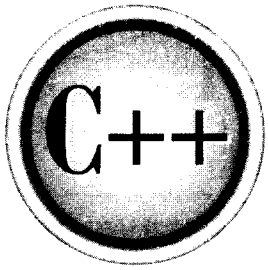
The Complete Reference



Chapter 13

Arrays, Pointers, References, and the Dynamic Allocation Operators

In Part One, pointers and arrays were examined as they relate to C++'s built-in types. Here, they are discussed relative to objects. This chapter also looks at a feature related to the pointer called a *reference*. The chapter concludes with an examination of C++'s dynamic allocation operators.

Arrays of Objects

In C++, it is possible to have arrays of objects. The syntax for declaring and using an object array is exactly the same as it is for any other type of array. For example, this program uses a three-element array of objects:

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

class cl {
  int i;
public:
  void set_i(int j) { i=j; }
  int get_i() { return i; }
};

int main()
{
  cl ob[3];
  int i;

  for(i=0; i<3; i++) ob[i].set_i(i+1);

  for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";

  return 0;
}
```

This program displays the numbers **1**, **2**, and **3** on the screen.

If a class defines a parameterized constructor, you may initialize each object in an array by specifying an initialization list, just like you do for other types of arrays. However, the exact form of the initialization list will be decided by the number of parameters required by the object's constructors. For objects whose constructors have only one parameter, you can simply specify a list of initial values, using the normal array-initialization syntax. As each element in the array is created, a value from the

list is passed to the constructor's parameter. For example, here is a slightly different version of the preceding program that uses an initialization:

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

class cl {
  int i;
public:
  cl(int j) { i=j; } // constructor
  int get_i() { return i; }
};

int main()
{
  cl ob[3] = {1, 2, 3}; // initializers
  int i;

  for(i=0; i<3; i++)
    cout << ob[i].get_i() << "\n";

  return 0;
}
```

As before, this program displays the numbers **1**, **2**, and **3** on the screen.

Actually, the initialization syntax shown in the preceding program is shorthand for this longer form:

```
cl ob[3] = { cl(1), cl(2), cl(3) };
```

Here, the constructor for **cl** is invoked explicitly. Of course, the short form used in the program is more common. The short form works because of the automatic conversion that applies to constructors taking only one argument (see Chapter 12). Thus, the short form can only be used to initialize object arrays whose constructors only require one argument.

If an object's constructor requires two or more arguments, you will have to use the longer initialization form. For example,

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

```
class cl {
  int h;
  int i;
public:
  cl(int j, int k) { h=j; i=k; } // constructor with 2 parameters
  int get_i() {return i;}
  int get_h() {return h;}
};

int main()
{
  cl ob[3] = {
    cl(1, 2), // initialize
    cl(3, 4),
    cl(5, 6)
  };

  int i;

  for(i=0; i<3; i++) {
    cout << ob[i].get_h();
    cout << ", ";
    cout << ob[i].get_i() << "\n";
  }

  return 0;
}
```

Here, **cl**'s constructor has two parameters and, therefore, requires two arguments. This means that the shorthand initialization format cannot be used and the long form, shown in the example, must be employed.

Creating Initialized vs. Uninitialized Arrays

A special case situation occurs if you intend to create both initialized and uninitialized arrays of objects. Consider the following **class.**

```
class cl {
  int i;
public:
  cl(int j) { i=j; }
```

```
  int get_i() { return i; }
};
```

Here, the constructor defined by **cl** requires one parameter. This implies that any array declared of this type must be initialized. That is, it precludes this array declaration:

```
cl a[9]; // error, constructor requires initializers
```

The reason that this statement isn't valid (as **cl** is currently defined) is that it implies that **cl** has a parameterless constructor because no initializers are specified. However, as it stands, **cl** does not have a parameterless constructor. Because there is no valid constructor that corresponds to this declaration, the compiler will report an error. To solve this problem, you need to overload the constructor, adding one that takes no parameters, as shown next. In this way, arrays that are initialized and those that are not are both allowed.

```
class cl {
  int i;
public:
  cl() { i=0; } // called for non-initialized arrays
  cl(int j) { i=j; } // called for initialized arrays
  int get_i() { return i; }
};
```

