APPENDIX **D**

C++ Primer

IN THIS APPENDIX

First, let's get the pronunciation of *primer* out of the way. For many years, I pronounced *primer* as though it rhymed with *timer*, but the truth is, that's wrong. My friend Mitch Waite (the founder of Waite Group) has made a pretty good living off the word *primer*, but he told me one day that an English author of his from the U.K. said that he was pronouncing the word wrong. The correct pronunciation of *primer* rhymes with *trimmer*, as in "prim and proper." *Primer* the way we were pronouncing it meant the stuff you put on before you paint, or the first stage in an explosive process. In any case, I don't know if I will ever say it right—"prime-er" just sounds better!

TIP

If you're a C++ programmer, you might be asking "Why does André always use C?" The answer is simple—C is easier to understand, and that's all there is to it. C++ programmers obviously know C because it's a subset, and most game programmers learn C before C++.

What Is C++?

C++ is simply C upgraded with *object-oriented (OO)* technology. It's really nothing more than a superset of C. C++ has the following major upgrades:

- Classes
- Inheritance
- Polymorphism

Let's take a quick look at each. *Classes* are simply a way of combining both data and functions. Normally when you program in C you have data structures to hold data, and functions that operate on the data, as shown in Part A of Figure D.1. However, with C++, both data and the functions to operate on the data are contained within a class, as shown in Part B of Figure D.1. Why is this good? Well, because you can think of a class as an object that has properties and can perform actions. It's just a tighter and more abstract way of thinking.

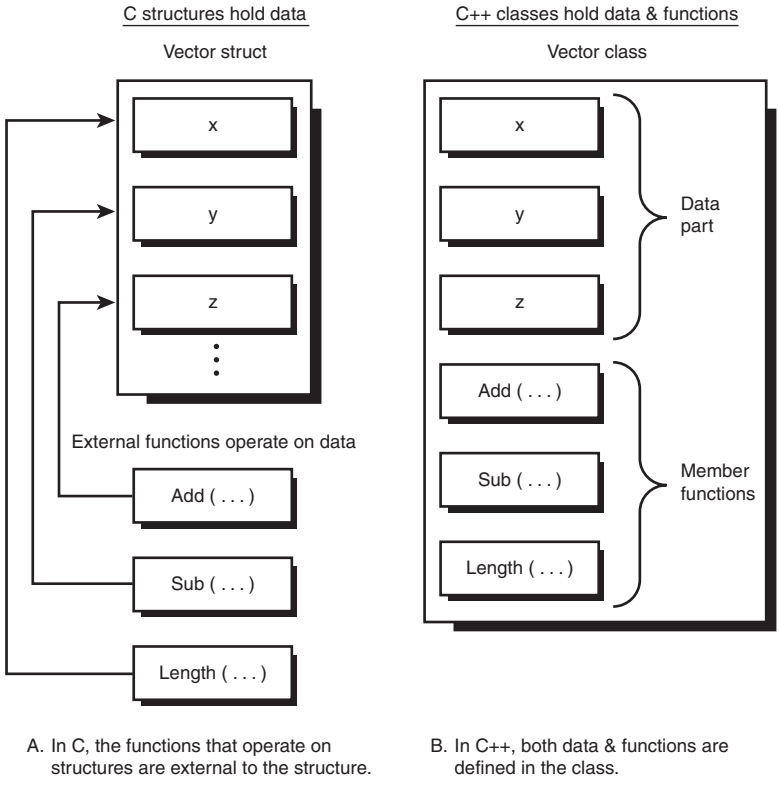

FIGURE D.1 The structure of a class.

The next cool thing about C++ is *inheritance*. After you create classes, they give you the abstract ability to create relationships between class objects and base one object or class upon another. It's done all the time in real life, so why not in software? For example, you might have a class called `person` that contains data about the person and maybe some class methods to operate on the data (don't worry about that for now). The point is, a person is fairly generic. But the power of inheritance comes into play when you want to create two different types of people: a software engineer and a hardware engineer, for example. Let's call them `sengineer` and `hengineer`.

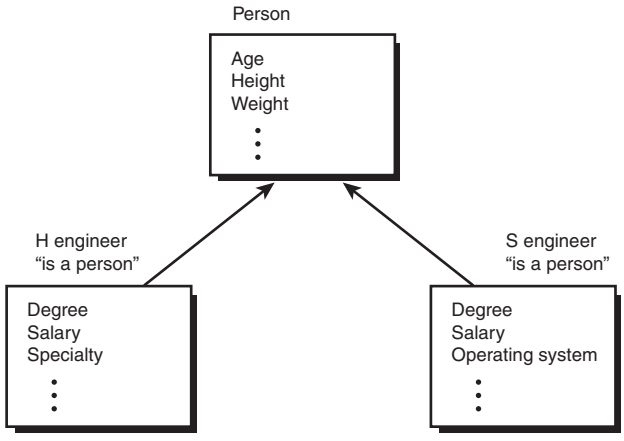

FIGURE D.2 Class inheritance.

