Pointers and Dynamic Arrays

Pointers and Dynamic Arrays

Memory is necessary for all the operations of reason.

BLAISE PASCAL, *PENSÉES*

Introduction

A *pointer* is a construct that gives you more control of the computer's memory. This chapter shows how pointers are used with arrays and introduces a new form of array called a *dynamic array.* **Dynamic arrays** are arrays whose size is determined while the program is running, rather than being fixed when the program is written.

Prerequisites

Section 12.1, which covers the basics of pointers, uses material from Chapters 2 through 7. It does not require any of the material from Chapters 8, 9, 10, or 11. In particular, it does not use arrays and uses only the most basic material on structs and classes.

Section 12.2, which covers dynamic arrays, uses material from Section 12.1, Chapters 2 through 7, and Chapter 10. It does not require any of the material from Chapters 8, 9, or 11. In particular, it uses only the most basic material on structs and classes.

Section 12.3, which covers the relationship between dynamic arrays and classes as well as some dynamic properties of classes, uses material from Sections 12.1 and 12.2 as well as from Chapters 2 to 10 and sections 11.1 and 11.2 of Chapter 11.

12.1 Pointers

Do not mistake the pointing finger for the moon.

ZEN SAYING

pointer

A **pointer** is the memory address of a variable. Recall that the computer's memory is divided into numbered memory locations (called bytes), and that variables are

implemented as a sequence of adjacent memory locations. Recall also that sometimes the C++ system uses these memory addresses as names for the variables. If a variable is implemented as, say, three memory locations, then the address of the first of these memory locations is sometimes used as a name for that variable. For example, when the variable is used as a call-by-reference argument, it is this address, not the identifier name of the variable, that is passed to the calling function.

An address that is used to name a variable in this way (by giving the address in memory where the variable starts) is called a *pointer* because the address can be thought of as "pointing" to the variable. The address "points" to the variable because it identifies the variable by telling *where* the variable is, rather than telling what the variable's name is. A variable that is, say, at location number 1007 can be pointed out by saying "it's the variable over there at location 1007."

You have already been using pointers in a number of situations. As we noted in the previous paragraph, when a variable is a call-by-reference argument in a function call, the function is given this argument variable in the form of a pointer to the variable. This is an important and powerful use for pointers, but it is done automatically for you by the C++ system. In this chapter we will show you how to write programs that manipulate pointers in any way you want, rather than relying on the system to manipulate the pointers for you.

Pointer Variables

A pointer can be stored in a variable. However, even though a pointer is a memory address and a memory address is a number, you cannot store a pointer in a variable of type `int` or `double`. A variable to hold a pointer must be declared to have a pointer type. For example, the following declares `p` to be a pointer variable that can hold one pointer that points to a variable of type `double`:

declaring pointer variables

```
double *p;
```

The variable `p` can hold pointers to variables of type `double`, but it cannot normally contain a pointer to a variable of some other type, such as `int` or `char`. Each variable type requires a different pointer type.

In general, to declare a variable that can hold pointers to other variables of a specific type, you declare the pointer variable just as you would declare an ordinary variable of that type, but you place an asterisk in front of the variable name. For example, the following declares the variables `p1` and `p2` so that they can hold pointers to variables of type `int`; it also declares two ordinary variables, `v1` and `v2`, of type `int`:

```
int *p1, *p2, v1, v2;
```

There must be an asterisk before *each* of the pointer variables. If you omit the second asterisk in the above declaration, then p2 will not be a pointer variable; it will instead be an ordinary variable of type *int*. The asterisk is the same symbol you have been using for multiplication, but in this context it has a totally different meaning.

Pointer Variable Declarations

A variable that can hold pointers to other variables of type *Type_Name* is declared similarly to the way you declare a variable of type *Type_Name*, except that you place an asterisk at the beginning of the variable name.

Syntax

```
Type_Name *Variable_Name1, *Variable_Name2, . . .;
```

Example

```
double *pointer1, *pointer2;
```

When discussing pointers and pointer variables, we usually speak of *pointing* rather than speaking of *addresses*. When a pointer variable, such as p1, contains the address of a variable, such as v1, the pointer variable is said to *point to the variable* v1 or to be *a pointer to the variable* v1.

Addresses and Numbers

A pointer is an address, and an address is an integer, but a pointer is not an integer. That is not crazy. That is abstraction! C++ insists that you use a pointer as an address and that you not use it as a number. A pointer is not a value of type *int* or of any other numeric type. You normally cannot store a pointer in a variable of type *int*. If you try, most C++ compilers will give you an error message or a warning message. Also, you cannot perform the normal arithmetic operations on pointers. (You can perform a kind of addition and a kind of subtraction on pointers, but they are not the usual integer addition and subtraction.)

the & operator

Pointer variables, like p1 and p2 declared above, can contain pointers to variables like v1 and v2. You can use the operator & to determine the address of a variable, and you can then assign that address to a pointer variable. For example, the following will set the variable p1 equal to a pointer that points to the variable v1:

```
p1 = &v1;
```

You now have two ways to refer to v1: You can call it v1 or you can call it "the variable pointed to by p1." In C++, the way that you say "the variable pointed to by p1" is *p1. This is the same asterisk that we used when we declared p1, but now it has yet another meaning. When the asterisk is used in this way, it is often called the **dereferencing operator,** and the pointer variable is said to be **dereferenced.**

the * operator

dereferencing

Putting these pieces together can produce some surprising results. Consider the following code:

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

This code will output the following to the screen:

```
42
42
```

As long as p1 contains a pointer that points to v1, then v1 and *p1 refer to the same variable. So when you set *p1 equal to 42, you are also setting v1 equal to 42.

The symbol & that is used to obtain the address of a variable is the same symbol that you use in function declarations to specify a call-by-reference parameter. This is not a coincidence. Recall that a call-by-reference argument is implemented by giving the address of the argument to the calling function. So, these two uses of the symbol & are very much the same. However, the usages are slightly different and we will consider them to be two different (although very closely related) usages of the symbol &.

pointers in assignment statements

You can assign the value of one pointer variable to another pointer variable. This copies an address from one pointer variable to another pointer variable. For example, if p1 is still pointing to v1, then the following will set p2 so that it also points to v1:

```
p2 = p1;
```

Provided we have not changed v1's value, the following will also output a 42 to the screen:

```
cout << *p2;
```

Be sure you do not confuse

```
p1 = p2;
```

and

```
*p1 = *p2;
```

The * and & Operators

The * operator in front of a pointer variable produces the variable it points to. When used this way, the * operator is called the **dereferencing operator.**

The operator & in front of an ordinary variable produces the address of that variable; that is, produces a pointer that points to the variable. The & operator is simply called the **address-of operator.**

For example, consider the declarations

```
double *p, v;
```

The following sets the value of p so that p points to the variable v:

```
p = &v;
```

*p produces the variable pointed to by p, so after the above assignment, *p and v refer to the same variable. For example, the following sets the value of v to 9.99, even though the name v is never explicitly used:

```
*p = 9.99;
```

When you add the asterisk, you are not dealing with the pointers p1 and p2, but with the variables that the pointers are pointing to. This is illustrated in Display 12.1.

new

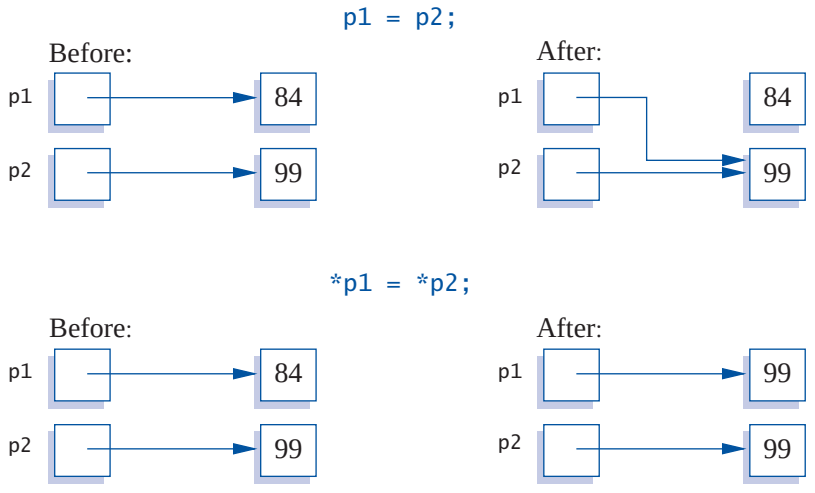
Since a pointer can be used to refer to a variable, your program can manipulate variables even if the variables have no identifiers to name them. The operator *new* can be used to create variables that have no identifiers to serve as their names. These nameless variables are referred to via pointers. For example, the following creates a new variable of type *int* and sets the pointer variable p1 equal to the address of this new variable (that is, p1 points to this new, nameless variable):

```
p1 = new int;
```

This new, nameless variable can be referred to as *p1 (that is, as the variable pointed to by p1). You can do anything with this nameless variable that you can do with any other variable of type *int*. For example, the following reads a value of type *int* from the keyboard into this nameless variable, adds 7 to the value, then outputs this new value:

```
cin >> *p1;
*p1 = *p1 + 7;
cout << *p1;
```