Given this **class**, both of the following statements are permissible:

```
cl a1[3] = {3, 5, 6}; // initialized

cl a2[34]; // uninitialized
```

Pointers to Objects

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow (->) operator instead of the dot operator. The next program illustrates how to access an object given a pointer to it:

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

```
class cl {
  int i;
public:
  cl(int j) { i=j; }
  int get_i() { return i; }
};

int main()
{
  cl ob(88), *p;

  p = &ob; // get address of ob

  cout << p->get_i(); // use -> to call get_i()

  return 0;
}
```

As you know, when a pointer is incremented, it points to the next element of its type. For example, an integer pointer will point to the next integer. In general, all pointer arithmetic is relative to the base type of the pointer. (That is, it is relative to the type of data that the pointer is declared as pointing to.) The same is true of pointers to objects. For example, this program uses a pointer to access all three elements of array **ob** after being assigned **ob**'s starting address:

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

class cl {
  int i;
public:
  cl() { i=0; }
  cl(int j) { i=j; }
  int get_i() { return i; }
};

int main()
{
  cl ob[3] = {1, 2, 3};
```

```
  cl *p;
  int i;

  p = ob; // get start of array
  for(i=0; i<3; i++) {
    cout << p->get_i() << "\n";
    p++; // point to next object
  }

  return 0;
}
```

You can assign the address of a public member of an object to a pointer and then access that member by using the pointer. For example, this is a valid C++ program that displays the number **1** on the screen:

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

class cl {
public:
  int i;
  cl(int j) { i=j; }
};

int main()
{
  cl ob(1);
  int *p;

  p = &ob.i; // get address of ob.i

  cout << *p; // access ob.i via p

  return 0;
}
```

Because **p** is pointing to an integer, it is declared as an integer pointer. It is irrelevant that **i** is a member of **ob** in this situation.

Type Checking C++ Pointers

There is one important thing to understand about pointers in C++: You may assign one pointer to another only if the two pointer types are compatible. For example, given:

```
int *pi;
float *pf;
```

in C++, the following assignment is illegal:

```
pi = pf; // error -- type mismatch
```

Of course, you can override any type incompatibilities using a cast, but doing so bypasses C++'s type-checking mechanism.

The this Pointer

When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called). This pointer is called **this**. To understand **this**, first consider a program that creates a class called **pwr** that computes the result of a number raised to some power:

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

class pwr {
  double b;
  int e;
  double val;
public:
  pwr(double base, int exp);
  double get_pwr() { return val; }
};

pwr::pwr(double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--) val = val * b;
```

```
}

int main()
{
  pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);

  cout << x.get_pwr() << " ";
  cout << y.get_pwr() << " ";
  cout << z.get_pwr() << "\n";

  return 0;
}
```

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus, inside **pwr( )**, the statement

```
b = base;
```

means that the copy of **b** associated with the invoking object will be assigned the value contained in **base**. However, the same statement can also be written like this:

```
this->b = base;
```

The **this** pointer points to the object that invoked **pwr( )**. Thus, **this ->b** refers to that object's copy of **b**. For example, if **pwr( )** had been invoked by **x** (as in **x(4.0, 2)**), then **this** in the preceding statement would have been pointing to **x**. Writing the statement without using **this** is really just shorthand.

Here is the entire **pwr( )** constructor written using the **this** pointer:

```
pwr::pwr(double base, int exp)
{
  this->b = base;
  this->e = exp;
  this->val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--)
    this->val = this->val * this->b;
}
```

Actually, no C++ programmer would write **pwr( )** as just shown because nothing is gained, and the standard form is easier. However, the **this** pointer is very important

when operators are overloaded and whenever a member function must utilize a pointer to the object that invoked it.

Remember that the **this** pointer is automatically passed to all member functions. Therefore, **get_pwr( )** could also be rewritten as shown here:

```
double get_pwr() { return this->val; }
```

In this case, if **get_pwr( )** is invoked like this:

