Hands On: C++ Programming

Contents

Chapter 1

On to C++

In this chapter we provide a stepping stone from C to C++. C++ supports all the features of C, with a few twists and a lot more features thrown in. This chapter starts with a comparison of C and C++, focusing on changes you'll need to make to compile your ANSI C code with an ANSI C++ compiler. It then moves on to some features unique to C++.

Think of C++ as a superset of C. For the most part, every single feature you've come to know and love in C is available in C++ (albeit with a few changes). As in C, C++ programs start with a `main()` function. All of C's keywords and functions work just fine in C++. Even ommand-line arguments `argc` and `argv`, are still supported in C++. In fact, with only a few tweaks here and there, your C programs should run quite well in the C++ world.

1.1 Function Prototypes Are Required

In C, function prototypes are optional, or at least C assumes that they defer to type `int` if they are not declared. As long as there's no type conflict between a function call and the same function's declaration, your program will compile.

In C++, a function prototype *is required* for each of your program's functions. Your C++ program *will not* compile unless each and every function prototype is in place. As in C, you can declare a function without a return type. If no return type is present, the function is assumed to have a return type of `int`.

1.2 Getting C Code to Run under C++

1.2.1 Automatic Type Conversion

C++ uses the same rules as C for automatic type conversion, but with a slight twist.

Although a `void` pointer can be assigned the value of another pointer type without explicit typecasting, the reverse is not true.

For example, although the following code compiles properly in C, it will not compile in C++:

```
void *voidPtr;
short *shortPtr;
voidPtr = shortPtr; /*  This line is just fine... */
shortPtr = voidPtr; /*  This line is fine in C,
                     but WILL NOT compile in C++ */

shortPtr = (short *)voidPtr; /*  This works in C++ */
```

1.2.2 Scope Issues

There are several subtle differences between C and C++ involving scope. A variable's scope defines the availability of the variable throughout the rest of a program. For example, a global variable is available throughout a program, while a local variable is limited to the block in which it is declared. Though C++ follows the same scope rules as C, there are a few subtleties you should be aware of. For example, take a look at the following code. Try to guess the value of size at the bottom of `main()`:

```
char dummy[32];

int main()
{
  long size;
  struct dummy
    {
      char myArray[ 64 ];
```

```
    };

  size = sizeof( dummy );
  return 0;
}
```

In C, `size` ends up with a value of 32; the reference to dummy in the `sizeof()` statement matches the global variable declared at the top of the program.

In C++, however, `size` ends up with a value of 64; the reference to `dummy` matches the `struct` tag inside `main()`.

In C++, a structure name declared in an inner scope can *hide* a name in an outer scope. This same rule holds true for an enumeration:

```
enum colour  red, green, blue ;
```

In C++, this `enum` creates a type named `colour` that can be used to declare other `enum`s and would obscure a global with the same name.

Here's another example:

```
int main()
{
   struct s
    {
     enum { good, bad, ugly } clint;
    };
   short good;
   return 0;
}
```

An ANSI C compiler will not compile this code, complaining that the identifier good was declared twice. The problem here is with the scope of the enumeration constant good. In C, an enumeration constant is granted the same scope as a local variable, even if it is embedded in a struct definition. When the compiler hits the short declaration, it complains that it already has a good identi-fier declared at that level.

In C++, this code compiles cleanly. Why? C++ enumeration constants embedded in a struct definition have the same scope as that struct's fields. Thus, the enumeration constant good is hidden from the short declaration at the bottom of `main()`. A third example involves multiple declarations of the same variable within the same scope. Consider the following code:

```
short gMyGlobal;
short gMyGlobal; /* Cool in C, error in C++ */
```

The C compiler will resolve these two variable declarations to a single declaration. The C++ compiler, on the other hand, will report an error if it hits two variable declarations with the same name.

It's useful to be aware of the difference between a declaration and a definition. A declaration specifies the types of all elements of an identifier. For example, a function prototype is a declaration. Here are some more declarations:

```
char name[ 20 ];
typedef int myType;
const short kMaxNameLength = 20;
extern char aLetter;
short MyFunc( short myParam );
```

As you can see, a declaration can do more than tie a type to an identifier. A declaration can also be a defini-tion. A definition instantiates an identifier, allocating the appropriate amount of memory. In this declaration:

```
const short kMaxNameLength = 20;
```

the constant `kMaxNameLength` is also defined and initialized.

1.3 New Features of C++

The remainder of this chapter will take you beyond C into the heart of C++. While we won't explore object programming in this chapter, we will cover just about every other C++ concept.

1.3.1 Comment block markers

C's comment block markers, `/*` and `*/`, perform the same func-tion in C++. In addition, C++ supports a single-line comment marker. When a C++ compiler encounters the characters `//`, it ignores the remainder of that line of code. Here's an example:

```
int main()
{
short numGuppies;  // May increase suddenly!!
return 0;
}
```

As you'd expect, the characters `//` are ignored inside a comment block. In the following example, `//` is included as part of the comment block:

```
int main()
{
  /* Just a comment... // */
return 0;
}
```

Conversely, the comment characters `/*` and `*/` have no special meaning inside a single-line comment. The start of the comment block in the following example is swallowed up by the single-line comment:

```
int main()
{
// Don't start a /* comment block
inside a single-line comment...
This code WILL NOT compile!!! */

return 0;
}
```

The compiler will definitely complain about this example!

1.3.2 The `iostream`

In a standard C program, input and output are usually handled by Standard Library routines such as `scanf()` and `printf()`. While you can call `scanf()` and `printf()` from within your C++ program, there is an elegant alternative. The `iostream` facility allows you to send a sequence of variables and constants to an output stream, just as `printf()` does. Also,

`iostream` makes it easy to convert data from an input stream into a sequence of variables, just as scanf() does.

Though the `iostream` features presented in this section may seem simplistic, don't be fooled. `iostream` is actually quite sophisticated. In fact, `iostream` is far more powerful than C's standard I/O facility. The material given here will allow you to perform the input and output you'll need to get through the next few chapters. Later, we'll explore `iostream` in more depth.

The `iostream` predefines three streams for input and output. `cin` is used for input, `cout` for normal output, and `cerr` for error output. The << operator is used to send data to a stream. The >> operator is used to retrieve data from a stream. The << operator is known as the insertion operator because it allows you to insert data into a stream. The >> operator is known as the extraction operator be-cause it allows you to extract data from a stream.

Here's an example of the << operator:

```
#include <iostream.h>
int main()
{
  cout << "Hello, world!";
  return 0;
}
```

This program sends the text string `Hello, world!` to the console, just as if you'd used `printf()`.

The include file <iostream.h> contains all of the definitions needed to use `iostream`. Since << is a binary operator, it requires two operands. In this case, the operands are `cout` and the string `Hello, world!` .

The destination stream always appears on the left side of the << operator. Just like the `&` and `*` operators, >> and << have more than one meaning (>> and << are also used as the right and left shift operators). Don't worry about confusion, however. The C++ compiler uses the operator's context to determine which meaning is appropriate.

As with any other operator, you can use more than one << on a single line. Here's another example:

```
#include <iostream.h>
int main()
```

```
{
short i = 20;
cout << "The value of i is " << i;
return 0;
}
```

This program produces the following output:

```
The value of i is 20
```

`iostream` knows all about C++'s built-in data types. This means that text strings are printed as text strings, shorts as shorts, and floats as floats, complete with decimal point. No special formatting is necessary.

An `iostream` Output Example

```
#include <iostream.h>
int main()
{
char *name = "Dr. Marshall";
cout << "char: " << name[ 0 ] << '\n'
<< "short: " << (short)(name[ 0 ]) << '\n'
<< "string: " << name << '\n'
<< "address: " << (unsigned long)name;
return 0;
}
```

The cout Source Code The program starts by initializing the char pointer name, point-ing it to the text string "Dr. Marshall". Next comes one giant statement featuring eleven different occurrences of the << opera-tor. This statement produces four lines of output.

The following line of code

```
cout << "char: " << name[ 0 ] << '\n'
```

produces this line of output:

```
char: D
```

As you'd expect, printing `name[ 0 ]` produces the first character in name, an uppercase D.

The next line of code is

```
<< "short: " << (short)(name[ 0 ]) << '\n'
```

The output associated with this line of code is as follows:

```
short: 68
```

This result was achieved by casting the character 'D' to a short. In general, `iostream` displays integral types (such as short and int) as an integer. As you'd expect, a float is displayed in floating-point format.

The next line of code

```
<< "string: " << name << '\n'
```

produces this line of output:

```
string: Dr. Marshall
```

When the << operator encounters a `char` pointer, it assumes you want to print a zero-terminated string.

The final chunk of code in our example shows another way to display the contents of a pointer:

```
<< "address: " << (unsigned long)name;
```

Again, name is printed, but this time is cast as an unsigned long. Here's the result:

```
address: 2150000
```

As you can see, `cout` does what it thinks makes sense for each type it prints. Later in Chapter 5, we will see how to customize cout by using it to print data in a specified format or teaching it how to print your own data types.

Let us now consider an `iostream` input example

```
#include <iostream.h>
const short kMaxNameLength = 40;
int main()
{
   char name[ kMaxNameLength ];
```

```
   short myShort;
   long myLong;
   float myFloat;

   cout << "Type in your first name: ";
   cin >> name;

   cout << "Short, long, float: ";
   cin >> myShort >> myLong >> myFloat;

   cout << "\nYour name is: " << name;
   cout << "\nmyShort: " << myShort;
   cout << "\nmyLong: " << myLong;
   cout << "\nmyFloat: " << myFloat;
   return 0;
}
```

As is always the case when you use `iostream`, the program starts by including the file <`iostream.h`>. Next, the constant `kMaxNameLength` is defined, providing a length for the char array name.

When a variable is defined using the const qualifier, an initial value must be provided in the definition, and that value cannot be changed for the duration of the program. Although some C programmers tend to use `#define` instead of const, C++ programmers prefer const to `#define`.

`cin` uses `cout` and << to prompt for a text string, a short, a long, and a float. `cin` and >> are used to read the values into the four variables `name, myShort, myLong`, and `myFloat`.

The next line uses ¡¡ to send a text string to the console:

```
cout << "Type in your first name: ";
```

Next, ¿¿ is used to read in a text string:

```
cin >> name;
```

When the ¿¿ operator reads a text string, it reads a character at a time until a white space character (like a space or a tab) is encountered.

Next, three more pieces of data are read using a *single* statement. First, display the prompt:

```
cout << "Short, long, float: ";
```

Then, read in the data, separating the three receiving variables by consecutive >> operators:

```
cin >> myShort >> myLong >> myFloat;
```

Be sure to separate each of the three numbers by a space (or some white space character). Also, make sure the numbers match the type of the corresponding variable. For example, it's probably not a good idea to enter 3.52 or 125000 as a short, although an integer like 47 works fine as a float.

Finally, display each of the variables we worked so hard to fill with `cout` calls.

1.3.3 `iostream` and Objects

So far, `iostream` might seem primitive compared to the routines in C's Standard Library. After all, routines like `scanf()` and `printf()` give you precise control over your input and output. Routines like `getchar()` and `putchar()` allow you to process one character at a time, letting you decide how to handle white space.

The `iostream` is very powerful. However, to unleash `iostream`'s true power, you must first come up to speed on object oriented programming. The `iostream` concepts presented here are the bare minimum you'll need to get through the sample programs in the next few chapters. For now, basic input and output are all we need.

1.3.4 Default Argument Initializers

C++ allows you to assign default values (known as *default argument initializers*) to a function's arguments. For example, here's a simple default routine:

```
void Deffun( short default = 40 )
{
// A default of 440 is  assigned if no other value passed in
}
```

If you call this function with a parameter, the value you pass in is used. For example, the call Deffun(30);

will assign a value of 30 to `default`,

If you call the function without specifying a value, the default value is used.

The call `Deffun();`

will assign a value of 40 to `default`,

This technique works with multiple parameters as well, although the rules get a bit more complicated. You can specify a default value for a parameter only if you also specify a default for all the parameters that follow it. For example, this declaration is cool:

```
void GotSomeDefaults( short manny, short moe=2,
char jack='x' );
```

Since the second parameter has a default, the third parameter must have a default.

The next declaration won't compile, however, because the first parameter specifies a default and the parameter that follows does not:

```
void WillNotCompile( long time=100L, short stack );
```

Default parameter values are specified in the function prototype rather than in the function's implementation. For example, here's a function prototype, followed by the function itself:

```
void MyFunc( short param1 = 27 );

void MyFunc( short param1 )
{
// Body of the function...
}
```

1.3.5 Reference Variables

In C, all parameters are passed by value as opposed to being passed by reference. When you pass a parameter to a C function, the value of the parameter is passed on to the function. Any changes you make to this value are not carried back to the calling function.

Here's an example:

```
void DoubleMyValue( short valueParam )
  {
     valueParam *= 2;
   }

int main()
  {
     short number = 10;
     DoubleMyValue( number );
     return 0;
  }
```

`main()` sets `number` to 10, then passes it to the function `DoubleMyValue()`. Since number is passed by value, the call to `DoubleMyValue()` has no effect on number. When `DoubleMyValue()` returns, `number` still has a value of 10.

Here's an updated version of the program:

```
void DoubleMyValue( short *numberPtr )
   {
     *numberPtr *= 2;
    }

int main()
   {
     short number = 10;
     DoubleMyValue( &number );
     return 0;
}
```

In this version, number's `address` is passed to `DoubleMyValue()`. By dereferencing this *pointer*,`DoubleMyValue()` can reach out and change the value of number. When `DoubleMyValue()` returns, number will have a value of 20.

In C++ Reference variables, howvere, allow you to pass a parameter by reference, without using pointers. Here's another version of the program, this time implemented with a reference variable:

```
void DoubleMyValue( short &referenceParam )
   {
     referenceParam *= 2;
   }


 int main()
  {
     short number = 10;
     DoubleMyValue( number );
     return 0;
  }
```

Notice that this code looks just like the first version, with one small exception: `DoubleMyValue()`'s parameter is defined using the `&` operator:

```
short &referenceParam
\begin{verbatim}

The {\tt \&} marks {\tt referenceParam} as a reference variable and tells
the compiler that {\tt referenceParam} and its corresponding input
parameter, number, are one and the same. Since both names refer
to the same location in memory, changing the value of
{\tt referenceParam} is exactly the same as changing number.


Here is an example call by reference program:

\begin{verbatim}
#include <iostream.h>

void CallByValue( short valueParam );
void CallByReference( short &refParam );

int main()
{
short number = 12;
long longNumber = 12L;
```

```
cout << "&number:     " <<
(unsigned long)&number << "\n";

cout << "&longNumber: " <<
(unsigned long)&longNumber << "\n\n";

CallByValue( number );
cout << "After ByValue: " << number << "\n\n";

CallByReference( number );
cout << "After ByRef( short ): " << number << "\n\n";

CallByReference( longNumber );
cout << "After ByRef( long ): " << longNumber << "\n";

return 0;
}

void CallByValue( short valueParam )
{
cout << "&valueParam: " <<
 (unsigned long)&valueParam << "\n";
valueParam *= 2;
}

void CallByReference( short &refParam )
{
cout << "&refParam:   " <<
 (unsigned long)&refParam << "\n";
refParam *= 2;
}
```

Reference variables are frequently used as *call-by-reference* parameters. However, they can also be used to establish a link between two variables in the same scope. Here's an example:

```
short romulus;
short &remus = romulus;
```

The first line of code defines a `short` with the name `romulus`. The second line of code declares a reference variable with the name `remus`, linking it to the variable `romulus`. Just as before, the `&` marks remus as a reference variable.

Now that `remus` and `romulus` are linked, they share the same location in memory. Changing the value of one is exactly the same as changing the value of the other.

It's important to note that a reference variable **must** be initialized with a variable as soon as it is declared.

The following code will not compile:

```
short romulus;
short &remus; // Will not compile!!!
remus = romulus;
```

The reference variable **must** also be of the same type as the variable it references. The following code won't work:

```
short romulus;
long &remus = romulus; // Type mismatch!!!
```

In addition, once established, the link between a reference and a regular variable cannot be changed as long as the reference remains in scope. In other words, once `remus` is linked to `romulus`, it *cannot* be set to reference a different variable.

1.3.6 Function Name Overloading

Function name overloading, allows you to write several functions that share the same name. Suppose you needed a function that would print the value of one of your variables, be it long, short, or a text string. You could write one function that takes four parameters:

```
Display( short whichType,
         long longParam,
         short shortParam,
         char *textParam );
```

The first parameter might act like a switch, determining which of the three types you were passing in for printing. The main code of the function might look like this:

```
if ( whichType == kIsLong )
   cout << "The long is: " << longParam << "\n";
 else if ( whichType == kIsShort )
     cout << "The short is: " << shortParam << "\n";
    else if ( whichType == kIsText )
       cout << "The text is: " << text << "\n";
```

Another solution is to write three separate functions, one for printing shorts, one for longs, and one for text strings:

```
void DisplayLong( long longParam );
void DisplayShort( short shortParam );
void DisplayText( char *text );
```

Each of these solutions has an advantage. The first solution groups all printing under a single umbrella, making the code somewhat easier to maintain. On the other hand, the second solution is more modular than the first. If you want to change the method you use to display longs, you modify only the routine that works with longs; you don't have to deal with the logic that displays other types.

As you might expect, there is a third solution that combines the benefits of the first two. Here's how it works.

As mentioned earlier, C++ allows several functions to share the same name by way of function name overloading. When an overloaded function is called, the compiler compares the parameters in the call with the parameter lists in each of the candidate functions. The candidate with the most closely matching parameter list is the one that gets called.

A function's parameter list is also known as its signature. A function's name and signature combine to distinguish it from all other functions. Note that a function's return type is not part of its signature.

Here's the third solution that takes advantage of function name overloading:

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

```
void Display( short shortParam );
void Display( long longParam );
void Display( char *text );

int main()
{
short myShort = 3;
long myLong = 12345678L;
char *text = "Make it so...";

Display( myShort );
Display( myLong );
Display( text );

return 0;
}

void Display( short shortParam )
{
cout << "The short is: " << shortParam << "\n";
}

void Display( long longParam )
{
cout << "The long is:  " << longParam << "\n";
}

void Display( char *text )
{
cout << "The text is:  " << text << "\n";
}
```

The output of this program is:

```
The short is: 3
The long is: 12345678
The text is: Make it so...
```

The program starts with three function prototypes, each of which shares the name `Display()` Notice that each version of Display() has a unique signature. This is important. You are **not allowed** to define two functions with the same name and the same signature.

main() starts by defining three variables: `myShort, myLong` and a `text` string.

Next, `Display()` is called three times.

- First, a `myShort` is passed as a parameter. Since this call exactly matches one of the `Display()` routines, the compiler doesn't have a problem deciding which function to call.
- Similarly, the calls passing `myLong` and a `text` string to `Display()` match perfectly with the `Display()`functions having `long` and `text` string signatures, repectively.

Matching Rules for Overloaded Functions

The above example was fairly straightforward. The compiler had no difficulty deciding which version of `Display()` to call because each of the calls matched perfectly with one of the `Display()` functions.

But what do you think would happen if you passed a float to `Display()`?

```
Display( 1.0 );
```

When the compiler can't find an exact match for an over-loaded function call, it turns to a set of rules that determine the best match for this call. After applying each of the rules, unless one and only one match is found, the compiler reports an error.

As you've already seen, the compiler starts the matching process by looking for an exact match between the name and signature of the function call and a declared function. If a match is not found, the compiler starts promoting the type of any integral parameters in the function call, following the rules for automatic type conversion similar to C. For example, a `char` or a `short` would be promoted to an `int` and a `float` would be promoted to a double.

If a match is still not found, the compiler starts promoting non-integral types. Finally, the ellipsis operator in a called function is taken into account, matching against zero or more parameters. In answer to our earlier question,

passing a float to `Display()` would result in an error, listing the function call as ambiguous. If we had written a version of `Display()` with a `float` or a `double` in its signature, the compiler would find the match.

1.3.7 The `new` and `delete` Operators

In C, memory allocation typically involves a call to `malloc()` paired with a call to free() when the memory is no longer needed. In C++, the same functionality is provided by the operators `new` and `delete`.

Call `new` when you want to allocate a block of memory.

For example, the following code allocates a block of `1024 chars`:

```
char *buffer;
buffer = new char[1024];
```

`new` takes a type as an operand, allocates a block of memory the same size as the type, and returns a pointer to the block.

To return the memory to the heap, use the `delete` operator. The next code frees up the memory just allocated:

```
delete [] buffer;
```

The brackets (`[]`) in the preceding line of code indicate that the item to be deleted is a pointer to an array. If you are deleting something other than a pointer to an array, leave the brackets out, for example:

```
int myIntPtr;
myIntPtr = new int;
delete myIntPtr;
```

`new` can be used with any legal C++ type, including those you create yourself.

Every program that allocates memory runs the risk that its request for memory will fail, most likely because there's no more memory left to allocate. If your program uses new to allocate memory, it had better detect, and handle, any failure on new's part.

Since `new` returns a value of 0 when it fails, the simplest approach just checks this return value, taking the appropriate action when new fails:

```
char *bufPtr;
bufPtr = new char[ 1024 ];

if ( bufPtr == 0 )
   cout << "Not enough memory!!!";
else
   DoSomething( bufPtr );
```

This code uses `new` to allocate a 1024-byte buffer. If new fails, an error message is printed; otherwise, the program goes on its merry way.

This approach requires that you check the return value every time you call new. If your program performs a lot of memory allocation, this memory-checking code can really add up. As your programs get larger and more sophisticated, you might want to consider a second strategy.

C++ allows you to specify a single routine, known as a new handler, that gets called if and when `new` fails. Design your new handler for the general case so that it can respond to any failed attempt to allocate memory.

Whether or not you designate a new handler, `new` will still return 0 if it fails. This means you can design a two-tiered memory management strategy combining a new handler and code that runs if `new` returns 0.

To specify a new handler, pass the handler's name to the function `set_new_handler()`. To use `set_new_handler()`, be sure to include the file <new.h>:

```
#include <new.h>

void NewFailed( void );

int main()
{
set_new_handler( NewFailed );
.
.
.
}
```

One possible memory allocation strategy is to allocate a block of memory at the beginning of your program, storing a pointer to the block in a global variable. Then, when new fails, your program can free up the spare memory

block, ensuring that it will have enough memory to perform any housekeeping chores that it requires in a memory emergency. A new Example Our next sample program repeatedly calls new until the program runs out of memory, keeps track of the number of memory re-quests, and then reports on the amount of memory allocated before failure. This program uses the spare memory scheme just described.

The code listing is as follows:

```
#include <iostream.h>
#include <new.h>

void NewFailed();

char gDone = false;
char *gSpareBlockPtr = 0;

int main()
{
char *myPtr;
long numBlocks = 0;

cout << "Installing NewHandler...\n";

set_new_handler( NewFailed );
gSpareBlockPtr = new char[20480];

while ( gDone == false )
{
myPtr = new char[1024];
numBlocks++;
}

cout << "Number of blocks allocated: " << numBlocks;

return 0;
}

void NewFailed()
```

```
{
if ( gSpareBlockPtr != 0 )
{
delete gSpareBlockPtr;
gSpareBlockPtr = 0;
}

gDone = true;
}
```

The number of blocks you can allocate before you run out of memory depends on the amount of memory you make available to your program.

`NewFailed()` is the function we want called if new fails in its attempt to allocate memory:

1.3.8 The Scope Resolution Operator, ::

C++'s scope resolution operator is denoted by `::`. The scope resolution operator precedes a variable, telling the compiler to look outside the current block for a variable of the same name.

Suppose you declare a global variable and a local variable with the same name:

```
short number;

int main()
{
   short number;
   number = 5; // local reference
   ::number = 10; // global reference
   return 0;
}
```

Inside `main()`, the first assignment statement refers to the local definition of `number`. The second assignment statement uses the scope operator to refer to the global definition of `number`. This code leaves the *local* `number` with a value of 5 and the *global* `number` with a value of 10.