Figure D.2 shows the relationship between `person`, `sengineer`, and `hengineer`. See how the two new classes are based on `person`? Both `sengineer` and `hengineer` are `persons`, but with extra data. Thus, you inherit the properties of a `person`, but add new ones to create `sengineer` and `hengineer`. This is the basis of inheritance—you build up more complex objects from pre-existing ones. In addition, there is *multiple inheritance*, which enables you to build a new object as a set of subclasses.

The third and last big deal about C++ and OO programming is *polymorphism*, which means "many forms." In the context of C++, polymorphism means that a function or operator means different things depending on the situation. For example, you know that the expression `(a + b)` in straight C means to add `a` and `b` together. You also know that `a` and `b` *must* be built in types like `int`, `float`, `char`, and `short`. In C, you can't define a new type and then say `(a + b)`. In C++, you can! Therefore, you can overload operators like `+`, `-`, `*`, `/`, `[]`, and so on, and make them do different things, depending on the data.

Furthermore, you can overload functions. For example, suppose you write a function `Compute()` like this:

```
int Compute (float x, float y)
{
// code
}
```

This function takes two floats, but if you send it integers, they're simply converted to floats and then passed to the function. Hence, you lose data. However, in C++, you can do this:

```
int Compute (float x, float y)
{
// code
}

int Compute (int x, int y)
{
// code
}
```

Even though the functions have the same names, they take different types. The compiler thinks they are completely different functions, so a call with integers calls the second function, while a call with floats calls the first. If you call the function with a float and an integer, things get more complex. Promotion rules then come into play, and the compiler decides which one to call using these rules.

That's really all there is to C++. Of course, there is some added syntax and lots of rules about all this stuff, but for the most part, all of it has to do with implementing these three new concepts—easy, huh?

The Minimum You Need to Know About C++

C++ is an extremely complex language, and using the new technologies too much, too fast can create totally unreliable programs with all kinds of memory leaks, performance issues, and so on. The problem with C++ is that it is a language of black boxes. There are a number of processes that go on behind the scenes, and you might never find bugs that you have created because of this. However, if you start off using just a little C++ here and there, and then add new features to your repertoire as you need them, you will be fine.

The only reason that I even wrote this appendix on C++ is that DirectX is based on it. However, most of the C++ is encapsulated in wrappers and COM interfaces that you communicate with via function pointer calls—that is, calls of the form `interface->function()`. If you have gotten this far in the book, you must have just dealt with that weird syntax. Moreover, the chapter on COM (Component Object Model) should have eased your nerves on the subject. In any event, we are going to cover just the basics, so you can better understand C++, talk about it with your friends, and have a good working knowledge of what's available.

We are going to cover some new types and conventions, memory management, stream I/O, basic classes, function and operator overloading, and that's about it—but believe me, that's enough! So, let's get started…

New Types, Keywords, and Conventions

Let's start off with something simple—the new comment operator (`//`). This has become part of C, so you might already use it, but the `//` operator is a single line comment in C++:

Comments

```
// this is a comment
```

You can still use the old comment style, `/* */`, if you like:

```
/* a C style multi line comment

every thing in here is a comment

*/
```

Constants

To create a constant in standard C, you can do one of two things:

```
#define PI 3.14
```

or

```
float PI = 3.14;
```

The problem with the first method is that `PI` isn't a real variable with a type. It's only a symbol that the pre-processor uses to do a text replacement, so it has no type, no size, and so on. The problem with the second type definition is that it is writable. Thus, C++ has a new type called `const`, which is like a read-only variable:

```
const float PI = 3.14;
```

You can use `PI` anywhere you want—its type is `float`, and its size is `sizeof(float)`—but you can't overwrite it. This is really a much better way to make constants.

Referential Variables

In C, there will be many times when you want to change the value of a variable in a function, so you pass a pointer, like this:

```
int counter = 0;

void foo(int *x)
{
(*x)++;
}
```

And if you make a call to `foo(&counter)`, `counter` will be equal to 1 after the call. Hence, the function changes the value of the sent variable. This is so common a practice that C++ has a new type of variable to help make this easier to do. It's called a *reference* variable, and it's denoted by the address operator `&`.

```
int counter = 0;

void foo(int &x)
{
x++;
}
```

Interesting, huh? But how do we call the function? Like this:

```
foo(counter);
```

Notice that we don't need to put the `&` in front of `counter` anymore. What happens is that `x` becomes an alias for whichever variable is sent. Therefore, `counter` is `x`, and no `&` is needed during the call.

You can also create references outside of functions like this:

```
int x;

int &x_alias = x;
```

`x_alias` is an alias to `x`. Wherever and however you use `x`, you can use `x_alias`—they are identical. I don't see much need for this, though.

