor” << endl;
        }

        ~A()
        {
                cout << “A destructor” << endl;
        }
```

```
    void callDeleteThis()
    {
        delete this;
    }
};

int main(int argc, char** argv)
{
    A Aobj;
    aobj.~A();
    aobj.var = 20;
    cout << “Aobj.A: “ << Aobj.var << endl;
    return 0;
}
```

```
A constructor
A destructor
Aobj.A: 20
A destructor
```

Here we are constructing an object, then (trying to) destruct it by calling the destructor explicitly, and then we assign a value to its member variable and read it back. All without having any issues. Now are you convinced that the destructor is just another function? Its only speciality is that it is automatically called just before the object is deleted.

The story’s different if we do this:

```
int main(int argc, char** argv)
{
    A Aobj;
    aobj.callDeleteThis();
    aobj.var = 20;
    cout << “Aobj.A: “ << Aobj.var << endl;
    return 0;
}
```

This is a runtime failure. Why? Because we destruct the object through *delete*. This actually deallocates the memory so the object does not exist anymore. And we are trying to access a non-existent object, which causes the runtime failure.

Why, then, would we need to call the destructor explicitly? There aren’t many cases where you should be calling the destructor explicitly. One case where you would call the destructor explicitly is when you want to destroy the object but not to deallocate the memory. When would we want to do that? Remember *‘placement new’*? Where we construct new objects on memory that is already allocated. This is one situation where we would call the destructor explicitly. We call the destructor explicitly to destroy the object, but not to deallocate the memory, but then use placement new to construct a new object on that block of memory. Like this:

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {
                cout << “A constructor” << endl;
        }

        ~A()
        {
                cout << “A destructor: “ << var << endl;
        }
};

int main(int argc, char** argv)
{
        A* Aptr = new A(1);
        cout << “Aptr->var: “ << Aptr->var << endl;
        aptr->~A();

        new (Aptr)A(2);
        cout << “Aptr->var: “ << Aptr->var << endl;
        delete Aptr;
        return 0;
}
```

```
A constructor
Aptr->var: 1
A destructor: 1
```

```
A constructor
Aptr->var: 2
A destructor: 2
```

You can confirm that we are indeed constructing two different objects in one place. We destruct the first object, which actually does nothing more than a printout, and then we call placement new to construct a new object on the memory block *Aptr* is allocated to.

I believe this topic provided you with some internals of the destructor. A destructor is almost a normal function, except for the fact that the compiler implicitly calls it before deallocating the object’s memory. Other than that it is just another pretty normal function.

Topic 16

Operator Overloading

Operator overloading is a new important feature added to C++. Every time you do *‘cout <<’* you are using operator overloading. As important as it is I’ve found that references that do a proper treatment of this topic is rather scarce. Operator overloading isn’t difficult but there are a few rules you need to know about it.

What is exactly operator overloading? It is the same as function overloading. You change the behavior of an existing operator to do something else you want. It’s important to remember that you can only overload existing C++ operators. You cannot invent new ones. And when you overload an operator, it must be overloaded to work with an user defined type. What that means is that you cannot overload an operator to something different on a built in type. For example when you do *‘x+y’*, where *‘x‘* and *‘y‘* are integers, operator ‘+’ will always do integer addition. You cannot change the behavior of operator ‘+’ for built in types such as integers. What you can do is to overload the operator ‘+’ for a class type you defined. Let’s say you have two instances of *myClass*, *obj1* and *obj2*; you can overload operator ‘+’ to define an implementation for *‘obj1+obj2’*. We’ll look at examples later, but first let’s have a look at the fundamental rules.

Rules of operator overloading

Here are the basic rules:

- Only existing operators can be overloaded.
- The following operators cannot be overloaded: *scope resolution (::), member access (. and .*), ternary conditional (?:).*
- Most operators have a fixed number of arguments.
- Most operators can be overloaded as either a member function or a non-member function.

Having said that, here’s the other thing about operators: most operators can be divided into either unary operators or binary operators. You must have noticed that I used ‘most’ in a few places; this is because there are a few exceptions. But the rule holds for the majority.

- Unary operators – These are the operators that take only one parameter.
    - Increment (++) and decrement (—) operators
    - Positive (+), negative (-) and logical not (!)
- Binary operators – These are the operators that take two parameters.
    - Arithmetic operators – plus(+), minus (-), division (/) and multiplication (*)
    - I/O operators - << and >>
    - Comparison operators – greater than (<, <=), less than (>, >=), equal (==)

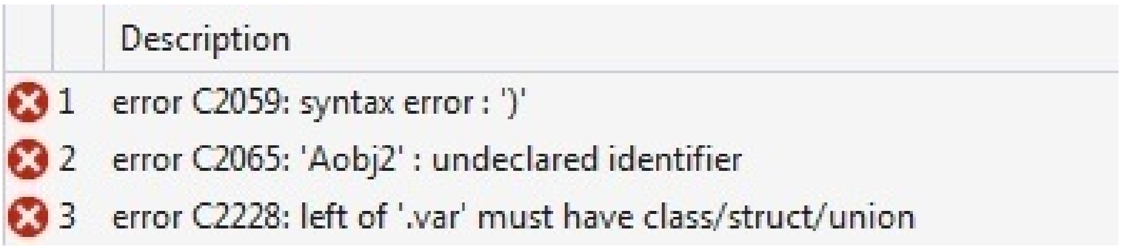
Unary operator overloading

Let’s start with non-member function for unary operator ‘+’.

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}
};

A operator+(const A& Aref)
{
        cout << “Received value: “ << Aref.var << endl;
        return A(Aref.var + 1);
}

int main(int argc, char** argv)
{
        A Aobj1(20);
        A Aobj2(+Aobj1);
        cout << “Aobj2.var: “ << Aobj2.var << endl;
        return 0;
}
```

```
Received value: 20
Aobj2.var: 21
```

It’s easy to understand the implementation. The operator ‘+’ simply increments the received reference’s value and returns an object with that value. Note how I called the operator on the object as ‘*+Aobj1*’? The operator is on the left of the object. What if it was on the right, like ‘*Aobj1+*‘? You can’t. You’d get an error similar to the following:



		Description
✖	1	error C2059: syntax error : ')'
✖	2	error C2065: 'Aobj2' : undeclared identifier
✖	3	error C2228: left of '.var' must have class/struct/union

Can you guess what is happening? Did you notice that the operator '+' is both a unary and a binary operator? When the operator appears on the right, the compiler treats it as the binary operator '+'. And binary operators need two arguments, the one on its left and the one on the right. When we write *'Aobj1+'*, the compiler parses it as the binary operator '+' and expects the second argument to the right of the operator. These compiler errors are because it is missing that argument. It would become clear when we look at binary operators but now keep in mind that unary operators must be on the left of the operand. Well, at least in most cases.

Now how would we implement this unary operator '+' as a member-function? It's pretty straight forward. When the function was a non-member we had to pass the class type as the parameter. And there must be only one parameter since this is a unary operator. When this operator overloaded function is a member of the class, we no longer need to specify a parameter, because every member function gets an implicit *'*this'*. And *'*this'* is the one argument that the operator requires. So for unary operators we must not have any function parameters. This is how we make this a member function.

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}

        A operator+()
        {
                cout << "'this' value: " << this->var << endl;
                return A(this->var + 1);
        }
};

int main(int argc, char** argv)
{
        A Aobj1(1);
        A Aobj2(+Aobj1);
        cout << “Aobj2.var: “ << Aobj2.var << endl;
        return 0;
}
```

```
‘this’ value: 1
Aobj2.var: 2
```

See, nothing much to it. The parameter we had in the non-member function now becomes the implicit *‘this’* in the member-function. Now let’s take a quick look at the increment and decrement operators.

You know that increment and decrement operators come in two versions: *prefix* and *postfix*. Prefix is when the operator is on the left of the operand and postfix is when it is on the right. Let’s look at the prefix increment. There isn’t anything different from the increment operator.

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}
};

A operator++(A& Aref)
{
        cout << “Received value: “ << Aref.var << endl;
        return A(Aref.var + 1);
}

int main(int argc, char** argv)
{
        A Aobj1(1);
        A Aobj2(++Aobj1);
        cout << “Aobj2.var: “ << Aobj2.var << endl;
        return 0;
}
```

```
Received value: 1
Aobj2.var: 2
```

How are we going to implement the postfix version? Because from increment operator we know that the operator needs to be on the left side. There is a special handling for this case. Insert a dummy integer parameter to the overloaded operator.

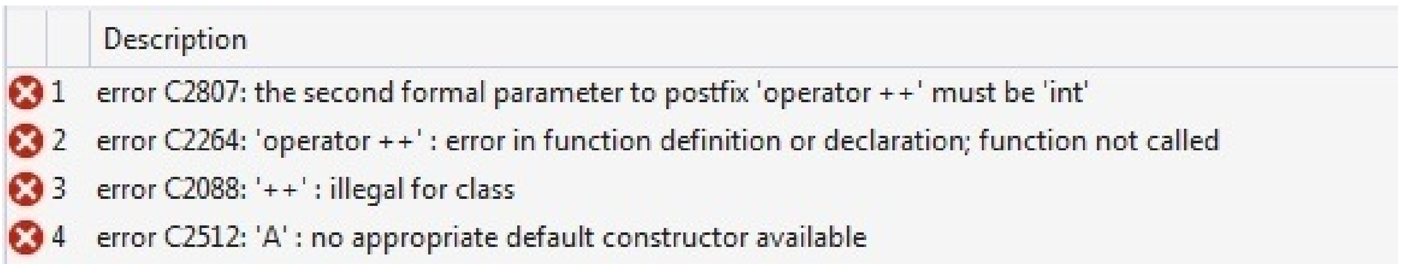
```
#include<iostream>
using namespace std;

class A
{
public:
        int var;
        A(int val) : var(val)
        {}
};

A operator++(A& Aref, int dummy)
{
        cout << “Received value: “ << Aref.var << endl;
        return A(Aref.var + 1);
}

int main(int argc, char** argv)
{
        A Aobj1(1);
        A Aobj2(Aobj1++);
        cout << “Aobj2.var: “ << Aobj2.var << endl;
        return 0;
}
```

```
Received value: 1
Aobj2.var: 2
```

Pretty neat right? But you must be wondering now about this dummy parameter. What does the dummy parameter do? And does it need to be an integer? Let’s put a float in there. After all this is a dummy parameter. It shouldn’t matter what type it is. Or does it?

```
…
```

```
…
A operator++(A& Aref, float dummy)
{
        cout << “Received value: “ << Aref.var << endl;
        return A(Aref.var + 1);
}
…
…
```



```
    Description
1   error C2807: the second formal parameter to postfix 'operator ++' must be 'int'
2   error C2264: 'operator ++' : error in function definition or declaration; function not called
3   error C2088: '++' : illegal for class
4   error C2512: 'A' : no appropriate default constructor available
```

That’s a bunch of errors but only look at the first one. The compiler is saying that the second parameter for the postfix operator must be an int. If you want to overload the postfix operator you must make the second operand an int. That’s the rule. Just obey it.

Now you must be scratching your head trying to generalize the rules of operator overloading. Don’t! Do not try to find a set of general rules for operator overloading. Certain operators have their special overload function signature. This will become clear to you when we look at overloading operator *new*. Just keep in mind that these operators have their way of being overloaded. You need to adhere to their ways. In case of increment/decrement postfix operator, the second parameter must be an int. Not a float, not a *void, but only an int.

To finish off unary operators let’s look at how we’d implement the postfix decrement operator as a member-function.

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}

        A operator—(int dummy)
        {
                cout << “Received value: “ << this->var << endl;
                return A(this->var - 1);
        }
};

int main(int argc, char** argv)
{
        A Aobj1(1);
        A Aobj2(Aobj1—);
        cout << "Aobj2.var: " << Aobj2.var << endl;
        return 0;
}
```

```
Received value: 1
Aobj2.var: 0
```

Binary operator overloading

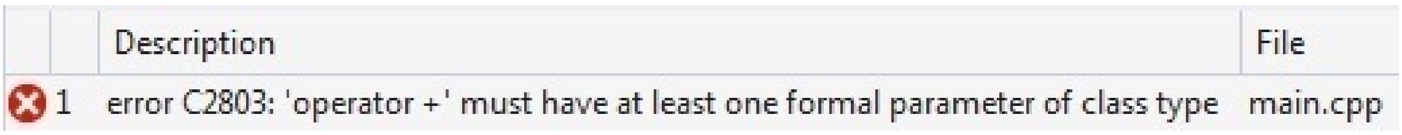
Now we'll move on to overloading binary operators. Let's start with the '+' operator. This is one operator that works as both a unary and a binary operator. Let's look at the non-member function first.

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}
};

A operator+(A& leftOp, A& rightOp)
{
        cout << "Received left: " << leftOp.var << endl;
        cout << "Received right: " << rightOp.var << endl;
        return A(leftOp.var + rightOp.var);
}

int main(int argc, char** argv)
{
        A Aobj1(1);
        A Aobj2(2);
        A Aobj3(Aobj1 + Aobj2);
        cout << “Aobj3.var: “ << Aobj3.var << endl;
        return 0;
}
```

```
Received left: 1
Received right: 2
Aobj3.var: 3
```

As this is a binary operator you need two operands. One for the operand on the left of the operator and the other for the one on the right. It is important to note the association of the function parameters and the operator operands. The first parameter *leftOp* is the one that comes to the left of the operator and the *rightOp* is the one that goes on the right. Pretty intuitive. Remember I said that you cannot overload operators for built-in types? What would happen if we try it?

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}
};

A operator+(int leftOp, int rightOp)
{
        cout << “Received left: “ << leftOp << endl;
        cout << “Received right: “ << rightOp << endl;
        return A(leftOp + rightOp);
}

int main(int argc, char** argv)
{
        A Aobj(1 + 2);
        cout << “Aobj.var: “ << Aobj.var << endl;
        return 0;
}
```



		Description	File
✖	1	error C2803: 'operator +' must have at least one formal parameter of class type	main.cpp

We cannot. And what does it mean that "*must have at least one formal parameter*?"

It means that when you overload an operator it must have a formal parameter. That is, a user defined type. Not built in types. So does that mean we can have one class type parameter and the a built-in type? Yes.

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}
};

A operator+(A& leftOp, int rightOp)
{
        cout << "Received left: " << leftOp.var << endl;
        cout << "Received right: " << rightOp << endl;
        return A(leftOp.var + rightOp);
}

int main(int argc, char** argv)
{
        A Aobj1(1);
        A Aobj2(Aobj1 + 2);
        cout << "Aobj2.var: " << Aobj2.var << endl;
        return 0;
}
```

```
Received left: 1
Received right: 2
Aobj3.var: 3
```

So you can make your class type work with built in types. But make sure you have the operand association. In the example above the left operand must be of our class type. It cannot be the other way around. That is, we couldn't write "*A Aobj2(2 + Aobj1)*". For that we need to have another overloaded operator "*A operator+(int leftOp, A& rightOp)*".

Now how would this look as a member-function?

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}

        A operator+(A& rightOp)
        {
                cout << "Received left: " << this->var << endl;
                cout << "Received right: " << rightOp.var << endl;
                return A(this->var + rightOp.var);
        }
};

int main(int argc, char** argv)
{
        A Aobj1(1);
        A Aobj2(2);
        A Aobj3(Aobj1 + Aobj2);
        cout << "Aobj3.var: " << Aobj3.var << endl;
        return 0;
}
```

```
Received left: 1
Received right: 2
Aobj3.var: 3
```

Remember, we discussed how the member functions always have an implicit '*this' that corresponds to the object that the function was called on. It's the same thing here. The left operand of the operator becomes the '*this'. In our example we are invoking the operator of object *Aobj1* and *Aobj2* becomes the *rightOp* argument. This would become clearer when you see that you can actually call the overloaded operator as calling a member function.

```
…
…
int main(int argc, char** argv)
{
	A Aobj1(1);
	A Aobj2(2);
	A Aobj3(Aobj1.operator+(Aobj2));
	cout << “Aobj3.var: “ << Aobj3.var << endl;
	return 0;
}
```

You see overloaded operators are just functions. Functions that the compiler handles a bit differently so you can call them as you would a normal operator. Would you really want to call the operator ‘+’ as *‘Aobj1.operator+(Aobj2)’*? What’s the point? You’d rather define a member function *‘A Plus(A& obj)’* to do that. But it helps to clear things, that it is another member function that is called on the object on the left of the operator.

Now how about if we want to have a member-function but with an int parameter like we discussed before in the non-member case *“A operator+(A& leftOp, int rightOp)*“? How would it work in the member-function version? We definitely could make a member-function version where the left operand is the class type.

```
…
…
A operator+(int rightOp)
{
	cout << “Received left: “ << this->var << endl;
	cout << “Received right: “ << rightOp << endl;
	return A(this->var + rightOp);
}
…
…
```

We can do this because the left operand should be of class type. But what about when we want to implement a member-function for having the integer as the left operand? That is, what is the member-function version of non-member function *“A operator+(int leftOp, A& rightOp)*“? There isn’t one. This is one limitation you have with the member-function. There is no special handling with dummy variables. You just cannot do this with a member function. If you want to overload the plus operator to have a built-in type left operand you must use a non-member function.

The rules are the same for the other binary operators: I/O operators and comparison operators. There is no restriction on the operator return type. So far we only looked at a simple case where we return an object. But you could have different return types depending on the operator. For example, comparison operators in all cases should be returning a boolean value. But you are free to return any value.

Finally we will look at two other operators that have some special handling.

Function operator overloading

First is overloading the function call operator '()', which is called the function operator. This has quite nifty use cases.