```
y.get_pwr();
```

then **this** will point to object **y**.

Two final points about **this**. First, **friend** functions are not members of a class and, therefore, are not passed a **this** pointer. Second, **static** member functions do not have a **this** pointer.

Pointers to Derived Types

In general, a pointer of one type cannot point to an object of a different type. However, there is an important exception to this rule that relates only to derived classes. To begin, assume two classes called **B** and **D**. Further, assume that **D** is derived from the base class **B**. In this situation, a pointer of type **B** * may also point to an object of type **D**. More generally, a base class pointer can also be used as a pointer to an object of any class derived from that base.

Although a base class pointer can be used to point to a derived object, the opposite is not true. A pointer of type **D** * may not point to an object of type **B**. Further, although you can use a base pointer to point to a derived object, you can access only the members of the derived type that were inherited from the base. That is, you won't be able to access any members added by the derived class. (You can cast a base pointer into a derived pointer and gain full access to the entire derived class, however.)

Here is a short program that illustrates the use of a base pointer to access derived objects.

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

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i() { return i; }
```

```
};
class derived: public base {
  int j;
public:
  void set_j(int num) { j=num; }
  int get_j() { return j; }
};

int main()
{
  base *bp;
  derived d;

  bp = &d; // base pointer points to derived object

  // access derived object using base pointer
  bp->set_i(10);
  cout << bp->get_i() << " ";

/* The following won't work. You can't access elements of
   a derived class using a base class pointer.

   bp->set_j(88); // error
   cout << bp->get_j(); // error

*/
  return 0;
}
```

As you can see, a base pointer is used to access an object of a derived class.

Although you must be careful, it is possible to cast a base pointer into a pointer of the derived type to access a member of the derived class through the base pointer. For example, this is valid C++ code:

```
// access now allowed because of cast
((derived *)bp)->set_j(88);
cout << ((derived *)bp)->get_j();
```

It is important to remember that pointer arithmetic is relative to the base type of the pointer. For this reason, when a base pointer is pointing to a derived object, incrementing the pointer does not cause it to point to the next object of the derived type. Instead, it will point to what it thinks is the next object of the base type. This,

of course, usually spells trouble. For example, this program, while syntactically correct, contains this error.

```
// This program contains an error.
#include <iostream>
using namespace std;

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i() { return i; }
};

class derived: public base {
  int j;
public:
  void set_j(int num) {j=num;}
  int get_j() {return j;}
};

int main()
{
  base *bp;
  derived d[2];

  bp = d;

  d[0].set_i(1);
  d[1].set_i(2);

  cout << bp->get_i() << " ";
  bp++; // relative to base, not derived
  cout << bp->get_i(); // garbage value displayed

  return 0;
}
```

The use of base pointers to derived types is most useful when creating run-time polymorphism through the mechanism of virtual functions (see Chapter 17).

Pointers to Class Members

C++ allows you to generate a special type of pointer that "points" generically to a member of a class, not to a specific instance of that member in an object. This sort of pointer is called a *pointer to a class member* or a *pointer-to-member*, for short. A pointer to a member is not the same as a normal C++ pointer. Instead, a pointer to a member provides only an offset into an object of the member's class at which that member can be found. Since member pointers are not true pointers, the . and -> cannot be applied to them. To access a member of a class given a pointer to it, you must use the special pointer-to-member operators .* and ->*. Their job is to allow you to access a member of a class given a pointer to that member.

Here is an example:

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

class cl {
public:
  cl(int i) { val=i; }
  int val;
  int double_val() { return val+val; }
};

int main()
{
  int cl::*data; // data member pointer
  int (cl::*func)(); // function member pointer
  cl ob1(1), ob2(2); // create objects

  data = &cl::val; // get offset of val
  func = &cl::double_val; // get offset of double_val()

  cout << "Here are values: ";
  cout << ob1.*data << " " << ob2.*data << "\n";

  cout << "Here they are doubled: ";
  cout << (ob1.*func)() << " ";
  cout << (ob2.*func)() << "\n";

