# 6 Pointers: Getting a Handle on Data

*Perhaps more than anything else, the C-based languages (of which C++ is a proud member) are characterized by the use of pointers—variables that store memory addresses. The subject sometimes gets a reputation for being difficult for beginners. It seems to some people that pointers are part of a plot by experienced programmers to wreak vengeance on the rest of us (because they didn't get chosen for basketball, invited to the prom, or whatever).*

*But a pointer is just another way of referring to data. There's nothing mysterious about pointers; you will always succeed in using them if you follow simple, specific steps. Of course, you may find yourself baffled (as I once did) that you need to go through these steps at all—why not just use simple variable names in all cases? True enough, the use of simple variables to refer to data is often sufficient.*

*But sometimes, programs have special needs. What I hope to show in this chapter is why the extra steps involved in pointer use are often justified.*

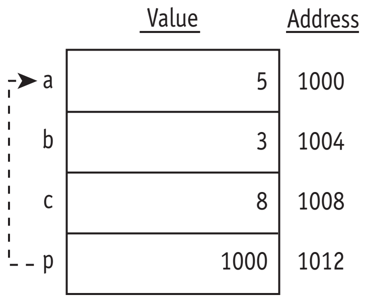
## The Concept of Pointer

The simplest programs consist of one function (**main**) that doesn't interact with the network or operating system. With such programs, you can probably go the rest of your life without pointers.

But when you write programs with multiple functions, you may need one function to hand off a data address to another. In C and C++ this is the only way a function can change the value of arguments passed to it. This is called *pass by reference*.

**Note** C++ features an alternative technique for enabling pass by reference, which I'll introduce in Chapter 12, but the underlying mechanics are similar. It's best to understand pointers first.

A pointer is a variable that contains an address. For example, suppose you have three **int** variables, a, b, c, and one pointer variable, p. Assume that a, b, and c are initialized to 3, 5, and 8, and the pointer is initialized to point to a. In that case, the layout of the computer's memory might look like this:



This figure assumes that variables a, b, c, and p have numeric addresses 1000, 1004, 1008, and 1012. Data addresses vary from one system to another, but the distances between the addresses are uniform. The **int** type and all pointer types on a 32-bit system take up four bytes each; a variable of type **double** takes up eight bytes.

All memory locations on a computer have a numeric address—even though you almost never know or care what it is. Yet at the level of machine code, addresses are about the only thing the CPU understands! A reference to a variable Num in your program, for example, is translated into a reference to a numeric address. (The compiler, linker, and loader assign this address to Num, so you don't have to think about it.)

Pointer usage is about *indirect* memory reference. CPUs are optimized to do these references efficiently, which is one reason why C-based languages encourage their use. At runtime an address is loaded into an on-board CPU register, which is then used to access items in memory.

*Interlude*

## What Do Addresses Really Look Like?

In the last section, I assumed that the variables a, b, and c had the physical addresses 1000, 1004, and 1008. While this is possible, it's unlikely—just a shot in the dark.

Actually, when hard-core programmers represent bit patterns or addresses, they use the hexadecimal numbering system. This means base 16 rather than base 10.

There's a good reason for using hexadecimal notation. Because 16 is an exact power of 2 (2 * 2 * 2 * 2 = 16), each hexadecimal digit corresponds to a unique pattern of exactly four binary digits—no more, no less. Table 6.1 shows how hexadecimal digits work.

Interlude

▼ continued

**Table 6.1: Hexadecimal Equivalents**

| HEXADECIMAL DIGIT | EQUIVALENT DECIMAL | EQUIVALENT BINARY |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

The advantage of using hexadecimal notation for addresses is you can tell how wide an address is just by looking at it. One hexadecimal digit always corresponds precisely to four binary digits. The address 0x8000 has four hexadecimal digits. (The "0x" prefix is the C++ notation that means the number is hexadecimal.) Four hexadecimal digits correspond precisely to 16 binary digits. The math is very simple: 4 * 4 = 16.

In contrast, it's hard to tell how many binary digits (or bits) the decimal number 1000 corresponds to. The answer is 10 bits. (Did you know that?) But decimal 5000 requires 13 bits to represent in binary. Decimal-to-binary is vastly more difficult than hexadecimal-to-binary; therefore, systems programmers favor hexadecimal.

On nearly all personal computers still in use today addresses are not 16 bits wide, but 32. An address such as 0x8FFF1000 is a 32-bit address, because it has eight hexadecimal digits (8 * 4 = 32). 0x00002222 is also a 32-bit address, because the assumption is that the leading zeros are preserved.