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}

        void operator() ()
        {
                cout << "In overloaded operator ()" << endl;
        }
};

int main(int argc, char** argv)
{
        A Aobj(1);
        aobj();
        return 0;
}
```

```
In overloaded operator ()
```

Did you notice how we called the overloaded operator? Pretty neat syntax, isn't it? And we are not limited to a no argument function. We can load it up.

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}

        void operator() (int a, int b)
        {
                cout << “In overloaded operator(): “ << a << ” and “ << b << endl;
        }
};

int main(int argc, char** argv)
{
        A Aobj(1);
        aobj(1, 2);
        return 0;
}
```

One thing you must adhere with this overloaded function operator is that it must be a member-function. And also that the parentheses must come to the left of the operand.

Operator new overloading

Our last operator is the *new* operator (and of course the associated *delete* operator). We actually did operator new overloading in the topic on *“Understanding new”*. Remember *placement new*? That was operator overloading. *Operator new* has three main formats:

- The global standard operator new has the signature

```
        void* operator new(std::size_t count);
```

- Placement new has the signature

```
        void* operator new(std::size_t count, void* ptr);
```

- An user defined overloaded operator has the signature

```
        void* operator new(std::size_t count, user_args…);
```

All three of the above operator formats are overloadable. There is a special operator new specific way of calling these functions. Remember I told you not to try to generalize the function overloading mechanism. This is another one of those operator specific ones.

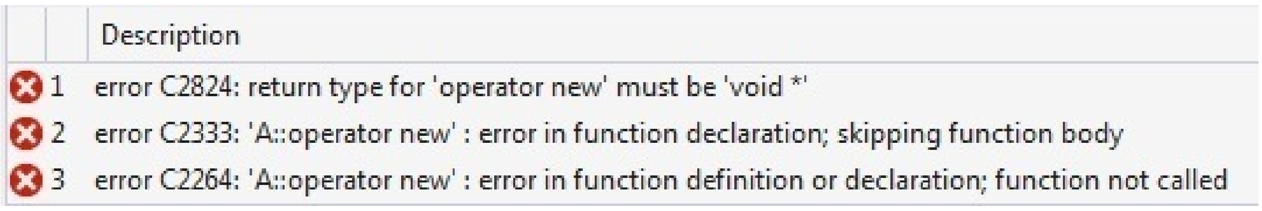
Let’s look at a simple example.

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

class A
{
public:
        int var;
        A(int val) : var(val)
        {}
};

int main(int argc, char** argv)
{
        A* Aptr = new A(1);        // Line 1
        return 0;
}
```

I want to point out something important in this example. See the call to *new* in line 1? It is important to understand that this is NOT the call to operator new. What we call in line 1 is the ‘*new-expression*’. ‘new-expression’ is the one that calls *operator new*. Operator new is only responsible for memory allocation while ‘new-expression’ does two things: It first calls the operator new to allocate memory and then constructs the object at the allocated memory by calling the constructor. Keep the distinction in mind. What we are overloading here is the operator new, the function that will be called by the new-expression first. So you must understand the importance of proper overloading of operator new. Because you see if we don’t do the proper memory allocation in our overloaded operator new, then when we call new-expression, it is not going to work. The compiler will do its part to make sure we overload it properly.

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

class A
{
```

```
public:
        int var;
        A(int val) : var(val)
        {}

        void operator new(size_t count)
        {
                cout << "Operator new" << endl;
        }
};

int main(int argc, char** argv)
{
        A* Aptr = new A(1);
        return 0;
}
```



```
Description
1   error C2824: return type for 'operator new' must be 'void *'
2   error C2333: 'A::operator new' : error in function declaration; skipping function body
3   error C2264: 'A::operator new' : error in function definition or declaration; function not called
```

See, the compiler wants us to have the correct return type. We cannot overload it as we wish. Also, notice that the new-expression calculates the size and passes it as the first argument to the operator new. Let's see how improper implementation can fail things.

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

class A
{
public:
        int var1;
        int var2;
        int var3;
        A(int val) : var3(val)
        {}

        void* operator new(size_t count)
        {
                return ::operator new(8);
        }
};

int main(int argc, char** argv)
{
        cout << “Size of A: “ << sizeof(A) << endl;
        A* Aptr = new A(1);
        cout << “Aptr->var: “ << Aptr->var3 << endl;
        return 0;
}
```

Here I put two more int variables to the class. This should make the size of a *A* object to be 12 bytes (one int is 32-bits). And in the operator new, which is supposed to allocate 12 bytes of heap memory, it only allocates half of it. So the memory block returned by operator new to the new-expression does not have enough space to put three ints. new-expression then constructs the 12 byte object in a 8 byte space and then we try to access the last int variable, which is clearly out of the object memory space. This should result in a runtime error. Depending on the system, even if we tried to access the first variable we could see a runtime failure because we are clearly violating the allocated memory boundaries. So if you are overloading the operator new, make sure you implement it correctly.

Now in the final part let’s see how we can overload the new operator for a custom signature.

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

class A
{
public:
        int var;

        A(int val) : var(val)
        {}

        void* operator new(size_t count, int var1, int var2, bool state)
        {
                if (state)
                {
                        cout << “TRUE state: var1- “ << var1 << “, var2- “ << var2 << endl;
                }
                return ::operator new(count);
        }
};

int main(int argc, char** argv)
{
        A* Aptr = new (1, 2, true) A(1);
        return 0;
}
```

```
TRUE state: var1- 1, var2- 2
```

You see, you can pass any number of arguments to your overloaded operator new. The first argument, *count*, is always passed implicitly by the new-expression. The rest of the arguments, however, need to be passed as a list before the operand. Notice how we pass the list of arguments. It comes before the operand. It looks peculiar but remember, the compiler has its way of handling operator overloads. Keep in mind that you don’t need to pass the first argument, which is passed by new-expression.

I hope you’ve realized how operator overloading works and how powerful that mechanism can be to make the operators perform customized functionality to suit your needs.

Topic 17

Multiple Inheritance

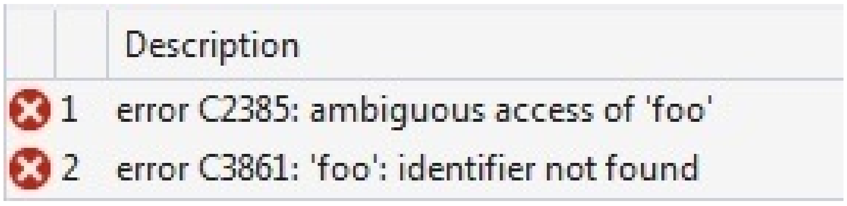
Multiple inheritance can be a powerful feature when done right. But the longer the inheritance hierarchy the more error prone it becomes. This topic will do a quick discussion on multiple inheritance, the dreaded diamond problem and then virtual inheritance.

Here's a simple multiple inheritance example.

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

class A
{
public:
        int var;

        A(int val) : var(val)
        {}

        void foo()
        {
                cout << "Class A foo()" << endl;
        }
};

class B
{
public:
        int var;

        B(int val) : var(val)
        {}

        void foo()
        {
                cout << "Class B foo()" << endl;
        }
};
```

```
class ABDerived : public A, public B
{
public:
        ABDerived(int var1, int var2) : A(var1), B(var2)
        {}
};

int main(int argc, char** argv)
{
        ABDerived ABDerObj(1, 2);
        aBDerObj.foo();
        return 0;
}
```

Classes *A* and *B* have common member names they share, *var* and *foo*. You know this is going to be trouble. Class *ABDerived* has access to function *foo* of both classes *A* and *B*. So which *foo* are we calling? Luckily the compiler doesn't randomly choose one. It complains that it doesn't know which one to choose.



```
Description
1  error C2385: ambiguous access of 'foo'
2  error C3861: 'foo': identifier not found
```

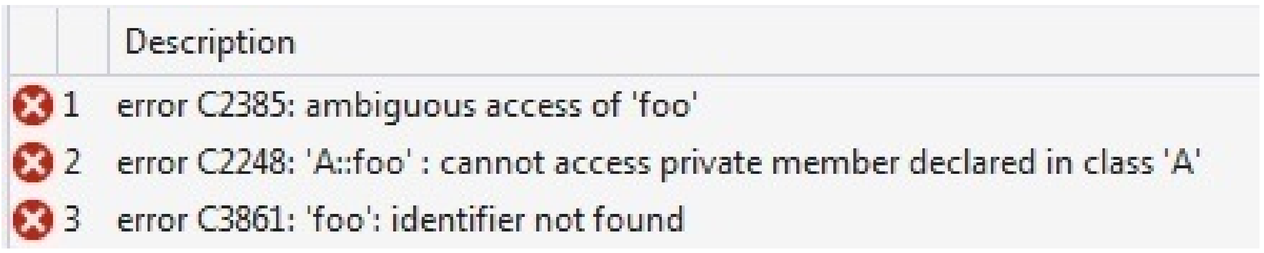
All we need to do is tell the compiler which *foo* we want to invoke by correctly qualifying the call with the class name.

```
…
…
int main(int argc, char** argv)
{
        ABDerived ABDerObj(1, 2);
        aBDerObj.A::foo();        //qualifying the function call
        return 0;
}
```

```
Class A foo()
```

What if *foo* was private in class *A*? In that case *ABDerived* has access to only *B::foo* so we don't need to qualify the call to *foo*, right? Nope.

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

class A
{
public:
        int var;

        A(int val) : var(val)
        {}

private:
        void foo()
        {
                cout << “Class A foo()” << endl;
        }
};

class B
{
public:
        int var;

        B(int val) : var(val)
        {}

        void foo()
        {
                cout << “Class B foo()” << endl;
        }
};

class ABDerived : public A, public B
{
public:
        ABDerived(int var1, int var2) : A(var1), B(var2)
        {}
};

int main(int argc, char** argv)
{
        ABDerived ABDerObj(1, 2);
        aBDerObj.foo();
        return 0;
}
```



```
Description
1   error C2385: ambiguous access of 'foo'
2   error C2248: 'A::foo' : cannot access private member declared in class 'A'
3   error C3861: 'foo': identifier not found
```

Why is the call still ambiguous to the compiler? There is obviously one *foo* that *ABDerived* has access to. This is because function access rights is the one that the compiler checks at the very end. Simply, it is because before the compiler knows that *ABDerived* has no right to access *foo*, it finds two possible *foo*s and complains about the call ambiguity before going further and finding out *A::foo()* is inaccessible. So even though one of the *foo*s is private, you still need to qualify the function call. Now let’s look at what the‘*dreaded diamond* problem is.

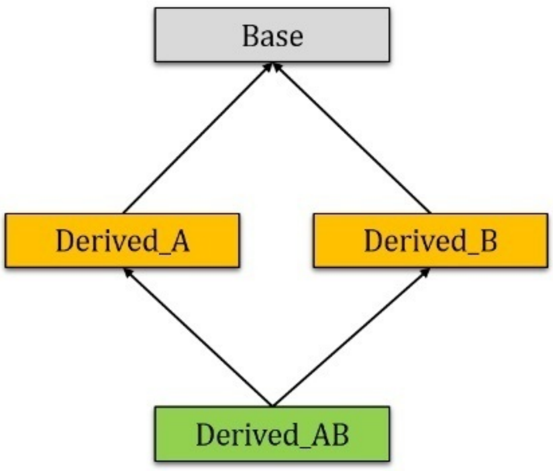
The diamond problem

This is our class hierarchy.

```
class Base
{
public:
    int var;

    Base(int val) : var(val)
    {}

    void foo()
    {
        cout << “Class Base foo()” << endl;
    }
};

class Derived_A : public Base
{
public:
    Derived_A(int val) : Base(val)
    {}

    void foo()
```

```
        {
                cout << "Class Derived_A foo()" << endl;
        }
};

class Derived_B : public Base
{
public:
        Derived_B(int val) : Base(val)
        {}

        void foo()
        {
                cout << "Class Derived_B foo()" << endl;
        }
};

class Derived_AB : public Derived_A, Derived_B
{
public:
        Derived_AB(int var1, int var2) : Derived_A(var1), Derived_B(var2)
        {}
};
```

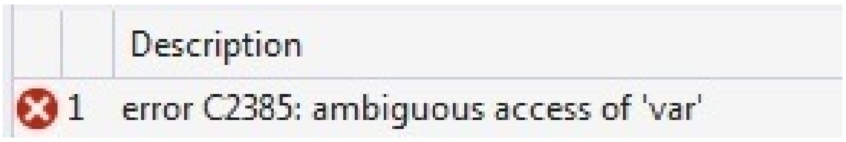
Pictorially this is what we have:



We have a *Base* class and then classes *Derived_A* and *Derived_B* inheriting from *Base* class. *Base* class function *foo* has been overridden by both derived classes. And then we have class *Derived_AB* inheriting from both *Derived_A* and *Derived_B* classes. You notice that this is nothing different from what we discussed in the earlier example. The only difference is that there wasn't a common base class. It's obvious where the diamond shape is coming from and this is a far more common occurrence, where two classes derive from the same base class and then another class doing multiple inheritance. You already know what needs to be done. You need to qualify your function call to let the compiler know which *foo* to call.

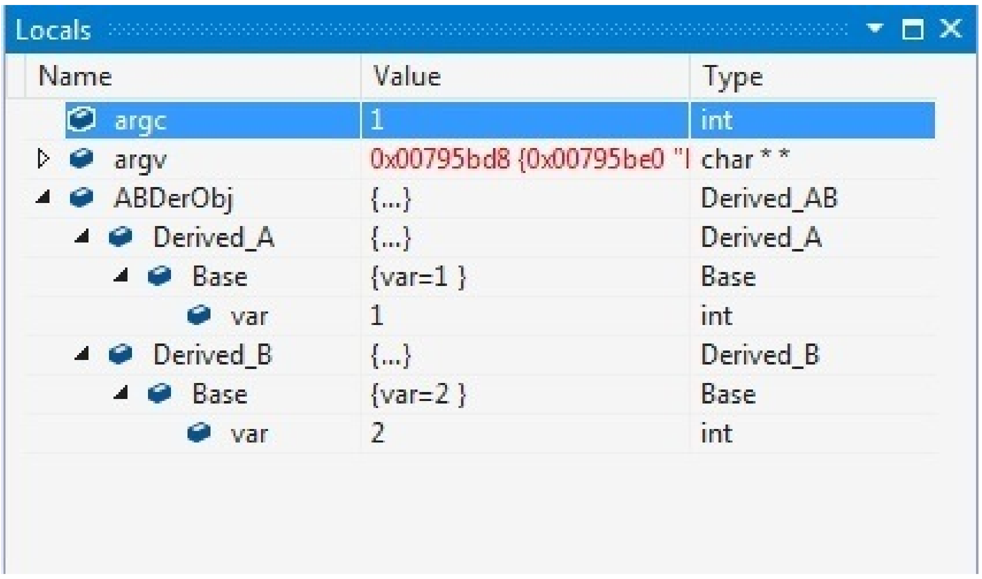
We are clear about calling *foo*. Both the derived classes have overridden *foo* functions so we need to explicitly qualify the call. But what about *var*? There is only *var* and it is in the *Base* class. Can there be any ambiguity in accessing *var*?

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

class Base
{
public:
        int var;

        Base(int val) : var(val)
        {}

        void foo()
        {
                cout << “Class Base foo()” << endl;
        }
};

class Derived_A : public Base
{
public:
        Derived_A(int val) : Base(val)
        {}

        void foo()
        {
                cout << “Class Derived_A foo()” << endl;
        }
};

class Derived_B : public Base
{
public:
        Derived_B(int val) : Base(val)
        {}
        void foo()
        {
                cout << "Class Derived_B foo()" << endl;
        }
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Derived_A(var1), Derived_B(var2)
        {}
};

int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        aBDerObj.var;
        return 0;
}
```



```
Description
1   error C2385: ambiguous access of 'var'
```

Yes. There still is ambiguity in *var* for the compiler. But why? Here's the debug view of the *Derived_AB* class object.



Locals

Name	Value	Type
argc	1	int
argv	0x00795bd8 {0x00795be0 "I	char * *
ABDerObj	{...}	Derived_AB
Derived_A	{...}	Derived_A
Base	{var=1 }	Base
var	1	int
Derived_B	{...}	Derived_B
Base	{var=2 }	Base
var	2	int

That should convince you why the compiler has ambiguity in finding *var*. There is only one *var* but there are two *Base* class instances. One in *Derived_A* and one in *Derived_B*. So there are two *var* variables in the *Derived_AB* object. So just as we did for the *foo* function call, we need to qualify *var* too.

