CHAPTER 9

Pointers

TOPICS

9.1 Getting the Address of a Variable

CONCEPT: The address operator (&) returns the memory address of a variable.

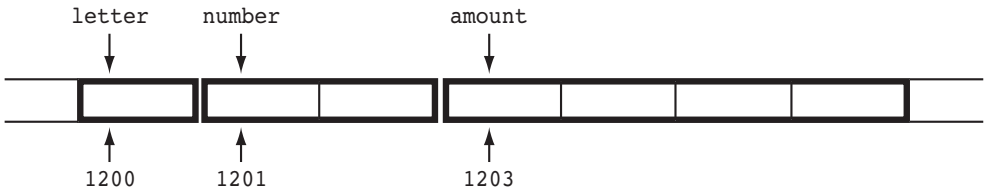
Every variable is allocated a section of memory large enough to hold a value of the variable's data type. On a PC, for instance, it's common for one byte to be allocated for `chars`, two bytes for `shorts`, four bytes for `ints`, `longs`, and `floats`, and eight bytes for `doubles`.

Each byte of memory has a unique *address*. A variable's address is the address of the first byte allocated to that variable. Suppose the following variables are defined in a program:

```
char letter;
short number;
float amount;
```

Figure 9-1 illustrates how they might be arranged in memory and shows their addresses.

Figure 9-1



In Figure 9-1, the variable `letter` is shown at address 1200, `number` is at address 1201, and `amount` is at address 1203.

NOTE: The addresses of the variables shown in Figure 9-1 are arbitrary values used only for illustration purposes.

Getting the address of a variable is accomplished with an operator in C++. When the address operator (`&`) is placed in front of a variable name, it returns the address of that variable. Here is an expression that returns the address of the variable `amount`:

```
&amount
```

And here is a statement that displays the variable's address on the screen:

```
cout << &amount;
```

NOTE: Do not confuse the address operator with the `&` symbol used when defining a reference variable.

Program 9-1 demonstrates the use of the address operator to display the address, size, and contents of a variable.

Program 9-1

```
1  // This program uses the & operator to determine a variable's
2  // address and the sizeof operator to determine its size.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8       int x = 25;
9
10      cout << "The address of x is " << &x << endl;
11      cout << "The size of x is " << sizeof(x) << " bytes\n";
12      cout << "The value in x is " << x << endl;
13      return 0;
14 }
```

Program Output

```
The address of x is 0x8f05
The size of x is 4 bytes
The value in x is 25
```

NOTE: The address of the variable `x` is displayed in hexadecimal. This is the way addresses are normally shown in C++.

9.2 Pointer Variables

CONCEPT: *Pointer variables*, **which are often just called *pointers*, are designed to hold memory addresses. With pointer variables you can indirectly manipulate data stored in other variables.**

A *pointer variable*, which often is just called a *pointer*, is a special variable that holds a memory address. Just as `int` variables are designed to hold integers, and `double` variables are designed to hold floating-point numbers, pointer variables are designed to hold memory addresses.

Memory addresses identify specific locations in the computer's memory. Because a pointer variable holds a memory address, it can be used to hold the location of some other piece of data. This should give you a clue as to why it is called a pointer: It "points" to some piece of data that is stored in the computer's memory. Pointer variables also allow you to work with the data that they point to.

We've already used memory addresses in this book to work with data. Recall from Chapter 6 that when we pass an array as an argument to a function, we are actually passing the array's beginning address. For example, suppose we have an array named `numbers` and we call the `showValues` function as shown here.

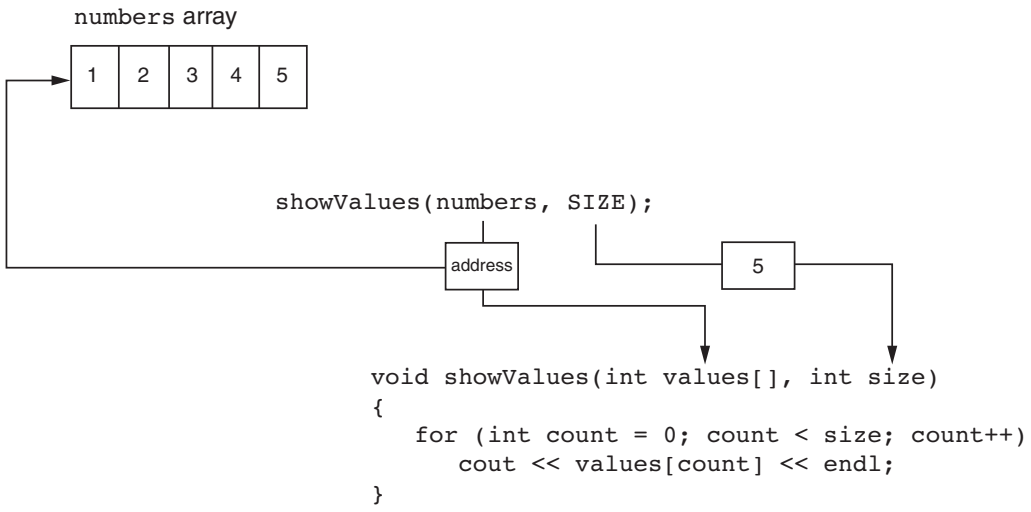
```
const int SIZE = 5;
int numbers[SIZE] = { 1, 2, 3, 4, 5 };
showValues(numbers, SIZE);
```

In this code we are passing the name of the array, `numbers`, and its size as arguments to the `showValues` function. Here is the definition for the `showValues` function:

```
void showValues(int values[], int size)
{
   for (int count = 0; count < size; count++)
      cout << values[count] << endl;
}
```

In the function, the `values` parameter receives the address of the `numbers` array. It works like a pointer because it "points" to the `numbers` array, as shown in Figure 9-2.

Figure 9-2



Inside the `showValues` function, anything that is done to the `values` parameter is actually done to the `numbers` array. We can say that the `values` parameter references the `numbers` array.

Also recall from Chapter 6 that we discussed reference variables. A reference variable acts as an alias for another variable. It is called a reference variable because it references another variable in the program. Anything that you do to the reference variable is actually done to the variable it references. For example, suppose we have the variable `jellyDonuts` and we pass the variable to the `getOrder` function, as shown here:

```
int jellyDonuts;
getOrder(jellyDonuts);
```

Here is the definition for the `getOrder` function:

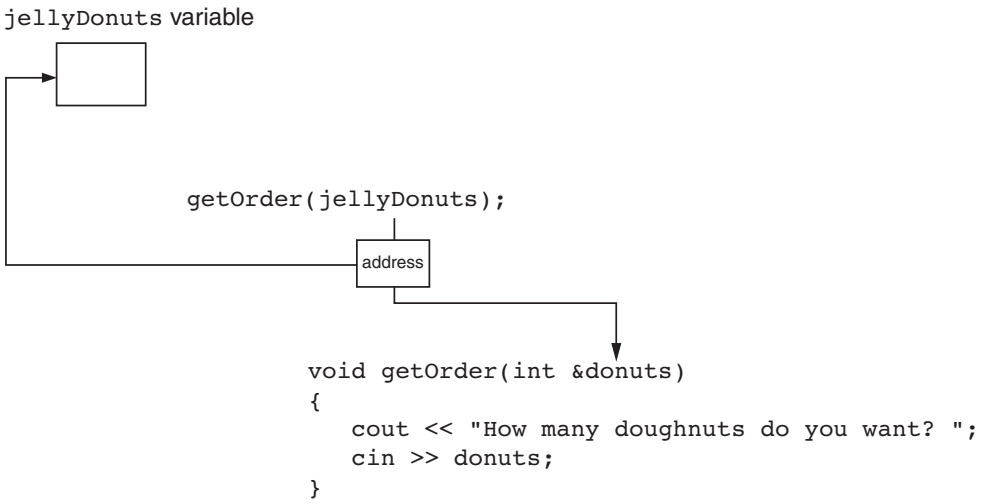
```
void getOrder(int &donuts)
{
   cout << "How many doughnuts do you want? ";
   cin >> donuts;
}
```

In the function, the `donuts` parameter is a reference variable, and it receives the address of the `jellyDonuts` variable. It works like a pointer because it "points" to the `jellyDonuts` variable as shown in Figure 9-3.

Inside the `getOrder` function, the `donuts` parameter references the `jellyDonuts` variable. Anything that is done to the `donuts` parameter is actually done to the `jellyDonuts` variable. When the user enters a value, the `cin` statement uses the `donuts` reference variable to indirectly store the value in the `jellyDonuts` variable.

Notice that the connection between the `donuts` reference variable and the `jellyDonuts` argument is automatically established by C++ when the function is called. When you are writing this code, you don't have go to the trouble of finding the memory address of the `jellyDonuts` variable and then properly storing that address in the `donuts` reference

Figure 9-3



variable. When you are storing a value in the `donuts` variable, you don't have to specify that the value should actually be stored in the `jellyDonuts` variable. C++ handles all of that automatically.

In C++, pointer variables are yet another mechanism for using memory addresses to work with pieces of data. Pointer variables are similar to reference variables, but pointer variables operate at a lower level. By this, I mean that C++ does not automatically do as much work for you with pointer variables as it does with reference variables. In order to make a pointer variable reference another item in memory, you have to write code that fetches the memory address of that item and assigns the address to the pointer variable. Also, when you use a pointer variable to store a value in the memory location that the pointer references, your code has to specify that the value should be stored in the location referenced by the pointer variable, and not in the pointer variable itself.

Because reference variables are easier to work with, you might be wondering why you would ever use pointers at all. In C++, pointers are useful, and even necessary, for many operations. One such operation is dynamic memory allocation. When you are writing a program that will need to work with an unknown amount of data, dynamic memory allocation allows you to create variables, arrays, and more complex data structures in memory while the program is running. We will discuss dynamic memory allocation in greater detail in this chapter. Pointers are also very useful in algorithms that manipulate arrays and work with certain types of strings. In object-oriented programming, which you will learn about in Chapters 13, 14, and 15, pointers are very useful for creating and working with objects and for sharing access to those objects.

Creating and Using Pointer Variables

The definition of a pointer variable looks pretty much like any other definition. Here is an example:

```
int *ptr;
```

The asterisk in front of the variable name indicates that `ptr` is a pointer variable. The `int` data type indicates that `ptr` can be used to hold the address of an integer variable. The definition statement above would read "`ptr` is a pointer to an `int`."

NOTE: In this definition, the word `int` does not mean that `ptr` is an integer variable. It means that `ptr` can hold the address of an integer variable. Remember, pointers only hold one kind of value: an address.

Some programmers prefer to define pointers with the asterisk next to the type name, rather than the variable name. For example, the previous definition shown above could be written as:

```
int* ptr;
```

This style of definition might visually reinforce the fact that `ptr`'s data type is not `int`, but pointer-to-`int`. Both definition styles are correct.

It is never a good idea to define a pointer variable without initializing it with a valid memory address. If you inadvertently use an uninitialized pointer variable, you will be affecting some unknown location in memory. For that reason, it is a good idea to initialize pointer variables with the special value `nullptr`.