Creating Variables On the Fly

One of the coolest new features of C++ is the capability to create variables within code blocks, and not just at the global or function level. For example, here's how you might write a loop in C:

```
void Scan(void)
{
int index;

// lots of code here…

// finally our loop
for (index = 0; index < 10; index++)
     Load_Data(index);

// more code here…

} // end Scan
```

There is nothing wrong with the code. However, `index` is only used as a loop index in one code segment. The designers of C++ saw this as non-robust, and felt that variables should be defined closer to where they are used. Moreover, a variable that is used in one code block shouldn't be visible to other code blocks. For example, if you have a set of code blocks like this:

```
void Scope(void)
{
int x = 1, y = 2; // global scope
printf("\nBefore Block A: Global Scope x=%d, y=%d",x,y);
    { // Block A
    int x = 3, y = 4;
    printf("\nIn Block A: x=%d, y=%d",x,y);
    } // end Block A
printf("\nAfter Block A: Global Scope x=%d, y=%d",x,y);
    { // Block B
    int x = 5, y = 6;
    printf("\nIn Block B: x=%d, y=%d",x,y);
    } // end Block B
printf("\nAfter Block B: Global Scope x=%d, y=%d",x,y);
} // end Scope
```

There are three different versions of `x` and `y`. The first `x` and `y` are globally defined. However, after code block A is entered, they go out of scope in light of the local `x` and `y` that come into scope. Then, when code block A exits, the old `x` and `y` come back into scope (with their old values), and the same process occurs for block B. With block-level scoping, you can better localize variables and their use. Moreover, you don't have to keep thinking of new variable names. You can continue to use `x`, `y`, and so on without worrying that the new variables will corrupt globals with the same name.

The really cool thing about this new variable scoping is that you can create a variable on the fly in code. For example, take a look at the same `for()` loop based on `index`, but using C++:

```
// finally our loop
for (int index = 0; index < 10; index++)
     Load_Data(index);
```

Isn't that the coolest? I defined `index` right as I used it, rather than at the top of the function. Just don't get too carried away with it.

Memory Management

C++ has a new memory management system based on the operators `new` and `delete`. They are equivalent to `malloc()` and `free()` for the most part, but are much smarter, because they take into consideration the type of data being requested/deleted. Here's an example:

To allocate 1000 `int`s from the heap in C:

```
int *x = (int*)malloc(1000*sizeof(int));
```

What a mess! Here's the same thing in C++:

```
int *x = new int[1000];
```

Much nicer, huh? You see, `new` already knows to send back a pointer to `int`—that is, an `int*`—so we don't have to cast it. Now, to release the memory in C, you would do this:

```
free(x);
```

In C++, you would do this:

```
delete x;
```

About the same, but the cool part is the `new` operator. Also, use either C or C++ to allocate memory. Don't mix calls to `new` with calls to `free()` and `malloc()` with `delete`.

Stream I/O

I love `printf()`. Nothing is more pure than

```
printf("\nGive me some sugar baby.");
```

The only problem with `printf()` is all the format specifiers like `%d`, `%x`, `%u`, and so forth. They're hard to remember. In addition, `scanf()` is even worse, because you can really mess up if you forget to use the address of a variable for storage. For example:

```
int x;

scanf("%d",x);
```

This is incorrect! We need the address of `x` or `&x`. Therefore, the correct syntax is

```
scanf("%d",&x);
```

I'm sure you have made this mistake. The only time you don't have to use the address operator is when you're working with strings, because the name is the address. In any case, this is reason why the new `IOSTREAM` class was created in C++. It knows the types of

the variables, so you don't need to specify them anymore. The `IOSTREAM` class functions are defined in `IOSTREAM.H`, so you need to include it in your C++ programs to use it. Once you do, you will have access to the streams `cin`, `cout`, `cerr`, and `cprn`, as shown in Table D.1.

TABLE D.1 C++ I/O Streams

Stream Name	Device	C Name	Meaning
`cin`	Keyboard	`stdin`	Standard Input
`cout`	Screen	`stdout`	Standard Output
`cerr`	Screen	`stderr`	Standard Error
`cprn`	Printer	`stdprn`	Printer

Using the I/O streams is a bit weird, because they're based on the overloaded operators << and >>. These normally signify bit-shifting in C, but in the context of the I/O streams, they're used to send and receive data. Here are some examples of using the standard output:

```
int i;
float f;
char c;
char string[80];

// in C
printf("\nHello world!");

// in C++
cout << "\nHello world!";

// in C
printf("%d", i);

// in C++
cout << i;

// in C
printf("%d,%f,%c,%s", i, f, c, string);

// in C++
cout << i << "," << f << "," << c << "," << string;
```

Isn't that cool? You don't need any type specifier: `cout` already knows the type and does it for you. The only thing really weird about the syntax is the way C++ allows you to concatenate the << operator to the end of each operation. The reason for this is that each operation returns a stream itself, so you can add << forever. The only downside to using streams for simple printing is the way you have to separate variables and string constants, like the `","` that separates each variable. However, you can put the << on each line if you want, like this:

```
cout << i
     << ","
     << f
     << ","
     << c
     << ","
     << string;
```

Remember that in both C and C++, white space is discarded, so the above coding is legal.