  return 0;
}
```

In **main( )**, this program creates two member pointers: **data** and **func**. Note carefully the syntax of each declaration. When declaring pointers to members, you must specify the class and use the scope resolution operator. The program also creates objects of **cl** called **ob1** and **ob2**. As the program illustrates, member pointers may point to either functions or data. Next, the program obtains the addresses of **val** and **double_val( )**. As stated earlier, these "addresses" are really just offsets into an object of type **cl**, at which point **val** and **double_val( )** will be found. Next, to display the values of each object's **val**, each is accessed through **data**. Finally, the program uses **func** to call the **double_val( )** function. The extra parentheses are necessary in order to correctly associate the **.*** operator.

When you are accessing a member of an object by using an object or a reference (discussed later in this chapter), you must use the **.*** operator. However, if you are using a pointer to the object, you need to use the **->*** operator, as illustrated in this version of the preceding program:

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

class cl {
public:
  cl(int i) { val=i; }
  int val;
  int double_val() { return val+val; }
};

int main()
{
  int cl::*data; // data member pointer
  int (cl::*func)(); // function member pointer
  cl ob1(1), ob2(2); // create objects
  cl *p1, *p2;

  p1 = &ob1; // access objects through a pointer
  p2 = &ob2;

  data = &cl::val; // get offset of val
  func = &cl::double_val; // get offset of double_val()

  cout << "Here are values: ";
  cout << p1->*data << " " << p2->*data << "\n";

  cout << "Here they are doubled: ";
```

```
  cout << (p1->*func)() << " ";
  cout << (p2->*func)() << "\n";

  return 0;
}
```

In this version, **p1** and **p2** are pointers to objects of type **cl**. Therefore, the **->*** operator is used to access **val** and **double_val( )**.

Remember, pointers to members are different from pointers to specific instances of elements of an object. Consider this fragment (assume that **cl** is declared as shown in the preceding programs):

```
int cl::*d;
int *p;
cl o;

p = &o.val // this is address of a specific val

d = &cl::val // this is offset of generic val
```

Here, **p** is a pointer to an integer inside a *specific* object. However, **d** is simply an offset that indicates where **val** will be found in any object of type **cl**.

In general, pointer-to-member operators are applied in special-case situations. They are not typically used in day-to-day programming.

References

C++ contains a feature that is related to the pointer called a *reference*. A reference is essentially an implicit pointer. There are three ways that a reference can be used: as a function parameter, as a function return value, or as a stand-alone reference. Each is examined here.

Reference Parameters

Probably the most important use for a reference is to allow you to create functions that automatically use call-by-reference parameter passing. As explained in Chapter 6, arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function. Call-by-reference passes the address of the argument to the function. By default, C++ uses call-by-value, but it provides two ways to achieve call-by-reference parameter passing. First, you can explicitly pass a pointer to the argument. Second,

you can use a reference parameter. For most circumstances the best way is to use a reference parameter.

To fully understand what a reference parameter is and why it is valuable, we will begin by reviewing how a call-by-reference can be generated using a pointer parameter. The following program manually creates a call-by-reference parameter using a pointer in the function called **neg( )**, which reverses the sign of the integer variable pointed to by its argument.

```
// Manually create a call-by-reference using a pointer.
#include <iostream>
using namespace std;

void neg(int *i);

int main()
{
  int x;

  x = 10;
  cout << x << " negated is ";

  neg(&x);
  cout << x << "\n";

  return 0;
}

void neg(int *i)
{
  *i = -*i;
}
```

In this program, **neg( )** takes as a parameter a pointer to the integer whose sign it will reverse. Therefore, **neg( )** must be explicitly called with the address of **x**. Further, inside **neg( )** the * operator must be used to access the variable pointed to by **i**. This is how you generate a "manual" call-by-reference in C++, and it is the only way to obtain a call-by-reference using the C subset. Fortunately, in C++ you can automate this feature by using a reference parameter.

