7.9 — The stack and the heap

BY ALEX ON AUGUST 10TH, 2007 | LAST MODIFIED BY ALEX ON DECEMBER 21ST, 2015

The memory a program uses is typically divided into a few different areas, called segments:

- The code segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
- The bss segment (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
- The data segment (also called the initialized data segment), where initialized global and static variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The call stack, where function parameters, local variables, and other function-related information are stored.

For this lesson, we'll focus primarily on the heap and the stack, as that is where most of the interesting stuff takes place.

The heap segment

The heap segment (also known as the "free store") keeps track of memory used for dynamic memory allocation. We talked about the heap a bit already in lesson **6.9 -- Dynamic memory allocation with new and delete**, so this will be a recap.

In C++, when you use the new operator to allocate memory, this memory is allocated in the application's heap segment.

```
int *ptr = new int; // ptr is assigned 4 bytes in the heap
int *array = new int[10]; // array is assigned 40 bytes in the heap
```

The address of this memory is passed back by operator new, and can then be stored in a pointer. You do not have to worry about the mechanics behind the process of how free memory is located and allocated to the user. However, it is worth knowing that sequential memory requests may not result in sequential memory addresses being allocated!

```
int *ptr1 = new int;
int *ptr2 = new int;
// ptr1 and ptr2 may not have sequential addresses
```

When a dynamically allocated variable is deleted, the memory is "returned" to the heap and can then be reassigned as future allocation requests are received. Remember that deleting a pointer does not delete the variable, it just returns the memory at the associated address back to the operating system.

The heap has advantages and disadvantages:

- Allocating memory on the heap is comparatively slow.
- Allocated memory stays allocated until it is specifically deallocated (beware memory leaks) or the application ends (at which point the OS should clean it up).
- Dynamically allocated memory must be accessed through a pointer. Dereferencing a pointer is slower than accessing a variable directly.
- Because the heap is a big pool of memory, large arrays, structures, or classes can be allocated here.

The call stack

The **call stack** (usually referred to as "the stack") has a much more interesting role to play. The call stack keeps track of all the active functions (those that have been called but have not yet terminated) from the start of the program to the current point of execution, and handles allocation of all function parameters and local variables.

The call stack is implemented as a stack data structure. So before we can talk about how the call stack works, we need to understand what a stack data structure is.

The stack data structure

A **data structure** is a programming mechanism for organizing data so that it can be used efficiently. You've already seen

several types of data structures, such as arrays and structs. Both of these data structures provide mechanisms for storing data and accessing that data in an efficient way. There are many additional data structures that are commonly used in programming, quite a few of which are implemented in the standard library, and a stack is one of those.

Consider a stack of plates in a cafeteria. Because each plate is heavy and they are stacked, you can really only do one of three things:
1) Look at the surface of the top plate
2) Take the top plate off the stack (exposing the one underneath, if it exists)
3) Put a new plate on top of the stack (hiding the one underneath, if it exists)

In computer programming, a stack is a container data structure that holds multiple variables (much like an array). However, whereas an array lets you access and modify elements in any order you wish (called **random access**), a stack is more limited. The operations that can be performed on a stack correspond to the three things mentioned above:

1) Look at the top item on the stack (usually done via a function called top(), but sometimes called peek())
2) Take the top item off of the stack (done via a function called pop())
3) Put a new item on top of the stack (done via a function called push())

A stack is a last-in, first-out (LIFO) structure. The last item pushed onto the stack will be the first item popped off. If you put a new plate on top of the stack, the first plate removed from the stack will be the plate you just pushed on last. Last on, first off. As items are pushed onto a stack, the stack grows larger -- as items are popped off, the stack grows smaller.

For example, here's a short sequence showing how pushing and popping on a stack works:

```
Stack: empty
Push 1
Stack: 1
Push 2
Stack: 1 2
Push 3
Stack: 1 2 3
Pop
Stack: 1 2
Pop
Stack: 1
```

The plate analogy is a pretty good analogy as to how the call stack works, but we can make a better analogy. Consider a bunch of mailboxes, all stacked on top of each other. Each mailbox can only hold one item, and all mailboxes start out empty. Furthermore, each mailbox is nailed to the mailbox below it, so the number of mailboxes can not be changed. If we can't change the number of mailboxes, how do we get a stack-like behavior?

First, we use a marker (like a post-it note) to keep track of where the bottom-most empty mailbox is. In the beginning, this will be the lowest mailbox. When we push an item onto our mailbox stack, we put it in the mailbox that is marked (which is the first empty mailbox), and move the marker up one mailbox. When we pop an item off the stack, we move the marker down one mailbox and remove the item from that mailbox. Anything below the marker is considered "on the stack". Anything at the marker or above the marker is not on the stack.

The call stack segment

The call stack segment holds the memory used for the call stack. When the application starts, the main() function is pushed on the call stack by the operating system. Then the program begins executing.

When a function call is encountered, the function is pushed onto the call stack. When the current function ends, that function is popped off the call stack. Thus, by looking at the functions pushed on the call stack, we can see all of the functions that were called to get to the current point of execution.

Our mailbox analogy above is fairly analogous to how the call stack works. The call stack is a fixed-size chunk of memory addresses. The mailboxes are memory addresses, and the "items" we're pushing and popping on the stack are called **stack**

frames. A stack frame keeps track of all of the data associated with one function call. We'll talk more about stack frames in a bit. The "marker" is a register (a small piece of memory in the CPU) known as the stack pointer (sometimes abbreviated "SP"). The stack pointer keeps track of where the top of the call stack currently is.

The only difference between our hypothetical mailbox stack and the call stack is that when we pop an item off the call stack, we don't have to erase the memory (the equivalent of emptying the mailbox). We can just leave it to be overwritten by the next item pushed to that piece of memory. Because the stack pointer will be below that memory location, we know that memory location is not on the stack.

The call stack in action

Let's examine in more detail how the call stack works. Here is the sequence of steps that takes place when a function is called:

1. The program encounters a function call.
2. A stack frame is constructed and pushed on the stack. The stack frame consists of:
    - The address of the instruction beyond the function call (called the **return address**). This is how the CPU remembers where to go after the function returns.
    - All function arguments are placed on the stack.
    - Local variables are pushed onto the stack.
    - Saved copies of any registers modified by the function that need to be restored when the function returns
3. The CPU jumps to the function's start point.
4. The instructions inside of the function begin executing.

When the function terminates, the following steps happen:

1. Registers are restored from the call stack
2. The stack frame is popped off the stack. This destroys all local variables and arguments.
3. The return value is handled.
4. The CPU resumes execution at the return address.

Return values can be handled in a number of different ways, depending on the computer's architecture. Some architectures include the return value as part of the stack frame. Others use CPU registers.

Typically, it is not important to know all the details about how the call stack works. However, understanding that functions are effectively pushed on the stack when they are called and popped off when they return gives you the fundamentals needed to understand recursion, as well as some other concepts that are useful when debugging.

A quick and dirty call stack example

Consider the following simple application:

```
1  int foo(int x)
2  {
3      // b
4      return x;
5  } // foo is popped off the call stack here
6
7  int main()
8  {
9      // a
10     foo(5); // foo is pushed on the call stack here
11     // c
12
13     return 0;
14 }
```

The call stack looks like the following at the labeled points:

a:

```
main()
```

b:

```
foo() (including parameter x)
main()
```

c:

main()

Stack overflow

The stack has a limited size, and consequently can only hold a limited amount of information. On Windows, the default stack size is 1MB. On some unix machines, it can be as large as 8MB. If the program tries to put too much information on the stack, stack overflow will result. **Stack overflow** happens when all the memory in the stack has been allocated -- in that case, further allocations begin overflowing into other sections of memory.

Stack overflow is generally the result of allocating too many variables on the stack, and/or making too many nested function calls (where function A calls function B calls function C calls function D etc…) Overflowing the stack will generally causes a program to crash.

Here is an example program that will likely cause a stack overflow. You can run it on your system and watch it crash:

```
int main()
{
    int stack[100000000];
    return 0;
}
```

This program tries to allocate a huge array on the stack. Because the stack is not large enough to handle this array, the array allocation overflows into portions of memory the program is not allowed to use. Consequently, the program crashes.

Here's another program that will cause a stack overflow for a different reason:

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

int main()
{
    foo();

    return 0;
}
```

In the above program, a stack frame is pushed on the stack every time function foo() is called. Since foo() calls itself infinitely, eventually the stack will run out of memory and cause an overflow.

The stack has advantages and disadvantages:

- Allocating memory on the stack is comparatively fast.
- Memory allocated on the stack stays in scope as long as it is on the stack. It is destroyed when it is popped off the stack.
- All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
- Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes passing by value or creating local variables of large arrays or other memory-intensive structures.


7.10 -- std::vector capacity and stack behavior


Index


7.8 -- Function Pointers

Share this:

C++ TUTORIAL | PRINT THIS POST

62 comments to 7.9 — The stack and the heap


Abhishek
January 11, 2008 at 5:57 am · Reply

Where can I find the stack in my CPU box?
Is it a separate hardware or just a space in RAM or Hard disk?


Alex
January 11, 2008 at 8:10 am · Reply

According to **Wikipedia**, “In most modern computer systems, each thread has a reserved region of memory referred to as its stack”. So it’s just RAM memory being used in a stack-like manner. As soon as the thread/program is killed, that memory can be reused for other stuff.

Jeff
April 24, 2008 at 2:05 pm · Reply

8.3 — Public vs private access specifiers

BY ALEX ON SEPTEMBER 4TH, 2007 | LAST MODIFIED BY ALEX ON DECEMBER 17TH, 2015

Public and private members

Consider the following struct:

```
1  struct DateStruct // members are public by default
2  {
3      int month; // public by default, can be accessed by anyone
4      int day; // public by default, can be accessed by anyone
5      int year; // public by default, can be accessed by anyone
6  };
7
8  int main()
9  {
10     DateStruct date;
11     date.month = 10;
12     date.day = 14;
13     date.year= 2020;
14
15     return 0;
16 }
```

In this program, we declare a DateStruct and then we directly access its members in order to initialize them. This works because all members of a struct are public members by default. **Public members** are members of a struct or class that can be accessed from outside of the struct or class. In this case, function main() is outside of the struct, but it can directly access members month, day, and year, because they are public.

On the other hand, consider the following almost-identical class:

```
1  class DateClass // members are private by default
2  {
3      int m_month; // private by default, can only be accessed by other members
4      int m_day; // private by default, can only be accessed by other members
5      int m_year; // private by default, can only be accessed by other members
6  };
7
8  int main()
9  {
10     DateClass date;
11     date.m_month = 10; // error
12     date.m_day = 14; // error
13     date.m_year = 2020; // error
14
15     return 0;
16 }
```

If you were to compile this program, you would receive errors. This is because by default, all members of a class are private. **Private members** are members of a class that can only be accessed by other members of the class. Because main() is not a member of DateClass, it does not have access to date's private members.

Access specifiers

Although class members are private by default, we can make them public by using the public keyword:

```
1  class DateClass
2  {
3  public: // note use of public keyword here, and the colon
4      int m_month; // public, can be accessed by anyone
5      int m_day; // public, can be accessed by anyone
6      int m_year; // public, can be accessed by anyone
```

```
};

int main()
{
    DateClass date;
    date.m_month = 10; // okay because m_month is public
    date.m_day = 14;  // okay because m_day is public
    date.m_year = 2020;  // okay because m_year is public

    return 0;
}
```

Because DateClass's members are now public, they can be accessed directly by main().

The public keyword, along with the following colon, is called an access specifier. **Access specifiers** determine who has access to the members that follow the specifier. Each of the members "acquires" the access level of the previous access specifier (or, if none is provided, the default access specifier).

C++ provides 3 different access specifier keywords: public, private, and protected. Public and private are used to make the members that follow them public members or private members respectively. The third access specifier, protected, works much like private does. We will discuss the difference between the private and protected access specifier when we cover inheritance.

Mixing access specifiers

Classes can (and almost always do) use multiple access specifiers to set the access levels of each of its members.

In general, member variables are usually made private, and member functions are usually made public. We'll take a closer look at why in the next lesson.

Rule: Make member variables private, and member functions public, unless you have a good reason not to.

Let's take a look at an example of a class that uses both private and public access:

```
#include <iostream>

class DateClass // members are private by default
{
    int m_month; // private by default, can only be accessed by other members
    int m_day; // private by default, can only be accessed by other members
    int m_year; // private by default, can only be accessed by other members

public:
    void setDate(int month, int day, int year) // public, can be accessed by anyone
    {
        // setDate() can access the private members of the class because it is a member of the
class itself
        m_month = month;
        m_day = day;
        m_year = year;
    }

    void print() // public, can be accessed by anyone
    {
        std::cout << m_month << "/" << m_day << "/" << m_year;
    }
};

int main()
{
    DateClass date;
    date.setDate(10, 14, 2020); // okay, because setDate() is public
    date.print(); // okay, because print() is public

    return 0;
```

```
}
```

This program prints:

```
10/14/2020
```

Note that although we can't access date's members variables m_month, m_day, and m_year directly from main (because they are private), we are able to access them indirectly through public member functions setDate() and print()!

The group of public members of a class are often referred to as a **public interface**. Because only public members can be accessed from outside of the class, the public interface defines how programs using the class will interface with the class. Note that main() is restricted to setting the date and printing the date. The class protects the member variables from being accessed or edited directly.

Some programmers prefer to list private members first, because the public members typically use the private ones, so it makes sense to define the private ones first. However, a good counterargument is that users of the class don't care about the private members, so the public ones should come first. Either way is fine.

Structs vs classes revisited

Now that we've talked about access specifiers, we can talk about the actual differences between a class and a struct in C++. A class defaults its members to private. A struct defaults its members to public. That's it!

Quiz time

1a) What is a public member?

Show Solution

1b) What is a private member?

Show Solution

1c) What is an access specifier?

Show Solution

1d) How many access specifiers are there, and what are they?

Show Solution

2) Write a simple class named Point3d. The class should contain:
* Three private member variables of type double named m_x, m_y, and m_z;
* A public member function named setValues() that allows you to set values for m_x, m_y, and m_z.
* A public member function named print() that prints the Point in the following format: <m_x, m_y, m_z>

Make sure the following program executes correctly:

```
int main()
{
    Point3d point;
    point.setValues(1.0, 2.0, 3.0);

    point.print();

    return 0;
}
```

This should print:

```
<1, 2, 3>
```

Show Solution

3) Now let's try something a little more complex. Let's write a class that implements a simple stack. Review lesson **7.9 -- The stack and the heap** if you need a refresher on a what a stack is.

The class should be named Stack, and should contain:
* A private fixed array of integers of length 10.
* A private integer to keep track of the length of the stack.
* A public member function named reset() that sets the length to 0 and all of the element values to 0.
* A public member function named push() that pushes a value on the stack. push() should return false if the array is already full, and true otherwise.
* A public member function named pop() that pops a value off the stack. If there are no values on the stack, it should return -1.
* A public member function named print() that prints all the values in the stack.

Make sure the following program executes correctly:

```
int main()
{
    Stack stack;
    stack.reset();

    stack.print();

    stack.push(5);
    stack.push(3);
    stack.push(8);
    stack.print();

    stack.pop();
    stack.print();

    stack.pop();
    stack.pop();

    stack.print();

    return 0;
}
```

This should print:

```
( )
( 5 3 8 )
( 5 3 )
( )
```

Show Solution

 **8.4 -- Access functions and encapsulation**

 **Index**

 **8.2 -- Classes and class members**

8.5b — Non-static member initialization

BY ALEX ON FEBRUARY 12TH, 2016 | LAST MODIFIED BY ALEX ON APRIL 22ND, 2016

When writing a class that has multiple constructors (which is most of them), having to specify default values for all members in each constructor results in redundant code. If you update the default value for a member, you need to touch each constructor.

Starting with C++11, it's possible to give non-static class member variables a default initialization value directly:

```
class Square
{
private:
    double m_length = 1.0; // m_length has a default value of 1.0
    double m_width = 1.0; // m_width has a default value of 1.0

public:
    Square()
    {
    // This constructor will use the default values above since they aren't overridden here
    }

    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }
};

int main()
{
    Square x; // x.m_length = 1.0, x.m_width = 1.0
    x.print();

    return 0;
}
```

This program produces the result:

```
length: 1.0, width: 1.0
```

Non-static member initialization provides default values for your member variables that your constructors will use if the constructors do not provide an initialization values for the members themselves (via the member initialization list).

However, note that constructors still determine what kind of objects may be created. Consider the following case:

```
class Square
{
private:
    double m_length = 1.0;
    double m_width = 1.0;

public:

    // note: No default constructor provided in this example

    Square(double length, double width)
        : m_length(length), m_width(width)
    {
        // m_length and m_width are initialized by the constructor (the default values aren't
used)
    }

```

```
    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }

};

int main()
{
    Square x; // will not compile, no default constructor exists, even though members have default initialization values

    return 0;
}
```

Even though we've provided default values for all members, no default constructor has been provided, so we are unable to create Square objects with no parameters.

If a default initialization value is provided and the constructor initializes the member via the member initializer list, the member initializer list will take precedence. The following example shows this:

```
class Square
{
private:
    double m_length = 1.0;
    double m_width = 1.0;

public:

    Square(double length, double width)
        : m_length(length), m_width(width)
    {
        // m_length and m_width are initialized by the constructor (the default values aren't used)
    }

    void print()
    {
        std::cout << "length: " << m_length << ", width: " << m_width << '\n';
    }

};

int main()
{
    Square x(2.0, 3.0);
    x.print();

    return 0;
}
```

This prints:

```
length: 2.0, width: 3.0
```

Rule: Favor use of non-static member initialization to give default values for your member variables.

Quiz time

1) Update the following program to use non-static member initialization and member initializer lists.

```
#include <string>
#include <iostream>
class Ball
{
```

```
private:
    std::string m_color;
    double m_radius;

public:
        // Default constructor with no parameters
    Ball()
    {
        m_color = "black";
        m_radius = 10.0;
    }

        // Constructor with only color parameter (radius will use default value)
    Ball(const std::string &color)
    {
        m_color = color;
        m_radius = 10.0;
    }

        // Constructor with only radius parameter (color will use default value)
    Ball(double radius)
    {
        m_color = "black";
        m_radius = radius;
    }

        // Constructor with both color and radius parameters
    Ball(const std::string &color, double radius)
    {
        m_color = color;
        m_radius = radius;
    }

    void print()
    {
        std::cout << "color: " << m_color << ", radius: " << m_radius << '\n';
    }
};

int main()
{
    Ball def;
    def.print();

    Ball blue("blue");
    blue.print();

    Ball twenty(20.0);
    twenty.print();

    Ball blueTwenty("blue", 20.0);
    blueTwenty.print();

    return 0;
}
```

This program should produce the result:

```
color: black, radius: 10
color: blue, radius: 10
color: black, radius: 20
color: blue, radius: 20
```

Show Solution

2) Why do we need to declare an empty default constructor in the program above, since all members are initialized via non-static member initialization?

Show Solution

8.6 -- Overlapping and delegating constructors

Index

8.5a -- Constructor member initializer lists

Share this:

C++ PROGRAMMING | PRINT THIS POST

12 comments to 8.5b — Non-static member initialization

Lokesh
February 8, 2016 at 3:08 am · Reply

I think the following program demonstrates all the rules about default constructors. I found the concept a bit twisty. I think you should use this particular example for quiz question no.1 as it serves as a summary(it just adds one or two concepts to what already exists in the question).