Many programmers start all their global variables with a lowercase g to differentiate them from local variables. If you use this convention, you'll never be in the situation where a local variable is obscuring a global variable. This doesn't mean you should ignore the scope resolution operator As we get into object programming, we'll find that the scope resolution operator is invaluable.

A Scope Resolution Operator Example

Our next sample program offers a quick demonstration of the scope resolution operator.

```
#include <iostream.h>

short myValue = 5;

int main()
{
short yourValue = myValue;

cout << "yourValue: " << yourValue << "\n";

short myValue = 10;
yourValue = myValue;

cout << "yourValue: " << yourValue << "\n";

yourValue = ::myValue;
cout << "yourValue: " << yourValue << "\n";

return 0;
}
```

The output of this program is:

```
yourValue: 5
yourValue: 10
yourValue: 5
```

First we define a global variable with the name `myValue`, initializing it to a value of 5: The we define a local variable named `yourValue` and assigns it the value in `myValue`.

Then, yourValue is printed out, showing it with a value of 5, the same as the global myValue:

Next, a local with the name `myValue` is defined and initialized with a value of 10. When `myValue` is copied to `yourValue` which variable is copied, the local or the global?

As you can see from the output, the reference to `myValue` matches with the local declaration, showing yourValue with a value of 10:

Then, the scope resolution operator is used to copy `myValue` to yourValue. When yourValue is printed again, it has a value of 5, showing that `::myValue` refers to the global declaration of

The scope resolution operator can be applied only when a match is available. Applying the scope resolution operator to a local variable without a corresponding global will generate a compile error.

1.3.9 Inline Functions

Traditionally, when a function is called, the CPU executes a set of instructions that move control from the calling function to the called function. Tiny as these instructions may be, they still take time. C++, however, provides *inline functions*, which allow you to bypass these instructions and save a bit of execution time.

When you declare a function using the *inline* keyword, the compiler copies the body of the function into the calling function, making the copied instructions a part of the calling function as if it were written that way originally. The benefit to you is a slight improvement in performance. The cost is in memory usage:

- If you call an inline function twenty times from within your program, twenty copies of the function will be grafted into your object code.

An Inline Function Example

In order to understabd inline function consider the following example program. The program features a single inline function that returns the value achieved when its first argument is raised to its second argument's power.

The listing for `inline.cpp` is as follows:

```
#include <iostream.h>

inline long power( short base, short exponent );

int main()
{
cout << "power( 2, 3 ): " <<
power( 2, 3 ) << "\n";

cout << "power( 3, 6 ): " <<
power( 3, 6 ) << "\n";

cout << "power( 5, 0 ): " <<
power( 2, 0 ) << "\n";

cout << "power( -3, 4 ): " <<
power( -3, 4 ) << "\n";

return 0;
}

inline long power( short base, short exponent )
{
long product = 1;
short i;

if ( exponent < 0 )
return( 0 );

for ( i=1; i<=exponent; i++ )
product *= base;

return product;
}
```

The output is as foillows:

```
power( 2, 3 ): 8
```

```
power( 3, 6 ): 729
power( 5, 0 ): 1
power( -3, 4 ): 81
```

`inline.cpp` starts with the standard include file, followed by a function prototype that features the keyword `inline`: `inline long power( short base, short exponent )`.

`main()` calls `power()` four times and prints the result of each call.

By preceding `power()`'s declaration by the `inline` keyword, we've asked the compiler to replace each of the four function calls in `main()` with the code in `power()`. Note that this replacement effects the object code and has no impact on the source code:

There are two clear benefits that arise from using `inline` code instead of a `#define macro`

- *type-safety*
- *side-effects protection*

Consider this `#define` macro:

```
#define square(a) ( (a) * (a) )
```

Compare that macro to this inline function:

```
inline int square( int a )
{
return( a * a );
}
```

The inline version restricts its parameter to an integral value while the `#define` performs a simple-minded text substitution.

Now suppose you call `square()` with a prefix operator:

```
xSquared = square( ++x );
```

The `#define` version expands this as follows:

```
xSquared = ( (++x) * (++x) );
```

which has the unwanted side-effect of incrementing `x` twice. The `inline` version doesn't do this. The upshot here is that both `#defines` and `inlines` offer an inline performance advantage, but the `inline` does its job a little more carefully.

Chapter 2

Object Oriented Programming

This chapter introduces concepts of object oriented programming and details how C++ implements objects we see how C++ may be used to create, destroy, and manipulate objects in very powerful ways.

First let's discuss the concept of an object.

2.1 Objects

There is nothing mysterious about the concept of an object. In C++, an object is any instance of a data type. For example, this line of code:

```
int myInt;
```

declares an int object.

The first real object we'll take a look at is the `struct` structure: One of the most valuable features shared by C and C++ is the structure. Without the structure, you'd have no way to group data that belonged together. For example, suppose you wanted to implement an employee data base that tracked an employee's name, employee ID, and salary. You might design a structure that looks like this:

```
const short kMaxNameSize = 20;

struct Employee
 {
   char name[kMaxNameSize];
```

```
   long id;
   float salary;
};
```

The great advantage of this structure is that it lets you bundle several pieces of information together under a single name. This concept is known as encapsulation.

For example, if you wrote a routine to print an employee's data, you could write:

```
Employee newHire;

......

PrintEmployee(newHire.name, newHire.id, newHire.salary);
```

Did you notice anything unusual about the declaration of `newHire` in the preceding code sample?

- In C, this code would not have compiled. Instead, the declaration would have looked like this:

  ```
  struct Employee newHire; /* The C version */
  ```

- When the C++ compiler sees a structure declaration, it uses the structure name to create a new data type, making it available for future structure declarations.

On the other hand, it would be so much more convenient to pass the data in its encapsulated form:

```
PrintEmployee( &newHire );
```

Encapsulation allows you to represent complex information in a more natural, easily accessible form. In the C language, the `struct` is the most sophisticated encapsulation mechanism available.

C++ takes encapsulation to a even higher level: Whilst C structures are limited strictly to data, C++ supports structures composed of both data and functions.

Here's an example of a C++ structure declaration:

```
const short kMaxNameSize = 20;
struct Employee
  {
    // Data members...
    char employeeName[ kMaxNameSize ];
    long employeeID;
    float employeeSalary;

    // Member functions...
    void PrintEmployee();
   };
```

This example declares a new type named `Employee`. You can use the `Employee` type to declare individual `Employee` objects:

- Each `Employee` object is said to be a member of the `Employee` *class*.
- The Employee `class` consists of *three data fields* as well as a *function* named `PrintEmployee()`.
  - In C++, a class's data fields are known as *data members* and its functions are known as *member* functions.
- Each `Employee` object you create gets its own copy of the `Employee` class data members.
- All `Employee` objects share a single set of Employee member functions.

You can also create classes, using the exact same syntax, substituting the keyword class for struct.It is more commom to declare the stucture as a `class`, to keep Object Oriented principles in tact:

```
const short kMaxNameSize = 20;

class Employee
  {
    // Data members...
    char employeeName[ kMaxNameSize ];
    long employeeID;
    float employeeSalary;
```

```
 // Member functions...
 void PrintEmployee();
};
```

The `class` definition is slightly different in how data members and member functions can be accessed which we will address later (See Access Privilege, Section 2.6).

The only difference is, the members of a struct are all `public` by default and the members of a class are all `private` by default.

Why use class instead of struct?

If you start with a struct, you give the world complete access to your class members unless you intentionally limit access using the appropriate access specifiers. If you start with a class, access to your class members is limited right from the start. You have to intentionally allow access by using the appropriate access specifiers.

We will soon be using the `class` for the remainder of this course, only in the next few examples will we use `struct` we'll use the class keyword to declare our classes.

2.2 Encapsulating Data and Functions

Later in this chapter, we'll see how to access an object's data members and member functions. For now, let's take a look at the mechanisms C++ provides to create and destroy objects.

There are two ways to create a new object. The simplest method is to define the object directly, just as you would a local variable:

```
Employee employee1;
```

This definition creates an `Employee` object whose name is `employee1`. `employee1` consists of a block of memory large enough to accommodate each of the three `Employee` data members. When you create an object by defining it directly, as we did above, memory for the object is allocated when the definition moves into scope. That same memory is freed up when the object drops out of scope.

For example, you might define an object at the beginning of a function:

```
void CreateEmployee()
{
Employee employee1;

....
}
```

When the function is called, memory for the object is allocated, right along with the function's other local objects. When the function exits, the object's memory is deallocated. When the memory for an object is deallocated, the object is said to be destroyed.

2.3 Creating an Object

If you want a little more control over when your object is destroyed, use `new` operator:

- First, define an object pointer,
- then call `new` to allocate the memory for your object. `new` returns a pointer to the newly created object.

An example that creates an tt Employee object this way follows:

```
Employee *employeePtr;

employeePtr = new Employee;
```

The first line of code defines a pointer designed to point to an `Employee` object. The second line uses `new` to create an `Employee` object. `new` returns a pointer to the newly created `Employee`.

Once you've created an object, you can modify its data members and call its member functions.

- If you've defined the object directly, you'll refer to its data members using the . operator:

  ```
  Employee employee1;
  employee1.employeeSalary = 200.0;
  ```

- If you're referencing the object through a pointer, use the $->$ operator:

```
Employee *employeePtr;
employeePtr = new Employee;
employeePtr->employeeSalary = 200.0;
```

- To call a member function, use the same technique. If the object was defined directly, you'll use the . operator:

```
Employee employee1;
employee1.PrintEmployee();
```

- If you're referencing the object through a pointer, you'll use the $->$ operator:

```
Employee *employeePtr;
employeePtr = new Employee;
employeePtr->PrintEmployee();
```

2.3.1 The Current Object

In the above examples, each reference to a data member or member function started with an object or object pointer. Inside a member function, however, the object or object pointer isn't necessary to refer to the object for which the member function is executing.

For example, inside the `PrintEmployee()` function, you can refer to the data member `employeeSalary` directly, without referring to an object or object pointer:

```
if ( employeeSalary <= 200 )
cout << "Give this person a raise!!!";
```

This code is kind of puzzling. What object does `employeeSalary` belong to? After all, you're used to writing:

```
myObject->employeeSalary
```

instead of just:

```
employeeSalary
```

The key to this puzzle lies in knowing which object spawned the call of `PrintEmployee()` in the first place. Although this may not be obvious, a call to a nonstatic member function must originate with a single object.

As we'll see later, class members may be declared as `static`. A `static` data member holds a value that is global to a class and not specific to a single object of that class. A *static member*function is usually designed to work with a class's static data members.

Suppose you called `PrintEmployee()` from a non-`Employee` function (such as `main()`). You must precede this call with a reference to an object:

```
employeePtr->PrintEmployee();
```

Whenever a member function is called, C++ keeps track of the object used to call the function. This object is known as the current object.

In the call of `PrintEmployee()` above, the object pointed to by `employeePtr` is the current object. Whenever this call of `PrintEmployee()` refers to an Employee data member or function without using an object reference, the current object (in this case, the object pointed to by `employeePtr`) is assumed.

Suppose `PrintEmployee()` then called another `Employee` function. The object pointed to by `employeePtr` is still considered the current object. A reference to `employeeSalary` would still refer to the current object's copy of `employeeSalary`. The point to remember is, a nonstatic member function always starts up with a single object in mind.

2.3.2 The `This` Object Pointer

C++ provides a generic object pointer, available inside any mem-ber function, that points to the current object. The generic pointer has the name `this`. For example, inside every `Employee` function, the line:

```
this->employeeSalary = 400;
```

is equivalent to this line:

```
employeeSalary = 400;
```

`this` is useful when a member function wants to return a pointer to the current object, pass the address of the current object on to another function, or just store the address somewhere. This line of code:

```
return this;
```

returns the address of the current object.

2.4 Destroying an Object

When you create an object using `new`, you've got to take responsibility for destroying the object at the appropriate time. Just as a C programmer balances a call to `malloc()` with a call to `free()`, a C++ programmer balances each use of the `new` operator with an eventual use of the `delete` operator.

Here's the syntax:

```
Employee *employeePtr;

employeePtr = new Employee;
delete employeePtr;
```

As you'd expect, `delete` destroys the specified object, freeing up any memory allocated for the object. Note that this freed up memory only includes memory for the actual object and does not include any extra memory you may have allocated.

For example, suppose the object is a structure and one of its data members is a pointer to another structure. When you delete the first structure, the second structure is not deleted. If delete is used with a pointer having a value of 0, delete does nothing. If the pointer has any other value delete will try to destroy the specified object.

2.5 Member Functions

Once your structure is declared, you're ready to write your member functions. Member functions behave in much the same way as ordinary functions, with a few small differences. One difference, pointed out earlier, is that a member function has access to the data members and member functions of the object used to call it. Another difference lies in the function implementation's title line. Here's a sample:

```
void Employee::PrintEmployee()
{
cout << "Employee Name: " << employeeName << "\n";
}
```

Notice that the function name is preceded by the class name and two colons (`Employee::`)

This notation is **mandatory** and tells the compiler that this function is a member of the specified class.

2.5.1 The Constructor Function

Typically, when you create an object, you'll want to perform some sort of initialization on the object. For instance, you might want to provide initial values for your object's data members.

The *constructor function* is C++'s built-in initialization mechanism. The constructor function (or just plain constructor) is a member function that has the **same name** as the object's class. For example, the constructor for the `Employee` class is named `Employee()`.

When an object is created, the constructor for that class gets called.

Consider this code:

```
Employee *employeePtr;

employeePtr = new Employee;
```

In the second line, the new operator allocates a `new Employee` object, then immediately calls the object's constructor. Once the constructor returns, the address of the new object is assigned to `employeePtr`.

This same scenario holds true in this declaration:

```
Employee employee1;
```

As soon as the object is created, its constructor is called. Here's our `Employee` struct declaration with the constructor declaration added in:

```
const short kMaxNameSize = 20;

struct Employee
```

```
  {
    // Data members...
    char employeeName[ kMaxNameSize ];
    long employeeID;
    float employeeSalary;

    // Member functions...
    Employee();
   void PrintEmployee();
 };
```

Notice that the constructor is declared *without* a return value. Constructors **never** return a value.

Here's a sample constructor:

```
Employee::Employee()
{
employeeSalary = 200.0;
}
```

Constructors are optional. If you don't have any initialization to perform, there is no need to define one.

You can add parameters to your constructor function. Constructor parameters are typically used to provide initial values for the object's data members. Here's a new version of the Employee() constructor:

```
Employee::Employee(char *name, long id, float salary)
{
   strncpy(employeeName, name, kMaxNameSize);
   employeeName[ kMaxNameSize - 1 ] = '\0';
   employeeID = id;
   employeeSalary = salary;
}
```

The constructor copies the three parameter values into the corresponding data members.

Notice that `strncpy()` was used, ensuring that the copy will work, even if the source string was not prop-erly terminated. A `NULL` terminator is provided at the end of the string for just such an emergency.

The object that was just created is always the constructor's current object. In other words, when the constructor refers to an `Employee` data member, such as `employeeName` or `employeeSalary`, it is referring to the copy of that data member in the newly created object.

This line of code supplies the new operator with a set of parameters to pass on to the constructor:

```
employeePtr = new Employee( "David Marshall", 1000, 200.0 );
```

This line of code does the same thing without using new:

```
Employee employee1( "David Marshall", 1000, 200.0 );
```

As you'd expect, this code creates an object named `employee1` by calling the `Employee` constructor, passing it the three specified parameters.

Just for completeness, here's the class declaration again, showing the new constructor:

```
struct Employee
  {
    // Data members...
    char employeeName[ kMaxNameSize ];
    long employeeID;
    float employeeSalary;

    // Member functions...
    Employee( char *name, long id, float salary );
    void PrintEmployee();
};
```

2.5.2 The Destructor Function

The destructor function is called automatically when you `delete` an object or it goes out of scope.

Use the destructor to clean up after your object before it goes away. For instance, you might use the destructor to deallocate any additional memory your object may have allocated.

The destructor function is named by a tilde character (∼) followed by the class name. The destructor for the Employee class is named ∼`Employee()`. The destructor has no return value and no parameters.

Here's a sample destructor:

```
Employee::~Employee()
{
cout << "Deleting employee #" << employeeID << "\n";
}
```

If you created your object using `new`, the destructor is called when you use delete:

```
Employee *employeePtr;

employeePtr = new Employee;
delete employeePtr;
```

If your object was defined directly, the destructor is called just before the object is destroyed. For example, if the object was declared at the beginning of a function, the destructor is called when the function exits.

If your object was declared as a global or static variable, its constructor will be called at the beginning of the program and its destructor will be called just before the program exits. Yes, global objects have scope, just as local objects do.

Here's an updated `Employee` *class* declaration showing the constructor and destructor:

```
struct Employee
 {
   // Data members...
   char employeeName[kMaxNameSize];
   long employeeID;
   float employeeSalary;

   // Member functions...
   Employee( char *name, long id, float salary );
   ~Employee();
   void PrintEmployee();
  };
```

When you declare a `class`, you need to decide which data members and functions you'd like to make available to the rest of your program. C++

gives you the power to hide a class's functions and data from all the other functions in your program, or allow access to a select few.

For example, consider the `Employee` class we've been working with throughout this chapter. In the current model, an Employee's name is stored in a single array of chars. Suppose you wrote some code that created a new Employee, specifying the name, id, and salary, then later in your program you decided to modify the Employee's name, perhaps adding a middle name provided while your program was running.

With the current design, you could access and modify the Employee's `employeeName` data member from anywhere in your program. As time passes and your program becomes more complex, you might find yourself accessing `employeeName` from several places in your code.

Now imagine what happens when you decide to change the implementation of `employeeName`. For example, you might decide to break the single employeeName into three separate data members, one each for the first, middle, and last names. Imagine the hassle of having to pore through your code finding and modifying every single reference to `employeeName`, making sure you adhere to the brand new model.

C++ allows you to hide the implementation details of a class (the specific type of each data member, for example), funneling all access to the implementation through a specific set of interface routines. By hiding the implementation details, the rest of your program is forced to go through the interface routines your class provides. That way, when you change the implementation, all you have to do is make whatever changes are necessary to the class's interface, without having to modify the rest of your program.

2.6 Access Priveleges

The mechanism C++ provides to control access to your class's implementation is called the access specifier.

C++ allows you to assign an *access specifier* to any of a class's data members and member functions. The access specifier defines which of your program's functions have access to the specified data member or function. The access specifier must be `public`, `privat`, or `protected`:

- If a data member or function is marked as `private`, access to it is limited to member functions of the same class (or, as you'll see later in

the chapter, to classes or member functions marked as a friend of the class).

- The `public` specifier gives complete access to the member function or data member, limited only by scope.

- The third C++ access code is `protected`. The protected access code offers the same protection as private, with one exception. A `protected` data member or function can also be accessed by a class derived from the current class. Since we won't get to derived classes till later in the book, we'll put off discussion of the protected access code till then.

By default, the data members and member functions of a class declared using the `struct` keyword are all `public`. By adding the `private` keyword to our `class` declaration, we can limit access to the Employee data members, forcing the outside world to go through the provided member functions:

```
struct Employee
{
   // Data members...
   private:
     char employeeName[ kMaxNameSize ];
     long employeeID;
     float employeeSalary;

   // Member functions...
   public:
     Employee( char *name, long id, float salary );
     ~Employee();
     void PrintEmployee();
};
```

Once the compiler encounters an access specifier, all data members and functions that follow are marked with that code, at least until another code is encountered. In this example, the three data members are marked as private and the three member functions are marked as public.

Note the : after the access specifier. Without it, your code won't compile!

Here's the new version of the Employee `class`:

class Employee // Data members... private: char employeeName[kMaxNameSize]; long employeeID; float employeeSalary;

// Member functions... public: Employee(char *name, long id, float salary); Employee(); void PrintEmployee(); ;

Notice that the private access specifier is still in place. Since the members of a class-based class are private by default, the private access specifier is not needed here, but it does make the code a little easier to read. The public access specifier is necessary, however, to give the rest of the program access to the Employee member functions.

2.7 employee.cpp, source code

We now give a complete code listing to bring togther all the facets we have met in this chapter so far.

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

const short kMaxNameSize = 20;

class Employee
{
// Data members...
private:
char employeeName[ kMaxNameSize ];
long employeeID;
float employeeSalary;

// Member functions...
public:
Employee( char *name, long id, float salary );
~Employee();
void PrintEmployee();
};

Employee::Employee( char *name, long id, float salary )
```

```
{
strncpy( employeeName, name, kMaxNameSize );

employeeName[ kMaxNameSize - 1 ] = '\0';

employeeID = id;
employeeSalary = salary;

cout << "Creating employee #" << employeeID << "\n";
}

Employee::~Employee()
{
cout << "Destroying employee #" << employeeID << "\n";
}

void Employee::PrintEmployee()
{
cout << "-----\n";
cout << "Name:   " << employeeName << "\n";
cout << "ID:     " << employeeID << "\n";
cout << "Salary: " << employeeSalary << "\n";
cout << "-----\n";
}

int main()
{
Employee employee1( "Dave Mark", 1, 200.0 );
Employee *employee2;

employee2 = new Employee( "Steve Baker", 2, 300.0 );

employee1.PrintEmployee();
employee2->PrintEmployee();

delete employee2;

return 0;
```

```
}
```

2.8 Friends

In our last program, the `Employee` class marked its data members as private and its member functions as public. As we discussed earlier, the idea behind this strategy is to hide the implementation details of a class from the rest of the program, funneling all access to the class's data members through a set of interface routines.

For example, suppose we wanted to provide access to the Employee class's `employeeSalary` data member. Since `employeeSalary` is marked as private, there's no way to access this data member outside the Employee class. If we wanted to, we could provide a pair of public member functions a user of the `Employee` class could use to retrieve (`GetEmployeeSalary()`) and modify (`ChangeEmployeeSalary()`) the value of employeeSalary.

Sometimes this strategy just doesn't work well: For example, suppose you created a Payroll class to generate paychecks for your Employees. Clearly, the Payroll class is going to need access to an Employee 's salary.

But if you create a public `GetEmployeeSalary()` member function (or mark `employeeSalary` as public) you'll make `employeeSalary` available to the entire program, something you might not want to do.

The solution to this problem is provided by C++'s *friend mechanism*. C++ allows you to designate a class or a single member function as a *friend* to a specific class. In the previous example, we could designate the Payroll class as a friend to the Employee class:

```
//---------------------------------------  Payroll

class Payroll
{
// Data members...
private:

// Member functions...
public:
Payroll();
~Payroll();
```

```
void PrintCheck( Employee *payee );
};

//---------------------------------------  Employee

class Employee
{
 friend class Payroll;

  // Data members...
  private:
   char employeeName[ kMaxNameSize ];
   long employeeID;
  float employeeSalary;

 // Member functions...
  public:
  Employee( char *name, long id, float salary );
  ~Employee();
    void PrintEmployee();
};
```

The `friend` statement, in the first line of the Employee class declaration, is *always* placed in the class whose data members and functions are being shared. In this case, the `Employee` class is willing to share its private data members and functions with its new friend, the `Payroll` class. Once the `Payroll` class has friend access to the Employee class, it can access private data members and functions like `employeeSalary`.

2.8.1 Three Types of Friends

There are three ways to designate a friend.

- (As we've already seen) You can designate an entire class as a friend to a second class.
- You can also designate a specific class function as a friend to a class. For example, the Payroll class we just declared contains a function

named `PrintCheck()`. We might want to designate the `PrintCheck()` function as a friend of the Employee class, rather than the entire Payroll class.