In C++ 11, the `nullptr` key word was introduced to represent the address 0. So, assigning `nullptr` to a pointer variable makes the variable point to the address 0. When a pointer is set to the address 0, it is referred to as a *null pointer* because it points to "nothing." Here is an example of how you define a pointer variable and initialize it with the value `nullptr`:

```
int *ptr = nullptr;
```

NOTE: If you are using an older compiler that does not support the C++ 11 standard, you should initialize pointers with the integer 0, or the value `NULL`. The value `NULL` is defined in the `iostream` header file (as well as other header files) to represent the value 0.

Program 9-2 demonstrates a very simple usage of a pointer: storing and printing the address of another variable.

Program 9-2

```
1  // This program stores the address of a variable in a pointer.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       int x = 25;          // int variable
8       int *ptr = nullptr;  // Pointer variable, can point to an int
9
10      ptr = &x;            // Store the address of x in ptr
11      cout << "The value in x is " << x << endl;
12      cout << "The address of x is " << ptr << endl;
13      return 0;
14 }
```

Program Output

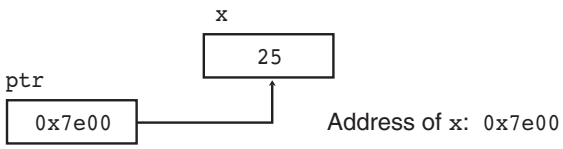
```
The value in x is 25
The address of x is 0x7e00
```

In Program 9-2, two variables are defined: `x` and `ptr`. The variable `x` is an `int`, and the variable `ptr` is a pointer to an `int`. The variable `x` is initialized with the value 25, and `ptr` is initialized with `nullptr`. Then, the variable `ptr` is assigned the address of `x` with the following statement in line 10:

```
ptr = &x;
```

Figure 9-4 illustrates the relationship between `ptr` and `x`.

Figure 9-4



As shown in Figure 9-4, `x`, which is located at memory address 0x7e00, contains the number 25. `ptr` contains the address 0x7e00. In essence, it "points" to the variable `x`.

The real benefit of pointers is that they allow you to indirectly access and modify the variable being pointed to. In Program 9-2, for instance, `ptr` could be used to change the contents of the variable `x`. This is done with the *indirection operator*, which is an asterisk (*). When the indirection operator is placed in front of a pointer variable name, it *dereferences* the pointer. When you are working with a dereferenced pointer, you are actually working with the value the pointer is pointing to. This is demonstrated in Program 9-3.

Program 9-3

```
1  // This program demonstrates the use of the indirection operator.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       int x = 25;           // int variable
8       int *ptr = nullptr;   // Pointer variable, can point to an int
9
10      ptr = &x;             // Store the address of x in ptr
11
12      // Use both x and ptr to display the value in x.
13      cout << "Here is the value in x, printed twice:\n";
14      cout << x << endl;    // Displays the contents of x
15      cout << *ptr << endl; // Displays the contents of x
16
17      // Assign 100 to the location pointed to by ptr. This
18      // will actually assign 100 to x.
19      *ptr = 100;
```

(program continues)

Program 9-3 *(continued)*

```
20
21      // Use both x and ptr to display the value in x.
22      cout << "Once again, here is the value in x:\n";
23      cout << x << endl;    // Displays the contents of x
24      cout << *ptr << endl; // Displays the contents of x
25      return 0;
26  }
```

Program Output

```
Here is the value in x, printed twice:
25
25
Once again, here is the value in x:
100
100
```

Take a closer look at the statement in line 10:

```
ptr = &x;
```

This statement assigns the address of the `x` variable to the `ptr` variable. Now look at line 15:

```
cout << *ptr << endl; // Displays the contents of x
```

When you apply the indirection operator (`*`) to a pointer variable, you are working, not with the pointer variable itself, but with the item it points to. Because this statement sends the expression `*ptr` to the `cout` object, it does not display the value in `ptr`, but the value that `ptr` points to. Since `ptr` points to the `x` variable, this statement displays the contents of the `x` variable.

Suppose the statement did not use the indirection operator. Suppose that statement had been written as:

```
cout << ptr << endl;  // Displays an address
```

Because the indirection operator is not applied to `ptr` in this statement, it works directly with the `ptr` variable. This statement would display the address that is stored in `ptr`.

Now take a look at the following statement, which appears in line 19:

```
*ptr = 100;
```

Notice the indirection operator being used with `ptr`. That means the statement is not affecting `ptr`, but the item that `ptr` points to. This statement assigns 100 to the item `ptr` points to, which is the `x` variable. After this statement executes, 100 will be stored in the `x` variable.

Program 9-4 demonstrates that pointers can point to different variables.

Program 9-4

```
1   // This program demonstrates a pointer variable referencing
2   // different variables.
3   #include <iostream>
4   using namespace std;
```

```
 5
 6  int main()
 7  {
 8       int x = 25, y = 50, z = 75;  // Three int variables
 9       int *ptr = nullptr;          // Pointer variable
10
11       // Display the contents of x, y, and z.
12       cout << "Here are the values of x, y, and z:\n";
13       cout << x << " " << y << " " << z << endl;
14
15       // Use the pointer to manipulate x, y, and z.
16
17       ptr = &x;       // Store the address of x in ptr.
18       *ptr += 100;    // Add 100 to the value in x.
19
20       ptr = &y;       // Store the address of y in ptr.
21       *ptr += 100;    // Add 100 to the value in y.
22
23       ptr = &z;       // Store the address of z in ptr.
24       *ptr += 100;    // Add 100 to the value in z.
25
26       // Display the contents of x, y, and z.
27       cout << "Once again, here are the values of x, y, and z:\n";
28       cout << x << " " << y << " " << z << endl;
29       return 0;
30  }
```

Program Output

```
Here are the values of x, y, and z:
25  50  75
Once again, here are the values of x, y, and z:
125 150 175
```

Take a closer look at the statement in line 17:

```
ptr = &x;
```

This statement assigns the address of the `x` variable to the `ptr` variable. Now look at line 18:

```
*ptr += 100;
```

In this statement notice that the indirection operator (`*`) is used with the `ptr` variable. When we apply the indirection operator to `ptr`, we are working, not with `ptr`, but with the item that `ptr` points to. When this statement executes, `ptr` is pointing at `x`, so the statement in line 18 adds 100 to the contents of `x`. Then the following statement, in line 20, executes:

```
ptr = &y;
```

This statement assigns the address of the `y` variable to the `ptr` variable. After this statement executes, `ptr` is no longer pointing at `x`. Rather, it will be pointing at `y`. The statement in line 21, shown here, adds 100 to the `y` variable.

```
*ptr += 100;
```

These steps are repeated with the `z` variable in lines 23 and 24.

NOTE: So far you've seen three different uses of the asterisk in C++:

- As the multiplication operator, in statements such as
  ```
  distance = speed * time;
  ```
- In the definition of a pointer variable, such as
  ```
  int *ptr = nullptr;
  ```
- As the indirection operator, in statements such as
  ```
  *ptr = 100;
  ```

9.3 The Relationship Between Arrays and Pointers

CONCEPT: **Array names can be used as constant pointers, and pointers can be used as array names.**

You learned in Chapter 7 that an array name, without brackets and a subscript, actually represents the starting address of the array. This means that an array name is really a pointer. Program 9-5 illustrates this by showing an array name being used with the indirection operator.

Program 9-5

```
1  // This program shows an array name being dereferenced with the *
2  // operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8       short numbers[] = {10, 20, 30, 40, 50};
9
10      cout << "The first element of the array is ";
11      cout << *numbers << endl;
12      return 0;
13 }
```

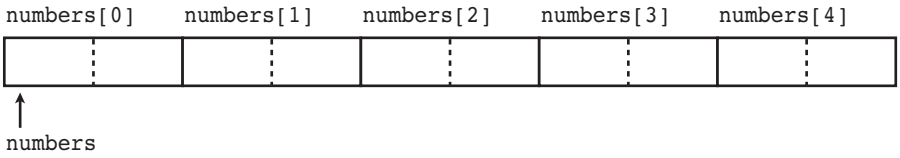
Program Output

```
The first element of the array is 10
```

Because `numbers` works like a pointer to the starting address of the array, the first element is retrieved when `numbers` is dereferenced. So how could the entire contents of an array be retrieved using the indirection operator? Remember, array elements are stored together in memory, as illustrated in Figure 9-5.

It makes sense that if `numbers` is the address of `numbers[0]`, values could be added to `numbers` to get the addresses of the other elements in the array. It's important to know, however, that pointers do not work like regular variables when used in mathematical statements. In C++, when you add a value to a pointer, you are actually adding that value *times*

Figure 9-5



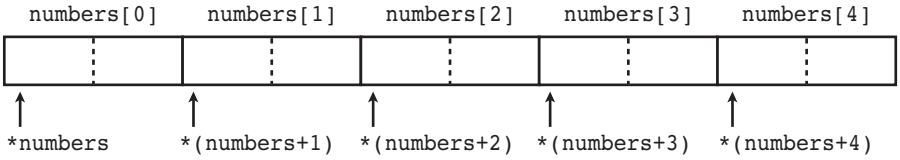
the size of the data type being referenced by the pointer. In other words, if you add one to `numbers`, you are actually adding `1 * sizeof(short)` to `numbers`. If you add two to `numbers`, the result is `numbers + 2 * sizeof(short)`, and so forth. On a PC, this means the following are true, because `short` integers typically use two bytes:

```
*(numbers + 1) is actually *(numbers + 1 * 2)
*(numbers + 2) is actually *(numbers + 2 * 2)
*(numbers + 3) is actually *(numbers + 3 * 2)
```

and so forth.

This automatic conversion means that an element in an array can be retrieved by using its subscript or by adding its subscript to a pointer to the array. If the expression `*numbers`, which is the same as `*(numbers + 0)`, retrieves the first element in the array, then `*(numbers + 1)` retrieves the second element. Likewise, `*(numbers + 2)` retrieves the third element, and so forth. Figure 9-6 shows the equivalence of subscript notation and pointer notation.

Figure 9-6



NOTE: The parentheses are critical when adding values to pointers. The `*` operator has precedence over the `+` operator, so the expression `*number + 1` is not equivalent to `*(number + 1)`. `*number + 1` adds one to the contents of the first element of the array, while `*(number + 1)` adds one to the address in `number`, then dereferences it.

Program 9-6 shows the entire contents of the array being accessed, using pointer notation.

Program 9-6

```
1  // This program processes an array using pointer notation.
2  #include <iostream>
3  using namespace std;
4
```

(program continues)

Program 9-6 *(continued)*

```
 5  int main()
 6  {
 7       const int SIZE = 5; // Size of the array
 8       int numbers[SIZE];  // Array of integers
 9       int count;          // Counter variable
10
11       // Get values to store in the array.
12       // Use pointer notation instead of subscripts.
13       cout << "Enter " << SIZE << " numbers: ";
14       for (count = 0; count < SIZE; count++)
15            cin >> *(numbers + count);
16
17       // Display the values in the array.
18       // Use pointer notation instead of subscripts.
19       cout << "Here are the numbers you entered:\n";
20       for (count = 0; count < SIZE; count++)
21            cout << *(numbers + count)<< " ";
22       cout << endl;
23       return 0;
24  }
```

Program Output with Example Input Shown in Bold

```
Enter 5 numbers: 5 10 15 20 25 [Enter]
Here are the numbers you entered:
5 10 15 20 25
```

When working with arrays, remember the following rule:

array`[index]` is equivalent to `*(`*array*` + index)`

WARNING! Remember that C++ performs no bounds checking with arrays. When stepping through an array with a pointer, it's possible to give the pointer an address outside of the array.