▼ continued on next page

Interlude

▼ continued

When personal computers were introduced in the 1970s, 16-bit addresses were the norm. The number of possible addresses were 2 to the 16th power, or 64k. That meant that no matter how many memory cards you purchased, the processor simply couldn't recognize more than 64k of memory.

The 8086 processor, utilized in the IBM PC and the early clones that followed, used a 20-bit address system, which supported up to 16 "segments" each of which could be 64k. (Again, this made 64k a magic number.) Addressable memory increased to a little more than one megabyte—better than 64k, but still woefully inadequate by today's standards.

By the mid 1990s, 32-bit addresses became the standard and Microsoft Windows fully supported it. Software was rewritten and recompiled so that it stopped using the awkward segmented addressing mode and went to clean 32-bit addresses. The number of possible memory addresses is now 2 to the 32nd power, or somewhat more than four *billion*—an upper limit of four gigabytes! But at the rate which memory is improving, it's only a matter of time until hardware capabilities exceed this limit. Stay tuned to see what workarounds computer architects devise when that day comes.

## Declaring and Using Pointers

A pointer declaration uses the following syntax:

```
type  *name;
```

For example, you can declare a pointer p, which can point to variables of type **int**:

```
int  *p;
```

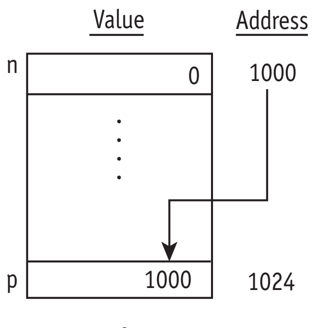
At the moment, the pointer is not initialized. All we know is that it can point to data objects of type **int**. Does this type matter? Yes. The base type of a pointer determines how the data it points to is interpreted. Assign it to point to some other type, and the wrong data format will be used. The pointer p has type **int***; so it should only be able to point to **int** variables.

The next statements declare an integer n, initialize it to 0, and assign its address to pointer p.

```
int n = 0;
p = &n;            // p now points to n!
```

The ampersand (&) is another new operator. Its purpose in life is to get an address. Again, you don't really care what the numeric address is; all that matters is that p contains the address of n—that is, p *points to* n.

After these statements are executed, a possible memory layout for the program is:



p = &n;

Here comes the interesting part. There are a couple of ways to use a pointer variable. One is to change the value of p itself.

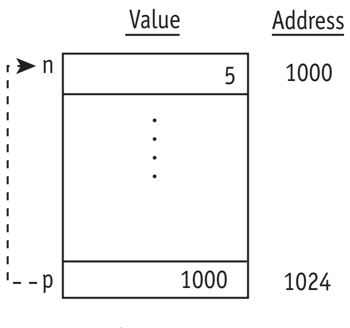
```
p++;     // Point to next item in memory.
```

This adds 1 to p, which makes it point to the variable in memory after n. The effects are unpredictable except in the case of array elements, which I describe later in the section "Pointers and Array Processing." Otherwise, this kind of operation should be avoided.

The other way to use a pointer is more useful—at least in this case. Applying the indirection operator (*) says "the thing pointed to by this pointer." Therefore, assigning a value to *p has the same effect as assigning to n, because n is what p points to.

```
*p = 5;    // Assign 5 to the int pointed to by p.
```

So, this operation changes the thing that p points to, not the value of p itself. Now the memory layout looks like this:



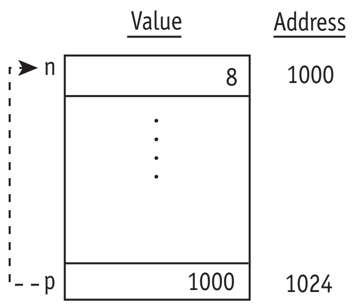
*p = 5;

The effect of the statement, in this case, is the same as "n = 5". The computer finds the memory location pointed to by p and puts the value 5 at that location.

You can use a value pointed to by a pointer both to get and to assign data. Here's another example of pointer use:

```
*p = *p + 3;    // Add 3 to the int pointed to by p.
```

The value of n changes yet again—this time from 5 to 8. The effect of this statement is the same as `n = n + 3`. The computer finds the memory location pointed to by p and adds 3 to the value at that location.