To create a reference parameter, precede the parameter's name with an **&**. For example, here is how to declare **neg( )** with **i** declared as a reference parameter:

```
void neg(int &i);
```

For all practical purposes, this causes **i** to become another name for whatever argument **neg( )** is called with. Any operations that are applied to **i** actually affect the calling argument. In technical terms, **i** is an implicit pointer that automatically refers to the argument used in the call to **neg( )**. Once **i** has been made into a reference, it is no longer necessary (or even legal) to apply the * operator. Instead, each time **i** is used, it is implicitly a reference to the argument and any changes made to **i** affect the argument. Further, when calling **neg( )**, it is no longer necessary (or legal) to precede the argument's name with the **&** operator. Instead, the compiler does this automatically. Here is the reference version of the preceding program:

```
// Use a reference parameter.
#include <iostream>
using namespace std;

void neg(int &i); // i now a reference

int main()
{
  int x;

  x = 10;
  cout << x << " negated is ";

  neg(x); // no longer need the & operator
  cout << x << "\n";

  return 0;
}

void neg(int &i)
{
  i = -i; // i is now a reference, don't need *
}
```

To review: When you create a reference parameter, it automatically refers to (implicitly points to) the argument used to call the function. Therefore, in the preceding program, the statement

```
i = -i ;
```

actually operates on **x**, not on a copy of **x**. There is no need to apply the **&** operator to an argument. Also, inside the function, the reference parameter is used directly without

the need to apply the * operator. In general, when you assign a value to a reference, you are actually assigning that value to the variable that the reference points to.

Inside the function, it is not possible to change what the reference parameter is pointing to. That is, a statement like

```
i++;
```

inside **neg( )** increments the value of the variable used in the call. It does not cause **i** to point to some new location.

Here is another example. This program uses reference parameters to swap the values of the variables it is called with. The **swap( )** function is the classic example of call-by-reference parameter passing.

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

void swap(int &i, int &j);

int main()
{
  int a, b, c, d;

  a = 1;
  b = 2;
  c = 3;
  d = 4;

  cout << "a and b: " << a << " " << b << "\n";
  swap(a, b); // no & operator needed
  cout << "a and b: " << a << " " << b << "\n";

  cout << "c and d: " << c << " " << d << "\n";
  swap(c, d);
  cout << "c and d: " << c << " " << d << "\n";

  return 0;
}

void swap(int &i, int &j)
{
  int t;

  t = i; // no * operator needed
```

```
  i = j;
  j = t;
}
```

This program displays the following:

```
a and b: 1 2
a and b: 2 1
c and d: 3 4
c and d: 4 3
```

Passing References to Objects

In Chapter 12 it was explained that when an object is passed as an argument to a function, a copy of that object is made. When the function terminates, the copy's destructor is called. However, when you pass by reference, no copy of the object is made. This means that no object used as a parameter is destroyed when the function terminates, and the parameter's destructor is not called. For example, try this program:

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

class cl {
  int id;
public:
  int i;
  cl(int i);
  ~cl();
  void neg(cl &o) { o.i = -o.i; } // no temporary created
};

cl::cl(int num)
{
  cout << "Constructing " << num << "\n";
  id = num;
}

cl::~cl()
{
  cout << "Destructing " << id << "\n";
}
```

```
int main()
{
  c1 o(1);

  o.i = 10;
  o.neg(o);

  cout << o.i << "\n";

  return 0;
}
```

Here is the output of this program:

```
Constructing 1
-10
Destructing 1
```

As you can see, only one call is made to **c1**'s destructor. Had **o** been passed by value, a second object would have been created inside **neg( )**, and the destructor would have been called a second time when that object was destroyed at the time **neg( )** terminated.

As the code inside **neg( )** illustrates, when you access a member of a class through a reference, you use the dot operator. The arrow operator is reserved for use with pointers only.

When passing parameters by reference, remember that changes to the object inside the function affect the calling object.

One other point: Passing all but the smallest objects by reference is faster than passing them by value. Arguments are usually passed on the stack. Thus, large objects take a considerable number of CPU cycles to push onto and pop from the stack.