The input stream works in much the same way, but with the >> operator instead. Here are some examples:

```
int i;
float f;
char c;
char string[80];

// in C
printf("\nWhat is your age?");
scanf("%d",&i);

// in C++
cout << "\nWhat is your age?";
cin >> i;

// in C
printf("\nWhat is your name and grade?");
scanf("%s %c", string, &c);

// in C++
cout << "\nWhat is your name and grade?";
cin >> string >> c;
```

A little nicer than C, isn't it. Of course, the `IOSTREAM` system has a million other functions and neat things that you can do with it, so check it out.

Classes

Classes in C++ are the most important addition to the language, and for the most part give C++ its OO zeal. As I discussed before, a `class` is simply a container of both data and *methods* (often called *member functions*) that operate on that data.

The New Struct in Town

Let's begin learning classes by starting with standard structures, but with a twist. In C, you defined a structure like this:

```
struct Point
{
int x,y;
};
```

You can then create an instance of a structure with this:

```
struct Point p1;
```

This creates an instance or object of the structure `Point` and names it `p1`. In C++, you don't need to use the `struct` keyword anymore to create an instance:

```
Point p1;
```

This creates an instance of the structure `Point` named `p1`. The reason for this is that C programmers have been creating types so they don't have to type `struct` anymore, like this:

```
typedef struct Point_tag
{
int x,y;
} Point;
```

Thus, the syntax

```
Point p1;
```

Classes are similar to the new structures in that you don't have to create a type—the definitions themselves are the types.

Just a Simple Class

A class in C++ is defined with the keyword class. Here's an example:

```
class Point
{
public:
int x,y;

};

Point p1;
```

This is almost identical to the `struct` version of `Point`. In fact, both versions of `p1` work in the exact same way. For example, to access data, you just use the normal syntax:

```
p1.x = 5;
p1.y = 6;
```

And of course, pointers work the same. So, if you define something like this:

```
Point *p1;
```

then you would have to allocate memory for it first with `malloc()` or `new`:

```
p1 = new Point;
```

Then you could assign values to `x` and `y` like this:

```
p1->x = 5;
p1->y = 6;
```

The bottom line is that for the most part, classes and structures are identical when accessing public data elements. The key term is *public*—what does this mean? If you noticed in my previous example of the `Point` class, defined as

```
class Point
{
public:
int x,y;
};
```

there is a keyword `public:` at the top of the definition before any declarations. This defines the visibility of the variables (and member functions). There are a number of visibility options, but usually only two are used: `public` and `private`.

Public Versus Private

If you place the keyword `public` at the top of all your class definitions and only have data in the classes, you have nothing more than a standard structure. That is, structures are

classes with public visibility. *Public visibility* means that anyone can look at the class data elements. As for code in the main, other functions, and member functions, the data is not hidden or encapsulated. *Private visibility*, on the other hand, gives you the capability to hide data that you don't want other functions that aren't part of the class to alter. For example, take a look at this class:

```
class Vector3D
{
public:
int x,y,z; // anyone can mess with these

private:
    int reference_count; // this is hidden

};
```

`Vector3D` has two different parts to it: the public data area and the private data area. The public data area has three fields that can be changed by anyone: `x`, `y`, and `z`. On the other hand, there is a hidden field in the private section called `reference_count`. This field is hidden to everything except the member functions of the class (there aren't any yet). Thus, if you were to write some code like this

```
Vector3D v;

v.reference_count = 1; // illegal!
```

the compiler would give you an error! So the question is, what good are private variables if you can't access them? Well, they're great for writing something like a black box class when you don't want or need the user to alter internal working variables. In that example, private is the way to go. However, to access the private members, you need to add member functions or methods to the class—this is where we jump off the deep end…

Class Member Functions (Methods)

A member function or method (depending on who you're talking to) is basically a function within a class that only works with the class. Here's an example:

```
class Vector3D
{
public:
int x,y,z; // anyone can mess with these

     // this is a member function
    int length(void)
         {
```

```
        return(sqrt(x*x + y*y + z*z);
        } // end length

private:
    int reference_count; // this is hidden

};
```

I have highlighted the member function `length()`. I have defined a function right in the class! Weird, huh? Let's see how to use it:

```
Vector3D v; // create a vector

// set the values
v.x = 1;
v.y = 2;
v.z = 3;

// here's the cool part
printf("\nlength = %d",v.length());
```

You call a class member function just like you access an element. And if `v` were a pointer, you would do this:

```
v->length();
```

Now, you might be saying, "I have about 100 functions that are going to have to access the class data—I can't possibly put them all in the class!" Well, you can if you want, but I agree that it would get messy. However, you can define class members function outside the class definition. We'll get to that in a minute. Right now, I want to add another member function to show how you might access that private data member `reference_count`:

```
class Vector3D
{
public:
int x,y,z; // anyone can mess with these

     // this is a member function
    int length(void)
         {
         return(sqrt(x*x + y*y + z*z);
         } // end length
```

```
    // data access member function
     void addref(void)
     {
     // this function increments the reference count
     reference_count++;

     } // end addref

private:
    int reference_count; // this is hidden

};
```

You talk to `reference_count` via the member function `addref()`. This may seem odd, but if you think about it, it's a good thing. Now the user can't do anything stupid to the data member. It always goes through your access function, which in this case only allows the caller to increment the `reference_count`, as in

```
v.addref();
```

The caller can't change the reference count, multiply it by a number, and so on, because `reference_count` is private. Only member functions of the class can access it—this is data hiding and encapsulation.

At this point, I think you're seeing the power of classes. You can fill them with data-like structure, add functions within the classes that operate on the data, and hide data—pretty cool! But it gets even better!

Constructors and Destructors

If you have been programming C for over a week, there's something that I'm sure you have had to do about a million times—initialize a structure. For example, say that you create a structure `Person`:

```
struct Person
{
int age;
char *address;
int salary;
};

Person people[1000];
```

Now you need to initialize 1,000 `people` structures. Maybe all you want to do is this:

```
for (int index = 0; index < 1000; index++)
{
people[index].age     = 18;
people[index].address = NULL;
people[index].salary  = 35000;

} // end for index
```

But what if you forget to initialize the data and then just use the structures? Well, you might wind up seeing your old friend General Protection Fault. You might also see it if you forget to initialize your data structures. Similarly, what if during the run of your program you allocate memory, and point the address field of a `Person` to the memory, like this?

```
people[20].address = malloc(1000);
```

Then you forget about it, and do this:

```
people[20].address = malloc(4000);
```

Oops! You just lost a thousand bytes of memory in never-never land. What you needed to do before allocating more memory was release the old memory with a call to `free()`:

```
free(people[20].address);
```

I think you have probably done this, too. C++ solves these housekeeping problems by giving you two new automatic functions that are called when you create a class: constructors and destructors.

Constructors are called when a class object is instantiated. For example, when this code is executed:

```
Vector3D v;
```

the default constructor is called, which doesn't do anything. Similarly, when `v` goes out of scope—that is, when the function `v` is defined in terminates, or if *v* is global when the program terminates—the default destructor is called, which also doesn't do anything. To see any action, we have to write a constructor and destructor. You don't have to if you don't want to, and you can define one or both.

Writing a Constructor

Let's use the `Person` structure converted to a class as an example:

```
class Person
{
public:
int age;
char *address;
int salary;

// this is the default constructor
// constructors can take a void, or any other set of parms
// but they never return anything, not even a void
Person()
      {
      age     = 0;
      address = NULL;
      salary  = 35000;
      } // end Person
};
```

Notice that the constructor has the same name as the class; in this case, `Person`. This is not a coincidence—it's a rule! Also, notice that the constructor returns nothing. This is also a must. However, the constructor can take parameters. In this case, there are no parameters. However, you can have constructors with parameters—in fact, you can have an infinite number of different constructors, each with a different calling list. This is how you can create various types of `Persons` with different calls. Anyway, to create a *Person* and have it automatically initialized, you just do this:

```
Person person1;
```

The constructor will be called automatically, and the following assignments will occur:

```
person1.age     = 0;
person1.address = NULL;
person1.salary  = 35000;
```

Cool, huh? Now, the power of the constructor comes into play when you code something like this:

```
Person people[1000];
```

The constructor will be called for every single instance of `Person`, and all 1,000 of them will be initialized without a single line of code on your part!

Now let's get a little more advanced. Remember how I told you that functions can be overloaded? Well, you can overload constructors, too. Suppose you want a constructor for

which you can set the age, address, and salary of a `Person` during its creation. You could do this:

```
class Person
{
public:
int age;
char *address;
int salary;

// this is the default constructor
// constructors can take a void, or any other set of parms
// but they never return anything, not even void
Person()
      {
      age     = 0; address = NULL; salary  = 35000;
      } // end Person

// here's our new more powerful constructor
Person(int new_age, char *new_address, int new_salary)
{
// set the age
age = new_age;

// allocate the memory for the address and set address
address = new char[strlen(new_address)+1];
strcpy(address, new_address);

// set salary
salary = new_salary;

} // end Person int, char *, int

};
```

Now we have two constructors: one that takes no parameters, and one that takes three (an `int`, a `char *`, and another `int`). Here's an example of creating a person that is 24 years old, lives at 500 Maple Street, and makes $52,000 a year.

```
Person person2(24,"500 Maple Street", 52000);
```

Isn't that the coolest? Of course, you might think you can initialize C structures as well with a different syntax, something like

```
Person person = {24, "500 Maple Street", 52000};
```

However, what about the memory allocation? What about the string copying, and so on? Straight C can do a blind copy, but that's it. C++ gives you the power to also run code, and logic when an object is created. This gives you much more control.