To demonstrate just how close the relationship is between array names and pointers, look at Program 9-7. It defines an array of `double`s and a `double` pointer, which is assigned the starting address of the array. Not only is pointer notation then used with the array name, but subscript notation is used with the pointer!

Program 9-7

```
1  // This program uses subscript notation with a pointer variable and
2  // pointer notation with an array name.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
```

```
int main()
{
     const int NUM_COINS = 5;
     double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
     double *doublePtr;    // Pointer to a double
     int count;            // Array index

     // Assign the address of the coins array to doublePtr.
     doublePtr = coins;

     // Display the contents of the coins array. Use subscripts
     // with the pointer!
     cout << "Here are the values in the coins array:\n";
     for (count = 0; count < NUM_COINS; count++)
          cout << doublePtr[count] << " ";

     // Display the contents of the array again, but this time
     // use pointer notation with the array name!
     cout << "\nAnd here they are again:\n";
     for (count = 0; count < NUM_COINS; count++)
          cout << *(coins + count) << " ";
     cout << endl;
     return 0;
}
```

Program Output

```
Here are the values in the coins array:
0.05 0.1 0.25 0.5 1
And here they are again:
0.05 0.1 0.25 0.5 1
```

Notice that the address operator is not needed when an array's address is assigned to a pointer. Because the name of an array is already an address, use of the & operator would be incorrect. You can, however, use the address operator to get the address of an individual element in an array. For instance, `&numbers[1]` gets the address of `numbers[1]`. This technique is used in Program 9-8.

Program 9-8

```
// This program uses the address of each element in the array.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
     const int NUM_COINS = 5;
     double coins[NUM_COINS] = {0.05, 0.1, 0.25, 0.5, 1.0};
     double *doublePtr = nullptr;  // Pointer to a double
     int count;                    // Array index

```

(program continues)

Program 9-8 *(continued)*

```
13      // Use the pointer to display the values in the array.
14      cout << "Here are the values in the coins array:\n";
15      for (count = 0; count < NUM_COINS; count++)
16      {
17          // Get the address of an array element.
18          doublePtr = &coins[count];
19
20          // Display the contents of the element.
21          cout << *doublePtr << " ";
22      }
23      cout << endl;
24      return 0;
25  }
```

Program Output

```
Here are the values in the coins array:
0.05 0.1 0.25 0.5 1
```

The only difference between array names and pointer variables is that you cannot change the address an array name points to. For example, consider the following definitions:

```
double readings[20], totals[20];
double *dptr = nullptr;
```

These statements are legal:

```
dptr = readings;  // Make dptr point to readings.
dptr = totals;    // Make dptr point to totals.
```

But these are illegal:

```
readings = totals;  // ILLEGAL! Cannot change readings.
totals = dptr;      // ILLEGAL! Cannot change totals.
```

Array names are *pointer constants*. You can't make them point to anything but the array they represent.

9.4 Pointer Arithmetic

CONCEPT: Some mathematical operations may be performed on pointers.

The contents of pointer variables may be changed with mathematical statements that perform addition or subtraction. This is demonstrated in Program 9-9. The first loop increments the pointer variable, stepping it through each element of the array. The second loop decrements the pointer, stepping it through the array backward.

Program 9-9

```
1  // This program uses a pointer to display the contents of an array.
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7       const int SIZE = 8;
8       int set[SIZE] = {5, 10, 15, 20, 25, 30, 35, 40};
9       int *numPtr = nullptr;  // Pointer
10      int count;              // Counter variable for loops
11
12      // Make numPtr point to the set array.
13      numPtr = set;
14
15      // Use the pointer to display the array contents.
16      cout << "The numbers in set are:\n";
17      for (count = 0; count < SIZE; count++)
18      {
19           cout << *numPtr << " ";
20           numPtr++;
21      }
22
23      // Display the array contents in reverse order.
24      cout << "\nThe numbers in set backward are:\n";
25      for (count = 0; count < SIZE; count++)
26      {
27           numPtr--;
28           cout << *numPtr << " ";
29      }
30      return 0;
31 }
```

Program Output

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backward are:
40 35 30 25 20 15 10 5
```

NOTE: Because `numPtr` is a pointer to an integer, the increment operator adds the size of one integer to `numPtr`, so it points to the next element in the array. Likewise, the decrement operator subtracts the size of one integer from the pointer.

Not all arithmetic operations may be performed on pointers. For example, you cannot multiply or divide a pointer. The following operations are allowable:

- The ++ and –– operators may be used to increment or decrement a pointer variable.
- An integer may be added to or subtracted from a pointer variable. This may be performed with the + and – operators, or the += and –= operators.
- A pointer may be subtracted from another pointer.

9.5 Initializing Pointers

CONCEPT: Pointers may be initialized with the address of an existing object.

Remember that a pointer is designed to point to an object of a specific data type. When a pointer is initialized with an address, it must be the address of an object the pointer can point to. For instance, the following definition of `pint` is legal because `myValue` is an integer:

```
int myValue;
int *pint = &myValue;
```

The following is also legal because `ages` is an array of integers:

```
int ages[20];
int *pint = ages;
```

But the following definition of `pint` is illegal because `myFloat` is not an `int`:

```
float myFloat;
int *pint = &myFloat;  // Illegal!
```

Pointers may be defined in the same statement as other variables of the same type. The following statement defines an integer variable, `myValue`, and then defines a pointer, `pint`, which is initialized with the address of `myValue`:

```
int myValue, *pint = &myValue;
```

And the following statement defines an array, `readings`, and a pointer, `marker`, which is initialized with the address of the first element in the array:

```
double readings[50], *marker = readings;
```

Of course, a pointer can only be initialized with the address of an object that has already been defined. The following is illegal because `pint` is being initialized with the address of an object that does not exist yet:

```
int *pint = &myValue;  // Illegal!
int myValue;
```

Checkpoint

9.1 Write a statement that displays the address of the variable `count`.

9.2 Write the definition statement for a variable `fltPtr`. The variable should be a pointer to a `float`.

9.3 List three uses of the `*` symbol in C++.

9.4 What is the output of the following code?

```
int x = 50, y = 60, z = 70;
int *ptr = nullptr;

cout << x << " " << y << " " << z << endl;
ptr = &x;
```

```
*ptr *= 10;
ptr = &y;
*ptr *= 5;
ptr = &z;
*ptr *= 2;
cout << x << " " << y << " " << z << endl;
```

9.5 Rewrite the following loop so it uses pointer notation (with the indirection operator) instead of subscript notation.

```
for (int x = 0; x < 100; x++)
   cout << arr[x] << endl;
```

9.6 Assume `ptr` is a pointer to an `int` and holds the address 12000. On a system with 4-byte integers, what address will be in `ptr` after the following statement?

```
ptr += 10;
```

9.7 Assume `pint` is a pointer variable. Is each of the following statements valid or invalid? If any is invalid, why?

A) `pint++;`

B) `--pint;`

C) `pint /= 2;`

D) `pint *= 4;`

E) `pint += x; // Assume x is an int.`

9.8 Is each of the following definitions valid or invalid? If any is invalid, why?

A)
```
int ivar;
int *iptr = &ivar;
```

B) `int ivar, *iptr = &ivar;`

C)
```
float fvar;
int *iptr = &fvar;
```

D) `int nums[50], *iptr = nums;`

E)
```
int *iptr = &ivar;
int ivar;
```

9.6 Comparing Pointers

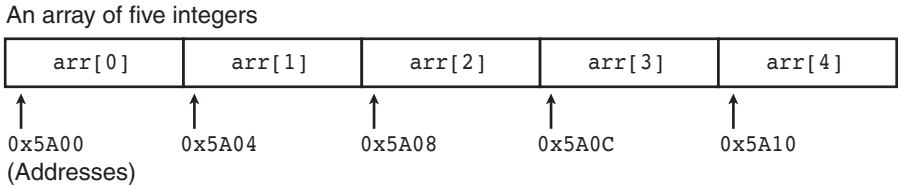
CONCEPT: **If one address comes before another address in memory, the first address is considered "less than" the second. C++'s relational operators may be used to compare pointer values.**

Pointers may be compared by using any of C++'s relational operators:

```
> < == != >= <=
```

In an array, all the elements are stored in consecutive memory locations, so the address of element 1 is greater than the address of element 0. This is illustrated in Figure 9-7.

Figure 9-7



Because the addresses grow larger for each subsequent element in the array, the following `if` statements are all true:

```
if (&arr[1] > &arr[0])
if (arr < &arr[4])
if (arr == &arr[0])
if (&arr[2] != &arr[3])
```

NOTE: Comparing two pointers is not the same as comparing the values the two pointers point to. For example, the following `if` statement compares the addresses stored in the pointer variables `ptr1` and `ptr2`:

```
if (ptr1 < ptr2)
```

The following statement, however, compares the values that `ptr1` and `ptr2` point to:

```
if (*ptr1 < *ptr2)
```

The capability of comparing addresses gives you another way to be sure a pointer does not go beyond the boundaries of an array. Program 9-10 initializes the pointer `nums` with the starting address of the array `set`. The `nums` pointer is then stepped through the array `set` until the address it contains is equal to the address of the last element of the array. Then the pointer is stepped backward through the array until it points to the first element.

Program 9-10

```
1  // This program uses a pointer to display the contents
2  // of an integer array.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8       int set[8] = {5, 10, 15, 20, 25, 30, 35, 40};
9       int *nums = set; // Make nums point to set
10
11      // Display the numbers in the array.
12      cout << "The numbers in set are:\n";
13      cout << *nums << " ";    // Display first element
```

```
14      while (nums < &set[7])
15      {
16         // Advance nums to point to the next element.
17         nums++;
18         // Display the value pointed to by nums.
19         cout << *nums << " ";
20      }
21
22      // Display the numbers in reverse order.
23      cout << "\nThe numbers in set backward are:\n";
24      cout << *nums << " ";  // Display first element
25      while (nums > set)
26      {
27         // Move backward to the previous element.
28         nums--;
29         // Display the value pointed to by nums.
30         cout << *nums << " ";
31      }
32      return 0;
33  }
```

Program Output

```
The numbers in set are:
5 10 15 20 25 30 35 40
The numbers in set backward are:
40 35 30 25 20 15 10 5
```

9.7 Pointers as Function Parameters

CONCEPT: **A pointer can be used as a function parameter. It gives the function access to the original argument, much like a reference parameter does.**

In Chapter 6 you were introduced to the concept of reference variables being used as function parameters. A reference variable acts as an alias to the original variable used as an argument. This gives the function access to the original argument variable, allowing it to change the variable's contents. When a variable is passed into a reference parameter, the argument is said to be passed by reference.

Another way to pass an argument by reference is to use a pointer variable as the parameter. Admittedly, reference variables are much easier to work with than pointers. Reference variables hide all the "mechanics" of dereferencing and indirection. You should still learn to use pointers as function arguments, however, because some tasks, especially when you are dealing with strings, are best done with pointers.* Also, the C++ library has many functions that use pointers as parameters.

* It is also important to learn this technique in case you ever need to write a C program. In C, the only way to pass a variable by reference is to use a pointer.

Here is the definition of a function that uses a pointer parameter:

```
void doubleValue(int *val)
{
   *val *= 2;
}
```

The purpose of this function is to double the variable pointed to by `val` with the following statement:

```
*val *= 2;
```

When `val` is dereferenced, the `*=` operator works on the variable pointed to by `val`. This statement multiplies the original variable, whose address is stored in `val`, by two. Of course, when the function is called, the address of the variable that is to be doubled must be used as the argument, not the variable itself. Here is an example of a call to the `doubleValue` function:

```
doubleValue(&number);
```

This statement uses the address operator (`&`) to pass the address of `number` into the `val` parameter. After the function executes, the contents of `number` will have been multiplied by two. The use of this function is illustrated in Program 9-11.

Program 9-11

```
1  // This program uses two functions that accept addresses of
2  // variables as arguments.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototypes
7  void getNumber(int *);
8  void doubleValue(int *);
9
10 int main()
11 {
12     int number;
13
14     // Call getNumber and pass the address of number.
15     getNumber(&number);
16
17     // Call doubleValue and pass the address of number.
18     doubleValue(&number);
19
20     // Display the value in number.
21     cout << "That value doubled is " << number << endl;
22     return 0;
23 }
24
```

```
25  //*************************************************************
26  // Definition of getNumber. The parameter, input, is a pointer. *
27  // This function asks the user for a number. The value entered  *
28  // is stored in the variable pointed to by input.               *
29  //*************************************************************
30
31  void getNumber(int *input)
32  {
33      cout << "Enter an integer number: ";
34      cin >> *input;
35  }
36
37  //*************************************************************
38  // Definition of doubleValue. The parameter, val, is a pointer. *
39  // This function multiplies the variable pointed to by val by   *
40  // two.                                                         *
41  //*************************************************************
42
43  void doubleValue(int *val)
44  {
45      *val *= 2;
46  }
```

Program Output with Example Input Shown in Bold

```
Enter an integer number: 10 [Enter]
That value doubled is 20
```

Program 9-11 has two functions that use pointers as parameters. Notice the function prototypes:

```
void getNumber(int *);
void doubleValue(int *);
```

Each one uses the notation `int *` to indicate the parameter is a pointer to an `int`. As with all other types of parameters, it isn't necessary to specify the name of the variable in the prototype. The `*` is required, though.

The `getNumber` function asks the user to enter an integer value. The following `cin` statement, in line 34, stores the value entered by the user in memory:

```
cin >> *input;
```

The indirection operator causes the value entered by the user to be stored, not in `input`, but in the variable pointed to by `input`.

WARNING! It's critical that the indirection operator be used in the statement above. Without it, `cin` would store the value entered by the user in `input`, as if the value were an address. If this happens, `input` will no longer point to the `number` variable in function `main`. Subsequent use of the pointer will result in erroneous, if not disastrous, results.

When the `getNumber` function is called in line 15, the address of the `number` variable in function `main` is passed as the argument. After the function executes, the value entered by the user is stored in `number`. Next, the `doubleValue` function is called in line 18, with the address of `number` passed as the argument. This causes `number` to be multiplied by two.

Pointer variables can also be used to accept array addresses as arguments. Either subscript or pointer notation may then be used to work with the contents of the array. This is demonstrated in Program 9-12.

Program 9-12

```
1  // This program demonstrates that a pointer may be used as a
2  // parameter to accept the address of an array.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // Function prototypes
8  void getSales(double *, int);
9  double totalSales(double *, int);
10
11 int main()
12 {
13     const int QTRS = 4;
14     double sales[QTRS];
15
16     // Get the sales data for all quarters.
17     getSales(sales, QTRS);
18
19     // Set the numeric output formatting.
20     cout << fixed << showpoint << setprecision(2);
21
22     // Display the total sales for the year.
23     cout << "The total sales for the year are $";
24     cout << totalSales(sales, QTRS) << endl;
25     return 0;
26 }
27
28 //***************************************************************
29 // Definition of getSales. This function uses a pointer to accept *
30 // the address of an array of doubles. The function asks the user *
31 // to enter sales figures and stores them in the array.           *
32 //***************************************************************
33 void getSales(double *arr, int size)
34 {
35     for (int count = 0; count < size; count++)
36     {
37         cout << "Enter the sales figure for quarter ";
38         cout << (count + 1) << ": ";
39         cin >> arr[count];
40     }
41 }
42
```

```
43  //*************************************************************
44  // Definition of totalSales. This function uses a pointer to      *
45  // accept the address of an array. The function returns the total *
46  // of the elements in the array.                                  *
47  //*************************************************************
48  double totalSales(double *arr, int size)
49  {
50       double sum = 0.0;
51
52       for (int count = 0; count < size; count++)
53       {
54            sum += *arr;
55            arr++;
56       }
57       return sum;
58  }
```

Program Output with Example Input Shown in Bold

```
Enter the sales figure for quarter 1: 10263.98 [Enter]
Enter the sales figure for quarter 2: 12369.69 [Enter]
Enter the sales figure for quarter 3: 11542.13 [Enter]
Enter the sales figure for quarter 4: 14792.06 [Enter]
The total sales for the year are $48967.86
```

Notice that in the `getSales` function in Program 9-12, even though the parameter `arr` is defined as a pointer, subscript notation is used in the `cin` statement in line 39:

```
cin >> arr[count];
```

In the `totalSales` function, `arr` is used with the indirection operator in line 54:

```
sum += *arr;
```

And in line 55, the address in `arr` is incremented to point to the next element:

```
arr++;
```

NOTE: The two previous statements could be combined into the following statement:

```
sum += *arr++;
```

The `*` operator will first dereference `arr`, then the `++` operator will increment the address in `arr`.

Pointers to Constants

You have seen how an item's address can be passed into a pointer parameter, and how the pointer can be used to modify the item that was passed as an argument. Sometimes it is necessary to pass the address of a `const` item into a pointer. When this is the case, the pointer must be defined as a pointer to a `const` item. For example, consider the following array definition:

```
const int SIZE = 6;
const double payRates[SIZE] = { 18.55, 17.45,
                                12.85, 14.97,
                                10.35, 18.89 };
```

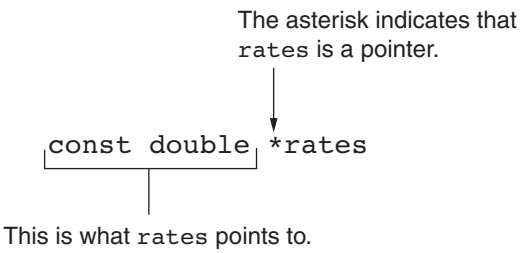
In this code, `payRates` is an array of `const doubles`. This means that each element in the array is a `const double`, and the compiler will not allow us to write code that changes the array's contents. If we want to pass the `payRates` array into a pointer parameter, the parameter must be declared as a pointer to `const double`. The following function shows such an example:

```
void displayPayRates(const double *rates, int size)
{
     // Set numeric output formatting.
     cout << setprecision(2) << fixed << showpoint;

     // Display all the pay rates.
     for (int count = 0; count < size; count++)
     {
          cout << "Pay rate for employee " << (count + 1)
               << " is $" << *(rates + count) << endl;
     }
}
```

In the function header, notice that the `rates` parameter is defined as a pointer to `const double`. It should be noted that the word `const` is applied to the thing that `rates` points to, not `rates` itself. This is illustrated in Figure 9-8.

Figure 9-8



Because `rates` is a pointer to a `const`, the compiler will not allow us to write code that changes the thing that `rates` points to.

In passing the address of a constant into a pointer variable, the variable must be defined as a pointer to a constant. If the word `const` had been left out of the definition of the `rates` parameter, a compiler error would have resulted.

Passing a Nonconstant Argument into a Pointer to a Constant

Although a constant's address can be passed only to a pointer to `const`, a pointer to `const` can also receive the address of a nonconstant item. For example, look at Program 9-13.

Program 9-13

```
// This program demonstrates a pointer to const parameter
#include <iostream>
using namespace std;

void displayValues(const int *, int);

int main()
{
     // Array sizes
     const int SIZE = 6;

     // Define an array of const ints.
     const int array1[SIZE] = { 1, 2, 3, 4, 5, 6 };

     // Define an array of nonconst ints.
     int array2[SIZE] = { 2, 4, 6, 8, 10, 12 };

     // Display the contents of the const array.
     displayValues(array1, SIZE);

     // Display the contents of the nonconst array.
     displayValues(array2, SIZE);
     return 0;
}

//*************************************************
// The displayValues function uses a pointer to    *
// parameter to display the contents of an array.  *
//*************************************************

void displayValues(const int *numbers, int size)
{
     // Display all the values.
     for (int count = 0; count < size; count++)
     {
          cout << *(numbers + count) << " ";
     }
     cout << endl;
}
```

Program Output

```
1 2 3 4 5 6
2 4 6 8 10 12
```

NOTE: When you are writing a function that uses a pointer parameter, and the function is not intended to change the data the parameter points to, it is always a good idea to make the parameter a pointer to `const`. Not only will this protect you from writing code in the function that accidentally changes the argument, but the function will be able to accept the addresses of both constant and nonconstant arguments.

Constant Pointers

In the previous section we discussed pointers to `const`. That is, pointers that point to `const` data. You can also use the `const` key word to define a constant pointer. Here is the difference between a pointer to `const` and a `const` pointer:

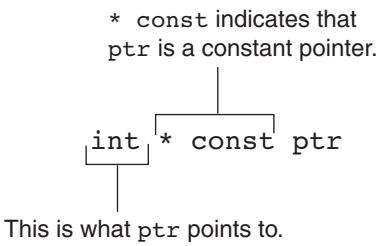
- A pointer to `const` points to a constant item. The data that the pointer points to cannot change, but the pointer itself can change.
- With a `const` pointer, it is the pointer itself that is constant. Once the pointer is initialized with an address, it cannot point to anything else.

The following code shows an example of a `const` pointer.

