Destructors

Just as objects are created, so they are destroyed. (I think there's a Biblical passage to that effect.) If a class can have a constructor to set things up, it should also have a special member function to take the object apart and put back any resources that the constructor may have allocated. This function is known as the *destructor*.

A destructor has the name of the class preceded by a tilde (~). Like a constructor, the destructor has no return type (not even `void`), and it cannot be invoked like a normal function.

Technically, you can call the destructor explicitly: `s.~Student()`. However, this is rarely done, and it's needed only in advanced programming techniques, such as allocating an object on a predetermined memory address.

In logic, the tilde is sometimes used to mean "NOT" so the destructor is the "NOT constructor." Get it? Cute.

C++ automatically invokes the destructor in the following three cases:

- A local object is passed to the destructor when it goes out of scope.
- An object allocated off the heap is passed to the destructor when it is passed to delete.
- A global object is passed to the destructor when the program terminates.

Looking at an example

The following StudentDestructor program features a Student class that allocates memory off of the heap in the constructor. Therefore, this class needs a destructor to return that memory to the heap.

Any class whose constructor allocates resources, in particular, a class that allocates memory off of the heap, requires a destructor to put that memory back.

The program creates a few objects within a function `fn()` and then allows those objects to go out of scope and get destructed when the function returns. The function returns a pointer to an object that `fn()` allocates off of the heap. This object is returned to the heap back in `main()`.

```
//
//  StudentDestructor - this program demonstrates the use
//                of the destructor to return resources
//                allocated by the constructor
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  protected:
    double* pdGrades;
    int*    pnHours;

  public:
    // constructor - init the student to a legal state
    Student()
    {
        cout << "Constructing a Student object" << endl;
        pdGrades = new double[128];
        pnHours  = new int[128];
    }
   ~Student()
    {
        cout << "Destructing a Student object" << endl;
        delete[] pdGrades;
        pdGrades = 0;

        delete[] pnHours;
        pnHours = 0;
    }
};

Student* fn()
{
    cout << "Entering fn()" << endl;

    // create a student and initialize it
    cout << "Creating the Student s" << endl;
    Student s;

    // create an array of Students
    cout << "Create an array of 5 Students" << endl;
    Student sArray[5];

    // now allocate one off of the heap
    cout << "Allocating a Student from the heap" << endl;
```

```
    Student *pS = new Student;

    cout << "Returning from fn()" << endl;
    return pS;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // now allocate one off of the heap
    Student *pS = fn();

    // delete the pointer returned by fn()
    cout << "Deleting the pointer returned by fn()"
         << endl;
    delete pS;
    pS = 0;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The output from the program appears as follows:

```
Entering fn()
Creating the Student s
Constructing a Student object
Create an array of 5 Students
Constructing a Student object
Constructing a Student object
Constructing a Student object
Constructing a Student object
Constructing a Student object
Allocating a Student from the heap
Constructing a Student object
Returning from fn()
Destructing a Student object
Destructing a Student object
Destructing a Student object
Destructing a Student object
Destructing a Student object
Destructing a Student object
Deleting the pointer returned by fn()
Destructing a Student object
Press any key to continue . . .
```

The first message is from `fn()` itself as it displays an opening banner to let us know that control has entered the function. The `fn()` function then creates

an object `s` that causes the constructor to output a message. It then creates an array of five `Student` objects, which causes five more messages from the `Student` constructor. And finally `fn()` allocates one more `Student` object from the heap using the `new` keyword.

The last thing `fn()` does before returning is output an exit banner message. C++ automatically calls the destructor six times: five times for the elements of the array and once for the `s` object created at the beginning of the function.

You can't tell from the output, but the objects are destructed in the reverse order that they are constructed.

The destructor is not invoked for the object allocated off of the heap until `main()` deletes the pointer returned by `fn()`.

A memory block allocated off of the heap does not go out of scope when the pointer to it goes out of scope. It is the programmer's responsibility to make sure that the object is returned to the heap using the `delete` command.

Return a pointer to a non-array with `delete`. Return an array using `delete[]`.

Destructing data members

Data members are also destructed automatically. Destruction occurs in the reverse order to the order of construction: The body of the destructor is invoked first, and then the destructor for each data member in the reverse order that the data members were constructed.

To demonstrate this, I added a destructor to the TutorPairConstructor program. The entire listing is a bit lengthy to include here, but it is contained on the enclosed CD-ROM as TutorPairDestructor. I include just the TutorPair class here:

```
class TutorPair
{
  protected:
    Student s;
    Teacher t;

    int nNumberOfMeetings;

  public:
```

```
    TutorPair()
    {
        cout << "Constructing the TutorPair members"
             << endl;
        nNumberOfMeetings = 0;
    }
   ~TutorPair()
    {
        cout << "Destructing the TutorPair object"
             << endl;
    }
};

void fn()
{
    // create a TutorPair and initialize it
    cout << "Creating the TutorPair tp" << endl;
    TutorPair tp;

    cout << "Returning from fn()" << endl;
}
```

The output from this program appears as follows:

```
Creating the TutorPair tp
Constructing a Student object
Constructing a Teacher object
Constructing the TutorPair members
Returning from fn()
Destructing the TutorPair object
Destructing a Teacher object
Destructing a Student object
Press any key to continue . . .
```

This program creates the `TutorPair` object within the function `fn()`. The messages from the constructors are identical to the TutorPairConstructor program. The messages from the `TutorPair` destructor appear as control is returning to main, and they appear in the exact reverse of the order of messages from the constructors, coming first from `~TutorPair` itself, then from `~Teacher`, and finally from `~Student`.

Static data members

A special type of data member that deserves separate mention is known as a class member or static member because it is flagged with the keyword `static`:

```
class Student
{
  protected:
    static int nNumberOfStudents;
    int nSemesterHours;
    double dGrade;

  public:
    Student()
    {
        nSemesterHours = 0;
        dGrade = 0.0;

        // count how many Students
        nNumberOfStudents++;
    }
   ~Student()
    {
        nNumberOfStudents--;
    }
};

// allocate space for the static member; be sure to
// initialize it here (when the program starts) because
// the class constructor will not initialize it
int Student::nNumberOfStudents = 0;
```

Static members are a property of the class and not of each object. In this example, a single variable `Student::nNumberOfStudents` is shared by all `Student` objects. This example demonstrates exactly what such members are good for: In this case, `nNumberOfStudents` keeps a running count of the number of `Student` objects that currently exist.

Static members are initialized when the program starts. You can manipulate them from the constructor for each object — in this case, I increment the counter in the `Student` constructor and decrement it in the destructor. In general, you do not want to initialize a static member in the class constructor since it will get reinitialized every time an object is created.

Chapter 26

Making Constructive Arguments

In This Chapter

- Creating and invoking a constructor with arguments
- Overloading the constructor
- Constructing data members with arguments
- Looking forward to a new format of constructor in the 2009 standard

The `Student` class in Chapter 25 was extremely simple — almost unreasonably so. After all, a student has a name and a student ID as well as a grade point average and other miscellaneous data. I chose GPA as the data to model in Chapter 25 because I knew how to initialize it without someone telling me — I could just zero out this field. But I can't just zero out the name and ID fields; a no-named student with a null ID probably does not represent a valid student. Somehow I need to pass arguments to the constructor to tell it how to initialize fields that start out with a value that's not otherwise predictable.

Constructors with Arguments

C++ allows the program to define a constructor with arguments as shown here:

```
class Student
{
  public:
    Student(const char* pszNewName, int nNewID)
    {
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
        nID = nNewID;
    }
```

```
    ~Student()
     {
         delete[] pszName;
         pszName = 0;
     }

  protected:
    char* pszName;
    int   nID;
};
```

Here the arguments to the constructor are a pointer to an ASCIIZ string that contains the name of the new student and the student's ID. The constructor first allocates space for the student's name. It then copies the new name into the `pszName` data member. Finally, it copies over the student ID.

A destructor is required to return the memory to the heap once the object is destroyed. Any class that allocates a resource like memory in the constructor must return that memory in the destructor.

Remember, you can't call a constructor like you call a function, so you have to somehow associate the arguments to the constructor with the object when it is declared. The following code snippets show how this is done:

```
void fn()
{
    // put arguments next to object normally
    Student s1("Stephen Davis", 1234);

    // or next to the class name when allocating
    // an object from the heap
    Student* pS2 = new Student("Kinsey Davis", 5678);
}
```

The arguments appear next to the object normally and next to the class name when allocating an object off of the heap.

Looking at an example

The following NamedStudent program uses a constructor similar to the one shown in the snippets to create a `Student` object and display my, I mean his, name:

```
//
//  NamedStudent - this program demonstrates the use
//                 of a constructors with arguments
//
```

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;

class Student
{
  protected:
    char*  pszName;
    int    nID;

  public:
    Student(const char* pszNewName, int nNewID)
    {
        cout << "Constructing " << pszNewName << endl;
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
        nID = nNewID;
    }
   ~Student()
    {
        cout << "Destructing " << pszName << endl;
        delete[] pszName;
        pszName = 0;
    }

    // getName() - return the student's name
    const char* getName()
    {
        return pszName;
    }

    // getID() - get the student's ID
    int getID()
    {
        return nID;
    }
};

Student* fn()
{
    // create a student and initialize it
    cout << "Constructing a local student in fn()" <<endl;
    Student student("Stephen Davis", 1234);

    // display the student's name
    cout << "The student's name is "
         << student.getName() << endl;

    // now allocate one off of the heap
```

```
    cout << "Allocating a Student from the heap" << endl;
    Student *pS = new Student("Kinsey Davis", 5678);

    // display this student's name
    cout << "The second student's name is "
         << pS->getName() << endl;

    cout << "Returning from fn()" << endl;
    return pS;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // call the function that creates student objects
    cout << "Calling fn()" << endl;
    Student* pS = fn();
    cout << "Back in main()" << endl;

    // delete the object returned by fn()
    delete pS;
    pS = 0;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The `main()` program starts by outputting a message and then calling the function `fn()`. This function creates a student with the unlikely name "Stephen Davis" and an ID of 1234. The function then asks the object for its name just to prove that the name was accurately noted in the object. The function goes on to create another `Student` object, this time off of the heap, and similarly asks it to display its name.

The `fn()` function then returns control to `main()`; this causes the `student` object to go out of scope, which causes C++ to invoke the destructor. `main()` restores the memory returned from `fn()` to the heap using the keyword `delete`. This invokes the destructor for that object.

The constructor for class `Student` accepts a pointer to an ASCIIZ string and an `int` student ID. The constructor allocates a new character array from the heap and then copies the string passed it into that array. It then copies the value of the student ID.

Refer to Chapter 16 if you don't remember what an ASCIIZ string is or what `strlen()` does.

The destructor for class `Student` simply restores the memory allocated by the constructor to the heap by passing the address in `pszName` to delete[].

Use `delete[]` when restoring an array to the heap and `delete` when restoring a single object.

The `getName()` and `getID()` member functions are access functions for the name and ID. Declaring the return type of `getName()` as `const char*` (read "pointer to constant `char`") — as opposed to simply `char*` — means that the caller cannot change the name using the address returned by `getName()`.

Refer to Chapter 18 if you don't remember the difference between a `const char*` and a `char * const` (or if you have no idea what I'm talking about).

The output from this program appears as follows:

```
Calling fn()
Constructing a local student in fn()
Constructing Stephen Davis
The student's name is Stephen Davis
Allocating a Student from the heap
Constructing Kinsey Davis
The second student's name is Kinsey Davis
Returning from fn()
Destructing Stephen Davis
Back in main()
Destructing Kinsey Davis
Press any key to continue . . .
```

I've said it before (and you probably ignored me), but I really must insist this time: You need to invoke the preceding constructor in the debugger to get a feel for what C++ is doing with your declaration.

But what if you need both a named constructor and a default constructor? Keep reading.

Overloading the Constructor

You can have two or more constructors as long as they can be differentiated by the number and types of their arguments. This is called *overloading the constructor*.

Overloading a function means to define two or more functions with the same short name but with different arguments. Refer to Chapter 11 for a discussion of function overloading.

Thus, the following `Student` class from the OverloadedStudent program has three constructors:

```
//
//  OverloadedStudent - this program overloads the Student
//                  constructor with 3 different choices
//                  that vary by number of arguments
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;

class Student
{
  protected:
    char*  pszName;
    int    nID;
    double dGrade;          // the student's GPA
    int    nSemesterHours;


  public:
    Student(const char* pszNewName, int nNewID,
            double dXferGrade, int nXferHours)
    {
        cout << "Constructing " << pszNewName
             << " as a transfer student." << endl;
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
        nID = nNewID;
        dGrade = dXferGrade;
        nSemesterHours = nXferHours;
    }
    Student(const char* pszNewName, int nNewID)
    {
        cout << "Constructing " << pszNewName
             << " as a new student." << endl;
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
        nID = nNewID;
        dGrade = 0.0;
        nSemesterHours = 0;
    }
    Student()
    {
        pszName = 0;
```

```
        nID = 0;
        dGrade = 0.0;
        nSemesterHours = 0;
    }
   ~Student()
    {
        cout << "Destructing " << pszName << endl;
        delete[] pszName;
        pszName = 0;
    }

    // access functions
    const char* getName()
    {
        return pszName;
    }
    int getID()
    {
        return nID;
    }
    double getGrade()
    {
        return dGrade;
    }
    int getHours()
    {
        return nSemesterHours;
    }

    // addGrade - add a grade to the GPA and total hours
    double addGrade(double dNewGrade, int nHours)
    {
        double dWtdHrs = dGrade * nSemesterHours;
        dWtdHrs += dNewGrade * nHours;
        nSemesterHours += nHours;
        dGrade = dWtdHrs / nSemesterHours;
        return dGrade;
    }
};

int main(int nNumberofArgs, char* pszArgs[])
{
    // create a student and initialize it
    Student student("Stephen Davis", 1234);

    // now create a transfer student with an initial grade
    Student xfer("Kinsey Davis", 5678, 3.5, 12);

    // give both students a B in the current class
    student.addGrade(3.0, 3);
```

```
    xfer.addGrade(3.0, 3);

    // display the student's name and grades
    cout << "Student "
         << student.getName()
         << " has a grade of "
         << student.getGrade()
         << endl;

    cout << "Student "
         << xfer.getName()
         << " has a grade of "
         << xfer.getGrade()
         << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

Starting with the `Student` class, you can see that the first constructor within `Student` accepts a name, a student ID, and transfer credit in the form of an initial grade point average (GPA) and number of semester hours. The second constructor accepts only a name and ID; this constructor is intended for new students as it initializes the GPA and hours to zero. It's unclear what the third constructor is for — this default constructor initializes everything to zero.

The `main()` function creates a new student using the second constructor with the name "Stephen Davis"; then it creates a transfer student with the name "Kinsey Davis" using the second constructor. The program adds three hours of credit to both (just to show that this still works) and displays the resulting GPA.