Writing a Destructor

After you have created an object, at some point it must die. This is where you might normally call a cleanup function in C, but in C++, the object cleans itself up with a call to its destructor. Writing a destructor is even simpler than writing a constructor because you have much less flexibility with destructors—they only have one form:

```
~classname();
```

No parameter, no return type—period. No exceptions! With this in mind, let's add a destructor to our `Person` class:

```
class Person
{
public:
int age;
char *address;
int salary;

// this is the default constructor
// constructors can take a void, or any other set of parms
// but they never return anything, not even void
Person()
      {
      age     = 0; address = NULL; salary  = 35000;
      } // end Person

// here's our new more powerful constructor
Person(int new_age, char *new_address, int new_salary)
{
// set the age
age = new_age;

// allocate the memory for the address and set address
address = new char[strlen(new_address)+1];
strcpy(address, new_address);
```

```
// set salary
salary = new_salary;

} // end Person int, char *, int


// here's our destructor
~Person()
       {
       free(address);
       } // end ~Person

};
```

I've highlighted the destructor. Notice there is nothing special about the code within it; I could have done anything that I wanted. With this new destructor, you don't have to worry about de-allocating memory. For example, in C, if you created a structure with internal pointers in a function and then exited the function without de-allocating the memory pointed to by the structure, that memory would be lost forever in never-never land—that's called a *memory leak*, as shown below with a C example:

```
struct
    {
    char *name;
    char *ext;
    } filename;

foo()
{
filename file; // here's a filename

file.name = malloc(80);
file.ext  = malloc(4);

} // end foo
```

The structure `file` is destroyed, but the 84 bytes we allocated are lost forever! But, in C++, with your destructor, this won't happen, because the compiler makes sure to call the destructor for you, de-allocating the memory!

Those are the basics about constructors and destructors, but there's a lot more. There are special constructors called copy constructors, assignment constructors, and so forth. But you have enough to get started. As for destructors, there's just the type I have shown you, so you're in good shape there.

The Scope Resolution Operator

There is a new operator in C++ called the *scope resolution operator*, represented by a double colon (`::`). It's used to make reference to class functions and data members at class scope. Don't worry too much about what that means—I'm just going to show you how to use it to define class functions outside the class.

Writing Class Member Functions Outside the Class Scope

Thus far, you've been defining class member functions right inside the class definition. Although this is totally acceptable for small classes, it gets to be a little problematic for large classes. Hence, you are free to write class member functions outside of the class, as long as you define them properly and let the compiler know that they are class functions and not normal file-level functions. You do this with the scope resolution operator and the following syntax:

```
return_type class_name::function_name(parm_list)
{
// function body
}
```

Of course, in the class itself you must still define the function with a prototype (minus the scope resolution operator and class name, of course), but you can hold off on the body until later. Let's try this with our `Person` class and see what we get. Here's the new class with the function bodies removed:

```
class Person
{
public:
int age;
char *address;
int salary;

// this is the default constructor
Person();

// here's our new more powerful constructor
Person(int new_age, char *new_address, int new_salary);

// here's our destructor
~Person();

};
```

And here are the function bodies which you would place with all your other functions after the class definition:

```
Person::Person()
{
// this is the default constructor
// constructors can take a void, or any other set of parms
// but they never return anything, not even void
age     = 0;
address = NULL;
salary  = 35000;

} // end Person

////////////////////////////////////////////////////

Person::Person(int new_age,
               char *new_address,
               int new_salary)
{
// here's our new more powerful constructor
// set the age
age = new_age;

// allocate the memory for the address and set address
address = new char[strlen(new_address)+1];
strcpy(address, new_address);

// set salary
salary = new_salary;

} // end Person int, char *, int

////////////////////////////////////////////////////

Person::~Person()
{
// here's our destructor
free(address);
} // end ~Person
```

TIP

Most programmers place a capital C before class names. I usually do, but I didn't want to trip you out. Thus, if I was programming, I probably would have called it `CPerson` instead of `Person`, or perhaps `CPERSON` in all caps.

Function and Operator Overloading

The last topic I want to talk about is *overloading*, which comes in two flavors: *function overloading* and *operator overloading*. I don't have time to explain operator overloading in detail, but I'll give you a general example. Imagine that you have our `Vector3D` class and you want to add two vectors, `v1 + v2`, and store the sum in `v3`. You might do something like this:

```
Vector3D v1 = {1,3,5},
         v2 = {5,9,8},
         v3 = {0,0,0};


// define an addition function, this could have
// been a class function
Vector3D Vector3D_Add(Vector3D v1, Vector3D v2)
{
Vector3D sum; // temporary used to hold sum

sum.x = v1.x+v2.x;
sum.y = v1.y+v2.y;
sum.z = v1.z+v2.z;

return(sum);

}// end Vector3D_Add
```

Then, to add the vectors with the function, you would write the following code:

```
v3 = Vector3D_Add(v1, v2);
```

This works, but it's crude. With C++ and operator overloading, you can actually overload the + operator and make a new version of it to add the vectors! So, you can write this:

```
v3 = v1+v2;
```

Cool, huh? The syntax of the overloaded operator function is below, so you can check it out, but you'll have to read a C++ book for the details:

```
class Vector3D
{
public:

int x,y,z; // anyone can mess with these

// this is a member function
int length(void)  {return(sqrt(x*x + y*y + z*z); }

// overloaded the + operator
Vector3D operator+(Vector3D &v2)
{
Vector3D sum; // temporary used to hold sum

sum.x = x+v2.x;
sum.y = y+v2.y;
sum.z = z+v2.z;

return(sum);
}

private:
    int reference_count; // this is hidden

};
```

Notice that the first parameter is implicitly the object, so the parameter list has only `v2`. Anyway, operator overloading is very powerful. With it, you can really create new data types and operators, so that you can perform all kinds of cool operations without making calls to functions.

You've already seen function overloading when I was talking about constructors. Function overloading is nothing more than writing two or more functions that have the same name but different parameter lists. Suppose you want to write a function called `plot pixel` that has the following functionality: If you call it without parameters, it simply plots a pixel at the current cursor position; if you call it with an `x,y`, it plots a pixel at the position `x,y`. Here's how you would code it:

```
int cursor_x, cursor_y; // global cursor position

// the first version of Plot_Pixel
void Plot_Pixel(void)
{
```

```
// plot a pixel at the cursor position
plot(cursor_x, cursor_y);
}

////////////////////////////////

// the second version of Plot_Pixel
void Plot_Pixel(int x, int y)
{
// plot a pixel at the sent position and update
// cursor
plot(cursor_x=x, cursor_y=y);
}
```

Now you can call the functions like this:

```
Plot_Pixel(10,10); // calls version 2

Plot_Pixel(); // calls version 1
```

TIP

The compiler knows the difference, because the real name of the functions is created by not only the function name, but also a mangled version of the parameter list, creating a unique name in the compiler's namespace.

Basic Templates

Templates have been around for a long time. In fact, you have probably used them or even invented them by accident in your programming endeavors. Let's start with a standard C example to illustrate what templates are and what problems they solve. Imagine you have the following set of math functions:

```
int add(int a, int b)
{
int sum = a+b;
return(sum);
} // end add

int mul(int a, int b)
{
int product = a*b;
return(product);
} // end mul
```

Well, that's great, but what if you want to have the same functional support for floats? Well, you could overload the functions like this:

```
float add(float a, float b)
{
float sum = a+b;
return(sum);
} // end add

float mul(float a, float b)
{
float product = a*b;
return(product);
} // end mul
```

But this gets to be tedious if you need to support more than a few data types. Additionally, if you copy the code over and over and create overloaded functions, a single bug from your "original" is copied multiple times, so it's a source disaster. This is why templates were invented: They are templates for a function (or class) that allow you to write the core logic for a function using a generic data type, and then the compiler creates new versions of the function on the fly as needed, so "write once" is the motto.

Here's an example of declaring a templated function for our `add()` program:

```
template<class T> T add(T a, T b)
{
T sum; // declare the generic type
// perform generic computation
sum = a + b;
// return results
return(sum);
} // end add
```

Let's analyze the function syntax carefully (I have bolded the tricky parts). To begin with, template functions start with the `template` keyword, followed by the `<class T>` declaration. `T` is just a dummy and could be anything. The `template` keyword tells the compiler that we are beginning a template function, and the `<class T>` tag tells the compiler that anytime it encounters the symbol `T`, it should replace it with whatever type is needed. Therefore, `T` is the generic data type that the template function will use to generate multiple functions. Next is the return value, which in this case also happens to be `T`. Thus, the function returns a generic type back; however, it could have been an intrinsic type like `int`, `float`, and so on. Following this is the function and parameter list. Notice that the function takes two parameters of type `T`; again, these are also generic. Here's an example:

```
int a = 1, b = 2;
int sum = add(a,b);
```

The compiler will generate a template function that takes two `int`s and returns an `int` with no help from us. Moreover, if you code the following lines

```
float a = 1, b = 2;
float sum = add(a,b);
```

the compiler will create yet another version of the template function that takes floats and returns a float!

In addition to template functions, there are template classes, but I am going to leave that up to you. For our needs, template functions are more than enough to help us generate generic functions for multiple data types.

TIP

Recently there has been an addition to the ANSI C++ standard to support templates called the STL, or Standard Template Library. STL has built-in support for numerous data types and advanced functionality. Definitely something you should try, but even though it's supposed to be an ANSI standard—watch out!