Returning References

A function may return a reference. This has the rather startling effect of allowing a function to be used on the left side of an assignment statement! For example, consider this simple program:

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

char &replace(int i); // return a reference
```

```
char s[80] = "Hello There";

int main()
{

  replace(5) = 'X'; // assign X to space after Hello

  cout << s;

  return 0;
}

char &replace(int i)
{
  return s[i];
}
```

This program replaces the space between **Hello** and **There** with an **X**. That is, the program displays **HelloXthere**. Take a look at how this is accomplished. First, **replace( )** is declared as returning a reference to a character. As **replace( )** is coded, it returns a reference to the element of **s** that is specified by its argument **i.** The reference returned by **replace( )** is then used in **main( )** to assign to that element the character **X**.

One thing you must be careful about when returning references is that the object being referred to does not go out of scope after the function terminates.

Independent References

By far the most common uses for references are to pass an argument using call-by-reference and to act as a return value from a function. However, you can declare a reference that is simply a variable. This type of reference is called an *independent reference.*

When you create an independent reference, all you are creating is another name for an object. All independent references must be initialized when they are created. The reason for this is easy to understand. Aside from initialization, you cannot change what object a reference variable points to. Therefore, it must be initialized when it is declared. (In C++, initialization is a wholly separate operation from assignment.)

The following program illustrates an independent reference:

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

```
int main()
{
  int a;
  int &ref = a; // independent reference

  a = 10;
  cout << a << " " << ref << "\n";

  ref = 100;
  cout << a << " " << ref << "\n";

  int b = 19;
  ref = b; // this puts b's value into a
  cout << a << " " << ref << "\n";

  ref--; // this decrements a
         // it does not affect what ref refers to

  cout << a << " " << ref << "\n";

  return 0;
}
```

The program displays this output:

```
10 10
100 100
19 19
18 18
```

Actually, independent references are of little real value because each one is, literally, just another name for another variable. Having two names to describe the same object is likely to confuse, not organize, your program.

References to Derived Types

Similar to the situation as described for pointers earlier, a base class reference can be used to refer to an object of a derived class. The most common application of this is found in function parameters. A base class reference parameter can receive objects of the base class as well as any other type derived from that base.

Restrictions to References

There are a number of restrictions that apply to references. You cannot reference another reference. Put differently, you cannot obtain the address of a reference. You cannot create arrays of references. You cannot create a pointer to a reference. You cannot reference a bit-field.

A reference variable must be initialized when it is declared unless it is a member of a class, a function parameter, or a return value. Null references are prohibited.

A Matter of Style

When declaring pointer and reference variables, some C++ programmers use a unique coding style that associates the * or the & with the type name and not the variable. For example, here are two functionally equivalent declarations:

```
int& p; // & associated with type
int &p; // & associated with variable
```

Associating the * or & with the type name reflects the desire of some programmers for C++ to contain a separate pointer type. However, the trouble with associating the & or * with the type name rather than the variable is that, according to the formal C++ syntax, neither the & nor the * is distributive over a list of variables. Thus, misleading declarations are easily created. For example, the following declaration creates *one, not two*, integer pointers.

```
int* a, b;
```

Here, **b** is declared as an integer (not an integer pointer) because, as specified by the C++ syntax, when used in a declaration, the * (or &) is linked to the individual variable that it precedes, not to the type that it follows. The trouble with this declaration is that the visual message suggests that both **a** and **b** are pointer types, even though, in fact, only **a** is a pointer. This visual confusion not only misleads novice C++ programmers, but occasionally old pros, too.

It is important to understand that, as far as the C++ compiler is concerned, it doesn't matter whether you write **int *p** or **int* p.** Thus, if you prefer to associate the * or & with the type rather than the variable, feel free to do so. However, to avoid confusion, this book will continue to associate the * and the & with the variables that they modify rather than their types.

C++'s Dynamic Allocation Operators

C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time. Dynamic allocation is an important part

of almost all real-world programs. As explained in Part One, C++ also supports dynamic memory allocation functions, called **malloc( )** and **free( ).** These are included for the sake of compatibility with C. However, for C++ code, you should use the **new** and **delete** operators because they have several advantages.

The **new** operator allocates memory and returns a pointer to the start of it. The **delete** operator frees memory previously allocated using **new**. The general forms of **new** and **delete** are shown here:

p_var = new *type*;
delete *p_var*;

Here, *p_var* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*.

Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then **new** will fail and a **bad_alloc** exception will be generated. This exception is defined in the header **<new>**. Your program should handle this exception and take appropriate action if a failure occurs. (Exception handling is described in Chapter 19.) If this exception is not handled by your program, then your program will be terminated.

The actions of **new** on failure as just described are specified by Standard C++. The trouble is that not all compilers, especially older ones, will have implemented **new** in compliance with Standard C++. When C++ was first invented, **new** returned null on failure. Later, this was changed such that **new** caused an exception on failure. Finally, it was decided that a **new** failure will generate an exception by default, but that a null pointer could be returned instead, as an option. Thus, **new** has been implemented differently, at different times, by compiler manufacturers. Although all compilers will eventually implement **new** in compliance with Standard C++, currently the only way to know the precise action of **new** on failure is to check your compiler's documentation.

Since Standard C++ specifies that **new** generates an exception on failure, this is the way the code in this book is written. If your compiler handles an allocation failure differently, you will need to make the appropriate changes.

Here is a program that allocates memory to hold an integer:

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

int main()
{
  int *p;

  try {
```

```
    p = new int; // allocate space for an int
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  *p = 100;

  cout << "At " << p << " ";
  cout << "is the value " << *p << "\n";

  delete p;

  return 0;
}
```