Display 12.1 Uses of the Assignment Operator



The *new* operator produces a new nameless variable and returns a pointer that points to this new variable. You specify the type for this new variable by writing the type name after the *new* operator. Variables that are created using the *new* operator are called **dynamic variables** because they are created and destroyed while the program is running. The program in Display 12.2 demonstrates some simple operations on pointers and dynamic variables. Display 12.3 illustrates the working of the program in Display 12.2. In Display 12.3 variables are represented as boxes and the value of the variable is written inside the box. We have not shown the actual numeric addresses in the pointer variables. The actual numbers are not important. What is important is that the number is the address of some particular variable. So, rather than use the actual number of the address, we have merely indicated the address with an arrow that points to the variable with that address. For example, in illustration (b) in Display 12.3, `p1` contains the address of a variable that has a question mark written in it.

dynamic variable

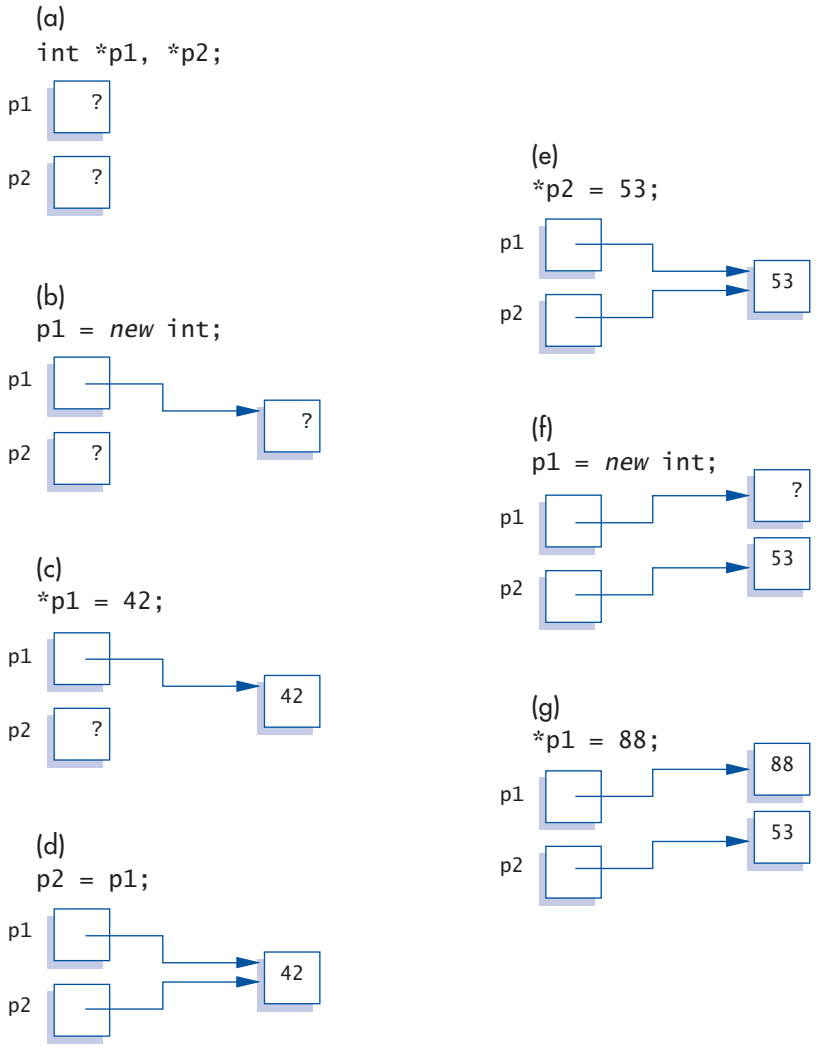
Display 12.2 Basic Pointer Manipulations


```
//Program to demonstrate pointers and dynamic variables.
#include <iostream>
using namespace std;

int main( )
{
    int *p1, *p2;

    p1 = new int;
    *p1 = 42;
    p2 = p1;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    *p2 = 53;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    p1 = new int;
    *p1 = 88;
    cout << "*p1 == " << *p1 << endl;
    cout << "*p2 == " << *p2 << endl;

    cout << "Hope you got the point of this example!\n";
    return 0;
}
```

Sample Dialogue

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

Display 12.3 Explanation of Display 12.2



Pointer Variables Used with =

If p1 and p2 are pointer variables, then the statement

```
p1 = p2;
```

will change p1 so that it points to the same thing that p2 is currently pointing to.

The *new* Operator

The *new* operator creates a new dynamic variable of a specified type and returns a pointer that points to this new variable. For example, the following creates a new dynamic variable of type MyType and leaves the pointer variable p pointing to this new variable:

```
MyType *p;
p = new MyType;
```

If the type is a class with a constructor, the default constructor is called for the newly created dynamic variable. Initializers can be specified that cause other constructors to be called:

```
int *n;
n = new int(17); // initializes n to 17
MyType *mtPtr;
mtPtr = new MyType(32.0, 17); // calls MyType(double, int);
```

The C++ standard specifies that if there is not sufficient memory available to create the new variable, then the *new* operator, by default, terminates the program.[1]

SELF-TEST EXERCISES

1 Explain the concept of a pointer in C++.

2 What unfortunate misinterpretation can occur with the following declaration?

```
int* int_ptr1, int_ptr2;
```

[1]Technically, the new operator throws an exception, which, if not caught, terminates the program. It is possible to "catch" the exception or install a new handler, but these topics are beyond the scope of this book.

3 Give at least two uses of the * operator. State what the * is doing, and name the use of the * that you present.

4 What is the output produced by the following code?

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
p1 = p2;
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

How would the output change if you were to replace

```
*p1 = 30;
```

with the following?

```
*p2 = 30;
```

5 What is the output produced by the following code?

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 10;
*p2 = 20;
cout << *p1 << " " << *p2 << endl;
*p1 = *p2; //This is different from Exercise 4
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

Basic Memory Management

A special area of memory, called the **freestore,**[2] is reserved for dynamic variables. Any new dynamic variable created by a program consumes some of the memory in

freestore

[2]The freestore is also sometimes called the *heap*.

the freestore. If your program creates too many dynamic variables, it will consume all of the memory in the freestore. If this happens, any additional calls to *new* will fail.

The size of the freestore varies from one implementation of C++ to another. It is typically large, and a modest program is not likely to use all the memory in the freestore. However, even on modest programs it is a good practice to recycle any freestore memory that is no longer needed. If your program no longer needs a dynamic variable, the memory used by that dynamic variable can be recycled. The

delete

delete operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore so that the memory can be reused. Suppose that p is a pointer variable that is pointing to a dynamic variable. The following will destroy the dynamic variable pointed to by p and return the memory used by the dynamic variable to the freestore:

```
delete p;
```

After the above call to *delete*, the value of p is undefined and p should be treated like an uninitialized variable.

The *delete* Operator

The *delete* operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore. The memory can then be reused to create new dynamic variables. For example, the following eliminates the dynamic variable pointed to by the pointer variable p:

```
delete p;
```

After a call to *delete*, the value of the pointer variable, like p above, is undefined. (A slightly different version of *delete*, discussed later in this chapter, is used when the dynamic variable is an array.)

PITFALL Dangling Pointers

When you apply *delete* to a pointer variable, the dynamic variable it is pointing to is destroyed. At that point, the value of the pointer variable is undefined, which means that you do not know where it is pointing, nor what the value is where it is pointing. Moreover, if some other pointer variable was pointing to the dynamic variable that was destroyed, then this other pointer variable is also undefined. These undefined pointer variables are called **dangling pointers.** If p is a dangling pointer

dangling pointer

and your program applies the dereferencing operator * to p (to produce the expression *p), the result is unpredictable and usually disastrous. Before you apply the dereferencing operator * to a pointer variable, you should be certain that the pointer variable points to some variable.

Static Variables and Automatic Variables

dynamic variables

Variables created with the *new* operator are called *dynamic variables* because they are created and destroyed while the program is running. When compared with these dynamic variables, ordinary variables seem static, but the terminology used by C++ programmers is a bit more involved than that, and ordinary variables are not called *static variables.*

The ordinary variables we have been using in previous chapters are not really static. If a variable is local to a function, then the variable is created by the C++ system when the function is called and is destroyed when the function call is completed. Since the `main` part of a program is really just a function called `main`, this is even true of the variables declared in the `main` part of your program. (Since the call to `main` does not end until the program ends, the variables declared in `main` are not destroyed until the program ends, but the mechanism for handling local variables is the same for `main` as it is for any other function.) The ordinary variables that we have been using (that is, the variables declared within `main` or within some other function definition) are called **automatic variables,** because their dynamic properties are controlled automatically for you; they are automatically created when the function in which they are declared is called and automatically destroyed when the function call ends. We will usually call these variables **ordinary variables,** but other books call them *automatic variables.*

automatic variables

global variables

There is one other category of variables, namely, **global variables.** Global variables are variables that are declared outside of any function definition (including being outside of main). We discussed global variables briefly in Chapter 3. As it turns out, we have no need for global variables and have not used them.

Programming TIP
Define Pointer Types

typedef

You can define a pointer type name so that pointer variables can be declared like other variables without the need to place an asterisk in front of each pointer variable.