Introduction to Exception Handling

Most C programmers either create very robust error-handling code or none at all. For example, a rookie might try something like this:

```
char *alloc_mem(int num_bytes)
{
char *ptr = NULL;
ptr = (char *)malloc(num_bytes);
memset(ptr, 0, num_bytes);
return(ptr);
} // end alloc_mem
```

Of course, the problem with the function is that if `malloc()` fails, a `NULL` pointer is returned. Worse yet, `num_bytes` of memory is overwritten, and a protection fault occurs! A more robust implementation would be:

```
char *alloc_mem(int num_bytes)
{
char *ptr = NULL;
if ((ptr = (char *)malloc(num_bytes))!=NULL)
    {
    memset(ptr, 0, num_bytes);
    return(ptr);
```

```
    } // end if
return(NULL);
} // end alloc_mem
```

In this version of the function, if the memory can't be allocated, the memory is zeroed out. Moreover, the function returns `NULL` to indicate that something went wrong. These techniques, along with error files, `printf()`, and other tactics, all work fine. However, the problem that many C programmers face is that their error handling tends to get intertwined with their logic. This can become rather cumbersome. Additionally, recovering from an error at all levels of functional nesting might become complex. Alas, new error-handling facilities have been added to C++, called *exception handling*.

The Components of Exception Handling

Exception handling consists of three main components:

- Try blocks—These sections of code are where you want to "try" to detect an error. To delineate a try block, you simply enclose in with the `try` keyword followed by braces to enclose the block, as follows:

  ```
  try {
  // code to try goes here
  } // end try block
  ```

- Catch blocks—Catch blocks are the fielders for the try blocks—anything that is tried is "caught" by a catch block. Catch blocks are started with the `catch` keyword and a variable declaration of the type of information that is going to be sent, followed by braces to enclose the block, as follows:

  ```
  catch(type v) {
  // error handling code for the try block goes here
  // this catch will handle data of the class "type"
  } // end catch block
  ```

- Throw statements—Last but not least are the throw statements. These are the actual lines of code that initiate the error handling (throw the error, to keep the analogy going). In other words, when the code logic detects an error, it throws the error to the nearest catch block from the current try block. In essence, program flow immediately jumps to the catch block. Throw statements can throw any type of data—strings, `int`s, `float`s, classes, and so on. The syntax is

  ```
  throw(type v);
  ```

Let's try a simple example with all three components:

```
try {
// do an error prone calculation..
// error occurred! throw it
throw("there was an error in the calculation module");
} // end try

// the catch block MUST be directly below the try block
catch(const char *str) {
// take action!
printf("\nError Thrown: %s", str);
} // end catch block
```

In this example, we have the simple structure of first the try block, then the catch block, and then the throw statement throwing a string variable, or more specifically, a `const char *`. That's why the catch block has the type declared as a `const char *`. However, there is no reason why we can't throw and catch multiple data types, as in this example:

```
try {
// do an error prone calculation..
// has bad thing 1 occurred? throw it
if (bad_thing1)
   throw("there was an error in the calculation module");
// bad thing 2
if (bad_thing2)
   throw(12);
} // end try

// the catch blocks MUST be directly below the try block
catch(const char *strerror) {
// take action for string throws
printf("\nError Thrown: %s", strerror);
} // end catch block

catch(int ierror) {
// take action for integer throws
printf("\nError Thrown, Code: %d", ierror);
} // end catch block
```

In the above example, we have one try block, but two possible errors; one throws a string, and one throws an integer. This is no problem—we simply need a catch block that can take a string or integer.

The last feature of exception handling I want to mention is that exceptions can occur in functions as function calls. For example, look at the code below:

```
void func(void)
{
// bad things happen here!
switch(rand()%3)
      {
      case 0: // string bad thing
           throw("Something went wrong with the text!");
           break;
      case 1: integer bad thing
           throw(1);
           break;
      case 2: floating point bad thing
           throw(0.5);
           break;
      default: // all bad things!
      } // end switch
} // end func

void main(void)
{

try {
// call our worker function WITHIN the try block
func();
} // end try

// the catch blocks MUST be directly below the try block
catch(const char *strerror) {
// take action for string throws
printf("\nError Thrown: %s", strerror);
} // end catch block

catch(int ierror) {
// take action for integer throws
printf("\nError Thrown, Integer Code: %d", ierror);
} // end catch block

catch(int ferror) {
// take action for float throws
```

```
printf("\nError Thrown, Floating Code: %d", ferror);
} // end catch block

catch(…) {
// any other error goes here
} // end catch all

} // end main
```

TIP

Notice the `catch(...)` statement. This is called the *catch all* statement. Anything that doesn't have an explicit `catch()` block will end up here.

Referring to the code listing, the function `func()` is called from `main()` within a try block; therefore, anything that happens within that try block will be caught.

Obviously, exception handling has a lot more functionality than what's shown here, but this is start. Here are some quick rules to code by:

- Don't use exception handling on small projects.
- Don't add exception handling to a code base that doesn't have it already. That is, a heterogeneous error-handling system isn't a good idea.
- If you already have a robust error-handling system, there's no need to recode it with exception handling.
- Exception handling works best when it helps separate logic and error handling. In fact, this is its strength.

Summary

Well, that's it for my crash course in C++. You might not be a C++ coder now, but you have a good idea of what the language adds to the C repertoire of functionality. If you're interested in learning more, here's the URL of the free online book, *Thinking in C++ 2nd Edition,* by Bruce Eckel:

`http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`

It's all about the free stuff!