```
class Employee
 {
   friend void Payroll::PrintCheck( Employee *payee );;

   // Data members...
   private:
     char employeeName[ kMaxNameSize ];
     long employeeID;
     float employeeSalary;

   // Member functions...
   public:
     Employee( char *name, long id, float salary );
     ~Employee();
     void PrintEmployee();
  };
```

This time, the friend definition specified the Payroll mem-ber function `Payroll::PrintCheck()` instead of the entire Payroll class. Since the friend statement referred to a member function of another class, the full name of the function (including the class name and the two colons) was included.

- You can also designate a nonmember function as a friend. For example, you could designate `main()` as a friend to the Employee class:

```
class Employee
 {
   friend int main();

   // Data members...
   private:
     char employeeName[ kMaxNameSize ];
     long employeeID;
```

```
    float employeeSalary;

  // Member functions...
  public:
    Employee( char *name, long id, float salary );
    ~Employee();
    void PrintEmployee();
};
```

This arrangement gives main() access to all Employee data members and functions, even those marked as private. Just because main() is a friend doesn't give any special priveleges to any other functions, however. Choose your friends carefully!

2.8.2 A Friendly Example

Our next example combines the Employee class created earlier with the Payroll class described in this section.

The `friends.cpp` source code follows:

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

const short kMaxNameSize = 20;

class Employee;


//---------------------------------------  Payroll

class Payroll
{
// Data members...
private:

// Member functions...
public:
Payroll();
```

```
~Payroll();
void PrintCheck( Employee *payee );
};

//---------------------------------------  Employee

class Employee
{
friend void Payroll::PrintCheck( Employee *payee );

// Data members...
private:
char employeeName[ kMaxNameSize ];
long employeeID;
float employeeSalary;

// Member functions...
public:
Employee( char *name, long id, float salary );
~Employee();
void PrintEmployee();
};


//-------------------  Payroll Member Functions

Payroll::Payroll()
{
cout << "Creating payroll object\n";
}

Payroll::~Payroll()
{
cout << "Destroying payroll object\n";
}

void Payroll::PrintCheck( Employee *payee )
{
```

```
cout << "Pay $" << payee->employeeSalary
 << " to the order of "
 << payee->employeeName << "...\n\n";
}


//-------------------  Employee Member Functions

Employee::Employee( char *name, long id, float salary )
{
strncpy( employeeName, name, kMaxNameSize );

employeeName[ kMaxNameSize - 1 ] = '\0';

employeeID = id;
employeeSalary = salary;

cout << "Creating employee #" << employeeID << "\n";
}

Employee::~Employee()
{
cout << "Destroying employee #" << employeeID << "\n";
}

void Employee::PrintEmployee()
{
cout << "-----\n";
cout << "Name:   " << employeeName << "\n";
cout << "ID:     " << employeeID << "\n";
cout << "Salary: " << employeeSalary << "\n";
cout << "-----\n";
}


//---------------------------------------  main

int main()
```

```
{
Employee *employee1Ptr;
Payroll *payroll1Ptr;

payroll1Ptr = new Payroll;

employee1Ptr = new Employee( "Carlos Derr", 1000, 500.0 );

employee1Ptr->PrintEmployee();

payroll1Ptr->PrintCheck( employee1Ptr );

delete employee1Ptr;
delete payroll1Ptr;

return 0;
}
```

The Output looks like this:

```
Creating payroll object
Creating employee #1000
----
Name: Carlos Derr
ID: 1000
Salary: 500
----
Pay $500 to the order of Carlos Derr...
Destroying employee #1000
Destroying payroll object
```

`friends.cpp` starts out just like `employee.cp`, with the same two `#include`s and the same `const` definition:

Since the Payroll class declaration refers to the Employee class (check out the parameter to `PrintCheck()`) and its declaration comes first, we'll need a forward declaration of the Employee

Next comes the declaration of the Payroll class. To keep this example as simple as possible, we've stripped Payroll down to its bones: no data

members, a constructor, a destructor, and a `PrintCheck()` function. Further down in the source, the Employee class will mark the `PrintCheck()` function as a friend. Next comes the Employee class declaration. You may have noticed that we didn't list the Payroll member functions right after the Payroll class declaration. This was because `Payroll::PrintCheck()` refers to the Employee data mem-ber employeeSalary, which hasn't been declared yet.

Take a look at the friend declaration inside the Employee class declaration. Notice that we've opted to make `Payroll::PrintCheck()` a friend of the Employee class. Now, `PrintCheck()` is the only Payroll function with access to the private Employee data members.

Interestingly, if you leave `PrintCheck()`'s parameter out of the friend statement, the code won't compile. Since you can have more than one function with the same name (Recall overloaded functions), if the parameter is left out, the compiler tries to match the friend statement with a version of `PrintCheck()` with no parameters. When it doesn't find one, the compiler reports an error.

Next come the Payroll member functions. The constructor and destructor print messages letting you know they were called, while `PrintCheck()` prints up a simulated check using the private Employee data members employeeSalary and employeeName.

The Employee member functions are the same as they were in declared next.

Once again, `main()` is where the action is. We start off by defining a couple of pointers, one to an Employee object and one to a Payroll object.

The two constructors are called and then `PrintEmployee()` is called.

Next, `PrintCheck()` is called. `PrintCheck()` takes a pointer to the Employee object as a parameter. A check is printed to the specified Employee using employeeName and employeeSalary:

Finally, both objects are `deleted`: The two destructors functions calles which print their respective messages.

Chapter 3

Inheritance, Derived Functions, Virtual Functions

C++ allow you to use one class dclaration, known as a *base class*, as the basis for the declaration of a second class, known as a *derived class*.

For example, you might declare an Employee class that describes your company's employees. Next, you might declare a Sales class, based on the Employee class, that describes employees in the sales department.

This chapter is emphasize the advantages of classes derived from other classes and gives several examples.

3.1 Inheritance

One of the most important features of class derivation is *inheritance*. A derived class inherits *all* of the data members and member functions from its base class.

As an example, consider the following class declaration:

```
class Base
{
  public:
   short baseMember;
   void SetBaseMember( short baseValue );
};
```

This class, `Base`, has two members, a data member named `baseMember` and a member function named `SetBaseMember()`.

Both of these members will be inherited by any classes derived from this class.

Here's another class declaration:

```
class Derived : Base
{
public:
short derivedMember;
void SetDerivedMember( short derivedValue );
}
```

This class is a derived class named, appropriately enough, `Derived`. The `: Base` at the end of the title tells you that this object is derived from the class named `Base`.

As you'd expect, this object has its own copy of the data member derivedMember as well as access to the member function `SetDerivedMember()`.

What you might not have expected are the members inherited by this object, that is, the `Base` class data member `baseMember` as well the Base class member function `SetBaseMember()`.

Here's some code that allocates a Derived object, then accesses various data members and functions:

```
Derived *derivedPtr;

derivedPtr = new Derived;
derivedPtr->SetDerivedMember( 20 );
cout << "derivedMember = " << derivedPtr->derivedMember;
derivedPtr->SetBaseMember( 20 );
cout << "\nbaseMember = " << derivedPtr->baseMember;
```

Notice that the object pointer derivedPtr is used to access its own data members and functions as well as its inherited data members and functions. Notice also that the example does not create a Base object. This is important. When an object inherits data members and functions from its base class, the compiler allocates the extra memory needed for all inherited members right along with memory for the object's own members. class is derived from the class named Base. As you'd expect, this object has its own copy of the data member derivedMember as well as access to the member function

3.2 Access and Inheritance

Although a derived class inherits all of the data members and member functions from its base class, it doesn't necessarily retain access to each member.

Here's how this works. When you declare a derived class, you declare its base class as either `public` or `private`. One way to do this is to include either `public` or `private` in the title line of the declared class.

For example, in the declaration below, the class `Base` is marked as `public`:

```
class Derived : public Base
{
   public:
     short derivedMember;
     void SetDerivedMember( short derivedValue );
}
```

In the next declaration, `Base` is marked as private:

```
class Derived : private {\tt Base}
{
public:
short derivedMember;
void SetDerivedMember( short derivedValue );
}
```

You can also mark a base class as `public` or `private` by leaving off the access specifier. If you use the `class` keyword to declare the derived class, the base class defaults to `private`. If you use the `struct` keyword to declare the derived class, the base class defaults to `public`.

Once you know whether the base class is public or private, you can determine the access of each of its inherited members by following these three rules:

- The derived class does not have access to private members inherited from the base class. This is true regardless of whether the base class is public or private.
- If the base class is public, the members inherited from the base class retain their access level (providing the inherited member is not private, of course). This means that an inherited public member remains public and an inherited protected member remains protected.

- If the base class is private, the members inherited from the base class are marked as private in the derived class.

Previously, we adopted the strategy of declaring our data members as private and our member functions as public. This approach works well if the class will never be used as a base class for later derivation. If you ever plan on using a class as the basis for other classes, declare your data members as protected and your member functions as public.

A protected member can be accessed only by members of its class or by members of any classes derived from its class. In a base class, protected is just like private. The advantage of protected is that it allows a derived class to access the member, while protecting it from the outside world. We'll get back to this strategy in a bit. For now, let's take a look at an example of class derivation.

3.3 A Class Derivation Example

So far in this chapter, we've learned how to derive one class from another and you've been introduced to the protected access specifier. Our first sample program brings these lessons to life.

```
#include <iostream.h>


//---------------------------------------  Base

class Base
{
// Data members...
private:
short baseMember;

// Member functions...
protected:
void SetBaseMember( short baseValue );
short GetBaseMember();
};
```

```
void Base::SetBaseMember( short baseValue )
{
baseMember = baseValue;
}

short Base::GetBaseMember()
{
return baseMember;
}


//---------------------------------------  Base:Derived

class Derived : public Base
{
// Data members...
private:
short derivedMember;

// Member functions...
public:
void SetMembers( short baseValue,
short derivedValue );
void PrintDataMembers();
};

void Derived::SetMembers( short baseValue,
short derivedValue )
{
derivedMember = derivedValue;
SetBaseMember( baseValue );
}

void Derived::PrintDataMembers()
{
cout << "baseMember was set to "
<< GetBaseMember() << '\n';
```

```
cout << "derivedMember was set to "
<< derivedMember << '\n';
}


//---------------------------------------  main()

int main()
{
Derived *derivedPtr;

derivedPtr = new Derived;

derivedPtr->SetMembers( 10, 20 );

derivedPtr->PrintDataMembers();

return 0;
}
```

The ouput of this program is:

```
baseMember was set to 10
derivedMember was set to 20
```

Let's take a look at the source code.

As usual, `derived.cpp` starts by including <iostream.h>:

Next, we declare a class named `Base`, which we'll use later as the basis for a second class named `Derived`. Base has a single data member, a short named `baseMember`, which is marked as `private`.

Base also includes two member functions, each marked as `protected`. `SetBaseMember()` sets baseMember to the specified value while `GetBaseMember()` returns the current value of `baseMember`.

Since `baseMember is private`, it cannot be accessed by any function outside the `Base` class. Since `SetBaseMember()` and `GetBaseMember()` are protected, they can only be accessed by `Base` member functions and from within any classes derived from `Base`. Note that `main()` cannot access either of these functions.

Our second class, `Derived`, is derived from Base:

The `public` keyword following the colon in the class title line marks `Base` as a public base class. As mentioned earlier, if a base class is declared as public, all inherited public members remain `public` and inherited `protected` members remain protected. Inherited `private` members are **not** accessible by the derived class.

If you marked `Base` as `private` instead of `public` all inherited members would be marked as `private` and would not be accessible by any classes derived from Derived. The point here is this:

- If you mark the base class as `private` you effectively **end** the inheritance chain.

As a general rule,*you should declare your derived classes using* `public` *inheritance*:

It's rare that you'd want to reduce the amount of information inherited by a derived class. Most of the time, a derived class is created to extend the reach of the base class by adding new data members and functions.

`Derived` has a single data member, a short named `derivedMember`, which is declared as private. `derivedMember` can only be accessed by a `Derived` member function. Derived contains two member functions, `SetMembers()` and `PrintDataMembers()`.

`SetMembers()` takes two shorts, assigns the first to `derivedMember`, and passes the second to `SetBaseMember()`. `SetBaseMember()` was used because `Derived` does not have direct access to `baseMember`.

`PrintDataMembers()` prints the values of `baseMember` and `derivedMember`. Since `Derived` doesn't have direct access to `baseMemeber`, `GetBaseMember()` is called to retrieve the value.

`main()` starts by declaring a `Derived` pointer and then using new to create a `new Derived` object (since we didn't include a constructor for either of our classes, nothing exciting has hap-pened yet):

It's important to understand that when the `Derived object` is created, it receives its own copy of `baseMember`, even though it doesn't have access to baseMember. If the `Derived` object wants to modify its copy of `baseMember`, it will have to do so via a call to `Base::SetBaseMember()`, which it also inherited. Now things start to get interesting. `main()` uses the pointer to the Derived object to call `SetMembers()`, setting its copy of `baseMember` to 10 and `derivedMember` to 20.

Next, we call the `Derived` member function `PrintDataMembers()`

The values of the two data members are successfully set when the program is ran:

`baseMember` was set to 10;

`derivedMember` was set to 20,

Just as the `Derived` object pointer is able to take advantage of inheritance to call `SetBaseMember(), PrintDataMembers()` is able to print the value of the inherited data member `baseMember` by calling `GetBaseMember()`.

3.4 Derivation, Constructors and Destructors

When an object is created, its constructor is called to initialize the object's data members. When the object is deleted, its destructor is called to perform any necessary cleanup.

Suppose the object belongs to a derived class, and suppose it inherits a few data members from its base class.

How do these inherited data members get initialized?

When the object is deleted, who does the cleanup for the inherited data members?

As it turns out, C++ solves this tricky issue for you. Before the compiler calls an object's constructor, it first checks to see whether the object belongs to a derived class. If so, the constructor belonging to the base class is called and then the object's own constructor is called. The base class constructor initializes the object's inherited members, while the object's own constructor initializes the members belonging to the object's class .

The reverse holds true for the destructor. When an object of a derived class is deleted, the derived class's destructor is called and then the base class's destructor is called.

3.4.1 The Derivation Chain

There will frequently be times when you derive a class from a base class that is, itself, derived from some other class. Each of these classes acts like a link in a derivation chain. The constructor/ destructor calling sequence just described still holds, no matter how long the derivation chain.

Suppose you declare three classes, A, B, and C, where class B is derived from A and C is derived from B.

When you create an object of class B, it will inherit the mem-bers from class A. When you create an object of class C, it will inherit the members from class B, which includes the previously inherited members from class A.

When an object from class C is created, the compiler follows the derivation chain from C to B to A and discovers that A is the ultimate base class in this chain. The compiler calls the class A constructor, then the class B constructor, and finally the class C constructor.

When the object is deleted, the class C destructor is called first, followed by the class B destructor and, finally, by the class A destructor.

3.4.2 A Derivation Chain Example

Our second sample program demonstrates the order of constructor and destructor calls in a three-class derivation chain. Close the current project by selecting from the menu.

The source code listing is:

```
#include <iostream.h>


//---------------------------------------  Gramps

class Gramps
{
// Data members...

// Member functions...
public:
Gramps();
~Gramps();
};

Gramps::Gramps()
{
cout << "Gramps' constructor was called!\n";
}

Gramps::~Gramps()
```

```
{
cout << "Gramps' destructor was called!\n";
}


//---------------------------------  Pops:Gramps

class Pops : public Gramps
{
// Data members...

// Member functions...
public:
Pops();
~Pops();
};

Pops::Pops()
{
cout << "Pops' constructor was called!\n";
}

Pops::~Pops()
{
cout << "Pops' destructor was called!\n";
}


//---------------------------------  Junior:Pops

class Junior : public Pops
{
// Data members...

// Member functions...
public:
Junior();
~Junior();
```

```
};

Junior::Junior()
{
cout << "Junior's constructor was called!\n";
}

Junior::~Junior()
{
cout << "Junior's destructor was called!\n";
}


//---------------------------------------  main

int main()
{
Junior *juniorPtr;

juniorPtr = new Junior;

cout << "----\n";

delete juniorPtr;

return 0;
}
```

The output of the program is as follows:

```
Gramps' constructor was called!
Pops' constructor was called!
Junior's constructor was called!
----
Junior's destructor was called!
Pops' destructor was called!
Gramps' destructor was called!
```

As you can see by the output, each class constructor was called once, then each class destructor was called once in reverse order.

Notice that none of the classes in this program have any data members. For the moment, we're interested only in the order of constructor and destructor calls. Both the constructor and the destructor are declared `public`. The `Gramps` constructor and destructor are pretty simple; each prints an appropriate message to the console:

Our next class, `pops`, is derived from the `Gramps` class. Notice that we use the `public` keyword in the `class` title line. This ensures that the constructor and the destructor inherited from `Gramps` are marked as `public` inside the `Pops` class. Once again, this class has no data members. Both the constructor and the destructor are marked as `public`. They'll be inherited by any class derived from `Pops`.

Just like those of `Gramps`, the `Pops` constructor and destructor are simple and to the point; each sends an appropriate message to the console.

`Junior` is to `Pops` what `Pops` is to `Gramps`. `Junior` inherits not only the `Pops` members but the `Gramps` members as well (as you'll see in a minute, when you create and then delete a `Junior` object, both the `Gramps` and the `Pops` constructor and destructor will be called)

The `Junior` constructor and destructor are just like those of `Gramps` and `Pops`; each sends an appropriate message to the console.

`main()`'s job is to create and delete a single `Junior` object. When the `Junior` object is created, the derivation chain is followed backward until the ultimate base class, `Gramps`, is reached. The `Gramps` constructor is called, giving the `Gramps` class a chance to initialize its data members. Next, the `Pops` constructor is called, and the `Junior` constructor is called.

Next, the `Junior` object is deleted, and, this time, the derivation chain is followed in the reverse order. The `Junior` destructor is called, then the `Pops` destructor, and, finally, the `Gramps` destructor is called.

Now consider this examle further:

Notice in the source code listing above that each of the three classes marked their constructor and destructor as `public`. What would happen if you changed the Gramps constructor to `private`?

Your code wouild not compile because `Junior` no longer has access to the `Gramps` constructor. Now if we change the Gramps constructor from `private` to `protected` the program will compile.

Why?

This time, when `Junior` inherited the `Gramps` constructor it had access to the `Gramps` constructor. Recall that when a derived class inherits a `protected` member from a `public` base class, the inherited member is marked as `protected`.

Now what happens if we change the `Junior` constructor from `public` to `protected` and recompile?

This time the compiler complains that the `Junior` constructor was not accessible. Since the `Junior` constructor was declared `protected` it is accessible by classes derived from `Junior`, but not by outside functions like `main()`.

When `main()` creates a new `Junior` object, it must have access to the `Junior` constructor. On the other hand, you've seen that `main()` does not need access to the `Gramps` or `Pops` constructors to create a `Junior` object. When you changed the `Gramps` constructor to `protected`, `Junior` had access to the `Gramps` constructor and `main()` didn't yet the program still compiles.

3.5 Base Classes and Constructors with Parameters

Our first program in this chapter, `derived`, declared two classes, `Base and Derived`. Neither of these classes included a constructor. Our second program, `gramps`, featured three classes. Though all three classes declared a constructor, none of the constructors declared any parameters.

Our next example enters uncharted waters by declaring classes whose constructors contain parameters.

When are constructor parameters important?

In a world without class derivation, not much. When you start working with derived classes, however, things get a bit more complex.

Consider a base class whose constructor has only a single parameter:

```
class Base
 {
   public:
     Base(short baseParam);
  };
```

Now, add a derived class based on this base class:

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

Notice that the derived class constructor is declared without a parameter. When a `Derived` object is created, the Base constructor is called. What parameter is passed to this constructor?

The secret lies in the definition of the derived class constructor. When a base class constructor has parameters, you have to provide some extra information in the derived class constructor's title line. This information tells the compiler how to map data from the derived class constructor to the base class constructor's parameter list.

For example, we might define the derived class constructor this way:

```
Derived::Derived() : Base(20)
{
cout << "Inside the Derived constructor";
}
```

Notice the `: Base( 20 )` at the end of the title line. This code tells the compiler to pass the number 20 as a parameter when the Base constructor is called.

This technique is really useful when your derived class constructor also has parameters. Check out the following piece of code:

```
Derived::Derived(short derivedParam) : Base(derivedParam)
{
}
```

This constructor takes a single parameter, `derivedParam`, and maps it to the single parameter in its base class constructor. When a Derived object is created, as follows,

```
Derived *derivedPtr;
derivedPtr = new Derived(20);
```

the parameter is passed to the `Base` constructor. Once the `Base` constructor returns, the same parameter is passed to the `Derived` constructor.

In the preceding example, the `Derived` constructor does nothing but pass along a parameter to the `Base` constructor. Though it may take some getting used to, this technique is quite legitimate. It is perfectly fine to define an empty function whose sole purpose is to map a parameter to a base class constructor.

Class Derivation Example

Our next example program combines the class derivation techniques from our first two programs with the constructor parameter-mapping mechanism described in the previous section.

The code listing, `square.cpp` is as follows:

```
#include <iostream.h>


//---------------------------------------  Rectangle

class Rectangle
{
// Data members...
protected:
short height;
short width;

// Member functions...
public:
Rectangle( short heightParam, short widthParam );
void DisplayArea();
};

Rectangle::Rectangle( short heightParam, short widthParam )
{
height = heightParam;
width = widthParam;
```

```
}

void Rectangle::DisplayArea()
{
cout << "Area is: " <<
height * width << '\n';
}


//---------------------------------------  Rectangle:Square

class Square : public Rectangle
{
// Data members...

// Member functions...
public:
Square( short side );
};

Square::Square( short side ) : Rectangle( side, side )
{
}


//---------------------------------------  main()

int main()
{
Square *mySquare;
Rectangle *myRectangle;

mySquare = new Square( 10 );
mySquare->DisplayArea();

myRectangle = new Rectangle( 10, 15 );
myRectangle->DisplayArea();
```

```
return 0;
}
```

The output of the program is:

```
Area is: 100
Area is: 150
```

square.cp starts in the usual way, by including <iostream.h>:

Next, the first of two classes is declared. `Rectangle` will act as a base class The data members of our base class are declared as `protected`; the member functions, `public`. `height` and `width` hold the height and width of a Rectangle object:

The `Rectangle()` constructor takes two parameters, `heightParam` and `widthParam`, that are used to initialize the `Rectangle` data members.

The member function `DisplayArea()` displays the area of the current object:

The `Square` class is derived from the `Rectangle` class. Just as (geometrically speaking) a square is a specialized form of rectangle (a rectangle whose sides are all equal), a `Square` object is a specialized `Rectangle` object.

The `Square` class has no data members, just a single member function, the `Square()` constructorw hich has one purpose in life — It maps the single `Square()` parameter to the two parameters required by the `Rectangle()` constructor. A square whose side has a length of side is equivalent to a rectangle with a height of side and a width of side:

This is expressed by:

```
Square::Square( short side ) : Rectangle( side, side )
{
}
```

`main()` starts by declaring a Square pointer and a Rectangle pointer. The Square pointer is used to create a new Square object with a side of 10:

As specified by the `Square()` constructor's title line, the compiler calls `Rectangle()`, passing 10 as both `heightParam` and `widthParam`. The `Rectangle()` constructor initializes the Square object's inherited data members `height` and `width` to 10, just as if you'd created a `Rectangle` with a height of 10 and a width of 10.

Next, the `Square` object's inherited member function, `DisplayArea()`, is called and this uses the inherited data members `height` and `width` to calculate the area of the Square (100 in this case.

As far as `DisplayArea()` is concerned, the object whose area it just calculated was a `Rectangle`. It had no idea it was working with data members inherited from a `Rectangle`. This illustrates part of the power of object programming.

Finally, the `Rectangle` pointer is used to create a new `Rectangle` object with a `height` of 10 and a `width` of 15: When we call `DisplayArea()` this time, it displays the appropriate area of 150.

This program demonstrates a *very important point.*

- With just a few lines of code, we can add a new dimension to an existing class without modifying the existing class.

The `Square` class takes advantage of what's already in place, building on the data members and member functions of its base class. Essentially, `Square` added a shortcut to the `Rectangle` class — a quicker way to create a `Rectangle` when the height and width are the same.