For example, the following defines a type called `IntPtr`, which is the type for pointer variables that contain pointers to *`int`* variables:

```
typedef int* IntPtr;
```

Thus, the following two pointer variable declarations are equivalent:

```
IntPtr p;
```

and

```
int *p;
```

You can use *`typedef`* to define an alias for any type name or definition. For example, the following defines the type name `Kilometers` to mean the same thing as the type name *`double`*:

```
typedef double Kilometers;
```

Once you have given this type definition, you can define a variable of type *`double`* as follows:

```
Kilometers distance;
```

Renaming existing types this way can occasionally be useful. However, our main use of *`typedef`* will be to define types for pointer variables.

There are two advantages to using defined pointer type names, such as `IntPtr` defined above. First, it avoids the mistake of omitting an asterisk. Remember, if you intend `p1` and `p2` to be pointers, then the following is a mistake:

```
int *p1, p2;
```

Since the * was omitted from the `p2`, the variable `p2` is just an ordinary *`int`* variable, not a pointer variable. If you get confused and place the * on the *`int`*, the problem is the same but is more difficult to notice. C++ allows you to place the * on the type name, such as *`int`*, so that the following is legal:

```
int* p1, p2;
```

Although the above is legal, it is misleading. It looks like both `p1` and `p2` are pointer variables, but in fact only `p1` is a pointer variable; `p2` is an ordinary *`int`* variable. As far as the C++ compiler is concerned, the * that is attached to the identifier *`int`* may

as well be attached to the identifier p1. One correct way to declare both p1 and p2 to be pointer variables is

```
int *p1, *p2;
```

An easier and less error-prone way to declare both p1 and p2 to be pointer variables is to use the defined type name IntPtr as follows:

```
IntPtr p1, p2;
```

The second advantage of using a defined pointer type, such as IntPtr, is seen when you define a function with a call-by-reference parameter for a pointer variable. Without the defined pointer type name, you would need to include both an * and an & in the function declaration for the function, and the details can get confusing. If you use a type name for the pointer type, then a call-by-reference parameter for a pointer type involves no complications. You define a call-by-reference parameter for a defined pointer type just like you define any other call-by-reference parameter. Here's a sample:

```
void sample_function(IntPtr& pointer_variable);
```

Type Definitions

You can assign a name to a type definition and then use the type name to declare variables. This is done with the keyword *typedef*. These type definitions are normally placed outside of the body of the main part of your program (and outside the body of other functions) in the same place as *struct* and class definitions. We will use type definitions to define names for pointer types, as shown in the example below.

Syntax

```
typedef Known_Type_Definition  New_Type_Name;
```

Example

```
typedef int* IntPtr;
```

The type name IntPtr can then be used to declare pointers to dynamic variables of type *int*, as in the following:

```
IntPtr pointer1, pointer2;
```

SELF-TEST EXERCISES

6 Suppose a dynamic variable were created as follows:

```
char *p;
p = new char;
```

Assuming that the value of the pointer variable p has not changed (so it still points to the same dynamic variable), how can you destroy this new dynamic variable and return the memory it uses to the freestore so that the memory can be reused to create new dynamic variables?

7 Write a definition for a type called NumberPtr that will be the type for pointer variables that hold pointers to dynamic variables of type *int*. Also, write a declaration for a pointer variable called my_point that is of type NumberPtr.

8 Describe the action of the *new* operator. What does the operator *new* return?

12.2 Dynamic Arrays

dynamic array

In this section you will see that array variables are actually pointer variables. You will also find out how to write programs with dynamic arrays. A **dynamic array** is an array whose size is not specified when you write the program, but is determined while the program is running.

Array Variables and Pointer Variables

In Chapter 10 we described how arrays are kept in memory. At that point we had not learned about pointers, so we discussed arrays in terms of memory addresses. But, a memory address is a pointer. So, in C++ an array variable is actually a pointer variable that points to the first indexed variable of the array. Given the following two variable declarations, p and a are the same kind of variable:

```
int a[10];
typedef int* IntPtr;
IntPtr p;
```

The fact that a and p are the same kind of variable is illustrated in Display 12.4. Since a is a pointer that points to a variable of type *int* (namely the variable a[0]), the value of a can be assigned to the pointer variable p as follows:

```
p = a;
```

Display 12.4 Arrays and Pointer Variables


```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main( )
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;

    p = a;

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

    for (index = 0; index < 10; index++)
        p[index] = p[index] + 1;

    for (index = 0; index < 10; index++)
        cout << a[index] << " ";
    cout << endl;

    return 0;
}
```

Note that changes to the array p *are also changes to the array* a.

Output

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

After this assignment, `p` points to the same memory location that `a` points to. So, `p[0]`, `p[1]`, … `p[9]` refer to the indexed variables `a[0]`, `a[1]`, … `a[9]`. The square bracket notation you have been using for arrays applies to pointer variables as long as the pointer variable points to an array in memory. After the above assignment, you can treat the identifier `p` as if it were an array identifier. You can also treat the identifier `a` as if it were a pointer variable, but there is one important reservation. *You cannot change the pointer value in an array variable, such as* `a`. You might be tempted to think the following is legal, but it is not:

```
IntPtr p2;
...//p2 is given some pointer value.
a = p2;//ILLEGAL. You cannot assign a different address to a.
```

Creating and Using Dynamic Arrays

One problem with the kinds of arrays you have used thus far is that you must specify the size of the array when you write the program—but you may not know what size array you need until the program is run. For example, an array might hold a list of student identification numbers, but the size of the class may be different each time the program is run. With the kinds of arrays you have used thus far, you must estimate the largest possible size you may need for the array, and hope that size is large enough. There are two problems with this. First, you may estimate too low, and then your program will not work in all situations. Second, since the array might have many unused positions, this can waste computer memory. Dynamic arrays avoid these problems. If your program uses a dynamic array for student identification numbers, then the size of the class can be entered as input to the program and the dynamic array can be created to be exactly that size.

creating a dynamic array

Dynamic arrays are created using the *new* operator. The creation and use of dynamic arrays is surprisingly simple. Since array variables are pointer variables, you can use the *new* operator to create dynamic variables that are arrays and treat these dynamic array variables as if they were ordinary arrays. For example, the following creates a dynamic array variable with ten array elements of type *double*:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

To obtain a dynamic array of elements of any other type, simply replace *double* with the desired type. In particular, you can replace the type *double* with a *struct* or class type. To obtain a dynamic array variable of any other size, simply replace 10 with the desired size.

How to Use a Dynamic Array

- *Define a pointer type:* Define a type for pointers to variables of the same type as the elements of the array. For example, if the dynamic array is an array of *double*, you might use the following:

```
typedef double* DoubleArrayPtr;
```

- *Declare a pointer variable:* Declare a pointer variable of this defined type. The pointer variable will point to the dynamic array in memory and will serve as the name of the dynamic array.

```
DoubleArrayPtr a;
```

- *Call new:* Create a dynamic array using the *new* operator:

```
a = new double[array_size];
```

  The size of the dynamic array is given in square brackets as in the above example. The size can be given using an *int* variable or other *int* expression. In the above example, `array_size` can be a variable of type *int* whose value is determined while the program is running.

- *Use like an ordinary array:* The pointer variable, such as `a`, is used just like an ordinary array. For example, the indexed variables are written in the usual way: `a[0]`, `a[1]`, and so forth. The pointer variable should not have any other pointer value assigned to it, but should be used like an array variable.

- *Call delete []:* When your program is finished with the dynamic variable, use *delete* and empty square brackets along with the pointer variable to eliminate the dynamic array and return the storage that it occupies to the freestore for reuse. For example:

```
delete [] a;
```

There are also a number of less obvious things to notice about this example. First, the pointer type that you use for a pointer to a dynamic array is the same as the pointer type you would use for a single element of the array. For instance, the pointer type for an array of elements of type *double* is the same as the pointer type you would use for a simple variable of type *double*. The pointer to the array is actually a pointer to the first indexed variable of the array. In the above example, an entire array with ten indexed variables is created, and the pointer `p` is left pointing to the first of these ten indexed variables.

Also notice that when you call *new*, the size of the dynamic array is given in square brackets after the type, which in this example is the type *double*. This tells the computer how much storage to reserve for the dynamic array. If you omit the square brackets and the 10, the computer will allocate enough storage for only one variable of type *double*, rather than for an array of ten indexed variables of type *double*. As illustrated in Display 12.5, you can use an *int* variable in place of the constant 10 so that the size of the dynamic array can be read into the program.

The program in Display 12.5 sorts a list of numbers. This program works for lists of any size because it uses a dynamic array to hold the numbers. The size of the array is determined when the program is run. The user is asked how many numbers there will be, and then the *new* operator creates a dynamic array of that size. The size of the dynamic array is given by the variable `array_size`.

delete []

Notice the *delete* statement, which destroys the dynamic array variable `a` in Display 12.5. Since the program is about to end anyway, we did not really need this *delete* statement; however, if the program went on to do other things with dynamic variables, you would want such a *delete* statement so that the memory used by this dynamic array is returned to the freestore. The *delete* statement for a dynamic array is similar to the *delete* statement you saw earlier, except that with a dynamic array you must include an empty pair of square brackets like so:

```
delete [] a;
```

The square brackets tell C++ that a dynamic array variable is being eliminated, so the system checks the size of the array and removes that many indexed variables. If you omit the square brackets, you would be telling the computer to eliminate only one variable of type *int*. For example:

```
delete a;
```

is not legal, but the error is not detected by most compilers. The ANSI C++ standard says that what happens when you do this is "undefined." That means the author of the compiler can have this do anything that is convenient—convenient for the compiler writer, not for us. Even if it does something useful, you have no guarantee that either the next version of that compiler or any other compiler you compile this code with will do the same thing. The moral is simple: Always use the

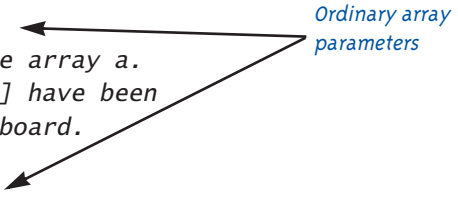
```
delete [] array_ptr;
```

syntax when you are deleting memory that was allocated with something like

```
array_ptr = new MyType[37];
```

Display 12.5 A Dynamic Array (*part 1 of 2*)


```
//Sorts a list of numbers entered at the keyboard.
#include <iostream>
#include <cstdlib>
#include <cstddef>

typedef int* IntArrayPtr;

void fill_array(int a[], int size);
//Precondition: size is the size of the array a.
//Postcondition: a[0] through a[size–1] have been
//filled with values read from the keyboard.

void sort(int a[], int size);
//Precondition: size is the size of the array a.
//The array elements a[0] through a[size–1] have values.
//Postcondition: The values of a[0] through a[size–1] have been rearranged
//so that a[0] <= a[1] <= ... <= a[size–1].

int main( )
{
    using namespace std;
    cout << "This program sorts numbers from lowest to highest.\n";

    int array_size;
    cout << "How many numbers will be sorted? ";
    cin >> array_size;

    IntArrayPtr a;
    a = new int[array_size];

    fill_array(a, array_size);
    sort(a, array_size);
```



Display 12.5 A Dynamic Array (*part 2 of 2*)

```
    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < array_size; index++)
        cout << a[index] << " ";
    cout << endl;

    delete [] a;

    return 0;
}

//Uses the library iostream:
void fill_array(int a[], int size)
{
    using namespace std;
    cout << "Enter " << size << " integers.\n";
    for (int index = 0; index < size; index++)
        cin >> a[index];
}

void sort(int a[], int size)
  <Any implementation of sort may be used. This may or may not require some
   additional function definitions. The implementation need not even know that
   sort will be called with a dynamic array. For example, you can use the
   implementation in Display 10.12 (with suitable adjustments to parameter
   names).>
```

The dynamic array a *is used like an ordinary array.*

You create a dynamic array with a call to *new* using a pointer, such as the pointer a in Display 12.5. After the call to *new*, you should not assign any other pointer value to this pointer variable, because that can confuse the system when the memory for the dynamic array is returned to the freestore with a call to *delete*.

Dynamic arrays are created using *new* and a pointer variable. When your program is finished using a dynamic array, you should return the array memory to the freestore with a call to *delete*. Other than that, a dynamic array can be used just like any other array.

SELF-TEST EXERCISES

9 Write a type definition for pointer variables that will be used to point to dynamic arrays. The array elements are to be of type *char*. Call the type `CharArray`.

10 Suppose your program contains code to create a dynamic array as follows:

```
int *entry;
entry = new int[10];
```

so that the pointer variable `entry` is pointing to this dynamic array. Write code to fill this array with ten numbers typed in at the keyboard.

11 Suppose your program contains code to create a dynamic array as in Self-Test Exercise 10, and suppose the pointer variable `entry` has not had its (pointer) value changed. Write code to destroy this new dynamic array and return the memory it uses to the freestore.

12 What is the output of the following code fragment? The code is assumed to be embedded in a correct and complete program.

```
int a[10];
int *p = a;
int i;
for (i = 0; i < 10; i++)
   a[i] = i;

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

13 What is the output of the following code fragment? The code is assumed to be embedded in a correct and complete program.

```
int array_size = 10;
int *a;
a = new int[array_size];
int *p = a;
int i;
for (i = 0; i < array_size; i++)
   a[i] = i;
p[0] = 10;
```

```
for (i = 0; i < array_size; i++)
   cout << a[i] << " ";
cout << endl;
```

Pointer Arithmetic (*Optional*)

There is a kind of arithmetic you can perform on pointers, but it is an arithmetic of addresses, not an arithmetic of numbers. For example, suppose your program contains the following code:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

addresses, not numbers

After these statements, `d` contains the address of the indexed variable `d[0]`. The expression `d + 1` evaluates to the address of `d[1]`, `d + 2` is the address of `d[2]`, and so forth. Notice that although the value of `d` is an address and an address is a number, `d + 1` does not simply add `1` to the number in `d`. If a variable of type *double* requires eight bytes (eight memory locations) and `d` contains the address `2001`, then `d + 1` evaluates to the memory address `2009`. Of course, the type *double* can be replaced by any other type, and then pointer addition moves in units of variables for that type.

This pointer arithmetic gives you an alternative way to manipulate arrays. For example, if `array_size` is the size of the dynamic array pointed to by `d`, then the following will output the contents of the dynamic array:

```
for (int i = 0; i < array_size; i++)
    cout << *(d + i)<< " ";
```

The above is equivalent to the following:

```
for (int i = 0; i < array_size; i++)
    cout << d[i] << " ";
```

You may not perform multiplication or division of pointers. All you can do is add an integer to a pointer, subtract an integer from a pointer, or subtract two pointers of the same type. When you subtract two pointers, the result is the number of indexed variables between the two addresses. Remember, for subtraction of two pointer values, these values must point into the same array! It makes little sense to subtract a pointer that points into one array from another pointer that points into a different array. You can use the increment and decrement operators `++` and `--`. For example, `d++` will advance the value of `d` so that it contains the address of the next indexed variable, and `d--` will change `d` so that it contains the address of the previous indexed variable.

`++` and `--`

SELF-TEST EXERCISES

These exercises apply to the optional section on pointer arithmetic.

14 What is the output of the following code fragment? The code is assumed to be embedded in a correct and complete program.

```
int array_size = 10;
int *a;
a = new int[array_size];
int i;
for (i = 0; i < array_size; i++)
   *(a + i) = i;

for (i = 0; i < array_size; i++)
   cout << a[i] << " ";
cout << endl;
```

15 What is the output of the following code fragment? The code is assumed to be embedded in a correct and complete program.

```
int array_size = 10;
int *a;
a = new int[array_size];
int i;
for (i = 0; i < array_size; i++)
   a[i] = i;

while (*a < 9)
{
   a++;
   cout << *a << " ";
}
cout << endl;
```

Multidimensional Dynamic Arrays (*Optional*)

You can have multidimensional dynamic arrays. You just need to remember that multidimensional arrays are arrays of arrays, or arrays of arrays of arrays, or so forth. For example, to create a two-dimensional dynamic array, you must remember that it is an array of arrays. To create a two-dimensional array of integers, you first create a one-dimensional dynamic array of pointers of type `int*`, which is the type

for a one-dimensional array of *int*s. Then you create a dynamic array of *int*s for each indexed variable of the array of pointers.

A type definition may help to keep things straight. The following is the variable type for an ordinary one-dimensional dynamic array of *int*s:

```
typedef int* IntArrayPtr;
```

To obtain a 3-by-4 array of *int*s, you want an array whose base type is `IntArrayPtr`. For example:

```
IntArrayPtr *m = new IntArrayPtr[3];
```

This is an array of three pointers, each of which can name a dynamic array of *int*s, as follows:

```
for (int i = 0; i < 3; i++)
    m[i] = new int[4];
```

The resulting array `m` is a 3-by-4 dynamic array. A simple program to illustrate this is given in Display 12.6.

delete []

Be sure to notice the use of *delete* in Display 12.6. Since the dynamic array m is an array of arrays, each of the arrays created with new in the *for* loop must be returned to the freestore manager with a call to *delete* []; then, the array m itself must be returned to the freestore with another call to *delete* []. There must be one call to *delete* [] for each call to new that created an array. (Since the program ends right after the calls to *delete* [], we could safely omit these calls, but we wanted to illustrate their usage.)

12.3 Classes and Dynamic Arrays

With all appliances and means to boot.

WILLIAM SHAKESPEARE, *KING HENRY IV, PART III*

A dynamic array can have a base type that is a class. A class can have a member variable that is a dynamic array. You can combine the techniques you learned about classes and the techniques you learned about dynamic arrays in just about any way. There are a few more things to worry about when using classes and dynamic arrays, but the basic techniques are the ones that you have already used. Let's start with an example.

Display 12.6 A Two-Dimensional Dynamic Array (*part 1 of 2*)


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

typedef int* IntArrayPtr;

int main( )
{
    int d1, d2;
    cout << "Enter the row and column dimensions of the array:\n";
    cin >> d1 >> d2;

    IntArrayPtr *m = new IntArrayPtr[d1];
    int i, j;
    for (i = 0; i < d1; i++)
        m[i] = new int[d2];
    //m is now a d1 by d2 array.

    cout << "Enter " << d1 << " rows of "
         << d2 << " integers each:\n";
    for (i = 0; i < d1; i++)
        for (j = 0; j < d2; j++)
            cin >> m[i][j];

    cout << "Echoing the two-dimensional array:\n";
    for (i = 0; i < d1; i++)
    {
        for (j = 0; j < d2; j++)
            cout << m[i][j] << " ";
        cout << endl;
    }
```

Display 12.6 A Two-Dimensional Dynamic Array (*part 2 of 2*)

```
    for (i = 0; i < d1; i++)
        delete[] m[i];
    delete[] m;

    return 0;
}
```

Note that there must be one call to `delete []` for each call to `new` that created an array. (These calls to `delete []` are not really needed since the program is ending, but in another context it could be important to include them.)

Sample Dialogue

```
Enter the row and column dimensions of the array:
3 4
Enter 3 rows of 4 integers each:
1 2 3 4
5 6 7 8
9 0 1 2
Echoing the two-dimensional array:
1 2 3 4
5 6 7 8
9 0 1 2
```

Programming EXAMPLE
A String Variable Class

In Chapter 11 we showed you how to define array variables to hold C strings. In the previous section you learned how to define dynamic arrays so that the size of the array can be determined when your program is run. In this example, we will define a class called `StringVar` whose objects are string variables. An object of the class `StringVar` will be implemented using a dynamic array whose size is determined when your program is run. So objects of type `StringVar` will have all the advantages of dynamic arrays, but they will also have some additional features. We will define `StringVar`'s member functions so that if you try to assign a string that is

too long to an object of type StringVar, you will get an error message. The version we define here provides only a small collection of operations for manipulating string objects. In Programming Project 1 you are asked to enhance the class definition by adding more member functions and overloaded operators.

Since you could use the standard class string, as discussed in Chapter 11, you do not really need the class StringVar, but it will be a good exercise to design and code it.

constructors

The interface for the type StringVar is given in Display 12.7. One constructor for the class StringVar takes a single argument of type *int*. This argument determines the maximum allowable length for a string value stored in the object. A default constructor creates an object with a maximum allowable length of 100. Another constructor takes an array argument that contains a C string of the kind discussed in Chapter 11. Note that this means the argument to this constructor can be a quoted string. This constructor initializes the object so that it can hold any string whose length is less than or equal to the length of its argument, and it initializes the object's string value to a copy of the value of its argument. For the moment, ignore the constructor that is labeled *Copy constructor*. Also ignore the member function named ~StringVar. Although it may look like a constructor, ~StringVar is not a constructor. We will discuss these two new kinds of member functions in later subsections. The meanings of the remaining member functions for the class StringVar are straightforward.

size of string value

A simple demonstration program is given in Display 12.8. Two objects, your_name and our_name, are declared within the definition of the function conversation. The object your_name can contain any string that is max_name_size or fewer characters long. The object our_name is initialized to the string value "Borg" and can have its value changed to any other string of length 4 or less.

implementation

As we indicated at the beginning of this subsection, the class StringVar is implemented using a dynamic array. The implementation is shown in Display 12.9. When an object of type StringVar is declared, a constructor is called to initialize the object. The constructor uses the *new* operator to create a new dynamic array of characters for the member variable value. The string value is stored in the array value as an ordinary string value, with '\0' used to mark the end of the string. Notice that the size of this array is not determined until the object is declared, at which point the constructor is called and the argument to the constructor determines the size of the dynamic array. As illustrated in Display 12.8, this argument can be a variable of type *int*. Look at the declaration of the object your_name in the definition of the function conversation. The argument to the constructor is the call-by-value parameter max_name_size. Recall that a call-by-value parameter is a local variable, so max_name_size is a variable. Any *int* variable may be used as the argument to the constructor in this way.

Display 12.7 Interface File for the `StringVar` Class (*part 1 of 2*)


```
//Header file strvar.h: This is the implementation for the class StringVar
//whose values are strings. An object is declared as follows.
//Note that you use (max_size), not [max_size]
//     StringVar the_object(max_size);
//where max_size is the longest string length allowed.
#ifndef STRVAR_H
#define STRVAR_H
#include <iostream>
using namespace std;
namespace strvarsavitch
{

    class StringVar
    {
    public:
        StringVar(int size);
        //Initializes the object so it can accept string values up to size
        //in length. Sets the value of the object equal to the empty string.

        StringVar( );
        //Initializes the object so it can accept string values of length 100
        //or less. Sets the value of the object equal to the empty string.

        StringVar(const char a[]);
        //Precondition: The array a contains characters terminated with '\0'.
        //Initializes the object so its value is the string stored in a and
        //so that it can later be set to string values up to strlen(a) in length

        StringVar(const StringVar& string_object);
        //Copy constructor.

        ~StringVar( );
        //Returns all the dynamic memory used by the object to the freestore.

        int length( ) const;
        //Returns the length of the current string value.
```

Display 12.7 Interface File for the `StringVar` Class (*part 2 of 2*)

```
        void input_line(istream& ins);
        //Precondition: If ins is a file input stream, then ins has been
        //connected to a file.
        //Action: The next text in the input stream ins, up to '\n', is copied
        //to the calling object. If there is not sufficient room, then
        //only as much as will fit is copied.

        friend ostream& operator <<(ostream& outs, const StringVar& the_string);
        //Overloads the << operator so it can be used to output values
        //of type StringVar
        //Precondition: If outs is a file output stream, then outs
        //has already been connected to a file.

    private:
        char *value; //pointer to dynamic array that holds the string value.
        int max_length; //declared max length of any string value.
    };
}//strvarsavitch

#endif //STRVAR_H
```

The implementation of the member functions `length`, `input_line`, and the overloaded output operator << are all straightforward. In the next few subsections we discuss the function `~StringVar` and the constructor labeled *Copy constructor*.

Destructors

There is one problem with dynamic variables. They do not go away unless your program makes a suitable call to *`delete`*. Even if the dynamic variable was created using a local pointer variable and the local pointer variable goes away at the end of a function call, the dynamic variable will remain unless there is a call to *`delete`*. If you do not eliminate dynamic variables with calls to *`delete`*, the dynamic variables will continue to occupy memory space, which may cause your program to abort because it used up all the memory in the freestore. Moreover, if the dynamic variable

Display 12.8 Program Using the `StringVar` Class


```
//Program to demonstrate use of the class StringVar.
#include <iostream>
#include "strvar.h"

void conversation(int max_name_size);
//Carries on a conversation with the user.

int main( )
{
    using namespace std;
    conversation(30);
    cout << "End of demonstration.\n";
    return 0;
}

// This is only a demonstration function:
void conversation(int max_name_size)
{
    using namespace std;
    using namespace strvarsavitch;

    StringVar your_name(max_name_size), our_name("Borg");

    cout << "What is your name?\n";
    your_name.input_line(cin);
    cout << "We are " << our_name << endl;
    cout << "We will meet again " << your_name << endl;
}
```

Memory is returned to the freestore when the function call ends.

Determines the size of the dynamic array

Sample Dialogue

```
What is your name?
Kathryn Janeway
We are Borg
We will meet again Kathryn Janeway
End of demonstration
```

Display 12.9 Implementation of `StringVar` (*part 1 of 2*)


```
//This is the implementation file: strvar.cpp
//(Your system may require some suffix other than .cpp.)
//This is the implementation of the class StringVar.
//The interface for the class StringVar is in the header file strvar.h.
#include <iostream>
#include <cstdlib>
#include <cstddef>
#include <cstring>
#include "strvar.h"
using namespace std;

namespace strvarsavitch
{
    //Uses cstddef and cstdlib:
    StringVar::StringVar(int size) : max_length(size)
    {
        value = new char[max_length + 1];//+1 is for '\0'.
        value[0] = '\0';
    }

    //Uses cstddef and cstdlib:
    StringVar::StringVar( ) : max_length(100)
    {
        value = new char[max_length + 1];//+1 is for '\0'.
        value[0] = '\0';
    }

    //Uses cstring, cstddef, and cstdlib:
    StringVar::StringVar(const char a[]) : max_length(strlen(a))
    {
        value = new char[max_length + 1];//+1 is for '\0'.
        strcpy(value, a);
    }
```

Display 12.9 Implementation of `StringVar` (*part 2 of 2*)

```
    //Uses cstring, cstddef, and cstdlib:
    StringVar::StringVar(const StringVar& string_object)
                            : max_length(string_object.length( ))
    {
        value = new char[max_length + 1];//+1 is for '\0'.
        strcpy(value, string_object.value);
    }

    StringVar::~StringVar( )
    {
        delete [] value;
    }

    //Uses cstring:
    int StringVar::length( ) const
    {
        return strlen(value);
    }

    //Uses iostream:
    void StringVar::input_line(istream& ins)
    {
        ins.getline(value, max_length + 1);
    }

    //Uses iostream:
    ostream& operator <<(ostream& outs, const StringVar& the_string)
    {
        outs << the_string.value;
        return outs;
    }

}//strvarsavitch
```

Copy constructor (discussed later in this chapter)

Destructor

is embedded in the implementation of a class, the programmer who uses the class does not know about the dynamic variable and cannot be expected to perform the call to `delete`. In fact, since the data members are normally private members, the programmer normally *cannot* access the needed pointer variables and so *cannot* call `delete` with these pointer variables. To handle this problem, C++ has a special kind of member function called a *destructor*.

destructor

A **destructor** is a member function that is called automatically when an object of the class passes out of scope. This means that if your program contains a local variable that is an object with a destructor, then when the function call ends, the destructor will be called automatically. If the destructor is defined correctly, the destructor will call `delete` to eliminate all the dynamic variables created by the object. This may be done with a single call to `delete` or it may require several calls to `delete`. You may also want your destructor to perform some other cleanup details as well, but returning memory to the freestore is the main job of the destructor.

destructor name

The member function `~StringVar` is the destructor for the class `StringVar` shown in Display 12.7. Like a constructor, a destructor always has the same name as the class it is a member of, but the destructor has the tilde symbol ~ at the beginning of its name (so you can tell that it is a destructor and not a constructor). Like a constructor, a destructor has no type for the value returned, not even the type `void`. A destructor has no parameters. Thus, a class can have only one destructor; you cannot overload the destructor for a class. Otherwise, a destructor is defined just like any other member function.

Destructor

A **destructor** is a member function of a class that is called automatically when an object of the class goes out of scope. Among other things, this means that if an object of the class type is a local variable for a function, then the destructor is automatically called as the last action before the function call ends. Destructors are used to eliminate any dynamic variables that have been created by the object so that the memory occupied by these dynamic variables is returned to the freestore. Destructors may perform other cleanup tasks as well. The name of a destructor must consist of the tilde symbol ~ followed by the name of the class.

~StringVar

Notice the definition of the destructor `~StringVar` given in Display 12.9. `~StringVar` calls `delete` to eliminate the dynamic array pointed to by the member pointer variable `value`. Look again at the function `conversation` in the sample program shown in Display 12.8. The local variables `your_name` and `our_name` both create dynamic arrays. If this class did not have a destructor, then after the call to

conversation has ended, these dynamic arrays would still be occupying memory, even though they are useless to the program. This would not be a problem here because the sample program ends soon after the call to conversation is completed; but if you wrote a program that made repeated calls to functions like conversation, and if the class StringVar did not have a suitable destructor, then the function calls could consume all the memory in the freestore and your program would then end abnormally.

PITFALL Pointers as Call-by-Value Parameters

When a call-by-value parameter is of a pointer type, its behavior can be subtle and troublesome. Consider the function call shown in Display 12.10. The parameter temp in the function sneaky is a call-by-value parameter, and hence it is a local variable. When the function is called, the value of temp is set to the value of the argument p and the function body is executed. Since temp is a local variable, no changes to temp should go outside of the function sneaky. In particular, the value of the pointer variable p should not be changed. Yet the sample dialogue makes it look as if the value of the pointer variable p had changed. Before the call to the function sneaky, the value of *p was 77, and after the call to sneaky the value of *p is 99. What has happened?

The situation is diagrammed in Display 12.11. Although the sample dialogue may make it look as if p were changed, the value of p was not changed by the function call to sneaky. Pointer p has two things associated with it: p's pointer value and the value stored where p points. Now, the value of p is a pointer (that is, a memory address). After the call to sneaky, the variable p contains the same pointer value (that is, the same memory address). The call to sneaky has changed the value of the variable pointed to by p, but it has not changed the value of p itself.

If the parameter type is a class or structure type that has member variables of a pointer type, the same kind of surprising changes can occur with call-by-value arguments of the class type. However, for class types, you can avoid (and control) these surprise changes by defining a *copy constructor,* as described in the next subsection.

Copy Constructors

A **copy constructor** is a constructor that has one parameter that is of the same type as the class. The one parameter must be a call-by-reference parameter, and normally the parameter is preceded by the *const* parameter modifier, so it is a constant parameter. In all other respects a copy constructor is defined in the same way as any other constructor and can be used just like other constructors.

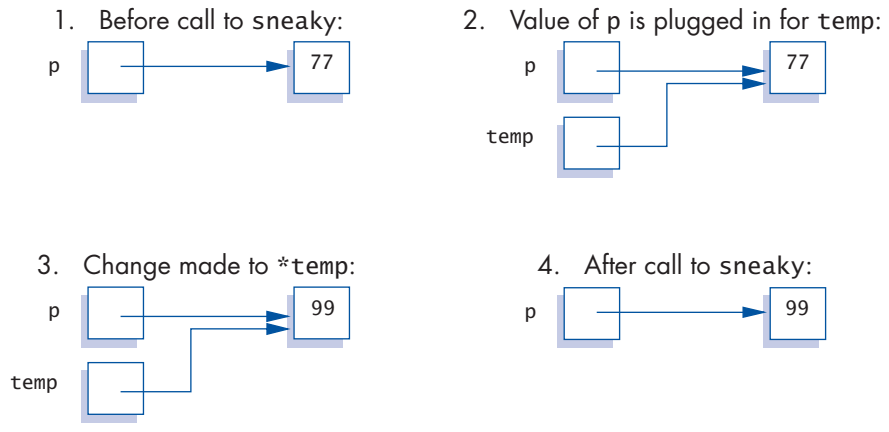
Display 12.10 A Call-by-Value Pointer Parameter


```
//Program to demonstrate the way call-by-value parameters
//behave with pointer arguments.
#include <iostream>
using namespace std;

typedef int* IntPointer;

void sneaky(IntPointer temp);

int main( )
{
    IntPointer p;

    p = new int;
    *p = 77;
    cout << "Before call to function *p == "
         << *p << endl;

    sneaky(p);

    cout << "After call to function *p == "
         << *p << endl;

    return 0;
}

void sneaky(IntPointer temp)
{
    *temp = 99;
    cout << "Inside function call *temp == "
         << *temp << endl;
}
```

Sample Dialogue

```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

Display 12.11 The Function Call `sneaky(p);`



called when an object is declared

For example, a program that uses the class `StringVar` defined in Display 12.7 might contain the following:

```
StringVar line(20), motto("Constructors can help.");
cout << "Enter a string of length 20 or less:\n";
line.input_line(cin);
StringVar temp(line);//Initialized by the copy constructor.
```

The constructor used to initialize each of the three objects of type `StringVar` is determined by the type of the argument given in parentheses after the object's name. The object `line` is initialized with the constructor that has a parameter of type *`int`*; the object `motto` is initialized by the constructor that has a parameter of type *`const char`* `a[]`. Similarly, the object `temp` is initialized by the constructor that has one argument of type *`const`* `StringVar&`. When used in this way a copy constructor is being used just like any other constructor.

A copy constructor should be defined so that the object being initialized becomes a complete, independent copy of its argument. So, in the declaration

```
StringVar temp(line);
```

The member variable `temp.value` is not simply set to the same value as `line.value`; that would produce two pointers pointing to the same dynamic array.

The definition of the copy constructor is shown in Display 12.9. Note that in the definition of the copy constructor, a new dynamic array is created and the contents of one dynamic array are copied to the other dynamic array. Thus, in the above declaration, `temp` is initialized so that its string value is equal to the string value of `line`, but `temp` has a separate dynamic array. Thus, any change that is made to `temp` will have no effect on `line`.

As you have seen, a copy constructor can be used just like any other constructor. A copy constructor is also called automatically in certain other situations. Roughly speaking, whenever C++ needs to make a copy of an object, it automatically calls the copy constructor. In particular, the copy constructor is called automatically in three circumstances: (1) when a class object is declared and is initialized by another object of the same type, (2) when a function returns a value of the class type, and (3) whenever an argument of the class type is "plugged in" for a call-by-value parameter. In this case, the copy constructor defines what is meant by "plugging in."

call-by-value parameters

To see why you need a copy constructor, let's see what would happen if we did not define a copy constructor for the class `StringVar`. *Suppose we did not include the copy constructor in the definition of the class* `StringVar` and suppose we used a call-by-value parameter in a function definition, for example:

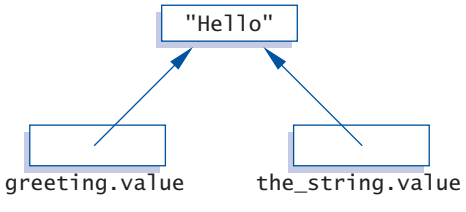
why a copy constructor is needed

```
void show_string(StringVar the_string)
{
    cout << "The string is: "
         << the_string << endl;
}
```

Consider the following code, which includes a function call:

```
StringVar greeting("Hello");
show_string(greeting);
cout << "After call: " << greeting << endl;
```

Assuming there is no copy constructor, things proceed as follows: When the function call is executed, the value of `greeting` is copied to the local variable `the_string`, so `the_string.value` is set equal to `greeting.value`. But these are pointer variables, so during the function call, `the_string.value` and `greeting.value` point to the same dynamic array, as follows:



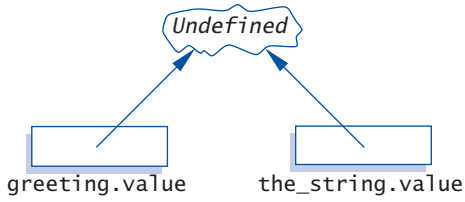
When the function call ends, the destructor for `StringVar` is called to return the memory used by `the_string` to the freestore. The definition of the destructor contains the following statement:

```
delete [] value;
```

Since the destructor is called with the object `the_string`, this statement is equivalent to:

```
delete [] the_string.value;
```

which changes the picture to the following:



Since `greeting.value` and `the_string.value` point to the same dynamic array, deleting `the_string.value` is the same as deleting `greeting.value`. Thus, `greeting.value` is undefined when the program reaches the statement

```
cout << "After call: " << greeting << endl;
```

This `cout` statement is therefore undefined. The `cout` statement may by chance give you the output you want, but sooner or later the fact that `greeting.value` is undefined will produce problems. One major problem occurs when the object `greeting` is a local variable in some function. In this case the destructor will be called with `greeting` when the function call ends. That destructor call will be equivalent to

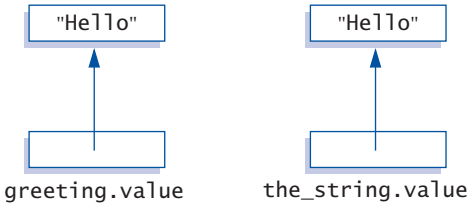
```
delete [] greeting.value;
```

But, as we just saw, the dynamic array pointed to by `greeting.value` has already been deleted once, and now the system is trying to delete it a second time. Calling *`delete`* twice to delete the same dynamic array (or other variable created with *`new`*) can produce a serious system error that can cause your program to crash.

That was what would happen if there were no copy constructor. Fortunately, we included a copy constructor in our definition of the class `StringVar`, so the copy constructor is called automatically when the following function call is executed:

```
StringVar greeting("Hello");
show_string(greeting);
```

The copy constructor defines what it means to "plug in" the argument `greeting` for the call-by-value parameter `the_string`, so that now the picture is as follows:



Thus, any change that is made to `the_string.value` has no effect on the argument `greeting`, and there are no problems with the destructor. If the destructor is called for `the_string` and then called for `greeting`, each call to the destructor deletes a different dynamic array.

returned value

When a function returns a value of a class type, the copy constructor is called automatically to copy the value specified by the return statement. If there is no copy constructor, then problems similar to what we described for value parameters will occur.

when you need a copy constructor

If a class definition involves pointers and dynamically allocated memory using the *new* operator, then you need to include a copy constructor. Classes that do not involve pointers or dynamically allocated memory do not need a copy constructor.

assignment statements

Contrary to what you might expect, the copy constructor is *not* called when you set one object equal to another using the assignment operator.[3] However, if you do not like what the default assignment operator does, you can redefine the assignment operator in the way described in the subsection entitled "Overloading the Assignment Operator."

[3]C++ makes a careful distinction between initialization (the three cases where the copy constructor is called) and assignment. Initialization uses the copy constructor to create a new object; the assignment operator takes an existing object and modifies it so that it is an identical copy (in all but location) of the right-hand side of the assignment.

Copy Constructor

A **copy constructor** is a constructor that has one call-by-reference parameter that is of the same type as the class. The one parameter must be a call-by-reference parameter; normally, the parameter is also a constant parameter, that is, preceded by the *const* parameter modifier. The copy constructor for a class is called automatically whenever a function returns a value of the class type. The copy constructor is also called automatically whenever an argument is "plugged in" for a call-by-value parameter of the class type. A copy constructor can also be used in the same ways as other constructors.

Any class that uses pointers and the *new* operator should have a copy constructor.

The Big Three

The **copy constructor,** the **= operator,** and the **destructor** are called the **big three** because experts say that if you need to define any of them, then you need to define all three. If any of these is missing, the compiler will create it, but it may not behave as you want. So it pays to define them yourself. The copy constructor and overloaded = operator that the compiler generates for you will work fine if all member variables are of predefined types such as *int* and *double*, but they may misbehave on classes that have class or pointer member variables. For any class that uses pointers and the *new* operator, it is safest to define your own copy constructor, overloaded =, and destructor.

SELF-TEST EXERCISES

16 If a class is named `MyClass` and it has a constructor, what is the constructor named? If `MyClass` has a destructor, what is the destructor named?

17 Suppose you change the definition of the destructor in Display 12.9 to the following. How would the sample dialogue in Display 12.8 change?

```
StringVar::~StringVar( )
{
    cout << endl
         << "Good-bye cruel world! The short life of\n"
         << "this dynamic array is about to end.\n";
    delete [] value;
}
```

18 The following is the first line of the copy constructor definition for the class `StringVar`. The identifier `StringVar` occurs three times and means something slightly different each time. What does it mean in each of the three cases?

```
StringVar::StringVar(const StringVar& string_object)
```

19 Answer these questions about destructors.

a. What is a destructor and what must the name of a destructor be?
b. When is a destructor called?
c. What does a destructor actually do?
d. What should a destructor do?

Overloading the Assignment Operator

Suppose `string1` and `string2` are declared as follows:

```
StringVar string1(10), string2(20);
```

The class `StringVar` was defined in Displays 12.7 and 12.9. If `string2` has somehow been given a value, then the following assignment statement is defined, but its meaning may not be what you would like it to be:

```
string1 = string2;
```

As usual, this predefined version of the assignment statement copies the value of each of the member variables of `string2` to the corresponding member variables of `string1`, so the value of `string1.max_length` is changed to be the same as `string2.max_length` and the value of `string1.value` is changed to be the same as `string2.value`. But this can cause problems with `string1` and probably even cause problems for `string2`.

The member variable `string1.value` contains a pointer, and the assignment statement sets this pointer equal to the same value as `string2.value`. Thus, both `string1.value` and `string2.value` point to the same place in memory. If you change the string value in `string1`, you will therefore also change the string value in `string2`. If you change the string value in `string2`, you will change the string value in `string1`.

In short, the predefined assignment statement does not do what we would like an assignment statement to do with objects of type `StringVar`. Using the predefined version of the assignment operator with the class `StringVar` can only cause problems. The way to fix this is to overload the assignment operator = so that it does what we want it to do with objects of the class `StringVar`.

= must be a member

The assignment operator cannot be overloaded in the way we have overloaded other operators, such as << and +. When you overload the assignment operator, it must be a member of the class; it cannot be a friend of the class. To add an overloaded version of the assignment operator to the class `StringVar`, the definition of `StringVar` should be changed to the following:

```
class StringVar
{
public:
    void operator =(const StringVar& right_side);
    //Overloads the assignment operator = to copy a string
    //from one object to another.
    <The rest of the definition of the class can be the same as in
    Display 12.7.>
```

The assignment operator is then used just as you always use the assignment operator. For example, consider the following:

```
string1 = string2;
```

calling object for =

In the above call, `string1` is the calling object and `string2` is the argument to the member operator =.

The definition of the assignment operator can be as follows:

```
//The following is acceptable, but
//we will give a better definition:
void StringVar::operator =(const StringVar& right_side)
{
    int new_length = strlen(right_side.value);
    if ((new_length) > max_length)
        new_length = max_length;

    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}
```

Notice that the length of the string in the object on the right side of the assignment operator is checked. If it is too long to fit in the object on the left side of the assignment operator (which is the calling object), then only as many characters as will fit are copied to the object receiving the string. But suppose you do not want to lose any characters in the copying process. To fit in all the characters, you can create a new, larger dynamic array for the object on the left-hand side of the assignment operator. You might try to redefine the assignment operator as follows:

```
//This version has a bug:
void StringVar::operator =(const StringVar& right_side)
{
    delete [] value;
    int new_length = strlen(right_side.value);
    max_length = new_length;

    value = new char[max_length + 1];

   for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}
```

This version has a problem when used in an assignment with the same object on both sides of the assignment operator, like the following:

```
my_string = my_string;
```

When this assignment is executed, the first statement executed is

```
delete [] value;
```

But the calling object is `my_string`, so this means

```
delete [] my_string.value;
```

So, the string value in `my_string` is deleted and the pointer `my_string.value` is undefined. The assignment operator has corrupted the object `my_string`, and this run of the program is probably ruined.

One way to fix this bug is to first check to see if there is sufficient room in the dynamic array member of the object on the left-hand side of the assignment operator and to only delete the array if extra space is needed. Our final definition of the overloaded assignment operator does just such a check:

```
//This is our final version:
void StringVar::operator =(const StringVar& right_side)
{
    int new_length = strlen(right_side.value);
    if (new_length > max_length)
    {
        delete [] value;
        max_length = new_length;
        value = new char[max_length + 1];
    }
```