```
…
…
int main(int argc, char** argv)
{
    Derived_AB ABDerObj(1, 2);
    cout << “var: “ << ABDerObj.Derived_B::var << endl;
    return 0;
}
```

```
var: 2
```

Could we qualify *var* with Base class? Like this:

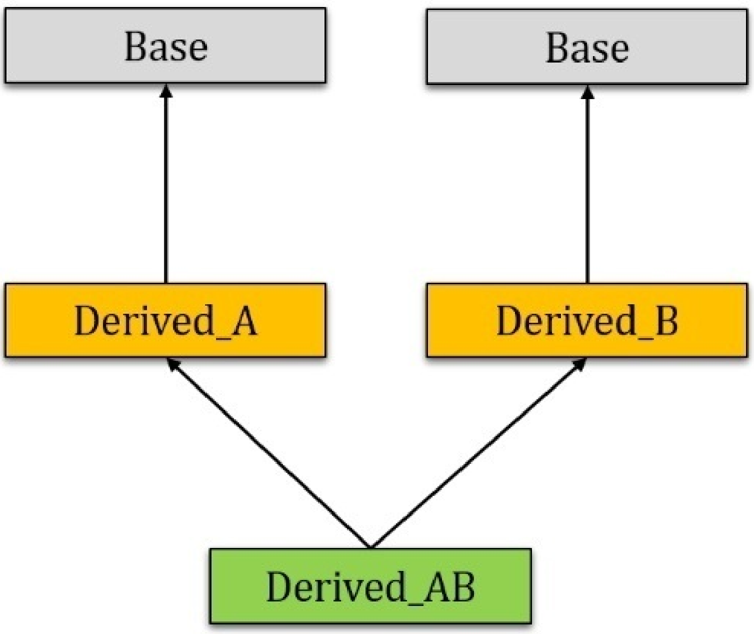
```
…
…
int main(int argc, char** argv)
{
    Derived_AB ABDerObj(1, 2);
    cout << “var: “ << ABDerObj.Base::var << endl;
    return 0;
}
```

```
var: 1
```

Well, it shouldn’t. Unfortunately it does compile and run in the version of Visual Studio I am running, although there is a tooltip warning that says “*base class Base is ambiguous*“. So it understands the ambiguity of the call yet does not even seem to warn, even at higher warning levels.

Base class function access

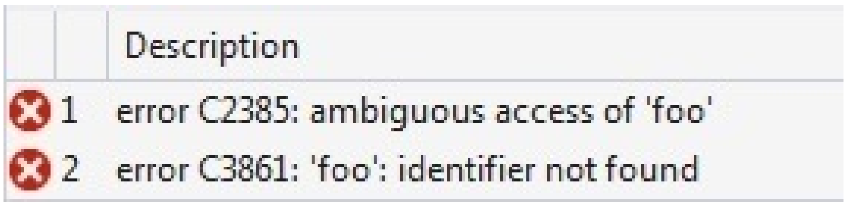
So you see, the real problem here is that our class hierarchy is actually like this:



We have two instances of *Base* and we need to explicitly tell the compiler which one we want to use. Now we understand this for *var*. There are two *var*'s, one with value 1 and the other 2. And we also understood about the ambiguity of *foo* call. There are two overridden versions. But what if we didn't override function *foo* in the derived classes? Like this:

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

class Base
{
public:
        int var;

        Base(int val) : var(val)
        {}

        void foo()
        {
                cout << "Class Base foo()" << endl;
        }
};

class Derived_A : public Base
{
public:
        Derived_A(int val) : Base(val)
        {}
};
```

```
class Derived_B : public Base
{
public:
        Derived_B(int val) : Base(val)
        {}
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Derived_A(var1), Derived_B(var2)
        {}
};

int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        aBDerObj.foo();
        return 0;
}
```



```
Description
1   error C2385: ambiguous access of 'foo'
2   error C3861: 'foo': identifier not found
```

There shouldn't be any ambiguity about function *foo*, right? Because *foo* is not overridden anywhere and *Base* class is the only place with *foo* implementation and functions are not instance specific. They are class specific. So there must be only one version of *foo*. Why is the call to *foo* still ambiguous then?

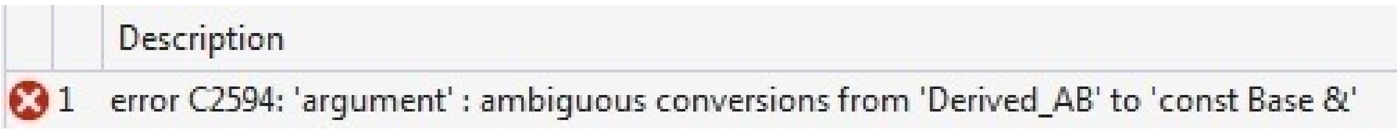
You see, there are actually two versions of *foo* the compiler can call. Remember that every member-function of a class has an implicit '**this*' argument passed by the compiler. That is why you can access object specific variables in member-functions. So the call to *foo* has an implicit '*Base**' argument passed to it by the compiler. The problem is our *Derived_AB* object has two *Base* sub-objects as we saw before. So which one should be passed to *foo*? This is where the compiler gets confused. So we need to explicitly specify which sub-object we want passed by qualifying the call.

Copy construction with multiple inheritance

Now these multiple base class instances are not only a problem for function and member variables accessing. It can be problematic in other places too, like shown here.

```
…
…
int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        Base BaseObj = static_cast< Base&> (ABDerObj);
        return 0;
}
```

Here we are trying to instantiate a *Base* object using a *Derived_AB* object. If there was no multiple inheritance this wouldn't have any issues. But there is ambiguity now.



```
Description
1   error C2594: 'argument' : ambiguous conversions from 'Derived_AB' to 'const Base &'
```

We are calling the *Base* class copy constructor, which takes an argument of type "*Base &*". The problem here is that the *Derived_AB* object has two instances of *Base* objects. Which one should the compiler use? As always, we need to explicitly tell it.

```
…
…
int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        Base BaseObj = static_cast<Derived_B&> (ABDerObj);
        return 0;
}
```

```
BaseObj.var: 2
```

We cast the object to a *Derived_B* instance. And you can see that the compiler indeed copied the correct *Base* instance, the one with *var* with a value of 2.

Virtual inheritance

Now there is a mechanism that avoids all of these messy qualified accesses and ambiguities. It's called *virtual inheritance*.

Let's see what exactly happens in our diamond class hierarchy when we instantiate a *Derived_AB* object.

```
#include<iostream>
```

```
using namespace std;

class Base
{
public:
        int var;

        Base(int val) : var(val)
        {
                cout << “Base constructor with var: “ << var << endl;
        }

        void foo()
        {
                cout << “Class Base foo()” << endl;
        }
};

class Derived_A : public Base
{
public:
        Derived_A(int val) : Base(val)
        {
                cout << “Derived_A constructor.” << endl;
        }
};

class Derived_B : public Base
{
public:
        Derived_B(int val) : Base(val)
        {
                cout << “Derived_B constructor.” << endl;
        }
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Derived_A(var1), Derived_B(var2)
        {
                cout << “Derived_AB constructor.” << endl;
        }
};

int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        return 0;
}
```

```
Base constructor with var: 1
Derived_A constructor.
Base constructor with var: 2
Derived_B constructor.
Derived_AB constructor.
```

Take a look at the sequence of object construction. First, the *Base* constructor with 1 is called by *Derived_A* constructor, then *Derived_A* constructor itself, then *Base* constructor is again called with 2 by *Derived_B* constructor, and then *Derived_B* constructor itself, and then finally *Derived_AB* constructor.

Order of base object construction

The constructor call sequence seems logical. It seems to be calling in the order we passed the arguments to *Derived_AB* constructor. *Derived_A* is initialized in the initializer list before *Derived_B*, so it seems that is the order in which the constructors are called. Let's change the order in the initializer list and see.

```
…
…
class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Derived_B(var1), Derived_A(var2) // initialization order changed
        {
                cout << "Derived_AB constructor." << endl;
        }
};
```

```
Base constructor with var: 2
Derived_A constructor.
Base constructor with var: 1
```

```
Derived_B constructor.
Derived_AB constructor.
```

Derived_A constructor is still called first. So it is not the order of initialization. Is it the order of derivation, then?

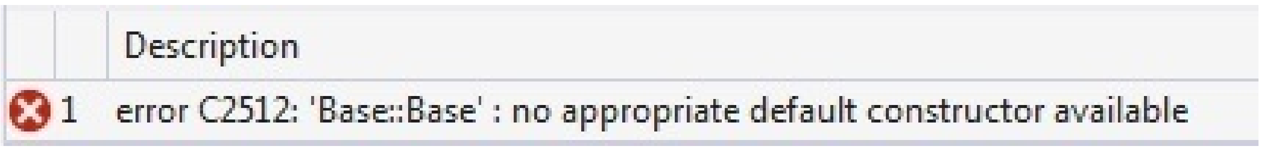
```
…
…
class Derived_AB : public Derived_B, public Derived_A   // Derivation order changed
{
public:
	Derived_AB(int var1, int var2) : Derived_A(var1), Derived_B(var2)
	{
		cout << “Derived_AB constructor.” << endl;
	}
};
```

```
Base constructor with var: 2
Derived_B constructor.
Base constructor with var: 1
Derived_A constructor.
Derived_AB constructor.
```

Yes, it is. The order of constructor invocation depends on the order of derivation.

So we have now confirmed that in multiple inheritance, where there is a shared base class, this base class gets constructed multiple times. And these multiple base class sub-objects are the source of many ambiguities for the compiler. C++ has the virtual inheritance mechanism to circumvent this situation. All you need to do is specify *‘virtual’* inheritance. Let’s redo our example with virtual inheritance.

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

class Base
{
public:
	int var;

	Base(int val) : var(val)
	{
        cout << “Base constructor with var: “ << var << endl;
    }

    void foo()
    {
        cout << “Base foo()” << endl;
    }
};

class Derived_A : public virtual Base       // virtual inheritance
{
public:
    Derived_A(int val) : Base(val)
    {
        cout << “Derived_A constructor.” << endl;
    }
};

class Derived_B : public virtual Base       // virtual inheritance
{
public:
    Derived_B(int val) : Base(val)
    {
        cout << “Derived_B constructor.” << endl;
    }
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
    Derived_AB(int var1, int var2) : Derived_A(var1), Derived_B(var2)
    {
        cout << “Derived_AB constructor.” << endl;
    }
};

int main(int argc, char** argv)
{
    Derived_AB ABDerObj(1, 2);
    return 0;
}
```


Description

1 error C2512: 'Base::Base' : no appropriate default constructor available

What is going on here? Why is the compiler trying to call the default constructor of *Base* class? So obviously just deriving virtually is not enough.

Let’s understand what virtual inheritance does. What it does is to have only one base class sub-object. We saw earlier that when we didn’t have virtual inheritance we called constructors of both *Derived_A* and *Derived_B*. These constructors in turn call the *Base* class constructor. So *Base* class gets constructed twice. And this is what we want to avoid.

Under virtual inheritance, the virtual base class, in our case *Base* class constructor, should not be called by the classes that inherit virtually from it. That means *Derived_A* and *Derived_B* classes’ constructors should not be calling the *Base* class constructor, because, that would mean that *Base* class will be constructed multiple times. Instead, with virtual inheritance, the final concrete class has the responsibility of calling the virtual base class’s constructor. Concrete class means the class that we are instantiating an object with, which in our case is *Derived_AB*. So the compiler expects the *Derived_AB* constructor to call the *Base* class constructor. And since we are not making a call to *Base* class constructor in *Derived_AB* constructor, the compiler tries to call the default constructor of *Base*, which it does not have, and hence the compiler error. Let’s implement a default constructor to *Base* class and see the result.

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

class Base
{
public:
	int var;

	Base()
	{
		cout << “Base default constructor.” << endl;
	}
	Base(int val) : var(val)
	{
		cout << “Base constructor with var: “ << var << endl;
	}

	void foo()
	{
		cout << “Base foo()” << endl;
```

```
    }
};

class Derived_A : public virtual Base
{
public:
    Derived_A(int val) : Base(val)
    {
        cout << “Derived_A constructor.” << endl;
    }
};

class Derived_B : public virtual Base
{
public:
    Derived_B(int val) : Base(val)
    {
        cout << “Derived_B constructor.” << endl;
    }
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
    Derived_AB(int var1, int var2) : Derived_A(var1), Derived_B(var2)
    {
        cout << “Derived_AB constructor.” << endl;
    }
};

int main(int argc, char** argv)
{
    Derived_AB ABDerObj(1, 2);
    return 0;
}
```

```
Base default constructor.
Derived_A constructor.
Derived_B constructor.
Derived_AB constructor.
```

It's working. The compiler implicitly makes a call to the *Base* class default constructor. And then *Derived_A* and *Derived_B* constructors are called. Did you notice something about the *Derived_A* and *Derived_B* constructor calls? They are not calling the *Base* class constructor anymore. Usually the derived class constructor always calls its base class constructor before executing its own constructor. But things happen a little differently under virtual inheritance. We discussed earlier how derived classes should not be calling the base class constructors as that would mean multiple base class objects. So when in virtual inheritance, the compiler avoids the calls to the base class constructor from the derived classes. The virtual base class constructor must only be called by the concrete class. Note that this is the only scenario where a derived class is allowed to call the constructor of a class which is not its immediate base class.

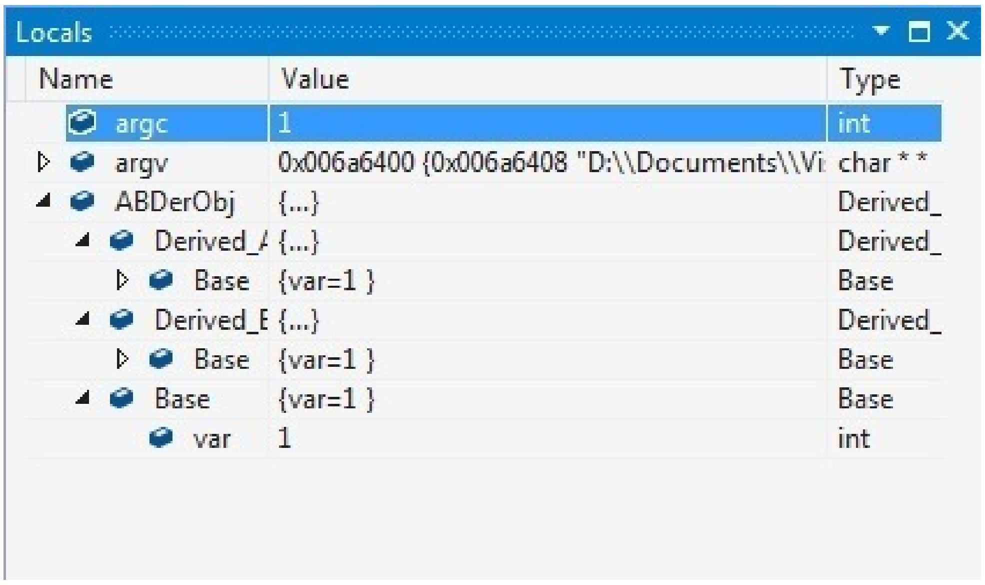
So our original example should be called as follows. Note the call to *Base* class constructor in the *Derived_AB* constructor.

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

class Base
{
public:
        int var;

        Base(int val) : var(val)
        {
                cout << "Base constructor with var: " << var << endl;
        }

        void foo()
        {
                cout << "Base foo()" << endl;
        }
};

class Derived_A : public virtual Base
{
public:
        Derived_A(int val) : Base(val)
        {
                cout << "Derived_A constructor." << endl;
        }
};
```

```
class Derived_B : public virtual Base
{
public:
        Derived_B(int val) : Base(val)
        {
                cout << “Derived_B constructor.” << endl;
        }
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Base(1), Derived_A(var1), Derived_B(var2)
        {
                cout << “Derived_AB constructor.” << endl;
        }
};

int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        aBDerObj.var;
        aBDerObj.foo();
        return 0;
}
```

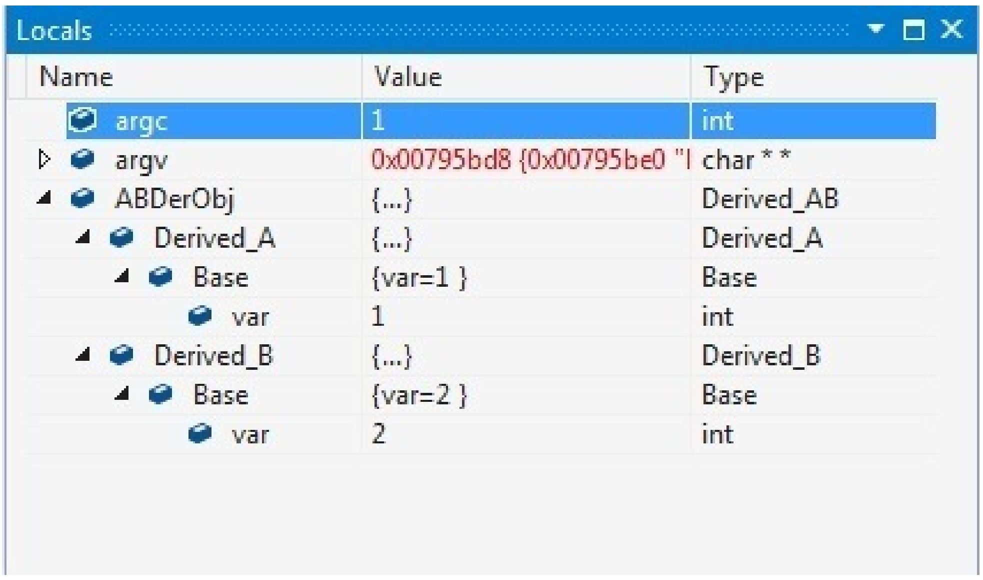
```
Base constructor with var: 1
Derived_A constructor.
Derived_B constructor.
Derived_AB constructor.
Base foo()
```

No more ambiguities for the compiler. Take a look at the object structure now.



Name	Value	Type
argc	1	int
▷ argv	0x006a6400 {0x006a6408 "D:\\Documents\\Vi	char * *
◢ ABDerObj	{...}	Derived_
◢ Derived_/	{...}	Derived_
▷ Base	{var=1 }	Base
◢ Derived_E	{...}	Derived_
▷ Base	{var=1 }	Base
◢ Base	{var=1 }	Base
var	1	int

Compare it with the object structure we had before (shown below) without virtual inheritance.



Name	Value	Type
argc	1	int
▷ argv	0x00795bd8 {0x00795be0 "I	char * *
◢ ABDerObj	{...}	Derived_AB
◢ Derived_A	{...}	Derived_A
◢ Base	{var=1 }	Base
var	1	int
◢ Derived_B	{...}	Derived_B
◢ Base	{var=2 }	Base
var	2	int

With multiple inheritance you can see a separate *Base* class sub-object directly under the *Derived_AB* object. Although this view shows there are *Base* class objects under *Derived_A* and *Derived_B*, there really aren't. There is only one *Base* sub-object. What this structure depicts is that this *Base* sub-object logically is inside *Derived_A* and *Derived_B*. That means we can access *Base* class as we did earlier when we didn't do virtual inheritance and had multiple *Base* sub-objects.