The output from this program appears as follows:

```
Constructing Stephen Davis as a new student.
Constructing Kinsey Davis as a transfer student.
Student Stephen Davis has a grade of 3
Student Kinsey Davis has a grade of 3.4
Press any key to continue . . .
```

Notice how similar the first two `Student` constructors are. This is not uncommon. This case is one in which you can create an `init()` function that both constructors call (only the constructors are shown in this example for brevity's sake):

```
class Student
{
  protected:
    void init(const char* pszNewName, int nNewID,
              double dXferGrade, int nXferHours)
    {
        cout << "Constructing " << pszNewName
             << " as a transfer student." << endl;
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
        nID = nNewID;
        dGrade = dXferGrade;
        nSemesterHours = nXferHours;
    }
  public:
    Student(const char* pszNewName, int nNewID,
            double dXferGrade, int nXferHours)
    {
        init(pszNewName, nNewID, dXferGrade, nXferHours);
    }
    Student(const char* pszNewName, int nNewID)
    {
        init(pszNewName, nNewID, 0.0, 0);
    }

    // ...class continues as before...
};
```

In general, the `init()` function will look like the most complicated constructor. All simpler constructors call `init()` passing default values for some of the arguments, such as a 0 for transfer grade and credit for new students.

You can also default the arguments to the constructor (or any function for that matter) as follows:

```
class Student
{
  public:
    Student(const char* pszNewName, int nNewID,
            double dXferGrade = 0.0, int nXferHours = 0);

    // ...and so it goes...
};
```

C++ will supply the defaulted arguments if they are not provided in the declaration. However, default arguments can generate strange error messages and are beyond the scope of this book.

You can also invoke one constructor from another starting with the C++ 2009 standard. However, as of this writing, no compiler that I know of supports this feature.

The Default default Constructor

As far as C++ is concerned, every class must have a constructor; otherwise, you can't create any objects of that class. If you don't provide a constructor for your class, C++ should probably just generate an error, but it doesn't. To provide compatibility with existing C code, which knows nothing about constructors, C++ automatically provides an implicitly defined default constructor (sort of a *default* default constructor) that invokes the default constructor for any data members. Sometimes I call this a Miranda constructor. You know, "If you cannot afford a constructor, a constructor will be provided for you."

If your class already has a constructor, however, C++ doesn't provide the automatic default constructor. (Having tipped your hand that this isn't a C program, C++ doesn't feel obliged to do any extra work to ensure compatibility.)

The result is: If you define a constructor for your class but you also want a default constructor, you must define it yourself.

The following code snippets help demonstrate this principle. The following is legal:

```
class Student
{
    // ...all the same stuff but no constructors...
};

void fn()
{
    Student s; // create Student using default constructor
}
```

Here, the object `s` is built using the default constructor. Because the programmer has not provided a constructor, C++ provides a default constructor that doesn't really do anything in this case.

However, the following snippet does not compile properly:

```
class Student
{
  public:
    Student(const char* pszName);

    // ...all the same stuff...
};

void fn()
{
    Student s; // doesn't compile
}
```

The seemingly innocuous addition of the `Student(const char*)` constructor precludes C++ from automatically providing a `Student()` constructor with which to build the `s` object. Now the compiler complains that it can no longer find `Student::Student()` with which to build `s`. Adding a default constructor solves the problem:

```
class Student
{
  public:
    Student(const char* pszName);
    Student();

    // ...all the same stuff...
};

void fn()
{
    Student s; // this does compile
}
```

It's just this type of illogic that explains why C++ programmers make the really big bucks.

Constructing Data Members

In the preceding examples, all of the data members have been simple types, like `int` and `double` and arrays of `char`. With these simple types it's sufficient to just assign the variable a value within the constructor. But what if the class contains data members of a user-defined class? There are two cases to consider here.

Initializing data members with the default constructor

Consider the following example:

```
class StudentID
{
  protected:
    static int nBaseValue;
    int        nValue;

  public:
    StudentID()
    {
        nValue = nBaseValue++;
    }

    int getID()
    {
        return nValue;
    }
};

// allocate space for the class property
int StudentID::nBaseValue = 1000;

class Student
{
  protected:
    char*        pszName;
    StudentID    sID;

  public:
    Student(const char* pszNewName)
    {
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
    }
   ~Student()
    {
        delete pszName;
        pszName = 0;
    }

    // getName() - return the student's name
    const char* getName()
    {
```

```
        return pszName;
    }

    // getID() - get the student's ID
    int getID()
    {
        return sID.getID();
    }
};
```

The class `StudentID` is designed to allocate student IDs sequentially. The class retains the "next value" in a static variable `StudentID::nBaseValue`.

Static data members, also known as class members, are shared among all objects.

Each time a `StudentID` is created, the constructor assigns `nValue` the "next value" from `nBaseValue` and then increments `nBaseValue` in preparation for the next time the constructor is called.

The `Student` class has been updated so that the `sID` field is now of type `StudentID`. The constructor now accepts the name of the student but relies on `StudentID` to assign the next sequential ID each time a new `Student` object is created.

The constructor for each data member, including `StudentID`, is invoked before control is passed to the body of the `Student` constructor.

All the `Student` constructor has to do is make a copy of the student's name — the `sID` field takes care of itself.

Initializing data members with a different constructor

So now the boss comes in and wants an addition to the program. Now she wants to update the program so that it can assign a new student ID instead of always accepting the default value handed over by the `StudentID` class.

Accordingly, I make the following changes:

```
class StudentID
{
  protected:
    static int nBaseValue;
    int        nValue;
```

```
  public:
    StudentID(int nNewID)
    {
        nValue = nNewID;
    }
    StudentID()
    {
        nValue = nBaseValue++;
    }

    int getID()
    {
        return nValue;
    }
};

// allocate space for the class property
int StudentID::nBaseValue = 1000;

class Student
{
  protected:
    char*        pszName;
    StudentID    sID;

    void initName(const char* pszNewName)
    {
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
    }

  public:
    Student(const char* pszNewName, int nNewID)
    {
        initName(pszNewName);
        StudentID sID(nNewID);
    }
    Student(const char* pszNewName)
    {
        initName(pszNewName);
    }
   ~Student()
    {
        delete[] pszName;
        pszName = 0;
    }

    // getName() - return the student's name
    const char* getName()
    {
        return pszName;
    }
```

```
    // getID() - get the student's ID
    int getID()
    {
        return sID.getID();
    }
};
```

I added a constructor to `StudentID` to allow the caller to pass a value to use for the student ID rather than accept the default. Now, if the program doesn't provide an ID, the student is assigned the next sequential ID. If the program does provide an ID, however, then it is used instead, and the static counter is left untouched.

I also added a constructor to `Student` to allow the program to provide a `studentID` when the student is created. This `Student(const char*, int)` constructor first initializes the student's name and then invokes the `StudentID(int)` constructor on `sID`.

When I execute the program, however, I am disappointed to find that this seems to have made no apparent difference. Students are still assigned sequential student IDs whether or not they are passed a value to use instead.

The problem, I quickly realize, is that the `Student(const char*, int)` constructor is not invoking the new `StudentID(int)` constructor on the data member `sID`. Instead, it is creating a new local object called `sID` within the constructor, which it then immediately discards without any effect on the data member of the same name.

Remember that the constructor for the data members is called before control is passed to the body of the constructor. Rather than create a new value locally, I need some way to tell C++ to use a constructor other than the default constructor when creating the data member `sID`.C++ uses the following syntax to initialize a data member with a specific constructor:

```
class Student
{
  public:
    Student(const char* pszName,
            int nNewID) : sID(nNewID)
    {
        initName(pszName);
    }

    // ...remainder of class unchanged...
};
```

The data member appears to the right of a colon used to separate such declarations from the arguments to the function but before the open brace of the function itself. This causes the `StudentID(int)` constructor to be invoked, passing the `nNewID` value to be used as the new student ID.

Looking at an example

The following CompoundStudent program creates one `Student` object with the default, sequential student ID, while assigning a specific student ID to a second `Student` object:

```
//
//  CompoundStudent - this version of the Student class
//                    includes a data member that's also
//                    of a user defined type
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;

class StudentID
{
  protected:
    static int nBaseValue;
    int        nValue;

  public:
    StudentID()
    {
        nValue = nBaseValue++;
    }

    StudentID(int nNewValue)
    {
        nValue = nNewValue;
    }

    int getID()
    {
        return nValue;
    }
};

// allocate space for the class property
int StudentID::nBaseValue = 1000;
```

```
class Student
{
  protected:
    char*        pszName;
    StudentID    sID;

    void initName(const char* pszNewName)
    {
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
    }

  public:
    Student(const char* pszNewName,
                int nNewID) : sID(nNewID)
    {
        initName(pszNewName);
    }
    Student(const char* pszNewName)
    {
        initName(pszNewName);
    }
   ~Student()
    {
        delete[] pszName;
        pszName = 0;
    }

    // getName() - return the student's name
    const char* getName()
    {
        return pszName;
    }

    // getID() - get the student's ID
    int getID()
    {
        return sID.getID();
    }
};


int main(int nNumberofArgs, char* pszArgs[])
{
    // create a student and initialize it
    Student student1("Stephen Davis");
```

```
    // display the student's name and ID
    cout << "The first student's name is "
         << student1.getName()
         << ", ID is "
         << student1.getID()
         << endl;

    // do the same for a second student
    Student student2("Janet Eddins");
    cout << "The second student's name is "
         << student2.getName()
         << ", ID is "
         << student2.getID()
         << endl;

    // now create a transfer student with a unique ID
    Student student3("Tiffany Amrich", 1234);
    cout << "The third student's name is "
         << student3.getName()
         << ", ID is "
         << student3.getID()
         << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The `Student` and `StudentID` classes are similar to those shown earlier. The `main()` function creates three students, the first two using the `Student(const char*)` constructor that allocates the default student ID. The third student is created using the `Student(const char*, int)` constructor and passed an ID of 1234. The resulting display confirms that the default IDs are being allocated sequentially and that the third student has a unique ID.

```
The first student's name is Stephen Davis, ID is 1000
The second student's name is Janet Eddins, ID is 1001
The third student's name is Tiffany Amrich, ID is 1234
Press any key to continue . . .
```

The `:` syntax here can also be used to initialize simple variables if you prefer:

```
class SomeClass
{
  protected:
    int nValue;
    const double PI;

  public:
    SomeClass(int n) : nValue(n), PI(3.14159) {}
};
```

Here, the data member `nValue` is initialized to `n`, and the constant `double` is initialized to 3.14159.

In fact, this is the only way to initialize a data member flagged as `const`. You can't put a `const` variable on the left-hand side of an assignment operator.

Notice that the body of the constructor is now empty since all the work is done in the header; however, the empty body is still required (otherwise, the definition would look like a prototype declaration).

New with C++ 2009

Starting with the 2009 standard, you can initialize data members to a constant value in the declaration itself, as in the following:

```
class SomeClass
{
  protected:
    int nValue;
    const double PI = 3.14159;
    char* pSomeString = new char[128];

  public:
    SomeClass(int n) : nValue(n) {}
};
```

The effect is the same as if you had written the constructor as follows:

```
class SomeClass
{
  protected:
    int nValue;
    const double PI;
    char* pSomeString;

  public:
    SomeClass(int n)
      : nValue(n), PI(3.14159), pSomeString(new char[128])
    {}
};
```

The earlier assignment format is easier to read and just seems more natural (it is accepted by other C++–like programming languages such as Java and C#). However, as of this writing, this format is not yet accepted by any C++ compiler, including the one enclosed in this book.

Chapter 27

Coping with the Copy Constructor

In This Chapter

- Letting C++ make copies of an object
- Creating your own copy constructor
- Making copies of data members
- Avoiding making copies completely

The constructor is a special function that C++ invokes when an object is created in order to allow the class to initialize the object to a legal state. Chapter 25 introduces the concept of the constructor. Chapter 26 demonstrates how to create constructors that take arguments. This chapter concludes the discussion of constructors by examining a particular constructor known as the copy constructor.

Copying an Object

A *copy constructor* is the constructor that C++ uses to make copies of objects. It carries the name `X::X(const X&)`, where `X` is the name of the class. That is, it's the constructor of class `X` that takes as its argument a reference to an object of class `X`. I know that sounds pretty useless, but let me explain why you need a constructor like that on your team.

A reference argument type like `fn(X&)` says, "pass a reference to the object" rather than "pass a copy of the object." I discuss reference arguments in Chapter 23.

Think for a minute about the following function call:

```
void fn(Student s)
{
    // ...whatever fn() does...
}

void someOtherFn()
{
    Student s;
    fn(s);
};
```

Here the function `someOtherFn()` creates a `Student` object and passes a copy of that object to `fn()`.

By default, C++ passes objects by value, meaning that it must make a copy of the object to pass to the functions it calls (refer to Chapter 23 for more).

Consider that creating a copy of an object means creating a new object and, by definition, means invoking a constructor. But what would the arguments to that constructor be? Why, a reference to the original object. That, by definition, is the copy constructor.

The default copy constructor

C++ provides a default copy constructor that works most of the time. This copy constructor does a member-by-member copy of the source object to the destination object.

A member-by-member copy is also known as a *shallow copy* for reasons that soon will become clear.

There are times when copying one member at a time is not a good thing, however. Consider the `Student` class from Chapter 26:

```
class Student
{
  protected:
    char* pszName;
    int   nID;

  // ...other stuff...
};
```

Copying the `int` data member `nID` from one object to another is no problem. However, copying the pointer `pszName` from the source to the destination object could cause problems.

For example, what if `pszName` points to heap memory (which it almost surely does)? Now you have two objects that both point to the same block of memory on the heap. This is shown in Figure 27-1.

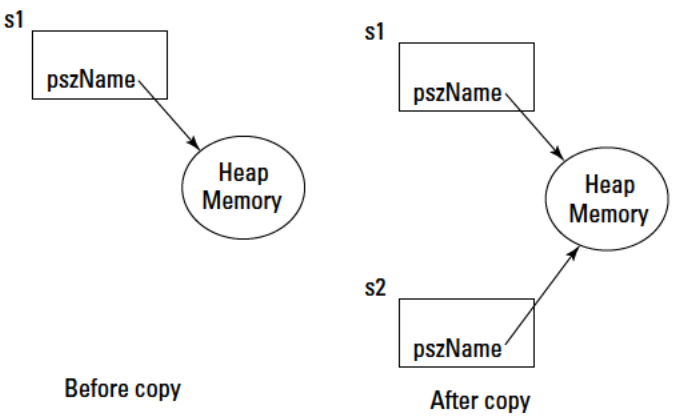


Figure 27-1: By default, C++ performs a member-by-member, "shallow" copy to create copies of objects, such as when passing an object to a function.

When the copy of the `Student` object goes out of scope, the destructor for that class will likely delete the `pszName` pointer, thereby returning the block of memory to the heap, even though the original object is still using that memory. When the original object deletes the same pointer again, the heap gets messed up, and the program is sure to crash with a bizarre and largely misleading error message.

Looking at an example

The following ShallowStudent program demonstrates how making a shallow copy can cause serious problems:

```
//
//  ShallowStudent - this program demonstrates why the
//                   default shallow copy constructor
//                   isn't always the right choice.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;
```

```
class Student
{
  protected:
    char*  pszName;
    int    nID;

  public:
    Student(const char* pszNewName, int nNewID)
    {
        cout << "Constructing " << pszNewName << endl;
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
        nID = nNewID;
    }
   ~Student()
    {
        cout << "Destructing " << pszName << endl;
        delete[] pszName;
        pszName = 0;
    }

    // access functions
    const char* getName()
    {
        return pszName;
    }
    int getID()
    {
        return nID;
    }
};

void someOtherFn(Student s)
{
    // we don't need to do anything here
}

void someFn()
{
    Student student("Adam Laskowski", 1234);
    someOtherFn(student);

    cout << "The student's name is now "
         << student.getName() << endl;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    someFn();
```

```
    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This deceptively simple program contains a serious problem. The function `main()` does nothing more than call the function `someFn()`. This function creates a local `student` object and passes it by value to the function `someOtherFn()`. This second function does nothing except return to the caller. The `someFn()` function then displays the name of the `student` and returns to `main()`.