```
1  #include <iostream>
2  #include <string>
3
4
5  /*
6  - Using non-static member initialization.
7  Important note:
```

8.11 — Static member variables

BY ALEX ON SEPTEMBER 14TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 21ST, 2016

Review of static keyword uses

In the lesson on **file scope and the static keyword**, you learned that static variables keep their values and are not destroyed even after they go out of scope. For example:

```
#include <iostream>

int generateID()
{
    static int s_id = 0;
    return ++s_id;
}

int main()
{
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';
    std::cout << generateID() << '\n';

    return 0;
}
```

This program prints:

```
1
2
3
```

Note that s_id has kept its value across multiple function calls.

The static keyword has another meaning when applied to global variables -- it gives them internal linkage (which restricts them from being seen/used outside of the file they are defined in). Because global variables are typically avoided, the static keyword is not often used in this capacity.

Static member variables

C++ introduces two more uses for the static keyword when applied to classes: static member variables, and static member functions. Fortunately, these uses are fairly straightforward. We'll talk about static member variables in this lesson, and static member functions in the next.

Before we go into the static keyword as applied to member variables, first consider the following class:

```
class Something
{
public:
    int m_value = 1;
};

int main()
{
    Something first;
    Something second;

    second.m_value = 2;

    std::cout << first.m_value << '\n';
    std::cout << second.m_value << '\n';
```

```
16
17      return 0;
18  }
```

When we instantiate a class object, each object gets its own copy of all normal member variables. In this case, because we have declared two Something class objects, we end up with two copies of m_value: first.m_value, and second.m_value. first.m_nValue is distinct from second.m_nValue. Consequently, the program above prints:

```
1
2
```

Member variables of a class can be made static by using the static keyword. Unlike normal member variables, static member variables are shared by all objects of the class. Consider the following program, similar to the above:

```
1   class Something
2   {
3   public:
4       static int s_value;
5   };
6
7   int Something::s_value = 1;
8
9   int main()
10  {
11      Something first;
12      Something second;
13
14      second.s_value = 2;
15
16      std::cout << first.s_value << '\n';
17      std::cout << second.s_value << '\n';
18      return 0;
19  }
```

This program produces the following output:

```
2
2
```

Because s_value is a static member variable, s_value is shared between all objects of the class. Consequently, first.s_value is the same variable as second.s_value. The above program shows that the value we set using first can be accessed using second!

Static members are not associated with class objects

Although you can access static members through objects of the class (as shown with first.s_value and second.s_value in the example above), it turns out that static members exist even if no objects of the class have been instantiated! Much like global variables, they are created when the program starts, and destroyed when the program ends.

Consequently, it is better to think of static members as belonging to the class itself, not to the objects of the class. Because s_value exists independently of any class objects, it can be accessed directly using the class name and the scope operator (in this case, Something::s_value):

```
1   class Something
2   {
3   public:
4       static int s_value; // declares the static member variable
5   };
6
7   int Something::s_value = 1; // defines the static member variable (we'll discuss this line bel
8   ow)
9
```

```
int main()
{
    // note: we're not instantiating any objects of type Something

    Something::s_value = 2;
    std::cout << Something::s_value << '\n';
    return 0;
}
```

In the above snippet, s_value is referenced by class name rather than through an object. Note that we have not even instantiated an object of type Something, but we are still able to access and use Something::s_value. This is the preferred method for accessing static members.

Defining and initializing static member variables

When we declare a static member variable inside a class, we're simply telling the class that a static member variable exists (much like a forward declaration). Because static member variables are not part of the individual class objects (they get initialized when the program starts), you must explicitly define the static member outside of the class, in the global scope.

In the example above, we do so via this line:

```
int Something::s_value = 1; // defines the static member variable
```

This line serves two purposes: it instantiates the static member variable (just like a global variable), and optionally initializes it. In this case, we're providing the initialization value 1. If no initializer is provided, C++ initializes the value to 0.

Note that this static member definition is not subject to access controls: you can define and initialize the value even if it's declared as private (or protected) in the class.

If the class is defined in a .h file, the static member definition is usually placed in the associated code file for the class (e.g. Something.cpp). If the class is defined in a .cpp file, the static member definition is usually placed directly underneath the class. Do not put the static member definition in a header file (much like a global variable, if that header file gets included more than once, you'll end up with multiple definitions, which will cause a compile error).

There is one exception where a static member definition line is not required: when the static member is of type const integer or const enum. Those can be initialized directly on the line in which they are declared:

```
class Whatever
{
public:
    static const int s_value = 4; // a static const int can be declared and initialized directl
y
};
```

In the above example, because the static member variable is a const int, no explicit definition line is needed.

An example of static member variables

Why use static variables inside classes? One great example is to assign a unique ID to every instance of the class. Here's an example of that:

```
class Something
{
private:
    static int s_idGenerator;
    int m_id;

public:
    Something() { m_id = s_idGenerator++; } // grab the next value from the id generator

    int getID() const { return m_id; }
};

// Note that we're defining and initializing s_idGenerator even though it is declared as priva
```

```
14 te above.
15 // This is okay since the definition isn't subject to access controls.
16 int Something::s_idGenerator = 1; // start our ID generator with value 1
17
18 int main()
19 {
20     Something first;
21     Something second;
22     Something third;
23
24     std::cout << first.getID() << '\n';
25     std::cout << second.getID() << '\n';
26     std::cout << third.getID() << '\n';
27     return 0;
   }
```

This program prints:

```
1
2
3
```

Because s_idGenerator is shared by all Something objects, when a new Something object is created, the constructor grabs the current value out of s_idGenerator and then increments the value for the next object. This guarantees that each instantiated Something object receives a unique id (incremented in the order of creation). This can really help when debugging multiple items in an array, as it provides a way to tell multiple objects of the same class type apart!

Static member variables can also be useful when the class needs to utilize an internal lookup table (e.g. an array used to store a set of pre-calculated values). By making the lookup table static, only one copy exists for all objects, rather than making a copy for each object instantiated. This can save substantial amounts of memory.

 **8.12 -- Static member functions**

 **Index**

 **8.10 -- Const class objects and member functions**

Share this:

8.13 — Friend functions and classes

BY ALEX ON SEPTEMBER 20TH, 2007 | LAST MODIFIED BY ALEX ON APRIL 17TH, 2016

For much of this chapter, we've been preaching the virtues of keeping your data private. However, you may occasionally find situations where you will find you have classes and functions outside of those classes that need to work very closely together. For example, you might have a class that stores data, and a function (or another class) that displays the data on the screen. Although the storage class and display code have been separated for easier maintenance, the display code is really intimately tied to the details of the storage class. Consequently, there isn't much to gain by hiding the storage classes details from the display code.

In situations like this, there are two options:
1) Have the display code use the publicly exposed functions of the storage class. However, this has several potential downsides. First, these public member functions have to be defined, which takes time, and can clutter up the interface of the storage class. Second, the storage class may have to expose functions for the display code that it doesn't really want accessible to anybody else. There is no way to say "this function is meant to be used by the display class only".

2) Alternatively, using friend classes and friend functions, you can give your display code access to the private details of the storage class. This lets the display code directly access all the private members and functions of the storage class, while keeping everyone else out! In this lesson, we'll take a closer look at how this is done.

Friend functions

A **friend function** is a function that can access the private members of a class as though it were a member of that class. In all other regards, the friend function is just like a normal function. A friend function may be either a normal function, or a member function of another class. To declare a friend function, simply use the *friend* keyword in front of the prototype of the function you wish to be a friend of the class. It does not matter whether you declare the friend function in the private or public section of the class.

Here's an example of using a friend function:

```
class Accumulator
{
private:
    int m_value;
public:
    Accumulator() { m_value = 0; }
    void add(int value) { m_value += value; }

    // Make the reset() function a friend of this class
    friend void reset(Accumulator &accumulator);
};

// reset() is now a friend of the Accumulator class
void reset(Accumulator &accumulator)
{
    // And can access the private data of Accumulator objects
    accumulator.m_value = 0;
}

int main()
{
    Accumulator acc;
    acc.add(5); // add 5 to the accumulator
    reset(acc); // reset the accumulator to 0

    return 0;
}
```

In this example, we've declared a function named reset() that takes an object of class Accumulator, and sets the value of

m_value to 0. Because reset() is not a member of the Accumulator class, normally reset() would not be able to access the private members of Accumulator. However, because Accumulator has specifically declared this reset() function to be a friend of the class, the reset() function is given access to the private members of Accumulator.

Note that we have to pass an Accumulator object to reset(). This is because reset() is not a member function. It does not have a *this pointer, nor does it have an Accumulator object to work with, unless given one.

Here's another example:

```
class Value
{
private:
    int m_value;
public:
    Value(int value) { m_value = value; }
    friend bool isEqual(const Value &value1, const Value &value2);
};

bool isEqual(const Value &value1, const Value &value2)
{
    return (value1.m_value == value2.m_value);
}
```

In this example, we declare the isEqual() function to be a friend of the Value class. isEqual() takes two Value objects as parameters. Because isEqual() is a friend of the Value class, it can access the private members of all Value objects. In this case, it uses that access to do a comparison on the two objects, and returns true if they are equal.

While both of the above examples are fairly contrived, the latter example is very similar to cases we'll encounter in chapter 9 when we discuss operator overloading!

Multiple friends

A function can be a friend of more than one class at the same time. For example, consider the following example:

```
class Humidity;

class Temperature
{
private:
    int m_temp;
public:
    Temperature(int temp=0) { m_temp = temp; }

    void setTemperature(int temp) { m_temp = temp; }

    friend void printWeather(const Temperature &temperature, const Humidity &humidity);
};

class Humidity
{
private:
    int m_humidity;
public:
    Humidity(int humidity=0) { m_humidity = humidity; }

    void setHumidity(int humidity) { m_humidity = humidity; }

    friend void printWeather(const Temperature &temperature, const Humidity &humidity);
};

void printWeather(const Temperature &temperature, const Humidity &humidity)
{
    std::cout << "The temperature is " << temperature.m_temp <<
       " and the humidity is " << humidity.m_humidity << '\n';
}
```

```
32
33  int main()
34  {
35      Humidity hum(10);
36      Temperature temp(12);
37
38      printWeather(temp, hum);
39
40      return 0;
41  }
```

There are two things worth noting about this example. First, because PrintWeather is a friend of both classes, it can access the private data from objects of both classes. Second, note the following line at the top of the example:

```
1  class Humidity;
```

This is a class prototype that tells the compiler that we are going to define a class called Humidity in the future. Without this line, the compiler would tell us it doesn't know what a Humidity is when parsing the prototype for PrintWeather() inside the Temperature class. Class prototypes serve the same role as function prototypes -- they tell the compiler what something looks like so it can be used now and defined later. However, unlike functions, classes have no return types or parameters, so class prototypes are always simply `class ClassName`, where ClassName is the name of the class.

Friend classes

It is also possible to make an entire class a friend of another class. This gives all of the members of the friend class access to the private members of the other class. Here is an example:

```
1   class Storage
2   {
3   private:
4       int m_nValue;
5       double m_dValue;
6   public:
7       Storage(int nValue, double dValue)
8       {
9           m_nValue = nValue;
10          m_dValue = dValue;
11      }
12
13      // Make the Display class a friend of Storage
14      friend class Display;
15  };
16
17  class Display
18  {
19  private:
20      bool m_displayIntFirst;
21
22  public:
23      Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
24
25      void displayItem(Storage &storage)
26      {
27          if (m_displayIntFirst)
28              std::cout << storage.m_nValue << " " << storage.m_dValue << '\n';
29          else // display double first
30              std::cout << storage.m_dValue << " " << storage.m_nValue << '\n';
31      }
32  };
33
34  int main()
35  {
36      Storage storage(5, 6.7);
37      Display display(false);
38
39      display.displayItem(storage);
```

```
40
41      return 0;
42  }
```

Because the Display class is a friend of Storage, any of Display's members that use a Storage class object can access the private members of Storage directly. This program produces the following result:

```
6.7 5
```

A few additional notes on friend classes. First, even though Display is a friend of Storage, Display has no direct access to the *this pointer of Storage objects. Second, just because Display is a friend of Storage, that does not mean Storage is also a friend of Display. If you want two classes to be friends of each other, both must declare the other as a friend. Finally, if class A is a friend of B, and B is a friend of C, that does not mean A is a friend of C.

Be careful when using friend functions and classes, because it allows the friend function or class to violate encapsulation. If the details of the class change, the details of the friend will also be forced to change. Consequently, limit your use of friend functions and classes to a minimum.

Friend member functions

Instead of making an entire class a friend, you can make a single member function a friend. This is done similarly to making a normal function a friend, except using the name of the member function with the className:: prefix included (e.g. Display::displayItem).

However, in actuality, this can be a little trickier than expected. Let's convert the previous example to make Display::displayItem a friend member function. You might try something like this:

```
1   class Display; // forward declaration for class Display
2
3   class Storage
4   {
5   private:
6       int m_nValue;
7       double m_dValue;
8   public:
9       Storage(int nValue, double dValue)
10      {
11          m_nValue = nValue;
12          m_dValue = dValue;
13      }
14
15      // Make the Display class a friend of Storage
16      friend void Display::displayItem(Storage& storage); // error: Storage hasn't see the full
17  declaration of class Display
18  };
19
20  class Display
21  {
22  private:
23      bool m_displayIntFirst;
24
25  public:
26      Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
27
28      void displayItem(Storage &storage)
29      {
30          if (m_displayIntFirst)
31              std::cout << storage.m_nValue << " " << storage.m_dValue << '\n';
32          else // display double first
33              std::cout << storage.m_dValue << " " << storage.m_nValue << '\n';
34      }
    };
```

However, it turns out this won't work. In order to make a member function a friend, the compiler has to have seen the full declaration for the class of the friend member function (not just a forward declaration). Since class Storage hasn't seen the full declaration for class Display yet, the compiler will error at the point where we try to make the member function a friend.

To resolve this, we can switch the order of class Display and class Storage. We will also need to move the definition of Display::displayItem() out of the Display class declaration, because it needs to have seen the definition of class Storage first.

```
class Storage; // forward declaration for class Storage

class Display
{
private:
    bool m_displayIntFirst;

public:
    Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }

    void displayItem(Storage &storage); // forward declaration above needed for this declarati
on line
};

class Storage
{
private:
    int m_nValue;
    double m_dValue;
public:
    Storage(int nValue, double dValue)
    {
        m_nValue = nValue;
        m_dValue = dValue;
    }

    // Make the Display class a friend of Storage (requires seeing the full declaration of cla
ss Display, as above)
    friend void Display::displayItem(Storage& storage);
};

// Now we can define Display::displayItem, which needs to have seen the full declaration of cl
ass Storage
void Display::displayItem(Storage &storage)
{
    if (m_displayIntFirst)
        std::cout << storage.m_nValue << " " << storage.m_dValue << '\n';
    else // display double first
        std::cout << storage.m_dValue << " " << storage.m_nValue << '\n';
}

int main()
{
    Storage storage(5, 6.7);
    Display display(false);

    display.displayItem(storage);

    return 0;
}
```

Now, this will compile, and Display::displayItem is a friend of class Storage.

However, a better solution would have been to put each class declaration in a separate header file, with the function bodies in corresponding .cpp files. That way, all of the class declarations would have been visible immediately, and no rearranging of classes or functions would have been necessary.

Summary

A friend function or class is a function or class that can access the private members of another class as though it were a member of that class. This allows the friend or class to work intimately with the other class, without making the other class expose its private members (e.g. via access functions).

Friending is uncommonly used when two or more classes need to work together in an intimate way, or much more commonly, when defining overloading operators (which we'll cover in chapter 9).

Note that making a class a friend only requires as forward declaration that the class exists. However, making a specific member function a friend requires the full declaration for the class of the member function to have been seen first.

Quiz time

1) In geometry, a point is a position in space. We can define a point in 3d-space as the set of coordinates x, y, and z. For example, the Point(2.0, 1.0, 0.0) would be the point at coordinate space x=2.0, y=1.0, and z=0.0.

In physics, a vector is a quantity that has a magnitude (length) and a direction (but no position). We can define a vector in 3d-space as an x, y, and z value representing the direction of the vector along the x, y, and z axis (the length can be derived from these). For example, the Vector(2.0, 0.0, 0.0) would be a vector representing a direction along the positive x-axis (only), with length 2.0.

A Vector can be applied to a Point to move the Point to a new position. This is done by adding the vector's direction to the point's position to yield a new position. For example, Point(2.0, 1.0, 0.0) + Vector(2.0, 0.0, 0.0) would yield the point (4.0, 1.0, 0.0).

Points and Vectors are often used in computer graphics (the point to represent vertices of shape, and vectors represent movement of the shape).

Given the following program:

```
#include <iostream>

class Vector3d
{
private:
    double m_x = 0.0, m_y = 0.0, m_z = 0.0;

public:
    Vector3d(double x = 0.0, double y = 0.0, double z = 0.0)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    void print()
    {
        std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";
    }
};

class Point3d
{
private:
    double m_x=0.0, m_y=0.0, m_z=0.0;

public:
    Point3d(double x = 0.0, double y = 0.0, double z = 0.0)
        : m_x(x), m_y(y), m_z(z)
    {

    }

    void print()
    {
        std::cout << "Point(" << m_x << " , " << m_y << " , " << m_z << ")\n";
```

```
36      }
37
38      void moveByVector(Vector3d &v)
39      {
40          // implement this function as a friend of class Vector3d
41      }
42  };
43
44  int main()
45  {
46      Point3d p(1.0, 2.0, 3.0);
47      Vector3d v(2.0, 2.0, -2.0);
48
49      p.print();
50      p.moveByVector(v);
51      p.print();
52
53      return 0;
54  }
```

1a) Make Point3d a friend class of Vector3d, and implement function Point3d::moveByVector()

Show Solution

1b) Instead of making class Point3d a friend of class Vector3d, make member function Point3d::moveByVector a friend of class Vector3d.

Show Solution

 **8.14 -- Anonymous variables and objects**

 **Index**

 **8.12 -- Static member functions**

Share this:

10.2 — Composition

BY ALEX ON DECEMBER 4TH, 2007 | LAST MODIFIED BY ALEX ON FEBRUARY 28TH, 2016

In real-life, complex objects are often built from smaller, simpler objects. For example, a car is built using a metal frame, an engine, some tires, a transmission, a steering wheel, and a large number of other parts. A personal computer is built from a CPU, a motherboard, some memory, etc… Even you are built from smaller parts: you have a head, a body, some legs, arms, and so on. This process of building complex objects from simpler ones is called **composition** (also known as object composition).

More specifically, composition is used for objects that have a *has-a* relationship to each other. A car *has-a* metal frame, *has-an* engine, and *has-a* transmission. A personal computer *has-a* CPU, a motherboard, and other components. You *have-a* head, a body, some limbs.

So far, all of the classes we have used in our examples have had member variables that are built-in data types (eg. int, double). While this is generally sufficient for designing and implementing small, simple classes, it quickly becomes burdensome for more complex classes, especially those built from many sub-parts. In order to facilitate the building of complex classes from simpler ones, C++ allows us to do object composition in a very simple way -- by using classes as member variables in other classes.

Lets take a look at some examples of how this is done. If we were designing a personal computer class, we might do it like this (assuming we'd already written a CPU, Motherboard, and RAM class):

```
#include "CPU.h"
#include "Motherboard.h"
#include "RAM.h"

class PersonalComputer
{
private:
    CPU m_cCPU;
    Motherboard m_cMotherboard;
    RAM m_cRAM;
};
```

Initializing class member variables

In the previous lesson on **initializer lists**, you learned that the preferred way to initialize class members is through initializer lists rather than assignment. So let's write a constructor for our PersonalComputer class that uses an initialization list to initialize the member variables. This constructor will take 3 parameters: a CPU speed, a motherboard model, and a RAM size, which it will then pass to the respective member variables when they are constructed.