```
…
…
int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        aBDerObj.Derived_A::foo();
        aBDerObj.Derived_A::var;
        aBDerObj.Derived_B::foo();
        aBDerObj.Derived_B::var;
        return 0;
}
```

```
Base constructor with var: 1
Derived_A constructor.
Derived_B constructor.
Derived_AB constructor.
Base foo()
Base foo()
```

Although we are qualifying the accesses we are effectively accessing the same *Base* sub-object.

This is the core of virtual inheritance. The mechanism is pretty simple. Let's finish this topic by looking at some different cases.

Function overriding in one derived class

Here we override *foo* in *Derived_B* class only.

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

class Base
{
public:
        int var;

        Base(int val) : var(val)
        {}

        void foo()
        {
                cout << "Base foo()" << endl;
        }
};

class Derived_A : public virtual Base
{
public:
        Derived_A(int val) : Base(val)
        {}
};

class Derived_B : public virtual Base
{
public:
        Derived_B(int val) : Base(val)
        {}

        void foo()
        {
                cout << “Derived_B foo()” << endl;
        }
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Base(1), Derived_A(var1), Derived_B(var2)
        {}
};

int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        aBDerObj.foo();
        aBDerObj.Derived_A::foo();
        aBDerObj.Derived_B::foo();
        return 0;
}
```

```
Derived_B foo()
Base foo()
Derived_B foo()
```

When we don’t qualify the function call it calls the overridden function, which is what we want. When we qualify the call to *Derived_A*, then it calls the *Base* class function, which again is what we want. *Derived_A* hasn’t overridden *foo* so *Base::foo* needs to be called.

Instantiating a virtually derived class.

What effect would virtual inheritance have if we instantiated a *Derived_A* or *Derived_B* class?

```
…
…
int main(int argc, char** argv)
{
        Derived_A derAobj(1);
        Derived_B derBobj(2);
        derAobj.foo();
        derBobj.foo();
        return 0;
}
```

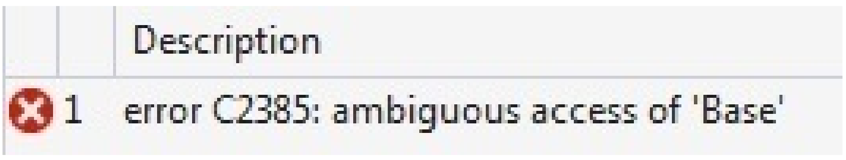
```
Base constructor with var: 1
Derived_A constructor.
Base constructor with var: 2
Derived_B constructor.
Class Base foo()
Derived_B foo()
```

As you see, virtual inheritance has no effect when we instantiate directly derived classes. They behave normally.

Only one class with virtual inheritance.

What would happen if we had only *Derived_A* with virtual inheritance?

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

class Base
{
public:
        int var;

        Base(int val) : var(val)
        {}
};

class Derived_A : public virtual Base
{
public:
        Derived_A(int val) : Base(val)
        {}
};

class Derived_B : public Base
{
public:
        Derived_B(int val) : Base(val)
        {}
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Base(1), Derived_A(var1), Derived_B(var2)
        {}
};

int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        return 0;
}
```

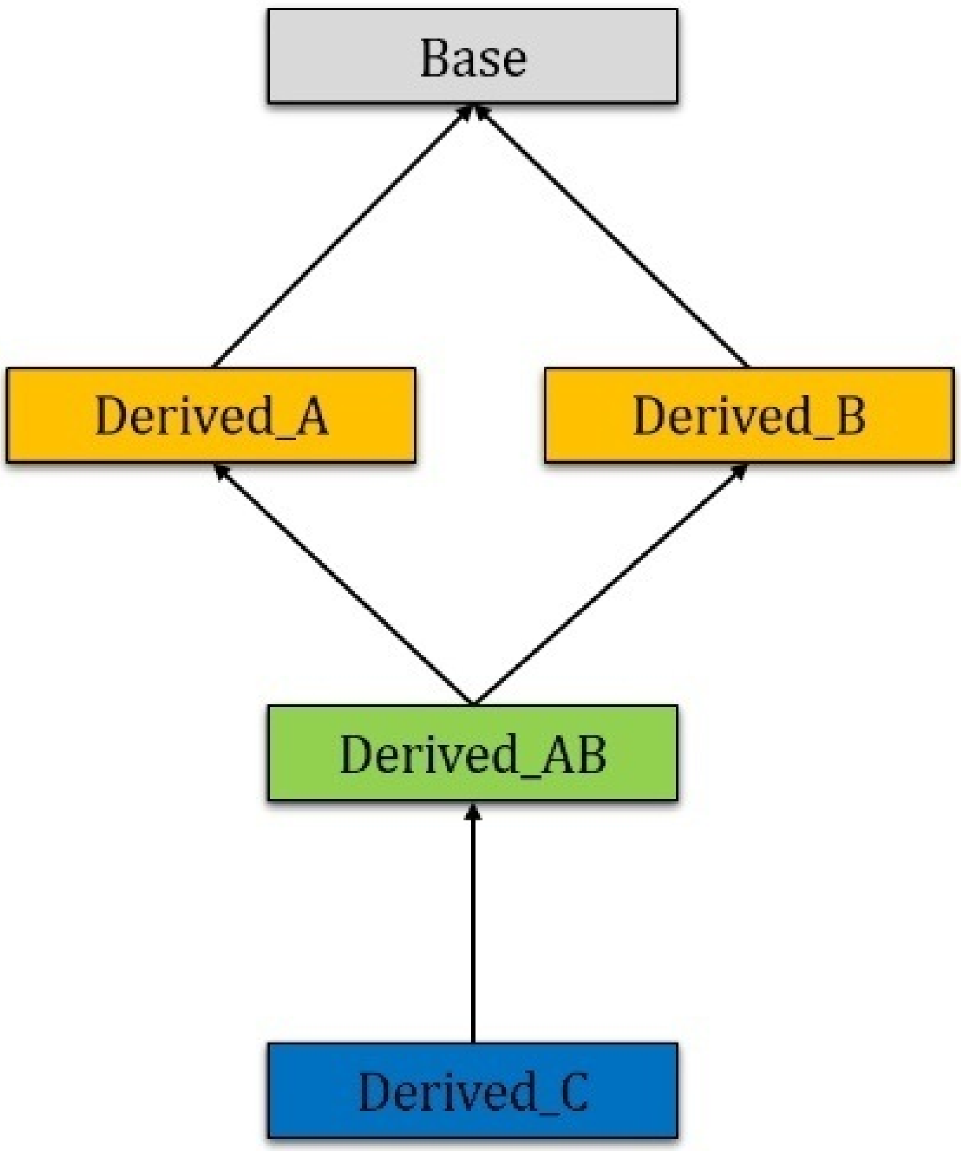


```
    Description
1   error C2385: ambiguous access of 'Base'
```

It doesn’t work. You need to have all of your inheriting classes deriving from *Base* to have virtual inheritance.

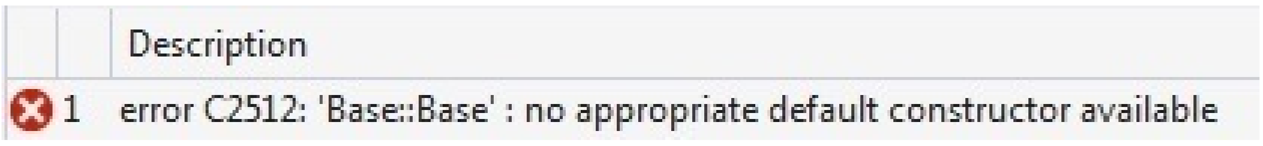
More levels of derivation.

Let’s see how things change when you have one more derived class. This is our class hierarchy now.



Since we are calling the virtual base class constructor in *Derived_AB*, we probably don’t need to do anything special in *Derived_C*. So let’s go ahead and try.

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

class Base
{
public:
        int var;

        Base(int val) : var(val)
        {}
};

class Derived_A : public virtual Base
{
public:
        Derived_A(int val) : Base(val)
        {}
};

class Derived_B : public virtual Base
{
public:
        Derived_B(int val) : Base(val)
        {}
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Base(1), Derived_A(var1), Derived_B(var2)
        {} // calling virtual Base constructor here
};

class Derived_C : public Derived_AB
{
public:
        Derived_D(int var1, int var2) : Derived_AB(var1, var2)
        {}
};

int main(int argc, char** argv)
{
        Derived_C CDerObj(1, 2);
        return 0;
}
```



```
Description
1   error C2512: 'Base::Base' : no appropriate default constructor available
```

The compiler is looking for the default constructor of *Base*. This is the same thing that happened earlier when we didn't call the *Base* class constructor in the *Derived_AB* constructor. So the compiler expects *Derived_C* to call the virtual base class constructor. Let's add the constructor call:

```
…
…
```

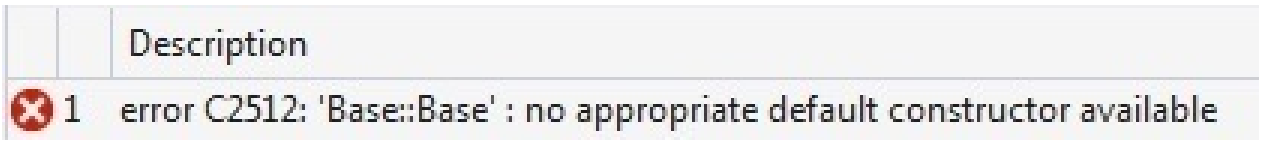
```
class Derived_C : public Derived_AB
{
public:
        Derived_C(int var1, int var2) : Base(10), Derived_AB(var1, var2)
        {}
};

int main(int argc, char** argv)
{
        Derived_C CDerObj(1, 2);
        cout << “CDerObj.var: “ << CDerObj.var << endl;
        return 0;
}
```

```
CDerObj.var: 10
```

This works fine. What does this mean, then?

It means that if you have virtual inheritance in your class hierarchy, you must call the virtual base class constructor from the most derived class. So can we remove the call to *Base* constructor from *Derived_AB* then? Nope.

```
…
…
class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Derived_A(var1), Derived_B(var2) // No call to Base constructor
        {}
};

class Derived_C : public Derived_AB
{
public:
        Derived_C(int var1, int var2) : Base(10), Derived_AB(var1, var2)
        {}
};

int main(int argc, char** argv)
{
        Derived_C CDerObj(1, 2);
        cout << “CDerObj.var: “ << CDerObj.var << endl;
        return 0;
}
```



```
Description
1   error C2512: 'Base::Base' : no appropriate default constructor available
```

Even though you are not instantiating a *Derived_AB* object, if your class is inheriting from a virtual derived class, you need to call the virtual base class constructor.

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

class Base
{
public:
        int var;

        Base(int val) : var(val)
        {}
};

class Derived_A : public virtual Base
{
public:
        Derived_A(int val) : Base(val)
        {}
};

class Derived_B : public virtual Base
{
public:
        Derived_B(int val) : Base(val)
        {}
};

class Derived_AB : public Derived_A, public Derived_B
{
public:
        Derived_AB(int var1, int var2) : Base(10), Derived_A(var1), Derived_B(var2)
        {}
};

class Derived_C : public Derived_AB
{
public:
        Derived_C(int var1, int var2) : Base(20), Derived_AB(var1, var2)
        {}
};

int main(int argc, char** argv)
{
        Derived_AB ABDerObj(1, 2);
        cout << “ABDerObj.var: “ << ABDerObj.var << endl;
        Derived_C CDerObj(3, 4);
        cout << “CDerObj.var: “ << CDerObj.var << endl;
        return 0;
}
```

```
ABDerObj.var: 10
CDerObj.var: 20
```

See, you need the call to *Base* in *Derived_AB* because if you are instantiating a *Derived_AB* object, then you need that call. You don’t need that for a *Derived_C* object. But the compiler doesn’t know which classes you are planning to instantiate so it wants you to call virtual base constructor if it is needed.

So that is virtual inheritance.

Topic 18

Casting

C++ introduced four types of casts. These casts fundamentally do the same casting functionality as C casts, but there are a few differences. Casts also have quite a bit of special usages, but in this topic I will limit the discussion to the most used types of castings.

Here are the four casts:

- Static cast
- Dynamic cast
- Reinterpret cast
- Const cast

Before we start discussing the casts it is important to define *upcast* and *downcast*. Upcast is when casting up the hierarchy. That is, casting to a base type from a derived type. Downcasting is casting to a derived type from a base type.

Static cast

Static cast is used in many implicit castings. Implicit casts are automatically done by the compiler and explicitly stating the cast is not necessary. But it is often good practice to explicitly cast when you want to let that fact be known. An upcast is implicit and does not require a static cast. But a downcast does.

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

class Base {

	int Base_ID;
public:
	Base(int val) : Base_ID(val)
	{}

	void func()
	{
		cout << “Base class func. ID: “ << Base_ID << endl;
	}
};
```

```
class Derived : public Base {

    int Derived_ID;
public:
    Derived(int baseID, int derivedID) : Base(baseID), Derived_ID(derivedID)
    {}
    void func()
    {
        cout << “Derived class func. ID: “ << Derived_ID << endl;
    }
};

int main(int argc, char** argv)
{
    Base* basePtr1 = new Base(1);
    Derived* derivedPtr1 = new Derived(2, 3);

    basePtr1->func();
    derivedPtr1->func();

    Base* basePtr2 = static_cast<Base*> (derivedPtr1);
    Derived* derivedPtr2 = static_cast<Derived*> (basePtr1);

    basePtr2->func();
    derivedPtr2->func();
    return 0;
}
```

```
Base class func. ID: 1
Derived class func. ID: 3
Base class func. ID: 2
Derived class func. ID: -33686019
```

Here we have a *Base* class and a *Derived* class. They both have implemented function *func* but note that this is not virtual. First, we define two pointers, one of type *Base* and the other *Derived*, and assign *Base* and *Derived* instances respectively. Then we call *func* through these pointers. They call the correct functions. Then we define two new pointers of types *Base* and *Derived*. This time we assign the *Derived* pointer to the new Base pointer and the *Base* pointer to the new *Derived* pointer and the call *func*.

First note that there are no compiler errors. Assigning *derivedPtr2* with *basePtr1* is not a proper assignment. A *Derived* instance has a *Base* instance inside and has two int variables. But the instance pointed to by *basePtr1* is only a *Base* object. So *derivedPtr2* is pointing to an incomplete *Derived* object, although the compiler believes it is pointing to a proper *Derived* object. Calling *func* through *basePtr2* works as expected. It prints the correct *Base_ID*. But *func* does not work well with the *derivedPtr2*. We know it is pointing to a *Base* object. Two things to note here. First, although *derivedPtr2* is pointing to a *Base* object, the invoked *func* correctly calls *Derived* version of *func*. Second, the printed value of *func* is a garbage value.

The reason why the Derived version of *func* is called is because a function is not part of the object. Remember that we discussed that a function resides outside of an object and is tied to the class type. So when we called *func* through a *Derived* pointer the compiler actually called the *Derived::func*. It didn't matter that the object it was pointing to is of type Base. The object does not have the implementation of *func*. But the object is passed to the function by the compiler. Every non-static member-function is passed the '**this*' by the compiler implicitly. So when we called 'func' through Derived type pointer, the compiler called Derived::func and passed it the *Base* object it is pointing to. And then the function tried to access *Derived_ID* from that object. Remember we discussed how each member variable of a class has an offset? The function *func* simply accessed a memory location at a particular offset, which is supposed to be the location of *Derived_ID*. But it is passed an object of *Base*, and as we have seen before, must be smaller than that of a *Derived* object. The offset for *Derived_ID* is past the memory block belonging to *basePtr1* and that is why we are seeing garbage values.

Now how would the behavior change if func was virtual? (I know we are going out of topic, but this is important to know.)

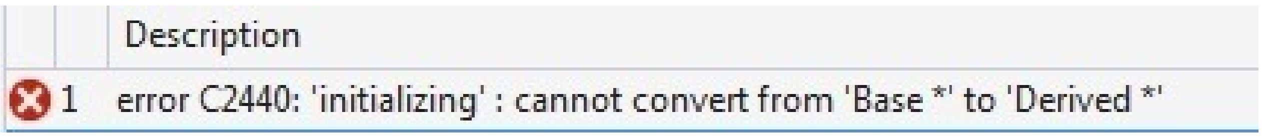
```
class Base {

    int Base_ID;
public:
    Base(int val) : Base_ID(val)
    {}

    virtual void func()
    {
        cout << “Base class func. ID: “ << Base_ID << endl;
    }
};
…
…
```

```
Base class func. ID: 1
Derived class func. ID: 3
Derived class func. ID: 3
Base class func. ID: 1
```

We are not seeing any problems here, although it may not be doing what we'd expect. There is no effect of virtualness for the first two func calls. The third func call is a classic example of virtual mechanism. Calling a derived class overridden function through a base class pointer. The last call, although it is invoking the correct function, is not the usual way of using the virtual mechanism. As you know, when a function is virtual the compiler refrains from binding the function call during compilation. The function to call is determined during runtime. Unlike when *func* was not virtual, in this case the function to call is determined through the *vtable*. And we've seen, the *vtable* is object specific. And in our case, *basePtr1* points to an *Base* object, whose *vtable* has the implementation of *Base::func*. So the *Base* version of *func* is called and it is passed an object of *Base*. This is same as before, but now the *func* is accessing *Base_ID*, not *Derived_ID*, so the offset is fine.