The output from the program shows some interesting results:

```
Constructing Adam Laskowski
Destructing Adam Laskowski
The student's name is now X$±
Destructing X$±
Press any key to continue . . .
```

The first message comes from the `Student` constructor as the `student` object is created at the beginning of `someFn()`. No message is generated by the default copy constructor that's called to create the copy of `Student` for `someOtherFn()`. The destructor message is invoked at the end of `someOtherFn()` when the local object `s` goes out of scope.

The output message in `someFn()` shows that the object is now messed up as the memory allocated by the `Student` constructor to hold the student's name has been returned to the heap. The subsequent destructor that's invoked at the end of `someFn()` verifies that things are amiss.

This type of error is normally fatal (to the program, not the programmer). The only reason this program didn't crash is that it was about to stop anyway.

Creating a Copy Constructor

Classes that allocate resources in their constructor should normally include a copy constructor to create copies of these resources. For example, the `Student` copy constructor should allocate another block of memory off the heap for the name and copy the original object's name into this new block. This is shown in Figure 27-2.

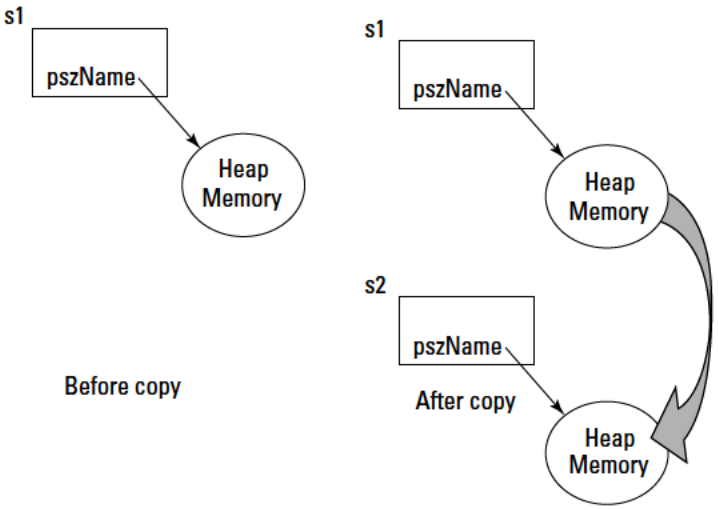


Figure 27-2: A class that allocates resources in the constructor requires a copy constructor that performs a so-called deep copy of the source object.

Allocating a new block of memory and copying the contents of the original into this new block is known as creating a *deep copy* (as opposed to the default shallow copy).

The following DeepStudent program includes a copy constructor that performs a deep copy of the `student` object:

```
//
//  DeepStudent - this program demonstrates how a copy
//                constructor that performs a deep copy
//                can be used to solve copy problems
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;

class Student
{
  protected:
    char*  pszName;
    int    nID;

  public:
    Student(const char* pszNewName, int nNewID)
    {
        cout << "Constructing " << pszNewName << endl;
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
        nID = nNewID;
    }
```

```
    Student(const Student& s)
    {
        cout<<"Constructing copy of "<< s.pszName << endl;

        int nLength = strlen(s.pszName) + 25;
        this->pszName = new char[nLength];
        strcpy(this->pszName, "Copy of ");
        strcat(this->pszName, s.pszName);
        this->nID = s.nID;
    }

   ~Student()
    {
        cout << "Destructing " << pszName << endl;
        delete[] pszName;
        pszName = 0;
    }

    // access functions
    const char* getName()
    {
        return pszName;
    }
    int getID()
    {
        return nID;
    }
};

void someOtherFn(Student s)
{
    // we don't need to do anything here
}

void someFn()
{
    Student student("Adam Laskowski", 1234);
    someOtherFn(student);

    cout << "The student's name is now "
         << student.getName() << endl;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    someFn();

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This program is identical to its ShallowStudent cousin except for the addition of the copy constructor `Student(const Student&)`, but what a difference it makes in the output from the program:

```
Constructing Adam Laskowski
Constructing copy of Adam Laskowski
Destructing Copy of Adam Laskowski
The student's name is now Adam Laskowski
Destructing Adam Laskowski
Press any key to continue . . .
```

The first message is output by the `Student(const char*, int)` constructor that's invoked when the `student` object is created at the beginning of `someFn()`. The second message comes from the copy constructor `Student(const Student&)` that's invoked to create the copy of `student` as part of the call to `SomeOtherFn()`.

This constructor first allocates a new block of heap memory for the `pszName` of the copy. It then copies the string `Copy of` into this field before concatenating the student's name in the next line.

You would normally make a true copy of the name and not tack `Copy of` onto the front; I do so for instructional reasons.

The destructor that's invoked as `s` goes out of scope at the end of `someOtherFn()` is now clearly returning the copy of the name to the heap and not the original string. This is verified back in `someFn()` when the student's name is intact (as you would expect). Finally, the destructor at the end of `someFn()` returns the original string to the heap.

Avoiding Copies

Passing arguments by value is just one of several reasons that C++ invokes a copy constructor to create temporary copies of your object. You may be wondering, "Doesn't all this creating and deleting copies of objects take time?" The obvious answer is, "You bet!" Is there some way to avoid creating copies?

One way is not to pass objects by value but to pass the address of the object. There wouldn't be a problem if `someOtherFn()` were declared as follows:

```
// the following does not cause a copy to be created
void someOtherFn(const Student *pS)
{
    // ...whatever goes here...
}
void someFn()
{
    Student student("Adam Laskowski", 1234);
    someOtherFn(&student);
}
```

This is faster because a single address is smaller than an entire `Student` object, but it also avoids the need to allocate memory off the heap for holding copies of the student's name.

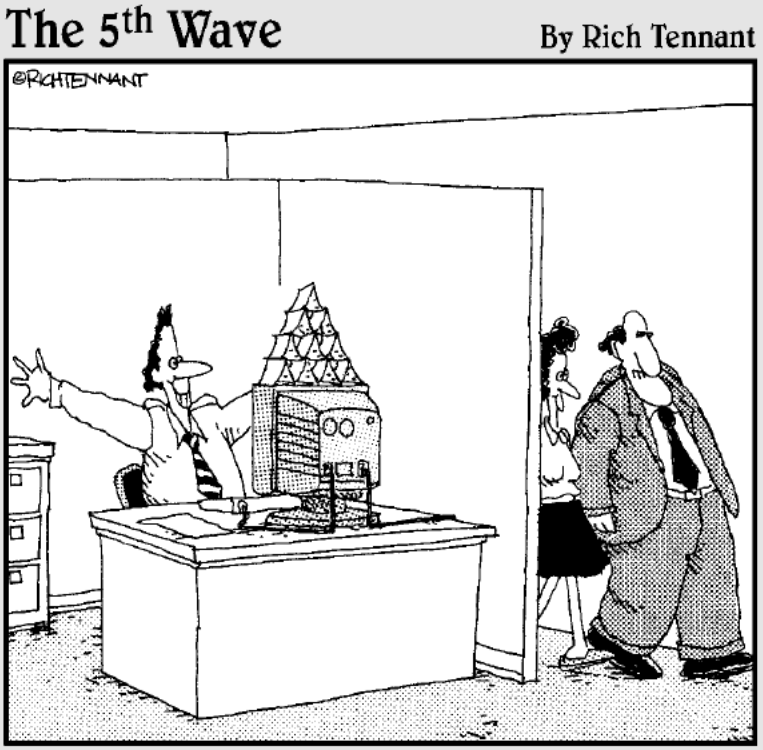
You can get the same effect using reference arguments as in the following:

```
// the following function doesn't create a copy either
void someOtherFn(const Student& s)
{
    // ...whatever you want to do...
}

void someFn()
{
     Student student("Adam Laskowski", 1234);
     someOtherFn(student);
}
```

See Chapter 23 if you don't remember about referential arguments.

Part VI
Advanced Strokes



"Why, of course. I'd be very interested in seeing this new milestone in the project."

In this part . . .

Here you pick up a few loose ends that are nevertheless important to any C++ programmer: You'll learn to overload the assignment operator, you'll learn how to perform file I/O, and you'll discover how to throw error exceptions.

Chapter 28

Inheriting a Class

In This Chapter

- Introducing inheritance
- Implementing inheritance in C++
- Reviewing an example program
- Comparing HAS_A to IS_A

Inheritance occurs all around us every day. I am a human. I inherit certain properties from the class `Human`, such as my ability to converse intelligently (more or less) and my dependence on air, water, and carbohydrate-based nourishment like Twinkies. These latter properties are not unique to humans. The class `Human` inherits these from class `Mammal` (along with something about bearing live young), which inherited them from class `Animal`, and so on.

The capability to pass down properties is a powerful one. It enables you to describe things in an economical way. For example, if my son asks me, "What's a duck?" I might say, "It's a bird that floats and goes quack." Despite your first reaction, that answer actually conveys a significant amount of knowledge. My son knows what a bird is. He knows that birds have wings, that birds can fly (he doesn't know about ostriches yet), and that birds lay eggs. Now, he knows all those same things about a duck *plus* the facts that ducks can float and make a quacking sound. (This might be a good time to refer to Chapter 21 for a discussion about Microwave ovens and their relationship to ovens and kitchen appliances.)

Object-oriented languages express this relationship by allowing one class to inherit from another. Thus, in C++ the class `Duck` might well inherit from `Bird`, and that class might also inherit from `Animal`. Exactly how C++ does this is the topic of this chapter.

Advantages of Inheritance

Inheritance was added to C++ for several reasons. Of course, the major reason is the capability to express the inheritance relationship: that `MicrowaveOven` is an `Oven` is a `KitchenAppliance` thing. More on the IS_A relationship a little later in this and the next chapter.

A minor reason is to reduce the amount of typing and the number of lines of code that you and I have to write. You may have noticed that the commands in C++ may be short, but you need a lot of them to do anything. C++ programs tend to get pretty lengthy, so anything that reduces typing is a good thing.

To see how inheritance can reduce typing, consider the `Duck` example. I don't have to document all the properties about `Duck` that have to do with flying and landing and eating and laying eggs. It inherits all that stuff from `Bird`. I just need to add `Duck`'s quackness property and its ability to float. That's a considerable savings.

A more important and related issue is the major buzzword, *reuse*. Software scientists realized some time ago that starting from scratch with each new project and rebuilding the same software components doesn't make much sense.

Compare the situation in the software industry to that in other industries. How many car manufacturers start from scratch each time they want to design a new car? None. Practitioners in other industries have found it makes more sense to start from screws, bolts, nuts, and even larger existing off-the-shelf components such as motors and transmissions when designing a car.

Unfortunately, except for very small functions like those found in the Standard C++ Library, it's rare to find much reuse of software components. One problem is that it's virtually impossible to find a component from an earlier program that does exactly what you want. Generally, these components require "tweaking." Inheritance allows you to adopt the major functionality of an existing class and tweak the smaller features to adapt an existing class to a new application.

This carries with it another benefit that's more subtle but just as important: *adaptability*. It never fails that as soon as users see your most recent program, they like it but want just one more fix or addition. Consider checking accounts for a moment. How long after I finish the program that handles checking accounts for a bank will it be before the bank comes out with a new "special" checking account that earns interest on the balance?

Not everyone gets this checking account, of course (that would be too easy) — only certain customers get `InterestChecking` accounts. With inheritance, however, I don't need to go through the entire program and recode all the checking account functions. All I need to do is create a new

subclass `InterestChecking` that inherits from `Checking` but has the one additional property of `accumulatesInterest()` and, *voilà*, the feature is implemented. (It isn't quite that easy, of course, but it's not much more difficult than that. I actually show you how to do this in Chapter 29.)

Learning the lingo

You need to get some terms straight before going much further. The class `Dog` inherits properties from class `Mammal`. This is called *inheritance*. We also say that `Dog` is a *subclass* of `Mammal`. Turning that sentence around, we say that `Mammal` is a *base class* of `Dog`. We can also say that `Dog` IS_A `Mammal`. (I use all caps as a way of expressing this unique relationship.) C++ shares this terminology with other object-oriented languages.

The term is adopted from other languages, but you will also find C++ programmers saying things like, "the class Dog *extends* Mammal with its barkiness and tail wagging properties." Well, maybe not in those exact words, but a subclass extends a base class by adding properties.

Notice that although `Dog` IS_A `Mammal`, the reverse is not true. A `Mammal` is not a `Dog`. (A statement like this always refers to the general case. It could be that a particular mammal is, in fact, a dog, but in general a mammal is not a dog.) This is because a `Dog` shares all the properties of other `Mammals`, but a `Mammal` does not have all the properties of a `Dog`. Not all `Mammals` can bark, for example, or wag their tails.

Implementing Inheritance in C++

The following is an outline of how to inherit one class from another:

```
class Student
{
    // ...whatever goes here...
};

class GraduateStudent : public Student
{
    // ...graduate student unique stuff goes here...
};
```

The class `Student` is declared the usual way. The class appears with the name followed by a colon, the keyword `public`, and the name of the base class, `Student`.

The keyword `public` implies that there's probably something called protected inheritance. It's true, there is; but protected inheritance is very uncommon, and I don't discuss it in this book.

Now, I can say that a `GraduateStudent` IS_A `Student`. More to the point, I can use a `GraduateStudent` object anywhere that a `Student` is required, including as arguments to functions. That is, the following is allowed:

```
void fn(Student* pS);
void someOtherFn()
{
    GraduateStudent gs;
    fn(&gs);
}
```

This is allowed because a `gs` object has all the properties of `Student`. Why? Because a `GraduateStudent` IS_A `Student`!

Looking at an example

The following GSInherit program makes this more concrete by creating a `Student` class and a `GraduateStudent` class and invoking functions of each:

```
//
//  GSInherit - demonstrate inheritance by creating
//              a class GraduateStudent that inherits
//              from Student.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;

class Student
{
  protected:
    char*  pszName;
    int    nID;
    double dGrade;          // the student's GPA
    int    nSemesterHours;

  public:
    Student(const char* pszNewName, int nNewID)
    {
        cout << "Constructing student "
```

```
            << pszNewName << endl;
        pszName = new char[strlen(pszNewName) + 1];
        strcpy(pszName, pszNewName);
        nID = nNewID;
        dGrade = 0.0;
        nSemesterHours = 0;
    }
   ~Student()
    {
        cout << "Destructing " << pszName << endl;
        delete[] pszName;
        pszName = 0;
    }

    // access functions
    const char* getName()
    {
        return pszName;
    }
    int getID()
    {
        return nID;
    }
    double getGrade()
    {
        return dGrade;
    }
    int getHours()
    {
        return nSemesterHours;
    }

    // addGrade - add a grade to the GPA and total hours
    double addGrade(double dNewGrade, int nHours)
    {
        double dWtdHrs = dGrade * nSemesterHours;
        dWtdHrs += dNewGrade * nHours;
        nSemesterHours += nHours;
        dGrade = dWtdHrs / nSemesterHours;
        return dGrade;
    }
};

class Advisor
{
  public:
    Advisor() { cout << "Advisor constructed" << endl;}
};

class GraduateStudent : public Student
```

```
{
  protected:
    double dQualifierGrade;
    Advisor advisor;

  public:
    GraduateStudent(const char* pszName, int nID) :
       Student(pszName, nID)
    {
        cout << "Constructing GraduateStudent" << endl;
        dQualifierGrade = 0.0;
    }
};

void someOtherFn(Student* pS)
{
    cout << "Passed student " << pS->getName() << endl;
}

void someFn()
{
    Student student("Lo Lee Undergrad", 1234);
    someOtherFn(&student);

    GraduateStudent gs("Upp R. Class", 5678);
    someOtherFn(&gs);
}

int main(int nNumberofArgs, char* pszArgs[])
{
    someFn();

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This example appears lengthy at first blush. Fortunately, however, the `Student` class is identical to its predecessors in earlier chapters.