*p = *p + 3;

To summarize, when p points to n, referring to *p has the same effect as referring to n. Here are more examples:

| WHEN P POINTS TO N, THIS STATEMENT | HAS THE SAME EFFECT AS THIS STATEMENT |
|---|---|
| *p = 33; | n = 33; |
| *p = *p + 2; | n = n + 2; |
| cout << *p; | cout << n; |
| cin >> *p; | cin >> n; |

But if using *p is the same as using n, why bother with *p in the first place? The answer is that (among other things) it achieves pass by reference—in which a function can change the value of an argument. Here's how it works:

- The caller of a function passes the address of a variable to be changed. For example, the caller passes &n (the address of n).
- The function has a pointer argument such as p that receives this address value. The function can then use *p to manipulate the value of n.

The next section shows a simple example that does just that.

## Example 6.1. *The Double-It Function*

Here's a program that uses a function named double_it, which doubles the value of a variable passed to it—or, more specifically, it doubles the value of a variable whose address is passed to it. That may sound a little convoluted, but the example should help make it clear.

**double_it.cpp**

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

void double_it(int *p);

int main() {
    int a = 5, b = 6;
    cout<< "Value of a before doubling is " << a << endl;
    cout<< "Value of b before doubling is " << b << endl;
    double_it(&a);     // Pass address of a.
    double_it(&b);     // Pass address of b.
    cout<< "Value of a after doubling is " << a << endl;
    cout<< "Value of b after doubling is " << b << endl;
    return 0;
}
void double_it(int *p) {
    *p = *p * 2;
}
```

### How It Works

This is a straightforward program. All the main function does is:

- Print the values of a and b.
- Call the double_it function to double the value of a, by passing the address of a (&a).
- Call the double_it function to double the value of b, by passing the address of b (&b).
- Print the values of a and b again.

This example needs pointers to work. You could write a version of double_it that took a simple **int** argument, but such a function would do nothing.

```
void double_it(int n) {    // THIS DOESN'T WORK!
    n = n * 2;
}
```

The problem here is that when an argument is passed to a function, the function gets a copy of the argument. But upon return, that copy is thrown away. But if the function gets the address of a variable, then it can use that address to make changes *to the original copy of the variable itself*.

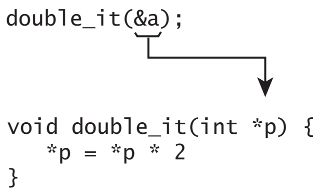
Here's an analogy. Getting a variable passed to you is like getting photocopies of a secret document. You can view the information, but you have no access to the originals. But getting a pointer is like getting the location and access codes for the original documents; you not only get to look at them, you can make changes!

So to enable a function to change the value of a variable, use pointers. This function does that by declaring an argument p, a pointer to an integer:

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

This declaration says that "the thing pointed to by p" has **int** type. Therefore, p itself is a pointer to an **int**.

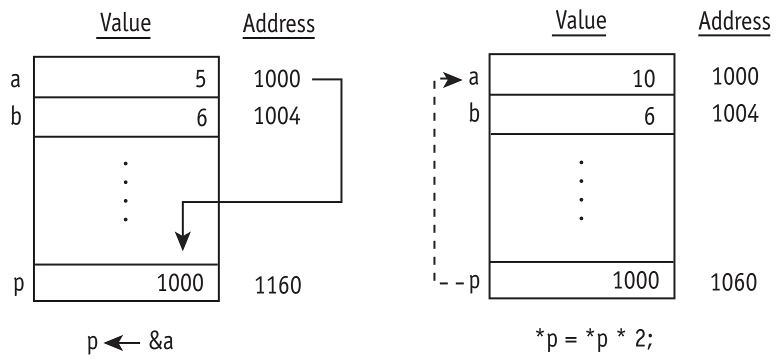
The caller must therefore pass an address, which it does by using the address operator (&).



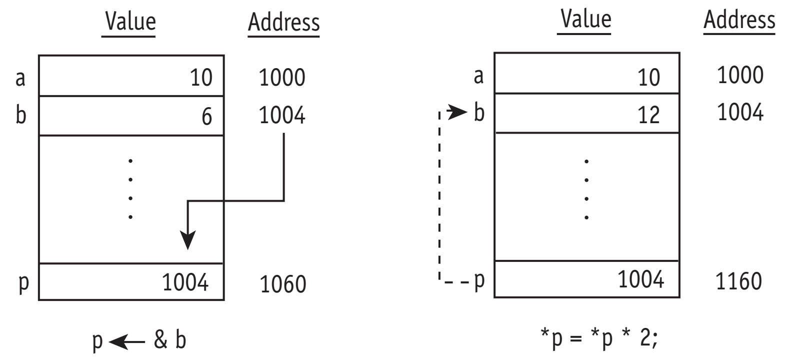
```
double_it(&a);