This program assigns to **p** an address in the heap that is large enough to hold an integer. It then assigns that memory the value 100 and displays the contents of the memory on the screen. Finally, it frees the dynamically allocated memory. Remember, if your compiler implements **new** such that it returns null on failure, you must change the preceding program appropriately.

The **delete** operator must be used only with a valid pointer previously allocated by using **new**. Using any other type of pointer with **delete** is undefined and will almost certainly cause serious problems, such as a system crash.

Although **new** and **delete** perform functions similar to **malloc( )** and **free( )**, they have several advantages. First, **new** automatically allocates enough memory to hold an object of the specified type. You do not need to use the **sizeof** operator. Because the size is computed automatically, it eliminates any possibility for error in this regard. Second, **new** automatically returns a pointer of the specified type. You don't need to use an explicit type cast as you do when allocating memory by using **malloc( )**. Finally, both **new** and **delete** can be overloaded, allowing you to create customized allocation systems.

Although there is no formal rule that states this, it is best not to mix **new** and **delete** with **malloc( )** and **free( )** in the same program. There is no guarantee that they are mutually compatible.

Initializing Allocated Memory

You can initialize allocated memory to some known value by putting an initializer after the type name in the **new** statement. Here is the general form of **new** when an initialization is included:

p_var = new *var_type (initializer);*

Of course, the type of the initializer must be compatible with the type of data for which memory is being allocated.

This program gives the allocated integer an initial value of 87:

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

int main()
{
  int *p;

  try {
    p = new int (87); // initialize to 87
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  cout << "At " << p << " ";
  cout << "is the value " << *p << "\n";

  delete p;

  return 0;
}
```

Allocating Arrays

You can allocate arrays using **new** by using this general form:

p_var = new *array_type [size];*

Here, *size* specifies the number of elements in the array.

To free an array, use this form of **delete:**

delete [ ] *p_var;*

Here, the [ ] informs **delete** that an array is being released.

For example, the next program allocates a 10-element integer array.

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

int main()
{
  int *p, i;

  try {
    p = new int [10]; // allocate 10 integer array
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  for(i=0; i<10; i++ )
    p[i] = i;

  for(i=0; i<10; i++)
    cout << p[i] << " ";

  delete [] p; // release the array

  return 0;
}
```

Notice the **delete** statement. As just mentioned, when an array allocated by **new** is released, **delete** must be made aware that an array is being freed by using the [ ]. (As you will see in the next section, this is especially important when you are allocating arrays of objects.)

One restriction applies to allocating arrays: They may not be given initial values. That is, you may not specify an initializer when allocating arrays.

Allocating Objects

You can allocate objects dynamically by using **new**. When you do this, an object is created and a pointer is returned to it. The dynamically created object acts just like any other object. When it is created, its constructor (if it has one) is called. When the object is freed, its destructor is executed.

Here is a short program that creates a class called **balance** that links a person's name with his or her account balance. Inside **main( )**, an object of type **balance** is created dynamically.