The fact that the `Student` class hasn't changed is an important point: You don't have to modify a class in order to inherit from it. I did not have to make any changes to `Student` in order to create the subclass `GraduateStudent`.

The `GraduateStudent` class extends `Student` by adding the data member `dQualifierGrade`. In addition, I provided `GraduateStudent` with a constructor that accepts the student name and ID. Of course, `GraduateStudent` doesn't need to manipulate the student's name and ID on its own — it calls the perfectly serviceable `Student` constructor to do that instead, as the following small excerpt demonstrates:

```
GraduateStudent(const char* pszName, int nID) :
   Student(pszName, nID)
{
    cout << "Constructing GraduateStudent" << endl;
    dQualifierGrade = 0.0;
}
```

The constructor for the base class is invoked before any part of the current class is constructed. Next to be invoked are the constructors for any data members — this accounts for the message from `Advisor`. Control passes into the body of the `GraduateStudent` constructor last.

The output from this program appears as follows:

```
Constructing student Lo Lee Undergrad
Passed student Lo Lee Undergrad
Constructing student Upp R. Class
Advisor constructed
Constructing GraduateStudent
Passed student Upp R. Class
Destructing Upp R. Class
Destructing Lo Lee Undergrad
Press any key to continue . . .
```

You can follow the chain of events by starting with `main()`. The `main()` function does nothing more than call `someFn()`. The `someFn()` function first creates a `Student` object `Lo Lee Undergrad`. The constructor for `Student` generates the first line of output.

`someFn()` then passes the address of "Lo Lee" to `someOtherFn(Student*)`. `someOtherFn()` does nothing more than display the student's name, which accounts for the second line of output.

The `someFn()` function then creates a `GraduateStudent` "Upp R. Class." Returning to the output for a minute, you can see that this invokes the `Student(const char*, int)` constructor first with the name `Upp R. Class`. Once that constructor has completed building the `Student` foundation, the `GraduateStudent` constructor gets a chance to output its message and build on the graduate student floor.

The `someFn()` function then does something rather curious: It passes the address of the `GraduateStudent` object to `someOtherFn(Student*)`. This apparent mismatch of object types is easily explained by the fact that (here it comes) a `GraduateStudent` IS_A `Student` and can be used anywhere a `Student` is required. (Similarly a `GraduateStudent*` can be used in place of a `Student*`.)

The remainder of the output is generated when both `student` and `gs` go out of scope at the return from `someFn()`. The objects are destructed in the

reverse order of their construction, so `gs` goes first and then `student`. In addition, the destructor for `GraduateStudent` is called before the destructor for `Student()`.

The destructor for the subclass should destruct only those fields that are unique to the subclass. Leave the destructing of the base class data members to the subclass's destructor.

Having a HAS_A Relationship

Notice that the class `GraduateStudent` includes the members of class `Student` and `Advisor` but in a different way. By defining a data member of class `Advisor`, a `GraduateStudent` contains all the members of `Advisor` within it. However, you can't say that a `GraduateStudent` IS_AN `Advisor`. Rather, a `GraduateStudent` HAS_AN `Advisor`.

The analogy is like a car with a motor. Logically, you can say that car is a subclass of vehicle, so it inherits the properties of all vehicles. At the same time, a car has a motor. If you buy a car, you can logically assume that you are buying a motor as well (unless you go to the used car lot where I got my last junk heap).

If some friends ask you to show up at a rally on Saturday with your vehicle of choice, and you arrive in your car, they can't complain and kick you out. But if you were to appear on foot carrying a motor, your friends would have reason to laugh you off the premises, because a motor is not a vehicle.

These assertions appear as follows when written in C++:

```
class Vehicle {};
class Motor {};
class Car : public Vehicle
{
  public:
    Motor motor;
};

void vehicleFn(Vehicle* pV);
void motorFn(Motor* pM);

void someFn()
{
    Car c;

    vehicleFn(&c);     // this is allowed
    motorFn(&c.motor); // so is this

    motorFn(&c);       // this is not allowed
}
```

Chapter 29

Are Virtual Functions for Real?

In This Chapter

- Overriding between functions that are members of a class
- Introducing virtual member functions
- Some special considerations for virtual functions
- Declaring your destructor virtual — when to and when not to do it

Inheritance gives users the ability to describe one class in terms of another. Just as important, it highlights the relationship between classes. I describe a duck as "a bird that . . .", and that description points out the relationship between duck and bird. From a C++ standpoint, however, a piece of the puzzle is still missing.

You have probably noticed this, but a microwave oven looks nothing like a conventional oven and nor does it work the same internally. Nevertheless, when I say "cook," I don't want to worry about the details of how each oven works internally. This chapter describes this problem in C++ terms and then goes on to describe the solution as well.

Overriding Member Functions

It has always been possible to overload a member function with another member function in the same class as long as the arguments differ:

```
class Student
{
  public:
    double grade();     // return the student's gpa
    double grade(double); // set the student's gpa

    // ...other stuff...
};
```

You see this in spades in Chapters 26 and 27 where I overload the constructor with a number of different types of constructors. It is also possible to overload a function in one class with a function in another class even if the arguments are the same, because the class is not the same:

```
class Student
{
  public:
    double grade(double); // set the student's gpa
};

class Hill
{
  public:
    double grade(double); // set the slope of the hill
};
```

Inheritance offers another way to confuse things: A member function in a subclass can overload a member function in the base class.

Overloading a base class member function is called *overriding*.

Early binding

Overriding is fairly straightforward. Consider, for example, the following EarlyBinding demonstration program:

```
//
//  EarlyBinding - demonstrates early binding in
//                 overriding one member function with
//                 another in a subclass.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  public:
    double calcTuition() { return 0.0; }
};

class GraduateStudent : public Student
{
  public:
    double calcTuition() { return 1.0; }
```

```
};

int main(int nNumberofArgs, char* pszArgs[])
{
    // the following calls Student::calcTuition()
    Student s;
    cout << "The value of s.calcTuition() is "
         << s.calcTuition()
         << endl;

    // the following calls GraduateStudent::calcTuition()
    GraduateStudent gs;
    cout << "The value of gs.calcTuition() is "
         << gs.calcTuition()
         << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

Here both the `Student` and `GraduateStudent` classes include a `calcTuition()` member function (and nothing else, just to keep the listings short). Presumably, the university calculates tuition for graduate and undergraduate students differently, but for this demonstration, determining which function is being called is the only important thing. Therefore, `Student::calcTuition()` returns a `0`, while `GraduateStudent::calcTuition()` returns a `1` — can't get much simpler than that!

The `main()` function first creates a `Student` object `s` and then invokes `s.calcTuition()`. Not surprisingly, this call is passed to `Student::calcTuition()` as is clear from the output of the program as quoted here. The `main()` function then does the same for `GraduateStudent` with predictable results:

```
The value of s.calcTuition() is 0
The value of gs.calcTuition() is 1
Press any key to continue . . .
```

In this program, the C++ compiler is able to decide at compile time which member function to call based upon the declared type of `s` and `gs`.

Resolving calls to overridden member functions based on the declared type of the object is called *compile-time* or *early binding*.

This simple example is not too surprising so far, but let me put a wrinkle in this simple fabric.

Ambiguous case

The following AmbiguousBinding program is virtually identical to the earlier EarlyBinding program. The only difference is that instead of invoking `calcTuition()` directly, this version of the program calls the function through a pointer passed to a function:

```
//
//  AmbiguousBindng - demonstrates a case where it's not
//                    clear what should happen. In this
//                    case, C++ goes with early binding
//                    while languages like Java and C#
//                    use late binding.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

class Student
{
  public:
    double calcTuition() { return 0.0; }
};

class GraduateStudent : public Student
{
  public:
    double calcTuition() { return 1.0; }
};

double someFn(Student* pS)
{
    return pS->calcTuition();
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // the following calls Student::calcTuition()
    Student s;
    cout << "The value of someFn(&s) is "
         << someFn(&s)
         << endl;

    // the following calls GraduateStudent::calcTuition()
```

```
    GraduateStudent gs;
    cout << "The value of someFn(&gs) is "
         << someFn(&gs)
         << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

Just as in the EarlyBinding example, this program starts by creating a `Student` object `s`. Rather than invoke `s.calcTuition()` directly, however, this version passes the address of the `s` to `someFn()` and that function does the honors. The program repeats the process with a `GraduateStudent` object `gs`.

Now without looking ahead, I have a question: Which `calcTuition()` will `pS->calcTuition()` call when `main()` passes the address of a `GraduateStudent` to `someFn()`? You could argue that it will call `Student::calcTuition()` because the declared type of `pS` is `Student*`. On the other hand, you could argue that the same call will invoke `GraduateStudent::calcTuition()` because the "real type" is `GraduateStudent*`.

The "real type" of an object is known as the *run-time type* (as opposed to the *declared type*). These are also known as *dynamic type* and *static type*, respectively.

The output from this program appears as follows:

```
The value of someFn(&s) is 0
The value of someFn(&gs) is 0
Press any key to continue . . .
```

You can see that, by default, C++ bases its decision on the declared type of the object. Therefore, `someFn()` calls `Student::calcTuition()` because that's the way the object is declared irrespective of the run-time type of the object provided in the call.

The alternative to early binding is to decide which member function to call based on the run-time type of the object. This is known as *late binding*.

Thus, we say that C++ prefers early binding.

Enter late binding

Early binding does not capture the essence of object-oriented programming. Let's return to how I made nachos in Chapter 21. In a sense, I acted as the late binder. The recipe said, "Heat the nachos in the oven." It didn't say, "If the type of oven is microwave, do this; if the type is convection oven, do this; if the type is conventional oven, do this; if using a campfire, do this." The recipe (the code) relied on me (the late binder) to decide what the action (member function) `heat` means when applied to the `oven` (the particular instance of class `Oven`) or any of its variations (subclasses), such as a `microwave` (`MicrowaveOven`). People think this way, and designing a language along these lines enables the software model to more accurately describe a real-world solution a person might think up.

There are also mundane reasons of maintainability and reusability to justify late binding. Suppose I write a great program around the class `Student`. This program, cool as it is, does lots of things, and one of the things it does is calculate the student's tuition for the upcoming year. After months of design, coding, and testing, I release the program to great acclaim and accolades from my peers.

Time passes and my boss asks me to change the rules for calculating the tuition on graduate students. I'm to leave the rules for students untouched, but I'm to give graduate students some type of break on their tuition so that the university can attract more and better postgraduate candidates. Deep within the program, `someFunction()` calls the `calcTuition()` member function as follows:

```
void someFunction(Student* pS)
{
    pS->calcTuition();

    // ...function continues on...
}
```

This should look familiar. If not, refer to the beginning of this chapter.

If C++ did not support late binding, I would need to edit `someFunction()` to do something similar to the following:

```
void someFunction(Student* pS)
{
    if (pS->type() == STUDENT)
    {
        pS->Student::calcTuition();
    }
```

```
    if (pS->type() == GRADUATESTUDENT)
    {
        pS->GraduateStudent::calcTuition();
    }

    // ...function continues on...
}
```

Using the extended name of the function, including the class name, forces the compiler to use the specific version of `calcTuition()`.

I would add a `member type()` to the class that would return some constant. I could establish the value of this constant in the constructor.

This change doesn't seem so bad until you consider that `calcTuition()` isn't called in just one place; it's called throughout the program. The chances are not good that I will find all the places that it's called.

And even if I do find them all, I'm editing (read "breaking") previously debugged, tested, checked in, and certified code. Edits can be time-consuming and boring, and they introduce opportunities for error. Any one of my edits could be wrong. At the very least, I will have to retest and recertify every path involving `calcTuition()`.

What happens when my boss wants another change? (My boss, like all bosses, is like that.) I get to repeat the entire process.

What I really want is for C++ to keep track of the real-time type of the object and to perform the call using late binding.

The ability to perform late binding is called *polymorphism* ("poly" meaning "varied" and "morph" meaning "form"). Thus, a single object may take varied actions based upon its run-time type.