void double_it(int *p) {
   *p = *p * 2
}
```

Visually, here's the effect of these statements in terms of the memory layout. The address of a is passed to the function, which then uses it to change the value of a.



The program then calls the function again, this time passing the address of b. The function uses this address to change the value of b.



## EXERCISES

**Exercise 6.1.1.** Write a program that calls a function triple_it that takes the address of an **int** and triples the value pointed to. Test it by passing an argument n, which is initialized to 15. Print out the value of n before and after the function is called. (Hint: the function should look similar to double_it in Example 6.1, so you can use that code and make the necessary alterations. When calling the function, remember to pass &n.)

**Exercise 6.1.2.** Write a program with a function named convert_temp: the function takes the address of a variable of type **double** and applies Centigrade-to-Fahrenheit conversion. A variable that contains a Centigrade temperature should, after the function is called, contain the equivalent Fahrenheit temperature. Initialize a variable named temperature to 10.0 and print out the value before and after the function is called. Hint: the relevant formula is F = (C * 1.8) + 32.

## *Swap: Another Function Using Pointers*

The double_it function showcased in the last section is fine for illustrating some basic mechanics, but it's probably not something that would appear in a real program. Now I'll move onto a function called swap, which you should find much more useful.

Suppose you have two **int** variables and you want to swap their values. It's easy to do this with a third variable temp, whose function is to hold a temporary value.

```
temp = a;
a = b;
b = temp;
```

Now wouldn't this be useful to put into a function that you could call whenever you needed to? Yes, but as I explained earlier, unless the arguments are passed by reference, changes to the variables are ignored.

Here's another solution. This one uses pointers to pass by reference.

```
// Swap function.
// Swap the values pointed to by p1 and p2.
//
void swap(int *p1, int *p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

The expressions *p1 and *p2 are integers, and you can use them as you would any integer variables. The effect here is to swap the values pointed to by p1 and p2. Therefore, if you pass the addresses of two integer variables a and b, the values of those variables get swapped.

Literally, here's what the three statements inside the function do:

1. Declare a local variable named temp and initialize it to the value pointed to by p1.
2. Assign the value pointed to by p2 to the value pointed to by p1.
3. Assign the value temp to the value pointed by p2.

p1 and p2 are addresses, and they do not change. The data that's altered is the data *pointed to* by p1 and p2. This is easy to see with an example.

Assume that big and little are initialized to 100 and 1, respectively.

```
int big = 100;
int little = 1;
```

The following statement calls the swap function, passing the addresses of these two variables. Note the use of the address operator (&) here.

```
swap(&big, &little);
```

Now if you print the values of these variables, you'll see the values have been exchanged, so that now `big` contains 1 and `little` contains 100.

```
cout << "The value of big is now " << big << endl;
cout << "The value of little is now " << little;
```

Note that the memory addresses of `big` and `little` do not change. But the values *at* those addresses change. This is why the indirection operator (*) is often called the "at" operator. The statement `*p = 0` alters the value stored *at* address p.

## Example 6.2. *Array Sorter*

Now it's time to show the power of this swap function. First I need to clarify that pointers are not limited to pointing only to simple variables—although I used that terminology at first to keep things simple. An **int** pointer (for example) can point to any memory location that stores an **int** value. This means that it can point to elements of an array, as well as pointing to a variable.

For example, here the swap function is used to swap the values of two elements of an array named arr:

```
int arr[5] = {0, 10, 30, 25, 50};
swap(&arr[2], &arr[3]);
```

Why am I so proud of this fact? Because given the right procedure, you can use the swap function to sort all the values of an array.

Consider a typical array. Take a look at arr again—this time with the data jumbled around.

| | | | | |
|---|---|---|---|---|
| 30 | 25 | 0 | 50 | 10 |
| arr[0] | arr[1] | arr[2] | arr[3] | arr[4] |

Here's an obvious solution to the sorting problem. You can easily verify that it works.

1 Find the lowest value and put that value in arr[0].

2 Find the *next* lowest value and put that value in arr[1].

3 Continue in this manner until you get to the end.

Don't laugh. This simple, brute-strength approach is not as dumb as it seems. A slight refinement gives us the essence of the classic selection-sort algorithm, which is what I use here. Here's the more refined version, where a[] is the array and n is the number of elements.

```
For i = 0 to n – 2,
    Find the lowest value in the range a[i] to a[n – 1]
    If i is not equal to the index of the lowest value found,
        Swap a[i] and a[index_of_lowest]
```

