

10

Pointers and Dynamic Arrays

10.1 Pointers	412
Pointer Variables	413
Basic Memory Management	421
Pitfall: Dangling Pointers	424
Dynamic Variables and Automatic Variables	424
Tip: Define Pointer Types	425
Pitfall: Pointers as Call-by-Value Parameters	427
Uses for Pointers	429
10.2 Dynamic Arrays	430
Array Variables and Pointer Variables	430
Creating and Using Dynamic Arrays	431
Example: A Function That Returns an Array	435
Pointer Arithmetic	437
Multidimensional Dynamic Arrays	438
10.3 Classes, Pointers, and Dynamic Arrays	441
The <code>-></code> Operator	441
The <code>this</code> Pointer	442
Overloading the Assignment Operator	443
Example: A Class for Partially Filled Arrays	445
Destructors	454
Copy Constructors	455
Chapter Summary	459
Answers to Self-Test Exercises	460
Programming Projects	462



10 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 will show you how pointers are used with arrays and will introduce a new form of array called a *dynamically allocated array*. Dynamically allocated arrays (dynamic arrays for short) are arrays whose size is determined while the program is running, rather than being fixed when the program is written.

Before reading Sections 10.1 and 10.2 on pointers and dynamically allocated arrays you should first read Chapters 1 through 6 (omitting the coverage of vectors if you wish), but you need not read any of Chapters 7 through 9. You can even read Sections 10.1 and 10.2 after reading just Chapters 1 to 5, provided you ignore the few passages that mention classes.

Section 10.3 discusses some tools for classes that only become relevant once you begin to use pointers and dynamically allocated data (such as dynamically allocated arrays). Before covering Section 10.3, you should read Chapters 1 through 8, although you may omit the coverage of vectors if you wish.

You may cover this chapter, Chapter 11 on separate compilation and namespaces, Chapter 12 on file I/O, and Chapter 13 on recursion in any order. If you do not read the Chapter 11 section on namespaces before this chapter, you might find it profitable to review the section of Chapter 1 entitled **Namespaces**.

10.1 Pointers

By indirections find directions out.

WILLIAM SHAKESPEARE, *Hamlet*

pointer

A **pointer** is the memory address of a variable. Recall from Chapter 5 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.

You have already been using pointers in a number of situations. As 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. As noted in Chapter 5, an array is given to a function (or to anything else, for that matter) by giving a pointer to the first array element. (At the time we called these pointers “memory addresses,” but that is the same thing as a pointer.) These are two powerful uses for pointers, but they are handled automatically by the C++ system. This chapter shows you how to write programs that directly manipulate pointers rather than relying on the system to manipulate the pointers for you.

Pointer Variables

declaring 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`:

```
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.¹

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 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`.

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`.

¹ There are ways to get a pointer of one type into a pointer variable for another type, but it does not happen automatically and is very poor style anyway.

Pointer Variable Declarations

A variable that can hold pointers to other variables of type *Type_Name* is declared similar 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;
```

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. (As you will see later in this chapter, 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;
```

the * operator

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 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 called the **dereferencing operator**, and the pointer variable is said to be *dereferenced*.

dereferencing operator

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;
```

Pointer Types

There is a bit of an inconsistency (or at least a potential for confusion) in how C++ names pointer types. If you want a parameter whose type is, for example, a pointer to variables of type `int`, then the type is written `int*`, as in the following example:

```
void manipulatePointer(int* p);
```

If you want to declare a variable of the same pointer type, the `*` goes with the variable, as in the following example:

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

In fact, the compiler does not care whether the `*` is attached to the `int` or the variable name, so the following are also accepted by the compiler and have the same meanings:

```
void manipulatePointer(int *p); //Accepted but not as nice.
int* p1, *p2; //Accepted but dangerous.
```

However, we find the first versions to be clearer. In particular, note that when declaring variables there must be one `*` for each pointer variable.

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 closely related, although they are not exactly the same.

You can assign the value of 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 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 to which it points. 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, it 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 to which the pointers are pointing. This is illustrated in Display 10.1, in which 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 because the 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.

Pointer Variables Used with =

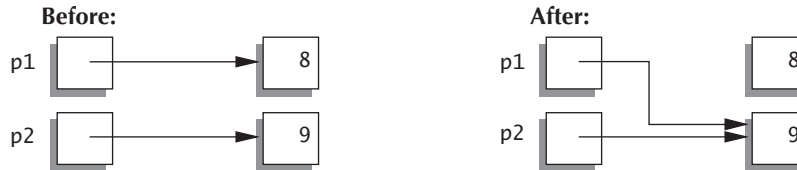
If `p1` and `p2` are pointer variables, then the statement

```
p1 = p2;
```

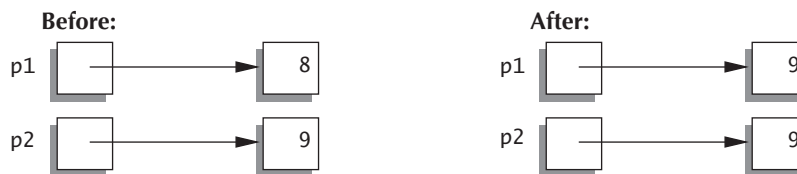
changes the value of `p1` so that it is the memory address (pointer) in `p2`. A common way to think of this is that the assignment will change `p1` so that it points to the same thing to which `p2` is currently pointing.

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```



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

**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 code reads a value of type `int` from the keyboard into this nameless variable, adds 7 to the value, and then outputs this new value:

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

**dynamic
variable**

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 **dynamically allocated variables** or simply **dynamic variables**, because they are created and destroyed while the program is running. The program in Display 10.2 demonstrates some simple operations on pointers and dynamic variables. Display 10.3 graphically illustrates the working of the program in Display 10.2.

Display 10.2 Basic Pointer Manipulations

```
1  //Program to demonstrate pointers and dynamic variables.
2  #include <iostream>
3  using std::cout;
4  using std::endl;

5  int main()
6  {
7      int *p1, *p2;

8      p1 = new int;
9      *p1 = 42;
10     p2 = p1;
11     cout << "*p1 == " << *p1 << endl;
12     cout << "*p2 == " << *p2 << endl;

13     *p2 = 53;
14     cout << "*p1 == " << *p1 << endl;
15     cout << "*p2 == " << *p2 << endl;

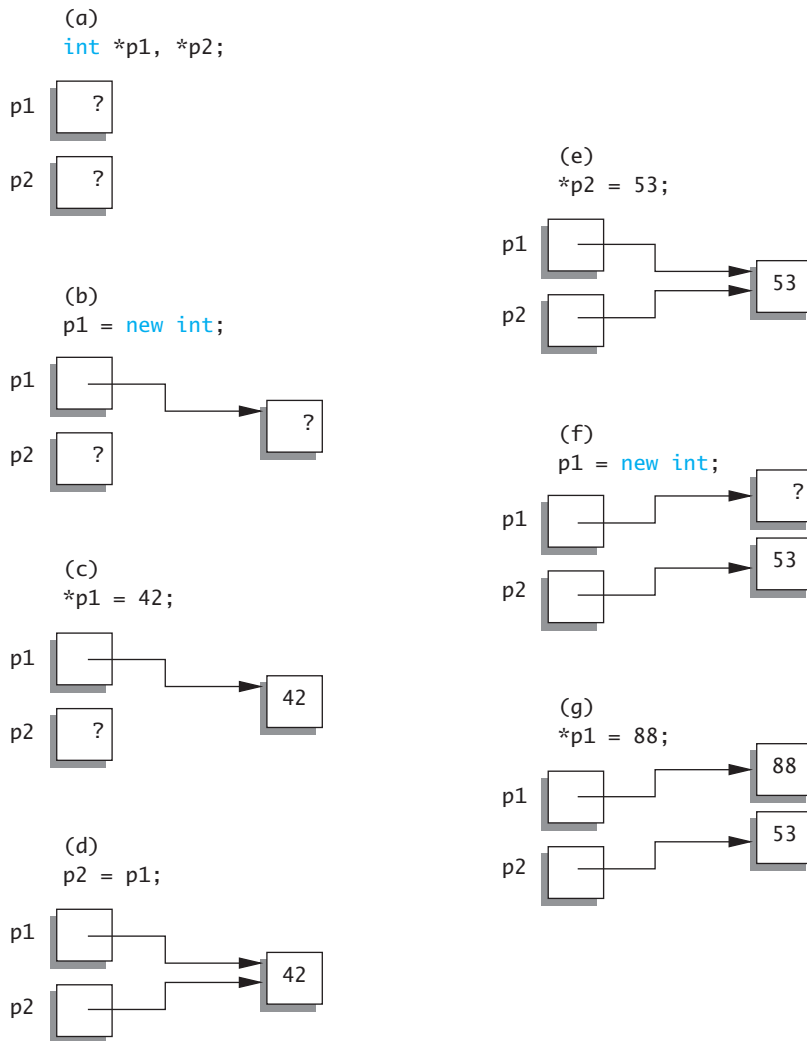
16     p1 = new int;
17     *p1 = 88;
18     cout << "*p1 == " << *p1 << endl;
19     cout << "*p2 == " << *p2 << endl;

20     cout << "Hope you got the point of this example!\n";
21     return 0;
22 }
```

Sample Dialogue

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

Display 10.3 Explanation of Display 10.2



When the `new` operator is used to create a dynamic variable of a class type, a constructor for the class is invoked. If you do not specify which constructor to use, the default constructor is invoked. For example, the following invokes the default constructor:

```
SomeClass *classPtr;  
classPtr = new SomeClass; //Calls default constructor.
```

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 type, the default constructor is called for the newly created dynamic variable. You can specify a different constructor by including arguments as follows:

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

A similar notation allows you to initialize dynamic variables of nonclass types, as illustrated below:

```
int *n;
n = new int(17); // initializes *n to 17
```

With earlier C++ compilers, if there was insufficient available memory to create the new variable, then `new` returned a special pointer named `NULL`. The C++ standard provides that if there is insufficient available memory to create the new variable, then the `new` operator, by default, terminates the program.

If you include constructor arguments, you can invoke a different constructor, as illustrated below:

```
classPtr = new SomeClass(32.0, 17); // Calls SomeClass(double, int).
```

A similar notation allows you to initialize dynamic variables of nonclass types, as illustrated below:

```
double *dPtr;
dPtr = new double(98.6); // Initializes *dPtr to 98.6.
```

pointer parameters

A pointer type is a full-fledged type and can be used in the same ways as other types. In particular, you can have a function parameter of a pointer type and you can have a function that returns a pointer type. For example, the following function has a parameter that is a pointer to an `int` variable and returns a (possibly different) pointer to an `int` variable:

```
int* findOtherPointer(int* p);
```

Self-Test Exercises

1. What is a pointer in C++?
2. Give at least three uses of the * operator. Name and describe each use.
3. 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;
```

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; //This is different from Exercise 3
cout << *p1 << " " << *p2 << endl;
*p1 = 30;
cout << *p1 << " " << *p2 << endl;
```

Basic Memory Management

freestore or heap

A special area of memory, called the **freestore** or the **heap**, is reserved for dynamically allocated variables. Any new dynamic variable created by a program consumes some of the memory in the freestore. If your program creates too many dynamic variables, it will consume all the memory in the freestore. If this happens, any additional calls to `new` will fail. What happens when you use `new` after you have exhausted all the memory

in the freestore (all the memory reserved for dynamically allocated variables) will depend on how up-to-date your compiler is. With earlier C++ compilers, if there was insufficient available memory to create the new variable, then `new` returned a special value named `NULL`. If you have a compiler that fully conforms to the newer C++ standard, then if there is insufficient available memory to create the new variable, the `new` operator terminates the program. Chapter 18 discusses ways to configure your program so that it can do things other than abort when `new` exhausts the freestore.²

If you have an older compiler, you can check to see if a call to `new` was successful by testing to see if `NULL` was returned by the call to `new`. For example, the following code tests to see if the attempt to create a new dynamic variable succeeded. The program will end with an error message if the call to `new` failed to create the desired dynamic variable:

```
int *p;
p = new int;
if (p == NULL)
{
    cout << "Error: Insufficient memory.\n";
    exit(1);
}
//If new succeeded, the program continues from here.
```

(Remember that since this code uses `exit`, you need an include directive for the library with header file `<cstdlib>` or, with some implementations, `<stdlib.h>`.)

NULL is 0

The constant `NULL` is actually the number `0`, but we prefer to think of it and spell it as `NULL` to make it clear that you mean this special-purpose value which you can assign to pointer variables. We will discuss other uses for `NULL` later in this book. The definition of the identifier `NULL` is in a number of the standard libraries, such as `<iostream>` and `<cstdlib>`, so you should use an include directive for either `<iostream>` or `<cstdlib>` (or another suitable library) when you use `NULL`.

As we said, `NULL` is actually just the number `0`. The definition of `NULL` is handled by the C++ preprocessor which replaces `NULL` with `0`. Thus, the compiler never actually sees “`NULL`” and so there is no namespace issue and no `using` directive is needed for `NULL`.³ While we prefer to use `NULL` rather than `0` in our code, we note that some authorities hold just the opposite view and advocate using `0` rather than `NULL`.

(Do not confuse the `NULL` pointer with the null character `'\0'` which is used to terminate C-strings. They are not the same. One is the integer `0` while the other is the character `'\0'`.)

²Technically, the `new` operator throws an exception, which, if not caught, terminates the program. It is possible to catch the exception and handle the exception. Exception handling is discussed in Chapter 18.

³The details are as follows: The definition of `NULL` uses `#define`, a form of definition that was inherited from the C language and that is handled by the preprocessor.

Newer compilers do not require the above explicit check to see if the new dynamic variable was created. On newer compilers, your program will automatically end with an error message if a call to `new` fails to create the desired dynamic variable. However, with any compiler, the above check will cause no harm and will make your program more portable.

NULL

NULL is a special constant pointer value that is used to give a value to a pointer variable that would not otherwise have a value. NULL can be assigned to a pointer variable of any type. The identifier NULL is defined in a number of libraries, including `<iostream>`. (The constant NULL is actually the integer 0.)

delete

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 in 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 returned to the **freestore manager** which recycles the memory to create other dynamic variables. The **delete** operator eliminates a dynamic variable and returns the memory that the dynamic variable occupied to the freestore manager 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 manager for reuse:

```
delete p;
```

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 dynamically allocated variable is an array.)



PITFALL: Dangling Pointers

dangling pointer

When you apply `delete` to a pointer variable, the dynamic variable to which it is pointing is destroyed. At that point, the value of the pointer variable is undefined, which means that you do not know 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 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.

C++ has no built-in test to check whether a pointer variable is a dangling pointer. One way to avoid dangling pointers is to set any dangling pointer variable equal to `NULL`. Then your program can test the pointer variable to see if it is equal to `NULL` before applying the dereferencing operator `*` to the pointer variable. When you use this technique, you follow a call to `delete` by code that sets all dangling pointers equal to `NULL`. Remember, other pointer variables may become dangling pointers besides the one pointer variable used in the call to `delete`, so be sure to set *all* dangling pointers to `NULL`. It is up to the programmer to keep track of dangling pointers and set them to `NULL` or otherwise ensure that they are not dereferenced. ■

Dynamic Variables and Automatic Variables

Variables created with the `new` operator are called **dynamic variables** (or **dynamically allocated variables**) because they are created and destroyed while the program is running. Local variables—that is, variables declared within a function definition—also have a certain dynamic characteristic, but they are not called dynamic variables. 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 for any other function.) These local variables are sometimes 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.

automatic variable

global variable

Variables declared outside any function or class definition, including outside `main`, are called **global variables**. These global variables are sometimes called **statically allocated variables**, because they are truly static in contrast to dynamic and automatic

variables. We discussed global variables briefly in Chapter 3. As it turns out, we have no need for global variables and have not used them.⁴



TIP: Define Pointer Types

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;
```

typedef 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.

Keep in mind that a `typedef` does not produce a new type but is simply an alias for the type definition. For example, given the previous definition of `Kilometers`, a variable of type `Kilometers` may be substituted for a parameter of type `double`. `Kilometers` and `double` are two names for the same type.

There are two advantages to using defined pointer type names, such as `IntPtr` defined previously. 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;
```

(continued)

⁴ Variables declared within a class using the modifier `static` are static in a different sense than the dynamic/static contrast we are discussing in this section.

**TIP** (continued)

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 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 an example:

```
void sampleFunction(IntPtr& pointerVariable); ■
```

Self-Test Exercises

5. What unfortunate misinterpretation can occur with the following declaration?

```
int* IntPtr1, IntPtr2;
```

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 manager so that the memory can be reused to create other new dynamic variables?

Self-Test Exercises (continued)

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 `double`. Also, write a declaration for a pointer variable called `myPoint`, which is of type `NumberPtr`.
8. Describe the action of the new operator. What does the new operator return? What are the indications of errors?

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 the body of the `main` part of your program and outside the body of other functions, typically near the start of a file. That way the `typedef` is global and available to your entire program. 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 example:

```
IntPtr pointer1, pointer2;
```



PITFALL: Pointers as Call-by-Value Parameters

When a call-by-value parameter is of a pointer type, its behavior can occasionally be subtle and troublesome. Consider the function call shown in Display 10.4. 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 the function `sneaky`. In particular, the value of the pointer variable `p` should not be changed. Yet the sample dialogue makes it look like the value of the pointer variable `p` has 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?

(continued)

Display 10.4 A Call-by-Value Pointer Parameter

```
1  //Program to demonstrate the way call-by-value parameters
2  //behave with pointer arguments.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;

7  typedef int* IntPtr;

8  void sneaky(IntPtr temp);

9  int main()
10 {
11     IntPtr p;

12     p = new int;
13     *p = 77;
14     cout << "Before call to function *p == "
15          << *p << endl;

16     sneaky(p);

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

19     return 0;
20 }
21 void sneaky(IntPtr temp)
22 {
23     *temp = 99;
24     cout << "Inside function call *temp == "
25          << *temp << endl;
26 }
```

Sample Dialogue

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



PITFALL (continued)

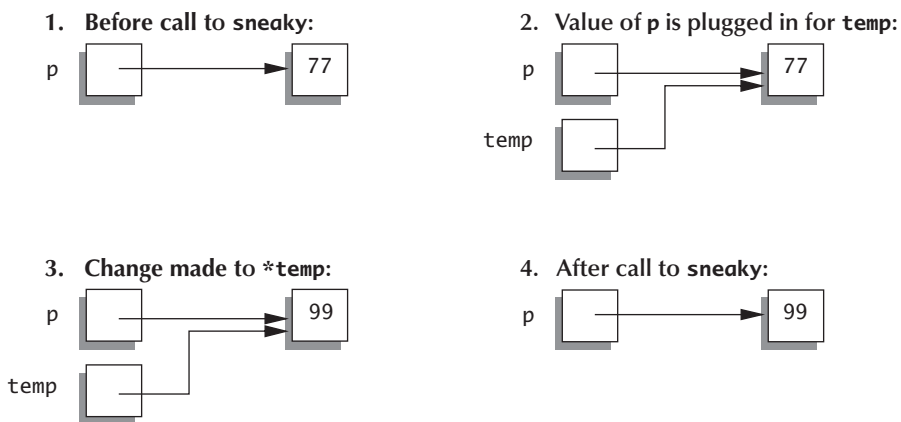
The situation is diagrammed in Display 10.5. Although the sample dialogue may make it look as if `p` were changed, the value of `p` was not changed by the function call to `sneaky`. Pointer `p` has two things associated with it: `p`'s pointer value and the value stored where `p` points. But the value of `p` is the 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 later in this chapter. ■

Uses for Pointers

Chapter 17 discusses ways to use pointers to create a number of useful data structures. This chapter only discusses one use of pointers, namely, to reference arrays, and in particular to create and reference a kind of array known as a *dynamically allocated array*. Dynamically allocated arrays are the topic of Section 10.2.

Display 10.5 The Function Call `sneaky(p)`;



dynamically
allocated array

10.2 Dynamic Arrays

In this section you will see that array variables are actually pointer variables. You will also find out how to write programs with dynamically allocated arrays. A **dynamically allocated array** (also called simply 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

Chapter 5 described how arrays are kept in memory. At that point we discussed arrays in terms of memory addresses. But a memory address is a pointer. So, in C++ an array variable is actually a kind of pointer variable that points to the first indexed variable of the array. Given the following two variable declarations, `p` and `a` are both pointer variables:

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

The fact that `a` and `p` are both pointer variables is illustrated in Display 10.6. 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;
```

After this assignment, `p` points to the same memory location that `a` points to. Thus, `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.* If the pointer variable `p2` has a value, you might be tempted to think the following is legal, but it is not:

```
a = p2; //ILLEGAL. You cannot assign a different address to a.
```

The underlying reason why this assignment does not work is that an array variable is not of type `int*`, but its type is a `const` version of `int*`. An array variable, like `a`, is a pointer variable with the modifier `const`, which means that its value cannot be changed.

(An array variable is actually more than an ordinary pointer variable since it carries some additional size information about the array, but an array variable does include a pointer to the array and the array variable can be assigned to a pointer variable. So, an array variable is a kind of pointer variable and can be treated like a pointer variable whose value cannot be changed.)

Display 10.6 Arrays and Pointer Variables

```

1  //Program to demonstrate that an array variable is a kind of pointer variable.
2  #include <iostream>
3  using std::cout;
4  using std::endl;

5  typedef int* IntPtr;

6  int main( )
7  {
8      IntPtr p;
9      int a[10];
10     int index;

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

13     p = a;

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

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

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

22     return 0;
23 }
```

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

Sample Dialogue

```

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

Creating and Using Dynamic Arrays

One problem with the kinds of arrays we discussed in Chapter 5 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

creating a
dynamic array

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. Dynamically allocated arrays avoid these problems. If your program uses a dynamically allocated array for student identification numbers, then the number of students can be entered as input to the program and the dynamically allocated array can be created to have a size exactly equal to the number of students.

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

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

To obtain a dynamically allocated 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 dynamically allocated array variable of any other size, simply replace `10` with the desired size.

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 dynamically allocated 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.

Second, notice that when you call `new`, the size of the dynamically allocated 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 omitted the square brackets and the `10` in this example, the computer would allocate enough storage for only one variable of type `double`, rather than for an array of ten indexed variables of type `double`.

Display 10.7 contains a program that illustrates the use of a dynamically allocated array. The program searches a list of numbers stored in a dynamically allocated array. 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 dynamically allocated array of that size. The size of the dynamic array is given by the variable `arraySize`. The size of a dynamic array need not be given by a constant. It can, as in Display 10.7, be given by a variable whose value is determined when the program is run.

Display 10.7 A Dynamically Allocated Array (part 1 of 2)

```

1  //Searches a list of numbers entered at the keyboard.
2  #include <iostream>
3  using std::cin;
4  using std::cout;

5  typedef int* IntPtr;

6  void fillArray(int a[], int size);
7  //Precondition: size is the size of the array a.
8  //Postcondition: a[0] through a[size-1] have been
9  //filled with values read from the keyboard.

10 int search(int a[], int size, int target);
11 //Precondition: size is the size of the array a.
12 //The array elements a[0] through a[size-1] have values.
13 //If target is in the array, returns the first index of target.
14 //If target is not in the array, returns -1.

15 int main( )
16 {
17     cout << "This program searches a list of numbers.\n";

18     int arraySize;
19     cout << "How many numbers will be on the list? ";
20     cin >> arraySize;
21     IntPtr a;
22     a = new int[arraySize];

23     fillArray(a, arraySize);

24     int target;
25     cout << "Enter a value to search for: ";
26     cin >> target;
27     int location = search(a, arraySize, target);
28     if (location == -1)
29         cout << target << " is not in the array.\n";
30     else
31         cout << target << " is element " << location << " in the array.\n";
32
33     delete [] a;
34
35     return 0;
36 }

37 //Uses the library <iostream>:
38 void fillArray(int a[], int size)

```

Ordinary array parameters

The dynamic array a is used like an ordinary array.

(continued)

Display 10.7 A Dynamically Allocated Array (part 2 of 2)

```

39  {
40      cout << "Enter " << size << " integers.\n";
41      for (int index = 0; index < size; index++)
42          cin >> a[index];
43  }

44  int search(int a[], int size, int target)
45  {
46      int index = 0;
47      while ((a[index] != target) && (index < size))
48          index++;
49      if (index == size)//if target is not in a.
50          index = -1;
51      return index;
52  }

```

Sample Dialogue

```

This program searches a list of numbers.
How many numbers will be on the list? 5
Enter 5 integers.
1 2 3 4 5
Enter a value to search for: 3
3 is element 2 in the array.

```

delete [] Notice the `delete` statement, which destroys the dynamically allocated array pointed to by `a` in Display 10.7. Since the program is about to end anyway, we did not really need this `delete` statement; if the program went on to do other things, however, you would want such a `delete` statement so that the memory used by this dynamically allocated array would be returned to the freestore manager. The `delete` statement for a dynamically allocated array is similar to the `delete` statement you saw earlier, except that with a dynamically allocated array you must include an empty pair of square brackets like so:

```
delete [] a;
```

The square brackets tell C++ that a dynamically allocated 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 will not be eliminating the entire array. For example:

```
delete a;
```

is not legal, but the error is not detected by most compilers. The 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 (for the compiler writer, not for you). Always use the

```
delete [] arrayPtr;
```

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

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

Also note the position of the square brackets in the delete statement

```
delete [] arrayPtr; //Correct  
delete arrayPtr[]; //ILLEGAL!
```

You create a dynamically allocated array with a call to new using a pointer, such as the pointer *a* in Display 10.7. 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 manager with a call to delete.

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

EXAMPLE: A Function That Returns an Array

In C++ an array type is not allowed as the return type of a function. For example, the following is illegal:

```
int [] someFunction( ); //ILLEGAL
```

If you want to create a function similar to this, you must return a pointer to the array base type and have the pointer point to the array. So, the function declaration would be as follows:

```
int* someFunction( ); //Legal
```

An example of a function that returns a pointer to an array is given in Display 10.8.

Display 10.8 Returning a Pointer to an Array

```

1  #include <iostream>
2  using std::cout;
3  using std::endl;

4  int* doubler(int a[], int size);
5  //Precondition; size is the size of the array a.
6  //All indexed variables of a have values.
7  //Returns: a pointer to an array of the same size as a in which
8  //each indexed variable is double the corresponding element in a.

9  int main( )
10 {
11     int a[] = {1, 2, 3, 4, 5};
12     int* b;

13     b = doubler(a, 5);

14     int i;
15     cout << "Array a:\n";
16     for (i = 0; i < 5; i++)
17         cout << a[i] << " ";
18     cout << endl;
19     cout << "Array b:\n";
20     for (i = 0; i < 5; i++)
21         cout << b[i] << " ";
22     cout << endl;

23     delete[] b;
24     return 0;
25 }

26 int* doubler(int a[], int size)
27 {
28     int* temp = new int[size];

29     for (int i=0; i < size; i++)
30         temp[i] = 2*a[i];

31     return temp;
32 }

```

This call to delete is not really needed since the program is ending, but in another context it could be important to include this delete.

Sample Dialogue

```

Array a:
1 2 3 4 5
Array b:
2 4 6 8 10

```

Self-Test Exercises

9. Write a type definition for pointer variables that will be used to point to dynamically allocated arrays. The array elements are to be of type `char`. Call the type `CharArray`.
10. Suppose your program contains code to create a dynamically allocated array as follows:

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

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

11. Suppose your program contains code to create a dynamically allocated 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 dynamically allocated array and return the memory it uses to the freestore manager.
12. What is the output of the following code fragment?

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

Pointer Arithmetic

You can perform a kind of arithmetic 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 2000, then `d + 1` evaluates

to the memory address 2008. 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 `arraySize` is the size of the dynamically allocated array pointed to by `d`, then the following will output the contents of the dynamic array:

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

The above is equivalent to the following:

```
for (int i = 0; i < arraySize; 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.

++ and --

You can also use the increment and decrement operators, `++` and `--`, to perform pointer arithmetic. 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.

Multidimensional Dynamic Arrays

You can have multidimensional dynamic arrays. You just need to remember that multidimensional arrays are arrays of arrays or arrays of arrays of arrays and 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 ints. Then you create a dynamic array of ints for each element of the array.

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

```
typedef int* IntArrayPtr;
```

To obtain a three-by-four array of ints, 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 ints, as follows:

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

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 `doubles`, 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 dynamically allocated array in memory and will serve as the name of the dynamic array.

```
DoubleArrayPtr a;
```

(Alternatively, without a defined pointer type, use `double *a;`).

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

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

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, `arraySize` 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 dynamically allocated array 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 manager for reuse. For example:

```
delete [] a;
```

The resulting array `m` is a three-by-four dynamic array. A simple program to illustrate this is given in Display 10.9.

delete []

Be sure to notice the use of `delete` in Display 10.9. Since the dynamic array `m` is an array of arrays, each of the arrays created with `new` in the `for` loop on lines 13 and 14 must be returned to the freestore manager with a call to `delete []`; then, the array `m` itself must be returned to the freestore manager with another call to `delete []`. There must be a 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 the calls to `delete []`, but we wanted to illustrate their usage.)

Display 10.9 A Two-Dimensional Dynamic Array (part 1 of 2)

```

1  #include <iostream>
2  using std::cin;
3  using std::cout;
4  using std::endl;

5  typedef int* IntArrayPtr;

6  int main( )
7  {
8      int d1, d2;
9      cout << "Enter the row and column dimensions of the array:\n";
10     cin >> d1 >> d2;

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

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

21     cout << "Echoing the two-dimensional array:\n";
22     for (i = 0; i < d1; i++)
23     {
24         for (j = 0; j < d2; j++)
25             cout << m[i][j] << " ";
26         cout << endl;
27     }

28
29     for (i = 0; i < d1; i++)
30         delete[] m[i];
31     delete[] m;

32     return 0;
33 }
```

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.)

Display 10.9 A Two-Dimensional Dynamic Array (part 2 of 2)

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
```

10.3 Classes, Pointers, and Dynamic Arrays

The combinations are endless.

Common advertisement copy

A dynamically allocated array can have a base type that is a class. A class can have a member variable that is a dynamically allocated array. You can combine classes and dynamically allocated arrays in just about any combination. There are a few more things to worry about when using classes and dynamically allocated arrays, but the basic techniques are the ones that you have already used. Many of the techniques presented in this section apply to all dynamically allocated structures, such as those we will discuss in Chapter 17, and not just to classes involving dynamically allocated arrays.

The $\rightarrow$ Operator

arrow operator

C++ has an operator that can be used with a pointer to simplify the notation for specifying the members of a struct or a class. The **arrow operator**, $\rightarrow$, combines the actions of a dereferencing operator, $*$, and a dot operator to specify a member of a dynamic struct or class object that is pointed to by a given pointer. For example, suppose you have the following definition:

```
struct Record
{
    int number;
    char grade;
};
```

The following creates a dynamically allocated variable of type `Record` and sets the member variables of the dynamic struct variable to 2001 and 'A':

```
Record *p;
p = new Record;
p->number = 2001;
p->grade = 'A';
```

The notations

`p->grade`

and

`(*p).grade`

have the same meaning. However, the first is more convenient and is almost always the notation used.

The `this` Pointer

When defining member functions for a class, you sometimes want to refer to the calling object. The `this` pointer is a predefined pointer that points to the calling object. For example, consider a class like the following:

```
class Sample
{
public:
    ...
    void showStuff( ) const;
    ...
private:
    int stuff;
    ...
};
```

The following two ways of defining the member function `showStuff` are equivalent:

```
void Sample::showStuff( ) const
{
    cout << stuff;
}

//Not good style, but this illustrates the this pointer:
void Sample::showStuff( )
{
    cout << this->stuff;
}
```

Notice that `this` is not the name of the calling object, but is the name of a pointer that points to the calling object. The `this` pointer cannot have its value changed; it always points to the calling object.

As our earlier comment indicated, you normally have no need for the pointer `this`. However, in a few situations it is handy. One place where the `this` pointer is commonly used is in overloading the assignment operator, `=`, which we discuss next.

Since the `this` pointer points to the calling object, you cannot use `this` in the definition of any static member functions. A static member function normally has no calling object to which the pointer `this` can point.

Overloading the Assignment Operator

In this book we usually use the assignment operator as if it were a void function. However, the predefined assignment operator returns a reference that allows for some specialized uses.

With the predefined assignment operator, you can chain assignment operators as follows: `a = b = c;`, which means `a = (b = c);`. The first operation, `b = c`, returns the new version of `b`. So, the action of

```
a = b = c;
```

is to set `a` as well as `b` equal to `c`. To ensure that your overloaded versions of the assignment operator can be used in this way, you need to define the assignment operator so it returns something of the same type as its left-hand side. As you will see shortly, the `this` pointer will allow you to do this. (No pun intended.) However, while this requires that the assignment operator return something of the type of its left-hand side, it does not require that it return a reference. Another use of the assignment operator explains why a reference is returned.

The reason that the predefined assignment operator returns a reference is so that you can invoke a member function with the value returned, as in

```
(a = b).f( );
```

where `f` is a member function. If you want your overloaded versions of the assignment operator to allow for invoking member functions in this way, then you should have them return a reference. This is not a very compelling reason for returning a reference, since this is a pretty minor property that is seldom used. However, it is traditional to return a reference, and it is not significantly more difficult to return a reference than to simply return a value.

For example, consider the following class (which might be used for some specialized string handling that is not easily handled by the predefined class `string`):

```
class StringClass
{
public:
    ...
    void someProcessing( );
    ...
    StringClass& operator=(const StringClass& rtSide);
    ...
}
```

```
private:
    char *a; //Dynamic array for characters in the string
    int capacity; //size of dynamic array a
    int length; //Number of characters in a
};
```

= must be a member

As noted in Chapter 8, when you overload the assignment operator it must be a member of the class; it cannot be a friend of the class. That is why the above definition has only one parameter for operator. For example, consider the following:

```
s1 = s2; //s1 and s2 in the class StringClass
```

calling object for =

In the above call, s1 is the calling object and s2 is the argument to the member operator =. The following definition of the overloaded assignment operator can be used in chains of assignments like

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

and can be used to invoke member functions as follows:

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

The definition of the overloaded assignment operator uses the this pointer to return the object on the left side of the = sign (which is the calling object):

```
//This version does not work in all cases.
StringClass& StringClass::operator=(const StringClass& rtSide)
{
    capacity = rtSide.capacity;
    length = rtSide.length;
    delete [] a;
    a = new char[capacity];
    for (int i = 0; i < length; i++)
        a[i] = rtSide.a[i];

    return *this;
}
```

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

```
s = s;
```

When this assignment is executed, the following statement is executed:

```
delete [] a;
```

But the calling object is s, so this means

```
delete [] s.a;
```

The pointer `s.a` is then undefined. The assignment operator has corrupted the object `s` and this run of the program is probably ruined.

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.

To avoid the problem we had with our first definition of the overloaded assignment operator, you can use the `this` pointer to test this special case as follows:

```
//Final version with bug fixed:
StringClass& StringClass::operator=(const StringClass& rtSide)
{
    if (this == &rtSide)
        //if the right side is the same as the left side
        {
            return *this;
        }
    else
    {
        capacity = rtSide.capacity;
        length = rtSide.length;
        delete [] a;
        a = new char[capacity];
        for (int i = 0; i < length; i++)
            a[i] = rtSide.a[i];

        return *this;
    }
}
```

A complete example with an overloaded assignment operator is given in the next programming example.

EXAMPLE: A Class for Partially Filled Arrays

The class `PFArrayD` in Displays 10.10 and 10.11 is a class for a partially filled array of doubles.⁵ As shown in the demonstration program in Display 10.12, an object of the class `PFArrayD` can be accessed using the square brackets just like an ordinary array, but the object also automatically keeps track of how much of the array is in use. Thus, it

(continued)

⁵ If you have already read the section of Chapter 7 on vectors, you will notice that the class defined here is a weak version of a vector. Even though you can use a vector any place that you would use this class, this is still an instructive example using many of the techniques we discussed in this chapter. Moreover, this example will give you some insight into how a vector class might be implemented.

Display 10.10 Definition of a Class with a Dynamic Array Member

```

1
2 //Objects of this class are partially filled arrays of doubles.
3 class PFArrayD
4 {
5 public:
6     PFArrayD( );
7     //Initializes with a capacity of 50.
8
9     PFArrayD(int capacityValue);
10
11     PFArrayD(const PFArrayD& pfaObject); ← Copy constructor
12
13     void addElement(double element);
14     //Precondition: The array is not full.
15     //Postcondition: The element has been added.
16
17     bool full( ) const { return (capacity == used); }
18     //Returns true if the array is full, false otherwise.
19
20     int getCapacity( ) const { return capacity; }
21
22     int getNumberUsed( ) const { return used; }
23
24     void emptyArray( ){ used = 0; }
25     //Empties the array.
26
27     double& operator[](int index);
28     //Read and change access to elements 0 through numberUsed - 1. Overloaded
29
30     PFArrayD& operator =(const PFArrayD& rightSide); ← assignment
31
32     ~PFArrayD( ); ← Destructor
33 private:
34     double *a; //For an array of doubles
35     int capacity; //For the size of the array
36     int used; //For the number of array positions currently in use
37 };

```

Display 10.11 Member Function Definitions for PFArrayD Class (part 1 of 2)

```
1
2 //These are the definitions for the member functions for the class PFArrayD.
3 //They require the following include and using directives:
4 //#include <iostream>
5 //using std::cout;

6 PFArrayD::PFArrayD( ) :capacity(50), used(0)
7 {
8     a = new double[capacity];
9 }

10 PFArrayD::PFArrayD(int size) :capacity(size), used(0)
11 {
12     a = new double[capacity];
13 }

14 PFArrayD::PFArrayD(const PFArrayD& pfaObject)
15     :capacity(pfaObject.getCapacity( )), used(pfaObject.getNumberUsed( ))
16 {
17     a = new double[capacity];
18     for (int i =0; i < used; i++)
19         a[i] = pfaObject.a[i];
20 }

21 void PFArrayD::addElement(double element)
22 {
23     if (used >= capacity)
24     {
25         cout << "Attempt to exceed capacity in PFArrayD.\n";
26         exit(0);
27     }
28     a[used] = element;
29     used++;
30 }

31
32 double& PFArrayD::operator[](int index)
33 {
34     if (index >= used)
35     {
36         cout << "Illegal index in PFArrayD.\n";
37         exit(0);
38     }

39     return a[index];
40 }
```

Display 10.11 Member Function Definitions for PFArrayD Class (part 2 of 2)

```

41 PFArrayD& PFArrayD::operator =(const PFArrayD& rightSide)
42 {
43     if (capacity != rightSide.capacity)
44     {
45         delete [] a;
46         a = new double[rightSide.capacity];
47     }
48
49     capacity = rightSide.capacity;
50     used = rightSide.used;
51     for (int i = 0; i < used; i++)
52         a[i] = rightSide.a[i];
53
54     return *this;
55 }
56
57 PFArrayD::~PFArrayD( )
58 {
59     delete [] a;
60 }

```

Note that this also checks for the case of having the same object on both sides of the assignment operator.

Display 10.12 Demonstration Program for PFArrayD (part 1 of 3)

```

1  //Program to demonstrate the class PFArrayD
2  #include <iostream>
3  using std::cin;
4  using std::cout;
5  using std::endl;
6
7  class PFArrayD
8  {
9      <The rest of the class definition is the same as in Display 10.10 .>
10 };
11
12 void testPFArrayD( );
13 //Conducts one test of the class PFArrayD.
14
15 int main( )
16 {
17     cout << "This program tests the class PFArrayD.\n";

```

In Section 11.1 of Chapter 11 we show you how to divide this long file into three shorter files corresponding roughly to Displays 10.10, 10.11, and this display without the code from Displays 10.10 and 10.11.

Display 10.12 Demonstration Program for PFArrayD (part 2 of 3)

```
15     char ans;
16     do
17     {
18         testPFArrayD( );
19         cout << "Test again? (y/n) ";
20         cin >> ans;
21     }while ((ans == 'y') || (ans == 'Y'));

22     return 0;
23 }

24     <The definitions of the member functions for the class PFArrayD go here.>

25 void testPFArrayD( )
26 {
27     int cap;
28     cout << "Enter capacity of this super array: ";
29     cin >> cap;
30     PFArrayD temp(cap);

31
32     cout << "Enter up to " << cap << " nonnegative numbers.\n";
33     cout << "Place a negative number at the end.\n";

34     double next;
35     cin >> next;
36     while ((next >= 0) && (!temp.full( )))
37     {
38         temp.addElement(next);
39         cin >> next;
40     }

41     cout << "You entered the following "
42         << temp.getNumberUsed( ) << " numbers:\n";
43     int index;
44     int count = temp.getNumberUsed( );
45     for (index = 0; index < count; index++)
46         cout << temp[index] << " ";
47     cout << endl;
48     cout << "(plus a sentinel value.)\n";
49 }
```

Display 10.12 Demonstration Program for PFArrayD (part 3 of 3)

Sample Dialogue

```

This program tests the class PFArrayD.
Enter capacity of this super array: 10
Enter up to 10 nonnegative numbers.
Place a negative number at the end.
1.1
2.2
3.3
4.4
-1
You entered the following 4 numbers:
1.1 2.2 3.3 4.4
(plus a sentinel value.)
Test again? (y/n) n

```

EXAMPLE (continued)

functions like a partially filled array. The member function `getNumberUsed` returns the number of array positions used and can thus be used in a `for` loop as in the following sample code:

```

PFArrayD stuff(cap); //cap is an int variable.
    <some code to fill object stuff with elements.>
for (int index = 0; index < stuff.getNumberUsed( ); index++)
    cout << stuff[index] << " ";

```

An object of the class `PFArrayD` has a dynamic array as a member variable. This member variable `array` stores the elements. The dynamic array member variable is actually a pointer variable. In each constructor, this member variable is set to point at a dynamic array. There are also two member variables of type `int`: The member variable `capacity` records the size of the dynamic array, and the member variable `used` records the number of array positions that have been filled so far. As is customary with partially filled arrays, the elements must be filled in order, going first into position 0, then 1, then 2, and so forth.

An object of the class `PFArrayD` can be used as a partially filled array of `doubles`. It has some advantages over an ordinary array of `doubles` or a dynamic array of `doubles`. Unlike the standard arrays, this array gives an error message if an illegal array index is

EXAMPLE (continued)

used. Also, an object of the class `PFArrayD` does not require an extra `int` variable to keep track of how much of the array is used. (You may protest that “There is such an `int` variable. It’s a member variable.” However, that member variable is a private member variable in the implementation, and a programmer who uses the class `PFArrayD` need never be aware of that member variable.)

An object of the class `PFArrayD` only works for storing values of type `double`. When we discuss templates in Chapter 16, you will see that it would be easy to convert the definition to a template class that would work for any type, but for now we will settle for storing elements of type `double`.

Most of the details in the definition of the class `PFArrayD` use only items covered before now, but there are three new items: a copy constructor, a destructor, and an overloading of the assignment operator. We explain the overloaded assignment operator next and discuss the copy constructor and destructor in the next two subsections.

To see why you want to overload the assignment operator, suppose that the overloading of the assignment operator was omitted from Displays 10.10 and 10.11. Suppose `list1` and `list2` are then declared as follows:

```
PFArrayD list1(10), list2(20);
```

If `list2` has been given a list of numbers with invocations of `list2.addElement`, then even though we are assuming that there is no overloading of the assignment operator, the following assignment statement is still defined, but its meaning may not be what you would like it to be:

```
list1 = list2;
```

With no overloading of the assignment operator, the default predefined assignment operator is used. As usual, this predefined version of the assignment operator copies the value of each of the member variables of `list2` to the corresponding member variables of `list1`. Thus, the value of `list1.a` is changed to be the same as `list2.a`, the value of `list1.capacity` is changed to be the same as `list2.capacity`, and the value of `list1.used` is changed to be the same as `list2.used`. But this can cause problems.

The member variable `list1.a` contains a pointer, and the assignment statement sets this pointer equal to the same value as `list2.a`. Both `list1.a` and `list2.a` therefore point to the same place in memory. Thus, if you change the array `list1.a`, you will also change the array `list2.a`. Similarly, if you change the array `list2.a`, you will also change the array `list1.a`. This is not what we normally want. We usually want the

(continued)

EXAMPLE (continued)

assignment operator to produce a completely independent copy of the thing on the right-hand side. 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 `PArrayD`. This is what we have done in Displays 10.10 and 10.11.

The definition of the overloaded assignment operator in Display 10.11 is reproduced below:

```
PArrayD& PArrayD::operator =(const PArrayD& rightSide)
{
    if (capacity != rightSide.capacity)
    {
        delete [] a;
        a = new double[rightSide.capacity];
    }

    capacity = rightSide.capacity;
    used = rightSide.used;
    for (int i = 0; i < used; i++)
        a[i] = rightSide.a[i];

    return *this;
}
```

When you overload the assignment operator it must be a member of the class; it cannot be a friend of the class. That is why the above definition has only one parameter. For example, consider the following:

```
list1 = list2;
```

In the above call, `list1` is the calling object and `list2` is the argument to the member operator `=`.

Notice that the capacities of the two objects are checked to see if they are equal. If they are not equal, then the array member variable `a` of the left side (that is, of the calling object) is destroyed using `delete` and a new array with the appropriate capacity is created using `new`. This ensures that the object on the left side of the assignment operator will have an array of the correct size, but also does something else that is very important: It ensures that if the same object occurs on both sides of the assignment operator, then the array named by the member variable `a` will not be deleted with a call to `delete`. To see why this is important, consider the following alternative and simpler definition of the overloaded assignment operator:

EXAMPLE (continued)

```
//This version has a bug:
PFArrayD& PFArrayD::operator =(const PFArrayD& rightSide)
{
    delete [] a;
    a = new double[rightSide.capacity];

    capacity = rightSide.capacity;
    used = rightSide.used;
    for (int i = 0; i < used; i++)
        a[i] = rightSide.a[i];

    return *this;
}
```

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

```
myList = myList;
```

When this assignment is executed, the first statement executed is

```
delete [] a;
```

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

```
delete [] myList.a;
```

The pointer `myList.a` is then undefined. The assignment operator has corrupted the object `myList`. This problem cannot happen with the definition of the overloaded assignment operator we gave in Display 10.11.

Shallow Copy and Deep Copy

When defining an overloaded assignment operator or a copy constructor, if your code simply copies the contents of member variables from one object to the other that is known as a *shallow copy*. The default assignment operator and the default copy constructor perform shallow copies. If there are no pointers or dynamically allocated data involved, this works fine. If some member variable names a dynamic array (or points to some other dynamic structure), then you normally do not want a shallow copy. Instead, you want to create a copy of what each member variable is pointing to, so that you get a separate but identical copy, as illustrated in Display 10.11. This is called a *deep copy* and is what we normally do when overloading the assignment operator or defining a copy constructor.

Destructors

Dynamically allocated variables have one problem: 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 by using up all the memory in the freestore manager. Moreover, if the dynamic variable is embedded in the implementation details of a class, the programmer who uses the class may not know about the dynamic variable and cannot be expected to perform the calls 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. If your program contains a local variable that names an object from a class 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 dynamically allocated 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 clean-up details as well, but returning memory to the freestore manager for reuse is the main job of the destructor.

destructor name

The member function `~PFArrayD` is the destructor for the class `PFArrayD` shown in Display 10.10. Like a constructor, a destructor always has the same name as the class of which it is a member, 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.

Notice the definition of the destructor `~PFArrayD` given in Display 10.11. `~PFArrayD` calls `delete` to eliminate the dynamically allocated array pointed to by the member pointer variable `a`. Look again at the function `testPFArrayD` in the sample program shown in Display 10.12. The local variable `temp` contains a dynamic array pointed to by the member variable `temp.a`. If this class did not have a destructor, then after the call to `testPFArrayD` ended, this dynamic array would still be occupying memory, even though the dynamic array is useless to the program. Moreover, every iteration of the `do-while` loop would produce another useless dynamic array to clutter up memory. If the loop is iterated enough times, the function calls could consume all the memory in the freestore manager and your program would then end abnormally.

Destructor

The destructor of a class 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 dynamically allocated variables that have been created by the object so that the memory occupied by these dynamic variables is returned to the freestore manager for reuse. Destructors may perform other clean-up tasks as well. The name of a destructor must consist of the tilde symbol, ~, followed by the name of the class.

Copy Constructors

copy constructor

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. For example, a program that uses the class `PFArrayD` defined in Display 10.10 might contain the following:

```
PFArrayD b(20);
for (int i = 0; i < 20; i++)
    b.addElement(i);
PFArrayD temp(b); //Initialized by the copy constructor
```

The object `b` is initialized with the constructor that has a parameter of type `int`. Similarly, the object `temp` is initialized by the constructor that has one argument of type `const PFArrayD&`. 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

```
PFArrayD temp(b);
```

the member variable `temp.a` should not be simply set to the same value as `b.a`; that would produce two pointers pointing to the same dynamic array. The definition of the copy constructor is shown in Display 10.11. 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 array member variable is different from the array member variable of `b`. The two array member variables, `temp.a` and `b.a`, contain the same values of type `double`, but if a change is made to one of these array member variables, it has no effect on the other array member variable. Thus, any change that is made to `temp` will have no effect on `b`.

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 being declared and is initialized by another object of the same type given in parentheses. (This is the case of using the copy constructor like any other constructor.)
2. When a function returns a value of the class type.
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.”

If you do not define a copy constructor for a class, C++ will automatically generate a copy constructor for you. However, this default copy constructor simply copies the contents of member variables and does not work correctly for classes with pointers or dynamic data in their member variables. Thus, if your class member variables involve pointers, dynamic arrays, or other dynamic data, you should define a copy constructor for the class.

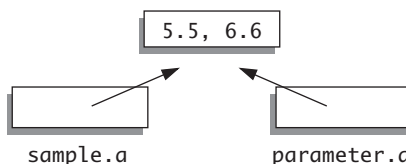
To see why you need a copy constructor, let’s see what would happen if we did not define a copy constructor for the class `PFArrayD`. *Suppose we did not include the copy constructor in the definition of the class `PFArrayD` and suppose we used a call-by-value parameter in a function definition, for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

```
void showPFArrayD(PFArrayD parameter)
{
    cout << "The first value is: "
          << parameter[0] << endl;
}
```

Consider the following code, which includes a function call:

```
PFArrayD sample(2);
sample.addElement(5.5);
sample.addElement(6.6);
showPFArrayD(sample);
cout << "After call: " << sample[0] << endl;
```

Because no copy constructor is defined for the class `PFArrayD`, the class has a default copy constructor that simply copies the contents of member variables. Things then proceed as follows. When the function call is executed, the value of `sample` is copied to the local variable `parameter`, so `parameter.a` is set equal to `sample.a`. But these are pointer variables, so during the function call `parameter.a` and `sample.a` point to the same dynamic array, as follows:



why a copy
constructor is
needed

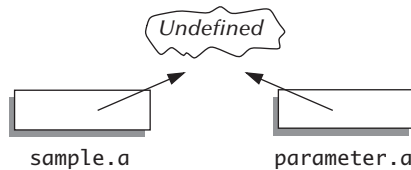
When the function call ends, the destructor for `PFArrayD` is called to return the memory used by `parameter` to the freestore manager so it can be reused. The definition of the destructor contains the following statement:

```
delete [] a;
```

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

```
delete [] parameter.a;
```

which changes the picture to the following:



Since `sample.a` and `parameter.a` point to the same dynamic array, deleting `parameter.a` is the same as deleting `sample.a`. Thus, `sample.a` is undefined when the program reaches the statement

```
cout << "After call: " << sample[0] << endl;
```

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

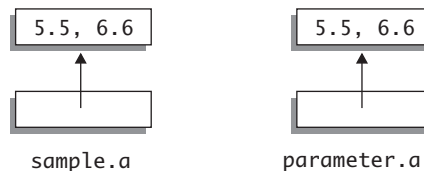
```
delete [] sample.a;
```

But, as we just saw, the dynamic array pointed to by `sample.a` 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 any 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 `PFArrayD`, so the copy constructor is called automatically when the following function call is executed:

```
showPFArrayD(sample);
```

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



Thus, any change that is made to `parameter.a` has no effect on the argument `sample`, and there are no problems with the destructor. If the destructor is called for `parameter` and then called for `sample`, 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, problems similar to what we described for call-by-value parameters will occur.

when you need a copy constructor assignment statements

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

Contrary to what you might expect, the copy constructor is *not* called when you set one object equal to another using the assignment operator.⁶ However, if you do not like what the default assignment operator does, you can redefine the assignment operator as we have done in Displays 10.10 and 10.11.

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, it 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 `=` assignment operator, and the destructor are called the *big three* because experts say that if you need any of them, you need all three. If any of these is missing, the compiler will create it, but the created item might not behave as you want. Thus, it pays to define them yourself. The copy constructor and overloaded `=` assignment operator that the compiler generates for you will work fine if all member variables are of predefined types such as `int` and `double`. For any class that uses pointers and the `new` operator, it is safest to define your own copy constructor, overloaded `=`, and a destructor.

⁶ C++ makes a 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.

Self-Test Exercises

13. 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?
14. Suppose you change the definition of the destructor in Display 10.11 to the following. How would the sample dialogue in Display 10.12 change?

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

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

```
PFArrayD::PFArrayD(const PFArrayD& pfaObject)
```

16. 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?
17.
 - 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 (also called dynamically allocated 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 managers. When a program is finished with a dynamic variable, the memory used by the dynamic variable can be returned to the freestore manager for reuse; this is done with a `delete` statement.

- A dynamically allocated array (also called simply 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 manager 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.
- When overloading the assignment operator, it must be overloaded as a member operator. Be sure to check that your overloading works when the same variable is on both sides of the overloaded assignment operator.

Answers to Self-Test Exercises

1. A pointer is the memory address of a variable.
2.

```
int *p;    // This declares a pointer to an int variable.
*p = 17;   //Here, * is the dereferencing operator. This assigns
           //17 to the memory location pointed to by p.
void func(int* p); // Declares p to be a pointer value
                  // parameter.
```
3.

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

If you replace `*p1 = 30;` with `*p2 = 30;`, the output would be the same.
4.

```
10 20
20 20
30 20
```
5. 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 `intPtr1` to be an int pointer, while `intPtr2` is an ordinary int variable.
6.

```
delete p;
```
7.

```
typedef double* NumberPtr;
NumberPtr myPoint;
```

8. The `new` operator takes a type for its argument. `new` allocates space on the freestore manager for a variable of the type of the argument. It returns a pointer to that memory, provided there is enough space. If there is not enough space, the `new` operator may return `NULL`, or may abort the program, depending on how your particular compiler works.

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

14. The dialogue would change to the following:

```
This program tests the class PFArrayD.
Enter capacity of this super array: 10
Enter up to 10 nonnegative numbers.
Place a negative number at the end.
1.1
2.2
3.3
4.4
-1
You entered the following 4 numbers:
1.1 2.2 3.3 4.4
(plus a sentinel value.)
```

```
Good-bye cruel world! The short life of
this dynamic array is about to end.
Test again? (y/n) n
```

15. The `PFArrayD` before the `::` is the name of the class. The `PFArrayD` 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 `PFArrayD` inside the parentheses is the type for the parameter `pfaObject`.
16. 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 clean-up tasks.

17. In the case of the assignment operator = and the copy constructor, if there are only built-in types for data, the 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



Many of these Programming Projects can be solved using AW's MyCodeMate. To access these please go to: www.mycodemate.com.

1. Reread the code in Display 10.9. Then, write a class `TwoD` that implements the two-dimensional dynamic array of `doubles` using ideas from this display in its constructors. You should have a private member of type pointer to `double` to point to the dynamic array, and two `int` (or `unsigned int`) values that are `MaxRows` and `MaxCols`.

You should supply a default constructor for which you are to choose a default maximum row and column sizes and a parameterized constructor that allows the programmer to set maximum row and column sizes.

Further, you should provide a `void` member function that allows setting a particular row and column entry and a member function that returns a particular row and column entry as a value of type `double`.

Remark: It's difficult or impossible (depending on the details) to overload `[]` so it works as you would like for two-dimensional arrays. So simply use accessor and mutator functions using ordinary function notation.

Overload the `+` operator as a friend function to add two two-dimensional arrays. This function should return the `TwoD` object whose *i*th row, *j*th column element is the sum of the *i*th row, *j*th column element of the left-hand operand `TwoD` object and the *i*th row, *j*th column element of the right-hand operand `TwoD` object.

Provide a copy constructor, an overloaded operator=, and a destructor.

Declare class member functions that do not change the data as `const` members.



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

Discussion: A variable in a polynomial does nothing but 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

$$x^3 + x + 1$$

Where is the term in x^3 ? One simple way to implement the polynomial class is to use an array of `doubles` to store the coefficients. The index of the array is the exponent of the corresponding term. 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 matrix 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.

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.



3. Write a program that accepts a C-string input from the user and reverses the contents of the string. Your program should work by using two pointers. The “head” pointer should be set to the address of the first character in the string, and the “tail” pointer should be set to the address of the last character in the string (i.e., the character before the terminating `null`). The program should swap the characters referenced by these pointers, increment “head” to point to the next character, decrement “tail” to point to the second-to-last character, and so on, until all characters have been swapped and the entire string reversed.
4. Create a class named `Student` that has three member variables:

<code>name</code>	– A string that stores the name of the student
<code>numClasses</code>	– An integer that tracks how many courses the student is currently enrolled in
<code>classList</code>	– A dynamic array of strings used to store the names of the classes that the student is enrolled in

Write appropriate constructor(s), mutator, and accessor functions for the class along with the following:

- A function that inputs all values from the user, including the list of class names. This function will have to support input for an arbitrary number of classes.
- A function that outputs the name and list of all courses.
- A function that resets the number of classes to 0 and the `classList` to an empty list.
- An overloaded assignment operator that correctly makes a new copy of the list of courses.
- A destructor that releases all memory that has been allocated.

Write a `main` function that tests all of your functions.