All I need to do is add the keyword `virtual` to the declaration of the member function in the base class as demonstrated in the following LateBinding example program:

```
//
//  LateBinding - addition of the keyword 'virtual'
//                changes C++ from early binding to late
//                binding.
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
```

```
using namespace std;

class Student
{
  public:
    virtual double calcTuition() { return 0.0; }
};

class GraduateStudent : public Student
{
  public:
    virtual double calcTuition() { return 1.0; }
};

double someFn(Student* pS)
{
    return pS->calcTuition();
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // the following calls Student::calcTuition()
    Student s;
    cout << "The value of someFn(&s) is "
         << someFn(&s)
         << endl;

    // the following calls GraduateStudent::calcTuition()
    GraduateStudent gs;
    cout << "The value of someFn(&gs) is "
         << someFn(&gs)
         << endl;

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

It's not necessary to add the `virtual` keyword to the subclass as well, but doing so is common practice. A member function that is bound late is known as a *virtual member function*.

Other than the `virtual` keyword, there is no other difference between the LateBinding program and its AmbiguousBinding predecessor, but the results are strikingly different:

```
The value of someFn(&s) is 0
The value of someFn(&gs) is 1
Press any key to continue . . .
```

This is exactly what I want: C++ is now deciding which version of `calcTuition()` to call, not based upon its declared type but based upon its run-time type.

It may seem surprising that the default for C++ is early binding, but the reason is simple. Late binding adds a small amount of overhead to every call to virtual member functions. The inventors of C++ did not want to give critics any reasons to reject the language, so, by default, C++ does not include the overhead of late binding with functions that are not virtual.

When Is Virtual Not?

Just because you think a particular function call is bound late doesn't mean that it is. C++ generates no indication at compile time which calls it thinks are bound early and which are bound late.

The most critical thing to watch for is that all of the member functions in question are declared identically, including the return type. If they aren't declared with the same arguments in the subclass, the member functions aren't overridden; without overriding, there can't be late binding. Consider the following code snippet:

```
class Base
{
  public:
    virtual void fn(int x);
};

class Subclass : public Base
{
  public:
    virtual void fn(double x);
};
void test(Base* pB)
{
    pB->fn(1);

    pB->fn(2.0);
};
```

The function `fn()` is not bound late because the arguments don't match. Not surprisingly, the first call to `fn()` within `test()` goes to `Base::fn(int)` even if `test()` is passed to an object of class `Subclass`. Somewhat surprisingly, the second call goes to `Base::fn(int)` as well after converting the `double` to an `int`. Again, no overriding, no late binding.

The only exception to this rule is best explained by the following example:

```
class Base
{
  public:
    virtual Base* fn();
};

class Subclass : public Base
{
  public:
    virtual Subclass* fn();
};
```

Here, the function `fn()` is bound late, even though the return type doesn't match exactly. In practice, this is quite natural. If a function is dealing with `Subclass` objects, it seems natural that it should return a `Subclass` object as well.

Virtual Considerations

Specifying the class name in the call forces the call to find out early whether the function is declared virtual or not. For example, the following call is to `Base::fn()` because that's what the programmer indicated that she intended:

```
void test(Base* pB)
{
    pB->Base::fn();  // this call is not bound late
}
```

Constructors cannot be declared virtual because there is no completed object at the time the constructor is invoked to use as the basis for late binding.

On the other hand, destructors should almost always be declared virtual. If not, you run the risk of not completely destructing the object, as demonstrated in the following snippet:

```
class Base
{
  public:
    ~Base() {}  // this should be declared virtual
};

class Subclass
```

```
{
  protected:
    MyObject* pMO;

  public:
    Subclass()
    {
        pMO = new MyObject;
    }
   ~Subclass()
    {
        delete pMO;
        pMO = 0;
    }
};

Base* someOtherFn()
{
    return new Subclass;
}

void someFn()
{
    Base* pB = someOtherFn();
    delete pB;
}
```

The program has a subtle but devastating bug. When `someFn()` is called, it immediately calls `someOtherFn()`, which creates an object of class `Subclass`. The constructor for `Subclass` allocates an object of class `MyObject` off the heap. Ostensibly, all is well because the destructor for `Subclass` returns `MyObject` to the heap when the `Subclass` object is destructed.

However, when `someFn()` calls `delete`, it passes a pointer of type `Base*`. If this call is allowed to bind early, it will invoke the destructor for `Base`, which knows nothing about `MyObject`. The memory will not be returned to the heap.

I realize that technically `delete` is a keyword and not a function call, but the semantics are the same.

Declaring the destructor for `Base` virtual solves the problem. Now the call to delete is bound late — realizing that the pointer passed to delete actually points to a `Subclass` object, `delete` invokes the `Subclass` destructor, and the memory is returned, as it's supposed to be.

So is there a case in which you don't want to declare the destructor virtual? Only one. Earlier I said that virtual functions introduce a "little" overhead. Let me be more specific. One thing they add is an additional hidden pointer to every object — not one pointer per virtual function, just one pointer, period. A class with no virtual functions does not have this pointer.

Now, one pointer doesn't sound like much, and it isn't, unless the following two conditions are true:

- The class doesn't have many data members (so that one pointer is a lot compared with what's there already).
- You create a lot of objects of this class (otherwise, the overhead doesn't matter).

If either of these two conditions is not true, always declare your destructors virtual.

Chapter 30

Overloading Assignment Operators

In This Chapter

- Overloading operators — in general, a bad idea
- Overloading the assignment operator — why that one is critical
- What to do when you just can't be bothered with writing an assignment operator

The little symbols like +, –, =, and so on are called *operators*. These operators are already defined for the intrinsic types like `int` and `double`. However, C++ allows you to define the existing operators for classes that you create. This is called *operator overloading*.

Operator overloading sounds like a great idea. The examples that are commonly named are classes like `Complex` that represent a complex number. (Don't worry if you don't know what a complex number is. Just know that C++ doesn't handle them intrinsically.) Having defined the class `Complex`, you can then define the addition, multiplication, subtraction, and division operators (all of these operations are defined for complex numbers). Then you write cool stuff like:

```
Complex c1(1, 0), c2(0, 1);
Complex c3 = c1 + c2;
```

Overloading operators turns out to be much more difficult in practice than in theory. So much so that I consider operator overloading beyond the scope of this book with two exceptions, one of which is the subject of this chapter: overloading the assignment operator. The second operator worth overloading is the subject of the next chapter.

Overloading an Operator

C++ considers an operator as a special case of a function call. It considers the + operator to be shorthand for the function `operator+()`. In fact, for

any operator %, the function version is known as operator%(). So to define what addition means when applied to a Complex object, for example, you need merely to define the following function:

```
Complex& operator+(Complex& c1, Complex& c2);
```

You can define what existing operators mean when applied to objects of your making, but there are a lot of things you *can't* do when overloading operators. Here are just a few:

- You can't define a new operator, only redefine what an existing operator means when applied to your user-defined class.
- You can't overload the intrinsic operators like operator+(int, int).
- You can't affect the precedence of the operators.

In addition, the assignment operator must be a member function — it cannot be a non-member function like the addition operator just defined.

Overloading the Assignment Operator Is Critical

The C++ language does provide a default assignment operator. That's why you can write things like the following:

```
Student s1("Stephen Davis", 1234);
Student s2;
s2 = s1;      // use the default assignment operator
```

The default assignment operator does a member-by-member copy of each data member from the object on the right into the object on the left. This is completely analogous to the default copy constructor. Remember that this member-by-member copy is called a *shallow copy*. (Refer to Chapter 27 for more on copy constructors and shallow copies.)

The problems inherent in the default assignment operator are similar to the copy constructor, only worse. Consider the following example snippet:

```
class Student
{
  protected:
    char*  pszName;
    int    nID;

  public:
```

```
    Student(const char* pszNewName, int nNewID)
    {
        cout << "Constructing " << pszNewName << endl;
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
        nID = nNewID;
    }
   ~Student()
    {
        cout << "Destructing " << pszName << endl;
        delete[] pszName;
        pszName = 0;
    }

    // ...other members...
};

void someFn()
{
    Student s1("Stephen Davis", 1234);
    Student s2("Cayden Amrich", 5678);

    s2 = s1;   // this is legal but very bad
}
```

The function `someFn()` first creates an object `s1`. The `Student(const char*, int)` constructor for `Student` allocates memory from the heap to use to store the student's name. The process is repeated for `s2`.

The function then assigns `s1` to `s2`. This does two things, both of which are bad:

- Copies the `s1.pszName` pointer into `s2.pszName` so that both objects now point to the same block of heap memory.
- Wipes out the previous value of `s2.pszName` so that the block of memory used to store the student name `Cayden Amrich` is lost.

Here's what the assignment operator for `Student` needs to do:

- Delete the memory block pointed at by `s2.pszName` (like a destructor).
- Perform a deep copy of the string from `s1.pszName` into a newly allocated array in `s2.pszName` (like a copy constructor). (Again, see Chapter 27 for a description of deep copying.)

In fact, you can make this general statement: An assignment operator acts like a destructor to wipe out the current values in the object followed by a copy constructor that copies new values into the object.

Looking at an Example

The following StudentAssignment program contains a `Student` class that has a constructor and a destructor along with a copy constructor and an assignment operator — everything a self-respecting class needs!

```
//
//  StudentAssignment - this program demonstrates how to
//                create an assignment operator that
//                performs the same deep copy as the copy
//                constructor
//
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
using namespace std;

class Student
{
  protected:
    char*  pszName;
    int    nID;

    void init(const char* pszNewName, int nNewID)
    {
        int nLength = strlen(pszNewName) + 1;
        pszName = new char[nLength];
        strcpy(pszName, pszNewName);
        nID = nNewID;
    }

    void destruct()
    {
        delete[] pszName;
        pszName = 0;
    }

  public:
    Student(const char* pszNewName, int nNewID)
    {
        cout << "Constructing " << pszNewName << endl;
        init(pszNewName, nNewID);
```

```
    }
    Student(Student& s)
    {
        cout<<"Constructing copy of "<< s.pszName << endl;
        init(s.pszName, s.nID);
    }

    virtual ~Student()
    {
        cout << "Destructing " << pszName << endl;
        destruct();
    }

    // overload the assignment operator
    Student& operator=(Student& source)
    {
        // don't do anything if we are assigned to
        // ourselves
        if (this != &source)
        {
            cout << "Assigning " << source.pszName
                 << " to "       << pszName << endl;

            // first destruct the existing object
            destruct();

            // now copy the source object
            init(source.pszName, source.nID);
        }

        return *this;
    }


    // access functions
    const char* getName()
    {
        return pszName;
    }
    int getID()
    {
        return nID;
    }
};

void someFn()
{
    Student s1("Adam Laskowski", 1234);
    Student s2("Vanessa Barbossa", 5678);
```

```
    s2 = s1;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    someFn();

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The data members of this `Student` class are the same as the versions from earlier chapters. The constructor and copy constructor are the same as well, except that the actual work is performed in an `init()` function that is invoked from both constructors. The assignment operator can reuse the same `init()` function as well to perform its construction function.

The code that implements the destruct sequence has also been transferred from `~Student()` to a protected `destruct()` member function.

Following the destructor is the assignment operator `operator=()`. This function first tests to see if the address of the object passed is the same as the current object. This is to detect the following case:

```
s1 = s1;
```

In this case, the assignment operator does nothing. If the source and current objects are not the same, the function first destructs the current object and then copies the contents of the source object into the current object. Finally, it returns a reference to the current object.

The `someFn()` function shows how this works in practice. After first declaring two `Student` objects `s1` and `s2`, `someFn()` executes the assignment

```
s2 = s1;
```

which is interpreted as if it had been written as

```
s2.operator=(s1);
```

That is, the assignment operator destructs `s2` and then copies the contents of `s1` into `s2`.

The destructor invoked at the end of `someFn()` demonstrates that the two objects, `s1` and `s2`, don't both refer to the same piece of heap memory. The output from the program appears as follows:

```
Constructing Adam Laskowski
Constructing Vanessa Barbossa
Assigning Adam Laskowski to Vanessa Barbossa
Destructing Adam Laskowski
Destructing Adam Laskowski
Press any key to continue . . .
```

The reason that the assignment operator returns a reference to the current object is to allow the following:

```
s3 = s1 = s2;
```

Writing Your Own (or Not)

I don't expect you to learn all the ins and outs of overloading operators; however, you can't go too wrong if you follow the pattern set out by the Student example:

1. Check to make sure that the left-hand and right-hand objects aren't the same — if so, return without taking any action.
2. Destruct the left-hand object (the current object which is the same object referred to by `this`).
3. Copy construct the left-hand object using the right-hand object as the source.
4. Return a reference to the left-hand object (that is, `return *this;`).

If this is too much, you can always do the following:

```
class Student
{
  protected:
    Student& operator=(Student&)
    {
        return *this;
    }

    // ...whatever else...
};
```

This assignment operator doesn't do anything, but by being declared protected, it precludes any application software from trying to use the default assignment operator. Now

```
s1 = s2;
```

will generate a compiler error.

Chapter 31

Performing Streaming I/O

In This Chapter

- Using stream I/O — an overview
- Opening an object for file input and output
- Detecting errors when performing file I/O
- Formatting output to a file
- Using the stream classes on internal buffers for easy string formatting

I gave you a template to follow when generating new programs in Chapter 2. Since you were just starting the journey to C++, I asked you to take a lot of what was in that template on faith; then throughout subsequent chapters, I explained each of the features of the template. There's just one item remaining: stream input/output (commonly shortened to just I/O).

I must warn you that stream I/O can't be covered completely in a single chapter — entire books are devoted to this one topic. Fortunately, however, you don't need to know too much about stream I/O in order to write the vast majority of programs.

How Stream I/O Works

Stream I/O is based on overloaded versions of `operator>>()` and `operator<<()` (known as the right-shift and left-shift operators, respectively)

Note: I don't cover the << (left-shift) and >> (right-shift) operators in my discussion of arithmetic operators in Chapter 4, as these perform bit operations that are beyond the scope of a beginning programmer.

The prototype declarations for the stream operators are found in the include file `iostream`. The code for these functions is part of the Standard C++ Library that your programs link with by default. That's why the standard

template starts out with `#include <iostream>` — without it, you can't perform stream I/O. The following excerpt shows just a few of the prototype declarations that appear in `iostream`:

```
//for input we have:
istream& operator>>(istream& source, int    &dest);
istream& operator>>(istream& source, double &dest);
istream& operator>>(istream& source, char   *pDest);
//...and so forth...

//for output we have:
ostream& operator<<(ostream& dest, const char *pSource);
ostream& operator<<(ostream& dest, int         source);
ostream& operator<<(ostream& dest, double      source);
//...and so it goes...
```

When overloaded to perform stream input, `operator<<()` is called the *extractor*. The class `istream` is the basic class for performing input from a file. C++ creates an `istream` object `cin` and associates it with the keyboard when your program first starts and before `main()` is executed.

The first prototype in the earlier extract from the `iostream` include file refers to the function that is invoked when you enter the following C++ code:

```
int i;
cin >> i;
```

As you've seen, extracting from `cin` is the standard way of performing keyboard input.

When overloaded to perform stream output, `operator<<()` is called the *inserter*. C++ uses the `ostream` class for performing formatted output from a file. C++ creates an `ostream` object `cout` at program start and associates it with the console display.

The first prototype among the output functions is called when you enter the following:

```
cout << "C++ programming is fn()";
```

Inserting to `cout` is the standard means for displaying stuff to the operator.

Both `cin` and `cout` are declared in the `iostream` include file. That's how your program knows what they are.

C++ opens a second `ostream` object at program startup. This object, `cerr`, is also associated with the display by default, but it is used as a standard error output. If you've used Linux, Unix, or the Windows console window much, you know that you can redirect standard input and output. For example, the command

```
myprogram <file1.txt >file2.txt
```

says, "Execute `myprogram.exe`, but read from `file1.txt` rather than the keyboard, and output to `file2.txt` rather than the display." That is, `cin` is associated with `file1.txt` and `cout` with `file2.txt`. In this case, if you send error messages to `cout`, the operator will never see them because they will be sent to the file. However, messages sent to `cerr` will continue to go to the display since it is not redirected with `cout`.

Always send error messages to `cerr` rather than `cout` just in case `cout` has been redirected.

Stream Input/Output

C++ provides separate classes for performing input and output to files. These classes, `ifstream` and `ofstream`, are defined in the include file `fstream`.

Collectively both `ifstream` and `ofstream` are known as `fstream` classes.