```
int value = 22;
int * const ptr = &value;
```

Notice in the definition of `ptr` the word `const` appears after the asterisk. This means that `ptr` is a `const` pointer. This is illustrated in Figure 9-9. In the code, `ptr` is initialized with the address of the `value` variable. Because `ptr` is a constant pointer, a compiler error will result if we write code that makes `ptr` point to anything else. An error will not result, however, if we use `ptr` to change the contents of `value`. This is because `value` is not constant, and `ptr` is not a pointer to `const`.

Figure 9-9



Constant pointers must be initialized with a starting value, as shown in the previous example code. If a constant pointer is used as a function parameter, the parameter will be initialized with the address that is passed as an argument into it and cannot be changed to point to anything else while the function is executing. Here is an example that attempts to violate this rule:

```
void setToZero(int * const ptr)
{
   ptr = 0; // ERROR!! Cannot change the contents of ptr.
}
```

This function's parameter, `ptr`, is a `const` pointer. It will not compile because we cannot have code in the function that changes the contents of `ptr`. However, `ptr` does not point to a `const`, so we can have code that changes the data that `ptr` points to. Here is an example of the function that will compile:

```
void setToZero(int * const ptr)
{
   *ptr = 0;
}
```

Although the parameter is const pointer, we can call the function multiple times with different arguments. The following code will successfully pass the addresses of x, y, and z to the setToZero function:

```
int x, y, z;
// Set x, y, and z to 0.
setToZero(&x);
setToZero(&y);
setToZero(&z);
```

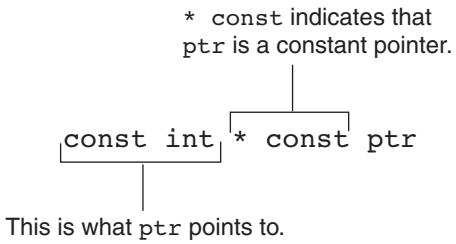
Constant Pointers to Constants

So far, when using const with pointers we've seen pointers to constants and we've seen constant pointers. You can also have constant pointers to constants. For example, look at the following code:

```
int value = 22;
const int * const ptr = &value;
```

In this code ptr is a const pointer to a const int. Notice the word const appears before int, indicating that ptr points to a const int, and it appears after the asterisk, indicating that ptr is a constant pointer. This is illustrated in Figure 9-10.

Figure 9-10



In the code, ptr is initialized with the address of value. Because ptr is a const pointer, we cannot write code that makes ptr point to anything else. Because ptr is a pointer to const, we cannot use it to change the contents of value. The following code shows one more example of a const pointer to a const. This is another version of the displayValues function in Program 9-13.

```
void displayValues(const int * const numbers, int size)
{
   // Display all the values.
   for (int count = 0; count < size; count++)
   {
      cout << *(numbers + count) << " ";
   }
   cout << endl;
}
```

In this code, the parameter numbers is a const pointer to a const int. Although we can call the function with different arguments, the function itself cannot change what numbers points to, and it cannot use numbers to change the contents of an argument.

9.8 Focus on Software Engineering: Dynamic Memory Allocation

CONCEPT: Variables may be created and destroyed while a program is running.

As long as you know how many variables you will need during the execution of a program, you can define those variables up front. For example, a program to calculate the area of a rectangle will need three variables: one for the rectangle's length, one for the rectangle's width, and one to hold the area. If you are writing a program to compute the payroll for 30 employees, you'll probably create an array of 30 elements to hold the amount of pay for each person.

But what about those times when you don't know how many variables you need? For instance, suppose you want to write a test-averaging program that will average any number of tests. Obviously the program would be very versatile, but how do you store the individual test scores in memory if you don't know how many variables to define? Quite simply, you allow the program to create its own variables "on the fly." This is called *dynamic memory allocation* and is only possible through the use of pointers.

To dynamically allocate memory means that a program, while running, asks the computer to set aside a chunk of unused memory large enough to hold a variable of a specific data type. Let's say a program needs to create an integer variable. It will make a request to the computer that it allocate enough bytes to store an `int`. When the computer fills this request, it finds and sets aside a chunk of unused memory large enough for the variable. It then gives the program the starting address of the chunk of memory. The program can only access the newly allocated memory through its address, so a pointer is required to use those bytes.

The way a C++ program requests dynamically allocated memory is through the `new` operator. Assume a program has a pointer to an `int` defined as

```
int *iptr = nullptr;
```

Here is an example of how this pointer may be used with the `new` operator:

```
iptr = new int;
```

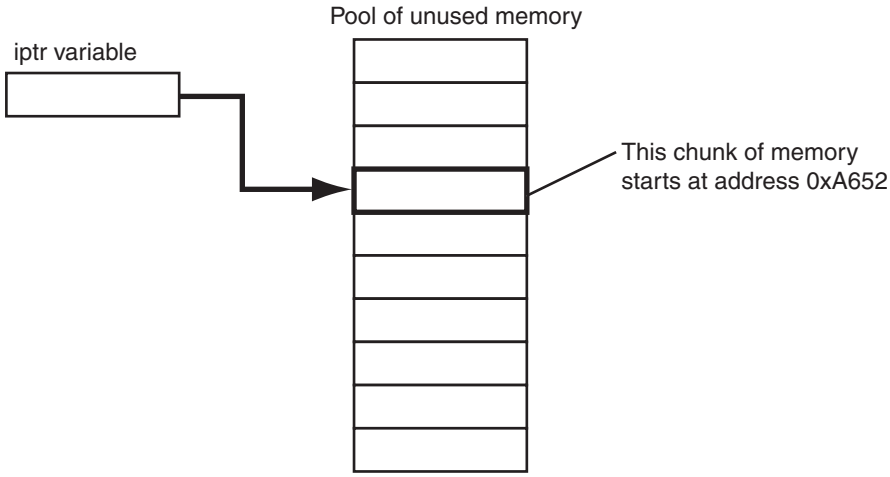
This statement is requesting that the computer allocate enough memory for a new `int` variable. The operand of the `new` operator is the data type of the variable being created. Once the statement executes, `iptr` will contain the address of the newly allocated memory. This is illustrated in Figure 9-11. A value may be stored in this new variable by dereferencing the pointer:

```
*iptr = 25;
```

Any other operation may be performed on the new variable by simply using the dereferenced pointer. Here are some example statements:

```
cout << *iptr;    // Display the contents of the new variable.
cin >> *iptr;     // Let the user input a value.
total += *iptr;   // Use the new variable in a computation.
```

Figure 9-11



Although the statements above illustrate the use of the `new` operator, there's little purpose in dynamically allocating a single variable. A more practical use of the `new` operator is to dynamically create an array. Here is an example of how a 100-element array of integers may be allocated:

```
iptr = new int[100];
```

VideoNote
Dynamically Allocating an Array

Once the array is created, the pointer may be used with subscript notation to access it. For instance, the following loop could be used to store the value 1 in each element:

```
for (int count = 0; count < 100; count++)
     iptr[count] = 1;
```

But what if there isn't enough free memory to accommodate the request? What if the program asks for a chunk large enough to hold a 100,000-element array of `floats`, and that much memory isn't available? When memory cannot be dynamically allocated, C++ throws an exception and terminates the program. *Throwing an exception* means the program signals that an error has occurred. You will learn more about exceptions in Chapter 16.

When a program is finished using a dynamically allocated chunk of memory, it should release it for future use. The `delete` operator is used to free memory that was allocated with `new`. Here is an example of how `delete` is used to free a single variable, pointed to by `iptr`:

```
delete iptr;
```

If `iptr` points to a dynamically allocated array, the `[]` symbol must be placed between `delete` and `iptr`:

```
delete [] iptr;
```

Failure to release dynamically allocated memory can cause a program to have a *memory leak*. For example, suppose the following `grabMemory` function is in a program:

```
void grabMemory()
{
   const int SIZE = 100;
   // Allocate space for a 100-element
   // array of integers.
   int *iptr = new int[SIZE];
   // The function ends without deleting the memory!
}
```

Notice that the function dynamically allocates a 100-element `int` array and assigns its address to the `iptr` pointer variable. But then, the function ends without deleting the memory. Once the function has ended, the `iptr` pointer variable no longer exists. As a result, the program does not have a pointer to the dynamically allocated memory, therefore, the memory cannot be accessed or deleted. It will sit unused as long as the program is running.

WARNING! Only use pointers with `delete` that were previously used with `new`. If you use a pointer with `delete` that does not reference dynamically allocated memory, unexpected problems could result!

Program 9-14 demonstrates the use of `new` and `delete`. It asks for sales figures for any number of days. The figures are stored in a dynamically allocated array and then totaled and averaged.

Program 9-14

```
1  // This program totals and averages the sales figures for any
2  // number of days. The figures are stored in a dynamically
3  // allocated array.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10     double *sales = nullptr, // To dynamically allocate an array
11            total = 0.0,      // Accumulator
12            average;          // To hold average sales
13     int numDays,             // To hold the number of days of sales
14         count;               // Counter variable
15
16     // Get the number of days of sales.
17     cout << "How many days of sales figures do you wish ";
18     cout << "to process? ";
19     cin >> numDays;
```

```
20
21      // Dynamically allocate an array large enough to hold
22      // that many days of sales amounts.
23      sales = new double[numDays];
24
25      // Get the sales figures for each day.
26      cout << "Enter the sales figures below.\n";
27      for (count = 0; count < numDays; count++)
28      {
29          cout << "Day " << (count + 1) << ": ";
30          cin >> sales[count];
31      }
32
33      // Calculate the total sales
34      for (count = 0; count < numDays; count++)
35      {
36          total += sales[count];
37      }
38
39      // Calculate the average sales per day
40      average = total / numDays;
41
42      // Display the results
43      cout << fixed << showpoint << setprecision(2);
44      cout << "\n\nTotal Sales: $" << total << endl;
45      cout << "Average Sales: $" << average << endl;
46
47      // Free dynamically allocated memory
48      delete [] sales;
49      sales = nullptr;    // Make sales a null pointer.
50
51      return 0;
52  }
```

Program Output with Example Input Shown in Bold

```
How many days of sales figures do you wish to process? 5 [Enter]
Enter the sales figures below.
Day 1: 898.63 [Enter]
Day 2: 652.32 [Enter]
Day 3: 741.85 [Enter]
Day 4: 852.96 [Enter]
Day 5: 921.37 [Enter]

Total Sales: $4067.13
Average Sales: $813.43
```

The statement in line 23 dynamically allocates memory for an array of `doubles`, using the value in `numDays` as the number of elements. The `new` operator returns the starting address of the chunk of memory, which is assigned to the `sales` pointer variable. The `sales` variable is then used throughout the program to store the sales amounts in the array and perform the necessary calculations. In line 48 the `delete` operator is used to free the allocated memory.

Notice that in line 49 the value `nullptr` is assigned to the `sales` pointer. It is a good practice to set a pointer variable to `nullptr` after using `delete` on it. First, it prevents code from inadvertently using the pointer to access the area of memory that was freed. Second, it prevents errors from occurring if `delete` is accidentally called on the pointer again. The `delete` operator is designed to have no effect when used on a null pointer.

9.9 Focus on Software Engineering: Returning Pointers from Functions

CONCEPT: Functions can return pointers, but you must be sure the item the pointer references still exists.

Like any other data type, functions may return pointers. For example, the following function locates the null terminator that appears at the end of a string (such as a string literal) and returns a pointer to it.

```
char *findNull(char *str)
{
   char *ptr = str;

   while (*ptr != '\0')
     ptr++;
   return ptr;
}
```

The `char *` return type in the function header indicates the function returns a pointer to a `char`:

```
char *findNull(char *str)
```

When writing functions that return pointers, you should take care not to create elusive bugs. For instance, see if you can determine what's wrong with the following function.