This may not seem like a significant gain to you, but there's an important lesson behind this example. C++ makes it easy to build upon existing models, to add functionality to your software by deriving from existing classes.

As you gain experience in object programming, you'll build up a library of classes that you'll use again and again. Sometimes, you'll use the classes as it is. At other times, you'll extend an existing class by deriving a new class from it. By deriving new classes from existing classes, you get the best of both worlds. Code that depends on the base classes will continue to work quite well without modification. Code that takes advantage of the new, derived classes will work just as well, allowing these classes to live in harmony with their base classes.

This examle serves to illustrate is what object programming is all about.

3.6 Overriding Member Function

In the preceding example, the derived class, `Square`, inherited the member function, `DisplayArea()`, from its base class, `Rectangle`. Sometimes, it is useful to *override* a member function from the base class with a more appropriate function in the derived class.

For example, you could have provided Square with its own version of `DisplayArea()` that based its area calculation on the fact that the height and width of a square are equal.

Let us consider another example: Suppose you create a base class named `Shape` and a series of derived classes such as `Rectangle`, `Circle`, and `Triangle`. You can create a `DisplayArea()` function for the Shape class, then override `DisplayArea()` in each of the derived classes.

Suppose you want to create a linked list of `Shape`s. To simplify matters for the software that manages the linked list, you can treat the derived objects as `Shape`s, no matter what their actual type. Then, when you call the `Shape`'s `DisplayArea()` function, their true identity will emerge. A `Triangle` will override the `Shape DisplayArea()` function with a `Triangle DisplayArea()` function. The `Rectangle` and `Circle` will have their own versions as well.

The trick is to get `C++` to call the proper overriding function, if one exists.

3.6.1 Creating a Virtual Function

The `Shape` linked list example we are developing has presented us with a slight problem:

Suppose the linked list contains a pointer to a `Shape` which is actually one of `Rectangle, Triangle`, or `Circle`. Now suppose that `Shape` pointer is used to call the member function `DisplayArea()` like:

```
myShapePtr->DisplayArea();
```

As expressed currently, this code will call the function `Shape::DisplayArea()` even if the Shape pointed to by `myShapePtr` is a `Rectangle, Triangle`, or `Circle`.

The solution to this problem:

Declare `Shape::DisplayArea()` as a *virtual function*.

To do this simply prefix the member function with the `virtual keyword`, in this example:

```
class Shape
{
// Data members...
```

```
// Member functions...
public:
virtual void WhatAmI();
};
```

By declaring a base member function as `virtual`, we are asking the compiler to call the overriding function instead of the base function, even if the object used to call the function belongs to the base class.

3.6.2 A Virtual Function Example

Let us now complete this `Shape` example of virtual function overriding. We will be using a base class named `Shape` and two derived classes, `Rectangle` and `Triangle`.

The complete listing for `shape.cpp` is as follows:

```
#include <iostream.h>


//---------------------------------------  Shape

class Shape
{
// Data members...

// Member functions...
public:
virtual void WhatAmI();
};

void Shape::WhatAmI()
{
cout << "I don't know what kind of shape I am!\n";
}


//---------------------------------------  Shape:Rectangle
```

```
class Rectangle : public Shape
{
// Data members...

// Member functions...
public:
void WhatAmI();
};

void Rectangle::WhatAmI()
{
cout << "I'm a rectangle!\n";
}


//---------------------------------------  Shape:Triangle

class Triangle : public Shape
{
// Data members...

// Member functions...
public:
void WhatAmI();
};

void Triangle::WhatAmI()
{
cout << "I'm a triangle!\n";
}


//---------------------------------------  main()

int main()
{
Shape *s1, *s2, *s3;
```

```
s1 = new Rectangle;
s2 = new Triangle;
s3 = new Shape;

s1->WhatAmI();
s2->WhatAmI();
s3->WhatAmI();

return 0;
}
```

The output is relatively obvious:

```
I'm a rectangle!
I'm a triangle!
I don't know what kind of shape I am!
```

Initially, the base class `Shape` is declared. `Shape` contains a single member function, `WhatAmI()`. When it is called, `WhatAmI()` tells you what kind of shape it belongs to. Notice that it is declared using the `virtual` keyword, which tells the compiler that you'd like any overriding function to be called, if one exists.

Notice in the actual definition of `Shape::WhatAmI()` that the virtual keyword isn't used here. The virtual keyword is *only allowed* in the function declaration inside the class declaration.

Our next class, `Rectangle`, is derived from the `Shape` class and also has a single member function named `WhatAmI()`: `Rectangle`'s version of `WhatAmI()` is called when the object doing the calling is a `Rectangle` and simply outputs "`I'm a rectangle!`"

The final class, `Triangle`, is also derived from `Shape`, and, once again, `Triangle` has its own version of `WhatAmI()`

`main()` declares three Shape point-ers, `s1, s2`, and `s3`:

Each of these pointers is used to create a new object, a `Rectangle`, a `Triangle`, and a `Shape`, respectively:

```
s1 = new Rectangle;
s2 = new Triangle;
s3 = new Shape;
```

You may be wondering why the three pointers are all declared as `Shape` while the objects assigned to the pointers are of three different types. This is intentional and normal.

If you're building a linked list of shapes, you can store a pointer to each object in the list as a `Shape` pointer rather than as a `Rectangle` pointer or `Triangle` pointer. In this way, your list management software doesn't have to know what type of shape it is dealing with and is much easier to deal with:

If you want to call `WhatAmI()` (or some other, more useful function) for each object in the list, you just step through the list, one object at a time, treating each object as if it were a `Shape`. If the object belongs to a derived class that overrides the function, C++ will make sure the correct function is called.

Once our three objects are created, we try using each object to call `WhatAmI()`:

```
s1->WhatAmI();
s2->WhatAmI();
s3->WhatAmI();
\end
{verbatim}

When the {\tt Rectangle} object ({\tt s1}) is used to call {\tt
WhatAmI()}, we get this result:

\begin{verbatim}
I'm a rectangle!
```

When the `Triangle` object (`s12`) is used to call `WhatAmI()`, we get this result:

```
I'm a triangle!
```

Finally, when the `Shape` object (`s3`) is used to call `WhatAmI()`, we get this result:

```
I don't know what kind of shape I am!
```

In this example, the `Shape`class exists just so that we can create useful, derived classes from it. Creating a `Shape` object is not particularly useful.

3.7 Exercises

Exercise 3.1 *Modify the* `shape.cpp` *prgram to be a true linked list implementation of* `Shape`*s.*

Exercise 3.2 *Modify the* `shape.cpp` *prgram to include a* `Circle` *derived class.*

Chapter 4

Operator Overloading

Operator overloading is another extremely powerful and useful feature of Object Oriented programming

4.1 Overiding Built-in Operators

In C++, you can even overload any of the built-in operators, such as + or * to suit paricular applications.

To see why this might be useful consider the following example:

Imagine that you're running a restaurant and you want to write a program to handle your billing, print your menus, and so on. Your program might create a `MenuItem` class that looks something like this:

```
class MenuItem
{
   private:
     float price;
     char name[ 40 ];

    public:
     MenuItem::MenuItem( float itemPrice, char *itemName );
     float MenuItem::GetPrice( void );
};
```

Your program could define a `MenuItem` object for each item on the menu. When someone orders, you'd calculate the bill by adding together the price of each `MenuItem` like this:

```
MenuItem chicken( 8.99, "Chicken Jalfrezi" );
MenuItem wine( 2.99, "Rioja" );

float total;

total = chicken->GetPrice() + wine->GetPrice();
```

This particular diner had the chicken and a glass of wine. The total is calculated using the member function `GetPrice()`. *Nothing new here*, yet.

Operator overloading provides an alternative way of totalling up the bill. If we program things properly, the compiler will interpret the statement

```
total = chicken + wine;
```

by adding the price of chicken to the price of wine.

Note: Currently, the compiler would complain if you tried to use a *non-integral* type with the *+ operator*.

In C++, You can "reprogram" this operation by giving the + operator a new meaning. To do this, we need to create a function to *overload* the + operator:

```
float operator+( MenuItem item1, MenuItem item2 )
{
return( item1.GetPrice() + item2.GetPrice() );
}
```

Notice the name of this new method. Any method whose name follows the form:

```
operator<C++ operator>
```

is said to *overload* the **specified operator**. When you overload an operator, you're asking the compiler to call your function *instead* of interpreting the operator as it normally would.

4.1.1 Calling an Operator Overloading Function

When the compiler calls an overloading function, it maps the operator's operands to the function's parameters. For example, suppose the function

```
float operator+( MenuItem item1, MenuItem item2 )
{
return( item1.GetPrice() + item2.GetPrice() );
}
```

is used to overload the `+` operator. When the compiler encounters the expression `chicken + wine` it calls `operator+()`, passing `chicken` as the first parameter and `wine` as the second parameter. `operator+()`'s `return` value is used as the result of the expression.

It is important to note that:

- The number of operands taken by an operator determines the number of parameters passed to its overloading function.

For example, a function designed to overload a *unary* operator takes a **single** parameter; a function designed to overload a *binary* operator takes **two** parameters.

4.1.2 Operator Overloading Using a Member Function

You can also use a member function to overload an operator. For example, the function

```
float MenuItem::operator+( MenuItem item )
{
return( GetPrice() + item.GetPrice() );
}
```

overloads the + operator and performs pretty much the same function as the previous example. The difference lies in the way a member function is called by the compiler.

When the compiler calls an overloading member function, it uses the first operand to call the function and passes the remainder of the operands as parameters. So with the function just given in place, the compiler handles the expression

`chicken + wine`

by calling `chicken.operator+()`, passing `wine` as a parameter, as if you had made the following call:

```
chicken.operator+( wine )
```

Again, the value returned by the function is used as the result of the expression.

4.1.3 Multiple Overloading Functions

The previous example brings up an interesting point. What will the compiler do when it encounters *several* functions that overload the same operator?

For example, both of the following functions overload the + operator:

```
float operator+( MenuItem item1, MenuItem item2 )

float MenuItem::operator+( MenuItem item )
```

If both are present, which one is called?

The answer to this question is, **neither**:

- The compiler will not allow you to create an ambiguous overloading situation.
  - You **cannot** overload an operator with similar type operands.
- You can create several functions that overload the same operator, however:
  - You might create one version of operator+() that handles MenuItems and another that allows you to add two arrays together.
  - The compiler chooses the proper overloading function based on the types of the operands.

4.1.4 An Operator Overloading Example

Here's an example that illustartes manu of the above concepts.

We will override the + and *= operators so that can deal with time arithmetic: we'll declare a `Timeclass` and use it to store a length of time specified in hours, minutes, and seconds. Then, we'll overload the + and *= operators and use them to add two times together and to multiply a time by a specified value.

The code listing for `time.cpp` now follows:

```
#include <iostream.h>


//---------------------------------------  Time

class Time
{
// Data members...
private:
short hours;
short minutes;
short seconds;

// Member functions...
void NormalizeTime();
public:
Time();
Time( short h, short m, short s );
void Display();
Time operator+( Time &aTime );
void operator*=( short num );
};

Time::Time()
{
seconds = 0;
minutes = 0;
hours = 0;
```

```
}

Time::Time( short h, short m, short s )
{
seconds = s;
minutes = m;
hours = h;

NormalizeTime();
}

void Time::NormalizeTime()
{
hours += ((minutes + (seconds/60)) / 60);

minutes = (minutes + (seconds/60)) % 60;

seconds %= 60;
}

void Time::Display()
{
cout << "(" << hours << ":" << minutes
<< ":" << seconds << ")\n";
}

Time Time::operator+( Time &aTime )
{
short h;
short m;
short s;

h = hours + aTime.hours;
m = minutes + aTime.minutes;
s = seconds + aTime.seconds;

Time tempTime( h, m, s );
```

```
return tempTime;
}

void Time::operator*=( short num )
{
hours *= num;
minutes *= num;
seconds *= num;

NormalizeTime();
}


//---------------------------------------  main

int main()
{
Time firstTime( 1, 10, 50 );
Time secondTime( 2, 24, 20 );
Time sumTime;

firstTime.Display();
secondTime.Display();

cout << "---------\n";

sumTime = firstTime + secondTime;
sumTime.Display();

cout << "*      2\n";
cout << "---------\n";

sumTime *= 2;
sumTime.Display();

return 0;
}
```

The output of the program is:

```
:
(1:10:50)
(2:24:20)
--------
(3:35:10)
* 2
--------
(7:10:20)
```

First, we declare the `Time` class, which is used to store time in `hours`, `minutes`, and `seconds`. We also declare the member functions, The first, `NormalizeTime()`, is declared as `private` whilst the rest of the member functions are declared to be `public`.

- The function, `NormalizeTime()`, converts any overflow in the seconds and minutes data members; for example, 70 seconds is converted to 1 minute and 10 seconds. `NormalizeTime()` will only be used from within the Time class. Since we're not planning on deriving any classes from Time, we've left it as private.

- The two constructors: `Time(), Time( short h, short m, short s )`. The first takes no parameters and is used to create a `Time` object with all three data members set to 0 (You'll see why later on in the code). The second `Time` constructor uses its three parameters to initialize the three Time data members and then calls `NormalizeTime()` to resolve any overflow.

- The `Display()` displays the time stored in the current object, `operator+()` overloads the `+` operator, and `operator*=()` overloads the `*=` operator.

- The overload operator `operator+()` is called when the + operator is used to add two `Time` objects together. The first operand is used as the current object, and the second operand corresponds to the parameter `aTime`. Notice that `aTime` is declared as a *reference parameter*.This code would also work if `aTime` were declared without the &. Without the &, the compiler would create a copy of the parameter to pass in to `operator+()`. Since C++ passes its parameters on the stack, this

could cause a problem if the parameter was big enough. With the &, `aTime` is a reference to the object passed in as a parameter.

The function then takes the hours, minutes, and seconds data members of the two objects and adds the together (stored in the local variables `h, m`, and `s`

A new `Time` object `tempTime` is created using `h, m`, and `s`. Finally, we return the newly created object. Since we are not using a reference, the compiler will make a copy of `tempTime`, then return the copy. The compiler is responsible for destroying this copy, so you don't have to worry about it.

- The other overload operator `operator*=()` is called when the `*=` operator is used to multiply a `Time` object by a constant. Notice that `operator*=()` does not return a value because the multiplication is performed *inside* the `Time` object that appears as the first operand. Each of the `Time` object's data members is multiplied by the specified `short num`

  Since we are not creating a new `Time` object, `NormalizeTime()` is called to fix any overflow problems that may have just been caused:

In general, your overloading functions return a value if it makes sense for the operator being overloaded. If the operator includes an =, chances are you'll make your changes in place and won't return a value, as we did with `operator*=()`. If the operator doesn't include an =, you'll most likely return a value, as we did with operator+().

Before you make the decision, build a few expressions using the operator under consideration. Do the expressions resolve to a single value? If so, then you want your overloading function to return a value.

The `main()` function starts by defining two Time objects (the values in parentheses represent the hours, minutes, and seconds, respectively). A third `Time` object, `sumTime`, is created, this time via a call to the `Time` constructor that doesn't take any parameters.

`Display()` is called to display the data members of the two Time objects, and then a line is drawn under the two Times bfore the results of the + and *= operators are displayed

4.2 A Few Overload Restrictions

Therw ae a few restrictions. First, you can only overload C++'s built-in operators with some restrictions (see below) This means that you can't create any new operators. You can't suddenly assign a new meaning to the letter `z`, for example.

You can overload these operators.

```
+ - * / %
^ & | ~ !
, = < > <= >=
== != && ||
++ -- += -= *= /=
%= ^= "&= |=
<<= >>=
[] () -> ->*
new delete
\end{vebatim}



However,  you {\bf cannot} overload
these operators.

\begin{verbatim}
. .* :: ?: sizeof()
```

Note that you **can't change** the way an operator works with a predefined type. For example, you can't write your own `operator()` function to add two `int`s together.

The following rule of thumb for is worth remembering.

If you want the compiler to even consider calling your overloading function, either

- make the function a class member function, *or else*
- make one of its parameters an object.

Remember, the compiler will complain if you write an `operator()` function designed to work solely with C++'s built-in types.

Also note: when you overload the `++` and `--` operators, you'll have to provide two versions of the operator function,

- one to support *prefix* notation, and
- one to support *postfix* notation.

The compiler distinguishes between the two by checking for a dummy int parameter. Prefix version of your operator function shouldn't have any parameters, while the postfix version takes a single `int`.

Here's an example of a prefix and postfix `++` overloading operator for the Time class from our last program.

First, the prefix operator function:

```
Time operator++()
{
*this = *this +1;
return *this;
}
```

Now here's a version of the postfix operator function:

```
Time operator++( int )
{
Time aTime = *this;
*this = *this + 1;
return aTime;
}
```

Notice the unused int parameter in the postfix `operator++()` function. That's how the compiler identifies this function as postfix.

You also **cannot change** an operator's precedence by overloading it. If you want to force an expression to be evaluated in a specific order, you must use *parentheses*.

Overloading functions cannot specify default parameters. This restriction makes sense since a function with default parameters can be called with a variable number of arguments.

For example, you could call the function

```
MyFunc( short a=0, short b=0, short c=0 )
```

using anywhere from zero to three arguments. If an `operator()` function allowed default parameters, you'd be able to use an operator without any operands! If you did that, how would the compiler know which overloading function to call.

You **cannot change** the number of operands handled by an operator. For example, you couldn't make a binary operator unary.

4.3 Mutiple Overloaded Operations

Earlier in the chapter, we looked at a function that overloaded the + operator and was designed to add the price of two MenuItems together:

```
float operator+( MenuItem item1, MenuItem item2 )
{
return( item1.GetPrice() + item2.GetPrice() );
}
```

When the compiler encountered an expression like

```
chicken + wine
```

where both `chicken` and `wine` were declared as `MenuItems`, it called `operator+()`, which passed the two operands as parameters. The `float` produced by adding both prices together was returned as the result of the expression.

What happens when the compiler evaluates an expression like

```
chicken + wine + dessert
```

This expression seems innocent enough, but look at it from the compiler's viewpoint. First, the subexpression

```
chicken + wine
```

is evaluated, resolving to a `float`. Next, this `float` is combined with `dessert` in the expression:

```
<float> + dessert
```

What does the compiler do with this expression?

We designed an overloading function that handles the + operator when its operands are both `MenuItems`, but we don't have one that handles a `float` as the first operand and a MenuItem as the second operand.

Now take a look at the following expression:

```
chicken + (wine + dessert)
\begin{verbatim}

First, the compiler evaluates the subexpression:

\begin{verbatim}
(wine + dessert)
```

resolving it to a `float`. That leaves us with the expression

```
chicken + <float>
```

Once again, we designed an `operator+()` function that handles + and two `MenuItems`, but we don't have one that handles a MenuItem as the first operand and a float as the second operand.

4.3.1 Overloading an Overloading Function

As you can see, you frequently need more than one version of the same `operator()` function. To accomplish this task, you use a technique introduced previously, *function overloading*.

Just as with any other function, you can overload an `operator()` function by providing more than one version, each with its own unique signature.

Remember, a function's signature is based on its para-meter list and not on its return value.

How Many Versions Are Needed?

Figuring out how many versions of an `operator()` function to provide is actually pretty straightforward. Start by making a list of the number of possible types you want to allow for each of the operator's operands. Don't forget to include the type returned by your `operator()` function.

In the previous example, we wanted `operator+()` to handle a float or a MenuItem as either operand, which yields the possibilities shown below:

```
float  + float
float  + MenuItem
MenuItem + float
MenuItem + MenuItem
```

As pointed out earlier, you can't create an `operator()` function based solely on built-in types. Fortunately, the compiler does a perfectly fine job of adding two `float`s together.

With this first case taken care of by the compiler, we're left to construct the remaining three `operator+()` functions.

Our next example program, `menu.cpp`, uses function overloading to do just that.

4.3.2 `menu.cpp`:An Overloader Overloading Example

The code listing for `menu.cpp` is as follows:

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

const short kMaxNameLength = 40;


//---------------------------------------  MenuItem

class MenuItem
{
private:
float price;
char name[ kMaxNameLength ];

public:
MenuItem( float itemPrice, char *itemName );
float GetPrice();
float operator+( MenuItem item );
```

```
float operator+( float subtotal );
};

MenuItem::MenuItem( float itemPrice, char *itemName )
{
price = itemPrice;
strcpy( name, itemName );
}

float MenuItem::GetPrice()
{
return( price );
}

float MenuItem::operator+( MenuItem item )
{
cout << "MenuItem::operator+( MenuItem item )\n";

return( GetPrice() + item.GetPrice() );
}

float MenuItem::operator+( float subtotal )
{
cout << "MenuItem::operator+( float subtotal )\n";

return( GetPrice() + subtotal );
}


float operator+( float subtotal, MenuItem item );
// I added the previous line, cause CodeWarrior reports (correctly)
// that there was no prototype for float operator+().
// Now there IS a prototype! Comment out the line
// if you want to see the warning message -- Dave Mark, 10/20/95


//--------------------------------------  operator+()
```

```
float operator+( float subtotal, MenuItem item )
{
cout << "operator+( float subtotal, MenuItem item )\n";

return( subtotal + item.GetPrice() );
}


//-------------------------------------  main()

int main()
{
MenuItem chicken( 8.99, "Chicken Jalfrezi" );
MenuItem wine( 2.99, "Rioja by the Glass" );
MenuItem dessert( 3.99, "Fresh Mangoes" );
float total;

total = chicken + wine + dessert;

cout << "\nTotal: " << total
<< "\n\n";

total = chicken + (wine + dessert);

cout << "\nTotal: " << total;

return 0;
}
```

The output of this program is:

```
MenuItem::operator+( MenuItem item )
operator+( float subtotal, MenuItem item )
Total: 15.969999
MenuItem::operator+( MenuItem item )
MenuItem::operator+( float subtotal )
Total: 15.969999
```

`menu.cpp` starts with two include files (<`iostream.h`¿ and <`string.h`>)and a single constant, `const short kMaxNameLength = 40`.

Next, the `MenuItem` class is declared. The `MenuItem` class contains two data members.`price` lists the price of the item while `name` contains the item's name as it might appear on a menu. Notice that both data members are marked as `private`, which shouldn't be a problem since we won't be deriving any new classes from `MenuItem`.

The `MenuItem` class features four member functions. The constructor, `MenuItem()`, initializes the `MenuItem` data members; the `GetPrice()` function returns the value of the `price` data member.

The two `operator+()` functions handle the cases where a `MenuItem` object appears as the first operand to the + operator. If the second operand is also a `MenuItem`, the first of the two functions is called; if the second operand is a `float`, the second function is called:

The `MenuItem()` constructor copies its first parameter into the `price` data member, and then it uses `strcpy()` to copy the second parameter into the name data member.

Note that we have to write three versions of `operator+()` even though only two are declared in the class:

- The first version of `operator+()` handles expressions of the form

  ```
  <MenuItem> + <MenuItem>
  ```

- The second version of operator+() handles expressions of the form

  ```
  <MenuItem> + <float>
  ```

- The third version of `operator+()` is, *by necessity*, not a member function of any class. To understand why this is so, take a look at the expressions this version of `operator+()` is designed to handle:

  ```
  <float> + <MenuItem>
  ```

  As mentioned earlier, the compiler uses the first operand to determine how the overloading `operator()` function is called. If the first parameter is an object, that object is used to call the `operator()` function

and all other operands are passed to the function as parameters. If the first parameter is not an object, the compiler's list of candidate overloading functions is reduced to the program's nonclass `operator()` functions. Once a matching function is located, the compiler calls it, passing all of the operands as parameters.

`main()` declares three `MenuItem` objects, initializing each with a price and a name. `main()` also declares a `float total` used to hold the result of our Menu addition. Next, the three `MenuItems` are added together, the result stored in `total = chicken + wine + dessert`, and the printed.

When the compiler encounters the expression

```
chicken + wine + dessert
```

it first processes the sub-expression

```
chicken + wine
```

Since we're adding two `MenuItems` together, the compiler calls the first of our three `operator+()` functions, as shown by the first line of output (`MenuItem::operator+( MenuItem item )`).

Next, this subtotal is used to process the remainder of the expression:

```
<subtotal> + dessert
```

Since we're now adding a float to a MenuItem, the compiler calls the third `operator+()` function, as shown by the next line of output (`operator+( float subtotal, MenuItem item )`)

Once the calculations are complete, the total is printed

Then, the three MenuItems are added together *again*, this time with *the addition of parentheses* wrapped around the last two operands:

```
total = chicken + (wine + dessert);
```

These parentheses force the compiler to start by evaluating the sub-expression

```
(wine + dessert)
```

Once again, we're adding two `MenuItems` together, as shown by the next line of output (`MenuItem::operator+( MenuItem item )`).

Next, this subtotal is used to process the remainder of the expression:

```
chicken + <subtotal>
```

Since we're now adding a `MenuItem` to a `float`, the compiler calls the second `operator+()` function, as shown by the following line of output (`MenuItem::operator+( float subtotal )`)

Finally, the total is printed a second time

4.4 Some Special Cases

The remainder of this chapter is dedicated to a few special cases. Specifically, we'll focus on writing operator() functions that overload the `new, delete, (), [], ->`, and `=` operators.

One characteristic shared by each of these operators is that they can only be overloaded by a nonstatic class member function. Basically, this means that you won't be using the non-class `operator()` function strategy from our previous example for any of the operators in this section.

4.4.1 Overloading `new` and `delete`

There are two ways you can overload `new` and `delete`.

- You can create two member functions named operator `new()` and operator `delete()` as part of your class design. You might do this if you wanted to implement your own memory management scheme for a specific class.
- You can overload the global new and delete operators by providing operator `new()` and operator `delete()` functions that are not members of a class. You might do this if you wanted new and delete to always initialize newly allocated memory.

Whatever your reasons for overloading `new` and `delete`, proceed with caution. No matter how you do it, once you overload new and delete, you are taking on a big responsibility, one that can get you in deep trouble if you don't handle things properly.

An operator `new` Example

Here's a small example you can use as the basis for your own `new` and `delete` `operator()` functions.

```
#include <iostream.h>


//---------------------------------------  Blob

class Blob
{
public:
void *operator new( size_t blobSize );
void operator delete( void *blobPtr, size_t blobSize );
};

void *Blob::operator new( size_t blobSize )
{
cout << "new: " << blobSize << " byte(s).\n";

return new char[ blobSize ];
}

void Blob::operator delete( void *blobPtr, size_t blobSize )
{
cout << "delete: " << blobSize << " byte(s).\n";

delete [] blobPtr;
}


//---------------------------------------  main()

int main()
{
Blob *blobPtr;

blobPtr = new Blob;
```

```
delete blobPtr;

return 0;
}
```

The output of this program is:

```
new: 2 byte(s).
delete: 2 byte(s).
```

`new.cpp` defines a class named Blob, which doesn't do much, but it does contain overloading functions for `new` and `delete`.

There are lots of details worth noting in the `new` and `delete operator()` functions.

- Notice the space between the words `operator` and `new` and between `operator` and `delete`. *Without* the space, the compiler might think you were creating a function called `operatornew()` —a perfectly legal C++ function name.

- The operator `new()` returns a `void *`. This is required. In general, your version of `new` will return a pointer to the newly allocated object or block of memory. If your memory management scheme calls for relocatable blocks, you might want to return a handle (pointer to a pointer) instead. The choice is yours.

- The operator new function must take at least one parameter of type `size_t`. The value for this parameter is provided automatically by the compiler and specifies the size of the object to be allocated. Any parameters passed to new will follow the `size_t` in the parameter list.

- The operator `delete` function never returns a value and **must** be declared to return a `void`. `delete` always takes at least one parameter, a pointer to the block to be deleted. The second parameter, a `size_t`, is optional. If you provide it, it will be filled with the size, in bytes, of the block pointed to by the first parameter.

  Sometimes the size passed as the second parameter to operator `delete()` isn't quite what you expected. If the pointer being deleted is a pointer

to a base class yet the object pointed to belongs to a class derived from the base class, the second parameter to operator `delete()` will be the *size of the base class*.

There is an exception to this rule. If the base class's destructor is virtual, the size parameter will hold the proper value, the size of the object actually being deleted.

`main()` creates a `new Blob` and then `delete`s it. When the `Blob` is created, the overriding `new` is called. When the object is deleted, the overloaded version of delete is called.

4.4.2 Overloading ( )

The next special case is the function that overloads the `()` operator, also known as the function *call* operator. One reason to overload the function call operator is to provide a shorthand notation for accessing an object's critical data members. As mentioned earlier, `()` can only be overloaded by a nonstatic class member function.

Consider the following an example, `call.cpp`:

```
#include <iostream.h>


//--------------------------------------  Item

class Item
{
private:
float price;

public:
Item( float itemPrice );
// Note that the default value for taxRate has been removed. The book
// uses a default value of 0. As it turns out, section 13,5 of
// the ANSI C++ draft states that an operator function cannot have
// default arguments.
//
// In this version, I just removed the default value for taxRate
```

```
// and changed the call stimpyDoll() to stimpyDoll( 0 ) to
// achieve the same result. Thanks to Khurram Quereshi for figuring
// this one out!! -- Dave Mark, 10/26/95
float operator()( float taxRate );
};

Item::Item( float itemPrice )
{
price = itemPrice;
}

float Item::operator()( float taxRate )
{
return( ((taxRate * .01) + 1) * price );
}


//--------------------------------------  main()

int main()
{
Item stimpyDoll( 36.99 );

cout << "Price of Stimpy doll: $" << stimpyDoll( 0 );
cout << "\nPrice with 4.5% tax:  $" << stimpyDoll( 4.5 );

return 0;
}
```

The output is as follow:

```
Price of Stimpy doll: $36.990002
Price with 4.5% tax: $38.654552
```

`call.cpp` starts by defining an `Item` class. An `Item` object represents an item for sale at Uncle Ren's Toy-o-rama.

`Item` features a single data member, `price`, and two member functions, the `Item()` constructor and a function designed to overload the `()` call operator

The `operator()()` function may look odd, but the syntax using two pairs of parentheses is correct:

```
float Item::operator()( float taxRate )
```

The first pair of parentheses designates the operator being overloaded; the second pair surrounds any parameters being passed to the function. In this case, one parameter, `taxRate`, is specified. Notice that `taxRate` has a default value of 0. You'll see why in a minute.

The `operator()()` function takes the specified `taxRate` and applies it to the `Item`'s price, returning the `Item`'s total (`((taxRate * .01) + 1) * price )`)

Since the function call operator can only be overloaded by a class member function, the previous reference to price refers to the data member of the object used in combination with the call operator.

`main()` starts by creating an `Item` object. Here's where the call overload comes into play:

```
cout << "Price of Stimpy doll: $" << stimpyDoll();
```

By taking advantage of the default parameter, the function call

```
stimpyDoll()
```

returns `stimpyDoll`'s price. We could have accomplished the same thing by coding

```
stimpyDoll.price
```

or

```
stimpyDoll( 0 )
```

Next, we use the same function to calculate the cost of the doll with 4.5% tax included:

```
cout << "\nPrice with 4.5% tax: $"
<< stimpyDoll( 4.5 );
```

Once again, we take advantage of the overloaded function call operator. This time, we provide a parameter. Notice that the same overloading function is used for two different (though closely related) purposes.

The key to properly overloading the function call operator is to use it to provide access to a key data member. If your object represents a character string, you might overload `()` to provide access to a substring, using a pair of parameters to provide the starting position and length of the substring.

Another strategy uses `()` as an iterator function for accessing data kept in a sequence or list. Each call to `()` bumps a master pointer to the next element in the list and returns the new data element. No question about it, the function call operator is a useful operator to overload.

4.4.3 Overloading `[]`

Another useful operator to overload is `[]`, also known as the *subscript* operator. Although it can be used for other things, `[]` is frequently overloaded to provide range checking for arrays. You'll see how to do this in a moment.

The subscript overloading syntax is similar to that of the function call operator. In the statement

```
myChar = myObject[ 10 ];
```

the `[]` overloading function belonging to the same class as `myObject` is called with a single parameter, 10. The value returned by the function is assigned to the variable `myChar`.

On the flip side of the coin, the $\widehat{[]}$ overloading function must support a `[]` expression on the left side of the assignment state-ment, like so:

```
myObject[ 10 ] = myChar;
```

The next example program, `subscript.cpp`, shows you how to properly overload `[]`:

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

const short kMaxNameLength = 40;
```

```
//---------------------------------------  Name

class Name
{
private:
char nameString[ kMaxNameLength ];
short nameLength;

public:
Name( char *name );
void operator()();
char &operator[]( short index );
};

Name::Name( char *name )
{
strcpy( nameString, name );
nameLength = strlen( name );
}

void Name::operator()()
{
cout << nameString << "\n";
}

char& Name::operator[]( short index )
{
if ( ( index < 0 ) || ( index >= nameLength ) )
{
cout << "index out of bounds!!!\n";
return( nameString[ 0 ] );
}
else
return( nameString[ index ] );
}


//---------------------------------------  main()
```

```
int main()
{
Name pres( "B. J. Clinton" );

pres[ 3 ] = 'X';
pres();

pres[ 25 ] = 'Z';
pres();

return 0;
}
```

The output is:

```
B. X. Clinton
index out of bounds!!!
Z. X. Clinton
```

`subscript.cpp` starts by defining a `Name` class. The `Name` class is fairly simple. It is designed to hold a `NULL`-terminated string containing a person's name as well as a short containing the length of the string. The member functions include a constructor as well as two operator overloading functions. One function overloads `[]` the other overloads `()`:

The constructor copies the provided string to the nameString data member and places the length of the string in the nameLength data member.

The `()` operator overloading function simply prints the character string in `nameString`:

The `[]` operator overloading function takes a single parameter, an index into the character string. Notice the unusual return type. By specifying a char reference as a return type, the function ensures that the `[]` operator can appear on either side of an assignment statement. Essentially, an expression such as

```
myObject[0]
```

is turned into a `char` variable containing the character returned by the `[]` overloading function:

```
char& Name::operator[]( short index )
```

Here's the real advantage to overloading the `[]` operator. Before you access the specified character, you can first do some bounds checking, making sure the character is actually in the character string! If the specified index is out-of-bounds, we print a message and point to the first character in the string.

`main()` first, creates a Name object. Next, the fourth character in the string is replaced by the character `X`. When `pres()` is called, the modified string is displayed: `B. X. Clinton`

Then, the character `Z` is placed well out-of-bounds and the string is displayed again.

The `[]` overloading function lets you know that the specified index is out-of-bounds and the assignment is performed on the first character of the string instead:

```
index out of bounds!!!
Z. X. Clinton
```

4.4.4 Overloading $->$

Next on the special cases list is the `->` operator, also known as the member access operator. Like the other operators presented in this section, overloading `->` provides a shorthand notation that can save you code and add an elegant twist to your program.

When the compiler encounters the `->` operator, it checks the type of the left-hand operand. If the operand is a pointer, `->` is evaluated normally. If the operand is an object or object reference, the compiler checks to see whether the object's class provides an `->` overloading function.

If no `->` overloading function is provided, the compiler reports an error, since the `->` operator requires a pointer, not an object. If the -¿ overloading function is present, the left operand is used to call the overloading function. When the overloading function returns, its return value is substituted for the original left operand, and the evaluation process is repeated. When used this way, the `->` operator is known as a smart pointer.

If these rules sound confusing, hold on. The next example, `smartPtr.cpp`, should explain things:

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

const short kMaxNameLength = 40;


//---------------------------------------  Name

class Name
{
private:
char first[ kMaxNameLength ];
char last[ kMaxNameLength ];

public:
Name( char *lastName, char *firstName );
void DisplayName();
};

Name::Name( char *lastName, char *firstName )
{
strcpy( last, lastName );
strcpy( first, firstName );
}

void Name::DisplayName()
{
cout << "Name: " << first << " " << last;
}


//---------------------------------------  Politician

class Politician
{
private:
Name *namePtr;
short age;
```

```
public:
Politician( Name *namePtr, short age );
Name *operator->();
};

Politician::Politician( Name *namePtr, short age )
{
this->namePtr = namePtr;
this->age = age;
}

Name *Politician::operator->()
{
return( namePtr );
}


//--------------------------------------  main()

int main()
{
Name myName( "Clinton", "Bill" );
Politician billClinton( &myName, 46 );

billClinton->DisplayName();

return 0;
}
```

The output is, simply:

```
Name: Bill Clinton
```

`smartPtr.cpp` defines two classes. The `Name` class which holds two zero-terminated strings containing a person's first and last names. and the `Politician` class which represents a politician. To keep things simple, the info is limited to the politician's age and a pointer to a Name object containing the

politician's name. The `Politician` class also contains a member function designed to overload the `->` operator. The function returns a pointer to the politician's Name object (the fact that it returns a pointer is key):

`main()` embeds a last and first name into a `Name` object and then uses that object to create a new `Politician` object (so far, no big deal)

There are several problems here. First, `billClinton` is an object and not a pointer, yet it is used with the `->` operator. Second, the member function `DisplayName()` is not a member of the `Politician` class. How can it be called directly from a `Politician` object?

Basically, the `->` overloading function is doing its thing as a smart pointer by bridging the gap between a `Politician` object and a `Name` member function. When the compiler encounters the `->` operator, it checks the type of the left operand. Since `billClinton` is not a pointer, the compiler checks for an `->` overloading function in the `Politician` class.

When the overloading function is found, it is called, using `billClinton` as the current object. The function returns a pointer to a `Name` object. The compiler substitutes this return value for the original, yielding

```
namePtr->DisplayName()
```

The compiler again checks the type of the left operand. This time, the operand is a pointer and the `->` operator is evaluated normally. The `namePtr` is used to call the `Name` function `DisplayName()`, resulting in the output of `Name:  Bill Clinton`

As you can see, overloading the `->` operator provides a shortcut that allows you to run a direct line between two different classes. You can take this model one step further by supposing that the `->` overloading function returns a `Name` object rather than a pointer to a `Name` object. The compiler then substitutes the `Name` object in the original expression and reevaluates:

```
myName->DisplayName();
```

Once again, since the left operand is an object and not a pointer, the left operand's class is examined in search of another `->` overloading function. This substitution and call of `->` overloading functions is repeated until a pointer is returned (the end of the chain is reached). Only then is the `->` operator evaluated in its traditional form. You can use this technique to walk along a chain of objects. Each `->` overloading function evaluates some

criteria, returning an object if the search should con-tinue or a pointer if the end condition has been met.

This is pretty Mind Blowing stuff.

4.4.5 Overloading =

The last of the special cases is the `operator=()` function.

Why overload the `=` *operator?*

To best understand why, take a look at what happens when you assign one object to another.

Suppose you define a String class, like this:

```
class String
 {
  private:
    char *s;
    short stringLength;

  public:
    String( char *theString );
 };
```

The data member `s` points to a `NULL`-terminated string. The data member stringLength contains the length of the string. The constructor `String()` initializes both data members. Notice that no memory has been allocated for `s`. This is done inside the constructor.

Now suppose you create a pair of Strings, like this:

```
String source( "from" );

String destination( "to" );
```

And then, you assign one of the String objects to the other, like this:

```
destination = source;
```

What happens?

As it turns out, the = operator copies one object to another by a process called memberwise assignment. Basically, this means that each data member within one object is copied, one at a time, to the corresponding data member in the receiving object.

The trouble with memberwise assignment is in the way it deals with allocated memory, such as you'd find with a null-terminated character string. When one `(char *)` is copied to another, the address stored in the `(char *)` is copied, not the data pointed to by the address. Once the statement

```
destination = source;
```

executes, both Strings point to the same `NULL`-terminated string in memory. The *default* = operator isn't smart enough to allocate the appropriate amount of new memory and then use `strcpy()` to make a copy of the string. That's where `operator=()` comes in.

If you want the ability to assign the contents of one object to another, and the objects contain allocated memory, you'll have to write a smart = overloading function that knows how to do it right.

He is an `operator=()` example, `equals.cpp;`

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


//---------------------------------------  String

class String
{
private:
char *s;
short stringLength;

public:
String( char *theString );
~String();
void DisplayAddress();
String &operator=( const String &fromString );
```

```
};

String::String( char *theString )
{
stringLength = strlen( theString );
s = new char[ stringLength + 1 ];

strcpy( s, theString );
}

String::~String()
{
delete [] s;
}

void String::DisplayAddress()
{
// I added an extra line to the DisplayAddress function
// because both sets of address in the program were
// turning out to be the same. I now print out the string
// along with the string address. Now when you run the program,
// you can see that the first time you print captain and doctor,
// they contain different strings, but the second time, they
// contain the same string, even though their addresses
// didn't change.
// Sorry for any confusion -- Dave Mark 10/31//95
cout << "String address: " << (unsigned long)s << "\n";
cout << "       content: " << s << "\n\n";
}

String &String::operator=( const String &fromString )
{
delete [] s;

stringLength = fromString.stringLength;

s = new char[ stringLength + 1 ];
```

```
strcpy( s, fromString.s );

return( *this );
}


//---------------------------------------  main()

int main()
{
String captain( "Picard" );
String doctor( "Crusher" );

captain.DisplayAddress();
doctor.DisplayAddress();

cout << "-----\n";

doctor = captain;

captain.DisplayAddress();
doctor.DisplayAddress();

return 0;
}
```

The output of this program is:

```
String address: 3259462
String address: 3259472
----
String address: 3259462
String address: 3261024
```

`equals.cpp` starts by defining the `String` class described earlier, with a few additions. The constructor still allocates the memory for the specified string, but now several new functions are added:

```
public:
String( char *theString );
~String();
void DisplayAddress();
String &operator=( const String &fromString );
```

The constructor starts by calculating the length of the specified string, storing the result in `stringLength`. Next, `new` is used to allocate the proper amount of memory (the extra byte is for the `NULL` terminator at the end of the string). Finally, `strcpy()` is called to copy the source string to the data member `s`.

If `s` is declared as an array of fixed size, instead of as a dynamic string pointer, memberwise initialization works just fine since the memory for the array is part of the object itself. Since s points to a block of memory outside the object, memberwise initialization passes it by.

The String destructor (`String()`) uses delete to destroy the array of chars pointed to by s by calling `delete [] s`

The member function `DisplayAddress()` provides a shorthand way of displaying the address of the first byte of a string:

```
cout << "String address: " << (unsigned long)s <<
```

The `=` overloading function, just like `operator[]()`, this function must return an `l-value`. In this case, we return a reference to a `String` object. We also take a `String` reference as a parameter. Since you can only assign an object to another object of the same class, the type of the return value will always agree with the type of the parameter. `const` in the parameter declaration just marks the parameter as *read only*. `operator=()` starts by freeing up the memory occupied by the old string. Next, the new value for the data member stringLength is copied from the source String. After that, `new` is used to allocate a block for the new string, and `strcpy()` is used to copy the source string into `s`.

Since this is a pointer to the current object, `*this`is the object itself. We return `*this` to satisfy our need to return an `l-value`.

`main()` puts everthing to the test. First, two `String` objects are created and initialized. Next, the address of each `String`'s string is displayed, using the overloaded `()` operator. Then, the object `captain` is assigned to the object `doctor`, and the addresses of the two text strings are again displayed

If the = operator is not overloaded, the address of the `captain` string simply copies into the `doctor` object's s data member and both addresses are the same.

To that all this work is necessary: If you comment out the `operator=()` function (every single line, not just the insides) as well as its declaration inside the String class declara tion, and run the program again. Without the `operator=()` function, the `String` destructor would try to delete the same block of memory twice!

Chapter 5

The `iostream`

The iostream's insertion operator (<<) has been used for all of our output and iostream's extraction operator (>>) has been used for all of our input. While these operators serve us well, there's much more to `iostream` than has been demonstrated so far.

There are 4 include files <iostream.h>, <fstream.h>, <iomanip.h¿, and <strstrea.h> associated with the complete C++ `iostream`. So far we have only used the basic <iostream.h> header.

The `iostream` is a powerful extension of the C++ language. As we will see shortly, we can easily customize iostream so that the >> and << operators recognize your own personally designed data structures and classes. The `iostream` can also be used to write to and read from files or even character arrays. .

5.1 The Character-Based Interface

The `iostream`'s basic unit of currency is the character. Before a number is written to a file, it is converted to a series of chars. When a number is read from the console, it is read as a series of chars and then, if necessary, converted to the appropriate numerical form and stored in a variable.

The `iostream` was designed to support a character-based user interface. As characters are typed on the user's keyboard, they appear on the console. When your program has something to say to the user, it uses iostream to send a stream of characters to the console. If you plan to write programs for environments such as a graphical version of Unix (Motif, X-window), the

Macintosh or MS Windows, you'll probably do all your user-interface development using class libraries that come with your development environment. The `iostream` doesn't know a thing about pull-down menus, windows, or even a mouse, but as you'll see, it's more than a library of user-interface routines.

5.1.1 The `iostream` Classes

Even if your user interface isn't character-based, the `iostream` still has a lot to offer. You can use the same mechanisms you'd use to manage your console I/O to manage your program's file I/O. The same methods you'd use to write a stream of characters to a file can be used to write those same characters to an array in memory. What links these disparate techniques is their common ancestry. The `iostream` library is built upon a set of powerful classes. The `iostream` base class is named `ios`. While you might not work directly with an `ios` object, you'll definitely work with `ios`' members as well as with classes derived from ios.

You've already started to work with two classes derived from `ios`. The `istream` class is designed to handle input from the keyboard. `cin` is an `istream` object that C++ automatically creates for you. The `ostream` class is designed to handle output to the console. `cout`, tt cerr, and `clog` are `ostream` objects that are also automatically created for you. As you've already seen, `cout` is used for standard output. `cerr` and `clog` are used in the same way as `cout`. They provide a mechanism for directing error messages.

Usually `cerr` is tied to the console, although some operating systems (*e.g* Unix) allow you to redirect `cerr`, perhaps sending the error output to a file or to another console. `cerr` is unbuffered which means that output sent to `cerr` appears immediately on the `cerr` device. `clog` is a buffered version of `cerr` and is not supported by all C++ development environments. To decide which error output vehicle to use (`clog` or `cerr`), consult your operating system manual.

5.1.2 The `istream` and `ostream` classes

Up to this point, your experience with iostream has centered on the extraction (>>) and insertion (<<) operators. For example, the following code reads in a number, stores it in a variable, and then prints out the value of the number:

```
short myNum;
cout << "Type a number: ";
cin >> myNum;
cout << "Your number was: " << myNum;
```

There are a couple of things worth noting in this example.

- `iostream` input and output *are buffered*. Just as in C, all input and all output are accumulated in buffers until either the buffers are filled or the buffers are flushed. On the input side, the buffer is traditionally flushed when a carriage return is entered. On the output side, the buffer is usually flushed either when input is requested or when the program ends. Later in the chapter, you'll learn how to flush your own buffers (how exciting!).

- >> eats up white space — >> ignores spaces and tabs in the input stream.

  If you're reading in a series of numbers, this works out pretty well. But if you're trying to read in a stream of text, you might want to preserve the white space interspersed throughout your input. Fortunately, `istream` offers some member functions that read white-space.

`get( )`

The `istream` member function `get()` reads a single character from the input stream. `get()` comes in three different flavors.

- The first version of `get()` takes a char reference as a parameter and returns a reference to an istream object:

  ```
  istream &get(char &destination);
  ```

  Since `get()` is an istream member function, you can use cin to call it (after all, `cin` is just an `istream` object):

  ```
  char c;
  cin.get( c );
  ```

This version of get() reads a single character from the input stream, writes the char into its char parameter (c), and then returns the input stream reference (cin). Since get() returns the input stream, it can be used in a sequence, as in the following example:

```
char c;
short myShort;
cout << "Type a char and a short: ";
cin.get( c ) >> myShort;
```

This code grabs the first character from the input stream and stores it in c. Next, the input stream is parsed for a short, and the short is placed in myShort.

- The second version of get() is declared as follows:

```
istream &get(char *buffer, int length, char
delimiter = '\n');
```

This version of get() extracts up to length - 1 characters and stores them in the memory pointed to by buffer. If the char delimiter is encountered in the input stream, the char is pushed back into the stream and the extraction stops. For example, the code:

```
char buffer[ 10 ];
cin.get( buffer, 10, '*' );
```

starts to read characters from the input stream. If a * is encoun tered, the extraction stops, the * is pushed back into the stream, and a NULL terminator is placed at the end of the string just read into buffer.

If no * is encountered, nine characters are read into buffer, and, again, buffer is NULL-terminated. Notice that get() reads only]
tt n - 1 characters, where n is specified as the second parameter; get() is smart enough to save one byte for the NULL terminator. If the third parameter is left out, this version of get() uses '\n' as the terminating character. This allows you to use get() to extract a full line of characters without overflowing your input buffer. For example, the code:

```
char buffer[ 50 ];
cin.get( buffer, 50 );
```

reads up to 49 characters or one line from the input stream, whichever is shorter. Either way, the string stored in `buffer` gets `NULL`-terminated.

- The third version of get() is declared as follows:

```
int get();
```

This version of `get()` reads a single character from the input stream and returns the character, cast as an int, as in the follow-ing example:

```
int c;
while ( (c = cin.get()) != 'q' )
cout << (char) c;
```

This code reads the input stream, one character at a time, until a `q` is read. Each character is echoed to the console as it is read. The third version of `get()` returns an `int` and not a `char` to allow it to return the end-of-file character. Typically, `EOF` has a value of `-1`. By returning an `int`, `get()` allows for 256 possible `char` values as well as for the end-of-file character.

Although `EOF` isn't particularly useful when reading from the console, we'll use this version of `get()` later to read the contents of a file.

`getline( )`

Another `istream` member function that you might find useful is `getline()`, which is prototypes by:

```
istream &getline(char *buffer, int length, char delimiter = '\n');
```

`getline()` behaves just like the second version of `get()`, but it returns the delimiter character instead of pushing it back into the input stream.

ignore( )

`ignore()` is used to discard characters from the input stream:

```
istream &ignore(int length = 1, int delimiter = EOF);
```

`ignore()` follows the same basic approach as `getline()`. It reads up to length characters from the input stream and discards them. This extraction stops if the specified delimiter is encountered. Notice that each of these parameters has a default value, which allows you to call `ignore()` without parameters. Here's an example:

```
char buffer[ 100 ];
cin.ignore( 3 ).getline( buffer, 100 );
cout << buffer;
```

This code drops the first three characters from the input stream and then reads the remainder of the first line of input into buffer. Next, the string stored in `buffer` is sent to the console. Notice that the value returned by `ignore()` is used to call `getline()`. This is equivalent to the following sequence of code (but shorter):

```
cin.ignore( 3 );
cin.getline( buffer, 100 );
```

peek( )

`peek( )` allows you to sneak a peek at the next character in the input stream *without* removing the character from the stream. It is prototyped by:

```
int peek();
```

Just like the third version of `get()`, `peek( )` returns an `int` rather than a `char`. This allows `peek( )` to return the end-of-file character, if appropriate, which makes `peek( )` perfect for peeking at the next byte in a file.

put( )

The ostream member function put() provides an alternative to the << operator for writing data to the output stream:

```
ostream &put(char c);
```

put() writes the specified character to the output stream. It then returns a reference to the stream, so put() can be used in a sequence. Here's an example:

```
cout.put( 'H' ).put( 'i' ).put( '!' );
```

As you might have guessed, the preceding line of code produces a Hi! message:

putback( )

putback() puts the specified char back into the input stream, making it the next character to be returned by the next input operation:

```
istream &putback(char c);
```

Note that c must be the last character extracted from the stream.

Since putback() returns an istream reference, it can be used in a sequence, similar to the example combining ignore() and getline() shown earlier.

seekg( ) and seekp( )

The istream member function seekg() gives you random access to an input stream:

```
istream &seekg(streampos p);
```

Call seekg() to position a stream's get pointer exactly where you want it.

A second version of seekg() allows you to position the get pointer relative to the beginning or end of a stream or relative to the current get position, this is defined by:

```
istream &seekg( streamoff offset, relative_to direction );
```

In this second version of `seekg()`, the second parameter is one of `ios::beg`, `ios::cur`, or `ios::end`.

The ostream member function `seekp()` gives you random access to an output stream:

```
ostream &seekp(streampos p);
```

Just like `seekg(), seekp()` allows you to position a stream's put pointer exactly where you want it. `seekp()` also comes in a second flavor:

```
ostream &seekp( streamoff offset, relative_to direction );
```

5.2 Some Useful Utilities

To aid you with your stream input and output operations, C++ provides a set of standard utilities that you may find useful (plain old ANSI C also provides these routines). To use any of the utilities described in this section, you must include the header file <`ctype.h`>.

Each of the thirteen functions takes an `int` as a parameter. The `int` represents an ASCII character. Two of the functions, `tolower()` and `toupper()`, map this character either to its lowercase or its uppercase ASCII equivalent. For example,

The remaining eleven functions return either 1 or 0, depend ing on the nature of the character passed in. The function

`isalpha()` returns 1 if its argument is a character in the range 'a' through 'z' or in the range 'A' through 'Z'.

The function `isdigit()` returns 1 if its argument is a character in the range '0' through '9'.

The function `isalnum()` returns 1 if its argu-ment causes either `isalpha()` or `isdigit()` to return 1.

The function `ispunct()` returns 1 if the character is a punc-tuation character. The punctuation characters are ASCII charac-ters in the ranges 33-47, 58-64, 91-96, and 123-126 (consult your nearest ASCII chart).

The function `isgraph()` returns 1 if its argument causes `isalpha(), isdigit()`, or `ispunct()` to return 1.

`islower()` returns 1 if the character is in the range 'a' through 'z'.

`isupper()` returns 1 if the character is in the range 'A' through 'Z'.
`isprint()` returns 1 if the character is a printable ASCII character.
`iscntrl()` returns 1 if the character is a control character.
`isspace()` returns 1 if the character has an ASCII value in the range 9-13 or if it has a value of 32 (space). Finally, isxdigit() returns 1 if the character is a legal hex digit (0-9, a-f, or A-F).

5.3 Reading Data from a File

The `ifstream` constructor comes in several varieties. The most widely used of these takes two parameters:

```
ifstream(const char* name, int mode=ios::in );
```

The first parameter is a `NULL`-terminated string containing the name of a file to be opened. The second describes the mode used to open the file. The legal modes are described in the table below:

Mode	Description
`ios::in`	Input allowed
`ios::out`	Output allowed
`ios::ate`	Seek to EOF at open
`ios::app`	Output allowed, append only
`ios::trunc`	Output allowed, discard existing contents
`ios::nocreate`	Open fails if file doesn't exist
`ios::noreplace`	Open fails if file does exist

They are declared as part of the ios class (defined in <`iostream.h`>). The default mode is `ios::in`, which opens the file for reading.

UNIX (and some other operating systems) support a third, optional parameter for `ifstream` (and for `ofstream` as well). The third parameter specifies the *protection level* used to open the file.

Since you'll most likely want to use the default mode of `ios::in` when you open a file for reading, you can leave off the last parameter when you create an ifstream object:

```
ifstream readMe( "My_File" );
```

This definition creates an `ifstream` object named `readMe`. Next, it opens a file named `My_File` for reading, attaching the open file to `readMe`.

ifstream objects have data members that track whether a file is attached to the stream and, if so, whether the file is open for reading. If a file is attached and open for reading, a get pointer is maintained that marks how far you've read into the file. Normally, the get pointer starts life at the very beginning of the file.

Once your file is opened for reading, you can use all of the iostream input functions described earlier to read data from the file. For example, the following code opens a file and then reads a single character from it:

```
char c;
ifstream readMe( "My File" );
readMe.get( c );
```

5.3.1 The iostream State Bits

Every stream, whether an istream or an ostream, has a series of four state bits associated with it:

```
enum io_state
 {
   goodbit=0,
   eofbit=1,
   failbit=2,
   badbit=4
 };
```

iostream uses these bits to indicate the relative health of their associated stream. You can poke and prod these bits yourself, but there are four functions that reflect each bit's setting:

- the function `int good();` returns nonzero if the stream used to call it is ready for I/O. Basically, if good() returns 1, you can assume that all is right with your stream and expect that your next I/O operation will succeed.

- The function `int eof();` returns 1 if the last I/O operation puts you at end-of-file.

- The function `dint fail();` returns 1 if the last operation fails for some reason. As an ex-ample, an input operation might fail if you try to read a short but encounter a text string instead.
- The function `int bad();` returns 1 if the last operation fails and the stream appears to be corrupted. When bad() returns 1, you're in TROUBLE.

There is also the function `void clear( int newState=0 );` which is used to reset the state bits to the state specified as a parameter. In general, you should call `clear()` without specifying a param-eter. clear()'s default parameter sets the state bits back to the pristine, good setting. If you don't clear the state bits after a failure, you won't be able to continue reading data from the stream.

For the most part, you should focus on the value returned by `good()`. As long as `good()` returns 1, there's no need to check any of the other functions. Once `good()` returns 0, you can find out why by querying the other three state functions.

The usual way to repeatedly read data is to usw a while loop and an iostream function as its conditional expression, for example:

```
ifstream readMe( "My_File" );

...

while ( readMe.get( c ) )
  cout << c;
```

What causes this while loop to exit?

`readMe.get( c )` returns a reference to `readMe`, correct?

Actually, this is where the C++ compiler displays a little sleight of hand. When the compiler detects an iostream I/O function used where an `int` *is expected*, it uses the current value of good() as the return value for the function. The previous while loop exits when `readMe.get( c )` either fails or hits an end-of-file.

We can do better by using `good( )`

The sample program, `stateBits.cpp`, demonstrates the basics of working with the `iostream` state bits and state bit functions. Close the current project by selecting from the menu.

The full listing for `stateBits.cpp` is a follows:

```
#include <iostream.h>

int main()
{
char done = false;
char c;
short number;

while ( ! done )
{
cout << "Type a number: ";
cin >> number;

if ( cin.good() )
{
if ( number == 0 )
{
cout << "Goodbye...";
done = true;
}
else
cout << "Your number is: " << number << "\n\n";
}
else if ( cin.fail() )
{
cin.clear();

cin.get( c );
cout << c << " is not a number...";
cout << "Type 0 to exit\n\n";
}
else if ( cin.bad() )
{
cout << "\nYikes!!! Gotta go...";
done = true;
}
}
```

```
return 0;
}
```

The sample output is as follows:

```
Type a number:
Type a number small enough to fit inside a short, like 256:
Type a number: 256
```

stateBits.cp starts with the usual `#include <iostream.h> #include` (since we won't be doing any file I/O, there's no need to include < fstream.h>):

`stateBits` creates a loop that reads in a number and then prints the number in the console window. If the number entered is 0, the program exits. Things start to get interesting when a letter is entered instead of a number.

Note that `done` acts as a Boolean logic operator. When it is set to `true`, the loop exits. `c` and `number` are used to hold data read from the console.

We enter the main loop, are prompted for a number and then use >> to read the `number` from the console.

If a number appropriate for a short is typed at the prompt, `cin.good()` returns true:

If the number typed is 0, we say goodbye and drop out of the loop; otherwise, we display the number and start all over again

If the input is of the wrong type (*e.g.* a letter or a `float`), or is a number that is too large (99999) or too small (-72999), the input operation fails and `cin.fail()` returns 1:

If a `fail` is detected, the first thing we must do is call `clear()` to reset the state bits — if we don't clear the state bits back to their healthy state and we won't be able to continue reading data from the stream. Once the state bits are reset, we read the character that caused the the stream to choke. Since we're not trying to interpret this character as a number, this read won't fail. Having read in the offending character, we display it, along with an appropriate message on the console

This example implements a pretty simple-minded recovery algorithm. If you typed in something like `xxzzy`, the loop would fail five times since you knock out only a single character with each recovery. You might want to try your hand at a more sophisticated approach. For example, you might use

`cin.ignore()` to suck in all the characters up to and including a carriage return. Better yet, you might use `cin.get()` to read in the remainder of the offending characters and then pack-age them in an appropriate error message.

The final possibility lies with a call to `bad()`. Since the bad bit will likely never be set, you'll probably never see this message.

5.4 Writing Data to a File

Earlier, the `ifstream` constructor was used to open a file for reading:

```
ifstream readMe( "My_File" );
```

In the same way, the ofstream constructor can be used to open a file for writing:

```
ofstream writeMe( "My_File" );
```

The `ofstream` constructor takes two parameters, with `ios::out` used as the default mode parameter. Note that you can pass more than one mode flag at a time. To open a file for writing if the file doesn't already exist, try something like this:

```
ofstream writeMe( "My_File", ios::out | ios::nocreate );
```

The rest of the mode flags as the same as for Reading a file above.

There is a way to open a file for both reading and writing. Use the `fstream` class and pass both the `ios::in` and `ios::out` mode flags, like this:

```
fstream inAndOut( "My_File", ios::in | ios::out );
```

The `fstream` class is set up with two file position indicators, one for reading and one for writing. Prottypes and definitions are found in the include file <`fstream.h`>.

Once your file is open, you can close it by calling the `close()` member function, for example:

```
writeMe.close();
```

In general, this call isn't really necessary since the `ifstream` and `ofstream` destructors automatically close the file attached to their associated stream.

You can also create an `ifstream` or `ofstream` without associating it with a file.

Why would you want to do this?

If you planned on opening a series of files, one at a time, you might want to do this by using a single stream, not by declaring one stream for each file. Using a single stream is more economical. Here's an example:

```
ifstream readMe;
readMe.open( "File_1" );
// Read contents - be sure to include error checking!
readMe.close();
readMe.open( "File_2" );
// Read contents - be sure to include error checking!
readMe.close();
// Repeat this as necessary...
```

5.5 read( ), write( ), and Others

There are some `istream` member functions that are particularly useful when dealing with files. The member function `read()` reads a block of size bytes and stores the bytes in the buffer pointed to by data:

```
istream &read(void *data, int size);
```

As you'd expect, if an end-of-file is reached before the requested bytes are read, the fail bit is set.

The member function `size_t istream::gcount()` returns the number of bytes successfully.

The member function `write()` inserts a block of size bytes from the buffer pointed to by data:

```
ostream &write(const void *data, size_t size);
```

The member function `size_t ostream::pcount()` returns the number of bytes inserted by the preceding `write()` call.

5.6 Customizing the iostream

There are times when the standard operators and member functions of iostream are not adequate. For example, remember the MenuItem class we declared:

```
class MenuItem
 {
   private:
    float price;
    char name[ 40 ];

   public:
    MenuItem::MenuItem( float itemPrice,char *itemName );
    float MenuItem::GetPrice();
};
```

Now suppose you want to display the contents of a MenuItem using iostream. You can write a DisplayMenuItem() member function that takes advantage of iostream, but that is somewhat awkward. If you want to display a MenuItem in the middle of a cout sequence, you have to break the sequence up, sandwiching a call to DisplayMenuItem() in the middle:

```
cout << "Today's special is: ";
myItem.DisplayMenuItem();
cout << "...\n";
```

Wouldn't it be nice if iostream knew about MenuItems so that you could do something more convenient, like this:

```
cout << "Today's special is: " << myItem << "...\n";
```

Well there is a way to do this: Using the techniques we have learned from operator overloading, you create an operator<<() function that knows exactly how you want your MenuItem displayed.

What's more, you can overload the >> operator, providing an operator>>() function that knows how to read in a MenuItem. The only restriction on both of these cases is that your >> and << overloading functions **must** return the appropriate stream reference so that you can use the >> and << operators in a sequence.

5.6.1 An >> and << Overloading Example

Our next sample program, `overload.cpp`, extends the `ostream` and `istream` classes by adding functions that overload both >> and <<.

The full code listing is:

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>

const short kMaxNameLength = 40;


//---------------------------------------  MenuItem

class MenuItem
{
private:
float price;
char name[ kMaxNameLength ];

public:
void SetName( char *itemName );
char *GetName();
void SetPrice( float itemPrice );
float GetPrice();
};


// I added these two prototypes. They should have been here
// in the first place...  -- Dave Mark 10/20/95
istream &operator>>( istream &is, MenuItem &item );
ostream  &operator<<( ostream &os, MenuItem &item );


void MenuItem::SetName( char *itemName )
{
strcpy( name, itemName );
}
```

```
char *MenuItem::GetName()
{
return( name );
}

void MenuItem::SetPrice( float itemPrice )
{
price = itemPrice;
}

float MenuItem::GetPrice()
{
return( price );
}


//-------------------------  iostream operators


istream &operator>>( istream &is, MenuItem &item )
{
float itemPrice;
char itemName[ kMaxNameLength ];

is.getline( itemName, kMaxNameLength );
item.SetName( itemName );

is >> itemPrice;
item.SetPrice( itemPrice );

is.ignore( 1, '\n' );

return( is );
}

ostream &operator<<( ostream &os, MenuItem &item )
{
```

```
os << item.GetName() << " ($"
<< item.GetPrice() << ") ";

return( os );
}


//---------------------------------------  main()

int main()
{
ifstream readMe( "Menu Items" );
MenuItem item;

while ( readMe >> item )
cout << item << "\n";

return 0;
}
```

The output of this program is:

```
Spring Rolls ($2.99)
Hot and Sour Soup ($3.99)
Hunan Chicken ($8.99)
General Tso's Shrimp ($9.99)
Spring Surprise ($15.99)
```

`overload.cpp` starts with some familiar `#includes` and a `const` definition (`const short kMaxNameLength = 40`.

The `MenuItem` class is a slightly modified version of the one in previous examples. For one thing, the constructor is left out. Instead of initializing the data members when a `MenuItem` is created, `iostream` is used to read in a series of `MenuItems` from a file and initialize each data member using the newly added `SetName()` and `SetPrice()` member functions:

`SetName()` is used to set the value of the name data member. `SetPrice()` is used to set the value of the price data member

`GetName()` returns a pointer to the name data member. By giving the caller of this public function direct access to name, we're sort of defeating the purpose of marking name as private. A more appropriate approach might be to have `GetName()` return a copy of name.

`GetPrice()` returns the value of the price data member.

The `operator>>()` function is called by the compiler whenever the >> operator is encountered having an `istream` as its left operand and a MenuItem as its right operand. Since all >> sequences are resolved to `istream` references, the left operand is always an `istream` object. To make this a little clearer, imagine an >> sequence with several objects in it:

```
cin >> a >> b;
```

`iostream` starts by evaluating this expression from the left, as if it were written like this:

```
(cin >> a) >> b;
```

Since the >> operator resolves to an `istream` object, the expression `cin >>` a resolves to `cin`, leaving this:

```
cin >> b;
```

The same logic holds true for the << operator:

```
cout << a << b;
```

As the compiler evaluates this expression from left to right, the left operand of the << operator is always an `ostream` object. The point is, whether `istream` or `ostream`, all an `operator()` function needs to do to support sequences is to return the stream reference passed in as the first parameter.

`operator>>()` reads a single MenuItem object from the specified input stream. First, `getline()` is used to read the item's name. Notice that the second parameter to `getline()` is used to limit the number of characters read in, ensuring that `itemName` doesn't exceed its bounds. `SetName()` is used to copy the entered name into the name data member.

Then, >> is used to read the item's price into `itemPrice`, and `SetPrice()` is used to copy `itemPrice` into the price data member.

When the extraction operator reads the price from the input stream, it leaves the carriage return following the number unread.

`ignore()` is used to grab the carriage return, leaving the stream set up to read the next `MenuItem`.

Finally, the stream passed in to the `operator>>()` function is returned, preserving the integrity of the sequence

`operator<<()` is somewhat simpler. It uses << to write the name and price data members.

Once again, the stream passed in as the first parameter is returned.

`main()` declares an ifstream object and ties it to the file named `Menu` Items. This file contains a list of `MenuItems` with the name and price of each item appearing on its own line.

`main()` also declares a `MenuItem` object named item. Notice that no parameters are passed because there's no constructor to do anything with the parameters.

Next, a `while` loop is used to read in all the `MenuItems` that can be read from the input stream (which is, in this case, a file named `Menu` Items). The overloaded version of >> is used to read in a `MenuItem`, and the overloaded version of << is used to display the MenuItem in the console window.

It's important to note that `operator>>()` and `operator<<()` are designed to work with any input and output stream. In this case, the `MenuItems` are read from a file and displayed in the console window. By making a few changes to `main()` — and not changing the two `operator()` functions — you can easily change the program to read from standard input (you'd probably want to add in a prompt or two) and send the output to a file. This is easy with the `iostream`.

5.6.2 Formatting Your Output

In the preceding program, we overloaded the << operator so that we could display a `MenuItem` precisely the way we wanted it to appear. Unfortunately, there's no way to overload the << operator to customize the appearance of *built-in* data types such as `short` or `float`.

Fortunately, iostream provides several mechanisms that allow you to customize your I/O operations.

In general, `iostream` follows some fairly simple rules when it comes to formatting output. If you insert a single `char` in a stream, exactly one character position is used. When some form of integral data is inserted, the

insertion is exactly as wide as the number inserted, including space for a sign, if applicable. No padding characters are used.

When a `float` is inserted, room is made for up to six places of precision to the right of the decimal place. Trailing zeros are dropped. If the number is either very large or very small (how big or how small depends on the implementation), *exponential notation is used*. Again, room is made for a sign, if applicable. For example, the number 1.234000 takes up five character positions in the stream since the trailing zeros are dropped: 1.234

When a string is inserted, each character, not including any `NULL` terminator, takes up one character position.

The Formatting Flags

The `ios` class maintains a set of flags that control various formatting features. You can use the `ios` member functions `setf()` and `unsetf()` to turn these formatting features on and off.

Each feature corresponds to a bit in a bit field maintained by the ios class.

Some features are independent, while others are grouped together. For example, the flag `ios::skipws` determines whether white space is skipped during extraction operations. This feature is not linked to any other features, so it may be turned on and off without impacting any of the other formatting flags.

To turn an independent flag on and off, you use the `setf()` and `unsetf()` member functions as follows:

```
cin.setf( ios::skipws ); // Skip whitespace on input
cin.unsetf( ios::skipws ); // Don't skip whitespace
                           // on input
```

Alternatively, you can use the `flag()` member function to retrieve the current flag settings as a group, OR the new flag into the group, and then use `flag()` again to reset the flag settings with the newly modified bit field:

```
int myFlags;
myFlags = cout.flag(); // returns flag bitfield
myFlags |= ios::skipws; // ORs in skipws flag
cout.flag( myFlags ); // resets flags
```

Unless you really need to work at this level, you're better off sticking with `setf()` and `unsetf()`.

Turning independent flags on and off individually is no problem, but things get interesting when flags are grouped. For example, the `radix` flags determine the default base used to represent numbers in output.

The `radix` flags are `dec`, `oct`, and `hex`, repre senting decimal, octal, and hexadecimal formats, respectively. The problem here is that only *one* of these flags should be turned on at a time. If you use `setf()`, you could easily turn all three flags on, producing unpredictable results.

To handle grouped flags, `setf()` makes use of a second, optional parameter that indicates which group a flag belongs to.

For example, the `radix` flags `dec, oct`, and `hex` belong to the group `basefield`. To set the `hex` flag, you make the following call:

```
cout.setf( ios::hex, ios::basefield );
```

This call ensures that when the specified flag is set, the remainder of the fields in the group get unset.

The grouped flags `left`, `right`, and `internal` are part of the `adjustfield` group. They are used in combination with the `width()` member function.

`width()` determines the minimum number of characters used in the **next** (and *only* next) numeric or string output operation. If the left flag is set, the next numeric or string output operation appears left-justified in the currently specified `width()`. The output is padded with the currently specified `fill()` character. You can use `fill()` to change this padding character.

5.6.3 A Formatting Example,`formatter.cpp`

Let us study a simple formatting example in order to illustrate many of above points.

The full code listing for `formatter.cpp` is as follows:

```
#include <iostream.h>


//--------------------------------------  main()

int main()
```

```
{
cout << 202 << '\n';

cout.width( 5 );
cout.fill( 'x' );
cout.setf( ios::left, ios::adjustfield );

cout << 202 << '\n';

cout.width( 10 );
cout.fill( '=' );
cout.setf( ios::internal, ios::adjustfield );

cout << -101 << '\n';

cout.width( 10 );
cout.fill( '*' );
cout.setf( ios::right, ios::adjustfield );

cout << "Hello";

return 0;
}
```

The output of this program is:

```
202
202xx
-======101
*****Hello
```

`formatter.cp` starts with the standard include file.

`main()` starts by displaying the number 202 in the console in standard fashion

```
cout << 202 << '\n';
```

As you'd expect, this code produces the following line of output: 202.

Next, `width()` is used to set the current width to 5, and `fill()` is used to make `x` the padding character:

Remember, `width()` applies only to the very next string or numeric output operation, even if it is part of a sequence. The padding character lasts until the next call of `fill()` or until the program exits.

If your output operation produces more characters than the current width setting, don't worry. All your characters will be printed.

Now, the left flag is set, asking `iostream` to left-justify the output in the field specified by `width()`:

```
cout.setf( ios::left, ios::adjustfield );
cout << 202 << '\n';
```

When the number 202 is printed again, it appears like this:

```
202xx
```

Then, `width()` is altered to 10, `fill()` is changed to `=`, and the `internal` flag is set. The `internal` flag asks `iostream` to place padding in between a number and its sign, if appropriate, so that it fills the `width()` field:

```
cout.width( 10 );
cout.fill( '=' );
cout.setf( ios::internal, ios::adjustfield );
cout << -101 << '\n';
```

Printing the number -101 produces the following line of output:

```
-======101
```

Finally, `width()` is reset to 10 (otherwise, it would have dropped to its default of 0), `fill()` is set to `*`, and the right flag is set to right-justify the output:

```
cout.width( 10 );
cout.fill( '*' );
cout.setf( ios::right, ios::adjustfield );
cout << "Hello";
```

When the string "Hello" is printed, this line of output appears:

```
*****Hello
```

5.6.4 More Flags and Methods

The `showbase` flag is independent. If it is set, octal numbers are displayed with a leading `zero` and `hex` output appears with the two leading characters `0x`. The `showpoint`, `uppercase`, and `showpos` flags are also independent. If `showpoint` is set, trailing zeros in floating-point output are displayed. If `uppercase` is set, `E` rather than `e` is used in scientific notation and `X` rather than `x` is used in displaying hex numbers. If `showpos` is set, positive numbers appear with a leading `+`.

The `scientific` and `fixed` flags belong to the `floatfield` group. If `scientific` is set, scientific notation is used to display floating-point output. If `fixed` is set, standard notation is used. If neither bit is set, the compiler uses its judgment and prints very large or very small numbers using scientific notation and all other numbers using standard notation. To turn off both bits, you pass a zero instead of fixed or scientific:

```
cout.setf( 0, ios::floatfield );
```

Both the `fixed` and `scientific` flags are tied to the `precision()` member function. `precision()` determines the number of digits displayed after the decimal point in floating-point output:

```
cout.precision( 6 ); // The default for precision...
```

Finally, the `unitbuf` and `stdio` flags are related but not grouped. If `unitbuf` is set, the output buffer is flushed after each output operation. `stdio`, which is only for using *C* I/O, flushes `stdout` and `stderr` after every insertion.

5.7 Manipulators

The `iostream` provides a set of special functions known as *manipulators* that allow you to perform specific I/O operations while you're in the middle of an insertion or an extraction. For example, consider this line of code:

```
cout << "Enter a number: " << flush;
```

This code makes use of the `flush` manipulator. When its turn comes along in the output sequence, the flush manipulator flushes the buffer associated with `cout`, forcing the output to appear immediately as opposed to waiting for the buffer to get flushed naturally.

Just as an I/O sequence can appear in different forms, a manipulator can be called in several different ways. Here are two more examples, each of which calls the flush manipulator:

```
cout.flush(); // Call as a stream member function
flush( cout ); // Call with the stream as a parameter
```

Use whichever form fits in with the I/O sequence you are currently building. If you plan on calling any manipulators that take parameters, be sure to include the file ¡iomanip.h¿. In addition, some iostream implementations require you to link with the math library to use certain manipulators. Check your develop-ment environment manual to be sure.

The Manipulators:

`dec()`, `oct()`, and `hex()` turn on the appropriate format flags, thus turning off the rest of the flags in the basefield group.

`endl()` places a carriage return (`'\n'`) in its output stream and then flushes the stream.

`ends()` places a null character in its output stream and then flushes the stream.

`ws()` eats up all the white space in its input stream until it hits either an end-of-file or the first non-white-space character.

None of the above manipulators take any parameters.

The six remaining to be discussed all take a single parameter and require the included file <`iomanip.h`>.

`setbase(int b)` sets the current radix to either `8, 10`, or `16`.
`setfill(int f)` is a manipulator version of the `fill()` member function.
`setprecision(int p)` is the manipulator version of precision().
`setw(int w)` is the manipulator version of `width()`.
`setiosflags(long f)` is the manipulator version of `setf()`.
`resetiosflags(long f)` is the manipulator version of `unsetf()`.
Here are two manipulator examples. The line

```
cout << setbase( 16 ) << 256 << endl;
```

produces this line of output:

```
100
```

And, the line

```
cout << setprecision( 5 ) << 102.12345;
```

produces this line of output:

```
102.12
```

5.8 The `istrstream` and `ostrstream`

We have met the C `stdio` function sprintf(), there is a similar feature in C++.

Recall `sprintf()` allows you to perform all the standard I/O functions normally associated with `printf()` and `fprintf()` on an array of characters.

The `istrstream` and `ostrstream` classes offer all the power of their ancestor classes (`istream` and `ostream` and, ultimately, `ios`) and allow you to write formatted data to a buffer that you create in memory. Here's an example program, `strstream.cpp`:

```
#include <iostream.h>
#include <sstream.h>

const short kNumberOfLetters = 26;

//---------------------------------------  main()

int main()
{
ostringstream ostr;
short i;

for ( i = 0; i < kNumberOfLetters; i++ )
ostr << (char)('a' + i);

cout << "Number of characters written: "
<< i << '\n';

cout << "Buffer contents: " << ostr.str();

return 0;
}
```

The output of this program is:

```
Number of characters written: 10
Buffer contents: abcdefghi
```

strstream.cp starts with two `#include`s, the standard <`iostream.h`> and the file required for the `istrstream` and `ostrstream` classes, <`strstrea.h`>.

The constant `kBufferSize` is used to define the size of the buffer that makes up the `ostrstream` object:

m̂ain() creates a buffer to hold the stream's characters. The `ostrstream` constructor takes two parameters, a pointer to the buffer and the size of the buffer. The variable `i` is used to keep track of the number of characters written to the ostrstream

Next, a `while` loop uses `ostr` just as it would use `cout`, writing characters to the stream until an end-of-file causes the loop to terminate. iostream generates the end-of-file when the `put()` pointer points beyond the last character in the stream's buffer (just like its `ifstream` counterpart).

When the loop exits, ten characters, from a to j, have been written to the stream's buffer. The number of characters written to the stream is then displayed.

Next, a `NULL` terminator is written on the last byte of the stream's buffer, creating a `NULL`-terminated string in buffer.

Finally, the contents of the stream are printed.

Just as an `ostrstream` object mirrors the behavior of `cout`, you can create a similar example, using an `istrstream` object, that mirrors the behavior of `cin`. The `istrstream` constructor takes the same two parameters as the `ostrstream` constructor.

Together, istrstream and ostrstream give you a powerful set of tools to use when you work with strings in memory.

5.9 Templates

5.9.1 The Need for Templates

When you design a class, you're forced to make some decisions about the data types that make up that class.

For example, if your class contains an array, the class declaration specifies the array's data type. In the following class declaration, an array of shorts is implemented:

```
class Array
 { private:
     short arraySize;
      // Number of array elements
     short *arrayPtr;
      // Pointer to the array
    public:
      Array( short size );
       // Allocate an array
       // of size shorts
      ~Array();
       // Delete the array
 };
```

In this class,

- the constructor allocates an array of arraySize elements, each element of type `short`.
- The destructor deletes the array.
- The data member `arrayPtr` points to the beginning of the array.
- To make the class truly useful, you'd probably want to add a member function that gives access to the elements of the array.

This Array class works just fine as long as an array of shorts meets your needs.

What happens when you decide that an Array of shorts is not what you need?

Perhaps you need to implement an array of `long`s or, even better, an array of your own data types.

One approach you can use is to make a copy of the Array class (member functions and all) and change it slightly to implement an array of the appropriate type.

For example, here's a version of the `Array` class designed to work with an array of longs:

```
class LongArray
 { private:
      short arraySize;
      long *arrayPtr;
   public:
       LongArray( short size );

       ~LongArray( void );
 };
```

There are definitely problems with this approach.

- You are creating a maintenance nightmare by duplicating the source code of one class to act as the basis for a second class.
- Suppose you add a new feature to your Array class.
- Are you going to make the same change to the LongArray class?

5.9.2 Defining Templates

C++ templates allow you to parameterize the data types used by a class or function.

Instead of embedding a specific type in a class declaration, you provide a template that defines the type used by that class.

An example should make this a little clearer.

Here's a templated version of the Array class presented earlier:

```
template <class T>

class Array
   {
        private:
          short arraySize;
          T *arrayPtr;
```

```
        public:
          Array( short size );

          ~Array( void );
};
```

- The keyword `template` tells the compiler that what follows is not your usual, run-of-the-mill `class` declaration.
- Following the keyword template is a pair of angle brackets (<>) that surround the `template`'s template argument list.
- This list consists of a series of comma-separated arguments
- one argument is the minimum

Once your class template is declared, you can use it to create an object. When you declare an object using a class template, you have to specify a template argument list along with the class name.

Here's an example:

```
Array<long> longArray( 20 );
```

The compiler uses the single parameter, `long`, to convert the `Array` template into an actual class declaration.

This declaration is known as a *template instantiation*.

The instantiation is then used to create the `longArray` object.

A Template Argument List Containing More Than One Type

A `template`'s argument list may contain more than one type.

The `class` keyword **must** precede each argument, and an argument name **can not** be repeated.

Here's an example:

```
template <class Able, class Baker>

class MyClass
  {
     public:
       MyClass( Able param );
```

```
      ~MyClass( void );
      Baker MemberFunction(
         Baker param );
   };
```

Here's a sample definition of a MyClass object:

```
MyClass<long, char *> myObject( 250L );
```

Take a look at the template arguments.
The first, long, will be substituted for `Able`.
The second, `char *`, will be substituted for `Baker`.

5.9.3 Function Templates

The template technique can also be applied to functions.

Here's an example of a function template declaration:

```
template <class T, class U>

T MyFunc( T param1,
          U param2 )
  {
     T var1;
     U var2;

     ......

   }
```

The types defined in the template argument list are then used freely throughout the remainder of the function declaration.

If you use a template to define a function, you must also include the same template information in the function's prototype. Here's a prototype for `MyFunc()`:

```
template <class T, class U>
T MyFunc( T param1,
          U param2 );
```

Function Template Instantiation

When you call a function that has been templated, the compiler uses the parameters passed to the function to determine the types of the template arguments.

Here's a simple example:

```
template <class T>

void MyFunc( T param1 );
```

Suppose this function template were called as follows:

```
char *s;

MyFunc( s );
```

The compiler would match the type of the calling parameter `(char *)` with the type of the receiving parameter (`T`).

In this case, an instantiation of the function is created, and the type `char *` is substituted for `T` everywhere it occurs.

Consider this template:

```
template <class T>

void MyFunc( T param1,
             T param2 );
```

This call of `MyFunc()` won't compile:

```
short i;
int j;
MyFunc( i, j );
```

First, the compiler matches the first parameter and determines that `T` is a short.

When the compiler moves on to the second parameter, it finds that `T` should be an `int`.

Even though an `int` and a `short` are *close*, but since they are not an exact match.

Chapter 6

Multiple Inheritance

6.1 What is Multiple Inheritance?

Our next topic is a variation on an earlier theme, class derivation. In the examples presented in earlier, each derived class was based on a]em single base class. That doesn't have to be the case, however.

C++ allows you to derive a class from more than one base class, a technique known as *multiple inheritance*. As its name implies, multiple inheritance means that a class derived from more than one base class inherits the data members and member functions from each of its base classes.

Why would you want to inherit members from more than one class? Check out the derivation chain in Figure 6.1. The ultimate base class, known as the root base class, in this chain is Computer. The two classes `ColourComputer` and `LaptopComputer` are special types of `Computers`, each inheriting the nonprivate members from `Computer` and adding members of their own as well.

Now we can bring multiple inheritance comes into play. The class `ColourLaptop` is derived from both `ColourComputer` and `LaptopComputer` and inherits members from each class.

Multiple inheritance allows you to take advantage of two different classes that work well together. If you want a program that models a colour, laptop computer and you already have a `ColourComputer` class that manages colour information and a `LaptopComputer` class that manages information about laptops, there is no point reinvent the wheels.Think of the `ColourLaptop` class as the best of both worlds — the union of two already designed classes.

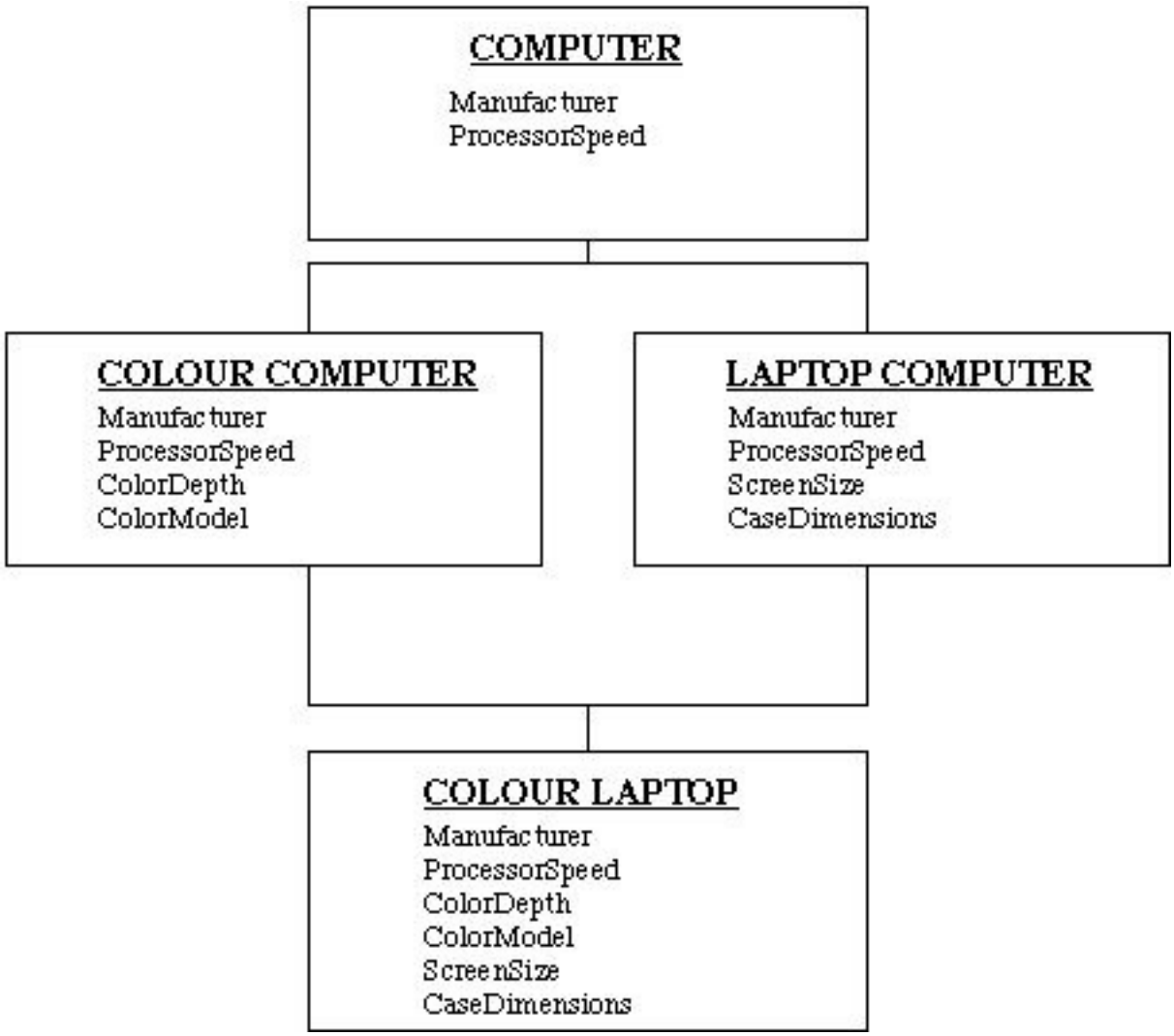

Figure 6.1: Mutliple Inheritance Example

Just as with single inheritance, there are times when multiple inheritance makes sense and times when it is inappropriate. Use the *is a* rule to guide your design. If the derived class *is a* subset of the base class, derivation is appropriate.

In our preceding example, a `ColourComputer` *is a* `Computer` and a `LaptopComputer` *is a* `Computer`. At the same time, a `ColourLaptop` is both a `ColourComputer` and a `LaptopComputer`. This model works just fine.

Let's look at another example.

Imagine a `Date` class and a `Time` class. The `Date` class holds a date, like 07/27/94, while the `Time` class holds a time of day, like 10:24 am. Now suppose you wanted to create a `TimeStamp` class, derived from both the `Date` and `Time` classes.

Would this make sense?

The answer is **NO**: A `TimeStamp` is **not** a `Date` and it is not a `Time`. Instead, a `TimeStamp` *has a* `Date` and *has a* `Time`. When your derivation fits the *has a* model rather than the *is a* model you should rethink your design.

In this case, the `TimeStamp` class *should include* `Date` and `Time` objects as data members, *rather than* using multiple inheritance.

The simple rule is:

- *is a* indicates inheritance.
- *has a* describes the relationship between your derived and base classes, and you should rethink your design.

6.2 A Multiple Inheritance Example, `multInherit.cpp`

Let us now look a complete code eample which demonstrates multiple inheritance as well as a few additional C++ features that you should find interesting.

The full code listing for `multInherit.cpp` is:

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

const short kMaxStringLength = 40;
```

```
//---------------------------------------  Predator

class Predator
{
private:
char favoritePrey[ kMaxStringLength ];

public:
Predator( char *prey );
~Predator();
};

Predator::Predator( char *prey )
{
strcpy( favoritePrey, prey );

cout << "Favorite prey: "
<< prey << "\n";
}

Predator::~Predator()
{
cout << "Predator destructor was called!\n\n";
}


//---------------------------------------  Pet

class Pet
{
private:
char favoriteToy[ kMaxStringLength ];

public:
Pet( char *toy );
~Pet();
};
```

```
Pet::Pet( char *toy )
{
strcpy( favoriteToy, toy );

cout << "Favorite toy: "
<< toy << "\n";
}

Pet::~Pet()
{
cout << "Pet destructor was called!\n";
}


//--------------------------  Cat:Predator,Pet

class Cat : public Predator, public Pet
{
private:
short catID;
static short lastCatID;

public:
Cat( char *prey, char *toy );
~Cat();
};

Cat::Cat( char *prey, char *toy ) :
Predator( prey ), Pet( toy )
{
catID = ++lastCatID;

cout << "catID: " << catID
<< "\n---------\n";
}

Cat::~Cat()
{
```

```
cout << "Cat destructor called: catID = "
<< catID << "...\n";
}

short Cat::lastCatID = 0;


//---------------------------------------  main()

int main()
{
Cat TC( "Mice", "Ball of yarn" );
Cat Benny( "Crickets", "Bottle cap" );
Cat Meow( "Moths", "Spool of thread" );

return 0;
}
```

The output is as follows:

```
Favorite prey: Mice
Favorite toy: Ball of yarn
catID: 1
---------Favorite
prey: Crickets
Favorite toy: Bottle cap
catID: 2
---------Favorite
prey: Moths
Favorite toy: Spool of thread
catID: 3
---------Cat
destructor called: catID = 3...
Pet destructor was called!
Predator destructor was called!
Cat destructor called: catID = 2...
Pet destructor was called!
```

```
Predator destructor was called!
Cat destructor called: catID = 1...
Pet destructor was called!
Predator destructor was called!
```

multInherit.cpp starts off with a few #includes and a familiar const short kMaxStringLength = 40.

Next, three classes are defined.

- The Predator class represents a predatory animal
- The Pet class represents a housepet.
- The Cat class is derived from both the Predator class and the Pet class – After all, a cat *is a* predator and a cat *is a* pet.

The Predator class is pretty simple. It features a single data member, a string containing the predator's favorite prey The Predator class also features a constructor and a destructor: The constructor initializes the favoritePrey data member and then prints its value; whilst The destructor prints an appropriate message, just to let you know it was called.

The Pet class is almost identical to the Predator class, with a favorite toy substituted for a favorite prey.

The Cat class is derived from both the Predator and Pet classes. Notice that the keyword public precedes each of the base class names and that the list of base classes is separated by commas:

```
class Cat : public Predator, public Pet
```

Cat contains two data members. The first, catID, contains a unique ID for each Cat. While numbering your cats might not be that useful, if we were talking about Employees or Computers, a unique employee ID or serial number can be an important part of your class design.

Notice that the second data member, lastCatID, is declared using the static keyword.

When you declare a data member or member function as *static*, the compiler creates a single version of the member that is shared by all objects in that class.

Why do this?

static members can be very useful. Since a static data member is shared by all objects, you can use it to share information between all objects in a class.

One way to think of a static member is as a global variable whose scope is limited to the class in which it is declared. This is especially true if the `static` member is declared as private or protected.

In this case, `lastCatID` is incremented every time a Cat object is created. Since `lastCatID` is not tied to a specific object, it always holds a unique serial number (which also happens to be the number of `Cat`s created).

The declaration of a `static` data member is just that, a declaration. When you declare a static within a class declaration you need to follow it up with a definition in the same scope.

Typically, you'll follow your class declaration immediately with a definition of the static data member, like this:

```
short Object::lastObjectID;
```

If you like, you can use this definition to initialize the `static` member. `static` data member scope is limited to the file it is declared in.

You'll typically stick your class declaration (along with the class's static member declarations) in a `.h` file.

This is not the case for your static member definition. The definition *should appear* in the `.cpp` file where it will be used.

Along with your `static` data members, you can also declare a static member function. Again, the function is not bound to a particular object and is shared with the entire class.

If the class `MyClass` included a static member function named `MyFunc()`, you could call the function using this syntax:

```
MyClass::MyFunc();
```

Since there is no current object when `MyFunc()` is called, you don't have the advantages of this and any references to other data members or member functions must be done through an object.

`static` member functions are usually written for the sole purpose of providing access to an associated static data member. To enhance your design, you might declare your static data member as private and provide an associated static member function marked as public or protected.

Back to the program.

The `Cat` class has a constructor and a destructor: The Cat constructor maps its input parameters to the `Predator` and `Prey` constructors as follows:

```
Cat::Cat( char *prey, char *toy ) :
Predator( prey ), Pet( toy )
```

The list that follows the constructor's parameter list is called the member initialization list. As you can see, a colon always precedes a constructor's member initialization list.

The Cat destructor also prints a message containing the catID, just to make the program a little easier to follow

Next, the `static` member `lastCatID` is defined. Without this definition, the program wouldn't compile. Notice also that we take advantage of this definition to initialize `lastCatID`.

`static` data members, just like C++ globals, are *automatically* initialized to 0. To make the code a little more obvious, we kept the initialization in there, even though it is redundant.

Finally, `main()` creates three `Cat` objects. Compare the three `Cat` declarations with the program's output. Notice the order of constructor and destructor calls. Note, the destructors are called in the reverse order of the constructors.

6.3 Resolving Ambiguities

Deriving a class from more than one base class brings up an interesting problem. Suppose the two base classes from our previous example, `Predator` and `Pet`, each have a data member with the same name (which is perfectly legal, by the way).

Let's call this data member `clone`. Now suppose that a `Cat` object is created, derived from both `Predator` and `Pet`. When this `Cat` refers to `clone`, which `clone` does it refer to, the one inherited from `Predator` or the one from `Pet`?

As it turns out, the compiler would complain if the `Cat` class referred to just plain `clone` because it can't resolve this ambiguity. To get around this problem, you can access each of the two clones by referring to

```
Predator::clone
```

or

`Pet::clone`

6.4 Multiple Roots

Here's another interesting problem brought on by multiple inheritance. Take a look at the derivation chain in Figure 6.2.

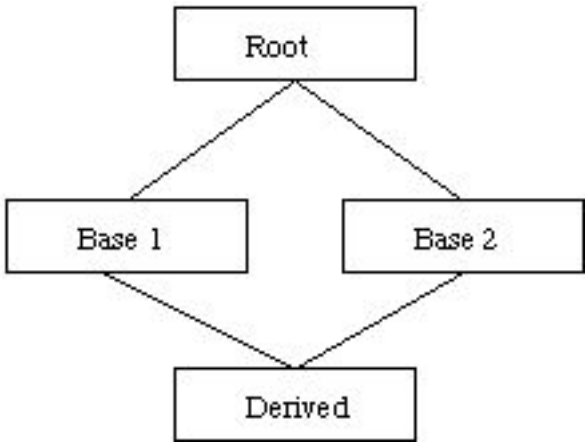

Figure 6.2: Mutliple Root Problem

Notice that the Derived class has two paths of inheritance back to its ultimate base class, Root. Since Derived is derived from both Base1 and Base2, when a Derived object is created, Base1 and Base2 objects are created as well. When the Base1 object is created, a Root object is created. When the Base2 object is created, a second Root object is created.

Why is this a problem?

Suppose Root contains a data member destined to be inherited by Derived. When Derived refers to the Root data member, which of the two Root objects contains the data member Derived is referring to? Sounds like another ambiguity to me.

6.4.1 A Multiple-Root Example, `nonVirtual.cpp`

Before we resolve this latest ambiguity, here's an example that shows what happens when a derived class has two paths back to its root class.

Th code listing for the `nonVirtual.cpp` program is:

```
#include <iostream.h>


//---------------------------------------  Root

class Root
{
public:
Root();
};

Root::Root()
{
cout << "Root constructor called\n";
}


//---------------------------------------  Base1

class Base1 : public Root
{
public:
Base1();
};

Base1::Base1()
{
cout << "Base1 constructor called\n";
}


//---------------------------------------  Base2

class Base2 : public Root
{
public:
```

```
Base2();
};

Base2::Base2()
{
cout << "Base2 constructor called\n";
}


//---------------------------------------  Derived

class Derived : public Base1, public Base2
{
public:
Derived();
};

Derived::Derived()
{
cout << "Derived constructor called\n";
}


//---------------------------------------  main()

int main()
{
Derived myDerived;

return 0;
}
```

The output is as follows:

```
Root constructor called
Base1 constructor called
Root constructor called
```

```
Base2 constructor called
Derived constructor called
```

nonVirtual.cpp starts by including <iostream.h>:

Four classes are then defined asshown in Figure 6.2.

Root consists of a constructor that prints a message letting you know it was called.

Base1 is derived from Root. Its constructor also prints a useful message.

Base2 is also derived from Root. Its constructor also prints a message in the console window.

Derived is derived from both Base1 and Base2. Just like all the other classes, Derived has its constructor print a message in the console window just to let you know it was called.

main() starts the constructor roller coaster by creating a Derived object. Since Base1 is listed first in the Derived derivation list, a Base1 object is created first. Since Base1 is derived from Root, it causes a Root object to be created. The Root constructor is called and then the Base1 constructor is called, resulting in the follow ing two lines of output:

```
Root constructor called
Base1 constructor called
```

Next, this process is repeated as a Base2 object is created. Since Base2 is also derived from Root, it causes a second Root object to be created. Once the Root constructor is called, control returns to Base2 and its constructor is called:

```
Root constructor called
Base2 constructor called
```

Once the Base2 object is created, control returns to the Derived class and the Derived constructor is called: Derived constructor called

6.4.2 The Virtual Base Class Alternative

Once again, think about the problem raised by this last example.

If the Root class contained a data member, how would the Derived object access the data member?

Which of the two `Root` objects would contain the real copy of the data member?

The answer to this problem lies in the use of *virtual base classes*. We have already declared a member function as virtual to allow a derived class to override the function. Basically, when a virtual function is called by dereferencing a pointer or reference to the base class, the compiler follows the derivation chain down from the root class to the most derived class and looks at each level for a function matching the virtual function. The lowest-level matching function is the one that is called.

Virtual functions are extremely useful. Here's why. Suppose you're writing a program that implements a window-based user interface. Let's say that your standard window is broken into several areas (we'll call them panes) and that each pane is broken into subpanes. When the time comes to draw the contents of your window, your Window class's `Draw()` member function is called. If your `Pane` class also has a `Draw()` member function and if the Window version of `Draw()` is declared as `virtual`, the `Pane`'s `Draw()` is called instead.

This same logic applies to your `SubPane` class and its `Draw()` function. If it is derived from `Pane`, the `SubPane`'s `Draw()` is called instead of the `Pane`'s `Draw()`. This strategy allows you to derive from an existing class using a new class whose actions are more appropriate or more efficient.

A similar technique can be used to remove the ambiguity brought up when a derived class has two different paths back to one of its ancestor classes. In our earlier example, `Root` was the root class, and `Base1` and `Base2` were derived from `Root`.

Finally, `Derived` was derived from both `Base1` and `Base2`. When we created a `Derived` object, we ended up creating two `Root` objects. Thus the ambiguity.

By declaring `Root` as a `virtual` base class, we're asking the compiler to merge the two `Root` object creation requests into a single `Root` object (you'll see how to mark a class as virtual in a moment). The compiler gathers every reference to the `virtual` base class from the different constructor member initialization lists and picks the one that's tied to the deepest constructor. That reference is used, and all the others are discarded. This will become clearer as you walk through the next sample program.

To create a virtual base class, you must insert the `virtual` keyword in the member initialization lists between the virtual base class and the potentially ambiguous derived class. You don't need to mark every class between `Root`

and `Derived` as long as the compiler has no path between `Root` and `Derived` that doesn't contain at least one virtual reference. The general strategy is to mark all direct descendants of the virtual base class. In this case, we'd need to place the virtual keyword in both the `Base1` and `Base2` member initialization lists.

Here's an example:

```
class Base1 : public virtual Root
{
public:
Base1();
};
```

The `virtual` keyword can appear either before or after the `public` keyword.

Once the `virtual` keywords are in place, the compiler ignores all member initialization list references to the `Root` class constructor except the deepest one. This sample `Derived` constructor includes a reference to the Root constructor:

```
Derived::Derived( short param ) : Root( param )
{
cout << "Derived constructor called\n";
}
```

Even if the `Base1` and `Base2` constructors map parameters to the `Root` constructor, their mappings are superseded by the deeper, Derived constructor. By overriding the constructor mappings, the compiler makes sure that only a single object of the `virtual` base class (in this case, `Root`) is created.

6.4.3 A Virtual Base Class Example, `virtual.cpp`

This next example brings these techniques to life.

The code listing for `virtual.cpp` is as follows:

```
#include <iostream.h>


//---------------------------------------  Root
```

```
class Root
{
protected:
short num;

public:
Root( short numParam );
};

Root::Root( short numParam )
{
num = numParam;

cout << "Root constructor called\n";
}


//---------------------------------------  Base1

class Base1 : public virtual Root
{
public:
Base1();
};

Base1::Base1() : Root( 1 )
{
cout << "Base1 constructor called\n";
}


//---------------------------------------  Base2

class Base2 : public virtual Root
{
public:
Base2();
```

```
};

Base2::Base2() : Root( 2 )
{
cout << "Base2 constructor called\n";
}


//---------------------------------------  Derived

class Derived : public Base1, public Base2
{
public:
Derived();
short GetNum();
};

Derived::Derived() : Root( 3 )
{
cout << "Derived constructor called\n";
}

short Derived::GetNum()
{
return( num );
}


//---------------------------------------  main()

int main()
{
Derived myDerived;

cout << "-------\n"
<< "num = " << myDerived.GetNum();

return 0;
```

```
}
```

The output of this program is:

```
Root constructor called
Base1 constructor called
Base2 constructor called
Derived constructor called
------
num = 3
```

As usual, `virtual.cpp` starts by including <`iostream.h`>:

This version of the `Root` class includes a data member named `num`. The `Root()` constructor takes a single parameter and uses it to initialize `num` (as you read through the code, try to figure out where the value for this parameter comes from).

`Base1` is derived from `Root`, but it treats `Root` as a

`virtual` base class. Notice that the `Base1()` constructor asks the compiler to call the `Root()` constructor and passes it a value of 1. Will this call take place?

`Base2` also declares `Root` as a `virtual` base class. Now there's no path down from Root that's not marked as `virtual`. The `Base2()` constructor asks the compiler to pass a value of 2 to the `Root()` constructor. Is this the value that is passed on to the Root() constructor?

The `Derived` class doesn't need the virtual keyword (although it wouldn't matter if virtual were used here). The `Derived()` constructor also asks the compiler to pass a value on to the `Root()` constructor. Since `Derived` is the deepest class, this is the constructor mapping that takes precedence. The Root data member num should be initialized with a value of 3. This function makes the value of num available to `main()`. Why can't `main()` reference num directly?

Derived inherits `num` and `main()` doesn't.

`main()` creates a `Derived` object, causing a sequence of constructor calls: Notice that the `Root` constructor is called only once. Finally, the value of `num` is printed.

As you've already seen, `num` has a value of 3, showing that the `Base1` and `Base2` constructor initializations are overridden by the deeper, `Derived` constructor initialization.

Chapter 7

Wrappers

This chapter examines a C++ class that is used as a software layer, or *wrapper* around a utility library written in C.

C++ has many featrure that make it a safer language than C — fewer tedious issues to address.

A C++ wrapper should improve the interface to a library. However, you should take care to ensure that this goal is met.

7.1 Wrapping Up a C libraray

Consider the C stabdard library for intergogating system directories on UNIX. There a five finctions we may wish to use in our C++. We would include them like this:

```
extern "C" {
   DIR *opendir(char *);
   dirent * readdir(DIR *);
   long telldir(DIR *);
   void seekdir( DIR *, long);
   void closedir(DIR *);
};
```

We have already seen these functions and structures in use with C. Here we focus on the C++ wrapping issues.

The `extern "C"` qualifying the function prototypes is a *linkage specification* — indicating that the functions are compiled by a C compiler.

A C++ wrapper class helps to manage the housekeeping by encapsulating pointers to C structures (such as `DIR` and using destructors to ensure that `closedir()` is called for example. However you must make sure you do this yourself.

Having declared the `extern "C"` functions we can now build our Directory wrapper class:

```
class Directory {
  Dir *dir;

  public:
    Directory(char *);
    ~Directory();
    const char *name();
    long tell();
    void seek();
};

Directory::Directory(char *path)
{
    dir = opendir(path);
}

Directory::~Directory()
{
   closedir(dir);
}

const char *Directory::name()

{  dirent *d = readdir(dir);
   return d ?  d->d_name : NULL;
}

long Directory::tell()

{
```

```
   return telldir(dir);
}

void Directory::seek(long loc)
{
   seekdir(dir, loc);
}
```

Note that we simply call thye C functions as they are called in C.

7.2 Standard C Header Files

If you look at many of the standard library header files you will see that they are already code up to deal with C++ compilation they have:

```
#ifdef  __cplusplus
extern "C" {
#endif
```

declared near the beginning of the file and

```
#ifdef  __cplusplus
 }
#endif
```

at the end of the file.

If a C++ compiler is being used then the `__cplusplus` macro will be set.

C++ prgrams can therefore safely `#include` standard C header files and have the data in the files safely linked for C++.

Chapter 8

Threads and C++

As we will see in this final example, writing threaded applications is no different in C++ than C.

In fact the best way is to study an example. Earlier when discussing threads (in C) we mentioned an example of multiplying matrices as a good application fro threads. Let's see how we can do this in C++.

8.1 Matrix Multiplication

The matrix multiplication example was written in C++ to show how an object-oriented program might be written to use threads. The concepts and ideas discussed thus far also apply to the C++ language. All the examples in this book could have just as well been written in C++ as in C.

This example simply performs a matrix multiplication, the multiplication of two simple NxN matrices. Essentially, the matrix multiplication performs many mathematical calculations that can be executed independently. This example executes the multiplication of the matrices in parallel by using multiple threads in the processes. The user is allowed to change the size of the matrix objects and the number of threads on which to execute the multiplication.

The example also places all the threaded code into a shared library. This way, all the threaded routines are hidden from the main program. Using this library concept also shows how programs can be changed to use threads without affecting all of the code. In this case, the main program could use a nonthreaded library as well as a threaded one.

This example is rather long, but it demonstrates many of the concepts that have been covered in previous chapters. The program falls into two main parts. The first part is the main thread that creates the matrix objects and starts the matrix multiplication. The second part is the library code, which performs the multiplication on the matrix objects.

The main thread creates the matrix objects, based on user-supplied arguments. The main thread then calls the `MatMult()` routine, which starts the multiplication. A global data structure (thread control block) is used in the library by all the threads in the program. The data structure contains the synchronization variables and other data needed to control the worker threads. This data structure is filled before any threads are created, because the threads use this data during their execution.

The worker threads are created as bound daemon threads. They are bound threads because of the compute nature of the work they do. For all the worker threads to execute in parallel, the level of concurrency would have to be increased, or the threads could be created as bound threads. The threads are also daemon threads because the worker threads should die when the main thread has finished executing. Because this example is compute intensive, the worker threads are created only once; there is no need to recreate the threads for each matrix multiply. The worker threads will always wait for more work to do. If there is no work to be done, then they go to sleep, waiting on a condition variable.

Once there is work to be done, signaled from the main thread, the worker threads wake up and perform the matrix operations on the data specified in the control data structure. At the same time, the main thread waits for all the worker threads to signal that they have finished the work. When the worker threads have finished and have signaled the main thread, they start over again, waiting for more work to do.

The source to Matrix.cpp is:

```
#define _REENTRANT
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <thread.h>
#include "Matrix.h"
```

```
// Main program
main(int argc, char **argv)
{
int size;
int num_threads;
hrtime_t start, stop;

if (argc != 3) {
        cout << "Usage: " << argv[0] << " Matrix-size Threads" << endl;
        exit(0);
        }

// set the size of the matrix and total threads for this run
size = atoi(argv[1]);
num_threads = atoi(argv[2]);
SetMaxThreads(num_threads);

if (size < num_threads) {
        cerr << "The size of the matrix MUST be greater then number of threads."
<< endl;
        exit(1);
        }

cout << "Matrix size: [" << size << "x" << size << "]" << endl;
cout << "Number of worker threads: " << num_threads << endl;

// Create the Matrix
Matrix a('A', size), b('B', size), c('C', size);

// fill A & B with data and clear C
a.fill(); b.fill(); c.clear();

// Start the timer
start = gethrtime();

// Do the matrix multiply
MatMult(a, b, c);
```

```
// Stop the timer
stop = gethrtime();

// Print the results -- Only if matrix size is small enough
cout << a << b << c;

// Print the run time
cout << "Matrix multiplication time = "
     << (double)(stop-start)/(double)1000000000
     << " seconds = " << stop-start << " nanoseconds" << endl;
}
```

The source to MatLib.cpp:

```
#define _REENTRANT
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <thread.h>
#include "Matrix.h"

const true = 1;
const false = 0;

// Thread control block - used by all threads as global data
struct thr_cntl_block {
  mutex_t start_mutex;
  cond_t  start_cond;
  mutex_t stop_mutex;
  cond_t  stop_cond;
  Matrix *a, *b, *c;
  int work2do;
  int thrs_running;
  int total_threads;
  int queue;
} TCB;

////////////////////////////////////////////////////////////////////////
```

```
//  Matrix Class Member Functions
//////////////////////////////////////////////////////////////////////

// Matrix constructor
Matrix::Matrix(char id, int size)
{
matid = id;
matsize = size;
data = new double[matsize*matsize];
}

// Matrix destructor
Matrix::~Matrix()
{
matsize = 0;
matid = 0;
delete[] data;
}

// Fills a matrix object with random data
void Matrix::fill()
{
int i;

for (i=0;i<matsize*matsize;i++)
        data[i] = double(rand()/1000);
srand(rand());
}

// Sets all elements of the matrix to 0.0
void Matrix::clear()
{
int i;

for (i=0;i<matsize*matsize;i++)
        data[i] = .0;
}
```

```
// Prints a Matrix object (if it is small enough)
void Matrix::print(ostream &s) const
{
int i;

if (matsize < 9) {
   s << "Matrix: " << matid << endl;

   for (i=0;i<matsize*matsize;i++)
        {
       s << setiosflags(ios::fixed) << setprecision(1)
            << setw(8) << data[i] << " ";

        if ((i%matsize) == matsize-1) s << endl;
        }

   s << endl << endl;
   }
}

// Overloaded << operator - for ease of printing
ostream &operator<<(ostream &s, const Matrix &mat)
{
   mat.print(s);
   return(s);
}

// Sets the maximum number of threads to use
void SetMaxThreads(int num)
{
TCB.total_threads = num;
}

// The matrix multiply subroutine
MatMult(Matrix &a, Matrix &b, Matrix &c)
{
int static running = false;
int i;
```

```
// Only run this code once, if MatMult is called multiple times
// then there is no need to recreate the threads
if (!running)
    {
    // Initialize the synch stuff.
    mutex_init(&TCB.start_mutex, USYNC_THREAD, 0);
    mutex_init(&TCB.stop_mutex, USYNC_THREAD, 0);
    cond_init(&TCB.start_cond, USYNC_THREAD, 0);
    cond_init(&TCB.stop_cond, USYNC_THREAD, 0);

    // set global variables
    TCB.work2do = 0;
    TCB.thrs_running = 0;
    TCB.queue = 0;
    if (!TCB.total_threads) TCB.total_threads = 1;

    // Create the threads - Bound daemon threads
    for (i = 0; i < TCB.total_threads; i++)
        thr_create(NULL,0, MultWorker, NULL, THR_BOUND|THR_DAEMON, NULL);

    // set the running flag to true so we don't execute this again
    running = true;
    }

// Assign global pointers to the Matrix objects
TCB.a = &a;
TCB.b = &b;
TCB.c = &c;

mutex_lock(&TCB.start_mutex);

  // Assign the number of threads and the amount of work to do
  TCB.work2do = TCB.total_threads;
  TCB.thrs_running = TCB.total_threads;
  TCB.queue = 0;

  // tell all the threads to wake up!
```

```
  cond_broadcast(&TCB.start_cond);

mutex_unlock(&TCB.start_mutex);

// yield this LWP
thr_yield();

// Wait for all the threads to finish
mutex_lock(&TCB.stop_mutex);

  while (TCB.thrs_running)
        cond_wait(&TCB.stop_cond, &TCB.stop_mutex);

mutex_unlock(&TCB.stop_mutex);

return(0);
}

// Thread routine called from thr_create() as a Bound Daemon Thread
void *MultWorker(void *arg)
{
  int row, col, j, start, stop, id, size;

  // Do this loop forever - or until all the Non-Daemon threads have exited
  while(true)
    {
      // Wait for some work to do
      mutex_lock(&TCB.start_mutex);

        while (!TCB.work2do)
          cond_wait(&TCB.start_cond, &TCB.start_mutex);

        // decrement the work to be done
        TCB.work2do--;

        // get a unique id for work to be done
        id = TCB.queue++;
```

```
      mutex_unlock(&TCB.start_mutex);

      // set up the boundary for matrix operation - based on the unique id
      size = TCB.a->getsize();
      start = id * (int)(size/TCB.total_threads);
      stop = start + (int)(size/TCB.total_threads) - 1;
      if (id == TCB.total_threads - 1) stop = size - 1;

      // print what this thread will work on
      //cout << "Thread " << thr_self() << ": Start Row = " << start
      //     << ", Stop Row = " << stop << endl << flush;
      // Do the matrix multiply - within the bounds set above

      for (row=start; row<=stop; row++)
          for (col = 0; col < size; col++)
              for (j = 0; j < size; j++)
                  TCB.c->getdata()[row*size+col] +=
                  TCB.a->getdata()[row*size+j] *
                  TCB.b->getdata()[j*size+col];

      // signal the main thread that this thread is done with the work
      mutex_lock(&TCB.stop_mutex);
      TCB.thrs_running--;
      cond_signal(&TCB.stop_cond);
      mutex_unlock(&TCB.stop_mutex);
    }
  return 0;
}
```

The source to Matrix.h:

```
#ifndef _matrix_h_
#define _matrix_h_

class Matrix
{
int matsize;
char matid;
```

```
double *data;

public:
Matrix(char id, int size);
virtual ~Matrix();
int getsize() {return(matsize);}
double *getdata() {return(data);}
void fill();
void clear();
void print(ostream &s) const;
};

// Function Prototypes
MatMult(Matrix &a, Matrix &b, Matrix &c);       // Matrix Multiply
void *MultWorker(void *arg);                    // Matrix Thread Function
ostream &operator<<(ostream &s, const Matrix &mat);  // Overloaded output
void SetMaxThreads(int num); // Sets the number of threads to use

#endif _matrix_h_
```

This example may look complicated at first, but spend some time here and make sure you understand how this program works. Also, you may want to try running this program with different size matrices and a different number of threads. Here is an example of some test runs, run on a SPARCstation 10 with four 50 MHz superSPARC CPUs:

```
> Matrix 400 1
    Matrix size: [400x400]
    Number of worker threads: 1
    Matrix multiplication time = 44.3368 seconds = 44336817000 nanoseconds
    (This defines the baseline for efficiency.)


> Matrix 400 2
    Matrix size: [400x400]
    Number of worker threads: 2
    Matrix multiplication time = 22.1987 seconds = 22198718000 nanoseconds
    (99.9% efficiency)
```

```
> Matrix 400 3
    Matrix size: [400x400]
    Number of worker threads: 3
    Matrix multiplication time = 14.8932 seconds = 14893245000 nanoseconds
     (99% efficiency)


> Matrix 400 4
    Matrix size: [400x400]
    Number of worker threads: 4
 Matrix multiplication time = 11.6228 seconds = 11622836000 nanoseconds
     (99% efficiency)


> Matrix 400 6
    Matrix size: [400x400]
    Number of worker threads: 6
    Matrix multiplication time = 13.1921 seconds = 13192145000 nanoseconds
     (75% efficiency)
```

Going from one thread to four threads reduced the time needed to perform the matrix multiplication. Also note that running with six threads did not cut the runtime down any more than four threads did, because the workstation had only four CPUs and the extra threads created a scheduling overhead.

Because the multiply routine uses a single global structure, it is not reentrant itself. For CPU-intensive problems such as this, that is not a major problem. Doing one multiply will completely saturate the machine, so there is nothing to be gained from running multiple versions of the multiply routine concurrently.