Creating an input object

The class `ifstream` provides a constructor used to open a file for input:

```
ifstream(const char* pszFileName,
                  ios_base::openmode mode);
```

This constructor opens a file, creates an object of class `ifstream`, and associates that object with the opened file to be used for input. The first argument to the constructor is a pointer to the name of the file to open. You can provide a full pathname or just the filename.

If you provide the filename without a path, C++ will look in the current directory for the file to read. When executing from your program from within Code::Blocks, the current directory is the directory that contains the project file.

Don't forget that a Windows/DOS backslash is written `"\\"` in C++. Refer to Chapter 5 for details.

The second argument directs some details about how the file is to be opened when the object is created. The type `openmode` is a user-defined type within the class `ios_base`. The legal values of `mode` are defined in Table 31-1. If `mode` is not provided, the default value is `ios_base::in`, which means open the file for input. (Pretty logical for a file called `ifstream`.)

The following example code snippet opens the text file `MyData.txt` and reads a few integers from it:

```
void someFn()
{
    // open the file MyData.txt in the current directory
    ifstream input("MyData.txt");

    int a, b, c;

    input >> a >> b >> c;
    cout <<   "a = " << a
         << ", b = " << b
         << ", c = " << c << endl;
}
```

To specify the full path, I could write something like the following:

```
ifstream input("C:\\\\MyFolder\\MyData.txt");
```

This opens the "C:\\MyFolder\MyData.txt" file.

The destructor for class `ifstream` closes the file. In the preceding snippet, the file "MyData.txt" is closed when control exits `someFn()` and the `input` object goes out of scope.

Table 31-1 Constants That Control How Files Are Opened for Input

Flag	*Meaning*
`ios_base::binary`	Open file in binary mode (alternative is text mode)
`ios_base::in`	Open file for input (implied for `istream`)

Creating an output object

The class `ofstream` is the output counterpart to `ifstream`. The constructor for this class opens a file for output using the inserter operator:

```
ofstream(const char* pszFileName,
          ios_base::openmode mode);
```

This constructor opens a file for output. Here again, `pszFileName` points to the name of the file, whereas mode controls some aspects about how the file is to be opened. Table 31-2 lists the possible values for mode. If you don't provide a mode, the default value is `out + trunc`, which means "open the file for output and truncate whatever is already in the file" (the alternative is to append whatever you output to the end of the existing file).

The following example code snippet opens the text file `MyData.txt` and writes some absolutely true information into it:

```
void someFn()
{
    // open the file MyData.txt in the current directory
    ofstream output("MyData.txt");

    output <<   "Stephen is suave and handsome\n"
           << "and definitely not balding prematurely"
           << endl;
}
```

The destructor for class `ofstream` flushes any buffers to disk and closes the file before destructing the object and returning any local memory buffers to the heap when the `output` object goes out of scope at the end of `someFn()`.

Table 31-2 Constants That Control How Files Are Opened for Output

Flag	*Meaning*
`ios_base::app`	Seek to End of File before each write
`ios_base::ate`	Seek to End of File immediately after opening the file
`ios_base::binary`	Open file in binary mode (alternative is text mode)
`ios_base::out`	Open file for output (implied for `ostream`)
`ios_base::trunc`	Truncate file, if it exists (default for `ostream`)

Open modes

Tables 31-1 and 31-2 show the different modes that are possible when opening a file. To set these values properly, you need to answer the following three questions:

- Do you want to read from the file or write to the file? Use `ifstream` to read and `ofstream` to write. If you intend to both read and write to the same file, then use the class `fstream` and set the mode to `in | out`, which opens the file for both input and output. Good luck, however, because getting this to work properly is difficult. It's much better to write to a file with one object and read from the file with another object.
- If you are writing to the file and it already exists, do you want to add to the existing contents (in which case, open with mode set to `out | ate`) or delete the contents and start over (in which case, open with mode set to `out | trunc`)?
- Are you reading or writing text or binary data? Both `ifstream` and `ofstream` default to text mode. Use `binary` mode if you are reading or writing raw, nontext data. (See the next section in this chapter for a short explanation of binary mode.)

The | is the "binary OR" operator. The result of `in | out` is an `int` with the `in` bit set and the `out` bit set. You can OR any of the mode flags together.

If the file does not exist when you create the `ofstream` object, C++ will create an empty output file.

What is binary mode?

You can open a file for input or output in either binary or text mode. The primary difference between binary and text mode lies in the way that newlines are handled. The Unix operating system was written in the days when typewriters were still fashionable (when it was called "typing" instead of "keyboarding" or the soon to become fashionable "iPhoning"). Unix ends sentences with a carriage return followed by a line feed.

Subsequent operating systems saw no reason to continue using two characters to end a sentence, but they couldn't agree on which character to use. Some used the carriage return and others the line feed, now renamed newline. The C++ standard is the single newline.

When a file is opened in text mode, the C++ library converts the single newline character into what is appropriate for your operating system on output, whether it's a carriage return plus line feed, a single carriage return, or a line feed (or something else entirely). C++ performs the opposite conversion when reading a file. The C++ library does no such conversions for a file opened in binary mode.

Always use binary mode when manipulating a file that's not in human-readable format. If you don't, the C++ library will modify any byte in the data stream that happens to be the same as a carriage return or linefeed.

Hey, file, what state are you in?

A properly constructed `ifstream` or `ofstream` object becomes a stand-in for the file that it's associated with.

The programmer tends to think of operations on the `fstream` objects as being the same as operations on the file itself. However, this is only true so long as the object is properly constructed. If an `fstream` object fails to construct properly, it might not be associated with a file — for example, if an `ifstream` object is created for a file that doesn't exist. In this case, C++ rejects stream operations without taking any action at all.

Fortunately, C++ tells you when something is wrong — the member function `bad()` returns a `true` if something is wrong with the `fstream` object and if it cannot be used for input or output. This usually happens when the object cannot be constructed for input because the file doesn't exist or for output because the program doesn't have permission to write to the disk or directory. Other system errors can also cause the `bad()` state to become true.

The term "bad" is descriptive, if a bit excessive (I don't like to think of computer programs as being bad or good). A lesser state called `fail()` is set to `true` if the last read or write operation failed. For example, if you try to read an `int` and the stream operator can find only characters, then C++ will set the `fail()` flag. You can call the member function `clear()` to clear the fail flag and try again — the next call may or may not work. You cannot clear the `bad()` flag — just like wine, an object gone bad is not recoverable.

Attempts to perform input from or output to an object with either the `bad()` or `fail()` flag set are ignored.

I mean this literally — no input or output is possible as long as the internal state of the `fstream` object has an error. The program won't even try to perform I/O, which isn't so bad on output — it's pretty obvious when your program isn't performing output the way it's supposed to. This situation can lead to some tricky bugs in programs that perform input, however. It's very easy to mistake garbage left in the variable, perhaps from a previous read, for valid input from the file.

Consider the following ReadIntegers program, which contains an `unsafeFn()` that reads values from an input file:

```
//
//  ReadIntegers - this program reads integers from
//                 an input file MyFile.txt contained
//                 in the current directory.
//
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
using namespace std;

void unsafeFn()
{
    ifstream myFile("MyFile.txt");
    int nInputValue;

    for(int n = 1; n <= 10; n++)
    {
        // read a value
        myFile >> nInputValue;

        // value successfully read - output it
        cout << n << " - " << nInputValue << endl;
    }
}

int main(int nNumberofArgs, char* pszArgs[])
{
    unsafeFn();

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The preceding `unsafeFn()` function reads ten values from `MyFile.txt` and displays them on the console. That sounds okay, but what if there aren't ten values in `MyFile.txt` — what if there are only nine (or five or none!)? This version of the program generated the following output when provided a sample `MyFile.txt`:

```
1 - 1
2 - 2
3 - 3
4 - 4
5 - 5
6 - 6
7 - 7
8 - 7
```

```
9 - 7
10 - 7
Press any key to continue . . .
```

The question is, did the file really contain the value 7 four times, or did an error occur after the seventh read? There is no way for the user to tell because once the program gets to the End of File, all subsequent read requests fail. The value of `nInputValue` is not set to zero or some other "special value." It retains whatever value it had on the last successful read request, which in this case is 7.

The most flexible means to avoid this problem is to exit the loop as soon as an error occurs using the member function `fail()`, as demonstrated by the following `safeFn()` version of the same function (also part of the ReadIntegers program on the enclosed CD-ROM):

```
//
//  ReadIntegers - this program reads integers from
//                 an input file MyFile.txt contained
//                 in the current directory.
//
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
using namespace std;

void safeFn()
{
    ifstream myFile("MyFile.txt");
    int nInputValue;

    for(int n = 0; n < 10; n++)
    {
        // read a value
        myFile >> nInputValue;

        // exit the loop on read error
        if (myFile.fail())
        {
            break;
        }

        // value successfully read - output it
        cout << n << " - " << nInputValue << endl;
    }
}

int main(int nNumberofArgs, char* pszArgs[])
```

```
{
    safeFn();

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

This version generated the following output when reading the same `MyFile.txt` file:

```
1 - 1
2 - 2
3 - 3
4 - 4
5 - 5
6 - 6
7 - 7
Press any key to continue . . .
```

Now it's obvious that there are only seven values in the file rather than the expected ten and that the number seven isn't repeated.

Always check the value of `fail()` after extracting data from an input file to make sure that you've actually read a new value.

Notice that the preceding ReadIntegers program adds the line `#include <fstream>` to the standard template I've used for all programs in earlier chapters. This extra include file is necessary to gain access to the `ifstream` and `ofstream` classes.

Don't overflow that buffer

If you look closely at some of the earlier programs in this book, you'll see C++ statements like the following:

```
char szStudentName[80];
cin >> szStudentName;
```

This snippet allocates 80 characters for the student's name (surely that's enough for anyone's name) and then extracts a string into that array. The problem is that the extractor doesn't know how large the array is — if the user types more than 80 characters before entering a return or whitespace, then the C++ library function will overflow the end of the array and overwrite memory. Hackers use this overflow capability in programs that interface directly to the Internet to overwrite the machine instructions in the program, thereby taking over control of your computer.

You can avoid this problem two ways. One way is to use the member function `getline()`. This function allows you to specify the length of the array as in the following:

```
char szStudentName[80];
cin.getline(szStudentName, 80);
```

This call reads input until the first newline or until 80 characters have been read, whichever comes first. Any characters not read are left in `cin` for the next read to pick up.

A second approach is to use the `string` class. This class acts like a char array except that it dynamically resizes to fit the amount of data. Thus, the following is safe:

```
string sStudentName;
cin >> sStudentName;
```

The string class will automatically allocate an array off the heap that's large enough to hold whatever data is input. Unfortunately, the `string` class is beyond the scope of a beginning book on programming.

Other Member Functions of the fstream Classes

The `fstream` classes provide a number of member functions, as shown in Table 31-3 (the list isn't a complete list of all the functions in these very large classes). The prototype declarations for these member functions reside in the `fstream` include file. They are described in the remainder of this section.

Table 31-3 Major Methods of the I/O Stream Classes

Method	*Meaning*
`bool bad()`	Returns true if a serious error has occurred.
`void clear(iostate flags = ios_base::goodbit)`	Clears (or sets) the I/O state flags.
`void close()`	Closes the file associated with a stream object.
`bool eof()`	Returns true if there are no more characters in the read pointer at the End of File.
`char fill()` `char fill(char newFill)`	Returns or sets the fill character.

(continued)

Table 31-3 *(continued)*

Method	*Meaning*
`fmtflags flags()` `fmtflags flags(fmtflags f)`	Returns or sets format flags. (See next section on format flags.)
`void flush()`	Flushes the output buffer to the disk.
`int gcount()`	Returns the number of bytes read during the last input.
`char get()`	Reads individual characters from file.

`char getline(`
`  char* buffer,`
`  int count,`
`  char delimiter = '\n')`
	Reads multiple characters up until either End of File, until delimiter encountered, or until `count - 1` characters read. Tacks a null onto the end of the line read. Does not store the delimiter read into the buffer. The delimiter defaults to newline, but you can provide a different one if you like.

`bool good()`	Returns true if no error conditions are set.

`void open(`
` const char* filename,`
` openmode mode)`
	Same arguments as the constructor. Performs the same file open on an existing object that the constructor performs when creating a new object.

`streamsize precision()`
`streamsize precision(`
`  streamsize s)`
	Reads or sets the number of digits displayed for floating point variables.

`ostream& put(char ch)`	Writes a single character to the stream.

`istream& read(`
`  char* buffer,`
`  streamsize num)`
	Reads a block of data. Reads either num bytes or until an End of File is encountered, whichever occurs first.

`fmtflags setf(fmtflags)`	Sets specific format flags. Returns old value.
`fmtflags unsetf(fmtflags)`	Clears specific format flags. Returns old value.
`int width()` `int width(int w)`	Reads or sets the number of characters to be displayed by the next formatted output statement.

`ostream& write(`
`  const char* buffer,`
`  streamsize num)`
	Writes a block of data to the output file.

Reading and writing streams directly

The inserter and extractor operators provide a convenient mechanism for reading formatted input. However, sometimes you just want to say, "Give it to me; I don't care what the format is." Several member functions are useful in this case.

The simplest function, `get()`, returns the next character in an input file. Its output equivalent is `put()`, which writes a single character to an output file. The function `getline()` returns a string of characters up to some terminator — the default terminator is a newline, but you can specify any other character you like as the third argument to the function. The `getline()` function strips off the terminator but makes no other attempt to reformat or otherwise interpret the input.

The member function `read()` is even more basic. This function reads the number of bytes that you specify, or less if the program encounters the End of File. The function `gcount()` always returns the actual number of bytes read. The output equivalent is `write()`.