```
PersonalComputer::PersonalComputer(int nCPUSpeed, char *strMotherboardModel, int nRAMSize)
    : m_cCPU(nCPUSpeed), m_cMotherboard(strMotherboardModel), m_cRAM(nRAMSize)
{
}
```

Now, when a PersonalComputer object is instantiated using this constructor, that PersonalComputer object will contain a CPU object initialized with nCPUSpeed, a Motherboard object initialized with strMotherboardModel, and a RAM object initialized with nRAMSize.

It is worth explicitly noting that composition implies ownership between the complex class and any subclasses. When the complex class is created, the subclasses are created. When the complex class is destroyed, the subclasses are similarly destroyed.

A full example

While the above example is useful in giving the general idea of how composition works, let's do a full example that you can compile yourself. Many games and simulations have creatures or objects that move around a board, map, or screen. The one

thing that all of these creatures/objects have in common is that they all *have-a* location. In this example, we are going to create a creature class that uses a point class to hold the creature's location.

First, let's design the point class. Our creature is going to live in a 2d world, so our point class will have 2 dimensions, X and Y. We will assume the world is made up of discrete squares, so these dimensions will always be integers.

Point2D.h:

```
#ifndef POINT2D_H
#define POINT2D_H

#include <iostream>

class Point2D
{
private:
    int m_nX;
    int m_nY;

public:
    // A default constructor
    Point2D()
        : m_nX(0), m_nY(0)
    {
    }

    // A specific constructor
    Point2D(int nX, int nY)
        : m_nX(nX), m_nY(nY)
    {
    }

    // An overloaded output operator
    friend std::ostream& operator<<(std::ostream& out, const Point2D &cPoint)
    {
        out << "(" << cPoint.GetX() << ", " << cPoint.GetY() << ")";
        return out;
    }

    // Access functions
    void SetPoint(int nX, int nY)
    {
        m_nX = nX;
        m_nY = nY;
    }

    int GetX() const { return m_nX; }
    int GetY() const { return m_nY; }
};

#endif
```

Note that because we've implemented all of our functions in the header file (for the sake of keeping the example concise), there is no Point2D.cpp.

Now let's design our Creature. Our Creature is going to have a few properties. It's going to have a name, which will be a string, and a location, which will be our Point2D class.

Creature.h:

```
#ifndef CREATURE_H
#define CREATURE_H

#include <iostream>
#include <string>
#include "Point2D.h"
```

```
7
8   class Creature
9   {
10  private:
11      std::string m_strName;
12      Point2D m_cLocation;
13
14  public:
15      Creature(std::string strName, const Point2D &cLocation)
16          : m_strName(strName), m_cLocation(cLocation)
17      {
18      }
19
20      friend std::ostream& operator<<(std::ostream& out, const Creature &cCreature)
21      {
22          out << cCreature.m_strName << " is at " << cCreature.m_cLocation;
23          return out;
24      }
25
26      void MoveTo(int nX, int nY)
27      {
28          m_cLocation.SetPoint(nX, nY);
29      }
30  };
31  #endif
```

And finally, Main.cpp:

```
1   #include <string>
2   #include <iostream>
3   #include "Creature.h"
4   #include "Point2D.h"
5
6   int main()
7   {
8       using namespace std;
9       cout << "Enter a name for your creature: ";
10      std::string cName;
11      cin >> cName;
12      Creature cCreature(cName, Point2D(4, 7));
13
14      while (1)
15      {
16          cout << cCreature << endl;
17          cout << "Enter new X location for creature (-1 to quit): ";
18          int nX=0;
19          cin >> nX;
20          if (nX == -1)
21              break;
22
23          cout << "Enter new Y location for creature (-1 to quit): ";
24          int nY=0;
25          cin >> nY;
26          if (nY == -1)
27              break;
28
29          cCreature.MoveTo(nX, nY);
30          }
31
32      return 0;
33  }
```

Here's a transcript of this code being run:

```
Enter a name for your creature: Marvin
Marvin is at (4, 7)
Enter new X location for creature (-1 to quit): 6
```

```
Enter new Y location for creature (-1 to quit): 12
Marvin is at (6, 12)
Enter new X location for creature (-1 to quit): 3
Enter new Y location for creature (-1 to quit): 2
Marvin is at (3, 2)
Enter new X location for creature (-1 to quit): -1
```

Why use composition?

Instead of using the Point2D class to implement the Creature's location, we could have instead just added 2 integers to the Creature class and written code in the Creature class to handle the positioning. However, using composition provides a number of useful benefits:

1. Each individual class can be kept relatively simple and straightforward, focused on performing one task. This makes those classes easier to write and much easier to understand. For example, Point2D only worries about point-related stuff, which helps keep it simple.
2. Each subobject can be self-contained, which makes them reusable. For example, we could reuse our Point2D class in a completely different application. Or if our creature ever needed another point (for example, a destination it was trying to get to), we can simply add another Point2D member variable.
3. The complex class can have the simple subclasses do most of the hard work, and instead focus on coordinating the data flow between the subclasses. This helps lower the overall complexity of the complex object, because it can delegate tasks to the sub-objects, who already know how to do them. For example, when we move our Creature, it delegates that task to the Point class, which already understands how to set a point. Thus, the Creature class does not have to worry about how such things would be implemented.

One question that new programmers often ask is "When should I use composition instead of direct implementation of a feature?". There's no 100% answer to that question. However, a good rule of thumb is that each class should be built to accomplish a single task. That task should either be the storage and manipulation of some kind of data (eg. Point2D), OR the coordination of subclasses (eg. Creature). Not both.

In this case of our example, it makes sense that Creature shouldn't have to worry about how Points are implemented, or how the name is being stored. Creature's job isn't to know those intimate details. Creature's job is to worry about how to coordinate the data flow and ensure that each of the subclasses knows *what* it is supposed to do. It's up to the individual subclasses to worry about *how* they will do it.


10.3 -- Aggregation


Index


9.12 -- Shallow vs deep copying

Share this:

10.3 — Aggregation

BY ALEX ON DECEMBER 7TH, 2007 | LAST MODIFIED BY ALEX ON DECEMBER 7TH, 2007

In the previous lesson on **composition**, you learned that compositions are complex classes that contain other subclasses as member variables. In addition, in a composition, the complex object "owns" all of the subobjects it is composed of. When a composition is destroyed, all of the subobjects are destroyed as well. For example, if you destroy a car, it's frame, engine, and other parts should be destroyed as well. If you destroy a PC, you would expect it's RAM and CPU to be destroyed as well.

Aggregration

An **aggregation** is a specific type of composition where no ownership between the complex object and the subobjects is implied. When an aggregate is destroyed, the subobjects are not destroyed.

For example, consider the math department of a school, which is made up of one or more teachers. Because the department does not own the teachers (they merely work there), the department should be an aggregate. When the department is destroyed, the teachers should still exist independently (they can go get jobs in other departments).

Because aggregations are just a special type of compositions, they are implemented almost identically, and the difference between them is mostly semantic. In a composition, we typically add our subclasses to the composition using either normal variables or pointers where the allocation and deallocation process is handled by the composition class.

In an aggregation, we also add other subclasses to our complex aggregate class as member variables. However, these member variables are typically either references or pointers that are used to point at objects that have been created outside the scope of the class. Consequently, an aggregate class usually either takes the objects it is going to point to as constructor parameters, or it begins empty and the subobjects are added later via access functions or operators.

Because these subclass objects live outside of the scope of the class, when the class is destroyed, the pointer or reference member variable will be destroyed, but the subclass objects themselves will still exist.

Let's take a look at our Teacher and Department example in more detail.

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

class Teacher
{
private:
    string m_strName;
public:
    Teacher(string strName)
        : m_strName(strName)
    {
    }

    string GetName() { return m_strName; }
};

class Department
{
private:
    Teacher *m_pcTeacher; // This dept holds only one teacher

public:
    Department(Teacher *pcTeacher=NULL)
        : m_pcTeacher(pcTeacher)
    {
    }
};

```

```
int main()
{
    // Create a teacher outside the scope of the Department
    Teacher *pTeacher = new Teacher("Bob"); // create a teacher
    {
        // Create a department and use the constructor parameter to pass
        // the teacher to it.
        Department cDept(pTeacher);

    } // cDept goes out of scope here and is destroyed

    // pTeacher still exists here because cDept did not destroy it
    delete pTeacher;
}
```

In this case, pTeacher is created independetly of cDept, and then passed into cDept's constructor. Note that the department class uses an initialization list to set the value of m_pcTeacher to the pTeacher value we passed in. When cDept is destroyed, the m_pcTeacher pointer destroyed, but pTeacher is not deallocated, so it still exists until it is independently destroyed.

To summarize the differences between composition and aggregation:

Compositions:

- Typically use normal member variables
- Can use pointer values if the composition class automatically handles allocation/deallocation
- Responsible for creation/destruction of subclasses

Aggregations:

- Typically use pointer variables that point to an object that lives outside the scope of the aggregate class
- Can use reference values that point to an object that lives outside the scope of the aggregate class
- Not responsible for creating/destroying subclasses

It is worth noting that the concepts of composition and aggregation are not mutually exclusive, and can be mixed freely within the same class. It is entirely possible to write a class that is responsible for the creation/destruction of some subclasses but not others. For example, our Department class could have a name and a teacher. The name would probably be added to the department by composition, and would be created and destroyed with the department. On the other hand, the teacher would be added to the department by aggregate, and created/destroyed independently.

It is also possible to create other hybrid aggregate/composition schemes, such as where a class holds independent subobjects like an aggregate, but will destroy them when the class goes out of scope like a composition.

While aggregates can be extremely useful (which we will see more of in the next lesson on container classes), they are also potentially dangerous. As noted several times, aggregates are not responsible for deallocating their subobjects when they are destroyed. Consequently, if there are no other pointers or references to those subobjects when the aggregate is destroyed, those subobjects will cause a memory leak. It is up to the programmer to ensure that this does not happen. This is generally handled by ensuring other pointers or references to those subobjects exist when the aggregate is destroyed.

 **10.4 -- Container classes**

 **Index**

 **10.2 -- Composition**

10.4 — Container classes

BY ALEX ON DECEMBER 14TH, 2007 | LAST MODIFIED BY ALEX ON OCTOBER 1ST, 2015

In real life, we use containers all the time. Your breakfast cereal comes in a box, the pages in your book come inside a cover and binding, and you might store any number of items in containers in your garage. Without containers, it would be extremely inconvenient to work with many of these objects. Imagine trying to read a book that didn't have any sort of binding, or eat cereal that didn't come in a box without using a bowl. It would be a mess. The value the container provides is largely in it's ability to help organize and store items that are put inside it.

Similarly, a **container class** is a class designed to hold and organize multiple instances of another class. There are many different kinds of container classes, each of which has various advantages, disadvantages, and restrictions in their use. By far the most commonly used container in programming is the **array**, which you have already seen many examples of. Although C++ has built-in array functionality, programmers will often use an array container class instead because of the additional benefits it provides. Unlike built-in arrays, array container classes generally provide dynamically resizing (when elements are added or removed) and do bounds-checking. This not only makes array container classes more convenient than normal arrays, but safer too.

Container classes typically implement a fairly standardized minimal set of functionality. Most well-defined containers will include functions that:

- Create an empty container (via a constructor)
- Insert a new object into the container
- Remove an object from the container
- Report the number of objects currently in the container
- Empty the container of all objects
- Provide access to the stored objects
- Sort the elements (optional)

Sometimes certain container classes will omit some of this functionality. For example, arrays container classes often omit the insert and delete functions because they are slow and the class designer does not want to encourage their use.

Container classes generally come in two different varieties. **Value containers** are **compositions** that store copies of the objects that they are holding (and thus are responsible for creating and destroying those copies). **Reference containers** are **aggregations** that store pointers or references to other objects (and thus are not responsible for creation or destruction of those objects).

Unlike in real life, where containers can hold whatever you put in them, in C++, containers typically only hold one type of data. For example, if you have an array of integers, it will only hold integers. Unlike some other languages, C++ generally does not allow you to mix types inside a container. If you want one container class that holds integers and another that holds doubles, you will have to write two separate containers to do this (or use templates, which is an advanced C++ feature). Despite the restrictions on their use, containers are immensely useful, and they make programming easier, safer, and faster.

An array container class

In this example, we are going to write an integer array class that implements most of the common functionality that containers should have. This array class is going to be a value container, which will hold copies of the elements its organizing.

First, let's create the IntArray.h file:

```
1 #ifndef INTARRAY_H
2 #define INTARRAY_H
3
4 class IntArray
5 {
6 };
7
```

```
#endif
```

Our IntArray is going to need to keep track of two values: the data itself, and the size of the array. Because we want our array to be able to change in size, we'll have to do some dynamic allocation, which means we'll have to use a pointer to store the data.

```
#ifndef INTARRAY_H
#define INTARRAY_H

class IntArray
{
private:
    int m_nLength;
    int *m_pnData;
};

#endif
```

Now we need to add some constructors that will allow us to create IntArrays. We are going to add two constructors: one that constructs an empty array, and one that will allow us to construct an array of a predetermined size.

```
#ifndef INTARRAY_H
#define INTARRAY_H

class IntArray
{
private:
    int m_nLength;
    int *m_pnData;

public:
    IntArray()
    {
        m_nLength = 0;
        m_pnData = 0;
    }

    IntArray(int nLength)
    {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }
};

#endif
```

We'll also need some functions to help us clean up IntArrays. First, we'll write a destructor, which simply deallocates any dynamically allocated data. Second, we'll write a function called Erase(), which will erase the array and set the length to 0.

```
    ~IntArray()
    {
        delete[] m_pnData;
    }

    void Erase()
    {
        delete[] m_pnData;
        // We need to make sure we set m_pnData to 0 here, otherwise it will
        // be left pointing at deallocated memory!
        m_pnData = 0;
        m_nLength = 0;
    }
```

Now let's overload the [] operator so we can access the elements of the array. We should bounds check the index to make sure it's valid, which is best done using the assert() function. We'll also add an access function to return the length of the array.

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <assert.h> // for assert()

class IntArray
{
private:
    int m_nLength;
    int *m_pnData;

public:
    IntArray()
    {
        m_nLength = 0;
        m_pnData = 0;
    }

    IntArray(int nLength)
    {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }

    ~IntArray()
    {
        delete[] m_pnData;
    }

    void Erase()
    {
        delete[] m_pnData;
        // We need to make sure we set m_pnData to 0 here, otherwise it will
        // be left pointing at deallocated memory!
        m_pnData = 0;
        m_nLength = 0;
    }

    int& operator[](int nIndex)
    {
        assert(nIndex >= 0 && nIndex < m_nLength);
        return m_pnData[nIndex];
    }

    int GetLength() { return m_nLength; }
};

#endif
```

At this point, we already have an IntArray class that we can use. We can allocate IntArrays of a given size, and we can use the [] operator to retrieve or change the value of the elements.

However, there are still a few thing we can't do with our IntArray. We still can't change it's size, still can't insert or delete elements, and we still can't sort it.

First, let's write some code that will allow us to resize an array. We are going to write two different functions to do this. The first function, Reallocate(), will destroy any existing elements in the array when it is resized, but it will be fast. The second function, Resize(), will keep any existing elements in the array when it is resized, but it will be slow.

```
    // Reallocate resizes the array.  Any existing elements will be destroyed.
    // This function operates quickly.
    void Reallocate(int nNewLength)
    {
        // First we delete any existing elements
        Erase();

```

```
        // If our array is going to be empty now, return here
        if (nNewLength<= 0)
            return;

        // Then we have to allocate new elements
        m_pnData = new int[nNewLength];
        m_nLength = nNewLength;
    }

    // Resize resizes the array.  Any existing elements will be kept.
    // This function operates slowly.
    void Resize(int nNewLength)
    {
        // If we are resizing to an empty array, do that and return
        if (nNewLength <= 0)
        {
            Erase();
            return;
        }

        // Now we can assume nNewLength is at least 1 element.  This algorithm
        // works as follows: First we are going to allocate a new array.  Then we
        // are going to copy elements from the existing array to the new array.
        // Once that is done, we can destroy the old array, and make m_pnData
        // point to the new array.

        // First we have to allocate a new array
        int *pnData = new int[nNewLength];

        // Then we have to figure out how many elements to copy from the existing
        // array to the new array.  We want to copy as many elements as there are
        // in the smaller of the two arrays.
        if (m_nLength > 0)
        {
            int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength : nNewLength;

            // Now copy the elements one by one
            for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
                pnData[nIndex] = m_pnData[nIndex];
        }

        // Now we can delete the old array because we don't need it any more
        delete[] m_pnData;

        // And use the new array instead!  Note that this simply makes m_pnData point
        // to the same address as the new array we dynamically allocated.  Because
        // pnData was dynamically allocated, it won't be destroyed when it goes out of scope.
        m_pnData = pnData;
        m_nLength = nNewLength;
    }
```

Whew! That was a little tricky!

Many array container classes would stop here. However, just in case you want to see how insert and delete functionality would be implemented we'll go ahead and write those too. Both of these algorithms are very similar to Resize().

```
    void InsertBefore(int nValue, int nIndex)
    {
        // Sanity check our nIndex value
        assert(nIndex >= 0 && nIndex <= m_nLength);

        // First create a new array one element larger than the old array
        int *pnData = new int[m_nLength+1];

        // Copy all of the elements up to the index
        for (int nBefore=0; nBefore < nIndex; nBefore++)
            pnData[nBefore] = m_pnData[nBefore];
```

```

        // Insert our new element into the new array
        pnData[nIndex] = nValue;

        // Copy all of the values after the inserted element
        for (int nAfter=nIndex; nAfter < m_nLength; nAfter++)
            pnData[nAfter+1] = m_pnData[nAfter];

        // Finally, delete the old array, and use the new array instead
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength += 1;
    }

    void Remove(int nIndex)
    {
        // Sanity check our nIndex value
        assert(nIndex >= 0 && nIndex < m_nLength);

        // First create a new array one element smaller than the old array
        int *pnData = new int[m_nLength-1];

        // Copy all of the elements up to the index
        for (int nBefore=0; nBefore < nIndex; nBefore++)
            pnData[nBefore] = m_pnData[nBefore];

        // Copy all of the values after the removed element
        for (int nAfter=nIndex+1; nAfter < m_nLength; nAfter++)
            pnData[nAfter-1] = m_pnData[nAfter];

        // Finally, delete the old array, and use the new array instead
        delete[] m_pnData;
        m_pnData = pnData;
        m_nLength -= 1;
    }