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

class balance {
  double cur_bal;
  char name[80];
public:
  void set(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }

  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;

  try {
    p = new balance;
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  p->set(12387.87, "Ralph Wilson");

  p->get_bal(n, s);

  cout << s << "'s balance is: " << n;
```

```
  cout << "\n";

  delete p;

  return 0;
}
```

Because **p** contains a pointer to an object, the arrow operator is used to access members of the object.

As stated, dynamically allocated objects may have constructors and destructors. Also, the constructors can be parameterized. Examine this version of the previous program:

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

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  ~balance() {
    cout << "Destructing ";
    cout << name << "\n";
  }
  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;
```

```
  // this version uses an initializer
  try {
    p = new balance (12387.87, "Ralph Wilson");
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  p->get_bal(n, s);

  cout << s << "'s balance is: " << n;
  cout << "\n";

  delete p;

  return 0;
}
```

Notice that the parameters to the object's constructor are specified after the type name, just as in other sorts of initializations.

You can allocate arrays of objects, but there is one catch. Since no array allocated by **new** can have an initializer, you must make sure that if the class contains constructors, one will be parameterless. If you don't, the C++ compiler will not find a matching constructor when you attempt to allocate the array and will not compile your program.

In this version of the preceding program, an array of **balance** objects is allocated, and the parameterless constructor is called.

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

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  balance() {} // parameterless constructor
```

```
  ~balance() {
    cout << "Destructing ";
    cout << name << "\n";
  }
  void set(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main()
{
  balance *p;
  char s[80];
  double n;
  int i;

  try {
    p = new balance [3]; // allocate entire array
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  // note use of dot, not arrow operators
  p[0].set(12387.87, "Ralph Wilson");
  p[1].set(144.00, "A. C. Conners");
  p[2].set(-11.23, "I. M. Overdrawn");

  for(i=0; i<3; i++) {
    p[i].get_bal(n, s);

    cout << s << "'s balance is: " << n;
    cout << "\n";
  }

  delete [] p;
  return 0;
}
```

The output from this program is shown here.

```
Ralph Wilson's balance is: 12387.9
A. C. Conners's balance is: 144
I. M. Overdrawn's balance is: -11.23
Destructing I. M. Overdrawn
Destructing A. C. Conners
Destructing Ralph Wilson
```

One reason that you need to use the **delete [ ]** form when deleting an array of dynamically allocated objects is so that the destructor can be called for each object in the array.

The nothrow Alternative

In Standard C++ it is possible to have **new** return **null** instead of throwing an exception when an allocation failure occurs. This form of **new** is most useful when you are compiling older code with a modern C++ compiler. It is also valuable when you are replacing calls to **malloc( )** with **new**. (This is common when updating C code to C++.) This form of **new** is shown here:

p_var = new(nothrow) *type*;

Here, *p_var* is a pointer variable of *type*. The **nothrow** form of **new** works like the original version of **new** from years ago. Since it returns null on failure, it can be "dropped into" older code without having to add exception handling. However, for new code, exceptions provide a better alternative. To use the **nothrow** option, you must include the header **<new>**.

The following program shows how to use the **new(nothrow)** alternative.

```
// Demonstrate nothrow version of new.
#include <iostream>
#include <new>
using namespace std;

int main()
{
  int *p, i;

  p = new(nothrow) int[32]; // use nothrow option
  if(!p) {
    cout << "Allocation failure.\n";
```

```
    return 1;
  }

  for(i=0; i<32; i++) p[i] = i;

  for(i=0; i<32; i++) cout << p[i] << " ";

  delete [] p; // free the memory

  return 0;
}
```

As this program demonstrates, when using the **nothrow** approach, you must check the pointer returned by **new** after each allocation request.

The Placement Form of new

There is a special form of **new**, called the *placement form*, that can be used to specify an alternative method of allocating memory. It is primarily useful when overloading the **new** operator for special circumstances. Here is its general form:

p_var = new (*arg-list*) *type*;

Here, *arg-list* is a comma-separated list of values passed to an overloaded form of **new**.