```
string *getFullName()
{
   string fullName[3];
   cout << "Enter your first name: ";
   getline(cin, fullName[0]);
   cout << "Enter your middle name: ";
   getline(cin, fullName[1]);
   cout << "Enter your last name: ";
   getline(cin, fullName[2]);
   return fullName;
}
```

The problem, of course, is that the function returns a pointer to an array that no longer exists. Because the `fullName` array is defined locally, it is destroyed when the function terminates. Attempting to use the pointer will result in erroneous and unpredictable results.

You should return a pointer from a function only if it is

- A pointer to an item that was passed into the function as an argument
- A pointer to a dynamically allocated chunk of memory

For instance, the following function is acceptable:

```
string *getFullName(string fullName[])
{
   cout << "Enter your first name: ";
   getline(cin, fullName[0]);
   cout << "Enter your middle name: ";
   getline(cin, fullName[1]);
   cout << "Enter your last name: ";
   getline(cin, fullName[2]);
   return fullName;
}
```

This function accepts a pointer to the memory location where the user's input is to be stored. Because the pointer references a memory location that was valid prior to the function being called, it is safe to return a pointer to the same location. Here is another acceptable function:

```
string *getFullName()
{
   string *fullName = new string[3];

   cout << "Enter your first name: ";
   getline(cin, fullName[0]);
   cout << "Enter your middle name: ";
   getline(cin, fullName[1]);
   cout << "Enter your last name: ";
   getline(cin, fullName[2]);
   return fullName;
}
```

This function uses the `new` operator to allocate a section of memory. This memory will remain allocated until the `delete` operator is used or the program ends, so it's safe to return a pointer to it.

Program 9-15 shows another example. This program uses a function, `getRandomNumbers`, to get a pointer to an array of random numbers. The function accepts an integer argument that is the number of random numbers in the array. The function dynamically allocates an array, uses the system clock to seed the random number generator, populates the array with random values, and then returns a pointer to the array.

Program 9-15

```
1  // This program demonstrates a function that returns
2  // a pointer.
3  #include <iostream>
4  #include <cstdlib>  // For rand and srand
5  #include <ctime>    // For the time function
6  using namespace std;
7
8  // Function prototype
9  int *getRandomNumbers(int);
10
```

(program continues)

Program 9-15 *(continued)*

```
11  int main()
12  {
13       int *numbers = nullptr;  // To point to the numbers
14
15       // Get an array of five random numbers.
16       numbers = getRandomNumbers(5);
17
18       // Display the numbers.
19       for (int count = 0; count < 5; count++)
20            cout << numbers[count] << endl;
21
22       // Free the memory.
23       delete [] numbers;
24       numbers = 0;
25       return 0;
26  }
27
28  //**************************************************
29  // The getRandomNumbers function returns a pointer  *
30  // to an array of random integers. The parameter    *
31  // indicates the number of numbers requested.       *
32  //**************************************************
33
34  int *getRandomNumbers(int num)
35  {
36       int *arr = nullptr;  // Array to hold the numbers
37
38       // Return a null pointer if num is zero or negative.
39       if (num <= 0)
40            return nullptr;
41
42       // Dynamically allocate the array.
43       arr = new int[num];
44
45       // Seed the random number generator by passing
46       // the return value of time(0) to srand.
47       srand( time(0) );
48
49       // Populate the array with random numbers.
50       for (int count = 0; count < num; count++)
51            arr[count] = rand();
52
53       // Return a pointer to the array.
54       return arr;
55  }
```

Program Output

```
2712
9656
24493
12483
7633
```

In the Spotlight:

Suppose you are developing a program that works with arrays of integers, and you find that you frequently need to duplicate the arrays. Rather than rewriting the array-duplicating code each time you need it, you decide to write a function that accepts an array and its size as arguments, creates a new array that is a copy of the argument array, and returns a pointer to the new array. The function will work as follows:

Accept an array and its size as arguments.
Dynamically allocate a new array that is the same size as the argument array.
Copy the elements of the argument array to the new array.
Return a pointer to the new array.

Program 9-16 demonstrates the function, which is named `duplicateArray`.

Program 9-16

```
1  // This program uses a function to duplicate
2  // an int array of any size.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  int *duplicateArray(const int *, int);
8  void displayArray(const int[], int);
9
10 int main()
11 {
12     // Define constants for the array sizes.
13     const int SIZE1 = 5, SIZE2 = 7, SIZE3 = 10;
14
15     // Define three arrays of different sizes.
16     int array1[SIZE1] = { 100, 200, 300, 400, 500 };
17     int array2[SIZE2] = { 10, 20, 30, 40, 50, 60, 70 };
18     int array3[SIZE3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
19
20     // Define three pointers for the duplicate arrays.
21     int *dup1 = nullptr, *dup2 = nullptr, *dup3 = nullptr;
22
23     // Duplicate the arrays.
24     dup1 = duplicateArray(array1, SIZE1);
25     dup2 = duplicateArray(array2, SIZE2);
26     dup3 = duplicateArray(array3, SIZE3);
27
28     // Display the original arrays.
29     cout << "Here are the original array contents:\n";
30     displayArray(array1, SIZE1);
31     displayArray(array2, SIZE2);
32     displayArray(array3, SIZE3);
33
34     // Display the new arrays.
```

(program continues)

Program 9-16 *(continued)*

```
35       cout << "\nHere are the duplicate arrays: \n";
36       displayArray(dup1, SIZE1);
37       displayArray(dup2, SIZE2);
38       displayArray(dup3, SIZE3);
39
40       // Free the dynamically allocated memory and
41       // set the pointers to 0.
42       delete [] dup1;
43       delete [] dup2;
44       delete [] dup3;
45       dup1 = nullptr;
46       dup2 = nullptr;
47       dup3 = nullptr;
48       return 0;
49    }
50    //*****************************************************
51    // The duplicateArray function accepts an int array   *
52    // and an int that indicates the array's size. The    *
53    // function creates a new array that is a duplicate   *
54    // of the argument array and returns a pointer to the *
55    // new array. If an invalid size is passed the        *
56    // function returns a null pointer.                   *
57    //*****************************************************
58
59    int *duplicateArray(const int *arr, int size)
60    {
61       int *newArray = nullptr;
62
63       // Validate the size. If 0 or a negative
64       // number was passed, return a null pointer.
65       if (size <= 0)
66          return nullptr;
67
68       // Allocate a new array.
69       newArray = new int[size];
70
71       // Copy the array's contents to the
72       // new array.
73       for (int index = 0; index < size; index++)
74          newArray[index] = arr[index];
75
76       // Return a pointer to the new array.
77       return newArray;
78    }
79
80    //************************************************
81    // The displayArray function accepts an int array *
82    // and its size as arguments and displays the     *
83    // contents of the array.                         *
84    //************************************************
85
```

```
86    void displayArray(const int arr[], int size)
87    {
88       for (int index = 0; index < size; index++)
89          cout << arr[index] << " ";
90       cout << endl;
91    }
```

Program Output

```
Here are the original array contents:
100 200 300 400 500
10 20 30 40 50 60 70
1 2 3 4 5 6 7 8 9 10

Here are the duplicate arrays:
100 200 300 400 500
10 20 30 40 50 60 70
1 2 3 4 5 6 7 8 9 10
```

The `duplicateArray` function appears in lines 59 through 78. The `if` statement in lines 65 through 66 validates that `size` contains a valid array size. If `size` is 0 or less, the function immediately returns `nullptr` to indicate that an invalid size was passed.

Line 69 allocates a new array and assigns its address to the `newArray` pointer. Then the loop in lines 73 through 74 copies the elements of the `arr` parameter to the new array. Then the `return` statement in line 77 returns a pointer to the new array.

Checkpoint

9.9 Assuming `arr` is an array of `int`s, will each of the following program segments display "True" or "False"?

A)
```
if (arr < &arr[1])
   cout << "True";
else
   cout << "False";
```

B)
```
if (&arr[4] < &arr[1])
   cout << "True";
else
   cout << "False";
```

C)
```
if (arr != &arr[2])
   cout << "True";
else
   cout << "False";
```

D)
```
if (arr != &arr[0])
   cout << "True";
else
   cout << "False";
```

9.10 Give an example of the proper way to call the following function:

```
void makeNegative(int *val)
{
   if (*val > 0)
```

```
   *val = -(*val);
}
```

9.11 Complete the following program skeleton. When finished, the program will ask the user for a length (in inches), convert that value to centimeters, and display the result. You are to write the function `convert`. (*Note:* 1 inch = 2.54 cm. Do not modify function main.)

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

// Write your function prototype here.

int main()
{
   double measurement;

   cout << "Enter a length in inches, and I will convert\n";
   cout << "it to centimeters: ";
   cin >> measurement;
   convert(&measurement);
   cout << fixed << setprecision(4);
   cout << "Value in centimeters: " << measurement << endl;
   return 0;
}
//
// Write the function convert here.
//
```

9.12 Look at the following array definition:

```
const int numbers[SIZE] = { 18, 17, 12, 14 };
```

Suppose we want to pass the array to the function `processArray` in the following manner:

```
processArray(numbers, SIZE);
```

Which of the following function headers is the correct one for the `processArray` function?

A) `void processArray(const int *arr, int size)`

B) `void processArray(int * const arr, int size)`

9.13 Assume `ip` is a pointer to an `int`. Write a statement that will dynamically allocate an integer variable and store its address in `ip`. Write a statement that will free the memory allocated in the statement you wrote above.

9.14 Assume `ip` is a pointer to an `int`. Then, write a statement that will dynamically allocate an array of 500 integers and store its address in `ip`. Write a statement that will free the memory allocated in the statement you just wrote.

9.15 What is a null pointer?

9.16 Give an example of a function that correctly returns a pointer.

9.17 Give an example of a function that incorrectly returns a pointer.

9.10 Using Smart Pointers to Avoid Memory Leaks

CONCEPT: **C++ 11 introduces smart pointers, objects that work like pointers, but have the ability to automatically delete dynamically allocated memory that is no longer being used.**

In C++ 11, you can use *smart pointers* to dynamically allocate memory and not worry about deleting the memory when you are finished using it. A smart pointer automatically deletes a chunk of dynamically allocated memory when the memory is no longer being used. This helps to prevent memory leaks from occurring.

C++ 11 provides three types of smart pointer: `unique_ptr`, `shared_ptr`, and `weak_ptr`. To use any of the smart pointers, you must `#include` the `memory` header file with the following directive:

```
#include <memory>
```

In this book, we introduce `unique_ptr`. The syntax for defining a `unique_ptr` is somewhat different from the syntax used in defining a regular pointer. Here is an example:

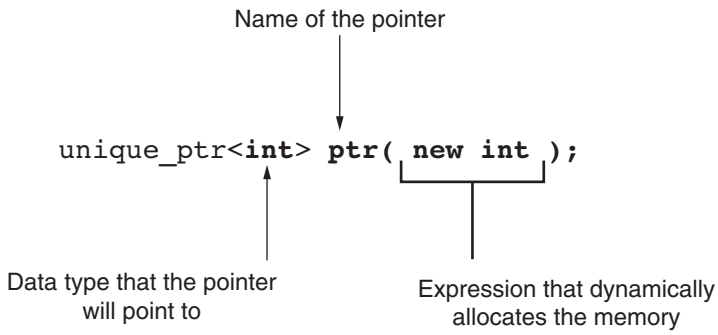
```
unique_ptr<int> ptr( new int );
```

This statement defines a `unique_ptr` named `ptr` that points to a dynamically allocated `int`. Here are some details about the statement:

- The notation `<int>` that appears immediately after `unique_ptr` indicates that the pointer can point to an `int`.
- The name of the pointer is `ptr`.
- The expression `new int` that appears inside the parentheses allocates a chunk of memory to hold an `int`. The address of the chunk of memory will be assigned to the `ptr` pointer.