    // A couple of additional functions just for convenience
    void InsertAtBeginning(int nValue) { InsertBefore(nValue, 0); }
    void InsertAtEnd(int nValue) { InsertBefore(nValue, m_nLength); }
```

Here is our IntArray container class in it's entirety:

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <assert.h> // for assert()

class IntArray
{
private:
    int m_nLength;
    int *m_pnData;

public:
    IntArray()
    {
        m_nLength = 0;
        m_pnData = 0;
    }

    IntArray(int nLength)
    {
        m_pnData = new int[nLength];
        m_nLength = nLength;
    }

    ~IntArray()
```

```
26      {
27          delete[] m_pnData;
28      }
29
30      void Erase()
31      {
32          delete[] m_pnData;
33          // We need to make sure we set m_pnData to 0 here, otherwise it will
34          // be left pointing at deallocated memory!
35          m_pnData = 0;
36          m_nLength = 0;
37      }
38
39      int& operator[](int nIndex)
40      {
41          assert(nIndex >= 0 && nIndex < m_nLength);
42          return m_pnData[nIndex];
43      }
44
45      // Reallocate resizes the array.  Any existing elements will be destroyed.
46      // This function operates quickly.
47      void Reallocate(int nNewLength)
48      {
49          // First we delete any existing elements
50          Erase();
51
52          // If our array is going to be empty now, return here
53          if (nNewLength<= 0)
54              return;
55
56          // Then we have to allocate new elements
57          m_pnData = new int[nNewLength];
58          m_nLength = nNewLength;
59      }
60
61      // Resize resizes the array.  Any existing elements will be kept.
62      // This function operates slowly.
63      void Resize(int nNewLength)
64      {
65          // If we are resizing to an empty array, do that and return
66          if (nNewLength <= 0)
67          {
68              Erase();
69              return;
70          }
71
72          // Now we can assume nNewLength is at least 1 element.  This algorithm
73          // works as follows: First we are going to allocate a new array.  Then we
74          // are going to copy elements from the existing array to the new array.
75          // Once that is done, we can destroy the old array, and make m_pnData
76          // point to the new array.
77
78          // First we have to allocate a new array
79          int *pnData = new int[nNewLength];
80
81          // Then we have to figure out how many elements to copy from the existing
82          // array to the new array.  We want to copy as many elements as there are
83          // in the smaller of the two arrays.
84          if (m_nLength > 0)
85          {
86              int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength : nNewLength;
87
88              // Now copy the elements one by one
89              for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
90                  pnData[nIndex] = m_pnData[nIndex];
91          }
92
```

```
93          // Now we can delete the old array because we don't need it any more
94          delete[] m_pnData;
95
96          // And use the new array instead!  Note that this simply makes m_pnData point
97          // to the same address as the new array we dynamically allocated.  Because
98          // pnData was dynamically allocated, it won't be destroyed when it goes out of scope.
99          m_pnData = pnData;
100         m_nLength = nNewLength;
101     }
102
103
104         void InsertBefore(int nValue, int nIndex)
105     {
106         // Sanity check our nIndex value
107         assert(nIndex >= 0 && nIndex <= m_nLength);
108
109         // First create a new array one element larger than the old array
110         int *pnData = new int[m_nLength+1];
111
112         // Copy all of the elements up to the index
113         for (int nBefore=0; nBefore < nIndex; nBefore++)
114             pnData[nBefore] = m_pnData[nBefore];
115
116         // insert our new element into the new array
117         pnData[nIndex] = nValue;
118
119         // Copy all of the values after the inserted element
120         for (int nAfter=nIndex; nAfter < m_nLength; nAfter++)
121             pnData[nAfter+1] = m_pnData[nAfter];
122
123         // Finally, delete the old array, and use the new array instead
124         delete[] m_pnData;
125         m_pnData = pnData;
126         m_nLength += 1;
127     }
128
129     void Remove(int nIndex)
130     {
131         // Sanity check our nIndex value
132         assert(nIndex >= 0 && nIndex < m_nLength);
133
134         // First create a new array one element smaller than the old array
135         int *pnData = new int[m_nLength-1];
136
137         // Copy all of the elements up to the index
138         for (int nBefore=0; nBefore < nIndex; nBefore++)
139             pnData[nBefore] = m_pnData[nBefore];
140
141         // Copy all of the values after the inserted element
142         for (int nAfter=nIndex+1; nAfter < m_nLength; nAfter++)
143             pnData[nAfter-1] = m_pnData[nAfter];
144
145         // Finally, delete the old array, and use the new array instead
146         delete[] m_pnData;
147         m_pnData = pnData;
148         m_nLength -= 1;
149     }
150
151     // A couple of additional functions just for convenience
152     void InsertAtBeginning(int nValue) { InsertBefore(nValue, 0); }
153     void InsertAtEnd(int nValue) { InsertBefore(nValue, m_nLength); }
154
155     int GetLength() { return m_nLength; }
156 };
157
158 #endif
```

Now, let's test it just to prove it works:

```
#include <iostream>
#include "IntArray.h"

using namespace std;

int main()
{
    // Declare an array with 10 elements
    IntArray cArray(10);

    // Fill the array with numbers 1 through 10
    for (int i=0; i<10; i++)
        cArray[i] = i+1;

    // Resize the array to 8 elements
    cArray.Resize(8);

    // Insert the number 20 before the 5th element
    cArray.InsertBefore(20, 5);

    // Remove the 3rd element
    cArray.Remove(3);

    // Add 30 and 40 to the end and beginning
    cArray.InsertAtEnd(30);
    cArray.InsertAtBeginning(40);

    // Print out all the numbers
    for (int j=0; j<cArray.GetLength(); j++)
        cout << cArray[j] << " ";

    return 0;
}
```

This produces the result:

```
40 1 2 3 5 20 6 7 8 30
```

Although writing container classes can be pretty complex, the good news is that you only have to write them once. Once the container class is working, you can use and reuse it as often as you like without any additional programming effort required.

It is also worth explicitly mentioning that even though our sample IntArray container class holds a built-in data type (int), we could have just as easily used a user-defined type (eg. a point class).

 **11.1 -- Introduction to inheritance**

 **Index**

 **10.3 -- Aggregation**

Share this:

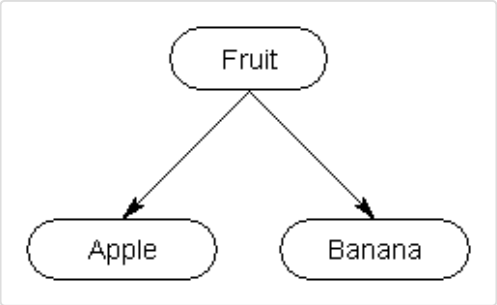
11.1 — Introduction to inheritance

BY ALEX ON DECEMBER 19TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 13TH, 2016

In the lesson on **composition**, you learned how to construct complex classes by combining simpler classes. Composition is perfect for building new objects that have a *has-a* relationship with their subobjects. However, composition (and aggregation) is just one of the two major ways that C++ lets you construct complex classes. The second way is through inheritance.

Unlike composition, which involves creating new objects by combining and connecting other objects, **inheritance** involves creating new objects by directly acquiring the attributes and behaviors of other objects and then extending or specializing them. Like composition, inheritance is everywhere in real life. You inherited your parents genes, and acquired physical attributes from both of them. Technological products (computers, cell phones, etc…) often inherit features from their predecessors. C++ inherited many features from C, the language upon which it is based, and C itself inherited many of its features from the programming languages that came before it.

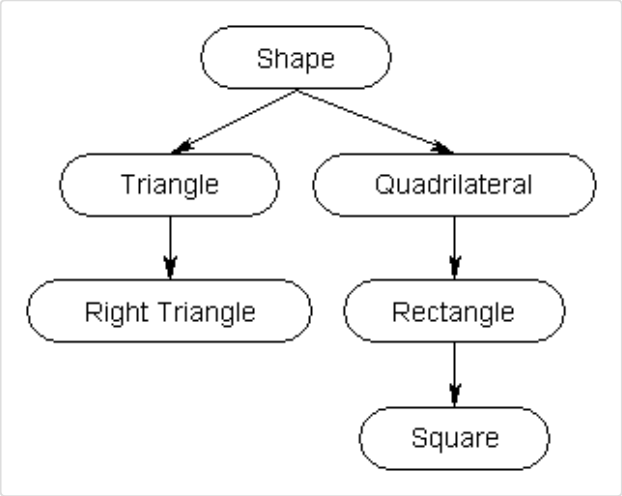
Consider an apple and a banana. Although an apple and a banana are different fruits, both have in common that they *are* fruits. Because apples and bananas *are* fruits, anything that is true of fruits is also true of apples and bananas. For example, all fruits have a name, a flavor, and are tangible objects. Thus, apples and bananas also have a name, a flavor, and are tangible objects. Apples and bananas inherit these properties from the concept of fruit because they *are* fruit. Apples and bananas then define some of these properties in different ways (apples and bananas have different flavors), which is what makes them distinct from each other.



The object being inherited from is called the **parent** or **base**, and the object doing the inheriting is called the **child** or **derived** object. In the above picture, “fruit” is the parent, and both “apple” and “banana” are children. Unlike in composition, where each object has a *has-a* relationship with its subobjects, in inheritance, each child has an *is-a* relationship with its parent. An apple *is-a* fruit. A triangle *is-a* shape. Red *is-a* color.

By default, the children receive all of the properties of the parents. However, the children are then free to define or redefine inherited properties (bananas have that unique banana flavor), add new properties (eg. bananas add the property of being “starchy”, which is not true of many other fruits), or even hide properties.

It is possible to define entire hierarchies of objects via inheritance. For example, a square is a rectangle, which is a quadrilateral, which is a shape. A right triangle is a triangle, which is also a shape.



Why the need for inheritance in C++?

One of the fundamental ideas behind object-oriented programming is that code should be reusable. However, existing code often does not do EXACTLY what you need it to. For example, what if you have a triangle and you need a square? In this case, we are presented with a number of choices on how to proceed, all of which have various benefits and downsides.

Perhaps the most obvious way to proceed is to change the existing code to do what you want. However, if we do this, we will no longer be able to use it for its original purpose, so this is rarely a good idea.

A slightly better idea is to make a copy of some or all of the existing code and change it to do what we want. However, this has several major downsides. First, although copy-and-paste seems simple enough, it's actually quite dangerous. A single omitted or misplaced line can cause the program to work incorrectly and can take days to find in a complex program. Renaming a class via search-and-replace can also be dangerous if you inadvertently replace something you didn't mean to. Second, to rewrite the code to make it do what you want, you need to have an intimate understanding what it does. This can be difficult when the code is complex and not adequately documented. Third, and perhaps most relevant, this generally involves duplicating of existing functionality, which causes a maintenance problem. Improvements or bug fixes have to be added to multiple copies of functions that do essentially the same thing, which wastes programmer time. And that's assuming the programmer realizes multiple copies even exist! If not, some copies may not get the improvements or bug fixes.

Inheritance solves most of these problems in an efficient way. Instead of manually copying and modifying every bit of code your program needs, inheritance allows you directly reuse existing code that meets your needs. You only need to add new features, redefine existing features that do not meet your needs, or hide features you do not want. This is typically much less work (as you are only defining what has changed compared to the base, rather than redefining everything), and safer too. Furthermore, any changes made to the base code automatically get propagated to the inherited code. This means it is possible to change one piece of code (eg. to apply a bug fix) and all derived objects will automatically be updated.

Inheritance does have a couple of potential downsides, but we will cover those in future lessons.

 **11.2 -- Basic inheritance in C++**

 **Index**

 **10.4 -- Container classes**

11.2 — Basic inheritance in C++

BY ALEX ON JANUARY 4TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 28TH, 2015

Now that we've talked about what inheritance is in an abstract sense, let's talk about how it's used within C++.

Inheritance in C++ takes place between classes. When one class inherits from another, the derived class inherits the variables and functions of the base class. These variables and functions become part of the derived class.

A Person base class

Here's a simple base class:

```
#include <string>
class Person
{
public:
    std::string m_strName;
    int m_nAge;
    bool m_bIsMale;

    std::string GetName() { return m_strName; }
    int GetAge() { return m_nAge; }
    bool IsMale() { return m_bIsMale; }

    Person(std::string strName = "", int nAge = 0, bool bIsMale = false)
        : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
    {
    }
};
```

This base class is meant to hold information about a person -- in this case, the name, age, and sex. There are two things to note here. First, we have only defined fields that are common to ALL people. This is a generic person class meant to be reused with anybody who is a person. Thus, it's appropriate to only include information used for all people.

Second, note that we've made all of our variables and functions public. This is purely for the sake of keeping these examples simple right now. Normally we would make the variables private. We will cover those cases in future lessons.

A BaseballPlayer derived class

Let's say we wanted to write a program that keeps track of information about some baseball players. Baseball players have information that only people who are baseball players -- for example, we might want to store a player's batting average, and the number of home runs they've hit. Here's our incomplete Baseball player class:

```
class BaseballPlayer
{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;
};
```

Note that we have not included the baseball player's name, age, or sex in this class, even though we want that information. While we could add member variables to hold this information directly to BaseballPlayer, we've already written a generic Person class that we can simply reuse to handle those details.

Logically, we know that BaseballPlayer and Person have some sort of relationship. Which makes more sense: a baseball player "has a" person, or a baseball player "is a" person? A baseball player "is a" person, therefore, our baseball player class will use inheritance rather than composition.

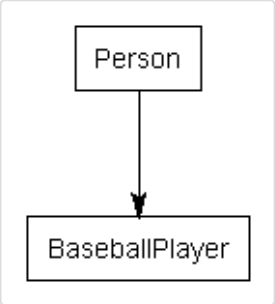
To inherit our Person class, the syntax is fairly simple. After the `class BaseballPlayer` declaration, we use a colon, the word "public", and the name of the class we wish to inherit. This is called *public inheritance*. We'll talk more about what

public inheritance means in a future section.

```
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;

    BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
       : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
    {
    }
};
```

Using a derivation diagram, our inheritance looks like this:



When BaseballPlayer inherits from Person, BaseballPlayer automatically receives the functions and variables from Person. Thus, BaseballPlayer objects will have 5 member variables (m_dBattingAverage and m_nHomeRuns from BaseballPlayer, and m_strName, m_nAge, and m_bIsMale from Person).

This is easy to prove:

```
#include <iostream>
int main()
{
    // Create a new BaseballPlayer object
    BaseballPlayer cJoe;
    // Assign it a name (we can do this directly because m_strName is public)
    cJoe.m_strName = "Joe";
    // Print out the name
    std::cout << cJoe.GetName() << std::endl;

    return 0;
}
```

Which prints the value:

```
Joe
```

This compiles and runs because cJoe is a BaseballPlayer, and all BaseballPlayer objects have a m_strName member variable that they inherit from the Person class.

An Employee derived class

Now let's write another class that also inherits from Person. This time, we'll write an Employee class. An employee "is a" person, so using inheritance is appropriate:

```
// Employee publicly inherits from Person
class Employee: public Person
{
public:
    std::string m_strEmployerName;
```

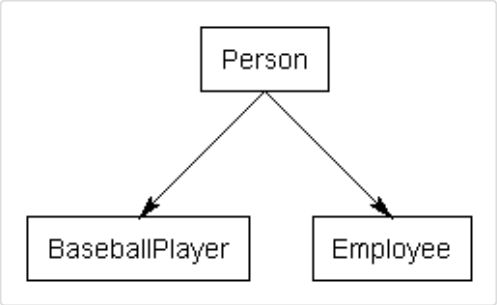
```
    double m_dHourlySalary;
    long m_lEmployeeID;

    Employee(std::string strEmployerName, double dHourlySalary, long lEmployeeID)
        : m_strEmployerName(strEmployerName), m_dHourlySalary(dHourlySalary),
            m_lEmployeeID(lEmployeeID)
    {
    }

    double GetHourlySalary() { return m_dHourlySalary; }
    void PrintNameAndSalary()
    {
        std::cout << m_strName << ": " << m_dHourlySalary << std::endl;
    }
};
```

Employee inherits m_strName, m_nAge, and m_bIsMale from Person (as well as the three access functions), and adds three more member variables and a couple of member function of it's own. Note that PrintNameAndSalary() uses variables both from the class it belongs to (Employee) and the parent class (Person).

This gives us a derivation chart that looks like this:



Note that Employee and BaseballPlayer don't have any direct relationship, even though they both inherit from Person.

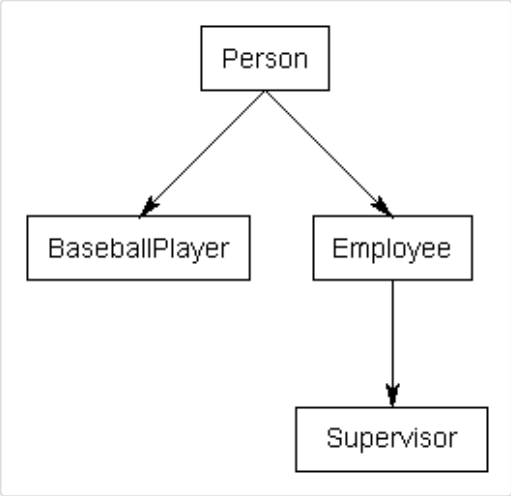
Inheritance chains

It's possible to inherit from a parent that is itself derived from another class. There is nothing noteworthy or special when doing so -- everything proceeds as in the examples above.

For example, let's write a Supervisor class. A supervisor is an employee, which is a person. We've already written an Employee class, so let's use that as the base class from which to derive Supervisor:

```
class Supervisor: public Employee
{
public:
    // This Supervisor can oversee a max of 5 employees
    long m_nOverseesIDs[5];
};
```

Now our derivation chart looks like this:



All Supervisor objects inherit the functions and variables from Employee, and add their own m_nOverseesIDs member variable.

By constructing such inheritance chains, we can create a set of reusable classes that are very general (at the top) and become progressively more specific at each level of inheritance.

Conclusion

Inheriting from a base class means we don't have to redefine the information from the base class in our derived classes. We automatically receive the member functions and member variables of the base class through inheritance, and then simply add the additional functions or member variables we want. This not only saves work, but also means that if we ever update or modify the base class (eg. add new functions, or fix a bug), all of our derived classes will automatically inherit the changes!

For example, if we ever added a new function to Person, both Employee and Supervisor would automatically gain access to it. If we added a new variable to Employee, Supervisor would also gain access to it. This allows us to construct new classes in an easy, intuitive, and low-maintenance way!

11.3 -- Order of construction of derived classes

Index

11.1 -- Introduction to inheritance

Share this:

11.3 — Order of construction of derived classes

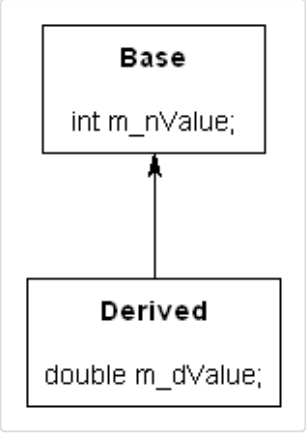
BY ALEX ON JANUARY 7TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 7TH, 2008

In the **previous lesson on basic inheritance in C++**, you learned that classes can inherit members and functions from other classes. In this lesson, we're going to take a closer look at the order of construction that happens when a derived class is instantiated.

First, let's introduce some new classes that will help us illustrate some important points.

```
1  class Base
2  {
3  public:
4      int m_nValue;
5
6      Base(int nValue=0)
7          : m_nValue(nValue)
8      {
9      }
10 };
11
12 class Derived: public Base
13 {
14 public:
15     double m_dValue;
16
17     Derived(double dValue=0.0)
18         : m_dValue(dValue)
19     {
20     }
21 };
```

In this example, Derived is derived from Base. Because Derived inherits functions and variables from Base, it is convenient to think of Derived as a two part class: one part Derived, and one part Base.



You've already seen plenty examples of what happens when we instantiate a normal (non-derived) class:

```
1  int main()
2  {
3      Base cBase;
4
5      return 0;
6  }
```

Base is a non-derived class because it does not inherit from anybody. C++ allocates memory for Base, then calls Base's default constructor to do the initialization.

Now let's take a look at what happens when we instantiate a derived class:

```
int main()
{
    Derived cDerived;

    return 0;
}
```

If you were to try this yourself, you wouldn't notice any difference from the previous example where we instantiate the non-derived class. But behind the scenes, things are slightly different. As mentioned, Derived is really two parts: a Base part, and a Derived part. When C++ constructs derived objects, it does so in pieces, starting with the base portion of the class. Once that is complete, it then walks through the inheritance tree and constructs each derived portion of the class.

So what actually happens in this example is that the Base portion of Derived is constructed first. Once the Base portion is finished, the Derived portion is constructed. At this point, there are no more derived classes, so we are done.

This process is actually easy to illustrate.

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

class Base
{
public:
    int m_nValue;

    Base(int nValue=0)
        : m_nValue(nValue)
    {
        cout << "Base" << endl;
    }
};

class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0)
        : m_dValue(dValue)
    {
        cout << "Derived" << endl;
    }
};

int main()
{
    cout << "Instantiating Base" << endl;
    Base cBase;

    cout << "Instantiating Derived" << endl;
    Derived cDerived;

    return 0;
}
```

This program produces the following result:

```
Instantiating Base
Base
Instantiating Derived
Base
Derived
```

As you can see, when we constructed Derived, the Base portion of Derived got constructed first. This makes sense: logically, a child can not exist without a parent. It's also the safe way to do things: the child class often uses variables and functions from the parent, but the parent class knows nothing about the child. Instantiating the parent class first ensures those variables are already initialized by the time the derived class is created and ready to use them.

Order of construction for inheritance chains

It is sometimes the case that classes are derived from other classes, which are themselves derived from other classes. For example:

```
class A
{
public:
    A()
    {
        cout << "A" << endl;
    }
};

class B: public A
{
public:
    B()
    {
        cout << "B" << endl;
    }
};

class C: public B
{
public:
    C()
    {
        cout << "C" << endl;
    }
};

class D: public C
{
public:
    D()
    {
        cout << "D" << endl;
    }
};
```

Remember that C++ always constructs the "first" or "most base" class first. It then walks through the inheritance tree in order and constructs each successive derived class.

Here's a short program that illustrates the order of creation all along the inheritance chain.