What if we omitted the static_casts?

```
…
…
int main(int argc, char** argv)
{
        Base* basePtr1 = new Base(1);
        Derived* derivedPtr1 = new Derived(2, 3);

        basePtr1->func();
        derivedPtr1->func();

        Base* basePtr2 = (derivedPtr1);             // no casting
        Derived* derivedPtr2 = (basePtr1);         // no casting

        basePtr2->func();
        derivedPtr2->func();
        return 0;
}
```


Description

1 error C2440: 'initializing' : cannot convert from 'Base *' to 'Derived *'

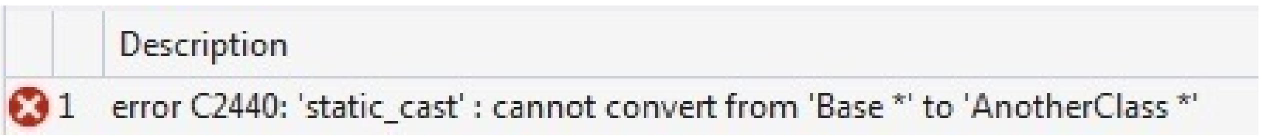
So you see, upcasting is implicit. You do not need to static_cast it. But downcast does. This is because the compiler knows that downcasting a base class object to a derived class type can be trouble. So it doesn't do it implicitly. But with a static_cast, we are explicitly letting the compiler know that we know what we are doing. Because *basePtr1* could very well be pointing to a *Derived* instance. Like this:

```
…
…
int main(int argc, char** argv)
{
        Base* basePtr1 = new Derived(4, 5); // Derived object
        Derived* derivedPtr1 = new Derived(2, 3);

        basePtr1->func();
        derivedPtr1->func();

        Base* basePtr2 = static_cast<Base*> (derivedPtr1);
        Derived* derivedPtr2 = static_cast<Derived*> (basePtr1);

        basePtr2->func();
        derivedPtr2->func();
        return 0;
}
```

```
Base class func. ID: 4
Derived class func. ID: 3
Base class func. ID: 2
Derived class func. ID: 5
```

This type checking is an important part of static_cast. It does not do any type checking at compile time or runtime. That is why we were able to cast a *Base* instance to a *Derived* pointer. But does static_cast ignore all checks?

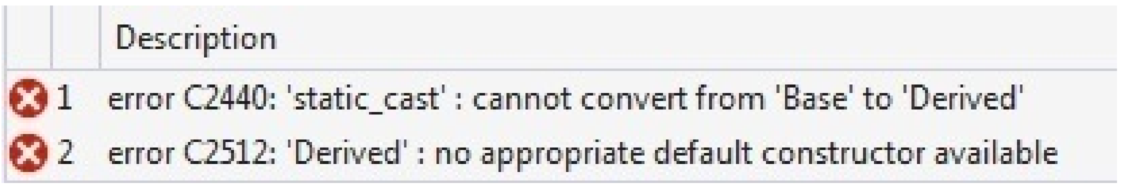
```
#include<iostream>
using namespace std;

class Base {

        int ID;
public:
        Base(int val) : ID(val)
        {}

        void func()
        {
                cout << “Base class func. ID: “ << ID << endl;
        }
};

class Derived : public Base {

        int ID;
public:
        Derived(int baseID, int derivedID) : Base(baseID), ID(derivedID)
        {}
        void func()
        {
                cout << “Derived class func. ID: “ << ID << endl;
        }
};

class AnotherClass {
public:
        void func()
        {
                cout << “AnotherClass func.” << endl;
        }
};

int main(int argc, char** argv)
{
        Base* basePtr1 = new Base(1);
        Derived* derivedPtr1 = new Derived(2, 3);
        AnotherClass* anotherPtr = new AnotherClass;

        anotherPtr = static_cast<AnotherClass*> (basePtr1);
        return 0;
}
```



```
Description
1   error C2440: 'static_cast' : cannot convert from 'Base *' to 'AnotherClass *'
```

Here we tried to cast a *Base* pointer to an *AnotherClass* pointer and the compiler isn’t happy with that. Although we were able to cast a base instance to a derived pointer, static_cast isn’t all that naive. It doesn’t let us cast between different types. And it also does not let us downcast when using objects:

```
…
…
int main(int argc, char** argv)
{
        Base baseObj1(1);
        Derived derivedObj1(2, 3);

        baseObj1.func();
        derivedObj1.func();

        Base baseObj2 = static_cast<Base>(derivedObj1);
        Derived derivedObj2 = static_cast<Derived>(baseObj1);

        baseObj2.func();
        derivedObj2.func();
        return 0;
}
```



```
     Description
1    error C2440: 'static_cast' : cannot convert from 'Base' to 'Derived'
2    error C2512: 'Derived' : no appropriate default constructor available
```

But just as with pointers, we can do the downcast with references.

```
…
…
int main(int argc, char** argv)
{
        Base baseObj1(1);
        Derived derivedObj1(2, 3);

        Base& baseRef1 = baseObj1;
        Derived& derivedRef1 = derivedObj1;

        baseRef1.func();
        derivedRef1.func();

        Base& baseRef2 = static_cast<Base&>(derivedRef1);
        Derived& derivedRef2 = static_cast<Derived&>(baseRef1);
```

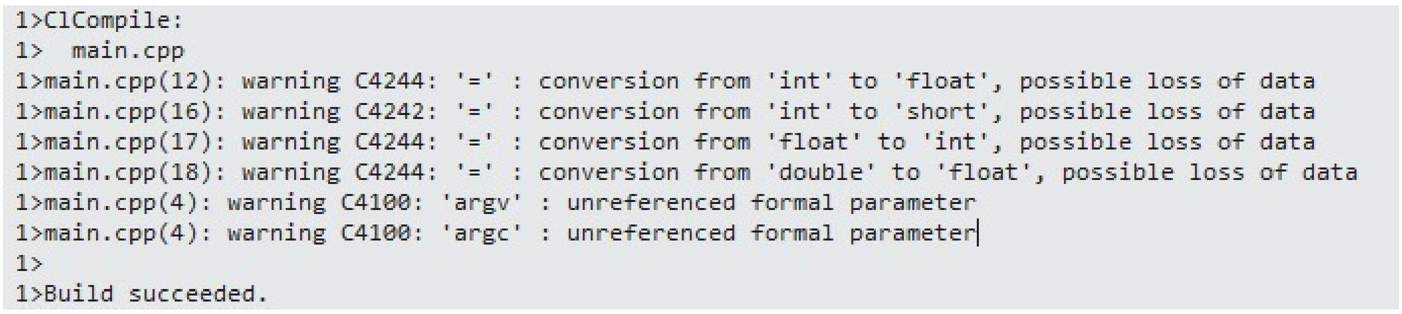
```
    baseRef2.func();
    derivedRef2.func();
    return 0;
}
```

```
Base class func. ID: 1
Derived class func. ID: 3
Base class func. ID: 2
Derived class func. ID: -858993460
```

So when you are using pointers and references we can downcast with static_cast and force the compiler to accept the type but not with objects.

As we saw before with downcasting, explicitly casting with a static_cast is a way of telling the compiler that you intended to do what you did. This works not only for class types, but also for built in types.

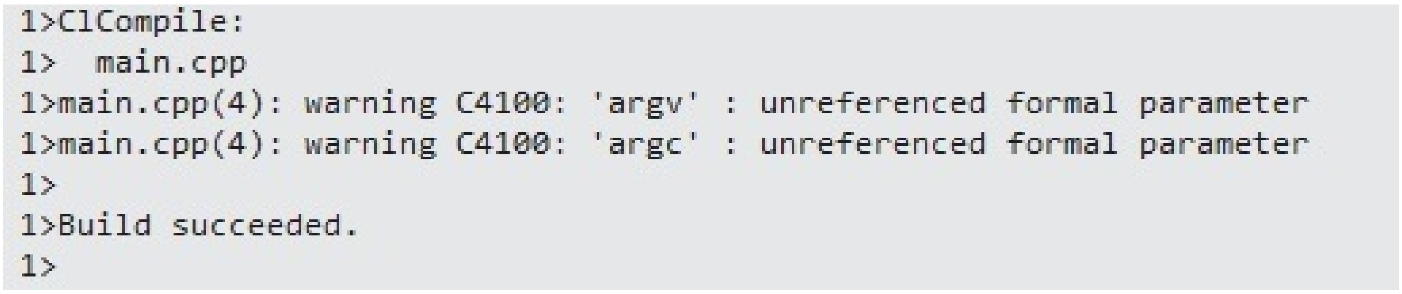
```
int main(int argc, char** argv)
{
    short shortVar = 1;
    int intVar = 2;
    float floatVar = 3.0f;
    double doubleVar = 4;

    intVar = shortVar;
    floatVar = intVar;
    doubleVar = floatVar;

    shortVar = intVar;
    intVar = floatVar;
    floatVar = doubleVar;

    return 0;
}
```



```
1>ClCompile:
1>  main.cpp
1>main.cpp(12): warning C4244: '=' : conversion from 'int' to 'float', possible loss of data
1>main.cpp(16): warning C4242: '=' : conversion from 'int' to 'short', possible loss of data
1>main.cpp(17): warning C4244: '=' : conversion from 'float' to 'int', possible loss of data
1>main.cpp(18): warning C4244: '=' : conversion from 'double' to 'float', possible loss of data
1>main.cpp(4): warning C4100: 'argv' : unreferenced formal parameter
1>main.cpp(4): warning C4100: 'argc' : unreferenced formal parameter
1>
1>Build succeeded.
```

The code above compiles with no errors (although all values end up just being 1 due to chain assignment), but as you see, the compiler does give out some warnings. Here we have four variables with different magnitudes. The compiler warns about possible loss of data during conversions. Usually there is no loss of data when the variable is promoted. But when an int is promoted to a float it can lose a bit of precision. But this loss of precision is very small compared to the loss of data that could happen when the values are truncated, for example when converting from int to short. Although the loss of precision or data cannot be avoided during conversion, using static_cast will help us get rid of the warnings.

```
int main(int argc, char** argv)
{
	short shortVar = 1;
	int intVar = 2;
	float floatVar = 3.0f;
	double doubleVar = 4;

	intVar = shortVar;
	floatVar = static_cast<float> (intVar);
	doubleVar = floatVar;

	shortVar = static_cast<short> (intVar);
	intVar = static_cast<int> (floatVar);
	floatVar = static_cast<float> (doubleVar);

	return 0;
}
```



```
1>ClCompile:
1>  main.cpp
1>main.cpp(4): warning C4100: 'argv' : unreferenced formal parameter
1>main.cpp(4): warning C4100: 'argc' : unreferenced formal parameter
1>
1>Build succeeded.
1>
```

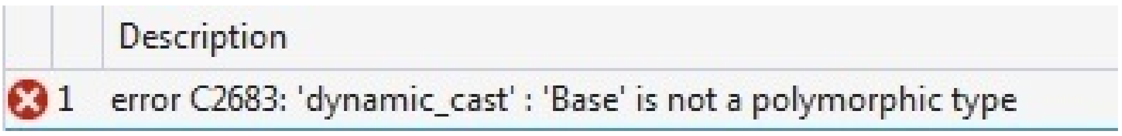
No warnings here (I have all of the warnings turned on here with ‘-Wall’).

Dynamic cast

Dynamic cast is similar to static cast in the sense that you can do both upcast and downcast, but it differs in one important way. Dynamic cast does check the types at compile time and runtime. As we will see, dynamic cast is a way of finding out the real type of a pointer or a reference. But there is a downside to using dynamic cast. That runtime checking comes at a a cost of a performance hit because the type needs to be checked at runtime. Let's start with the same example.

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

class Base {

	int Base_ID;
public:
	Base(int val) : Base_ID(val)
	{}

	void func()
	{
		cout << “Base class func. ID: “ << Base_ID << endl;
	}
};


class Derived : public Base {

	int Derived_ID;
public:
	Derived(int baseID, int derivedID) : Base(baseID), Derived_ID(derivedID)
	{}
	void func()
	{
		cout << “Derived class func. ID: “ << Derived_ID << endl;
	}
};

int main(int argc, char** argv)
{
	Base* basePtr1 = new Base(1);
	Derived* derivedPtr1 = new Derived(2, 3);
```

```
    basePtr1->func();
    derivedPtr1->func();

    Base* basePtr2 = dynamic_cast<Base*> (derivedPtr1);
    Derived* derivedPtr2 = dynamic_cast<Derived*> (basePtr1);

    basePtr2->func();
    derivedPtr2->func();
    return 0;
}
```



```
Description
1   error C2683: 'dynamic_cast' : 'Base' is not a polymorphic type
```

The compiler doesn’t let us cast *basePtr1* to a Derived pointer because *Base* is not a polymorphic type. What the compiler is basically complaining about is that *Base* does not have any virtual functions. Polymorphism is used through virtual functions. So to downcast with dynamic_cast we first need the class type to be polymorphic. Let’s make *func* virtual.

```
class Base {

    int Base_ID;
public:
    Base(int val) : Base_ID(val)
    {}

    virtual void func()
    {
        cout << “Base class func. ID: “ << Base_ID << endl;
    }
};
…
…
```

This time the code compiles fine and runs but you will most definitely get a runtime error. This is because we are dereferencing a NULL pointer. Let’s see this.

```
…
…
int main(int argc, char** argv)
```

```
{
        Base* basePtr1 = new Base(1);
        Derived* derivedPtr1 = new Derived(2, 3);

        basePtr1->func();
        derivedPtr1->func();

        Base* basePtr2 = dynamic_cast<Base*> (derivedPtr1);
        Derived* derivedPtr2 = dynamic_cast<Derived*> (basePtr1);

        if (basePtr2 == NULL)
        {
                cout << "basePtr2 == NULL" << endl;
        }
        else
        {
                basePtr2->func();
        }

        if (derivedPtr2 == NULL)
        {
                cout << "derivedPtr2 == NULL" << endl;
        }
        else
        {
                derivedPtr2->func();
        }
        return 0;
}
```

```
Base class func. ID: 1
Derived class func. ID: 3
Derived class func. ID: 3
derivedPtr2 == NULL
```

See, *derivedPtr2* was NULL and we tried to dereference it. That's why the runtime error.

But why was *derivedPtr2* NULL? This is because of the checking dynamic_cast does at runtime. Dynamic cast checks if the object pointed to by *basePtr1* is indeed of type *Derived*, and if not returns a NULL pointer.

So you see, dynamic casting checks the casting type against the object to make sure the cast is valid. And this is done at runtime and uses the object's *typeid* to determine the type. Now let's see what happens with references (Hint: there's a surprise).

```
…
…
int main(int argc, char** argv)
{
        Base baseObj1(1);
        Derived derivedObj1(2, 3);

        Base& baseRef1 = baseObj1;
        Derived& derivedRef1 = derivedObj1;

        baseRef1.func();
        derivedRef1.func();

        Base& baseRef2 = dynamic_cast<Base&>(derivedRef1);
        Derived& derivedRef2 = dynamic_cast<Derived&>(baseRef1);

        baseRef2.func();
        derivedRef2.func();
        return 0;
}
```

This code too compiles and runs but will crash with an unhandled exception. You see, whereas a NULL pointer is returned when casting pointers, with references it throws an exception. Let's try to catch it.

```
…
…
int main(int argc, char** argv)
{
        Base baseObj1(1);
        Derived derivedObj1(2, 3);

        Base& baseRef1 = baseObj1;
        Derived& derivedRef1 = derivedObj1;

        baseRef1.func();
        derivedRef1.func();
```

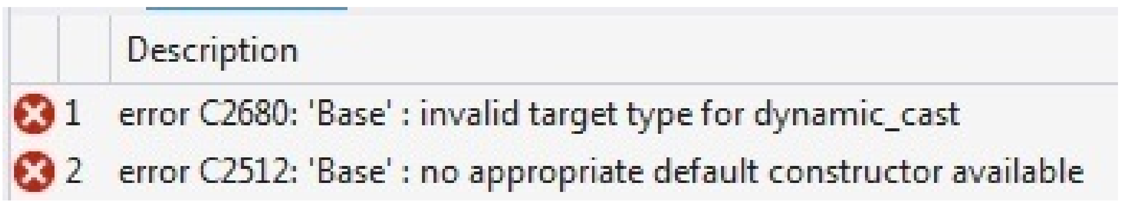
```
	try
	{
		base& baseRef2 = dynamic_cast<Base&>(derivedRef1);
		Derived& derivedRef2 = dynamic_cast<Derived&>(baseRef1);

		baseRef2.func();
		derivedRef2.func();
	}
	catch (exception e)
	{
		cout << e.what() << endl;
	}
	return 0;
}
```

```
Base class func. ID: 1
Derived class func. ID: 3
Bad dynamic_cast!
```

Runtime throws a *bad cast* exception.

These are the two important differences of dynamic casting. If the casted types don't match it will return a NULL pointer in case of pointers, or throw a bad cast exception for references. Then what about using dynamic casts on an object like we upcasted with static cast?