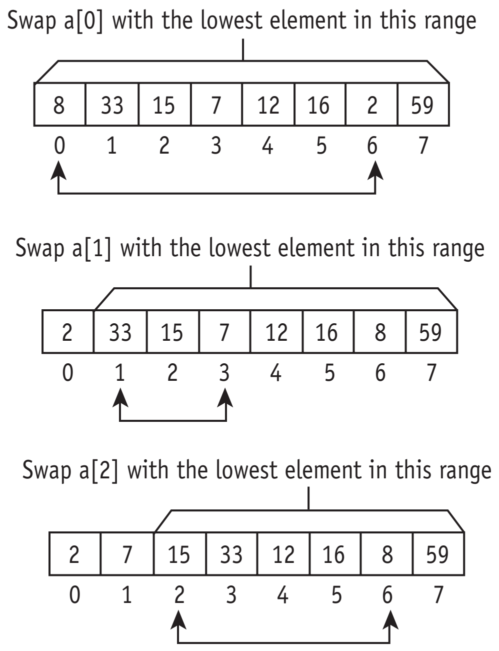
That's the basic plan. The effect will be to put the lowest value in a[0], the next lowest value in a[1], and so on. Note that by

```
For i = 0 to n – 2
```

I mean a **for** loop in which i is set to 0 during the first cycle of the loop, 1 during the next cycle of the loop, and so on, until i is set to n - 2, at which point it completes the last cycle. Each cycle of the loop places the correct element in a[i] and then increments i.

Inside the loop, a[i] is compared to a range that includes itself and all *the remaining elements* (the range a[i] to a[n - 1], which includes all elements on the *right*). By the time every value of i has been processed through the loop, the entire array will have been sorted.

Here's an example illustrating the first three cycles of the loop. The essence of the procedure is to compare each element in turn to all the elements on its right, swapping as needed.


Swap a[0] with the lowest element in this range
8 33 15 7 12 16 2 59
0 1 2 3 4 5 6 7
Swap a[1] with the lowest element in this range
2 33 15 7 12 16 8 59
0 1 2 3 4 5 6 7
Swap a[2] with the lowest element in this range
2 7 15 33 12 16 8 59
0 1 2 3 4 5 6 7

But how do we find the lowest value in the range a[i] to a[n - 1]? Remember we must be careful never to throw away an element, because we'll need it later. We need another algorithm.

What the following algorithm does is (1) start by assuming that i is the lowest element and so initialize "low" to i; and (2) whenever a lower element is found, this becomes the new "low" element.

**To find the lowest value in the range a[i] to a[n – 1]:**

```
Set low to i
```

```
For j = i + 1 to n – 1,
    If a[j] is less than a[low]
        Set low to j
```

This algorithm uses two additional **int** variables, j and low: j is another loop variable, and low is an integer that gets set to the index of the lowest element found so far. Whenever a lower element is found, the value of low gets updated.

We then combine the two algorithms together. After this, it's an easy matter to write the C++ code.

```
For i = 0 to n – 2,
    Set low to i
    For j = i + 1 to n – 1,
        If a[j] is less than a[low]
            Set low to j
    If i is not equal to low,
        Swap a[i] and a[low]
```

Here's the complete program that uses this algorithm to sort an array:

**sort.cpp**

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

void sort(int n);
void swap(int *p1, int *p2);
int a[10];

int main () {
    int i;
    for (i = 0; i < 10; i++) {
        cout << "Enter array element #" << i << ": ";
        cin >> a[i];
    }
    sort(10);
    cout << "Here are all the array elements, sorted:"
      << endl;
    for (i = 0; i < 10; i++)
        cout << a[i] << " "?;
    return 0;
}
```

▼ continued on next page

**sort.cpp, cont.**

```
// Sort array function: sort array named a, having n
// elements.
//
void sort (int n) {
    int i, j, low;
    for(i = 0; i < n - 1; i++) {
        // This part of the loop finds the lowest
        //  element in the range i to n-1; the index
        //  is set to the variable named low.
        low = i;
        for (j = i + 1; j < n; j++)
            if (a[j] < a[low])
                low = j;
        // This part of the loop performs a swap if
        //  needed.
        if (i != low)
            swap(&a[i], &a[low]);
    }
}

// Swap function.
// Swap the values pointed to by p1 and p2.
//
void swap(int *p1, int *p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

## How It Works

Only two parts of this example are directly relevant to understanding pointers. The first is the call to the swap function, which passes the addresses of a[i] and a[low]:

```
swap(&a[i], &a[low]);
```

An important point here is that you can use the address operator (&) to take the address of array elements, just as you can with variables.