```
int main()
{
    cout << "Constructing A: " << endl;
    A cA;

    cout << "Constructing B: " << endl;
    B cB;

    cout << "Constructing C: " << endl;
    C cC;

    cout << "Constructing D: " << endl;
    D cD;
}
```

This code prints the following:

```
Constructing A:
A
Constructing B:
A
B
Constructing C:
A
B
C
Constructing D:
A
B
C
D
```

Conclusion

You will note that our example classes in this section have all used default constructors. In the next lesson, we will take a closer look at the special role of constructors in the process of constructing derived classes.

11.4 -- Constructors and initialization of derived classes

Index

11.2 -- Basic inheritance in C++

Share this:

11.4 — Constructors and initialization of derived classes

BY ALEX ON JANUARY 9TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 28TH, 2015

In the past two lessons,we've looked at some basics about inheritance in C++ and explored the order that derived classes are initialized. In this lesson, we'll take a closer look at the role of constructors in the initialization of derived classes. To do so, we will continue to use the simple Base and Derived class we developed in the previous lesson:

```
class Base
{
public:
    int m_nValue;

    Base(int nValue=0)
        : m_nValue(nValue)
    {
    }
};

class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0)
        : m_dValue(dValue)
    {
    }
};
```

With non-derived classes, constructors only have to worry about their own members. For example, consider Base. We can create a Base object like this:

```
int main()
{
    Base cBase(5); // use Base(int) constructor

    return 0;
}
```

Here's what actually happens when cBase is instantiated:

1. Memory for cBase is set aside
2. The appropriate Base constructor is called
3. The initialization list initializes variables
4. The body of the constructor executes
5. Control is returned to the caller

This is pretty straightforward. With derived classes, things are slightly more complex:

```
int main()
{
    Derived cDerived(1.3); // use Derived(double) constructor

    return 0;
}
```

Here's what actually happens when cDerived is instantiated:

1. Memory for cDerived is set aside (enough for both the Base and Derived portions).
2. The appropriate Derived constructor is called
3. **The Base object is constructed first using the appropriate Base constructor**

4. The initialization list initializes variables
5. The body of the constructor executes
6. Control is returned to the caller

The only real difference between this case and the non-inherited case is that before the Derived constructor can do anything substantial, the Base constructor is called first. The Base constructor sets up the Base portion of the object, control is returned to the Derived constructor, and the Derived constructor is allowed to finish up it's job.

Initializing base class members

One of the current shortcomings of our Derived class as written is that there is no way to initialize m_nValue when we create a Derived object. What if we want to set both m_dValue (from the Derived portion of the object) and m_nValue (from the Base portion of the object) when we create a Derived object?

New programmers often attempt to solve this problem as follows:

```
class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0, int nValue=0)
        // does not work
        : m_dValue(dValue), m_nValue(nValue)
    {
    }
};
```

This is a good attempt, and is almost the right idea. We definitely need to add another parameter to our constructor, otherwise C++ will have no way of knowing what value we want to initialize m_nValue to.

However, C++ prevents classes from initializing inherited member variables in the initialization list of a constructor. In other words, the value of a variable can only be set in an initialization list of a constructor belonging to the same class as the variable.

Why does C++ do this? The answer has to do with const and reference variables. Consider what would happen if m_nValue were const. Because const variables must be initialized with a value at the time of creation, the base class constructor must set its value when the variable is created. However, when the base class constructor finishes, the derived class constructors initialization lists are then executed. Each derived class would then have the opportunity to initialize that variable, potentially changing it's value! By restricting the initialization of variables to the constructor of the class those variables belong to, C++ ensures that all variables are initialized only once.

The end result is that the above example does not work because m_nValue was inherited from Base, and only non-inherited variables can be changed in the initialization list.

However, inherited variables can still have their values changed in the body of the constructor using an assignment. Consequently, new programmers often also try this:

```
class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0, int nValue=0)
        : m_dValue(dValue)
    {
        m_nValue = nValue;
    }
};
```

While this actually works in this case, it wouldn't work if m_nValue were a const or a reference (because const values and references have to be initialized in the initialization list of the constructor). It's also inefficient because m_nValue gets assigned a value twice: once in the initialization list of the Base class constructor, and then again in the body of the Derived

class constructor.

So how do we properly initialize m_nValue when creating a Derived class object?

In all of the examples so far, when we instantiate a Derived class object, the Base class portion has been created using the default Base constructor. Why does it always use the default Base constructor? Because we never told it to do otherwise!

Fortunately, C++ gives us the ability to explicitly choose which Base class constructor will be called! To do this, simply add a call to the base class Constructor in the initialization list of the derived class:

```
class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0, int nValue=0)
        : Base(nValue), // Call Base(int) constructor with value nValue!
            m_dValue(dValue)
    {
    }
};
```

Now, when we execute this code:

```
int main()
{
    Derived cDerived(1.3, 5); // use Derived(double) constructor

    return 0;
}
```

The base class constructor Base(int) will be used to initialize m_nValue to 5, and the derived class constructor will be used to initialize m_dValue to 1.3!

In more detail, here's what happens:

1. Memory for cDerived is allocated.
2. The Derived(double, int) constructor is called, where dValue = 1.3, and nValue = 5
3. The compiler looks to see if we've asked for a particular Base class constructor. We have! So it calls Base(int) with nValue = 5.
4. The base class constructor initialization list sets m_nValue to 5
5. The base class constructor body executes
6. The base class constructor returns
7. The derived class constuctor initialization list sets m_dValue to 1.3
8. The derived class constructor body executes
9. The derived class constructor returns

This may seem somewhat complex, but it's actually very simple. All that's happening is that the Derived constructor is calling a specific Base constructor to initialize the Base portion of the object. Because m_nValue lives in the Base portion of the object, the Base constructor is the only constructor that can initialize it's value.

Another example

Let's take a look at another pair of class we've previously worked with:

```
#include <string>
class Person
{
public:
    std::string m_strName;
    int m_nAge;
    bool m_bIsMale;

    std::string GetName() { return m_strName; }
```

```
    int GetAge() { return m_nAge; }
    bool IsMale() { return m_bIsMale; }

    Person(std::string strName = "", int nAge = 0, bool bIsMale = false)
        : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
    {
    }
};

// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;

    BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
       : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
    {
    }
};
```

As we'd previously written it, BaseballPlayer only initializes its own members and does not specify a Person constructor to use. This means every BaseballPlayer we create is going to use the default Person constructor, which will initialize the name to blank and age to 0. Because it makes sense to give our BaseballPlayer a name and age when we create them, we should modify this constructor to add those parameters.

Here's our new BaseballPlayer class with a constructor that calls the Person constructor to initialize the inherited Person member variables.

```
// BaseballPlayer publicly inheriting Person
class BaseballPlayer : public Person
{
public:
    double m_dBattingAverage;
    int m_nHomeRuns;

    BaseballPlayer(std::string strName = "", int nAge = 0, bool bIsMale = false,
        double dBattingAverage = 0.0, int nHomeRuns = 0)
        : Person(strName, nAge, bIsMale), // call Person(std::string, int, bool) to initialize
these fields
            m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
    {
    }
};
```

Now we can create baseball players like this:

```
int main()
{
    BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.342, 42);

    return 0;
}
```

To prove that it works:

```
int main()
{
    BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.342, 42);

    using namespace std;
    cout << cPlayer.m_strName << endl;
    cout << cPlayer.m_nAge << endl;
    cout << cPlayer.m_nHomeRuns;

```

```
    return 0;
}
```

This outputs:

```
Pedro Cerrano
32
42
```

As you can see, the name and age in the base class were properly initialized, as was the number of home runs in the derived class.

Inheritance chains

Classes in an inheritance chain work in exactly the same way.

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

class A
{
public:
    A(int nValue)
    {
        cout << "A: " << nValue << endl;
    }
};

class B: public A
{
public:
    B(int nValue, double dValue)
    : A(nValue)
    {
        cout << "B: " << dValue << endl;
    }
};

class C: public B
{
public:
    C(int nValue, double dValue, char chValue)
    : B(nValue, dValue)
    {
        cout << "C: " << chValue << endl;
    }
};

int main()
{
    C cClass(5, 4.3, 'R');

    return 0;
}
```

In this example, class C is derived from class B, which is derived from class A. So what happens when we instantiate an object of class C?

First, main() calls C(int, double, char). The C constructor calls B(int, double). The B constructor calls A(int). Because A is not inherited, this is the first class we'll construct. A is constructed, prints the value 5, and returns control to B. B is constructed, prints the value 4.3, and returns control to C. C is constructed, prints the value 'R', and returns control to main(). And we're done!

Thus, this program prints:

```
A: 5
B: 4.3
C: R
```

It is worth mentioning that constructors can only call constructors from their immediate parent/base class. Consequently, the C constructor could not call or pass parameters to the A constructor directly. The C constructor can only call the B constructor (which has the responsibility of calling the A constructor).

Destructors

When a derived class is destroyed, each destructor is called in the reverse order of construction. In the above example, when cClass is destroyed, the C destructor is called first, then the B destructor, then the A destructor.

Summary

Although it is true that the most base class is initialized first, this actually only happens after each constructor has called the parent constructor in turn. This gives us the opportunity to specify which of the parent's constructors we want to use to initialize inherited members. Once the base constructor has finished constructing the base portion of the class, control returns to the derived constructor and it executes as normal.

One of the primary advantages of using a base class constructor to initialize the base class members is that if the base class constructor is ever changed, both the base class and all inherited classes will automatically use the changes! This helps keep maintenance and duplicate code down.

 **11.5 -- Inheritance and access specifiers**

 **Index**

 **11.3 -- Order of construction of derived classes**

Share this:

11.5 — Inheritance and access specifiers

BY ALEX ON JANUARY 14TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 28TH, 2015

In the previous lessons on inheritance, we've been making all of our data members public in order to simplify the examples. In this section, we'll talk about the role of access specifiers in the inheritance process, as well as cover the different types of inheritance possible in C++.

To this point, you've seen the private and public access specifiers, which determine who can access the members of a class. As a quick refresher, public members can be accessed by anybody. Private members can only be accessed by member functions of the same class. Note that this means derived classes can not access private members!

```
class Base
{
private:
    int m_nPrivate; // can only be accessed by Base member functions (not derived classes)
public:
    int m_nPublic; // can be accessed by anybody
};
```

When dealing with inherited classes, things get a bit more complex.

First, there is a third access specifier that we have yet to talk about because it's only useful in an inheritance context. The **protected** access specifier restricts access to member functions of the same class, or those of derived classes.

```
class Base
{
public:
    int m_nPublic; // can be accessed by anybody
private:
    int m_nPrivate; // can only be accessed by Base member functions (but not derived classes)
protected:
    int m_nProtected; // can be accessed by Base member functions, or derived classes.
};

class Derived: public Base
{
public:
    Derived()
    {
        // Derived's access to Base members is not influenced by the type of inheritance used,
        // so the following is always true:

        m_nPublic = 1; // allowed: can access public base members from derived class
        m_nPrivate = 2; // not allowed: can not access private base members from derived class
        m_nProtected = 3; // allowed: can access protected base members from derived class
    }
};

int main()
{
    Base cBase;
    cBase.m_nPublic = 1; // allowed: can access public members from outside class
    cBase.m_nPrivate = 2; // not allowed: can not access private members from outside class
    cBase.m_nProtected = 3; // not allowed: can not access protected members from outside clas
s
}
```

Second, when a derived class inherits from a base class, the access specifiers may change depending on the method of inheritance. There are three different ways for classes to inherit from other classes: public, private, and protected.

To do so, simply specify which type of access you want when choosing the class to inherit from:

```
// Inherit from Base publicly
class Pub: public Base
{
};

// Inherit from Base privately
class Pri: private Base
{
};

// Inherit from Base protectedly
class Pro: protected Base
{
};

class Def: Base // Defaults to private inheritance
{
};
```

If you do not choose an inheritance type, C++ defaults to private inheritance (just like members default to private access if you do not specify otherwise).

That gives us 9 combinations: 3 member access specifiers (public, private, and protected), and 3 inheritance types (public, private, and protected).

The rest of this section will be devoted to explaining the difference between these.

Before we get started, the following should be kept in mind as we step through the examples. There are three ways that members can be accessed:

- A class can always access its own members regardless of access specifier.
- The public accesses the members of a class based on the access specifiers of that class.
- A derived class accesses inherited members based on the access specifiers of its immediate parent. A derived class can always access its own members regardless of access specifier.

This may be a little confusing at first, but hopefully will become clearer as we step through the examples.

Public inheritance

Public inheritance is by far the most commonly used type of inheritance. In fact, very rarely will you use the other types of inheritance, so your primary focus should be on understanding this section. Fortunately, public inheritance is also the easiest to understand. When you inherit a base class publicly, all members keep their original access specifications. Private members stay private, protected members stay protected, and public members stay public.

```
class Base
{
public:
    int m_nPublic;
private:
    int m_nPrivate;
protected:
    int m_nProtected;
};

class Pub: public Base
{
    // Public inheritance means:
    // m_nPublic stays public
    // m_nPrivate stays private
    // m_nProtected stays protected

    Pub()
    {
        // The derived class always uses the immediate parent's class access specifications
        // Thus, Pub uses Base's access specifiers
```

```
        m_nPublic = 1; // okay: anybody can access public members
        m_nPrivate = 2; // not okay: derived classes can't access private members in the base class!
        m_nProtected = 3; // okay: derived classes can access protected members
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    // In this case, the access specifiers of cPub.  Because Pub has inherited publicly from Base,
    // no access specifiers have been changed.
    Pub cPub;
    cPub.m_nPublic = 1; // okay: anybody can access public members
    cPub.m_nPrivate = 2; // not okay: can not access private members from outside class
    cPub.m_nProtected = 3; // not okay: can not access protected members from outside class
}
```

This is fairly straightforward. The things worth noting are:

1. Derived classes can not directly access private members of the base class.
2. The protected access specifier allows derived classes to directly access members of the base class while not exposing those members to the public.
3. The derived class uses access specifiers from the base class.
4. The outside uses access specifiers from the derived class.

To summarize in table form:

	Public inheritance		
Base access specifier	Derived access specifier	Derived class access?	Public access?
Public	Public	Yes	Yes
Private	Private	No	No
Protected	Protected	Yes	No

Private inheritance

With private inheritance, all members from the base class are inherited as private. This means private members stay private, and protected and public members become private.

Note that this does not affect that way that the derived class accesses members inherited from its parent! It only affects the code trying to access those members through the derived class.

```
class Base
{
public:
    int m_nPublic;
private:
    int m_nPrivate;
protected:
    int m_nProtected;
};

class Pri: private Base
{
    // Private inheritance means:
    // m_nPublic becomes private
    // m_nPrivate stays private
    // m_nProtected becomes private

    Pri()
    {
```

```
        // The derived class always uses the immediate parent's class access specifications
        // Thus, Pub uses Base's access specifiers
        m_nPublic = 1; // okay: anybody can access public members
        m_nPrivate = 2; // not okay: derived classes can't access private members in the base class!
        m_nProtected = 3; // okay: derived classes can access protected members
    }
};

int main()
{
    // Outside access uses the access specifiers of the class being accessed.
    // Note that because Pri has inherited privately from Base,
    // all members of Base have become private when access through Pri.
    Pri cPri;
    cPri.m_nPublic = 1; // not okay: m_nPublic is now a private member when accessed through Pri
    cPri.m_nPrivate = 2; // not okay: can not access private members from outside class
    cPri.m_nProtected = 3; // not okay: m_nProtected is now a private member when accessed through Pri

    // However, we can still access Base members as normal through Base:
    Base cBase;
    cBase.m_nPublic = 1; // okay, m_nPublic is public
    cBase.m_nPrivate = 2; // not okay, m_nPrivate is private
    cBase.m_nProtected = 3; // not okay, m_nProtected is protected
}
```

To summarize in table form:

Private inheritance

Base access specifier	Derived access specifier	Derived class access?	Public access?
Public	Private	Yes	No
Private	Private	No	No
Protected	Private	Yes	No

Protected inheritance

Protected inheritance is the last method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and private members stay private.

To summarize in table form:

Protected inheritance

Base access specifier	Derived access specifier	Derived class access?	Public access?
Public	Protected	Yes	No
Private	Private	No	No
Protected	Protected	Yes	No

Protected inheritance is similar to private inheritance. However, classes derived from the derived class still have access to the public and protected members directly. The public (stuff outside the class) does not.

Summary

The way that the access specifiers, inheritance types, and derived classes interact causes a lot of confusion. To try and clarify things as much as possible:

First, the base class sets its access specifiers. The base class can always access its own members. The access specifiers

only affect whether outsiders and derived classes can access those members.

Second, derived classes have access to base class members based on the access specifiers of the immediate parent. The way a derived class accesses inherited members is not affected by the inheritance method used!

Finally, derived classes can change the access type of inherited members based on the inheritance method used. This does not affect the derived classes' members, which have their own access specifiers. It only affects whether outsiders and classes derived from the derived class can access those inherited members.

A final example:

```
class Base
{
public:
    int m_nPublic;
private:
    int m_nPrivate;
protected:
    int m_nProtected;
};
```

Base can access its own members without restriction. The public can only access m_nPublic. Derived classes can access m_nPublic and m_nProtected.

```
class D2: private Base
{
public:
    int m_nPublic2;
private:
    int m_nPrivate2;
protected:
    int m_nProtected2;
}
```

D2 can access its own members without restriction. D2 can access Base's members based on Base's access specifiers. Thus, it can access m_nPublic and m_nProtected, but not m_nPrivate. Because D2 inherited Base privately, m_nPublic, m_nPrivate, and m_nProtected are now private when accessed through D2. This means the public can not access any of these variables when using a D2 object, nor can any classes derived from D2.

```
class D3: public D2
{
public:
    int m_nPublic3;
private:
    int m_nPrivate3;
protected:
    int m_nProtected3;
};
```

D3 can access its own members without restriction. D3 can access D2's members based on D2's access specifiers. Thus, D3 has access to m_nPublic2 and m_nProtected2, but not m_nPrivate2. D3's access to Base members is controlled by the access specifier of its immediate parent. This means D3 does not have access to any of Base's members because they all became private when D2 inherited them.

11.6 -- Adding, changing, and hiding members in a derived class

Index

11.6 — Adding, changing, and hiding members in a derived class

BY ALEX ON JANUARY 17TH, 2008 | LAST MODIFIED BY ALEX ON FEBRUARY 15TH, 2016

In the **introduction to inheritance** lesson, we mentioned that one of the biggest benefits of using derived classes is the ability to reuse already written code. You can inherit the base class functionality and then add new functionality, modify existing functionality, or hide functionality you don't want. In this lesson, we'll take a closer look at how this is done.

First, let's start with a simple base class:

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

class Base
{
protected:
    int m_nValue;

public:
    Base(int nValue)
        : m_nValue(nValue)
    {
    }

    void Identify() { cout << "I am a Base" << endl; }
};
```

Now, let's create a derived class that inherits from Base. Because we want Derived to be able to set the value of m_nValue when Derived objects are instantiated, we'll make the Derived constructor call the Base constructor in the initialization list.