```
…
…
int main(int argc, char** argv)
{
	Base baseObj1(1);
	Derived derivedObj1(2, 3);

	baseObj1.func();
	derivedObj1.func();

	Base baseObj2 = dynamic_cast<Base>(derivedObj1);
	baseObj2.func();
	return 0;
}
```



```
Description
1   error C2680: 'Base' : invalid target type for dynamic_cast
2   error C2512: 'Base' : no appropriate default constructor available
```

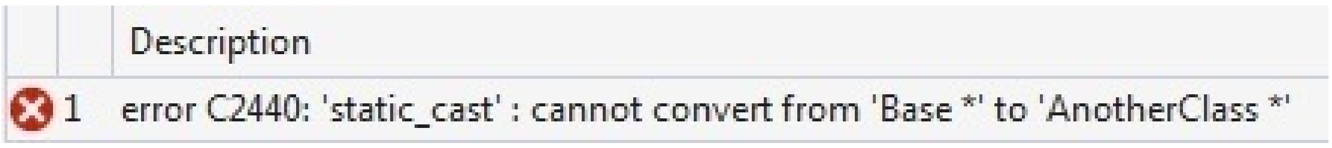
Dynamic cast wouldn’t even let you upcast in this case, which was perfectly fine with static cast. This is because dynamic casting can only used with pointers and references.

Reinterpret cast

This is a dangerous one. It will let you do things static and dynamic cast won’t. Reinterpret cast literally makes the compiler interpret a certain pointer or reference as the casting type. No questions asked. Let’s see an example first with static cast.

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

class Base {
public:
        int Base_ID;

        Base(int val) : Base_ID(val)
        {}

        void func()
        {
                cout << “Base class func. Base_ID: “ << Base_ID << endl;
        }
};

class AnotherClass {
public:
        int AnotherClass_ID;

        AnotherClass(int val) : AnotherClass_ID(val)
        {}

        void func()
        {
                cout << “AnotherClass func. AnotherClass_ID: “ << AnotherClass_ID << endl;
        }
};
```

```
int main(int argc, char** argv)
{
        Base* basePtr1 = new Base(10);
        AnotherClass* anotherPtr = new AnotherClass(20);

        anotherPtr = static_cast<AnotherClass*> (basePtr1);
        anotherPtr->func();
        return 0;
}
```


```
    Description
1   error C2440: 'static_cast' : cannot convert from 'Base *' to 'AnotherClass *'
```

We have seen this error before. Although static casting can be used to downcast in the hierarchy, it doesn’t let us cast to a different class type. Now let’s try reinterpret cast.

```
…
…
int main(int argc, char** argv)
{
        Base* basePtr1 = new Base(10);
        AnotherClass* anotherPtr = new AnotherClass(20);

        anotherPtr = reinterpret_cast<AnotherClass*> (basePtr1);
        anotherPtr->func();
        return 0;
}
```

```
AnotherClass func. AnotherClass_ID: 10
```

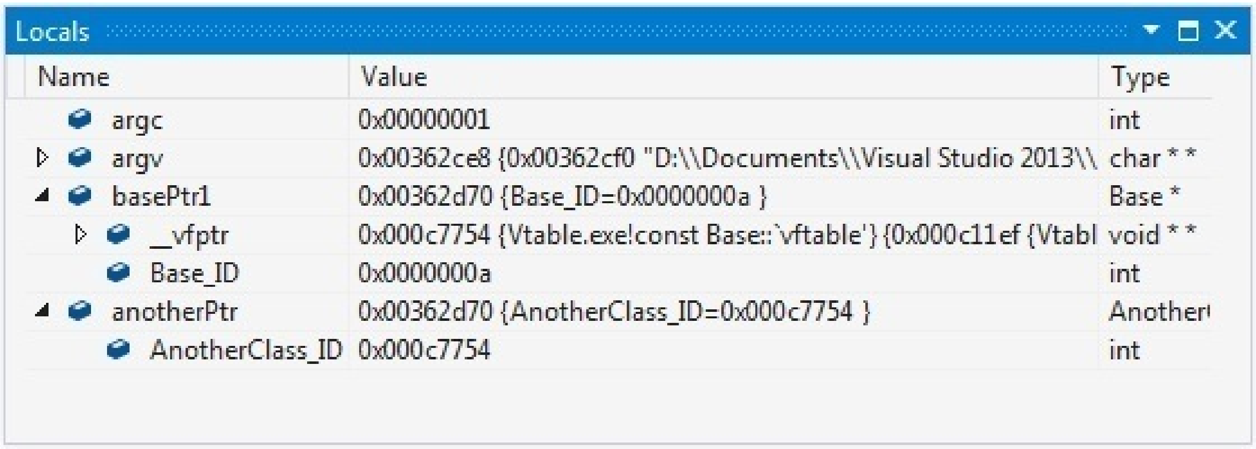
It not only compiles, the code almost works! We assigned a *Base* type pointer to an *AnotherClass* pointer and invoked the function in *AnotherClass*. By now you should know why the correct function is called but the wrong value is printed. What the reinterpret cast does here is to tell the compiler to treat the object pointed to by *anotherPtr* as an object of *AnotherClass*. The compiler does not question your actions here. And then when we invoked the function it called the correct function because functions are not part of the instance. They are class specific. This function was suppoed to get a ‘**anotherPtr*‘ implicitly. But it really was a ‘**Base*’. So why didn’t we get any runtime exceptions? Because luckily *Base* and *AnotherClass* have the same structure. They both have one int variable and this variable is at the top of the object. That means *Base::ID* and

AnothehrClass::ID both have the same offset. But it doesn't take much to crash this program. Let's make *func* virtual in *Base* class.

```
class Base {
public:
        int Base_ID;

        Base(int val) : Base_ID(val)
        {}

        virtual void func()
        {
                cout << "Base class func. Base_ID: " << Base_ID << endl;
        }
};
…
…
```

```
AnotherClass func. AnotherClass_ID: 4076188
```

As you know making the function virtual puts the *vptr* at the top of the object (remember this is compiler specific). So *vptr* is at offset zero and the variable is after that. But in *AnotherClass* the variable offset doesn't change. Looking at the debug class structure will convince you even more.



Name	Value	Type
argc	0x00000001	int
▷ argv	0x00362ce8 {0x00362cf0 "D:\\Documents\\Visual Studio 2013\\	char * *
◢ basePtr1	0x00362d70 {Base_ID=0x0000000a }	Base *
▷ __vfptr	0x000c7754 {Vtable.exe!const Base::`vftable'} {0x000c11ef {Vtabl	void * *
Base_ID	0x0000000a	int
◢ anotherPtr	0x00362d70 {AnotherClass_ID=0x000c7754 }	Another
AnotherClass_ID	0x000c7754	int

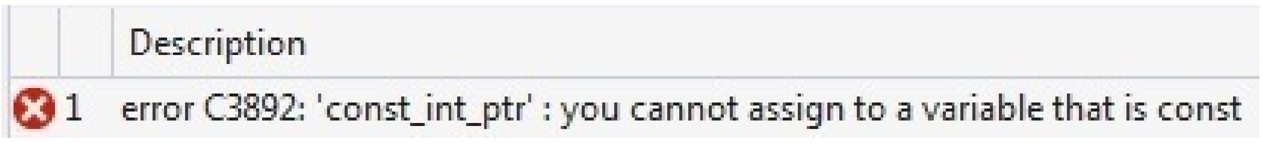
Note how *__vfptr* of *basePtr* and *AnotherClass_ID* of *anotherPtr* have the same value.

So you see that reinterpret cast can be dangerous if you cast the wrong type. Unlike dynamic cast, reinterpret casting does not have any runtime hit. Reinterpret cast literally tells the compiler to interpret the pointer or reference as the desired type.

Const cast

Const cast is pretty simple but yet it does something that all three previous casts cannot do. Const cast is able to remove or add the constness of a pointer or reference. But it is important to understand when this wouldn't work. Let's start with a simple example.

```
int main(int argc, char** argv)
{
        int non_const_int = 1;
        const int* const_int_ptr = &non_const_int;
        *const_int_ptr = 2;
        return 0;
}
```



		Description
✖	1	error C3892: 'const_int_ptr' : you cannot assign to a variable that is const

We cannot change *non_const_int* through *const_int_ptr* because the pointer is const. This is where we can use const cast.

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

int main(int argc, char** argv)
{
        int non_const_int = 1;
        const int* const_int_ptr = &non_const_int;
        int* non_const_int_ptr = const_cast<int*> (const_int_ptr);
        *non_const_int_ptr = 2;
        cout << "non_const_int: " << non_const_int << endl;

        return 0;
}
```

```
non_const_int: 2
```

Here we used *const_cast* to get a pointer without the constness and then use it to modify *non_const_int*.

Now let's see where const cast fails.

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

```
int main(int argc, char** argv)
{
        const int const_int = 1;
        const int* const_int_ptr = &const_int;
        int* non_const_int_ptr = const_cast<int*> (const_int_ptr);
        *non_const_int_ptr = 2;
        cout << “const_int: “ << const_int << endl;

        return 0;
}
```

```
non_const_int: 1
```

See that the value is not changed. In this case (Visual C++ 2013) the code compiles and runs without an issue (except the value not being modified as we wanted), but actually this behavior is undefined. So you might see different results with other compilers. The important thing to note is that you cannot use const_cast to modify a value that was declared const. Here *const_int* was declared as a constant. The const cast cannot make this a non-const.

I’m going to leave the topic of casts at that. We discussed the basics of the casts and some special cases. Each cast has its own use and now you should know where to use them.

Topic 19

Conversions and Promotions

A conversion is when a value of a certain type is converted to that of a different type. Promotion is similar in the sense that it also changes the type of the value, but it is promoted to a type that is higher in the hierarchy. We will go through examples of different kinds of conversions and promotions in this topic.

Before we get into examples, where do you think conversions and promotions happen? They can happen in the following cases:

- When using different types of operands with an operator
- When passing an argument to a function
- When initializing objects
- In if/switch statements

Let's look at these cases with examples now.

Arithmetic conversions

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

void Func(short val)
{
    cout << "Short arg. Result: " << val << endl;
}

void Func(int val)
{
    cout << "Int arg. Result: " << val << endl;
}

void Func(float val)
{
    cout << "Float arg. Result: " << val << endl;
}

int main(int argc, char** argv)
{
    short shortVal = 1;
    int intVal = 2;
    float floatVal = 3.0f;
```

```
        Func(shortVal + intVal);
        Func(shortVal + floatVal);
        Func(intVal + floatVal);
        return 0;
}
```

```
Int arg. Result: 3
Float arg. Result: 4
Float arg. Result: 5
```

We have three different overloaded functions, taking arguments of types short, int and float. Then we perform additions between short, int and float types and pass them as arguments to *Func*. It is not difficult to understand what is happening. When an operator gets different types of operands, one of the operands needs to be converted to bring both operands to a common type. For example, in the case of passing a short and an int to the '+' operator, short operand is converted to an int. The result is then an int, which is then passed to the function that takes an int parameter.

These types of conversions are called '*arithmetic conversions*'.

Let's see a few more examples of arithmetic conversions involving chars and booleans.

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

void Func(bool val)
{
        cout << "Bool arg. Result: " << val << endl;
}

void Func(char val)
{
        cout << "Char arg. Result: " << val << endl;
}

void Func(short val)
{
        cout << "Short arg. Result: " << val << endl;
}

void Func(int val)
```

```
{
	cout << “Int arg. Result: “ << val << endl;
}

int main(int argc, char** argv)
{
	bool boolVal = true;
	char charVal = ‘a’;
	short shortVal = 1;
	int intVal = 2;

	Func(charVal + boolVal);
	Func(charVal + shortVal);
	Func(charVal + charVal);
	Func(shortVal + shortVal);
	Func(charVal + intVal);
	return 0;
}
```

```
Int arg. Result: 98
Int arg. Result: 98
Int arg. Result: 194
Int arg. Result: 2
Int arg. Result: 99
```

You see, all of the additions resulted in an integer result. So when the operands are converted they are not converted to the higher type between the two of them. For example, earlier we saw when a short and an int resulted in an int. The short was converted to an int. But here, when we added two chars or two shorts, the result was not a char or a short. They were integers. So you see, if you are adding two operands which are lower than ints, both operands are converted to ints before the operation. The situation is different when you have types larger than integers.

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

void Func(int val)
{
	cout << “Int arg. Result: “ << val << endl;
}
```

```
void Func(float val)
{
        cout << “Float arg. Result: “ << val << endl;
}

void Func(double val)
{
        cout << “double arg. Result: “ << val << endl;
}

void Func(long int val)
{
        cout << “Long int arg. Result: “ << val << endl;
}

void Func(long double val)
{
        cout << “Long double arg. Result: “ << val << endl;
}

int main(int argc, char** argv)
{
        int intVal = 1;
        float floatVal = 2.0f;
        double doubleVal = 3;
        long int lIntVal = 4;
        long double lDoubleVal = 5;

        cout << “Func(intVal + intVal) - “; Func(intVal + intVal);
        cout << “Func(intVal + floatVal) - “; Func(intVal + floatVal);
        cout << “Func(intVal + doubleVal) - “; Func(intVal + doubleVal);
        cout << “Func(intVal + lIntVal) - “; Func(intVal + lIntVal);
        cout << “Func(intVal + lDoubleVal) - “; Func(intVal + lDoubleVal);
        cout << endl;
        cout << “Func(floatVal + doubleVal) - “; Func(floatVal + doubleVal);
        cout << “Func(floatVal + lIntVal) - “; Func(floatVal + lIntVal);
        cout << “Func(floatVal + lDoubleVal) - “; Func(floatVal + lDoubleVal);
        cout << endl;
        cout << “Func(doubleVal + lIntVal) - “; Func(doubleVal + lIntVal);
        cout << “Func(doubleVal + lDoubleVal) - “; Func(doubleVal + lDoubleVal);

        return 0;
}
```

```
Func(intVal + intVal) - Int arg. Result: 2
Func(intVal + floatVal) - Float arg. Result: 3
Func(intVal + doubleVal) - double arg. Result: 4
Func(intVal + lIntVal) - Long int arg. Result: 5
Func(intVal + lDoubleVal) - Long double arg. Result: 6

Func(floatVal + doubleVal) - double arg. Result: 5
Func(floatVal + lIntVal) - Float arg. Result: 6
Func(floatVal + lDoubleVal) - Long double arg. Result: 7

Func(doubleVal + lIntVal) - double arg. Result: 7
Func(doubleVal + lDoubleVal) - Long double arg. Result: 8
```

Here we look at larger types: floats, doubles, long ints and long doubles. The conversion is different than when we had chars and shorts. You can see that there is an order of level: int, long int, float, double, long double. When we mix operands of these types, the smaller type gets converted to the larger type.

Now let’s finally look at how arithmetic conversions happen with unsigned types.

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

void Func(int val)
{
        cout << “Int arg. Result: “ << val << endl;
}

void Func(unsigned int val)
{
        cout << “Unsigned int arg. Result: “ << val << endl;
}

void Func(float val)
{
        cout << “Float arg. Result: “ << val << endl;
}

void Func(double val)
```

```
{
        cout << “Double arg. Result: “ << val << endl;
}

int main(int argc, char** argv)
{
        int intVal = -10;
        unsigned int unsIntVal = 5;
        float floatVal = 1.0f;
        double doubleVal = 2;

        cout << “Func(intVal + unsIntVal) - “; Func(intVal + unsIntVal);
        cout << “Func(unsIntVal + unsIntVal) - “; Func(unsIntVal + unsIntVal);
        cout << “Func(unsIntVal + floatVal) - “; Func(intVal + floatVal);
        cout << “Func(unsIntVal + doubleVal) - “; Func(unsIntVal + doubleVal);

        return 0;
}
```

```
Func(intVal + unsIntVal) - Unsigned int arg. Result: 4294967291
Func(unsIntVal + unsIntVal) - Unsigned int arg. Result: 10
Func(unsIntVal + floatVal) - Float arg. Result: -9
Func(unsIntVal + doubleVal) - Double arg. Result: 7
```

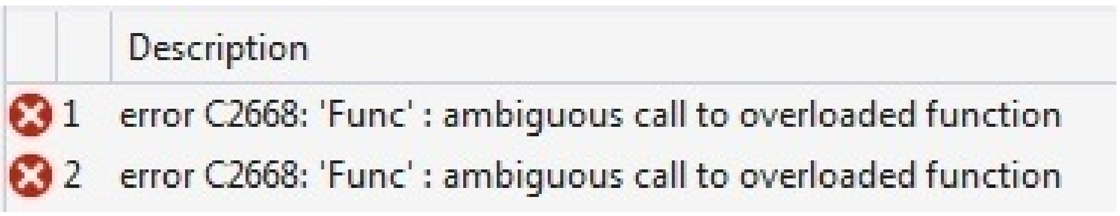
We have an int, an unsigned int, a float and a double. The important fact to note here is that when you have an int operand and an unsigned int operand, the int is converted to an unsigned. You can see that the result of the addition of the signed int and the unsigned is not what we would have expected. It’s giving us a very large value. This is because when the (signed) int was converted to an unsigned, its sign-ness was lost. The negative values of signed ints are depicted by two’s complement, which uses the most significant bits to denote the sign. So in this case, when our int was -10, the most significant bits were set. And when it was converted to an unsigned, these bits makes a very large unsigned value. That is the reason we see that large value. The rest of the combinations behave as we would expect. Float and double are still larger than unsigned int. So you need to be careful when you mix signed types with unsigned.

Function argument conversion

Let’s look at the second case now. What if we didn’t have a function that takes a parameter of type unsigned?

```
#include<iostream>
```

```
using namespace std;

void Func(int val)
{
        cout << “Int arg. Result: “ << val << endl;
}

void Func(float val)
{
        cout << “Float arg. Result: “ << val << endl;
}

void Func(double val)
{
        cout << “Double arg. Result: “ << val << endl;
}

int main(int argc, char** argv)
{
        int intVal = -10;
        unsigned int unsIntVal = 5;
        float floatVal = 1.0f;
        double doubleVal = 2;

        cout << “Func(intVal + unsIntVal) - “; Func(intVal + unsIntVal);
        cout << “Func(unsIntVal + unsIntVal) - “; Func(unsIntVal + unsIntVal);

        return 0;
}
```



```
Description
1  error C2668: 'Func' : ambiguous call to overloaded function
2  error C2668: 'Func' : ambiguous call to overloaded function
```

The compiler is facing an ambiguity. You see, the result of the additions are of type unsigned. And when there is no function that takes an unsigned, the compiler tries to convert the unsigned result to a type of a function that is defined. The unsigned can be converted to a type of int and we have a function that takes an int. The ambiguity is because we have two other functions that takes floats and doubles. The unsigned can be converted to an int, but then the int can be converted to a float or a double. So the result can be passed to any of the three functions. That is why we get the error. Let’s leave only the function with int parameter.

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

void Func(int val)
{
	cout << “Int arg. Result: “ << val << endl;
}

int main(int argc, char** argv)
{
	int intVal = -10;
	unsigned int unsIntVal = 5;
	float floatVal = 1.0f;
	double doubleVal = 2;

	cout << “Func(intVal + unsIntVal) - “; Func(intVal + unsIntVal);
	cout << “Func(unsIntVal + unsIntVal) - “; Func(unsIntVal + unsIntVal);
	return 0;
}
```

```
Func(intVal + unsIntVal) - Int arg. Result: -5
Func(unsIntVal + unsIntVal) - Int arg. Result: 10
```

No problems. Now let’s have only the float parameter function.

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

void Func(int val)
{
	cout << “Int arg. Result: “ << val << endl;
}

int main(int argc, char** argv)
{
	int intVal = -10;
	unsigned int unsIntVal = 5;

	cout << “Func(intVal + unsIntVal) - “; Func(intVal + unsIntVal);
```

```
    cout << “Func(unsIntVal + unsIntVal) - “; Func(unsIntVal + unsIntVal);
    return 0;
}
```

```
Func(intVal + unsIntVal) - Float arg. Result: 4.29497e+009
Func(unsIntVal + unsIntVal) - Float arg. Result: 10
```

The results are implicitly converted to floats. The conversion to float happens after the operator result. That is why you see a large number because the result of the signed and unsigned ints is an unsigned, which is a very large value as we saw earlier.

So what you see here is implicit function argument conversion. The compiler will convert the passed argument to the available function parameter type, if the conversion is possible. Keep in mind that if the correct parameter type is not available the compiler may do a conversion which would yield completely unexpected results as we saw in the example above. It is also possible that you lose the resolution of the value during conversion. When a float is converted to an int, the fractional part is discarded. Likewise an int may not be exactly representable as a float. So you need to pay attention when you have functions which take numeric arguments and also the result types of arithmetic operators.

Object instantiation

Now let’s discuss how argument conversion happens when we instantiate classes.

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

class conversion
{
public:
    float var;
    conversion(float val) : var(val)
    {
        cout << “Constructor” << endl;
    }

    conversion(const conversion& objToCopy)
    {
        var = objToCopy.var;
        cout << “Copy constructor” << endl;
    }

    conversion & operator=(const conversion &objToCopy)
```

```
        {
                var = objToCopy.var;
                cout << “Copy assignment operator” << endl;
                return *this;
        }
};

int main(int argc, char** argv)
{
        float floatVal = 1.0f;
        int intVal = 2;

        cout << “Line #1: “; conversion convObj1(floatVal);
        cout << “Line #2: “; conversion convObj2 = convObj1;
        cout << “Line #3: “; conversion convObj3(convObj2);
        cout << “Line #4: “; conversion convObj4 = floatVal;

        cout << “Line #5: “; conversion convObj5(intVal);
        cout << “Line #6: “; conversion convObj6 = intVal;
        return 0;
}
```

```
Line #1: Constructor
Line #2: Copy constructor
Line #3: Copy constructor
Line #4: Constructor
Line #5: Constructor
Line #6: Constructor
```

There isn’t anything of note happening in lines 1 to 4. The constructor and copy constructor are being called as we expect. Lines 5 and 6 are what we want to see. We are instantiating with an int. And it works the same way as we passed an int to a function with a float parameter. The compiler converts the int to a float before passing to the constructor. The int is promoted, in this case, to a float.

So you see argument conversions and promotions are happening in many places. These implicit conversions and promotions can make things easier but it is important to keep an eye out for these cases because they can result in unexpected results.

Topic 20

Name Lookup

Name lookup is the mechanism of finding the correct declaration of a name. Name look ups can be used for functions, types (including built-in ones) and enumerations. The mechanism of name lookup can be a complex mechanism when it involves multiple levels of inheritance, namespaces and templates. In this topic we will look at the basic functionality of name lookup when accessing functions or types.

All name lookups are one of two types only:

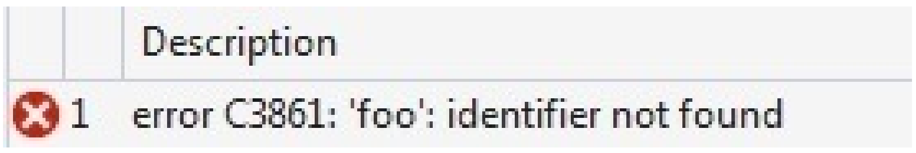
- Qualified name lookup
- Unqualified name lookup

We have used both types so far in the examples we have done. Let's start with qualified name lookup.

Qualified Name Lookup

Qualified name lookup takes place when you explicitly qualify the name with the scope resolution operator (::). Let's first look at an example of using a namespace.

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

namespace A {
    void foo()
    {
        cout << "A::foo()" << endl;
    }
}

int main(int argc, char** argv)
{
    foo();
    return 0;
}
```



```
Description
1   error C3861: 'foo': identifier not found
```

Here we declare a namespace *A* and define the function *foo* in that scope. Then we call *foo* in main. Note that this is an unqualified call, as we are not qualifying *foo* with any namespace of type. But we don't have to, do we? Because there is only one *foo* and that is in namespace *A*. You see, even though *foo* is defined it is not in the *scope*. You need to explicitly bring namespace *A* into the scope. There are two ways you can do that:

- With the *using* directive
- With qualifying the call

If we want to bring *foo* into the scope with 'using' directive:

```
…
…
int main(int argc, char** argv)
{
	using namespace A;
	foo();
	return 0;
}
```

```
A::foo()
```

You do not need to use 'using' directive right before calling the function. It can be anywhere between the function call and the namespace definition.

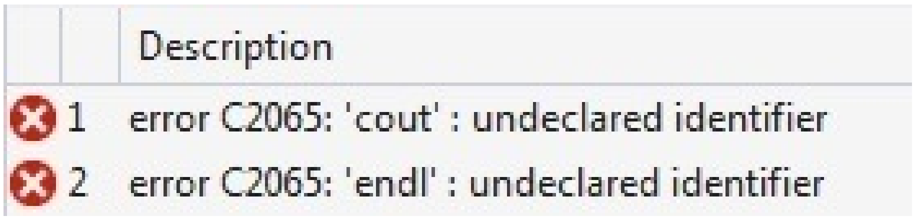
Now if we wanted to qualify the function call we do this:

```
…
…
int main(int argc, char** argv)
{
	a::foo();		// qualified call
	return 0;
}
```

Before we continue with the qualified name lookup, a word on the namespaces.

Namespace

As you saw in the example above we need to explicitly bring the namespace into the scope. The compiler will not look in a namespace even if it is in the same compilation unit. This is why we need to always do '*using namespace std*' for using *cout* and *endl*. If we removed '*using namespace std*' we'd get:



```
Description
1   error C2065: 'cout' : undeclared identifier
2   error C2065: 'endl' : undeclared identifier
```

The functions are defined in *iostream* and we include it in our compilation unit, but that is not enough. When the function or type is declared inside a namespace you need to explicitly bring it into the scope, or qualify the call. So if we removed ‘using namespace std’ we’d have to qualify our calls for *cout* and *endl*.

```
namespace A {
        void foo()
        {
                std::cout << “A::foo()” << std::endl; //qualified calls
        }
}
…
…
```

Qualified name lookup is pretty much that, qualifying the calling name. We will look at different aspects of this in the following examples.

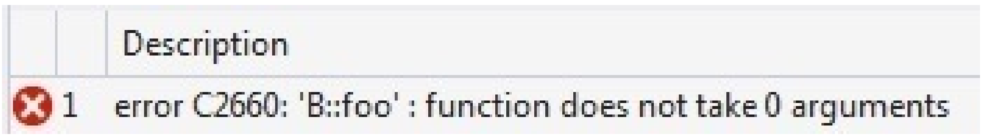
Multiple namespaces

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

void foo()
{
        cout << “::foo()” << endl;
}

namespace A {
        void foo()
        {
                cout << “A::foo()” << endl;
        }
}

namespace B {
        using namespace A;

        void foo(int val)
        {
                cout << “B::foo(int)” << endl;
        }
}

int main(int argc, char** argv)
{
        b::foo(1);
        return 0;
}
```

```
B::foo(int)
```

Here we have three *foo* implementations: in global scope, in namespace *A* and in namespace *B*. Note *B::foo(int)* is overloaded and namespace *B* includes namespace *A*. Then we are making a qualified function call to *B::foo(int)*. The compiler has no problem locating it in namespace *B*. Now we would like to call *A::foo()*. We need to qualify this call as there is another *foo* in the global scope. We can do this in two ways. First we can qualify with namespace *A* and then we can also qualify through namespace *B*. Note that namespace *B* brings in namespace *A* into its scope.

```
…
…
int main(int argc, char** argv)
{
        b::foo();
        return 0;
}
```



```
Description
1    error C2660: 'B::foo' : function does not take 0 arguments
```

It seems the compiler is having a problem with finding *B::foo()* though. But why so? *A::foo()* is in the namespace of *B*. We saw earlier that we can bring in a namespace into the scope by the ‘using’ directive. Let’s do a small modification. Change function name *B::foo(int)* into something different and try.

```
…
…
namespace B {
        using namespace A;

        void fooBar(int val)
        {
                cout << "B::fooBar(int)" << endl;
        }
}
…
…
```

```
A::foo()
A::foo()
```

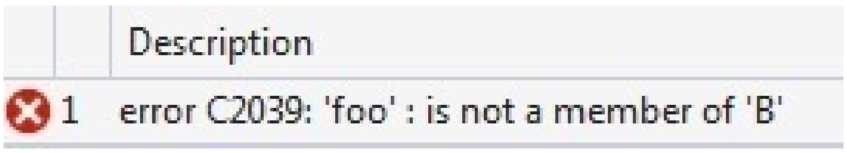
When we changed the function name in namespace *B*, the compiler had no problems finding *A::foo()* in namespace *B*. What happened?

The problem is, when we qualify the function name with the namespace the compiler first searches in that namespace only. So in our case, the compiler searched for *foo* in namespace *B* and it found a *foo* function. But that *foo* didn't match our call as it takes an int argument. So the compiler marks this as an error and complains that there is no *foo* function that takes 0 arguments. The problem is, once the compiler finds a declaration for *foo* it terminates the search. The compiler is not going to keep looking for a function *foo* that takes 0 arguments. Once it finds a *foo* it tries to call that function. If the function declaration doesn't fit the call, then it's an error. Then why did the call to *B::foo()* succeeed when we changed the function name to *fooBar*? Because now there is no *foo* in the namespace of *B* so the compiler now moves on to search for namespaces that are in the scope by the 'using' directive. Namespace B has brought in namespace A with the 'using' directive and the compiler goes in to *A* to find *foo*, and finds it. So rememeber that when we qualify the name, the compiler will first check in the qualified namespace and only if it cannot find it, will move to other namespaces included in there.

What about the global namespace? Will the iterative search ultimately move to the global scope? Let's change the names of *A::foo* and *B::foo* and leave only *::foo*.

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

void foo()
{
        cout << "::foo()" << endl;
}
```

```
namespace A {
        void fooA()
        {
                cout << “A::fooA()” << endl;
        }
}

namespace B {
        using namespace A;

        void fooB(int val)
        {
                cout << “B::fooB(int)” << endl;
        }
}

int main(int argc, char** argv)
{
        b::foo();
        return 0;
}
```

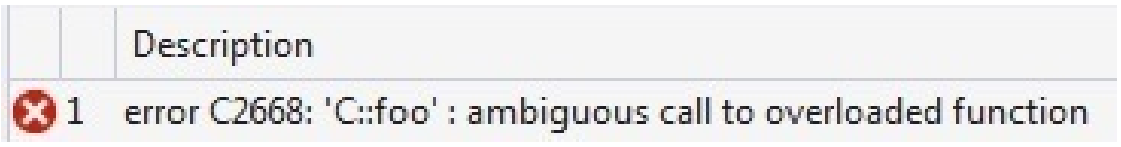


```
        Description
1       error C2039: 'foo' : is not a member of 'B'
```

No. The compiler will not move on to the global namespace. So keep in mind that when you do a qualified lookup, the compiler will only search the specified namespace and the included namespaces thereof. Although global namespace is, you know, global, it doesn’t come into the scope when the compiler does a qualified name look up.

Let’s add two more namespaces.

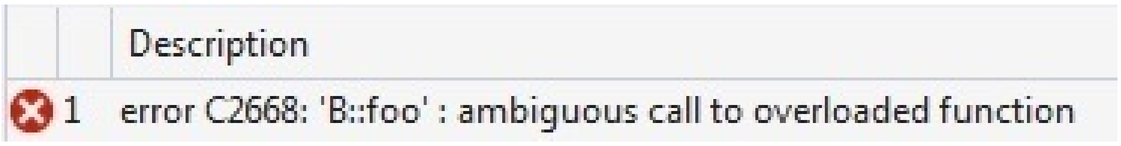
```
#include<iostream>
using namespace std;

void foo()
{
        cout << “::foo()” << endl;
}

namespace A {
        void foo()
        {
                cout << "A::foo()" << endl;
        }
}

namespace B {
        using namespace A;

        void foo(int val)
        {
                cout << "B::foo(int)" << endl;
        }
}

namespace C {
        void foo(int val)
        {
                cout << "C::foo(int)" << endl;
        }
}

namespace D {
        using namespace B;
        using namespace C;
}

int main(int argc, char** argv)
{
        d::foo(1);
        return 0;
}
```



```
Description
1   error C2668: 'C::foo' : ambiguous call to overloaded function
```

Namespace *D* includes namespaces *B* and *C* and both of these have *foo(int)* defined. And the compiler is complaining about an ambiguous call. This is because there are two *foo(int)* definitions. This shows that when searching included namespaces, the compiler does not go through them one after the other. If that is the case then the compiler must've found either *B::foo(int)* or *C::foo(int)* first and invoked it. But the compiler clearly has both *foo(int)* implementations available in the scope. That is why the ambiguity. But there is something odd in the error message. It says ambiguous call of *C::foo*, when we are doing *D::foo*. It sounds a little odd but there is nothing to it. Let's change the order of the using directives in namespace D.

```
…
….
namespace D {
        using namespace C;
        using namespace B;
}
…
…
```


Description
1 error C2668: 'B::foo' : ambiguous call to overloaded function

This time it complains about an ambiguous call to *B::foo*. It depends on which order the compiler brings in the namespaces into the scope. This is all very compiler dependent. But what matters is that all of the included namespaces are brought in to scope iteratively. Let's look at one more example.

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

void foo()
{
        cout << "::foo()" << endl;
}

namespace A {
        void foo()
        {
                cout << "A::foo()" << endl;
        }
}

namespace B {
        using namespace A;

        void fooBar(int val)
        {
                cout << "B::foo(int)" << endl;
        }
}

namespace C {
        void foo()
        {
                cout << “C::foo()” << endl;
        }
}

namespace D {
        using namespace C;
        using namespace B;
}

int main(int argc, char** argv)
{
        d::foo();
        return 0;
}
```

```
C::foo()
```

Now we have only two *foo()* implementations in the namespaces (excluding *::foo*, as global namespace is not looked at). The qualified call to *D::foo()* can find either *C::foo()* or *A::foo()* through namespace *B*. But you see, there is no ambiguity. The compiler resolves the call to *C::foo()*. Why wasn’t *A::foo()* found? The search didn’t reach to that level. There is no *D::foo()* so the search went one included namespace level up, which are *C* and *B*, where the compiler found *C::foo()* in the scope. The search stopped and *C::foo()* was called. There was no need to go up to namespace *A*. If there was no function *C::foo()*, then the search will move to included namespaces, in this case namespace *A* included in *B*, and have found *A::foo()*. If instead there was *C::foo(int)*, then we would’ve gotten the compiler error about *C::foo(int)* not taking 0 arguments.

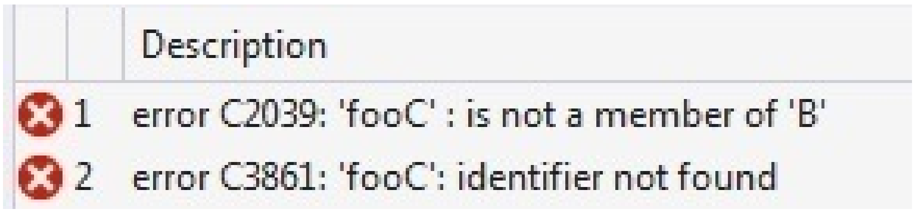
(However in Visual C++, although it compiles fine and doesn’t give out any warnings, there is a tooltip below “*D::foo()*” that says multiple instances of *foo()* is found.)

Nested namespaces

What about a namespace inside a namespace?

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

```
namespace A {
	void foo()
	{
		cout << “A::foo()” << endl;
	}
}

namespace B {
	using namespace A;

	void fooB()
	{
		cout << “B::fooB()” << endl;
	}

	namespace C {
		void foo()
		{
			cout << “B::C::foo()” << endl;
		}

		void fooC()
		{
			cout << “B::C::fooC()” << endl;
		}
	}
}

int main(int argc, char** argv)
{
	b::foo();
	b::fooC();
	b::C::foo();
	return 0;
}
```



```
    Description
1   error C2039: 'fooC' : is not a member of 'B'
2   error C3861: 'fooC': identifier not found
```

Here we have namespace *A*, and namespace *B*, which includes namespace *A*, and then namespace *C* is defined within *B*. And there is *foo()* defined in namespace *C* as well. So there are two *foo* functions within namespace *B*.

As you can see from the compiler error, the compiler could not find *fooC*. So although namespace *C* is defined within B, the functions inside *C* are not in the scope. If we comment out *B::fooC()*, we get:

```
A::foo()
B::C::foo()
```

So you see, the *foo* in *B*'s scope is only *A::foo*. *B::C::foo()* is not in the scope. If we want to call functions in namespace *C* we need to qualify the call. But look at this example.

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

namespace A {
    void fooA()
    {
        cout << "A::fooA()" << endl;
    }
}

namespace B {
    using namespace A;

    void fooB()
    {
        cout << "B::fooB()" << endl;
    }

    namespace C {
        void fooC()
        {
            fooA();
            fooB();
        }
    }
}

int main(int argc, char** argv)
{
    b::C::fooC();
    return 0;
}
```

```
A::fooA()
B::fooB()
```

Even though namespace *C* was in the scope of namespace *B*, everything in *B*, including the namespaces brought in to scope with ‘using’, is in the scope of namespace *C*.

This is pretty much all there is to common uses of qualified name lookups with namespaces. Remember the iterative process of namespace look up when you qualify the call. This is a bit different from what happens when we do an unqualified call.

Unqualified name lookup

Unqualified name look up is exactly what it means. It is name lookup when the name is not qualified. Earlier we discussed about qualified calls where the calling name is qualified with the scope resolution operator. Whereas in qualified name lookup we mainly considered namespaces, unqualified name lookup is all about the scope. Let’s start with an example.

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

namespace A {
        int x = 1;
        void foo1()
        {
                cout << “A::foo1 - x= “ << x << endl;
        }
}

namespace B {

        int x = 2;
        void foo2()
        {
                cout << “B::foo2 - x =” << x << endl;
        }

        using namespace A;
```

```
    void foo3()
    {
        cout << “B::foo3 - x =” << x << endl;
    }


    namespace C {
        void foo4()
        {
            cout << “B::C::foo4 - x =” << x << endl;
        }

        int x = 3;
        void foo5()
        {
            cout << “B::C::foo5 - x =” << x << endl;
        }
    }
}

int main(int argc, char** argv)
{
    a::foo1();
    b::foo1();
    b::foo2();
    b::foo3();
    b::C::foo4();
    b::C::foo5();
    return 0;
}
```

```
A::foo1 - x= 1
A::foo1 - x= 1
B::foo2 - x =2
B::foo3 - x =2
B::C::foo4 - x =2
B::C::foo5 - x =3
```

It’s not difficult to understand what is happening here. We are trying to see the scope boundaries of variable *x*. As we saw in the last example of qualified name lookup, the inner namespace has in its scope everything in the enclosing namespace. So namespace *C* has variable *x* in *B* in its scope.

In qualified name lookup we saw that namespaces are searched iteratively for the name. First the qualified namespace is searched, then if the name is not found the included namespaces are searched and it continues like that until it is found or the last namespace was reached. We also saw that the search does not go to the global namespace. The same mechanism is happening in unqualified lookup too. The lookup for the name starts with the current scope and moves upwards.

So when I mentioned earlier that inner namespace has outer namespaces in its scope, it was actually incorrect. It is incorrect to say that innermost namespace has outer namespaces in its 'scope'. It doesn't. But the thing about unqualified name lookup is that name lookup moves to outer scopes.

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

namespace A {
        int a = 1;

        namespace B {
                int b = 2;

                namespace C {
                        int c = 3;
                        int d = 4;

                        namespace D {
                                int d = 5;

                                void scopeCheck()
                                {
                                        cout << “a = “ << a << endl;
                                        cout << “b = “ << b << endl;
                                        cout << “c = “ << c << endl;
                                        cout << “d = “ << d << endl;
                                }
                        }
                }
        }
}
```

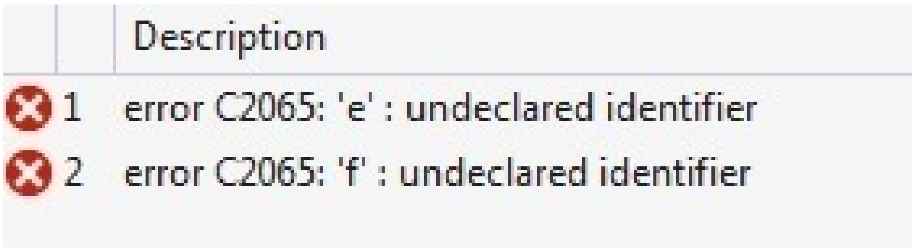
```
int main(int argc, char** argv)
{
        a::B::C::D::scopeCheck();
        return 0;
}
```

```
a = 1
b = 2
c = 3
d = 5
```

Here you see that unqualified name lookup in namespace *D* moves out to the outermost namespace scope. It is clear that the outer namespaces' scope doesn't extend to the inner namespace and vice versa. That is why we don't have any multiple definition errors for variable *d*. Now see what happens when we have variables defined after the function call.

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

namespace A {
        int a = 1;

        namespace B {
                int b = 2;

                namespace C {
                        int c = 3;
                        int d = 4;

                        namespace D {
                                int d = 5;

                                void scopeCheck()
                                {
                                        cout << "e = " << e << endl;
                                        cout << "f = " << f << endl;
                                }
                                int e = 5;
                        }
                        int f = 6;
```

```
            }
        }
}


int main(int argc, char** argv)
{
        a::B::C::D::scopeCheck();
        return 0;
}
```



```
Description
1   error C2065: 'e' : undeclared identifier
2   error C2065: 'f' : undeclared identifier
```

Although variable is in namespace *C*'s scope, it is not found in the lookup. Anything that is defined after the function is not in the scope search.

The rules of unqualified name lookup apply to classes the same way.

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

class A {
public:
        static const int a = 1;

        class B {
        public:
                static const int b = 2;

                class C {
                public:
                        static const int c = 3;
                        static const int d = 4;

                        class D {
                        public:
                                static const int d = 5;

                                static void scopeCheck()
```

```
                    {
                            cout << “a = “ << a << endl;
                            cout << “b = “ << b << endl;
                            cout << “c = “ << c << endl;
                            cout << “d = “ << d << endl;
                    }
                };
            };
        };
};

int main(int argc, char** argv)
{
        A::B::C::D::scopeCheck();
        return 0;
}
```

```
a = 1
b = 2
c = 3
d = 5
```

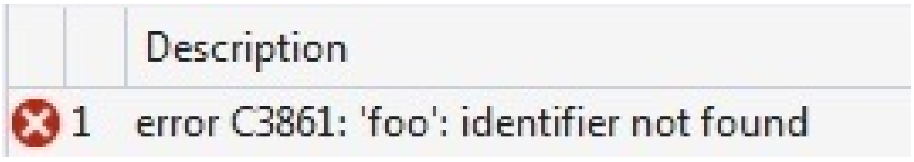
Next we move to the final topic on name lookups, ‘argument dependent lookup’.

Argument dependent lookup

Argument dependent lookup is also called “*Koenig lookup*” and is part of the unqualified name lookup. Let’s discuss this with an example.

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

namespace A {
        struct A_struct
        {};

        void foo()
        {
                cout << “A:::foo” << endl;
        }
};
```

```
int main(int argc, char** argv)
{
	foo();
	return 0;
}
```


```
	Description
1	error C3861: 'foo': identifier not found
```

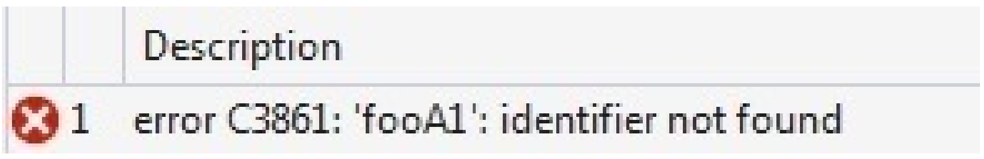
It’s easy to understand why the compiler cannot find *foo*. It is in namespace *A* and since we are doing an unqualified call the search only extends to the global namespace, not into namespace *A*. Completely normal behavior. Now let’s pass an argument to *foo*, an argument that is part of namespace *A* and see how things change.

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

namespace A {
	struct A_struct
	{};

	void foo(A_struct)
	{
		cout << “A::foo” << endl;
	}
};

int main(int argc, char** argv)
{
	a::A_struct structA;
	foo(structA);
	return 0;
}
```

```
A:::foo
```

This is *Koening lookup* or *argument dependent lookup*. But how was the compiler able to find *foo*? Why did the compiler decide to search in namespace *A*? Because *foo* has a parameter that is in namespace *A*. This is what argument dependent lookup is. The compiler will lookup in namespaces of the passed arguments. Let’s expand this a little bit.

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

namespace A {
        struct A_struct
        {};

        void fooA1()
        {
                cout << “A::fooA1” << endl;
        }

        void fooA2(A_struct)
        {
                cout << “A::fooA2” << endl;
        }
};

namespace B {
        struct B_struct
        {};

        void fooB(A::A_struct structA)
        {
                cout << “B::fooB calling fooA…” << endl;
                fooA1();
        }
}

int main(int argc, char** argv)
{
        a::A_struct structA;
        b::fooB(structA);
        return 0;
}
```



```
Description
1   error C3861: 'fooA1': identifier not found
```

Now we have two namespaces, *A* and *B*, and *fooB* in namespace *B* is trying to call *fooA1* in namespace *A*. The compiler cannot find *fooA1* in the search scope. But why? We are passing an argument of namespace *A* to *fooB*. See, it doesn't work like that. The function itself must have an argument passed to it with the namespace. Although *A_struct* is passed to *fooB*, it is not passed to *fooA1*, so the compiler does not look for *fooA1* in *A_struct*'s namespace. Then what about *fooA2*?

```
…
…
namespace B {
        struct B_struct
        {};

        void fooB(A::A_struct structA)
        {
                cout << "B::fooB calling fooA…" << endl;
                fooA2(structA);
        }
}
…
…
```

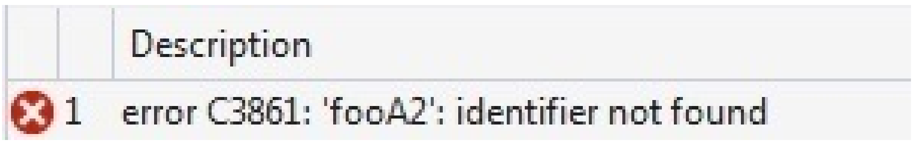
```
B::fooB calling fooA…
A::fooA2
```

As expected, now the compiler searches for *fooA2* in namespace of *A* because we are passing an argument in *A*. This is the basic mechanism of argument dependent lookup (ADL). However, there are some limitations to this.

Does not work with built-in type arguments:

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

namespace A {
        int A_var = 1;

        struct A_struct
        {};

        void fooA1()
        {
```

```
        cout << “A::fooA1” << endl;
    }

    void fooA2(int var)
    {
        cout << “A::fooA2” << endl;
    }
};

namespace B {
    struct B_struct
    {};

    void fooB(A::A_struct structA)
    {
        cout << “B::fooB calling fooA…” << endl;
        fooA2(A::A_var);
    }
}

int main(int argc, char** argv)
{
    a::A_struct structA;
    b::fooB(structA);
    return 0;
}
```



		Description
⊗	1	error C3861: 'fooA2': identifier not found

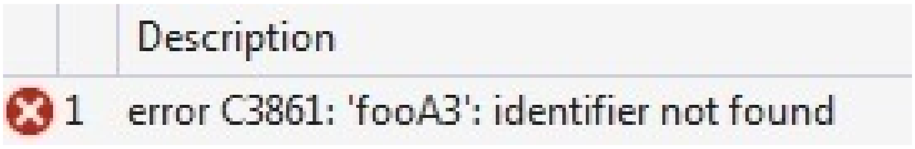
We change *fooA2* to take an int parameter instead of *A::struct_A* and then pass it *A_var*, the int variable in namespace *A*. Apparently the compiler doesn’t care to go into namespace *A* in this case although we are passing an int in namespace *A*. So ADL does not work for built-in types. It works on class types, as we saw with the struct before, and also for enumerations. There are quite a few rules and conditions governing the ADL lookup but let’s keep this discussion to the most common ways of using ADL.

Namespace search is not iterative:

Earlier in qualified name lookup we saw that the lookup does the search iteratively in the namespaces within namespaces. This is not true for ADL.

```
#include<iostream>
```

```
using namespace std;

namespace A {
        struct A_struct
        {};

        void fooA1()
        {
                cout << “A::fooA1” << endl;
        }

        void fooA2(A_struct)
        {
                cout << “A::fooA2” << endl;
        }

        namespace insideA {

                void fooA3(A_struct)
                {
                        cout << “A::insideA::fooA3” << endl;
                }
        }
        using namespace insideA;
};

namespace B {
        struct B_struct
        {};

        void fooB(A::A_struct structA)
        {
                cout << “B::fooB calling fooA…” << endl;
                fooA3(structA);
        }
}

int main(int argc, char** argv)
{
        a::A_struct structA;
        b::fooB(structA);
        return 0;
}
```



```
Description
1   error C3861: 'fooA3': identifier not found
```

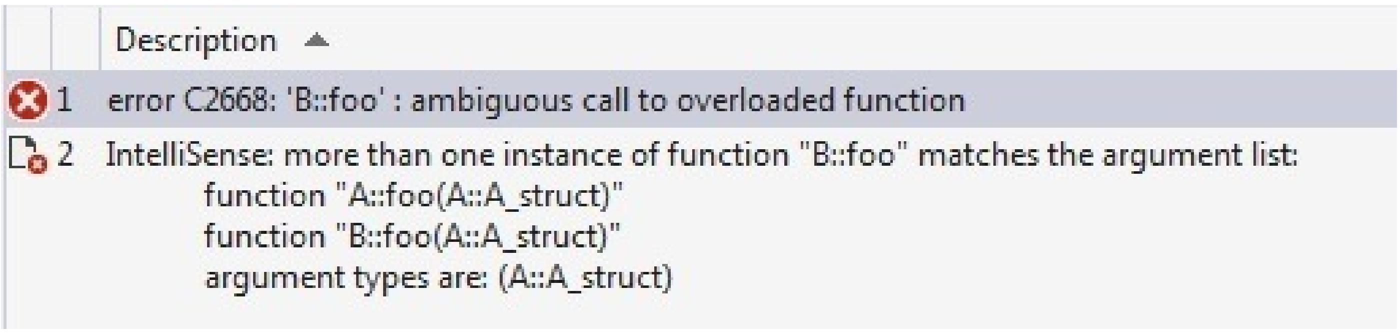
Here we define a namespace *insideA* within *A* and define *fooA3* that takes a parameter of type *A*. But you see the compiler is not able to find it, even though we have included namespace *insideA* with the *using* directive. So ADL limits its search just to the namespace of the argument.

ADL is different for classes and namespaces:

Let’s look at namespaces first. We define function *foo* in namespace *A* and also in the namespace *B*, and then call from namespace *B*.

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

namespace A {
        struct A_struct
        {};

        void foo(A_struct)
        {
                cout << “A::foo” << endl;
        }
};

namespace B {

        void foo(A::A_struct)
        {
                cout << “B::foo” << endl;
        }

        void fooB(A::A_struct structA)
        {
                cout << “B::fooB calling foo” << endl;
                foo(structA);
        }
}

int main(int argc, char** argv)
```

```
{
        a::A_struct structA;
        b::fooB(structA);
        return 0;
}
```



```
    Description ▲
✖ 1  error C2668: 'B::foo' : ambiguous call to overloaded function
  2  IntelliSense: more than one instance of function "B::foo" matches the argument list:
            function "A::foo(A::A_struct)"
            function "B::foo(A::A_struct)"
            argument types are: (A::A_struct)
```

There is ambiguity because the compiler finds both the *foo* functions, *B::foo* in the current scope and *A::foo* through ADL.

But things are a little different with classes. Let’s turn the same program to use classes instead of namespaces.

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

class A {
public:
        struct A_struct
        {};

        static void foo(A_struct)
        {
                cout << “A::foo” << endl;
        }
};

class B {
public:
        static void foo(A::A_struct)
        {
                cout << “B::foo” << endl;
        }

        static void fooB(A::A_struct structA)
        {
            cout << “B::fooB calling foo” << endl;
            foo(structA);
        }
};

int main(int argc, char** argv)
{
        A::A_struct structA;
        B::fooB(structA);
        return 0;
}
```

```
B::fooB calling foo
B::foo
```

With classes there is no ambiguity. Compiler finds *B::foo* in its current scope and calls it. So when working with classes, ADL is used only when the compiler cannot find a class member function.

As discussed earlier, there are quite a few rules and conditions regarding ADL. So qualify your calls to make sure the code is calling the function you want. Keep an eye out for iterative scope searches because you might not be accessing the variable you intend to.