Figure 9-12 shows the different parts of the definition statement.

Figure 9-12



Once you have defined a `unique_ptr`, you can use it in the same way as a regular pointer. This is demonstrated in Program 9-17.

Program 9-17

```
1  // This program demonstrates a unique_ptr.
2  #include <iostream>
3  #include <memory>
4  using namespace std;
5
6  int main()
7  {
8       // Define a unique_ptr smart pointer, pointing
9       // to a dynamically allocated int.
10      unique_ptr<int> ptr( new int );
11
12      // Assign 99 to the dynamically allocated int.
13      *ptr = 99;
14
15      // Display the value of the dynamically allocated int.
16      cout << *ptr << endl;
17      return 0;
18 }
```

Program Output

```
99
```

In line 3, we have a `#include` directive for the `memory` header file. Line 10 defines a `unique_ptr` named `ptr`, pointing to a dynamically allocated `int`. Line 13 assigns the value 99 to the dynamically allocated `int`. Notice that the indirection operator (`*`) is used with the `unique_ptr`, just as if it were a regular pointer. Line 16 displays the value stored in the dynamically allocated `int`, once again using the indirection operator with the `unique_ptr`. Notice there is no `delete` statement to free the dynamically allocated memory. It is unnecessary to delete the memory because the smart pointer will automatically delete it as the function comes to an end.

Program 9-17 demonstrates a `unique_ptr`, but it isn't very practical. Dynamically allocating an array is more useful than allocating a single integer. The following code shows an example of how to use a `unique_ptr` to dynamically allocate an array of integers.

```
const int SIZE = 100;
unique_ptr<int[]> ptr( new int[SIZE] );
```

The first statement defines an `int` constant named `SIZE`, set to the value 100. The second statement defines a `unique_ptr` named `ptr` that points to a dynamically allocated array of 100 `int`s. Notice the following things about the definition statement:

- Following `unique_ptr`, the notation `<int[]>` indicates that the pointer will point to an array of `int`s.
- The expression inside the parentheses, `new int[SIZE]`, allocates an array of `int`s.

The address of the dynamically allocated array of `int`s will be assigned to the `ptr` pointer. After the definition statement, you can use the `[]` operator with subscripts to access the array elements. Here is an example:

```
ptr[0] = 99;
cout << ptr[0] << endl;
```

The first statement assigns the value 99 to `ptr[0]`, and the second statement displays the value of `ptr[0]`. Program 9-18 gives a more complete demonstration.

Program 9-18

```
1  // This program demonstrates a unique_ptr pointing
2  // to a dynamically allocated array of integers.
3  #include <iostream>
4  #include <memory>
5  using namespace std;
6
7  int main()
8  {
9       int max; // Max size of the array
10
11      // Get the number of values to store.
12      cout << "How many numbers do you want to enter? ";
13      cin >> max;
14
15      // Define a unique_ptr smart pointer, pointing
16      // to a dynamically allocated array of ints.
17      unique_ptr<int[]> ptr( new int[max]);
18
19      // Get values for the array.
20      for (int index = 0; index < max; index++)
21      {
22           cout << "Enter an integer number: ";
23           cin >> ptr[index];
24      }
25
26      // Display the values in the array.
27      cout << "Here are the values you entered:\n";
28      for (int index = 0; index < max; index++)
29           cout << ptr[index] << endl;
30
31      return 0;
32 }
```

Program Output with Example Input Shown in Bold

```
How many numbers do you want to enter? 5 [Enter]
Enter an integer number: 1 [Enter]
Enter an integer number: 2 [Enter]
Enter an integer number: 3 [Enter]
Enter an integer number: 4 [Enter]
Enter an integer number: 5 [Enter]
Here are the values you entered:
1
2
3
4
5
```

9.11 Focus on Problem Solving and Program Design: A Case Study

CONCEPT: **This case study demonstrates how an array of pointers can be used to display the contents of a second array in sorted order, without sorting the `second` array.**

The United Cause, a charitable relief agency, solicits donations from businesses. The local United Cause office received the following donations from the employees of CK Graphics, Inc.:

```
$5, $100, $5, $25, $10, $5, $25, $5, $5, $100, $10, $15, $10, $5, $10
```

The donations were received in the order they appear. The United Cause manager has asked you to write a program that displays the donations in ascending order, as well as in their original order.

Variables

Table 9-1 shows the major variables needed.

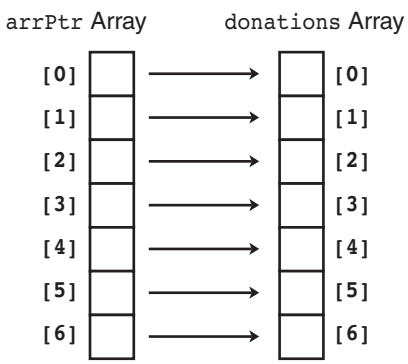
Table 9-1

Variable	Description
`NUM_DONATIONS`	A constant integer initialized with the number of donations received from CK Graphics, Inc. This value will be used in the definition of the program's arrays.
`donations`	An array of integers containing the donation amounts.
`arrPtr`	An array of pointers to integers. This array has the same number of elements as the `donations` array. Each element of `arrPtr` will be initialized to point to an element of the `donations` array.

Programming Strategy

In this program the `donations` array will contain the donations in the order they were received. The elements of the `arrPtr` array are pointers to integers. They will point to the elements of the `donations` array, as illustrated in Figure 9-13.

Figure 9-13

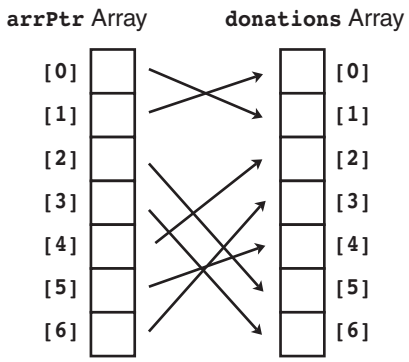


The `arrPtr` array will initially be set up to point to the elements of the `donations` array in their natural order. In other words, `arrPtr[0]` will point to `donations[0]`, `arrPtr[1]` will point to `donations[1]`, and so forth. In that arrangement, the following statement would cause the contents of `donations[5]` to be displayed:

```
cout << *(arrPtr[5]) << endl;
```

After the `arrPtr` array is sorted, however, `arrPtr[0]` will point to the smallest element of `donations`, `arrPtr[1]` will point to the next-to-smallest element of `donations`, and so forth. This is illustrated in Figure 9-14.

Figure 9-14



This technique gives us access to the elements of the `donations` array in a sorted order without actually disturbing the contents of the `donations` array itself.

Modules

The program will consist of the functions listed in Table 9-2.

Table 9-2

Function	Description
`main`	The program's `main` function. It calls the program's other functions.
`arrSelectSort`	Performs an ascending order selection sort on its parameter, `arr`, which is an array of pointers. Each element of `arr` points to an element of a second array. After the sort, `arr` will point to the elements of the second array in ascending order.
`showArray`	Displays the contents of its parameter, `arr`, which is an array of integers. This function is used to display the donations in their original order.
`showArrPtr`	Accepts an array of pointers to integers as an argument. Displays the contents of what each element of the array points to. This function is used to display the contents of the `donations` array in sorted order.

Function `main`

In addition to containing the variable definitions, function `main` sets up the `arrPtr` array to point to the elements of the `donations` array. Then the function `arrSelectSort` is called

to sort the elements of `arrPtr`. Last, the functions `showArrPtr` and `showArray` are called to display the donations. Here is the pseudocode for `main`'s executable statements:

```
For count is set to the values 0 through the number of donations
   Set arrPtr[count] to the address of donations[count].
End For
Call arrSelectSort.
Call showArrPtr.
Call showArray.
```

The `arrSelectSort` Function

The `arrSelectSort` function is a modified version of the selection sort algorithm shown in Chapter 8. The only difference is that `arr` is now an array of pointers. Instead of sorting on the contents of `arr`'s elements, `arr` is sorted on the contents of what its elements point to. Here is the pseudocode:

```
For startScan is set to the values 0 up to (but not including) the
         next-to-last subscript in arr
  Set index variable to startScan.
  Set minIndex variable to startScan.
  Set minElem pointer to arr[startScan].
  For index variable is set to the values from (startScan + 1) through
               the last subscript in arr
     If *(arr[index]) is less than *minElem
        Set minElem to arr[index].
        Set minIndex to index.
     End If.
  End For.
  Set arr[minIndex] to arr[startScan].
  Set arr[startScan] to minElem.
End For.
```

The `showArrPtr` Function

The `showArrPtr` function accepts an array of pointers as its argument. It displays the values pointed to by the elements of the array. Here is its pseudocode:

```
For every element in the arr
   Dereference the element and display what it points to.
End For.
```

The `showArray` Function

The `showArray` function simply displays the contents of `arr` sequentially. Here is its pseudocode:

```
For every element in arr
   Display the element's contents
End For.
```

The Entire Program

Program 9-19 shows the entire program's source code.

Program 9-19

```
1  // This program shows the donations made to the United Cause
2  // by the employees of CK Graphics, Inc. It displays
3  // the donations in order from lowest to highest
4  // and in the original order they were received.
5  #include <iostream>
6  using namespace std;
7
8  // Function prototypes
9  void arrSelectSort(int *[], int);
10 void showArray(const int [], int);
11 void showArrPtr(int *[], int);
12
13 int main()
14 {
15     const int NUM_DONATIONS = 15;    // Number of donations
16
17     // An array containing the donation amounts.
18     int donations[NUM_DONATIONS] = { 5,  100, 5,  25, 10,
19                                      5,  25,  5,  5,  100,
20                                      10, 15,  10, 5,  10 };
21
22     // An array of pointers to int.
23     int *arrPtr[NUM_DONATIONS] = { nullptr, nullptr, nullptr, nullptr, nullptr,
24                                    nullptr, nullptr, nullptr, nullptr, nullptr,
25                                    nullptr, nullptr, nullptr, nullptr, nullptr };
26
27     // Each element of arrPtr is a pointer to int. Make each
28     // element point to an element in the donations array.
29     for (int count = 0; count < NUM_DONATIONS; count++)
30         arrPtr[count] = &donations[count];
31
32     // Sort the elements of the array of pointers.
33     arrSelectSort(arrPtr, NUM_DONATIONS);
34
35     // Display the donations using the array of pointers. This
36     // will display them in sorted order.
37     cout << "The donations, sorted in ascending order are: \n";
38     showArrPtr(arrPtr, NUM_DONATIONS);
39
40     // Display the donations in their original order.
41     cout << "The donations, in their original order are: \n";
42     showArray(donations, NUM_DONATIONS);
43     return 0;
44 }
45
```

(program continues)