The following FileCopy program uses the `read()` and `write()` functions to create a backup of any file you give it by making a copy with the string `".backup"` appended to the name:

```
//
//  CopyFiles - make backup copies of whatever files
//              are passed to the program by creating
//              a file with the same name plus the name
//              ".backup" appended.
//
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <cstring>
using namespace std;

void copyFile(const char* pszSrcFileName)
{
    // create a copy of the specified filename with
    // ",backup" added to the end
    int nTargetNameLength = strlen(pszSrcFileName) + 10;
    char *pszTargetFileName = new char[nTargetNameLength];
    strcpy(pszTargetFileName, pszSrcFileName);
    strcat(pszTargetFileName, ".backup");

    // now open the source file for input and
```

```
    // the target file for output
    ifstream input(pszSrcFileName,
                   ios_base::in | ios_base::binary);
    if (input.good())
    {
        ofstream output(pszTargetFileName,
      ios_base::out | ios_base::binary | ios_base::trunc);
        if (output.good())
        {

            while (!input.eof() && input.good())
            {
                char buffer[4096];
                input.read(buffer, 4096);
                output.write(buffer, input.gcount());
            }
        }
    }

    // restore memory to the heap
    delete pszTargetFileName;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    // pass every file name provided to main() to
    // the copyFile() function, one name at a time
    for (int i = 1; i < nNumberofArgs; i++)
    {
        cout << "Copying " << pszArgs[i] << endl;
        copyFile(pszArgs[i]);
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The program iterates through the arguments passed to it, remembering that `pszArgs[0]` points to the name of the program itself. The program passes each argument, one at a time, to the function `copyFile()`.

The `copyFile()` function first creates a copy of the name passed it in the array `pszTargetFileName`. It then concatenates the string `".backup"` to that name. Finally, you get to the good part: `copyFile()` opens the source file whose name was passed as the argument to the `copyFile()` function for binary input.

Note: The `ios_base::` is necessary when using the `in`, `out`, `binary`, and `trunc` flags as these are `const` static members of the `ios_base` class.

Use binary mode if you are working with non-text files or you don't intend to display the contents. In this case, I did not limit the program to work only with text files.

The function only continues executing if `input.good()` is `true`, indicating that the input object was created successfully, since it will be impossible to read from the file if the open did not work properly.

In a real-world program, I would have displayed some useful error message before returning to the caller.

If the input object is created okay, `copyFile()` creates an output object using the `pszTargetFileName` created earlier. This file is opened for binary output. The mode flag is also set to truncate to delete the contents of the target file if it already exists. If `output.good()` is `true`, the function executes the next section of the function; otherwise, control jumps to the end.

The function is now ready to copy the contents of one file to the other: it enters a loop in which it reads 4K blocks from the input file and writes them to the output file.

Notice in the call to `write()`, `copyFile()` uses the value returned from `input.gcount()` rather than a hardcoded 4096. This is because unless the source file just happens to be an integer multiple of 4096 bytes in length (not very likely), the last call to `read()` will fetch less than the requested number of bytes before encountering the End of File.

The loop terminates when either `input` reaches the End of File or the `input` object is no longer good.

The `!` operator (pronounced "the NOT operator") inverts the sense of a Boolean expression. In other words, `!true` is `false` and `!false` is `true`. (You read that last phrase as "NOT true is false and NOT false is true.")

Immediately before exiting, the function returns the `pszTargetFileName` array to the heap. Exiting the function causes the destructor for both `input` and `output` to be called, which closes both the input and output files.

To execute the program within the Code::Blocks environment, I first selected Project⇨Set Programs' Arguments to open the Select target dialog box. In the Program arguments field, I entered `main.cpp` and clicked OK. I could just as well have selected and dropped several files onto the name of the `CopyFiles` executable file or entered the command name followed by the names of the files to "backup" at the command prompt.

Chapter 18 discusses the various ways to pass arguments to your program.

When I run the program, I get the following output:

```
Copying main.cpp
Press any key to continue . . .
```

Looking into the folder containing the `main.cpp` source file, I now see a second `main.cpp.backup` file that has the identical size and contents as the original.

Controlling format

The `flags()`, `setf()`, and `unsetf()` member functions are all used to set or retrieve a set of format flags used to control the format of input extracted from an `ifstream` or inserted into an `ofstream` object. The flags get set to some default value that makes sense most of the time when the object is created. However, you can change these flags to control the format of input and/or output. Table 31-4 describes the flags that can be used with the `flags()`, `setf()`, and `unsetf()` member functions.

Table 31-4 I/O Stream Format Flags Used with setf(), unsetf(), and flags()

Flag	*If Flag Is True Then . . .*
`boolalpha`	Displays variables of type `bool` as either true or false rather than 1 or 0
`dec`	Reads or writes integers in decimal format (default)
`fixed`	Displays floating point in fixed point as opposed to scientific (default)
`hex`	Reads or writes integers in hexadecimal
`left`	Displays output left-justified (that is, pads on the right)
`oct`	Reads or writes integers in octal
`right`	Displays output right-justified (that is, pads on the left)
`scientific`	Displays floating point in scientific format
`showbase`	Displays a leading 0 for octal output and leading 0x for hexadecimal output
`showpoint`	Displays a decimal point for floating point output even if the fractional portion is zero
`skipws`	Skips over whitespace when reading using the extractor
`unitbuf`	Flushes output after each output operation
`uppercase`	Replaces lowercase letters with their uppercase equivalents on output

For example, the following code segment displays integer values in hexadecimal (rather than the default, decimal):

```
// fetch the previous value so we can restore it
ios_base::fmtflags prevValue = cout.flags();

// clear the decimal flag
cout.unsetf(cout.dec);

// now set the hexadecimal flag
cout.setf(cout.hex);

// ...do stuff...

// restore output to previous state
cout.flags(prevValue);
```

This example first queries the `cout` object for the current value of the format flags using the `flags()` member function. The type of the value returned is `ios_base::fmtflags`.

I didn't discuss user-defined types defined within classes — that's an advanced topic — so just trust me that this type makes sense.

It's always a good idea to record the format flags of an input or output object before changing them so that you can restore them to their previous value once you're finished.

The program then clears the decimal flag using the `unsetf()` function (it does this because hexadecimal, octal, and decimal are mutually exclusive format modes) before setting the hex mode using `setf()`. The `setf()` sets the hexadecimal flag without changing the value of any other format flags that may be set. Every time an integer is inserted into the `cout` object for the remainder of the function, C++ will display the value in hexadecimal.

Once the function finishes displaying values in hexadecimal format, it restores the previous value by calling `flags(fmtflags)`. This member function overwrites the current flags without whatever value you pass it.

Further format control is provided by the `width(int)` member function that sets the minimum width of the next output operation. In the event that the field doesn't take up the full width specified, the inserter adds the requisite number of fill characters. The default fill character is a space, but you change this by calling `fill(char)`. Whether C++ adds the fill characters on the left or right is determined by whether the `left` or `right` format flag is set.

For example, the code snippet

```
int i = 123;
cout.setf(cout.right);
cout.unsetf(cout.left);
cout.fill('+');
cout << "i = [";
cout.width(10);
cout << i;
cout << "]" << endl;
```

generates the following output:

```
i = [+++++++123]
```

Notice that the call to `width(int)` appears immediately before `cout << i`. Unlike the other formatting flags, the `width(int)` call applies only to the very next value that you insert. It must be reset after every value that you output.

What's up with endl?

Most programs in this book terminate an output stream by inserting the object `endl`. However, some programs include `\n` within the text to output a newline. What's the deal?

The `endl` object inserts a newline into the output stream, but it takes one more step. Disks are slow devices (compared to computer processors). Writing to disk more often than necessary will slow your program considerably. To avoid this, the `ofstream` class collects output into an internal buffer. The class writes the contents to disk when the buffer is full.

A memory buffer used to speed up output to a slow device like a disk is known as a *cache* — pronounced "cash." Writing the contents of the buffer to disk is known as *flushing the cache*.

The `endl` object adds a newline to the buffer and then flushes the cache to disk. You can also flush the cache manually by calling the member function `flush()`.

Note that C++ does not cache output to the standard error object, `cerr`.

Manipulating Manipulators

The span of some formatting member functions is fairly short. The best example of this is the `width(n)` member function — this function is good

only for the next value output. After that it must be reset. You saw this implication in the preceding snippet — the call to `cout.width(n)` had to appear right in the middle of the inserters:

```
cout << "i = [";
cout.width(10);
cout << i;
cout << "]" << endl;
```

The call to `cout.width(10)` is good only for the very next output `cout << i`; it has no effect on the following output `cout << "]"`.

Other functions have a short span, usually because you need to change their value often. For example, switching back and forth between decimal and hexadecimal mode while performing output requires multiple calls to `setf(hex)` and `setf(dec)` throughout the program.

Since this process can be a bit clumsy, C++ defines a more convenient means to invoke these common member functions. Table 31-5 defines a set of so-called *manipulators* that can be inserted directly in the output stream. These manipulators defined in the include file `iomanip` have the same effect as calling the corresponding member function.

Table 31-5 Common Manipulators and Their Equivalent Member Functions

Manipulator	*Member Function*	*Description*
`dec`	`setf(dec)`	Set display radix to decimal
`hex`	`setf(hex)`	Set display radix to hexadecimal
`oct`	`setf(oct)`	Set display radix to octal
`setfill(c)`	`fill(c)`	Set the fill character to c
`setprecision(n)`	`precision(n)`	Set the display precision to n
`setw(n)`	`width(n)`	Set the minimum field width for the next output to n

For example, the earlier snippet can be written as follows:

```
cout << "i = [" << setw(10) << i << "]" << endl;
```

I/O manipulators are nothing more than a labor-saving device — they don't add any new capability.

You must include `iomanip` if you intend to use I/O manipulators.

Using the stringstream Classes

After some practice, you get pretty good at parsing input from a file using the extractors and generating attractive output using the format controls provided with the inserter. It's a shame that you can't use that skill to parse character strings that are already in memory.

Well, of course, C++ provides just such a capability (I wouldn't have mentioned it otherwise). C++ provides two pairs of classes that allow you to parse a string in memory using the same member functions that you're accustomed to using for file I/O. An object of class `istrstream` or `istringstream` "looks and feels" like an `ifstream` object. Similarly, an object of class `ostrstream` or `ostringstream` responds to the same commands as an `ofstream` object.

The difference between the two sets of classes is not how they operate but how they are constructed. The `istrstream` class must be constructed with an ASCIIZ array as its base. All input is performed from this array. The `istringstream` class takes an object of class `string` as its base.

I don't discuss the `string` class in this book since in practice it's a little beyond the scope of a beginning programmer. However, the `string` class acts like an ASCIIZ array whose size changes automatically to conform to the size of the string it's asked to hold.

Similarly, the class `ostrstream` writes into an ASCIIZ array that is provided in the constructor, whereas the `ostringstream` class creates a `string` object for output.

The `istrstream` and `ostrstream` classes are defined in the `strstream` include file. The `istringstream` and `ostringstream` classes are defined in the `sstream` include file.

The following StringStream program parses `Student` information from an input file by first reading in a line using `getline()` before parsing it with `istrstream`.

The `strstream` classes are being phased out of the language in favor of the `stringstream` classes; however, I used the `strstream` classes here since they are based on the ASCIIZ character arrays that you are already familiar with. You will want to convert over to using the `stringstream` classes once you become familiar with the `string` class. The Code::Blocks compiler generates a warning when building the `StringStream` program that the `strstream` classes are *deprecated*, meaning that they are subject to removal.


```
// StringStream - demonstrate the use of string stream

//                classes for parsing input safely
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <strstream>
#include <cstring>
using namespace std;

struct Student
{
  protected:
    char szFirstName[128];
    char szLastName[128];
    int  nStudentID;

  public:
    Student(const char* pszFN, const char* pszLN,int nSID)
    {
        strncpy(szFirstName, pszFN, 128);
        strncpy(szLastName, pszLN, 128);
        nStudentID = nSID;
    }

    // display - write the student's data into the
    //           array provided; don't exceed the size
    //           of the array set by nLength
    void display(char* pszBuffer, int nLength)
    {
        ostrstream out(pszBuffer, nLength);

        out << szFirstName << " " << szLastName
            << " [" << nStudentID << "]" << ends;
    }
};

int main(int nNumberofArgs, char* pszArgs[])
{
    Student *pStudents[128];
    int nCount = 0;

    cout << "Input student <last name, first name ID>\n"
         << "(Input a blank line to stop input)" << endl;

    for(;;)
    {
        // get another line to parse
```

```
        char szLine[256];
        cin.getline(szLine, 256);

        // terminate if line is blank
        if (strlen(szLine) == 0)
        {
            break;
        }

        // associate an istrstream object with the
        // array that we just read
        istrstream input(szLine, 256);

        // now try to parse the buffer read
        char szLastName[256];
        char szFirstName[256];
        int nSSID;

        // read the last name up to a comma separator
        input.getline(szLastName, 256, ',');

        // read the first name until encountering white
        // space
        input >> szFirstName;

        // now read the student id
        input >> nSSID;

        // skip this line if anything didn't work
        if (input.fail())
        {
            cerr << "Bad input: " << szLine << endl;
            continue;
        }

        // create a Student object with the data
        // input and store it in the array of pointers
        pStudents[nCount++] = new Student(szFirstName,
                                          szLastName, nSSID);
    }

    // display the students input - use the Student's
    // output function to format the student data
    for(int n = 0; n < nCount; n++)
    {
        char szBuffer[256];
        pStudents[n]->display(szBuffer, 256);
        cout << szBuffer << endl;
    }
```

```
    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The program starts by creating an array of pointers that it will use to store the `Student` objects that it creates. It then prompts the user for the format that it expects for the student data to be read.

The program then enters a loop in which it first reads an entire line of input up to and including the terminating newline. If the length of the line read is zero, meaning that nothing was entered but a newline, the program breaks out of the input loop.

If something was input, the program associates an `istrstream` object input with the buffer. Subsequent read requests will be from this `szLine` buffer. The `istrstream` buffer must also tell the constructor how long the buffer is so that it doesn't read beyond the end.

The next section reads the last name, first name, and Social Security number.

These reads are safe — they cannot overflow the `szLastName` and `szFirstName` buffers because the extractor cannot possibly return more than 256 characters in any single read — that's how long the `szLine` array is.

Notice how the program calls `getline()` passing a ',' as the terminator. This reads characters up to and including the comma that separates the last name and first name.

Once the program has read the three student fields, it checks the input object to see if everything worked by calling `input.fail()`. If `fail()` is `true`, the program throws away whatever it read and spits back out the line to the user with an error message.

The `Student` constructor is typical of those you've seen elsewhere in the book. The program uses the `Student::display()` function to display the contents of a `Student` object. It does this in a fairly elegant fashion by simply associating an `ostrstream` object with the array provided and then inserting to the object. All `main()` has to do is output the result.

This is much more flexible than the alternative of inserting output directly to `cout` — the program can do anything it wants with the `szBuffer` array containing the `Student` data. It can write it to a file, send it to `cout`, or put it in a table, to name just three possibilities.

Notice that the last object `display()` inserts is the object `ends`. This is sort of the `strstream` version of `endl`; however, `ends` does not insert a newline. Instead, it inserts a null to terminate the ASCIIZ string within the buffer.

Always insert an `ends` last to terminate the ASCIIZ string that you build.

The output from the program appears as follows:

```
Input student <last name, first name ID>
(Input a blank line to stop input)
Davis, Stephen 12345678
Ashley 23456789
Bad input: Ashley 23456789
Webb, Jessalyn 34567890

Stephen Davis [12345678]
Jessalyn Webb [34567890]
Press any key to continue . . .
```

Notice how the second line is rejected since it doesn't follow the specified input format, but the program recovers gracefully to accept input again on the third line. This graceful recovery is very difficult to do any other way.

Chapter 32

I Take Exception!

In This Chapter

- Introducing the exception mechanism for handling program errors
- Examining the mechanism in detail
- Creating your own custom exception class

I know it's hard to accept, but occasionally programs don't work properly — not even mine. The traditional means of reporting a failure within a function is to return some indication to the caller, usually as a return value. Historically, C and C++ programmers have used 0 as the "all clear" indicator and anything else as meaning an error occurred — the exact value returned indicates the nature of the error.

The problem with this approach is that people generally don't check all of the possible error returns. It's too much trouble. And if you were to check all of the possible error returns, pretty soon you wouldn't see the "real code" because of all the error paths that are almost never executed.