```
class Derived: public Base
{
public:
    Derived(int nValue)
        :Base(nValue)
    {
    }
};
```

We'll develop Derived over the course of this lesson.

Adding new functionality

Because we have access to the source code of the Base class, we could add functionality directly to Base. However, there may be times when we do not want to, or can not. Consider the case where you have just purchased a library of code from a 3rd party vendor, but need some extra functionality. You could add to the original code, but this isn't the best solution. What if the vendor sends you an update? Either your additions will be overwritten, or you'll have to manually migrate them. It's also common for developers to release header files containing class definitions, but release the implementation code precompiled -- this means you can use the code, but you won't have the ability to modify it directly.

In either case, the best answer is to derive your own class, and add the functionality you want to the derived class.

One obvious omission from the Base class is a way for the public to access m_nValue. Normally we'd write an access function in the Base class -- but for the sake of example we're going to add it to the derived class instead. Because m_nValue has been declared as protected in the Base class, Derived has direct access to it.

To add new functionality to a derived class, simply declare that functionality in the derived class like normal:

```
class Derived: public Base
{
public:
```

```
    Derived(int nValue)
        :Base(nValue)
    {
    }

    int GetValue() { return m_nValue; }
};
```

Now the public will be able to call GetValue() in order to access the value of m_nValue.

```
int main()
{
    Derived cDerived(5);
    cout << "cDerived has value " << cDerived.GetValue() << endl;

    return 0;
}
```

This produces the result:

```
cDerived has value 5
```

Although it may be obvious, objects of type Base have no access to the GetValue() function in Derived. The following does not work:

```
int main()
{
    Base cBase(5);
    cout << "cBase has value " << cBase.GetValue() << endl;

    return 0;
}
```

This is because there is no GetValue() function in Base. GetValue() belongs to Derived. Because Derived is a Base, Derived has access to stuff in Base. However, Base does not have access to anything in Derived.

Redefining functionality

When a member function is called with a derived class object, the compiler first looks to see if that member exists in the derived class. If not, it begins walking up the inheritance chain and checking whether the member has been defined in any of the inherited classes. It uses the first one it finds.

Consequently, take a look at the following example:

```
int main()
{
    Base cBase(5);
    cBase.Identify();

    Derived cDerived(7);
    cDerived.Identify()

    return 0;
}
```

This prints

```
I am a Base
I am a Base
```

When cDerived.Identify() is called, the compiler looks to see if Identify() has been defined in the Derived class. It hasn't. Then it starts looking in the inherited classes (which in this case is Base). Base has defined a Identify() function, so it uses

that one. In other words, Base::Identify() was used because Derived::Identify() doesn't exist.

However, if we had defined Derived::Identify() in the Derived class, it would have been used instead. This means that we can make functions work differently with our derived classes by redefining them in the derived class!

In our above example, it would be more accurate if `cDerived.Identify()` printed "I am a Derived". Let's modify Identify() so it returns the correct response when we call Identify() with a Derived object.

To modify a function the way a function defined in a base class works in the derived class, simply redefine the function in the derived class.

```
class Derived: public Base
{
public:
    Derived(int nValue)
        :Base(nValue)
    {
    }

    int GetValue() { return m_nValue; }

    // Here's our modified function
    void Identify() { cout << "I am a Derived" << endl; }
};
```

Here's the same example as above, using the new Derived::Identify() function:

```
int main()
{
    Base cBase(5);
    cBase.Identify();

    Derived cDerived(7);
    cDerived.Identify()

    return 0;
}
```

```
I am a Base
I am a Derived
```

Note that when you redefine a function in the derived class, the derived function does not inherit the access specifier of the function with the same name in the base class. It uses whatever access specifier it is defined under in the derived class. Therefore, a function that is defined as private in the base class can be redefined as public in the derived class, or vice-versa!

Adding to existing functionality

Sometimes we don't want to completely replace a base class function, but instead want to add additional functionality to it. In the above example, note that Derived::Identify() completely hides Base::Identify()! This may not be what we want. It is possible to have our Derived function call the Base function of the same name (in order to reuse code) and then add additional functionality to it.

To have a derived function call a base function of the same name, simply do a normal function call, but prefix the function with the scope qualifier (the name of the base class and two colons). The following example redefines Derived::Identify() so it first calls Base::Identify() and then does it's own additional stuff.

```
class Derived: public Base
{
public:
    Derived(int nValue)
        :Base(nValue)
    {
```

```
    }

    int GetValue() { return m_nValue; }

    void Identify()
    {
        Base::Identify(); // call Base::Identify() first
        cout << "I am a Derived"; // then identify ourselves
    }
};
```

Now consider the following example:

```
int main()
{
    Base cBase(5);
    cBase.Identify();

    Derived cDerived(7);
    cDerived.Identify()

    return 0;
}
```

```
I am a Base
I am a Base
I am a Derived
```

When `cDerived.Identify()` is executed, it resolves to Derived::Identify(). However, the first thing Derived::Identify() does is call Base::Identify(), which prints “I am a Base”. When Base::Identify() returns, Derived::Identify() continues executing and prints “I am a Derived”.

This is all pretty straightforward. The real lesson to take away from this is that if you want to call a function in a base class that has been redefined in the derived class, you need to use the scope resolution operator (::) to explicitly say which version of the function you want.

If we had defined Derived::Identify() like this:

```
    void Identify()
    {
        Identify(); // Note: no scope resolution!
        cout << "I am a Derived"; // then identify ourselves
    }
```

Identify() without a scope resolution qualifier would default to the Identify() in the current class, which would be Derived::Identify(). This would cause Derived::Identify() to call itself, which would lead to an infinite loop!

Hiding functionality

In C++, it is not possible to remove functionality from a class. However, it is possible to hide existing functionality.

As mentioned above, if you redefine a function, it uses whatever access specifier it’s declared under in the derived class. Therefore, we could redefine a public function as private in our derived class, and the public would lose access to it. However, C++ also gives us the ability to change a base member’s access specifier in the derived class without even redefining the member! This is done by simply naming the member (using the scope resolution operator) to have it’s access changed in the derived class under the new access specifier.

For example, consider the following Base:

```
class Base
{
private:
    int m_nValue;
```

```
5
6   public:
7       Base(int nValue)
8           : m_nValue(nValue)
9       {
10      }
11
12  protected:
13      void PrintValue() { cout << m_nValue; }
14  };
```

Because Base::PrintValue() has been declared as protected, it can only be called by Base or it's derived classes. The public can not access it.

Let's define a Derived class that changes the access specifier of PrintValue() to public:

```
1   class Derived: public Base
2   {
3   public:
4       Derived(int nValue)
5           : Base(nValue)
6       {
7       }
8
9       // Base::PrintValue was inherited as protected, so the public has no access
10      // But we're changing it to public by declaring it in the public section
11      Base::PrintValue;
12  };
```

This means that this code will now work:

```
1   int main()
2   {
3       Derived cDerived(7);
4
5       // PrintValue is public in Derived, so this is okay
6       cDerived.PrintValue(); // prints 7
7       return 0;
8   }
```

Note that Base::PrintValue does not have the function call operator (()) attached to it.

We can also use this to make public members private:

```
1   class Base
2   {
3   public:
4       int m_nValue;
5   };
6
7   class Derived: public Base
8   {
9   private:
10      Base::m_nValue;
11
12  public:
13      Derived(int nValue)
14      {
15          m_nValue = nValue;
16      }
17  };
18
19  int main()
20  {
21      Derived cDerived(7);
22
23      // The following won't work because m_nValue has been redefined as private
```

```
24      cout << cDerived.m_nValue;
25
26      return 0;
27  }
```

Note that this allowed us to take a poorly designed base class and encapsulate it's data in our derived class. (Alternatively, instead of inheriting Base's members publicly and making m_nValue private by overriding it's access specifier, we could have inherited Base privately, which would have caused all of Base's member to be inherited privately in the first place).

One word of caution: you can only change the access specifiers of base members the class would normally be able to access. Therefore, you can never change the access specifier of a base member from private to protected or public, because derived classes do not have access to private members of the base class.

11.7 -- Multiple inheritance

Index

11.5 -- Inheritance and access specifiers

Share this:

C++ TUTORIAL | PRINT THIS POST

39 comments to 11.6 — Adding, changing, and hiding members in a derived class

Zafer
January 31, 2008 at 3:03 pm · Reply

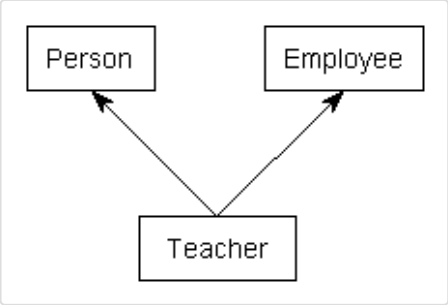
I think the following statement about the last example is confusing: "Note that this allowed us to take a poorly designed base class and encapsulate it's data in our derived class (alternatively, we could have

11.7 — Multiple inheritance

BY ALEX ON JANUARY 24TH, 2008 | LAST MODIFIED BY ALEX ON OCTOBER 21ST, 2015

So far, all of the examples of inheritance we've presented have been single inheritance -- that is, each inherited class has one and only one parent. However, C++ provides the ability to do multiple inheritance. **Multiple inheritance** enables a derived class to inherit members from more than one parent.

Let's say we wanted to write a program to keep track of a bunch of teachers. A teacher is a person. However, a teacher is also an employee (they are their own employer if working for themselves). Multiple inheritance can be used to create a Teacher class that inherits properties from both Person and Employee. To use multiple inheritance, simply specify each base class (just like in single inheritance), separated by a comma.




```
#include <string>
class Person
{
private:
    std::string m_strName;
    int m_nAge;
    bool m_bIsMale;

public:
    Person(std::string strName, int nAge, bool bIsMale)
        : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
    {
    }

    std::string GetName() { return m_strName; }
    int GetAge() { return m_nAge; }
    bool IsMale() { return m_bIsMale; }
};

class Employee
{
private:
    std::string m_strEmployer;
    double m_dWage;

public:
    Employee(std::string strEmployer, double dWage)
        : m_strEmployer(strEmployer), m_dWage(dWage)
    {
    }

    std::string GetEmployer() { return m_strEmployer; }
    double GetWage() { return m_dWage; }
};

// Teacher publicly inherits Person and Employee
class Teacher: public Person, public Employee
{
```

```
private:
     int m_nTeachesGrade;

public:
    Teacher(std::string strName, int nAge, bool bIsMale, std::string strEmployer, double dWag
e, int nTeachesGrade)
        : Person(strName, nAge, bIsMale), Employee(strEmployer, dWage), m_nTeachesGrade(nTeach
esGrade)
    {
    }
};
```

Problems with multiple inheritance

While multiple inheritance seems like a simple extension of single inheritance, multiple inheritance introduces a lot of issues that can markedly increase the complexity of programs and make them a maintenance nightmare. Let's take a look at some of these situations.

First, ambiguity can result when multiple base classes contain a function with the same name. For example:

```
class USBDevice
{
private:
    long m_lID;

public:
    USBDevice(long lID)
        : m_lID(lID)
    {
    }

    long GetID() { return m_lID; }
};

class NetworkDevice
{
private:
    long m_lID;

public:
    NetworkDevice(long lID)
        : m_lID(lID)
    {
    }

    long GetID() { return m_lID; }
};

class WirelessAdaptor: public USBDevice, public NetworkDevice
{
public:
    WirelessAdaptor(long lUSBID, long lNetworkID)
        : USBDevice(lUSBID), NetworkDevice(lNetworkID)
    {
    }
};

int main()
{
    WirelessAdaptor c54G(5442, 181742);
    cout << c54G.GetID(); // Which GetID() do we call?

    return 0;
}
```

When `c54G.GetID()` is evaluated, the compiler looks to see if WirelessAdaptor contains a function named GetID(). It

doesn't. The compiler then looks to see if any of the base classes have a function named GetID(). See the problem here? The problem is that c54G actually contains TWO GetID() functions: one inherited from USBDevice, and one inherited from NetworkDevice. Consequently, this function call is ambiguous, and you will receive a compiler error if you try to compile it.

However, there is a way to work around this problem: you can explicitly specify which version you meant to call:

```
int main()
{
    WirelessAdaptor c54G(5442, 181742);
    cout << c54G.USBDevice::GetID();

    return 0;
}
```

While this workaround is pretty simple, you can see how things can get complex when your class inherits from four or six base classes, which inherit from other classes themselves. The potential for naming conflicts increases exponentially as you inherit more classes, and each of these naming conflicts needs to be resolved explicitly.

Second, and more serious is the **diamond problem**, which your author likes to call the "diamond of doom". This occurs when a class multiply inherits from two classes which each inherit from a single base class. This leads to a diamond shaped inheritance pattern.

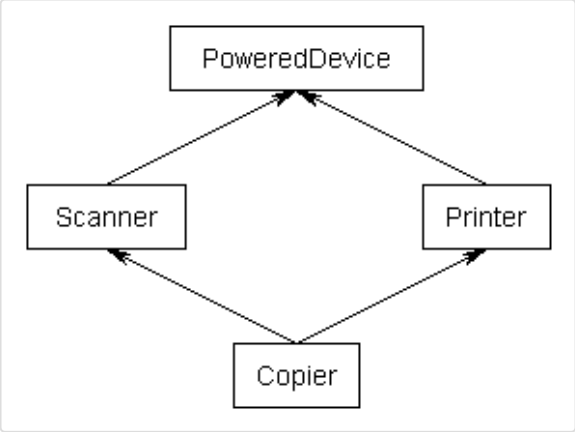
For example, consider the following set of classes:

```
class PoweredDevice
{
};

class Scanner: public PoweredDevice
{
};

class Printer: public PoweredDevice
{
};

class Copier: public Scanner, public Printer
{
};
```



Scanners and printers are both powered devices, so they derived from PoweredDevice. However, a copy machine incorporates the functionality of both Scanners and Printers.

There are many issues that arise in this context, including whether Copier should have one or two copies of PoweredDevice, and how to resolve certain types of ambiguous references. While most of these issues can be addressed through explicit scoping, the maintenance overhead added to your classes in order to deal with the added complexity can cause development time to skyrocket.

Is multiple inheritance more trouble than it's worth?

As it turns out, most of the problems that can be solved using multiple inheritance can be solved using single inheritance as well. Many object-oriented languages (eg. Smalltalk, PHP) do not even support multiple inheritance. Many relatively modern languages such as Java and C# restricts classes to single inheritance of normal classes, but allow multiple inheritance of interface classes (which we will talk about later). The driving idea behind disallowing multiple inheritance in these languages is that it simply makes the language too complex, and ultimately causes more problems than it fixes.

Many authors and experienced programmers believe multiple inheritance in C++ should be avoided at all costs due to the many potential problems it brings. Your author does not agree with this approach, because there are times and situations when multiple inheritance is the best way to proceed. However, multiple inheritance should be used extremely judiciously.

As an interesting aside, you have already been using classes written using multiple inherited without knowing it: the iostream library objects cin and cout are both implemented using multiple inheritance.

 **11.8 -- Virtual base classes**

 **Index**

 **11.6 -- Adding, changing, and hiding members in a derived class**

Share this:

C++ TUTORIAL | PRINT THIS POST

12 comments to 11.7 — Multiple inheritance

som shekhar
November 6, 2008 at 2:06 am · Reply

The compiler then looks to see if any of the derived classes have a function named GetID().
In the description of the class USBDevice and network adapter example::

11.8 — Virtual base classes

BY ALEX ON JANUARY 28TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 19TH, 2016

Note: This section is an advanced topic and can be skipped or skimmed if desired.

In the previous section on multiple inheritance, we left off talking about the “diamond problem”. In this section, we will resume this discussion.

Virtual base classes

Here is our example from the previous lesson, with some constructors:

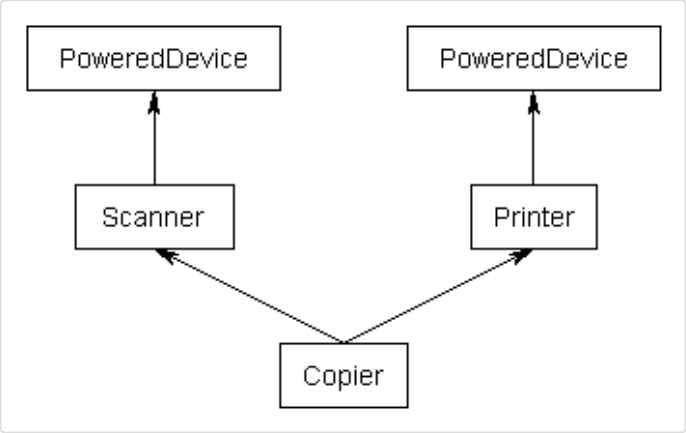
```
class PoweredDevice
{
public:
    PoweredDevice(int nPower)
    {
        cout << "PoweredDevice: " << nPower << endl;
    }
};

class Scanner: public PoweredDevice
{
public:
    Scanner(int nScanner, int nPower)
        : PoweredDevice(nPower)
    {
        cout << "Scanner: " << nScanner << endl;
    }
};

class Printer: public PoweredDevice
{
public:
    Printer(int nPrinter, int nPower)
        : PoweredDevice(nPower)
    {
        cout << "Printer: " << nPrinter << endl;
    }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int nScanner, int nPrinter, int nPower)
        : Scanner(nScanner, nPower), Printer(nPrinter, nPower)
    {
    }
};
```

If you were to create a Copier class object, by default you would end up with two copies of the PoweredDevice class -- one from Printer, and one from Scanner. This has the following structure:



We can create a short example that will show this in action:

```
int main()
{
    Copier cCopier(1, 2, 3);
}
```

This produces the result:

```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
```

As you can see, PoweredDevice got constructed twice.

While this is sometimes what you want, other times you may want only one copy of PoweredDevice to be shared by both Scanner and Printer. To share a base class, simply insert the “virtual” keyword in the inheritance list of the derived class. This creates what is called a **virtual base class**, which means there is only one base object that is shared. Here is the an example (without constructors for simplicity) showing how to use to virtual keyword to create a shared base class:

```
class PoweredDevice
{
};

class Scanner: virtual public PoweredDevice
{
};

class Printer: virtual public PoweredDevice
{
};

class Copier: public Scanner, public Printer
{
};
```

Now, when you create a Copier class, you will get only one copy of PoweredDevice that will be shared by both Scanner and Printer.

However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsible for creating it? The answer, as it turns out, is Copier. The Copier constructor is responsible for creating PoweredDevice. Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly:

```
class PoweredDevice
{
public:
    PoweredDevice(int nPower)
```

```
    {
        cout << "PoweredDevice: " << nPower << endl;
    }
};

class Scanner: virtual public PoweredDevice
{
public:
    Scanner(int nScanner, int nPower)
        : PoweredDevice(nPower)
    {
        cout << "Scanner: " << nScanner << endl;
    }
};

class Printer: virtual public PoweredDevice
{
public:
    Printer(int nPrinter, int nPower)
        : PoweredDevice(nPower)
    {
        cout << "Printer: " << nPrinter << endl;
    }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int nScanner, int nPrinter, int nPower)
        : Scanner(nScanner, nPower), Printer(nPrinter, nPower), PoweredDevice(nPower)
    {
    }
};
```

This time, our previous example:

```
int main()
{
    Copier cCopier(1, 2, 3);
}
```

produces the result:

```
PoweredDevice: 3
Scanner: 1
Printer: 2
```

As you can see, PoweredDevice only gets constructed once.

There are a few details that we would be remiss if we did not mention.

First, virtual base classes are created before non-virtual base classes, which ensures all bases get created before their derived classes.

Second, note that the Scanner and Printer constructors still have calls to the PoweredDevice constructor. If we are creating an instance of Copier, these constructor calls are simply ignored because Copier is responsible for creating the PoweredDevice, not Scanner or Printer. However, if we were to create an instance of Scanner or Printer, the virtual keyword is ignored, those constructor calls would be used, and normal inheritance rules apply.

Third, if a class inherits one or more classes that have virtual parents, the most derived class is responsible for constructing the virtual base class. In this case, Copier inherits Printer and Scanner, both of which have a PoweredDevice virtual base class. Copier, the most derived class, is responsible for creation of PoweredDevice. Note that this is true even in a single inheritance case: if Copier was singly inherited from Printer, and Printer was virtually inherited from PoweredDevice, Copier

is still responsible for creating PoweredDevice.

12.1 -- Pointers and references to the base class of derived objects

Index

11.7 -- Multiple inheritance

Share this:

C++ TUTORIAL | PRINT THIS POST

30 comments to 11.8 — Virtual base classes

prABU
June 11, 2008 at 9:18 pm · Reply

i was not satisfied with ur explian, i need more definitions for vitual base classes

bookworm
August 10, 2009 at 3:27 pm · Reply

You should not read advanced topics before you fully understand fundamental stuff.

sandhya
July 8, 2008 at 2:05 am · Reply

Hi Alex,

12.3 — Virtual destructors, virtual assignment, and overriding virtualization

BY ALEX ON FEBRUARY 1ST, 2008 | LAST MODIFIED BY ALEX ON MARCH 6TH, 2016

Virtual destructors

Although C++ provides a default destructor for your classes if you do not provide one yourself, it is sometimes the case that you will want to provide your own destructor (particularly if the class needs to deallocate memory). You should **always** make your destructors virtual if you're dealing with inheritance. Consider the following example:

```
class Base
{
public:
    ~Base()
    {
        cout << "Calling ~Base()" << endl;
    }
};

class Derived: public Base
{
private:
    int* m_pnArray;

public:
    Derived(int nLength)
    {
        m_pnArray = new int[nLength];
    }

    ~Derived() // note: not virtual
    {
        cout << "Calling ~Derived()" << endl;
        delete[] m_pnArray;
    }
};

int main()
{
    Derived *pDerived = new Derived(5);
    Base *pBase = pDerived;
    delete pBase;

    return 0;
}
```

Because pBase is a Base pointer, when pBase is deleted, the program looks to see if the Base destructor is virtual. It's not, so it assumes it only needs to call the Base destructor. We can see this in the fact that the above example prints:

```
Calling ~Base()
```

However, we really want the delete function to call Derived's destructor (which will call Base's destructor in turn). We do this by making Base's destructor virtual:

```
class Base
{
public:
    virtual ~Base()
    {
```

```
6           cout << "Calling ~Base()" << endl;
7       }
8   };
9
10  class Derived: public Base
11  {
12  private:
13      int* m_pnArray;
14
15  public:
16      Derived(int nLength)
17      {
18          m_pnArray = new int[nLength];
19      }
20
21      virtual ~Derived()
22      {
23          cout << "Calling ~Derived()" << endl;
24          delete[] m_pnArray;
25      }
26  };
27
28  int main()
29  {
30      Derived *pDerived = new Derived(5);
31      Base *pBase = pDerived;
32      delete pBase;
33
34      return 0;
35  }
```

Now this program produces the following result:

```
Calling ~Derived()
Calling ~Base()
```

Again, whenever you are dealing with inheritance, you should make your destructors virtual.

Virtual assignment

It is possible to make the assignment operator virtual. However, unlike the destructor case where virtualization is always a good idea, virtualizing the assignment operator really opens up a bag full of worms and gets into some advanced topics outside of the scope of this tutorial. Consequently, we are going to recommend you leave your assignments non-virtual for now, in the interest of simplicity.

Overriding virtualization

Very rarely you may want to override the virtualization of a function. For example, consider the following code:

```
1   class Base
2   {
3   public:
4       virtual const char* GetName() { return "Base"; }
5   };
6
7   class Derived: public Base
8   {
9   public:
10      virtual const char* GetName() { return "Derived"; }
11  };
```

There may be cases where you want a Base pointer to a Derived object to call Base::GetName() instead of Derived::GetName(). To do so, simply use the scope resolution operator:

```
int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    // Calls Base::GetName() instead of the virtualized Derived::GetName()
    cout << rBase.Base::GetName() << endl;
}
```

You probably won't use this very often, but it's good to know it's at least possible.

The downside of virtual functions

Since most of the time you'll want your functions to be virtual, why not just make all functions virtual? The answer is because it's inefficient -- resolving a virtual function call takes longer than a resolving a regular one. Furthermore, the compiler also has to allocate an extra pointer for each class object that has one or more virtual functions. We'll talk about this more in the next couple of lessons.

12.4 -- Early binding and late binding

Index

12.2 -- Virtual functions

Share this:

C++ TUTORIAL | PRINT THIS POST

26 comments to 12.3 — Virtual destructors, virtual assignment, and overriding virtualization

sandhya
July 11, 2008 at 12:40 am · Reply

12.4 — Early binding and late binding

BY ALEX ON FEBRUARY 7TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 28TH, 2015

In this chapter and the next, we are going to take a closer look at how virtual functions are implemented. While this information is not strictly necessary to effectively use virtual functions, it is interesting. Nevertheless, you can consider both sections optional reading.

When a C++ program is executed, it executes sequentially, beginning at the top of main(). When a function call is encountered, the point of execution jumps to the beginning of the function being called. How does the CPU know to do this?

When a program is compiled, the compiler converts each statement in your C++ program into one or more lines of machine language. Each line of machine language is given its own unique sequential address. This is no different for functions -- when a function is encountered, it is converted into machine language and given the next available address. Thus, each function ends up with a unique machine language address.

Binding refers to the process that is used to convert identifiers (such as variable and function names) into machine language addresses. Although binding is used for both variables and functions, in this lesson we're going to focus on function binding.

Early binding

Most of the function calls the compiler encounters will be direct function calls. A direct function call is a statement that directly calls a function. For example:

```
#include <iostream>

void PrintValue(int nValue)
{
    std::cout << nValue;
}

int main()
{
    PrintValue(5); // This is a direct function call
    return 0;
}
```

Direct function calls can be resolved using a process known as early binding. **Early binding** (also called static binding) means the compiler is able to directly associate the identifier name (such as a function or variable name) with a machine address. Remember that all functions have a unique machine address. So when the compiler encounters a function call, it replaces the function call with a machine language instruction that tells the CPU to jump to the address of the function.

Let's take a look at a simple calculator program that uses early binding:

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

int Add(int nX, int nY)
{
    return nX + nY;
}

int Subtract(int nX, int nY)
{
    return nX - nY;
}

int Multiply(int nX, int nY)
{
    return nX * nY;
}
```

```
18
19  int main()
20  {
21      int nX;
22      cout << "Enter a number: ";
23      cin >> nX;
24
25      int nY;
26      cout << "Enter another number: ";
27      cin >> nY;
28
29      int nOperation;
30      do
31      {
32          cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
33          cin >> nOperation;
34      } while (nOperation < 0 || nOperation > 2);
35
36      int nResult = 0;
37      switch (nOperation)
38      {
39          case 0: nResult = Add(nX, nY); break;
40          case 1: nResult = Subtract(nX, nY); break;
41          case 2: nResult = Multiply(nX, nY); break;
42      }
43
44      cout << "The answer is: " << nResult << endl;
45
46      return 0;
47  }
```

Because Add(), Subtract(), and Multiply() are all direct function calls, the compiler will use early binding to resolve the Add(), Subtract(), and Multiply() function calls. The compiler will replace the Add() function call with an instruction that tells the CPU to jump to the address of the Add() function. The same holds true for Subtract() and Multiply().

Late Binding

In some programs, it is not possible to know which function will be called until runtime (when the program is run). This is known as **late binding** (or dynamic binding). In C++, one way to get late binding is to use function pointers. To review function pointers briefly, a function pointer is a type of pointer that points to a function instead of a variable. The function that a function pointer points to can be called by using the function call operator (()) on the pointer.

For example, the following code calls the Add() function:

```
1   int Add(int nX, int nY)
2   {
3       return nX + nY;
4   }
5
6   int main()
7   {
8       // Create a function pointer and make it point to the Add function
9       int (*pFcn)(int, int) = Add;
10      cout << pFcn(5, 3) << endl; // add 5 + 3
11
12      return 0;
13  }
```

Calling a function via a function pointer is also known as an indirect function call. The following calculator program is functionally identical to the calculator example above, except it uses a function pointer instead of a direct function call:

```
1   #include <iostream>
2   using namespace std;
3
4   int Add(int nX, int nY)
5   {
```

```
6       return nX + nY;
7   }
8
9   int Subtract(int nX, int nY)
10  {
11      return nX - nY;
12  }
13
14  int Multiply(int nX, int nY)
15  {
16      return nX * nY;
17  }
18
19  int main()
20  {
21      int nX;
22      cout << "Enter a number: ";
23      cin >> nX;
24
25      int nY;
26      cout << "Enter another number: ";
27      cin >> nY;
28
29      int nOperation;
30      do
31      {
32          cout << "Enter an operation (0=add, 1=subtract, 2=multiply): ";
33          cin >> nOperation;
34      } while (nOperation < 0 || nOperation > 2);
35
36      // Create a function pointer named pFcn (yes, the syntax is ugly)
37      int (*pFcn)(int, int);
38
39      // Set pFcn to point to the function the user chose
40      switch (nOperation)
41      {
42          case 0: pFcn = Add; break;
43          case 1: pFcn = Subtract; break;
44          case 2: pFcn = Multiply; break;
45      }
46
47      // Call the function that pFcn is pointing to with nX and nY as parameters
48      cout << "The answer is: " << pFcn(nX, nY) << endl;
49
50      return 0;
51  }
```

In this example, instead of calling the Add(), Subtract(), or Multiply() function directly, we've instead set pFcn to point at the function we wish to call. Then we call the function through the pointer. The compiler is unable to use early binding to resolve the function call `pFcn(nX, nY)` because it can not tell which function pFcn will be pointing to at compile time!

Late binding is slightly less efficient since it involves an extra level of indirection. With early binding, the compiler can tell the CPU to jump directly to the function's address. With late binding, the program has to read the address held in the pointer and then jump to that address. This involves one extra step, making it slightly slower. However, the advantage of late binding is that it is more flexible than early binding, because decisions about what function to call do not need to be made until run time.

In the next lesson, we'll take a look at how late binding is used to implement virtual functions.

12.5 -- The virtual table

Index

12.3 -- Virtual destructors, virtual assignment, and overriding virtualization

Share this:

C++ TUTORIAL | PRINT THIS POST

38 comments to 12.4 — Early binding and late binding

Arijit Chattopadhyay
July 29, 2008 at 4:24 am · Reply

Good example.. Easy to understand

sssss
August 26, 2008 at 4:20 am · Reply

very good example

RajenKumar
March 23, 2010 at 9:20 am · Reply

Vary good example….

gswrg
October 23, 2008 at 1:25 am · Reply

Why do you use the “&” in the first example

12.5 — The virtual table

BY ALEX ON FEBRUARY 8TH, 2008 | LAST MODIFIED BY ALEX ON FEBRUARY 8TH, 2008

To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as “vtable”, “virtual function table”, “virtual method table”, or “dispatch table”.

Because knowing how the virtual table works is not necessary to use virtual functions, this section can be considered optional reading.

The virtual table is actually quite simple, though it’s a little complex to describe in words. First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it’s own virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

Second, the compiler also adds a hidden pointer to the base class, which we will call *__vptr. *__vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class. Unlike the *this pointer, which is actually a function parameter used by the compiler to resolve self-references, *__vptr is a real pointer. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that *__vptr is inherited by derived classes, which is important.

By now, you’re probably confused as to how these things all fit together, so let’s take a look at a simple example:

```
1  class Base
2  {
3  public:
4      virtual void function1() {};
5      virtual void function2() {};
6  };
7
8  class D1: public Base
9  {
10 public:
11     virtual void function1() {};
12 };
13
14 class D2: public Base
15 {
16 public:
17     virtual void function2() {};
18 };
```

Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.

The compiler also adds a hidden pointer to the most base class that uses virtual functions. Although the compiler does this automatically, we’ll put it in the next example just to show where it’s added:

```
1  class Base
2  {
3  public:
4      FunctionPointer *__vptr;
5      virtual void function1() {};
6      virtual void function2() {};
7  };
8
9  class D1: public Base
10 {
11 public:
12     virtual void function1() {};
```

```
};

class D2: public Base
{
public:
    virtual void function2() {};
};
```

When a class object is created, *__vptr is set to point to the virtual table for that class. For example, when a object of type Base is created, *__vptr is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, *__vptr is set to point to the virtual table for D1 or D2 respectively.

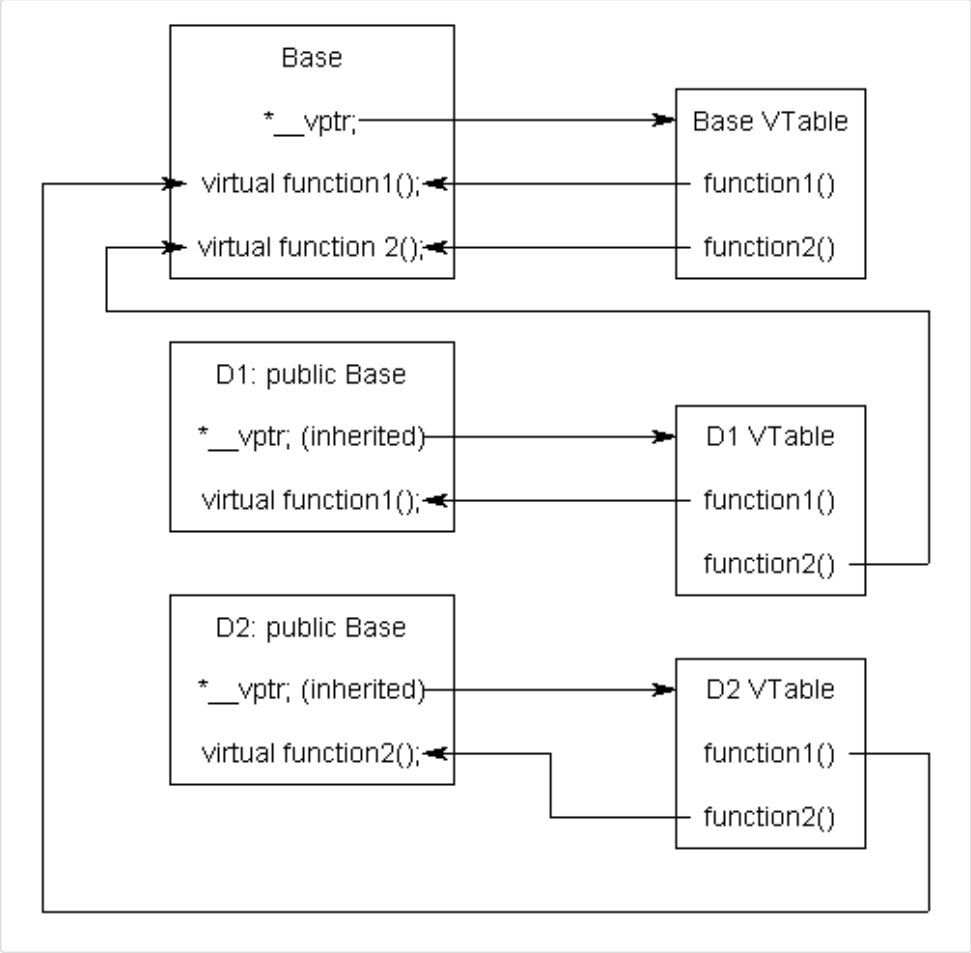
Now, let's talk about how these virtual tables are filled out. Because there are only two virtual functions here, each virtual table will have two entries (one for function1(), and one for function2()). Remember that when these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.

Base's virtual table is simple. An object of type Base can only access the members of Base. Base has no access to D1 or D2 functions. Consequently, the entry for function1 points to Base::function1(), and the entry for function2 points to Base::function2().

D1's virtual table is slightly more complex. An object of type D1 can access members of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn't overridden function2(), so the entry for function2 will point to Base::function2().

D2's virtual table is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().

Here's a picture of this graphically:



Although this diagram is kind of crazy looking, it's really quite simple: the *__vptr in each class points to the virtual table for that class. The entries in the virtual table point to the most-derived version of the function objects of that class are allowed to

call.

So consider what happens when we create an object of type D1:

```
int main()
{
    D1 cClass;
}
```

Because cClass is a D1 object, cClass has it's *__vptr set to the D1 virtual table.

Now, let's set a base pointer to D1:

```
int main()
{
    D1 cClass;
    Base *pClass = &cClass;
}
```

Note that because pClass is a base pointer, it only points to the Base portion of cClass. However, also note that *__vptr is in the Base portion of the class, so pClass has access to this pointer. Finally, note that pClass->__vptr points to the D1 virtual table! Consequently, even though pClass is of type Base, it still has access to D1's virtual table.

So what happens when we try to call pClass->function1()?

```
int main()
{
    D1 cClass;
    Base *pClass = &cClass;
    pClass->function1();
}
```

First, the program recognizes that function1() is a virtual function. Second, uses pClass->__vptr to get to D1's virtual table. Third, it looks up which version of function1() to call in D1's virtual table. This has been set to D1::function1(). Therefore, pClass->function1() resolves to D1::function1()!

Now, you might be saying, "But what if Base really pointed to a Base object instead of a D1 object. Would it still call D1::function1()?". The answer is no.

```
int main()
{
    Base cClass;
    Base *pClass = &cClass;
    pClass->function1();
}
```

In this case, when cClass is created, __vptr points to Base's virtual table, not D1's virtual table. Consequently, pClass->__vptr will also be pointing to Base's virtual table. Base's virtual table entry for function1() points to Base::function1(). Thus, pClass->function1() resolves to Base::function1(), which is the most-derived version of function1() that a Base object should be able to call.

By using these tables, the compiler and program are able to ensure function calls resolve to the appropriate virtual function, even if you're only using a pointer or reference to a base class!

Calling a virtual function is slower than calling a non-virtual function for a couple of reasons: First, we have to use the *__vptr to get to the appropriate virtual table. Second, we have to index the virtual table to find the correct function to call. Only then can we call the function. As a result, we have to do 3 operations to find the function to call, as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call. However, with modern computers, this added time is usually fairly insignificant.

12.6 -- Pure virtual functions, abstract base classes, and interface classes

12.6 — Pure virtual functions, abstract base classes, and interface classes

BY ALEX ON FEBRUARY 13TH, 2008 | LAST MODIFIED BY ALEX ON FEBRUARY 13TH, 2008

Finally, we arrive at the end of our long journey through inheritance! This is the last topic we will cover on the subject. So congratulations in advance on making it through the hardest part of the language!

Pure virtual (abstract) functions and abstract base classes

So far, all of the virtual functions we have written have a body (a definition). However, C++ allows you to create a special kind of virtual function called a **pure virtual function** (or **abstract function**) that has no body at all! A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.

To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

```
class Base
{
public:
    const char* SayHi() { return "Hi"; } // a normal non-virtual function

    virtual const char* GetName() { return "Base"; } // a normal virtual function

    virtual int GetValue() = 0; // a pure virtual function
};
```

When we add a pure virtual function to our class, we are effectively saying, “it is up to the derived classes to implement this function”.

Using a pure virtual function has two main consequences: First, any class with one or more pure virtual functions becomes an **abstract base class**, which means that it can not be instantiated! Consider what would happen if we could create an instance of Base:

```
int main()
{
    Base cBase; // pretend this was legal
    cBase.GetValue(); // what would this do?
}
```

Second, any derived class must define a body for this function, or that derived class will be considered an abstract base class as well.