The other part of the example that's relevant to pointer use is the function definition for swap, which I described in the previous section.

```
// Swap function.
// Swap the values pointed to by p1 and p2.
//
void swap(int *p1, int *p2) {
    int temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

As for the sort function, the key to understanding it is to note what each part of the main loop does. The main **for** loop successively sets i to 0, 1, 2 . . . up to and including n – 2. Why n – 2? Because by the time it gets to the last element (n – 1), all the sorting will have been done. (There is no need to compare the last element to itself.)

```
for(i = 0; i < n - 1; i++) {
    //...
}
```

The first part of the loop finds the lowest element in the range that includes a[i] and all the elements *to its right*. (Items on the left of a[i] are ignored, since they've already been sorted.) An inner loop conducts this search using a variable j, initialized to start at i + 1 (one position to the right of i).

```
low = i;
for (j = i + 1; j < n; j++)
    if (a[j] < a[low])
        low = j;
```

This, by the way, is an example of a *nested loop,* and it's completely legal. A **for** statement is just another kind of statement; therefore, it can be put inside another **if**, **while**, or **for** statement, to any degree of complexity.

The other part of the loop has an easy job. All it has to do is ask whether i differs from the index of the lowest element (stored in the variable "low"). Remember that the != operator means "not equal." There's no reason to do the swap if a[i] is already the lowest element in the range; that's the reason for the **if** condition here.

```
if (i != low)
    swap(&a[i], &a[low]);
```

## EXERCISES

**Exercise 6.2.1.** Rewrite the example so that instead of ordering the array from low to high, it sorts the array in reverse order: high to low. This is actually much easier than it may look. The new program must look for the highest value in each range. You should therefore rename the variable low as “high”. Otherwise, you need change only one statement; this statement involves a comparison. (Hint: that comparison is not part of a loop condition.)

**Exercise 6.2.2.** Rewrite the example so that it sorts an array with elements of type **double**. (This supports more flexibility for the user to enter values into the array.) It’s essential that you rewrite the swap function to work on data of the right type. But note that you should not change the type of any variables that serve as loop counters or array indexes—such variables should always have type **int**, regardless of the rest of the data.

## *Pointer Arithmetic*

Although pass-by-reference is the most obvious example, especially when you are first learning C++, pointers have a number of important uses. One of these is efficiently processing arrays. This is not an essential feature of the language, but it is often favored by programmers trying to write the tightest possible code.

Suppose you declare an array:

```
int arr[5] = {5, 15, 25, 35, 45};
```

The elements arr[0] through arr[4], of course, can all be used like individual integer variables. You can, for example, write statements such as `arr[1] = 10;`.

But what is the expression `arr` itself? Can `arr` ever appear by itself?

Yes: `arr` is a constant that translates into an address—specifically, the address of the first element. Because it’s a constant, you cannot change the value of `arr` itself. You can, however, use it to assign a value to a pointer variable:

```
int *p;
p = arr;
```

The statement `p = arr` is equivalent to this:

```
p = &arr[0];
```

So we’ve found a more concise, cleaner way to initialize a pointer to the address of the first element arr[0]. Is there a similar technique for the other elements? You betcha. For example, to assign p the address of arr[2]:

```
p = arr + 2;                // p = &arr[2];
```

In fact, under the covers, C++ interprets all array references as pointer references. A reference to arr[2] translates into:

```
*(arr + 2)
```

If you've been paying close attention all along, you may at first think this looks wrong. We add 2 to the address of the start of the array, arr. But arr is an **int** array, not an array of bytes. The element arr[2] is therefore not two but rather eight bytes away (four for each integer—assuming you are using a 32-bit system)! Yet this still works. Why?

It's because of *pointer arithmetic*. Only certain arithmetic operations are allowed on pointers and other address expressions (such as arr). These are:

- *address_expression + integer*
- *integer + address_expression*
- *address_expression - integer*
- *address_expression – address_expression*

When an integer and an address expression are added together, the result is another address expression. Before the calculation is completed, however, the integer is automatically *scaled* by the size of the base type. The C++ compiler performs this scaling for you.

*new_address = old_address + (integer * size_of_base_type)*

So, for example, if p has base type **int**, adding 2 to p has the effect of increasing it by 8—because 2 times the size of the base type (4 bytes) yields 8.

Scaling is an extremely convenient feature of C++ because it means that when a pointer p points to an element of an array and it is incremented by 1, this always has the effect of making p point to the next element:

```
p++;     // Point to next element in the array.
```

This in turn makes the code in the next section easier to write. It's also worth stating another cardinal rule. This is one of the most important things to remember when using pointers.

**When an integer value is added or subtracted from an address expression, the compiler automatically multiplies that integer by the size of the base type.**

Address expressions can also be compared to each other. Again, you should not make assumptions about a memory layout except where array elements are involved. The following expression is always true:

```
&arr[2] < &arr[3]
```

which is another way of saying that the following is always true, just as you'd expect:

```
arr + 2 < arr + 3
```

## *Pointers and Array Processing*

Because pointer arithmetic works the way it does, functions can access elements through pointer references rather than array indexing. The result is the same, but the pointer version (as I'll show) executes slightly faster.

In these days of incredibly fast CPUs, such minor speed increases make little difference for most programs. CPU efficiency was far more important in the 1970s, with its comparatively slow processors. CPU time was often at a premium.

But for a certain class of programs, the superior efficiency gained from C and C++ can still be useful. C and C++ are the languages of choice for people who write operating systems, and subroutines in an operating system or device driver may be called upon to execute thousands or even millions of times a second. In such cases, the small efficiencies due to pointer usage can actually matter.

Here's a function that uses direct pointer reference to zero out an integer array of size n elements.

```
void zero_out_array(int *p, int n) {
    while (n-- > 0) {       // Do n times:
        *p = 0;             //    Assign 0 to element
                            //     pointed to by p.
        p++;                //    Point to next element.
    }
}
```

This is a remarkably compact function, which would appear more compact still without the comments. (But remember that comments have no effect whatsoever on a program at runtime.)

Here's another version of the function using code that may look more familiar.

```
void zero_out_array2(int *arr, int n) {
    int i;
    for (i = 0; i < n; i++) {
        arr[i] = 0;
    }
}
```

But this version, while nearly as compact, runs a bit slower. Here's why: in the loop statement, the value of i must be scaled and added to arr each and every time through the loop, to get the location of the array element arr[i].

```
arr[i] = 0;
```

This in turn is equivalent to:

```
*(arr + i) = 0;
```

It's actually worse than that, because the scaling effect has to be done at run-time; so at the level of machine code the calculation is:

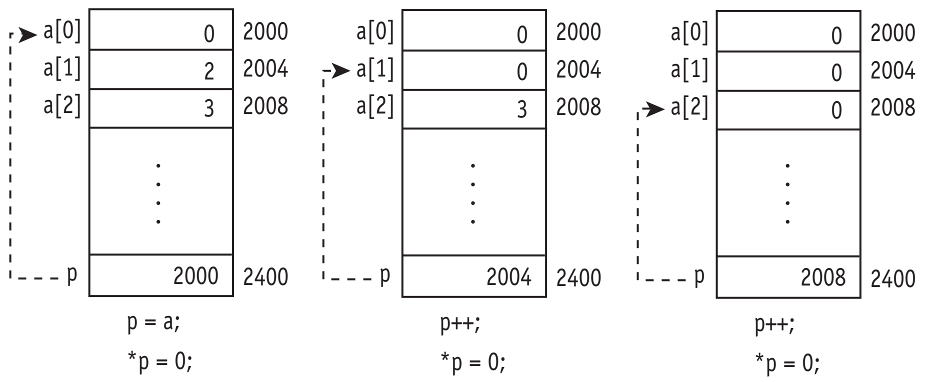
```
*(arr + (i * 4)) = 0;
```

The problem here is that the address has to be recalculated over and over again. In the direct-pointer version the address arr is only figured in once. The loop statement does less work:

```
*p = 0;
```

Of course, p has to be incremented each time through the loop; but both versions have a loop variable to update. Incrementing p is no more work than incrementing i.

Conceptually, here's how the direct-pointer version works. Each time through the loop, *p is set to 0 and then p itself is incremented to the next element. (Because of scaling, p is actually increased by 4 each time through the loop, but that's an easy operation.)



## Example 6.3. *Zero Out an Array*

Here's the zero_out_array function in the context of a complete example. All this program does is initialize an array, call the function, and then print the elements so that you can see that it worked. It's not very exciting in and of itself, but it does show how this kind of pointer usage works.

**zero_out.cpp**

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

void zero_out_array(int *arr, int n);
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

int main() {
    int i;
    zero_out_array(a, 10);
    // Print out all the elements of the array.
    for (i = 0; i < 10; i++)
        cout << a[i] << "  ";
    return 0;
}

// Zero-out-array function.
// Assign 0 to all elements of an int array of size n.
//
void zero_out_array(int *p, int n) {
    while (n-- > 0) {        // Do n times:
        *p = 0;              //    Assign 0 to element
                             //     pointed to by p.
        p++;                 //    Point to next element.
    }
}
```

## How It Works

I explained how the function zero_out_array works in the previous section, “Pointers and Array Processing.” Again, the key to understanding that function is to remember that adding 1 to a pointer makes it point to the next element of an array:

```
p++;
```

The other thing that’s notable about this program example is that it demonstrates how to pass an array in C++. The first argument to the function is the array name. Remember that usage of an array name translates into the address of the beginning of the array (i.e., the address of its first element).

```
    zero_out_array(a, 10);
```

To pass an array, therefore, just use the array name. The function must expect an address expression (i.e., a pointer) as the argument type. Here the function expects an argument of type **int***, so using the array name as the argument is correct behavior: as I mentioned, this statement passes the address of the first element.