Program 9-19 *(continued)*

```
46  //*************************************************************
47  // Definition of function arrSelectSort.                      *
48  // This function performs an ascending order selection sort on  *
49  // arr, which is an array of pointers. Each element of array    *
50  // points to an element of a second array. After the sort,      *
51  // arr will point to the elements of the second array in        *
52  // ascending order.                                           *
53  //*************************************************************
54
55  void arrSelectSort(int *arr[], int size)
56  {
57      int startScan, minIndex;
58      int *minElem;
59
60      for (startScan = 0; startScan < (size - 1); startScan++)
61      {
62          minIndex = startScan;
63          minElem = arr[startScan];
64          for(int index = startScan + 1; index < size; index++)
65          {
66              if (*(arr[index]) < *minElem)
67              {
68                  minElem = arr[index];
69                  minIndex = index;
70              }
71          }
72          arr[minIndex] = arr[startScan];
73          arr[startScan] = minElem;
74      }
75  }
76
77  //*********************************************************
78  // Definition of function showArray.                      *
79  // This function displays the contents of arr. size is the *
80  // number of elements.                                    *
81  //*********************************************************
82
83  void showArray(const int arr[], int size)
84  {
85      for (int count = 0; count < size; count++)
86          cout << arr[count] << " ";
87      cout << endl;
88  }
89
90  //*************************************************************
91  // Definition of function showArrPtr.                         *
92  // This function displays the contents of the array pointed to *
93  // by arr. size is the number of elements.                    *
94  //*************************************************************
95
```

```
 96  void showArrPtr(int *arr[], int size)
 97  {
 98       for (int count = 0; count < size; count++)
 99            cout << *(arr[count]) << " ";
100       cout << endl;
101  }
```

Program Output

```
The donations, sorted in ascending order, are:
5 5 5 5 5 5 10 10 10 10 15 25 25 100 100
The donations, in their original order, are:
5 100 5 25 10 5 25 5 5 100 10 15 10 5 10
```

Review Questions and Exercises

Short Answer

1. What does the indirection operator do?
2. Look at the following code.

   ```
   int x = 7;
   int *iptr = &x;
   ```

   What will be displayed if you send the expression `*iptr` to `cout`? What happens if you send the expression `ptr` to `cout`?
3. So far you have learned three different uses for the `*` operator. What are they?
4. What math operations are allowed on pointers?
5. Assuming that `ptr` is a pointer to an `int`, what happens when you add 4 to `ptr`?
6. Look at the following array definition.

   ```
   int numbers[] = { 2, 4, 6, 8, 10 };
   ```

   What will the following statement display?

   ```
   cout << *(numbers + 3) << endl;
   ```
7. What is the purpose of the `new` operator?
8. What happens when a program uses the `new` operator to allocate a block of memory, but the amount of requested memory isn't available? How do programs written with older compilers handle this?
9. What is the purpose of the `delete` operator?
10. Under what circumstances can you successfully return a pointer from a function?
11. What is the difference between a pointer to a constant and a constant pointer?
12. What are two advantages of declaring a pointer parameter as a constant pointer?

Fill-in-the-Blank

13. Each byte in memory is assigned a unique _____.
14. The _____ operator can be used to determine a variable's address.

15. _____ variables are designed to hold addresses.
16. The _____ operator can be used to work with the variable a pointer points to.
17. Array names can be used as _____, and vice versa.
18. Creating variables while a program is running is called _____.
19. The _____ operator is used to dynamically allocate memory.
20. Under older compilers, if the `new` operator cannot allocate the amount of memory requested, it returns _____.
21. A pointer that contains the address 0 is called a(n) _____ pointer.
22. When a program is finished with a chunk of dynamically allocated memory, it should free it with the _____ operator.
23. You should only use pointers with `delete` that were previously used with _____.

Algorithm Workbench

24. Look at the following code.

```
double value = 29.7;
double *ptr = &value;
```

Write a `cout` statement that uses the `ptr` variable to display the contents of the `value` variable.

25. Look at the following array definition.

```
int set[10];
```

Write a statement using pointer notation that stores the value 99 in `set[7];`

26. Write code that dynamically allocates an array of 20 integers, then uses a loop to allow the user to enter values for each element of the array.
27. Assume that `tempNumbers` is a pointer that points to a dynamically allocated array. Write code that releases the memory used by the array.
28. Look at the following function definition.

```
void getNumber(int &n)
{
   cout << "Enter a number: ";
   cin >> n;
}
```

In this function, the parameter `n` is a reference variable. Rewrite the function so that `n` is a pointer.

29. Write the definition of `ptr`, a pointer to a constant `int`.
30. Write the definition of `ptr`, a constant pointer to an `int`.

True or False

31. T F Each byte of memory is assigned a unique address.
32. T F The `*` operator is used to get the address of a variable.
33. T F Pointer variables are designed to hold addresses.
34. T F The `&` symbol is called the indirection operator.
35. T F The `&` operator dereferences a pointer.

36. T F When the indirection operator is used with a pointer variable, you are actually working with the value the pointer is pointing to.
37. T F Array names cannot be dereferenced with the indirection operator.
38. T F When you add a value to a pointer, you are actually adding that number times the size of the data type referenced by the pointer.
39. T F The address operator is not needed to assign an array's address to a pointer.
40. T F You can change the address that an array name points to.
41. T F Any mathematical operation, including multiplication and division, may be performed on a pointer.
42. T F Pointers may be compared using the relational operators.
43. T F When used as function parameters, reference variables are much easier to work with than pointers.
44. T F The `new` operator dynamically allocates memory.
45. T F A pointer variable that has not been initialized is called a null pointer.
46. T F The address 0 is generally considered unusable.
47. T F In using a pointer with the `delete` operator, it is not necessary for the pointer to have been previously used with the `new` operator.

Find the Error

Each of the following definitions and program segments has errors. Locate as many as you can.

48.
```
int ptr* = nullptr;
```

49.
```
int x, *ptr = nullptr;
&x = ptr;
```

50.
```
int x, *ptr = nullptr;
*ptr = &x;
```

51.
```
int x, *ptr = nullptr;
ptr = &x;
ptr = 100; // Store 100 in x
cout << x << endl;
```

52.
```
int numbers[] = {10, 20, 30, 40, 50};
cout << "The third element in the array is ";
cout << *numbers + 3 << endl;
```

53.
```
int values[20], *iptr = nullptr;
iptr = values;
iptr *= 2;
```

54.
```
float level;
int fptr = &level;
```

55.
```
int *iptr = &ivalue;
int ivalue;
```

56.
```
void doubleVal(int val)
{
   *val *= 2;
}
```

57.
```
int *pint = nullptr;
new pint;
```

58.
```
int *pint = nullptr;
pint = new int;
if (pint == nullptr)
   *pint = 100;
else
   cout << "Memory allocation error\n";
```

59.
```
int *pint = nullptr;
pint = new int[100]; // Allocate memory
   .
   .
```
Code that processes the array.
```
   .
   .
delete pint;  // Free memory
```

60.
```
int *getNum()
{
   int wholeNum;
   cout << "Enter a number: ";
   cin >> wholeNum;
   return &wholeNum;
}
```

61.
```
const int arr[] = { 1, 2, 3 };
int *ptr = arr;
```

62.
```
void doSomething(int * const ptr)
{
   int localArray[] = { 1, 2, 3 };
   ptr = localArray;
}
```

Programming Challenges

1. **Array Allocator**

   Write a function that dynamically allocates an array of integers. The function should accept an integer argument indicating the number of elements to allocate. The function should return a pointer to the array.

2. **Test Scores #1**

   Write a program that dynamically allocates an array large enough to hold a user-defined number of test scores. Once all the scores are entered, the array should be passed to a function that sorts them in ascending order. Another function should be

called that calculates the average score. The program should display the sorted list of scores and averages with appropriate headings. Use pointer notation rather than array notation whenever possible.

Input Validation: Do not accept negative numbers for test scores.

3. Drop Lowest Score

Modify Problem 2 above so the lowest test score is dropped. This score should not be included in the calculation of the average.

4. Test Scores #2

Modify the program of Programming Challenge 2 to allow the user to enter name-score pairs. For each student taking a test, the user types the student's name followed by the student's integer test score. Modify the sorting function so it takes an array holding the student names and an array holding the student test scores. When the sorted list of scores is displayed, each student's name should be displayed along with his or her score. In stepping through the arrays, use pointers rather than array subscripts.

5. Pointer Rewrite

VideoNote
Solving the Pointer Rewrite Problem

The following function uses reference variables as parameters. Rewrite the function so it uses pointers instead of reference variables, and then demonstrate the function in a complete program.

```
int doSomething(int &x, int &y)
{
    int temp = x;
    x = y * 10;
    y = temp * 10;
    return x + y;
}
```

6. Case Study Modification #1

Modify Program 9-19 (the United Cause case study program) so it can be used with any set of donations. The program should dynamically allocate the `donations` array and ask the user to input its values.

7. Case Study Modification #2

Modify Program 9-19 (the United Cause case study program) so the `arrptr` array is sorted in descending order instead of ascending order.

8. Mode Function

In statistics, the *mode* of a set of values is the value that occurs most often or with the greatest frequency. Write a function that accepts as arguments the following:

A) An array of integers

B) An integer that indicates the number of elements in the array

The function should determine the mode of the array. That is, it should determine which value in the array occurs most often. The mode is the value the function should return. If the array has no mode (none of the values occur more than once), the function should return –1. (Assume the array will always contain nonnegative values.)

Demonstrate your pointer prowess by using pointer notation instead of array notation in this function.

9. Median Function

In statistics, when a set of values is sorted in ascending or descending order, its *median* is the middle value. If the set contains an even number of values, the median is the mean, or average, of the two middle values. Write a function that accepts as arguments the following:

A) An array of integers

B) An integer that indicates the number of elements in the array

The function should determine the median of the array. This value should be returned as a `double`. (Assume the values in the array are already sorted.)

Demonstrate your pointer prowess by using pointer notation instead of array notation in this function.

10. Reverse Array

Write a function that accepts an `int` array and the array's size as arguments. The function should create a copy of the array, except that the element values should be reversed in the copy. The function should return a pointer to the new array. Demonstrate the function in a complete program.

11. Array Expander

Write a function that accepts an `int` array and the array's size as arguments. The function should create a new array that is twice the size of the argument array. The function should copy the contents of the argument array to the new array and initialize the unused elements of the second array with 0. The function should return a pointer to the new array.

12. Element Shifter

Write a function that accepts an `int` array and the array's size as arguments. The function should create a new array that is one element larger than the argument array. The first element of the new array should be set to 0. Element 0 of the argument array should be copied to element 1 of the new array, element 1 of the argument array should be copied to element 2 of the new array, and so forth. The function should return a pointer to the new array.

13. Movie Statistics

Write a program that can be used to gather statistical data about the number of movies college students see in a month. The program should perform the following steps:

A) Ask the user how many students were surveyed. An array of integers with this many elements should then be dynamically allocated.

B) Allow the user to enter the number of movies each student saw into the array.

C) Calculate and display the average, median, and mode of the values entered. (Use the functions you wrote in Problems 8 and 9 to calculate the median and mode.)

Input Validation: Do not accept negative numbers for input.