```
    for (int i = 0; i < new_length; i++)
        value[i] = right_side.value[i];
    value[new_length] = '\0';
}
```

For many classes, the obvious definition for overloading the assignment operator does not work correctly when the same object is on both sides of the assignment operator. You should always check this case, and be careful to write your definition of the overloaded assignment operator so that it also works in this case.

SELF-TEST EXERCISE

20 a. Explain carefully why no overloaded assignment operator is needed when the only data consists of built-in types.

b. Same as part (a) for a copy constructor.

c. Same as part (a) for a destructor.

CHAPTER SUMMARY

- A **pointer** is a memory address, so a pointer provides a way to indirectly name a variable by naming the address of the variable in the computer's memory.
- **Dynamic variables** are variables that are created (and destroyed) while a program is running.
- Memory for dynamic variables is in a special portion of the computer's memory called the **freestore.** When a program is finished with a dynamic variable, the memory used by the dynamic variable can be returned to the freestore for reuse; this is done with a `delete` statement.
- A **dynamic array** is an array whose size is determined when the program is running. A dynamic array is implemented as a dynamic variable of an array type.
- A **destructor** is a special kind of member function for a class. A destructor is called automatically when an object of the class passes out of scope. The main reason for destructors is to return memory to the freestore so the memory can be reused.
- A **copy constructor** is a constructor that has a single argument that is of the same type as the class. If you define a copy constructor, it will be called

automatically whenever a function returns a value of the class type and whenever an argument is "plugged in" for a call-by-value parameter of the class type. Any class that uses pointers and the operator *new* should have a copy constructor.

- The assignment operator = can be overloaded for a class so that it behaves as you wish for that class. However, it must be overloaded as a member of the class. It cannot be overloaded as a friend. Any class that uses pointers and the operator *new* should overload the assignment operator for use with that class.

Answers to Self-Test Exercises

1 A pointer is the memory address of a variable.

2 To the unwary, or to the neophyte, this looks like two objects of type pointer to *int*, that is, *int**. Unfortunately, the * binds to the *identifier*, not to the type (that is, not to the *int*). The result is that this declaration declares int_ptr1 to be an *int* pointer, while int_ptr2 is just an ordinary *int* variable.

3
```
int *p;   //This declares a pointer variable that can
          //hold a pointer to an int variable.
*p = 17;  //Here, * is the dereference operator.
//This assigns 17 to the memory location pointed to by p.
```

4
```
10 20
20 20
30 30
```

If you replace `*p1 = 30;` with `*p2 = 30;`, the output would be the same.

5
```
10 20
20 20
30 20
```

6
```
delete p;
```

7
```
typedef int* NumberPtr;
NumberPtr my_point;
```

8 The *new* operator takes a type for its argument. *new* allocates space on the freestore of an appropriate size for a variable of the type of the argument. It returns a pointer to that memory (that is, a pointer to that new dynamic variable), provided there is enough available memory in the freestore. If there is not enough memory available in the freestore, your program ends.

9
```
typedef char* CharArray;
```

10
```
cout << "Enter 10 integers:\n";
for (int i = 0; i < 10; i++)
    cin >> entry[i];
```

11
```
delete [] entry;
```

12 0 1 2 3 4 5 6 7 8 9

13 10 1 2 3 4 5 6 7 8 9

14 0 1 2 3 4 5 6 7 8 9

15 1 2 3 4 5 6 7 8 9

16 The *constructor* is named `MyClass`, the same name as the name of the class. The *destructor* is named `~MyClass`.

17 The dialogue would change to the following:

```
What is your name?
Kathryn Janeway
We are Borg
We will meet again Kathryn Janeway

Good-bye cruel world! The short life of
this dynamic array is about to end.

Good-bye cruel world! The short life of
this dynamic array is about to end.
End of demonstration
```

18 The `StringVar` before the `::` is the name of the class. The `StringVar` right after the `::` is the name of the member function. (Remember, a constructor is a member function that has the same name as the class.) The `StringVar` inside the parentheses is the type for the parameter `string_object`.

19 a. A destructor is a member function of a class. A destructor's name always begins with a tilde, ~, followed by the class name.

b. A destructor is called when a class object goes out of scope.

c. A destructor actually does whatever the class author programs it to do!

d. A destructor is supposed to delete dynamic variables that have been allocated by constructors for the class. Destructors may also do other cleanup tasks.

20 In the case of the assignment operator = and the copy constructor, if there are only built-in types for data, the default copy mechanism is exactly what you want, so the default works fine. In the case of the destructor, no dynamic memory allocation is done (no pointers), so the default do-nothing action is again what you want.

Programming Projects

1 Enhance the definition of the class `StringVar` given in Display 12.7 and Display 12.9 by adding all of the following:

- Member function `copy_piece`, which returns a specified substring; member function `one_char`, which return a specified single character; and member function `set_char`, which changes a specified character.
- An overloaded version of the == operator (note that only the string values have to be equal; the values of `max_length` need not be the same).
- An overloaded version of + that performs concatenation of strings of type `StringVar`.
- An overloaded version of the extraction operator >> that reads one word (as opposed to `input_line`, which reads a whole line).

If you did the section on overloading the assignment operator, then add it as well. Also write a suitable test program and thoroughly test your class definition.

2 Do Programming Project 10 in Chapter 10 using a dynamic array. In this version of the class, one constructor should have a single argument of type `int` that specifies the maximum number of entries in the list.

3 Do Programming Project 9 in Chapter 10 using dynamic arrays. The program should ask the user the number of checks in each category and use this information to determine the sizes of the dynamic arrays.

a 

b 

4 In Chapter 11 we discussed vectors, which are like arrays that can grow in size. Suppose that vectors were not defined in C++. Define a class called `VectorDouble` that is like a class for a vector with base type `double`. Your class `VectorDouble` will have a private member variable for a dynamic array of `double`s. It will also have two member variables of type `int`; one called `max_count` for the size of the dynamic array of `double`s, and one called `count` for the number of array positions currently holding values. (`max_count` is the same as the capacity of a vector; `count` is the same as the size of a vector.)

If you attempt to add an element (a value of type `double`) to the vector object of the class `VectorDouble` and there is no more room, then a new dynamic array with twice the capacity of the old dynamic array is created and the values of the old dynamic array are copied to the new dynamic array.

Your class should have all of the following:

- Three constructors: a default constructor that creates a dynamic array for 50 elements, a constructor with one `int` argument for the number of elements in the initial dynamic array, and a copy constructor.
- A destructor.
- A suitable overloading of the assignment operator =.
- A suitable overloading of the equality operator ==. To be equal, the values of `count` and the `count` array elements must be equal, but the values of `max_count` need not be equal.
- Member functions `push_back`, `capacity`, `size`, `reserve`, and `resize` that behave the same as the member functions of the same names for vectors.
- Two member functions to give your class the same utility as the square brackets: `value_at(i)`, which returns the value of the `i`th element in the dynamic array; and `change_value_at(d, i)`, which changes the `double` value at the `i`th element of the dynamic array to `d`. Enforce suitable restrictions on the arguments to `value_at` and `change_value_at`. (Your class will not work with the square brackets. It can be made to work with square brackets, but we have not covered the material which tells you how to do that.)

5 Define a class called `Text` whose objects store lists of words. The class `Text` will be just like the class `StringVar` except that the class `Text` will use a dynamic array with base type `StringVar` rather than base type *`char`* and will mark the end of the array with a `StringVar` object consisting of a single blank, rather than using `'\0'` as the end marker. Intuitively, an object of the class `Text` represents some text consisting of words separated by blanks. Enforce the restriction that the array elements of type `StringVar` contain no blanks (except for the end marker elements of type `StringVar`).

Your class `Text` will have member functions corresponding to all the member functions of `StringVar`. The constructor with an argument of type *`const char`* `a[]` will initialize the `Text` object in the same was as described below for `input_line`. If the C-string argument contains the new-line symbol `'\n'`, that is considered an error and ends the program with an error message.

The member function `input_line` will read blank separated strings and store each string in one element of the dynamic array with base type `StringVar`. Multiple blank spaces are treated the same as a single blank space. When outputting an object of the class `Text`, insert one blank between each value of type `StringVar`. You may either assume that no tab symbols are used or you can treat the tab symbols the same as a blank; if this is a class assignment, ask your instructor how you should treat the tab symbol.

Add the enhancements described in Programming Project 1. The overloaded version of the extraction operator `>>` will fill only one element of the dynamic array.

6 Using dynamic arrays, implement a polynomial class with polynomial addition, subtraction, and multiplication.

a

b

Discussion: A variable in a polynomial does very little other than act as a placeholder for the coefficients. Hence, the only interesting thing about polynomials is the array of coefficients and the corresponding exponent. Think about the polynomial

```
x*x*x + x + 1
```

One simple way to implement the polynomial class is to use an array of *`double`*s to store the coefficients. The index of the array is the exponent of the corresponding term. Where is the term in `x*x` in the above example? If a term is missing, then it simply has a zero coefficient.

There are techniques for representing polynomials of high degree with many missing terms. These use so-called sparse polynomial techniques. Unless you already know these techniques, or learn very quickly, don't use these techniques.

Provide a default constructor, a copy constructor, and a parameterized constructor that enables an arbitrary polynomial to be constructed. Also supply an overloaded operator = and a destructor.

Provide these operations:

- polynomial + polynomial
- constant + polynomial
- polynomial + constant
- polynomial – polynomial
- constant – polynomial
- polynomial – constant.
- polynomial * polynomial
- constant * polynomial
- polynomial * constant

Supply functions to assign and extract coefficients, indexed by exponent.

Supply a function to evaluate the polynomial at a value of type *double*.

You should decide whether to implement these functions as members, friends, or standalone functions.