Let’s take a look at an example of a pure virtual function in action. In a previous lesson, we wrote a simple Animal base class and derived a Cat and a Dog class from it. Here’s the code as we left it:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
```

```
    virtual const char* Speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Woof"; }
};
```

We've prevented people from allocating objects of type Animal by making the constructor protected. However, there's one problem that has not been addressed. It is still possible to create derived classes that do not redefine Speak(). For example:

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    // We forgot to redefine Speak
};

int main()
{
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

This will print:

```
Betsy says ???
```

What happened? We forgot to redefine Speak, so cCow.Speak() resolved to Animal.Speak(), which isn't what we wanted.

A better solution to this problem is to use a pure virtual function:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

public:
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

```

```
    std::string GetName() { return m_strName; }
    virtual const char* Speak() = 0; // pure virtual function
};
```

There are a couple of things to note here. First, Speak() is now a pure virtual function. This means Animal is an abstract base class, and can not be instantiated. Consequently, we do not need to make the constructor protected any longer (though it doesn't hurt). Second, because our Cow class was derived from Animal, but we did not define Cow::Speak(), Cow is also an abstract base class. Now when we try to compile this code:

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    // We forgot to redefine Speak
};

int main()
{
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

The compiler will give us a warning because Cow is an abstract base class and we can not create instances of abstract base classes:

```
C:\Test.cpp(141) : error C2259: 'Cow' : cannot instantiate abstract class due to following
        C:Test.cpp(128) : see declaration of 'Cow'
C:\Test.cpp(141) : warning C4259: 'const char *__thiscall Animal::Speak(void)' : pure virt
```

This tells us that we will only be able to instantiate Cow if Cow provides a body for Speak().

Let's go ahead and do that:

```
class Cow: public Animal
{
public:
    Cow(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Moo"; }
};

int main()
{
    Cow cCow("Betsy");
    cout << cCow.GetName() << " says " << cCow.Speak() << endl;
}
```

Now this program will compile and print:

```
Betsy says Moo
```

A pure virtual function is useful when we have a function that we want to put in the base class, but only the derived classes know what it should return. A pure virtual function makes it so the base class can not be instantiated, and the derived classes are forced to define these function before they can be instantiated. This helps ensure the derived classes do not

forget to redefine functions that the base class was expecting them to.

Interface classes

An **interface class** is a class that has no members variables, and where all of the functions are pure virtual! In other words, the class is purely a definition, and has no actual implementation. Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

Interface classes are often named beginning with an I. Here's a sample interface class:

```
class IErrorLog
{
    virtual bool OpenLog(const char *strFilename) = 0;
    virtual bool CloseLog() = 0;

    virtual bool WriteError(const char *strErrorMessage) = 0;
};
```

Any class inheriting from IErrorLog must provide implementations for all three functions in order to be instantiated. You could derive a class named FileErrorLog, where OpenLog() opens a file on disk, CloseLog() closes it, and WriteError() writes the message to the file. You could derive another class called ScreenErrorLog, where OpenLog() and CloseLog() do nothing, and WriteError() prints the message in a pop-up message box on the screen.

Now, let's say you need to write some code that uses an error log. If you write your code so it includes FileErrorLog or ScreenErrorLog directly, then you're effectively stuck using that kind of error log. For example, the following function effectively forces callers of MySqrt() to use a FileErrorLog, which may or may not be what they want.

```
double MySqrt(double dValue, FileErrorLog &cLog)
{
    if (dValue < 0.0)
    {
        cLog.WriteError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return dValue;
}
```

A much better way to implement this function is to use IErrorLog instead:

```
double MySqrt(double dValue, IErrorLog &cLog)
{
    if (dValue < 0.0)
    {
        cLog.WriteError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return dValue;
}
```

Now the caller can pass in any class that conforms to the IErrorLog interface. If they want the error to go to a file, they can pass in an instance of FileErrorLog. If they want it to go to the screen, they can pass in an instance of ScreenErrorLog. Or if they want to do something you haven't even thought of, such as sending an email to someone when there's an error, they can derive a new class from IErrorLog (eg. EmailErrorLog) and use an instance of that! By using IErrorLog, your function becomes more independent and flexible.

Interface classes have become extremely popular because they are easy to use, easy to extend, and easy to maintain. In fact, some modern languages, such as Java and C#, have added an "interface" keyword that allows programmers to directly define an interface class without having to explicitly mark all of the member functions as abstract. Furthermore, although Java and C# will not let you use multiple inheritance on normal classes, they will let you multiply inherit as many interfaces as you like. Because interfaces have no data and no function bodies, they avoid a lot of the traditional problems with multiple

inheritance while still providing much of the flexibility.

13.1 -- Input and output (I/O) streams

Index

12.5 -- The virtual table

Share this:

C++ TUTORIAL | PRINT THIS POST

35 comments to 12.6 — Pure virtual functions, abstract base classes, and interface classes

Susan
April 14, 2008 at 12:58 am · Reply

Can you please answer why a pure virtual function should be equated to 0 and not to any other number.

Alex
April 15, 2008 at 10:18 am · Reply

As far as I know, =0 is just a syntactic convention for letting the compiler know that this is a pure virtual function.

You can think about it this way: normal functions have a body of code that needs to be executed when they are called. This code has to live somewhere in memory, so the function's name is essentially a pointer to that code.

On the other hand, pure virtual functions have no body, and thus, do not have need for an address at all. 0 is the NULL address.

12.1 — Pointers and references to the base class of derived objects

BY ALEX ON JANUARY 29TH, 2008 | LAST MODIFIED BY ALEX ON SEPTEMBER 22ND, 2015

In the previous chapter, you learned all about how to use inheritance to derive new classes from existing classes. In this chapter, we are going to focus on one of the most important and powerful aspects of inheritance -- virtual functions.

But before we discuss what virtual functions are, let's first set the table for why we need them.

In the chapter on **construction of derived classes**, you learned that when you create a derived class, it is composed of multiple parts: one part for each inherited class, and a part for itself.

For example, here's a simple case:

```
class Base
{
protected:
    int m_nValue;

public:
    Base(int nValue)
        : m_nValue(nValue)
    {
    }

    const char* GetName() { return "Base"; }
    int GetValue() { return m_nValue; }
};

class Derived: public Base
{
public:
    Derived(int nValue)
        : Base(nValue)
    {
    }

    const char* GetName() { return "Derived"; }
    int GetValueDoubled() { return m_nValue * 2; }
};
```

When we create a Derived object, it contains a Base part (which is constructed first), and a Derived part (which is constructed second). Remember that inheritance implies an is-a relationship between two classes. Since a Derived is-a Base, it is appropriate that Derived contain a Base part of it.

Pointers, references, and derived classes

It should be fairly intuitive that we can set Derived pointers and references to Derived objects:

```
int main()
{
    using namespace std;
    Derived cDerived(5);
    cout << "cDerived is a " << cDerived.GetName() << " and has value " << cDerived.GetValue()
<< endl;

    Derived &rDerived = cDerived;
    cout << "rDerived is a " << rDerived.GetName() << " and has value " << rDerived.GetValue()
<< endl;

    Derived *pDerived = &cDerived;
    cout << "pDerived is a " << pDerived->GetName() << " and has value " << pDerived->GetValue
() << endl;
```

```
    return 0;
}
```

This produces the following output:

```
cDerived is a Derived and has value 5
rDerived is a Derived and has value 5
pDerived is a Derived and has value 5
```

However, since Derived has a Base part, a more interesting question is whether C++ will let us set a Base pointer or reference to a Derived object. It turns out, we can!

```
int main()
{
    using namespace std;
    Derived cDerived(5);

    // These are both legal!
    Base &rBase = cDerived;
    Base *pBase = &cDerived;

    cout << "cDerived is a " << cDerived.GetName() << " and has value " << cDerived.GetValue()
<< endl;
    cout << "rBase is a " << rBase.GetName() << " and has value " << rBase.GetValue() << endl;
    cout << "pBase is a " << pBase->GetName() << " and has value " << pBase->GetValue() << end
l;

    return 0;
}
```

This produces the result:

```
cDerived is a Derived and has value 5
rBase is a Base and has value 5
pBase is a Base and has value 5
```

This result may not be quite what you were expecting at first!

It turns out that because rBase and pBase are a Base reference and pointer, they can only see members of Base (or any classes that Base inherited). So even though Derived::GetName() is an override of Base::GetName(), the Base pointer/reference can not see Derived::GetName(). Consequently, they call Base::GetName(), which is why rBase and pBase report that they are a Base rather than a Derived.

Note that this also means it is not possible to call Derived::GetValueDoubled() using rBase or pBase. They are unable to see anything in Derived.

Here's another slightly more complex example that we'll build on in the next lesson:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }
```

```
14
15  public:
16      std::string GetName() { return m_strName; }
17      const char* Speak() { return "???"; }
18  };
19
20  class Cat: public Animal
21  {
22  public:
23      Cat(std::string strName)
24          : Animal(strName)
25      {
26      }
27
28      const char* Speak() { return "Meow"; }
29  };
30
31  class Dog: public Animal
32  {
33  public:
34      Dog(std::string strName)
35          : Animal(strName)
36      {
37      }
38
39      const char* Speak() { return "Woof"; }
40  };
41
42  int main()
43  {
44      Cat cCat("Fred");
45      cout << "cCat is named " << cCat.GetName() << ", and it says " << cCat.Speak() << endl;
46
47      Dog cDog("Garbo");
48      cout << "cDog is named " << cDog.GetName() << ", and it says " << cDog.Speak() << endl;
49
50      Animal *pAnimal = &cCat;
51      cout << "pAnimal is named " << pAnimal->GetName() << ", and it says " << pAnimal->Speak()
52  << endl;
53
54      pAnimal = &cDog;
55      cout << "pAnimal is named " << pAnimal->GetName() << ", and it says " << pAnimal->Speak()
56  << endl;
57
      return 0;
  }
```

This produces the result:

```
cCat is named Fred, and it says Meow
cDog is named Garbo, and it says Woof
pAnimal is named Fred, and it says ???
pAnimal is named Garbo, and it says ???
```

We see the same issue here. Because pAnimal is an Animal pointer, it can only see the Animal class. Consequently, `pAnimal->Speak()` calls Animal::Speak() rather than the Dog::Speak() or Cat::Speak() function.

Use for pointers and references to base classes

Now you might be saying, “The above examples seem kind of silly. Why would I set a pointer or reference to the base class of a derived object when I can just use the derived object?” It turns out that there are quite a few good reasons.

First, let’s say you wanted to write a function that printed an animal’s name and sound. Without using a pointer to a base class, you’d have to write it like this:

```
void Report(Cat &cCat)
{
    cout << cCat.GetName() << " says " << cCat.Speak() << endl;
}

void Report(Dog &cDog)
{
    cout << cDog.GetName() << " says " << cDog.Speak() << endl;
}
```

Not too difficult, but consider what would happen if we had 30 different animal types instead of 2. You'd have to write 30 almost identical functions! Plus, if you ever added a new type of animal, you'd have to write a new function for that one too. This is a huge waste of time considering the only real difference is the type of the parameter.

However, because Cat and Dog are derived from Animal, Cat and Dog have an Animal part. Therefore, it makes sense that we should be able to do something like this:

```
void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
}
```

This would let us pass in any class derived from Animal, even ones that we haven't thought of yet! Instead of one function per animal, we get one function that works with all classes derived from Animal!

The problem, is of course, that because cAnimal is an Animal reference, `cAnimal.Speak()` will call Animal::Speak() instead of the derived version of Speak().

Second, let's say you had 3 cats and 3 dogs that you wanted to keep in an array for easy access. Because arrays can only hold objects of one type, without a pointer or reference to a base class, you'd have to do it like this:

```
Cat acCats[] = { Cat("Fred"), Cat("Tyson"), Cat("Zeke") };
Dog acDogs[] = { Dog("Garbo"), Dog("Pooky"), Dog("Truffle") };

for (int iii=0; iii < 3; iii++)
    cout << acCats[iii].GetName() << " says " << acCats[iii].Speak() << endl;

for (int iii=0; iii < 3; iii++)
    cout << acDogs[iii].GetName() << " says " << acDogs[iii].Speak() << endl;
```

Now, consider what would happen if you had 30 different types of animals. You'd need 30 arrays, one for each type of animal!

However, because both Cat and Dog are Animal, it makes sense that we should be able to do something like this:

```
Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");
Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");

// Set up an array of pointers to animals, and set those pointers to our Cat and Dog objects
Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson, &cZeke };
for (int iii=0; iii < 6; iii++)
    cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]->Speak() << endl;
```

While this compiles and executes, unfortunately the fact that apcAnimals is a pointer to an Animal means that `apcAnimals[iii]->Speak()` will call Animal::Speak() instead of the proper derived class version of Speak().

Although both of these techniques could save us a lot of time and energy, they have the same problem. The pointer or reference to the base class calls the base version of the function rather than the derived version. If only there was some way to make those base pointers call the derived version of a function instead of the base version…

Want to take a guess what virtual functions are for? 🙂

12.2 — Virtual functions and polymorphism

BY ALEX ON JANUARY 30TH, 2008 | LAST MODIFIED BY ALEX ON DECEMBER 28TH, 2015

In the previous lesson on **pointers and references to the base class of derived objects**, we took a look at a number of examples where using pointers or references to a base class had the potential to simplify code. However, in every case, we ran up against the problem that the base pointer or reference was only able to call the base version of a function, not a derived version.

Here's a simple example of this behavior:

```
class Base
{
protected:

public:
    const char* GetName() { return "Base"; }
};

class Derived: public Base
{
public:
    const char* GetName() { return "Derived"; }
};

int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    cout << "rBase is a " << rBase.GetName() << endl;
}
```

This example prints the result:

```
rBase is a Base
```

Because rBase is a Base pointer, it calls Base::GetName(), even though it's actually pointing to the Base portion of a Derived object.

In this lesson, we will address this issue using virtual functions.

Virtual functions and polymorphism

A **virtual function** is a special type of function that resolves to the most-derived version of the function with the same signature. This capability is known as **polymorphism**.

To make a function virtual, simply place the "virtual" keyword before the function declaration.

Note that virtual functions and **virtual base classes** are two entirely different concepts, even though they share the same keyword.

Here's the above example with a virtual function:

```
class Base
{
protected:

public:
    virtual const char* GetName() { return "Base"; }
};

```

```
class Derived: public Base
{
public:
    virtual const char* GetName() { return "Derived"; }
};

int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

This example prints the result:

```
rBase is a Derived
```

Because rBase is a pointer to the Base portion of a Derived object, when **rBase.GetName()** is evaluated, it would normally resolve to Base::GetName(). However, Base::GetName() is virtual, which tells the program to go look and see if there are any more-derived versions of the function available. Because the Base object that rBase is pointing to is actually part of a Derived object, the program will check every inherited class between Base and Derived and use the most-derived version of the function that it finds. In this case, that is Derived::GetName()!

Let's take a look at a slightly more complex example:

```
class A
{
public:
    virtual const char* GetName() { return "A"; }
};

class B: public A
{
public:
    virtual const char* GetName() { return "B"; }
};

class C: public B
{
public:
    virtual const char* GetName() { return "C"; }
};

class D: public C
{
public:
    virtual const char* GetName() { return "D"; }
};

int main()
{
    C cClass;
    A &rBase = cClass;
    cout << "rBase is a " << rBase.GetName() << endl;

    return 0;
}
```

What do you think this program will output?

Let's look at how this works. First, we instantiate a C class object. rBase is an A pointer, which we set to point to the A portion of the C object. Finally, we call rBase.GetName(). rBase.GetName() evaluates to A::GetName(). However,

A::GetName() is virtual, so the compiler will check all the classes between A and C to see if it can find a more-derived match. First, it checks B::GetName(), and finds a match. Then it checks C::GetName() and finds a better match. It does not check D::GetName() because our original object was a C, not a D. Consequently, rBase.GetName() resolves to C::GetName().

As a result, our program outputs:

```
rBase is a C
```

A more complex example

Let's take another look at the Animal example we were working with in the previous lesson. Here's the original class:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
    const char* Speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    const char* Speak() { return "Woof"; }
};
```

And here's the class with virtual functions:

```
#include <string>
class Animal
{
protected:
    std::string m_strName;

    // We're making this constructor protected because
    // we don't want people creating Animal objects directly,
    // but we still want derived classes to be able to use it.
```

```
    Animal(std::string strName)
        : m_strName(strName)
    {
    }

public:
    std::string GetName() { return m_strName; }
    virtual const char* Speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Meow"; }
};

class Dog: public Animal
{
public:
    Dog(std::string strName)
        : Animal(strName)
    {
    }

    virtual const char* Speak() { return "Woof"; }
};
```

Note that we didn't make Animal::GetName() virtual. This is because GetName() is never overridden in any of the derived classes, therefore there is no need.

Now, using the virtual Speak() function, the following function should work correctly:

```
void Report(Animal &rAnimal)
{
    cout << rAnimal.GetName() << " says " << rAnimal.Speak() << endl;
}

int main()
{
    Cat cCat("Fred");
    Dog cDog("Garbo");

    Report(cCat);
    Report(cDog);
}
```

Indeed, this program produces the result:

```
Fred says Meow
Garbo says Woof
```

When cAnimal.Speak() is evaluated, the program notes that it is a virtual function. In the case where rAnimal is pointing to the Animal portion of a Cat object, the program looks at all the classes between Animal and Cat to see if it can find a more derived function. In that case, it finds Cat::Speak(). In the case where rAnimal points to the Animal portion of a Dog object, the program resolves the function call to Dog::Speak().

Similarly, the following array example now works as expected:

```
Cat cFred("Fred"), cTyson("Tyson"), cZeke("Zeke");
Dog cGarbo("Garbo"), cPooky("Pooky"), cTruffle("Truffle");
```

```
3
4   // Set up an array of pointers to animals, and set those pointers to our Cat and Dog objects
5   Animal *apcAnimals[] = { &cFred, &cGarbo, &cPooky, &cTruffle, &cTyson, &cZeke };
6   for (int iii=0; iii < 6; iii++)
7       cout << apcAnimals[iii]->GetName() << " says " << apcAnimals[iii]->Speak() << endl;
```

Which produces the result:

```
Fred says Meow
Garbo says Woof
Pooky says Woof
Truffle says Woof
Tyson says Meow
Zeke says Meow
```

Even though these two examples only use Cat and Dog, any other classes we derive from Animal would also work with our Report() function and Animal array without further modification! This is perhaps the biggest benefit of virtual functions -- the ability to structure your code in such a way that newly derived classes will automatically work with the old code without modification!

A word of warning: the signature of the derived class function must exactly match the signature of the base class virtual function in order for the derived class function to be used. If the derived class function has different parameter types, the program will likely still compile fine, but the virtual function will not resolve as intended.

Use of the virtual keyword

Technically, the virtual keyword is not needed in derived class. For example:

```
1   class Base
2   {
3   protected:
4
5   public:
6       virtual const char* GetName() { return "Base"; }
7   };
8
9   class Derived: public Base
10  {
11  public:
12      const char* GetName() { return "Derived"; } // note lack of virtual keyword
13  };
14
15  int main()
16  {
17      Derived cDerived;
18      Base &rBase = cDerived;
19      cout << "rBase is a " << rBase.GetName() << endl;
20
21      return 0;
22  }
```

prints

```
rBase is a Derived
```

Exactly the same as if Derived::GetName() was explicitly tagged as virtual. Only the most base class function needs to be tagged as virtual for all of the derived functions to work virtually. However, having the keyword virtual on the derived functions does not hurt, and it serves as a useful reminder that the function is a virtual function rather than a normal one. Consequently, it's generally a good idea to use the virtual keyword for virtualized functions in derived classes even though it's not strictly necessary.

Return types of virtual functions

Under normal circumstances, the return type of a virtual function and its override must match. Thus, the following will not work:

```
class Base
{
public:
    virtual int GetValue() { return 5; }
};

class Derived: public Base
{
public:
    virtual double GetValue() { return 6.78; }
};
```

However, there is one special case in which this is not true. If the return type of a virtual function is a pointer or a reference to a class, override functions can return a pointer or a reference to a derived class. These are called covariant return types. Here is an example:

```
class Base
{
public:
    // This version of GetThis() returns a pointer to a Base class
    virtual Base* GetThis() { return this; }
};

class Derived: public Base
{
    // Normally override functions have to return objects of the same type as the base functio
n
    // However, because Derived is derived from Base, it's okay to return Derived* instead of
Base*
    virtual Derived* GetThis() { return this; }
};
```

Note that some older compilers (eg. Visual Studio 6) do not support covariant return types.

12.3 -- Virtual destructors, virtual assignment, and overriding virtualization

Index

12.1 -- Pointers and references to the base class of derived objects

Share this: