

## 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.

```
1 | int *ptr = new int; // ptr is assigned 4 bytes in the heap
2 | 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!

```
1 | int *ptr1 = new int;
2 | int *ptr2 = new int;
3 | // 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:

```
1  int main()
2  {
3      int stack[100000000];
4      return 0;
5  }
```

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:

```
1  void foo()
2  {
3      foo();
4  }
5
6  int main()
7  {
8      foo();
9
10     return 0;
11 }
```

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](#)

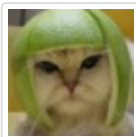
### 62 comments to 7.9 — The stack and the heap



**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
```

```

7   };
8
9   int main()
10  {
11      DateClass date;
12      date.m_month = 10; // okay because m_month is public
13      date.m_day = 14;  // okay because m_day is public
14      date.m_year = 2020; // okay because m_year is public
15
16      return 0;
17  }

```

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:

```

1   #include <iostream>
2
3   class DateClass // members are private by default
4   {
5       int m_month; // private by default, can only be accessed by other members
6       int m_day;   // private by default, can only be accessed by other members
7       int m_year;  // private by default, can only be accessed by other members
8
9   public:
10      void setDate(int month, int day, int year) // public, can be accessed by anyone
11      {
12          // setDate() can access the private members of the class because it is a member of the
13      class itself
14          m_month = month;
15          m_day = day;
16          m_year = year;
17      }
18
19      void print() // public, can be accessed by anyone
20      {
21          std::cout << m_month << "/" << m_day << "/" << m_year;
22      }
23  };
24
25  int main()
26  {
27      DateClass date;
28      date.setDate(10, 14, 2020); // okay, because setDate() is public
29      date.print(); // okay, because print() is public
30
31      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:

```
1  int main()
2  {
3      Point3d point;
4      point.setValues(1.0, 2.0, 3.0);
5
6      point.print();
7
8      return 0;
9  }
```

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 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:

```

1  int main()
2  {
3      Stack stack;
4      stack.reset();
5
6      stack.print();
7
8      stack.push(5);
9      stack.push(3);
10     stack.push(8);
11     stack.print();
12
13     stack.pop();
14     stack.print();
15
16     stack.pop();
17     stack.pop();
18
19     stack.print();
20
21     return 0;
22 }
```

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:

```
1  class Square
2  {
3  private:
4      double m_length = 1.0; // m_length has a default value of 1.0
5      double m_width = 1.0; // m_width has a default value of 1.0
6
7  public:
8      Square()
9      {
10         // This constructor will use the default values above since they aren't overridden here
11     }
12
13     void print()
14     {
15         std::cout << "length: " << m_length << ", width: " << m_width << '\n';
16     }
17 };
18
19 int main()
20 {
21     Square x; // x.m_length = 1.0, x.m_width = 1.0
22     x.print();
23
24     return 0;
25 }
```

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:

```
1  class Square
2  {
3  private:
4      double m_length = 1.0;
5      double m_width = 1.0;
6
7  public:
8
9      // note: No default constructor provided in this example
10
11     Square(double length, double width)
12         : m_length(length), m_width(width)
13     {
14         // m_length and m_width are initialized by the constructor (the default values aren't
15         used)
16     }
17 }
```

```

18     void print()
19     {
20         std::cout << "length: " << m_length << ", width: " << m_width << '\n';
21     }
22
23 };
24
25 int main()
26 {
27     Square x; // will not compile, no default constructor exists, even though members have default initialization values
28
29     return 0;
30 }

```

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:

```

1  class Square
2  {
3  private:
4      double m_length = 1.0;
5      double m_width = 1.0;
6
7  public:
8
9      Square(double length, double width)
10         : m_length(length), m_width(width)
11     {
12         // m_length and m_width are initialized by the constructor (the default values aren't
13         used)
14     }
15
16     void print()
17     {
18         std::cout << "length: " << m_length << ", width: " << m_width << '\n';
19     }
20
21 };
22
23 int main()
24 {
25     Square x(2.0, 3.0);
26     x.print();
27
28     return 0;
29 }

```

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.

```

1  #include <string>
2  #include <iostream>
3  class Ball
4  {

```

```
5 private:
6     std::string m_color;
7     double m_radius;
8
9 public:
10    // Default constructor with no parameters
11    Ball()
12    {
13        m_color = "black";
14        m_radius = 10.0;
15    }
16
17    // Constructor with only color parameter (radius will use default value)
18    Ball(const std::string &color)
19    {
20        m_color = color;
21        m_radius = 10.0;
22    }
23
24    // Constructor with only radius parameter (color will use default value)
25    Ball(double radius)
26    {
27        m_color = "black";
28        m_radius = radius;
29    }
30
31    // Constructor with both color and radius parameters
32    Ball(const std::string &color, double radius)
33    {
34        m_color = color;
35        m_radius = radius;
36    }
37
38    void print()
39    {
40        std::cout << "color: " << m_color << ", radius: " << m_radius << '\n';
41    }
42 };
43
44 int main()
45 {
46     Ball def;
47     def.print();
48
49     Ball blue("blue");
50     blue.print();
51
52     Ball twenty(20.0);
53     twenty.print();
54
55     Ball blueTwenty("blue", 20.0);
56     blueTwenty.print();
57
58     return 0;
59 }
```

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](#)

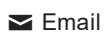


[Index](#)



[8.5a -- Constructor member initializer lists](#)

### Share this:



Email



Facebook



Twitter



Google



Pinterest

[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:

```
1  #include <iostream>
2
3  int generateID()
4  {
5      static int s_id = 0;
6      return ++s_id;
7  }
8
9  int main()
10 {
11     std::cout << generateID() << '\n';
12     std::cout << generateID() << '\n';
13     std::cout << generateID() << '\n';
14
15     return 0;
16 }
```

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:

```
1  class Something
2  {
3  public:
4      int m_value = 1;
5  };
6
7  int main()
8  {
9      Something first;
10     Something second;
11
12     second.m_value = 2;
13
14     std::cout << first.m_value << '\n';
15     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 below)
8
9

```

```

10 | int main()
11 | {
12 |     // note: we're not instantiating any objects of type Something
13 |
14 |     Something::s_value = 2;
15 |     std::cout << Something::s_value << '\n';
16 |     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:

```

1 | 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:

```

1 | class Whatever
2 | {
3 | public:
4 |     static const int s_value = 4; // a static const int can be declared and initialized directly
5 | }

```

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:

```

1 | class Something
2 | {
3 | private:
4 |     static int s_idGenerator;
5 |     int m_id;
6 |
7 | public:
8 |     Something() { m_id = s_idGenerator++; } // grab the next value from the id generator
9 |
10 |    int getID() const { return m_id; }
11 | };
12 |
13 | // Note that we're defining and initializing s_idGenerator even though it is declared as private

```



```
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:



Email



Facebook 36



Twitter



Google



Pinterest

## 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:

```

1  class Accumulator
2  {
3  private:
4      int m_value;
5  public:
6      Accumulator() { m_value = 0; }
7      void add(int value) { m_value += value; }
8
9      // Make the reset() function a friend of this class
10     friend void reset(Accumulator &accumulator);
11 };
12
13 // reset() is now a friend of the Accumulator class
14 void reset(Accumulator &accumulator)
15 {
16     // And can access the private data of Accumulator objects
17     accumulator.m_value = 0;
18 }
19
20 int main()
21 {
22     Accumulator acc;
23     acc.add(5); // add 5 to the accumulator
24     reset(acc); // reset the accumulator to 0
25
26     return 0;
27 }
```

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:

```

1  class Value
2  {
3  private:
4      int m_value;
5  public:
6      Value(int value) { m_value = value; }
7      friend bool isEqual(const Value &value1, const Value &value2);
8  };
9
10 bool isEqual(const Value &value1, const Value &value2)
11 {
12     return (value1.m_value == value2.m_value);
13 }
```

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:

```

1  class Humidity;
2
3  class Temperature
4  {
5  private:
6      int m_temp;
7  public:
8      Temperature(int temp=0) { m_temp = temp; }
9
10     void setTemperature(int temp) { m_temp = temp; }
11
12     friend void printWeather(const Temperature &temperature, const Humidity &humidity);
13 };
14
15 class Humidity
16 {
17 private:
18     int m_humidity;
19 public:
20     Humidity(int humidity=0) { m_humidity = humidity; }
21
22     void setHumidity(int humidity) { m_humidity = humidity; }
23
24     friend void printWeather(const Temperature &temperature, const Humidity &humidity);
25 };
26
27 void printWeather(const Temperature &temperature, const Humidity &humidity)
28 {
29     std::cout << "The temperature is " << temperature.m_temp <<
30         " and the humidity is " << humidity.m_humidity << '\n';
31 }
```

```

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     }
35 };

```

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.

```

1  class Storage; // forward declaration for class Storage
2
3  class Display
4  {
5  private:
6      bool m_displayIntFirst;
7
8  public:
9      Display(bool displayIntFirst) { m_displayIntFirst = displayIntFirst; }
10
11     void displayItem(Storage &storage); // forward declaration above needed for this declarati
12 on line
13 };
14
15 class Storage
16 {
17 private:
18     int m_nValue;
19     double m_dValue;
20 public:
21     Storage(int nValue, double dValue)
22     {
23         m_nValue = nValue;
24         m_dValue = dValue;
25     }
26
27     // Make the Display class a friend of Storage (requires seeing the full declaration of cla
28 ss Display, as above)
29     friend void Display::displayItem(Storage& storage);
30 };
31
32 // Now we can define Display::displayItem, which needs to have seen the full declaration of cl
33 ass Storage
34 void Display::displayItem(Storage &storage)
35 {
36     if (m_displayIntFirst)
37         std::cout << storage.m_nValue << " " << storage.m_dValue << '\n';
38     else // display double first
39         std::cout << storage.m_dValue << " " << storage.m_nValue << '\n';
40 }
41
42 int main()
43 {
44     Storage storage(5, 6.7);
45     Display display(false);
46
47     display.displayItem(storage);
48
49     return 0;
50 }

```

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).

Friendship 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:

```

1  #include <iostream>
2
3  class Vector3d
4  {
5  private:
6      double m_x = 0.0, m_y = 0.0, m_z = 0.0;
7
8  public:
9      Vector3d(double x = 0.0, double y = 0.0, double z = 0.0)
10         : m_x(x), m_y(y), m_z(z)
11     {
12
13     }
14
15     void print()
16     {
17         std::cout << "Vector(" << m_x << " , " << m_y << " , " << m_z << ")\n";
18     }
19 };
20
21 class Point3d
22 {
23 private:
24     double m_x=0.0, m_y=0.0, m_z=0.0;
25
26 public:
27     Point3d(double x = 0.0, double y = 0.0, double z = 0.0)
28         : m_x(x), m_y(y), m_z(z)
29     {
30
31     }
32
33     void print()
34     {
35         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):

```
1  #include "CPU.h"
2  #include "Motherboard.h"
3  #include "RAM.h"
4
5  class PersonalComputer
6  {
7  private:
8      CPU m_cCPU;
9      Motherboard m_cMotherboard;
10     RAM m_cRAM;
11 };
```

### 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.

```
1  PersonalComputer::PersonalComputer(int nCPUSpeed, char *strMotherboardModel, int nRAMSize)
2      : m_cCPU(nCPUSpeed), m_cMotherboard(strMotherboardModel), m_cRAM(nRAMSize)
3  {
4  }
```

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:

```

1  #ifndef POINT2D_H
2  #define POINT2D_H
3
4  #include <iostream>
5
6  class Point2D
7  {
8  private:
9      int m_nX;
10     int m_nY;
11
12 public:
13     // A default constructor
14     Point2D()
15         : m_nX(0), m_nY(0)
16     {
17     }
18
19     // A specific constructor
20     Point2D(int nX, int nY)
21         : m_nX(nX), m_nY(nY)
22     {
23     }
24
25     // An overloaded output operator
26     friend std::ostream& operator<<(std::ostream& out, const Point2D &cPoint)
27     {
28         out << "(" << cPoint.GetX() << ", " << cPoint.GetY() << ")";
29         return out;
30     }
31
32     // Access functions
33     void SetPoint(int nX, int nY)
34     {
35         m_nX = nX;
36         m_nY = nY;
37     }
38
39     int GetX() const { return m_nX; }
40     int GetY() const { return m_nY; }
41 };
42
43 #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:

```

1  #ifndef CREATURE_H
2  #define CREATURE_H
3
4  #include <iostream>
5  #include <string>
6  #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](#)

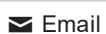


### [Index](#)



### [9.12 -- Shallow vs deep copying](#)

## Share this:

[Email](#)[Facebook](#) 14[Twitter](#)[Google](#)[Pinterest](#)

## 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.

### Aggregation

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.

```
1  #include <string>
2  using namespace std;
3
4  class Teacher
5  {
6  private:
7      string m_strName;
8  public:
9      Teacher(string strName)
10         : m_strName(strName)
11     {
12     }
13
14     string GetName() { return m_strName; }
15 };
16
17 class Department
18 {
19 private:
20     Teacher *m_pcTeacher; // This dept holds only one teacher
21
22 public:
23     Department(Teacher *pcTeacher=NULL)
24         : m_pcTeacher(pcTeacher)
25     {
26     }
27 };
28
```

```

29  int main()
30  {
31      // Create a teacher outside the scope of the Department
32      Teacher *pTeacher = new Teacher("Bob"); // create a teacher
33      {
34          // Create a department and use the constructor parameter to pass
35          // the teacher to it.
36          Department cDept(pTeacher);
37
38      } // cDept goes out of scope here and is destroyed
39
40      // pTeacher still exists here because cDept did not destroy it
41      delete pTeacher;
42  }

```

In this case, `pTeacher` is created independently 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 is 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 its 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
```

```
8 | #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.

```
1 | #ifndef INTARRAY_H
2 | #define INTARRAY_H
3 |
4 | class IntArray
5 | {
6 | private:
7 |     int m_nLength;
8 |     int *m_pnData;
9 | };
10 |
11 | #endif
```

Now we need to add some constructors that will allow us to create `IntArray`s. 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.

```
1 | #ifndef INTARRAY_H
2 | #define INTARRAY_H
3 |
4 | class IntArray
5 | {
6 | private:
7 |     int m_nLength;
8 |     int *m_pnData;
9 |
10 | public:
11 |     IntArray()
12 |     {
13 |         m_nLength = 0;
14 |         m_pnData = 0;
15 |     }
16 |
17 |     IntArray(int nLength)
18 |     {
19 |         m_pnData = new int[nLength];
20 |         m_nLength = nLength;
21 |     }
22 | };
23 |
24 | #endif
```

We'll also need some functions to help us clean up `IntArray`s. 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.

```
1 | ~IntArray()
2 | {
3 |     delete[] m_pnData;
4 | }
5 |
6 | void Erase()
7 | {
8 |     delete[] m_pnData;
9 |     // We need to make sure we set m_pnData to 0 here, otherwise it will
10 |    // be left pointing at deallocated memory!
11 |     m_pnData = 0;
12 |     m_nLength = 0;
13 | }
```

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.



```

1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3
4  #include <assert.h> // for assert()
5
6  class IntArray
7  {
8  private:
9      int m_nLength;
10     int *m_pnData;
11
12 public:
13     IntArray()
14     {
15         m_nLength = 0;
16         m_pnData = 0;
17     }
18
19     IntArray(int nLength)
20     {
21         m_pnData = new int[nLength];
22         m_nLength = nLength;
23     }
24
25     ~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     int GetLength() { return m_nLength; }
46 };
47
48 #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.

```

1  // Reallocate resizes the array. Any existing elements will be destroyed.
2  // This function operates quickly.
3  void Reallocate(int nNewLength)
4  {
5      // First we delete any existing elements
6      Erase();
7

```

```

8      // If our array is going to be empty now, return here
9      if (nNewLength<= 0)
10         return;
11
12     // Then we have to allocate new elements
13     m_pnData = new int[nNewLength];
14     m_nLength = nNewLength;
15 }
16
17 // Resize resizes the array. Any existing elements will be kept.
18 // This function operates slowly.
19 void Resize(int nNewLength)
20 {
21     // If we are resizing to an empty array, do that and return
22     if (nNewLength <= 0)
23     {
24         Erase();
25         return;
26     }
27
28     // Now we can assume nNewLength is at least 1 element. This algorithm
29     // works as follows: First we are going to allocate a new array. Then we
30     // are going to copy elements from the existing array to the new array.
31     // Once that is done, we can destroy the old array, and make m_pnData
32     // point to the new array.
33
34     // First we have to allocate a new array
35     int *pnData = new int[nNewLength];
36
37     // Then we have to figure out how many elements to copy from the existing
38     // array to the new array. We want to copy as many elements as there are
39     // in the smaller of the two arrays.
40     if (m_nLength > 0)
41     {
42         int nElementsToCopy = (nNewLength > m_nLength) ? m_nLength : nNewLength;
43
44         // Now copy the elements one by one
45         for (int nIndex=0; nIndex < nElementsToCopy; nIndex++)
46             pnData[nIndex] = m_pnData[nIndex];
47     }
48
49     // Now we can delete the old array because we don't need it any more
50     delete[] m_pnData;
51
52     // And use the new array instead! Note that this simply makes m_pnData point
53     // to the same address as the new array we dynamically allocated. Because
54     // pnData was dynamically allocated, it won't be destroyed when it goes out of scope.
55     m_pnData = pnData;
56     m_nLength = nNewLength;
57 }

```

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().

```

1      void InsertBefore(int nValue, int nIndex)
2      {
3          // Sanity check our nIndex value
4          assert(nIndex >= 0 && nIndex <= m_nLength);
5
6          // First create a new array one element larger than the old array
7          int *pnData = new int[m_nLength+1];
8
9          // Copy all of the elements up to the index
10         for (int nBefore=0; nBefore < nIndex; nBefore++)
11             pnData[nBefore] = m_pnData[nBefore];

```

```

12
13 // Insert our new element into the new array
14 pData[nIndex] = nValue;
15
16 // Copy all of the values after the inserted element
17 for (int nAfter=nIndex; nAfter < m_nLength; nAfter++)
18     pData[nAfter+1] = m_pnData[nAfter];
19
20 // Finally, delete the old array, and use the new array instead
21 delete[] m_pnData;
22 m_pnData = pData;
23 m_nLength += 1;
24 }
25
26 void Remove(int nIndex)
27 {
28     // Sanity check our nIndex value
29     assert(nIndex >= 0 && nIndex < m_nLength);
30
31     // First create a new array one element smaller than the old array
32     int *pData = new int[m_nLength-1];
33
34     // Copy all of the elements up to the index
35     for (int nBefore=0; nBefore < nIndex; nBefore++)
36         pData[nBefore] = m_pnData[nBefore];
37
38     // Copy all of the values after the removed element
39     for (int nAfter=nIndex+1; nAfter < m_nLength; nAfter++)
40         pData[nAfter-1] = m_pnData[nAfter];
41
42     // Finally, delete the old array, and use the new array instead
43     delete[] m_pnData;
44     m_pnData = pData;
45     m_nLength -= 1;
46 }
47
48 // A couple of additional functions just for convenience
49 void InsertAtBeginning(int nValue) { InsertBefore(nValue, 0); }
50 void InsertAtEnd(int nValue) { InsertBefore(nValue, m_nLength); }

```

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

```

1  #ifndef INTARRAY_H
2  #define INTARRAY_H
3
4  #include <assert.h> // for assert()
5
6  class IntArray
7  {
8  private:
9      int m_nLength;
10     int *m_pnData;
11
12 public:
13     IntArray()
14     {
15         m_nLength = 0;
16         m_pnData = 0;
17     }
18
19     IntArray(int nLength)
20     {
21         m_pnData = new int[nLength];
22         m_nLength = nLength;
23     }
24
25     ~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:

```
1  #include <iostream>
2  #include "IntArray.h"
3
4  using namespace std;
5
6  int main()
7  {
8      // Declare an array with 10 elements
9      IntArray cArray(10);
10
11     // Fill the array with numbers 1 through 10
12     for (int i=0; i<10; i++)
13         cArray[i] = i+1;
14
15     // Resize the array to 8 elements
16     cArray.Resize(8);
17
18     // Insert the number 20 before the 5th element
19     cArray.InsertBefore(20, 5);
20
21     // Remove the 3rd element
22     cArray.Remove(3);
23
24     // Add 30 and 40 to the end and beginning
25     cArray.InsertAtEnd(30);
26     cArray.InsertAtBeginning(40);
27
28     // Print out all the numbers
29     for (int j=0; j<cArray.GetLength(); j++)
30         cout << cArray[j] << " ";
31
32     return 0;
33 }
```

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:**

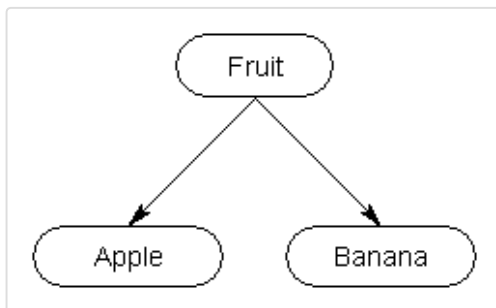
## 11.1 — Introduction to inheritance

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.

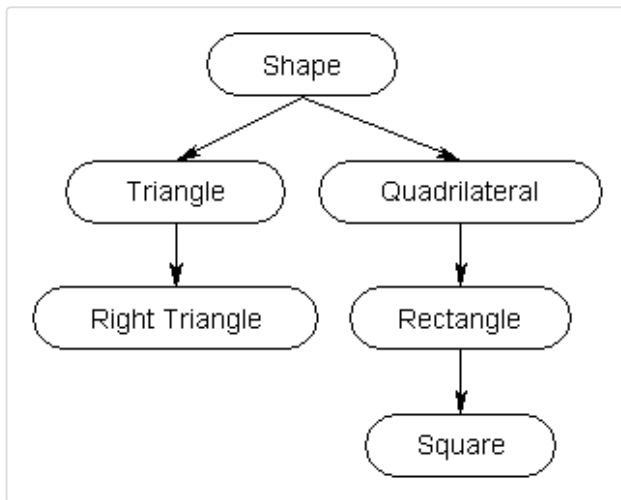
Consider an apple and a banana. Although an apple and a banana are different fruits, both have in common that they *are* fruits. Because apples and bananas *are* fruits, anything that is true of fruits is also true of apples and bananas. For example, all fruits have a name, a flavor, and are tangible 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:

```

1  #include <string>
2  class Person
3  {
4  public:
5      std::string m_strName;
6      int m_nAge;
7      bool m_bIsMale;
8
9      std::string GetName() { return m_strName; }
10     int GetAge() { return m_nAge; }
11     bool IsMale() { return m_bIsMale; }
12
13     Person(std::string strName = "", int nAge = 0, bool bIsMale = false)
14         : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
15     {
16     }
17 };

```

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:

```

1  class BaseballPlayer
2  {
3  public:
4      double m_dBattingAverage;
5      int m_nHomeRuns;
6  };

```

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

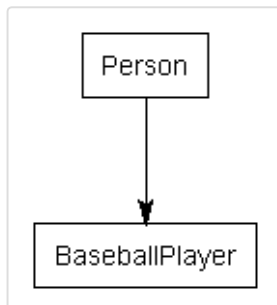
public inheritance means in a future section.

```

1  // BaseballPlayer publicly inheriting Person
2  class BaseballPlayer : public Person
3  {
4  public:
5      double m_dBattingAverage;
6      int m_nHomeRuns;
7
8      BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
9          : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
10     {
11     }
12 };

```

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:

```

1  #include <iostream>
2  int main()
3  {
4      // Create a new BaseballPlayer object
5      BaseballPlayer cJoe;
6      // Assign it a name (we can do this directly because m_strName is public)
7      cJoe.m_strName = "Joe";
8      // Print out the name
9      std::cout << cJoe.GetName() << std::endl;
10
11     return 0;
12 }

```

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:

```

1  // Employee publicly inherits from Person
2  class Employee: public Person
3  {
4  public:
5      std::string m_strEmployerName;

```

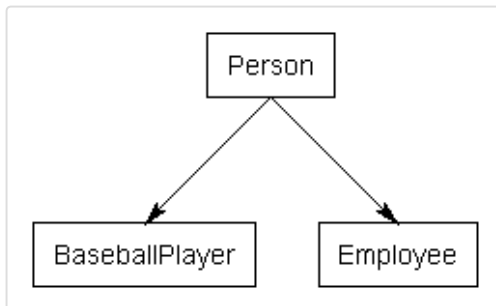
```

6      double m_dHourlySalary;
7      long m_lEmployeeID;
8
9      Employee(std::string strEmployerName, double dHourlySalary, long lEmployeeID)
10         : m_strEmployerName(strEmployerName), m_dHourlySalary(dHourlySalary),
11           m_lEmployeeID(lEmployeeID)
12     {
13     }
14
15     double GetHourlySalary() { return m_dHourlySalary; }
16     void PrintNameAndSalary()
17     {
18         std::cout << m_strName << ": " << m_dHourlySalary << std::endl;
19     }
20 };

```

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.

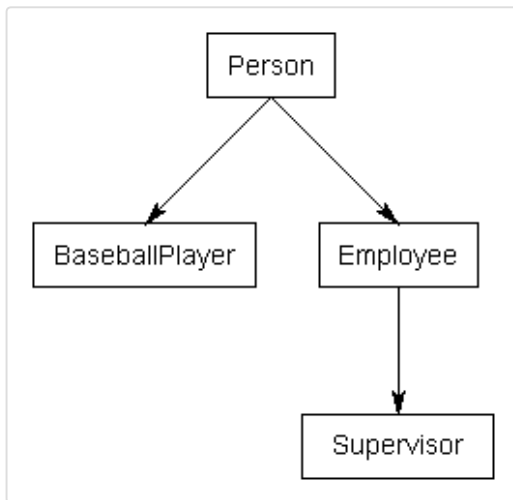
For example, let's write a `Supervisor` class. A supervisor is an employee, which is a person. We've already written an `Employee` class, so let's use that as the base class from which to derive `Supervisor`:

```

1  class Supervisor: public Employee
2  {
3  public:
4      // This Supervisor can oversee a max of 5 employees
5      long m_nOverseesIDs[5];
6  };

```

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

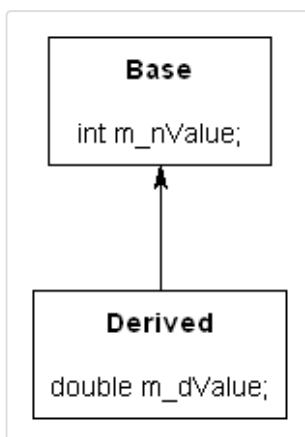
BY ALEX ON JANUARY 7TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 7TH, 2008

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:

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

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.

```
1  #include <iostream>
2  using namespace std;
3
4  class Base
5  {
6  public:
7      int m_nValue;
8
9      Base(int nValue=0)
10         : m_nValue(nValue)
11     {
12         cout << "Base" << endl;
13     }
14 };
15
16 class Derived: public Base
17 {
18 public:
19     double m_dValue;
20
21     Derived(double dValue=0.0)
22         : m_dValue(dValue)
23     {
24         cout << "Derived" << endl;
25     }
26 };
27
28 int main()
29 {
30     cout << "Instantiating Base" << endl;
31     Base cBase;
32
33     cout << "Instantiating Derived" << endl;
34     Derived cDerived;
35
36     return 0;
37 }
```

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:

```
1  class A
2  {
3  public:
4      A()
5      {
6          cout << "A" << endl;
7      }
8  };
9
10 class B: public A
11 {
12 public:
13     B()
14     {
15         cout << "B" << endl;
16     }
17 };
18
19 class C: public B
20 {
21 public:
22     C()
23     {
24         cout << "C" << endl;
25     }
26 };
27
28 class D: public C
29 {
30 public:
31     D()
32     {
33         cout << "D" << endl;
34     }
35 };
```

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.

```
1  int main()
2  {
3      cout << "Constructing A: " << endl;
4      A cA;
5
6      cout << "Constructing B: " << endl;
7      B cB;
8
9      cout << "Constructing C: " << endl;
10     C cC;
11
12     cout << "Constructing D: " << endl;
13     D cD;
14 }
```

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:



Email



Facebook 2



Twitter



Google



Pinterest



## 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:

```

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 };

```

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:

```

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

```

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:

```

1  int main()
2  {
3      Derived cDerived(1.3); // use Derived(double) constructor
4
5      return 0;
6  }

```

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:

```

1  class Derived: public Base
2  {
3  public:
4      double m_dValue;
5
6      Derived(double dValue=0.0, int nValue=0)
7          // does not work
8          : m_dValue(dValue), m_nValue(nValue)
9      {
10     }
11 };

```

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:

```

1  class Derived: public Base
2  {
3  public:
4      double m_dValue;
5
6      Derived(double dValue=0.0, int nValue=0)
7          : m_dValue(dValue)
8      {
9          m_nValue = nValue;
10     }
11 };

```

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:

```

1  class Derived: public Base
2  {
3  public:
4      double m_dValue;
5
6      Derived(double dValue=0.0, int nValue=0)
7          : Base(nValue), // Call Base(int) constructor with value nValue!
8            m_dValue(dValue)
9      {
10     }
11 };

```

Now, when we execute this code:

```

1  int main()
2  {
3      Derived cDerived(1.3, 5); // use Derived(double) constructor
4
5      return 0;
6  }

```

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 constructor 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:

```

1  #include <string>
2  class Person
3  {
4  public:
5      std::string m_strName;
6      int m_nAge;
7      bool m_bIsMale;
8
9      std::string GetName() { return m_strName; }

```

```

10     int GetAge() { return m_nAge; }
11     bool IsMale() { return m_bIsMale; }
12
13     Person(std::string strName = "", int nAge = 0, bool bIsMale = false)
14         : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
15     {
16     }
17 };
18
19 // BaseballPlayer publicly inheriting Person
20 class BaseballPlayer : public Person
21 {
22 public:
23     double m_dBattingAverage;
24     int m_nHomeRuns;
25
26     BaseballPlayer(double dBattingAverage = 0.0, int nHomeRuns = 0)
27         : m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
28     {
29     }
30 };

```

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.

```

1 // BaseballPlayer publicly inheriting Person
2 class BaseballPlayer : public Person
3 {
4 public:
5     double m_dBattingAverage;
6     int m_nHomeRuns;
7
8     BaseballPlayer(std::string strName = "", int nAge = 0, bool bIsMale = false,
9                     double dBattingAverage = 0.0, int nHomeRuns = 0)
10        : Person(strName, nAge, bIsMale), // call Person(std::string, int, bool) to initialize
11        these fields
12        m_dBattingAverage(dBattingAverage), m_nHomeRuns(nHomeRuns)
13    {
14    }
15 };

```

Now we can create baseball players like this:

```

1 int main()
2 {
3     BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.342, 42);
4
5     return 0;
6 }

```

To prove that it works:

```

1 int main()
2 {
3     BaseballPlayer cPlayer("Pedro Cerrano", 32, true, 0.342, 42);
4
5     using namespace std;
6     cout << cPlayer.m_strName << endl;
7     cout << cPlayer.m_nAge << endl;
8     cout << cPlayer.m_nHomeRuns;
9 }

```

```

10     return 0;
11 }

```

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.

```

1  #include <iostream>
2  using namespace std;
3
4  class A
5  {
6  public:
7      A(int nValue)
8      {
9          cout << "A: " << nValue << endl;
10     }
11 };
12
13 class B: public A
14 {
15 public:
16     B(int nValue, double dValue)
17     : A(nValue)
18     {
19         cout << "B: " << dValue << endl;
20     }
21 };
22
23 class C: public B
24 {
25 public:
26     C(int nValue, double dValue, char chValue)
27     : B(nValue, dValue)
28     {
29         cout << "C: " << chValue << endl;
30     }
31 };
32
33 int main()
34 {
35     C cClass(5, 4.3, 'R');
36
37     return 0;
38 }

```

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:



Email



Facebook 8



Twitter



Google



Pinterest

## 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!

```

1  class Base
2  {
3  private:
4      int m_nPrivate; // can only be accessed by Base member functions (not derived classes)
5  public:
6      int m_nPublic; // can be accessed by anybody
7  };

```

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.

```

1  class Base
2  {
3  public:
4      int m_nPublic; // can be accessed by anybody
5  private:
6      int m_nPrivate; // can only be accessed by Base member functions (but not derived classes)
7  protected:
8      int m_nProtected; // can be accessed by Base member functions, or derived classes.
9  };
10
11 class Derived: public Base
12 {
13 public:
14     Derived()
15     {
16         // Derived's access to Base members is not influenced by the type of inheritance used,
17         // so the following is always true:
18
19         m_nPublic = 1; // allowed: can access public base members from derived class
20         m_nPrivate = 2; // not allowed: can not access private base members from derived class
21         m_nProtected = 3; // allowed: can access protected base members from derived class
22     }
23 };
24
25 int main()
26 {
27     Base cBase;
28     cBase.m_nPublic = 1; // allowed: can access public members from outside class
29     cBase.m_nPrivate = 2; // not allowed: can not access private members from outside class
30     cBase.m_nProtected = 3; // not allowed: can not access protected members from outside clas
31 }

```

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:

```

1  // Inherit from Base publicly
2  class Pub: public Base
3  {
4  };
5
6  // Inherit from Base privately
7  class Pri: private Base
8  {
9  };
10
11 // Inherit from Base protectedly
12 class Pro: protected Base
13 {
14 };
15
16 class Def: Base // Defaults to private inheritance
17 {
18 };

```

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.

```

1  class Base
2  {
3  public:
4      int m_nPublic;
5  private:
6      int m_nPrivate;
7  protected:
8      int m_nProtected;
9  };
10
11 class Pub: public Base
12 {
13     // Public inheritance means:
14     // m_nPublic stays public
15     // m_nPrivate stays private
16     // m_nProtected stays protected
17
18     Pub()
19     {
20         // The derived class always uses the immediate parent's class access specifications
21         // Thus, Pub uses Base's access specifiers

```



```

22     m_nPublic = 1; // okay: anybody can access public members
23     m_nPrivate = 2; // not okay: derived classes can't access private members in the base
24 class!
25     m_nProtected = 3; // okay: derived classes can access protected members
26 }
27 };
28
29 int main()
30 {
31     // Outside access uses the access specifiers of the class being accessed.
32     // In this case, the access specifiers of cPub. Because Pub has inherited publicly from B
33     ase,
34     // no access specifiers have been changed.
35     Pub cPub;
36     cPub.m_nPublic = 1; // okay: anybody can access public members
37     cPub.m_nPrivate = 2; // not okay: can not access private members from outside class
38     cPub.m_nProtected = 3; // not okay: can not access protected members from outside class
39 }

```

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.

```

1  class Base
2  {
3  public:
4      int m_nPublic;
5  private:
6      int m_nPrivate;
7  protected:
8      int m_nProtected;
9  };
10
11 class Pri: private Base
12 {
13     // Private inheritance means:
14     // m_nPublic becomes private
15     // m_nPrivate stays private
16     // m_nProtected becomes private
17
18     Pri()
19     {

```

```

20 // The derived class always uses the immediate parent's class access specifications
21 // Thus, Pub uses Base's access specifiers
22 m_nPublic = 1; // okay: anybody can access public members
23 m_nPrivate = 2; // not okay: derived classes can't access private members in the base
24 class!
25     m_nProtected = 3; // okay: derived classes can access protected members
26 }
27 };
28
29 int main()
30 {
31     // Outside access uses the access specifiers of the class being accessed.
32     // Note that because Pri has inherited privately from Base,
33     // all members of Base have become private when access through Pri.
34     Pri cPri;
35     cPri.m_nPublic = 1; // not okay: m_nPublic is now a private member when accessed through P
36     ri
37     cPri.m_nPrivate = 2; // not okay: can not access private members from outside class
38     cPri.m_nProtected = 3; // not okay: m_nProtected is now a private member when accessed thr
39     ough Pri
40
41     // However, we can still access Base members as normal through Base:
42     Base cBase;
43     cBase.m_nPublic = 1; // okay, m_nPublic is public
44     cBase.m_nPrivate = 2; // not okay, m_nPrivate is private
45     cBase.m_nProtected = 3; // not okay, m_nProtected is protected
46 }

```

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:

```
1  class Base
2  {
3  public:
4      int m_nPublic;
5  private:
6      int m_nPrivate;
7  protected:
8      int m_nProtected;
9  };
```

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

```
1  class D2: private Base
2  {
3  public:
4      int m_nPublic2;
5  private:
6      int m_nPrivate2;
7  protected:
8      int m_nProtected2;
9  }
```

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.

```
1  class D3: public D2
2  {
3  public:
4      int m_nPublic3;
5  private:
6      int m_nPrivate3;
7  protected:
8      int m_nProtected3;
9  };
```

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:

```
1  #include <iostream>
2  using namespace std;
3
4  class Base
5  {
6  protected:
7      int m_nValue;
8
9  public:
10     Base(int nValue)
11         : m_nValue(nValue)
12     {
13     }
14
15     void Identify() { cout << "I am a Base" << endl; }
16 };
```

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.

```
1  class Derived: public Base
2  {
3  public:
4      Derived(int nValue)
5          :Base(nValue)
6      {
7      }
8  };
```

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:

```
1  class Derived: public Base
2  {
3  public:
```

```

4     Derived(int nValue)
5         :Base(nValue)
6     {
7     }
8
9     int GetValue() { return m_nValue; }
10 };

```

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

```

1  int main()
2  {
3      Derived cDerived(5);
4      cout << "cDerived has value " << cDerived.GetValue() << endl;
5
6      return 0;
7  }

```

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:

```

1  int main()
2  {
3      Base cBase(5);
4      cout << "cBase has value " << cBase.GetValue() << endl;
5
6      return 0;
7  }

```

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:

```

1  int main()
2  {
3      Base cBase(5);
4      cBase.Identify();
5
6      Derived cDerived(7);
7      cDerived.Identify()
8
9      return 0;
10 }

```

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.

```

1  class Derived: public Base
2  {
3  public:
4      Derived(int nValue)
5          :Base(nValue)
6      {
7      }
8
9      int GetValue() { return m_nValue; }
10
11     // Here's our modified function
12     void Identify() { cout << "I am a Derived" << endl; }
13 };

```

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

```

1  int main()
2  {
3      Base cBase(5);
4      cBase.Identify();
5
6      Derived cDerived(7);
7      cDerived.Identify()
8
9      return 0;
10 }

```

```

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 its own additional stuff.

```

1  class Derived: public Base
2  {
3  public:
4      Derived(int nValue)
5          :Base(nValue)
6      {

```

```

7     }
8
9     int GetValue() { return m_nValue; }
10
11    void Identify()
12    {
13        Base::Identify(); // call Base::Identify() first
14        cout << "I am a Derived"; // then identify ourselves
15    }
16 };

```

Now consider the following example:

```

1    int main()
2    {
3        Base cBase(5);
4        cBase.Identify();
5
6        Derived cDerived(7);
7        cDerived.Identify()
8
9        return 0;
10   }

```

```

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:

```

1    void Identify()
2    {
3        Identify(); // Note: no scope resolution!
4        cout << "I am a Derived"; // then identify ourselves
5    }

```

`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 its access changed in the derived class under the new access specifier.

For example, consider the following Base:

```

1    class Base
2    {
3    private:
4        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 its 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](#)

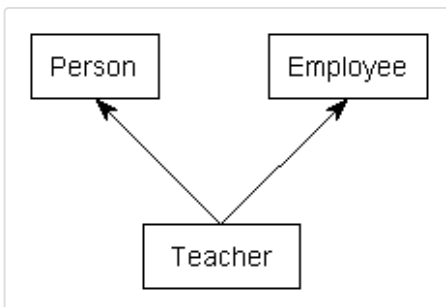
I think the following statement about the last example is confusing: "Note that this allowed us to take a poorly designed base class and encapsulate it's data in our derived class (alternatively, we could have

## 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.



```

1  #include <string>
2  class Person
3  {
4  private:
5      std::string m_strName;
6      int m_nAge;
7      bool m_bIsMale;
8
9  public:
10     Person(std::string strName, int nAge, bool bIsMale)
11         : m_strName(strName), m_nAge(nAge), m_bIsMale(bIsMale)
12     {
13     }
14
15     std::string GetName() { return m_strName; }
16     int GetAge() { return m_nAge; }
17     bool IsMale() { return m_bIsMale; }
18 };
19
20 class Employee
21 {
22 private:
23     std::string m_strEmployer;
24     double m_dWage;
25
26 public:
27     Employee(std::string strEmployer, double dWage)
28         : m_strEmployer(strEmployer), m_dWage(dWage)
29     {
30     }
31
32     std::string GetEmployer() { return m_strEmployer; }
33     double GetWage() { return m_dWage; }
34 };
35
36 // Teacher publicly inherits Person and Employee
37 class Teacher: public Person, public Employee
38 {

```

```

39 private:
40     int m_nTeachesGrade;
41
42 public:
43     Teacher(std::string strName, int nAge, bool bIsMale, std::string strEmployer, double dWage, int nTeachesGrade)
44         : Person(strName, nAge, bIsMale), Employee(strEmployer, dWage), m_nTeachesGrade(nTeachesGrade)
45     {
46     }
47 };

```

### 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:

```

1  class USBDevice
2  {
3  private:
4      long m_lID;
5
6  public:
7      USBDevice(long lID)
8          : m_lID(lID)
9      {
10     }
11
12     long GetID() { return m_lID; }
13 };
14
15 class NetworkDevice
16 {
17 private:
18     long m_lID;
19
20 public:
21     NetworkDevice(long lID)
22         : m_lID(lID)
23     {
24     }
25
26     long GetID() { return m_lID; }
27 };
28
29 class WirelessAdaptor: public USBDevice, public NetworkDevice
30 {
31 public:
32     WirelessAdaptor(long lUSBID, long lNetworkID)
33         : USBDevice(lUSBID), NetworkDevice(lNetworkID)
34     {
35     }
36 };
37
38 int main()
39 {
40     WirelessAdaptor c54G(5442, 181742);
41     cout << c54G.GetID(); // Which GetID() do we call?
42
43     return 0;
44 }

```

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:

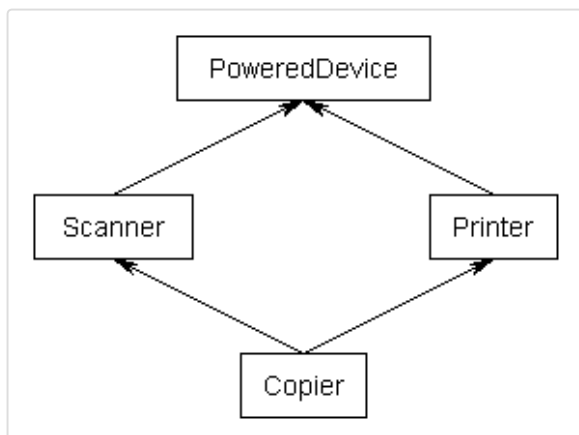
```
1  int main()
2  {
3      WirelessAdaptor c54G(5442, 181742);
4      cout << c54G.USBDevice::GetID();
5
6      return 0;
7  }
```

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:

```
1  class PoweredDevice
2  {
3  };
4
5  class Scanner: public PoweredDevice
6  {
7  };
8
9  class Printer: public PoweredDevice
10 {
11 };
12
13 class Copier: public Scanner, public Printer
14 {
15 };
```



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 inheritance 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.

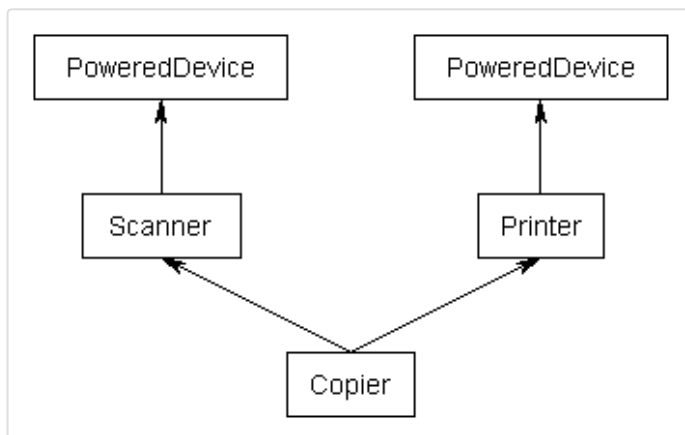
In the previous section on multiple inheritance, we left off talking about the “diamond problem”. In this section, we will resume this discussion.

### Virtual base classes

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

```
1  class PoweredDevice
2  {
3  public:
4      PoweredDevice(int nPower)
5      {
6          cout << "PoweredDevice: " << nPower << endl;
7      }
8  };
9
10 class Scanner: public PoweredDevice
11 {
12 public:
13     Scanner(int nScanner, int nPower)
14         : PoweredDevice(nPower)
15     {
16         cout << "Scanner: " << nScanner << endl;
17     }
18 };
19
20 class Printer: public PoweredDevice
21 {
22 public:
23     Printer(int nPrinter, int nPower)
24         : PoweredDevice(nPower)
25     {
26         cout << "Printer: " << nPrinter << endl;
27     }
28 };
29
30 class Copier: public Scanner, public Printer
31 {
32 public:
33     Copier(int nScanner, int nPrinter, int nPower)
34         : Scanner(nScanner, nPower), Printer(nPrinter, nPower)
35     {
36     }
37 };
```

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:

```

1  int main()
2  {
3      Copier cCopier(1, 2, 3);
4  }

```

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 the virtual keyword to create a shared base class:

```

1  class PoweredDevice
2  {
3  };
4
5  class Scanner: virtual public PoweredDevice
6  {
7  };
8
9  class Printer: virtual public PoweredDevice
10 {
11 };
12
13 class Copier: public Scanner, public Printer
14 {
15 };

```

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:

```

1  class PoweredDevice
2  {
3  public:
4      PoweredDevice(int nPower)

```

```

5      {
6          cout << "PoweredDevice: " << nPower << endl;
7      }
8  };
9
10     class Scanner: virtual public PoweredDevice
11     {
12     public:
13         Scanner(int nScanner, int nPower)
14             : PoweredDevice(nPower)
15         {
16             cout << "Scanner: " << nScanner << endl;
17         }
18     };
19
20     class Printer: virtual public PoweredDevice
21     {
22     public:
23         Printer(int nPrinter, int nPower)
24             : PoweredDevice(nPower)
25         {
26             cout << "Printer: " << nPrinter << endl;
27         }
28     };
29
30     class Copier: public Scanner, public Printer
31     {
32     public:
33         Copier(int nScanner, int nPrinter, int nPower)
34             : Scanner(nScanner, nPower), Printer(nPrinter, nPower), PoweredDevice(nPower)
35         {
36         }
37     };

```

This time, our previous example:

```

1  int main()
2  {
3      Copier cCopier(1, 2, 3);
4  }

```

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**

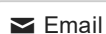


## **Index**



## **11.7 – Multiple inheritance**

### **Share this:**

[Email](#)[Facebook 5](#)[Twitter](#)[Google](#)[Pinterest](#)

 [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 virtual 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,