This behavior may seem a little inconsistent if you're not used to it. Remember that passing a simple variable causes a copy of the value to be passed; passing an array name passes an address. For a simple array with base type **int**, the array name has type **int***. (A two-dimensional array name has type **int****.)

## Optimizing the Program

Strictly speaking, the technique described here does not optimize anything, in the sense of generating different instructions to be executed at runtime. You can, however, write even more compact code if you have a mind to.

As I've mentioned, the designers of C (the forerunner to C++) had an obsession for being able to write the most compact statements possible. This is why it's often possible to do multiple things in a single statement. This kind of programming can be dangerous unless you know what you're doing. Nevertheless. . . .

The **while** loop in the zero_out_array function does two things: zero-out an element and then increment the pointer so it points to the next element:

```
while (n-- > 0) {
    *p = 0;
    p++;
}
```

If you recall from past chapters, p++ is just an expression, and expressions can be used within larger expressions. That means we can combine the pointer-access and increment operations to produce:

```
while (n-- > 0)
    *p++ = 0;
```

What in the world does this do? To properly interpret *p++, I have to talk about two aspects of expression-evaluation that I've glossed over until now: precedence and associativity. Operators such as assignment (=) and test-for-equality (==) have relatively low precedence, meaning that they are applied only after other operations are resolved.

The pointer-indirection (*) and increment (++) operators have the same level of precedence as each other, but (unlike most operators) they associate right-to-left. Therefore, the statement `*p++ = 0;` is evaluated as if it were written this way:

```
*(p++) = 0;
```

This means: Increment pointer p but only after using its value in the operation

```
*p = 0;
```

Incidentally, using parentheses differently would produce an expression that is legal but not, in this case, useful.

```
(*p)++ = 0;  // Assign 0 to *p and then increment *p.
```

The effect of this statement would be to set the first array element to 0 and then to 1; p itself would never get incremented.

Whew! That's a lot of analysis required to understand a tiny piece of code. You're to be forgiven if you swear never to write such cryptic statements yourself. But now you're forearmed if you come across such code written by other C++ programmers.

**Note** Appendix A summarizes precedence and associativity for all C++ operators.

## EXERCISES

**Exercise 6.3.1.** Rewrite Example 6.3 to use direct-pointer reference for the loop that prints out the values of the array. Declare a pointer p and initialize it to start of the array. The loop condition should be p < a + 10.

**Exercise 6.3.2.** Write and test a copy_array function that copies the contents of one **int** array to another array of same size and base type. The function should take two pointer arguments. The operation inside the loop should be:

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

If you want to write more compact but cryptic code, you can use this statement:

```
*(p1++) = *(p2++);
```

or even the following, which means the same thing:

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

# Chapter 6 *Summary*

Several useful points were brought up in the discussion of pointers. These are summarized as follows:

- A pointer is a variable that can contain a numeric memory address. You can declare a pointer by using the following syntax:

  ```
  type *p;
  ```

- You can initialize a pointer by using the address operator (&):

  ```
  p = &n;          // Assign address of n to p.
  ```

- Once a pointer is initialized, you can use the indirection operator (*) to manipulate data pointed to by the pointer.

  ```
  p = &n;
  *p = 5;          // Assign 5 to n.
  ```

- To enable a function to manipulate data (pass by reference), pass an address.

  ```
  double_it(&n);
  ```

- To receive an address, declare an argument having pointer type.

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

- An array name is a constant that translates into the address of its first element.
- A reference to an array element a[n] is translated into the pointer reference:

  ```
  *(a + n)
  ```

- When an integer value is added to an address expression, C++ performs scaling, multiplying the integer by the size of the address expression's base type.

  ```
  new_address = address_expression + (integer *
  size_of_base_type)
  ```

- The unary operators * and ++ operators associate right-to-left. Consequently, this statement:

  ```
  *p++ = 0;
  ```

  does the same thing as the following statement, which sets *p to 0 and then increments the pointer p to point to the next element:

  ```
  *(p++) = 0;
  ```