Finally, you can embed just so much information in a single return value. For example, the `factorial()` function could return a –1 for "negative argument" (the factorial of a negative number is not defined) and a –2 for "argument too large" (factorials get large very quickly — factorial(100) is well beyond the range of an `int`). But if the program were to return a –2, wouldn't you like to know the value of that "too large argument"? There's no easy way to embed that information in the return.

The fathers (and mothers) of C++ decided that the language needed a better way of handling errors, so they invented the exception mechanism that has since been duplicated in many similar languages. Exceptions are the subject of this chapter.

The Exception Mechanism

The exception mechanism is a way for functions to report errors so that the error is handled even if the calling function does nothing. It's based on three

new keywords: `try`, `catch`, and `throw` (that's right, more variable names that you can't use). The exception mechanism works like this: A function *trys* to make it through a block of code without error. If the program does detect a problem, it *throws* an error indicator that a calling function can *catch* for processing.

The following FactorialException demonstrates how this works in ones and zeros:

```
// FactorialException - demonstrate the Exception error
//                      handling mechanism with a
//                      factorial function.
#include <cstdio>
#include <cstdlib>
#include <iostream>
using namespace std;

// factorial - compute factorial
int factorial(int n)
{
    // argument must be positive; throw exception if
    // n is negative
    if (n < 0)
    {
        throw "Argument for factorial is negative";
    }

    // anything over 100 will overflow
    if (n > 100)
    {
        throw "Argument too large";
    }

    // go ahead and calculate factorial
    int nAccum = 1;
    while(n > 1)
    {
        nAccum *= n--;
    }
    return nAccum;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    try
    {
        cout << "Factorial of 3 is "
             << factorial(3)
             << endl;

        cout << "Factorial of -1 is "
```

```
            << factorial(-1)
            << endl;

        cout << "Factorial of 5 is "
            << factorial(5)
            << endl;
    }
    catch(const char* pMsg)
    {
        cerr << "Error occurred: " << pMsg << endl;
    }
    catch(...)
    {
        cerr << "Unexpected error thrown" << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The `main()` function starts with the keyword `try` followed by an open brace and, eventually, a closed brace. Everything within the braces is said to be within a *try block*. The function then proceeds to display the factorial of three values: 3, –1, and 5. The only problem is that the factorial of a negative number is not defined.

You can see this within the `factorial()` function. This version of the function now contains a check for a negative argument and for an argument that is so large that it will overflow the `int`. In the event that either condition is true, control passes to a statement consisting of the keyword `throw` followed by an ASCIIZ string containing a description of the error.

Back in `main()`, at the end of the try block are two *catch phrases*. Each consists of the keyword `catch` followed by an argument. These catch phrases are designed to catch any exceptions thrown from within the try block. The first catch phrase will catch a pointer to an ASCIIZ string. This catch phrase displays the string. The second catch phrase, the one with the ellipses for an argument, is designed to catch anything. This wild-card catch phrase also displays a message, but since the catch phrase is so generic, it has no idea from where the exception was thrown or how to interpret the exception, so it just outputs a generic error message.

In practice, the program works like this: The first call to `factorial(3)` skips over both error conditions and returns the value 6. No problem so far.

The second call, `factorial(-1)` causes control to pass to the statement `throw "Argument for factorial is negative"`. This command

passes control immediately out of `factorial()` and to the end of the try block where C++ starts comparing the type of "`Argument for factorial is negative`" (which is `const char*` by the way — but you knew that) to each of the catch arguments.

Fortunately, the type of object thrown matches the type of the first catch phrase. This displays the string `"Error occurred:"` followed by the string thrown from within `factorial()`. Control then passes to the first statement after the last catch phrase, which is the usual call to `system("PAUSE")`.

In execution, the output from the program appears as follows:

```
Factorial of 3 is 6
Error occurred: Argument for factorial is negative
Press any key to continue . . .
```

Notice that the call to `factorial(5)` never gets executed. There is no way to return from a catch.

Examining the exception mechanism in detail

Now, take a closer look at how C++ processes an exception.

When C++ encounters a throw, it first copies the object thrown to some neutral place other than the local memory of the function. It then starts looking in the current function for the end of the current try block. If it does not encounter one, it then executes a return from the function and continues the search. C++ continues to return and search, return and search until it finds the end of the current try block. This is known as *unwinding the stack*.

An important feature of stack unwinding is that as each stack is unwound, objects that go out of scope are destructed just as though the function had executed a `return` statement. This keeps the program from losing assets or leaving objects dangling.

When an enclosing try block is found, the code searches the first catch phrase to see if the argument type matches the object thrown. If not, it checks the next catch phrase and so on until a match is found.

If no matching catch phrase is found, then C++ resumes looking for the next higher try block in an ever outward spiral until an appropriate catch can be found. If no matching catch phrase is found, the program terminates.

Once a catch phrase is found, the exception is said to be handled and control passes to the statement following the last catch phrase.

The phrase `catch(...)` catches all exceptions.

Special considerations for throwing

I need to mention a few special considerations in regard to throwing exceptions. You need to be careful not to throw a pointer to an object in local memory. As the stack is unwound, all local variables are destroyed. C++ will copy the object into a safe memory location to keep it from being destroyed, but there's no way that C++ can tell what a pointer might be pointing to.

Note that I avoided this problem in the earlier example by throwing a pointer to a `const` string — these are kept in a different memory area and not on the stack. You'll see a better way to avoid this problem in the next section.

Don't catch an exception if you don't know what to do with the error. That may sound obvious, but it isn't really. The exception mechanism allows programmers to handle errors at a level at which they can truly do something about them. For example, if you are writing a data storage function and you get an exception from a write to the disk, there's not much point in catching it. The destructor for the output object should close the file, and C++ calls that automagically. Better to let the error propagate up to a level where the program knows what it's trying to do.

A catch phrase can rethrow an exception by executing the keyword `throw;` alone (without an argument). This allows the programmer to partially process an error. For example, a database function might catch an exception, close any open tables or databases, and rethrow the exception to the application software to be handled there for good. (Assuming that the destructors haven't done that stuff already.)

Finally, a function can declare the types of objects that it will throw as part of the declaration. In other words, I could have declared `factorial()` as follows:

```
int factorial(int n) throw(const char*);
```

I say "could" because, though some people consider exception declarations good form, they aren't mandatory. It isn't even clear, if exception declarations are a good idea. (Personally, I don't think so.)

If you declare the types of object that a function throws, then each of the throws is compared to that list, and an error is generated if the function tries to throw something else. If you do not include a throw in the declaration, then the function can throw anything it likes.

Creating a Custom Exception Class

The thing following a throw is actually an expression that creates an object of some kind. In the earlier example, the object was a pointer, but it could have been any object that you like (with one exception that I'll mention a little later in this section).

For example, I could have created my own class specifically for the purpose of holding information about errors. For the `factorial()` example, I could have created a class `ArgOutOfRange` that included everything you need to know about out-of-range arguments. In this way, I could store as much information as needed to debug the error (if it is an error), process the exception, and report the problem accurately to the user.

The following CustomExceptionClass program creates an `ArgOutOfRange` class and uses it to provide an accurate description of the error encountered in `factorial()`:

```
// CustomExceptionClass - demonstrate the flexibility of
//                    the exception mechanism by creating
//                    a custom exception class.
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
#include <exception>
#include <strstream>
using namespace std;

class ArgOutOfRange : public exception
{
  protected:
    char szMsg[256];
    int nValue;
    int nMaxLegal;
    int nMinLegal;

  public:
    ArgOutOfRange(const char* pszFName, int nVal,
                  int nMin = 0, int nMax = 0)
    {
```

```
        nValue = nVal;
        nMinLegal = nMin;
        nMaxLegal = nMax;

        ostrstream out(szMsg, 256);
        out << "Argument out of range in " << pszFName
            << ", arg is " << nValue;
        if (nMin != nMax)
        {
            out << ", legal range is "
                << nMin << " to " << nMax;
        }
        out << ends;
    }

    virtual const char* what()
    {
        return szMsg;
    }
};


// factorial - compute factorial
int factorial(int n)
{
    // argument must be positive; throw exception if
    // n is negative
    if (n < 0)
    {
        throw ArgOutOfRange("factorial()", n, 0, 100);
    }

    // anything over 100 will overflow
    if (n > 100)
    {
        throw ArgOutOfRange("factorial()", n, 0, 100);
    }

    // go ahead and calculate factorial
    int nAccum = 1;
    while(n > 1)
    {
        nAccum *= n--;
    }
    return nAccum;
}

int main(int nNumberofArgs, char* pszArgs[])
{
    try
```

```
    {
        cout << "Factorial of 3 is "
             << factorial(3)
             << endl;

        cout << "Factorial of -1 is "
             << factorial(-1)
             << endl;

        cout << "Factorial of 5 is "
             << factorial(5)
             << endl;
    }
    catch(ArgOutOfRange e)
    {
        cerr << "Error occurred:\n" << e.what() << endl;
    }
    catch(...)
    {
        cerr << "Unexpected error thrown" << endl;
    }

    // wait until user is ready before terminating program
    // to allow the user to see the program results
    system("PAUSE");
    return 0;
}
```

The `main()` program starts just like the previous example. The `factorial()` function contains the same tests. Rather than throw a simple character string, however, this version of `factorial()` throws an object of class `ArgOutRange`. The constructor for `ArgOutOfRange` provides room for the name of the function, the value of the offending argument, and the range of legal values for the argument.

All the real work is done in the `ArgOutOfRange` class. First, this class extends the class `exception`, which is defined in the `exception` include file. The `exception` class defines the virtual member function `what()` that you must override with a version that outputs your message. Everything else is optional.

User-defined exception classes should extend `exception` so that C++ will know what to do with your exception should you fail to catch it.

The constructor to `ArgOutOfRange` accepts the name of the function, the value of the argument, and the minimum and maximum legal argument values. Providing a default value for these arguments makes them optional.

The constructor uses the `ostrstream` class (discussed in Chapter 31) to create a complex description of the problem in the internal array `szMsg`. It also saves off the arguments themselves.

A complete version of `ArgOutOfRange` would provide access functions to allow each of these values to be queried from the application code, if desired. I have to leave these details out in order to keep the programs as short as possible.

Back in `factorial()`, the two throws now throw `ArgOutOfRange` objects with the appropriate information. The catch back in `main()` is for an `ArgOutOfRange` object. This block does nothing more than display an error message along with the description returned by `ArgOutRange::what()`.

Since all the real work was done in the constructor, the `what()` function doesn't have to do anything except return a pointer to the message stored within the object.

The output from the program is now very descriptive:

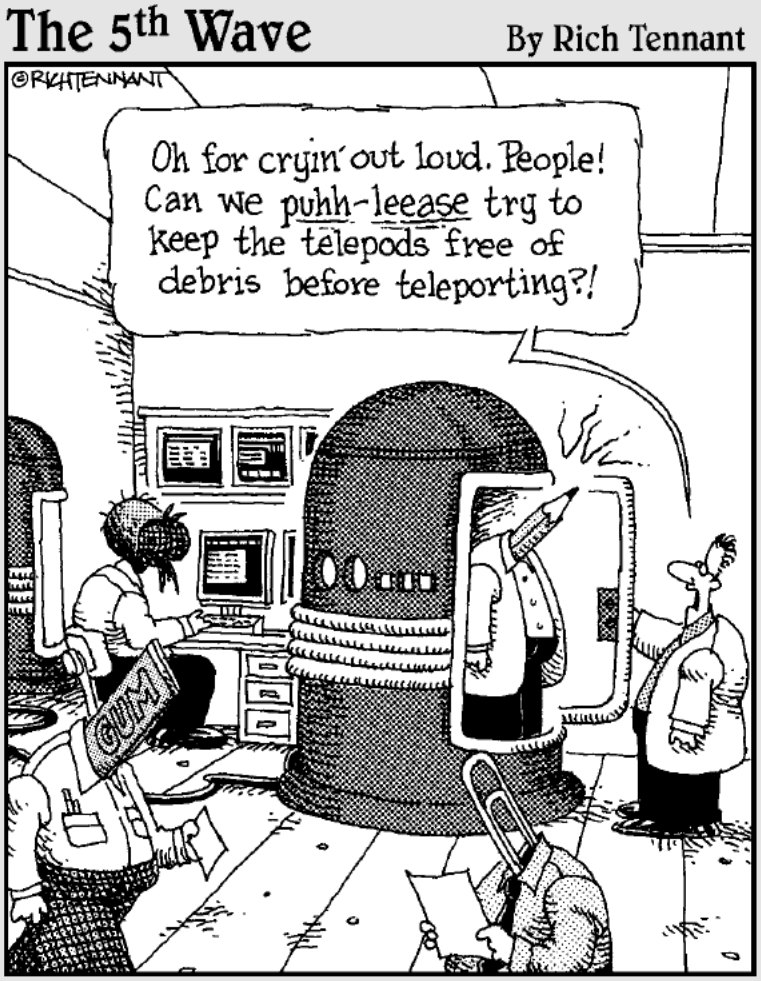
```
Factorial of 3 is 6
Error occurred:
Argument out of range in factorial(), arg is -1, legal
          range is 0 to 100
Press any key to continue . . .
```

Restrictions on exception classes

I mentioned that the exception mechanism can throw almost any type of object. The only real restriction is that the class must be copyable. That means either the default copy constructor provided by C++ is sufficient (that was the case for `ArgOutOfRange`) or the class provides its own copy constructor.

This restriction is because C++ has to copy the exception object out of local storage and to some "safe place" before unwinding the stack. C++ uses the copy constructor again to copy the object to the catch's storage area.

Part VII
The Part of Tens



In this part . . .

No *For Dummies* book would be complete without its Part of Tens. In this part, you'll see ten ways to avoid the most common coding mistakes and ten advanced language features you may want to tackle when you're a little more experienced with the C++ language.

Chapter 33

Ten Ways to Avoid Bugs

In This Chapter

- Enable all compiler warning messages
- Adopt a clear and consistent coding style
- Comment your code while you write it
- Single-step every path in the debugger at least once
- Limit the visibility of members
- Keep track of heap memory
- Zero out pointers after deleting what they point to
- Use exceptions to handle errors
- Declare destructors virtual
- Provide a copy constructor and overloaded assignment operator

It's an unfortunate fact that you will spend more time searching for and removing bugs than you will spend actually writing your programs in the first place. The suggestions in this section may help you minimize the number of errors you introduce into your programs to make programming a more enjoyable experience.

Enable All Warnings and Error Messages

The syntax of C++ allows for a lot of error checking. When the compiler encounters a construct that it just can't decipher, it has no choice but to output a message. It tries to sync back up with the source code (sometimes less than successfully), but it will not generate an executable. This forces the programmer to fix all error messages — she has no choice.

However, when C++ comes across a structure that it can figure out but the structure smells fishy anyway, C++ generates a warning message. Because C++ is pretty sure that it understands what you want, it goes ahead and creates an executable file so you can ignore warnings if you like. In fact, if you really don't want to be bothered, you can disable warnings.