12 Pointers and Arrays

Optimization hinders evolution.

Chapter 11 introduced pointers and showed how they're used as function arguments and as values returned by functions. This chapter covers another application for pointers. When pointers point to array elements, C allows us to perform arithmetic—addition and subtraction—on the pointers, which leads to an alternative way of processing arrays in which pointers take the place of array subscripts.

The relationship between pointers and arrays in C is a close one, as we'll soon see. We'll exploit this relationship in subsequent chapters, including Chapter 13 (Strings) and Chapter 17 (Advanced Uses of Pointers). Understanding the connection between pointers and arrays is critical for mastering C: it will give you insight into how C was designed and help you understand existing programs. Be aware, however, that one of the primary reasons for using pointers to process arrays—efficiency—is no longer as important as it once was, thanks to improved compilers.

Section 12.1 discusses pointer arithmetic and shows how pointers can be compared using the relational and equality operators. Section 12.2 then demonstrates how we can use pointer arithmetic for processing array elements. Section 12.3 reveals a key fact about arrays—an array name can serve as a pointer to the array's first element—and uses it to show how array arguments really work. Section 12.4 shows how the topics of the first three sections apply to multidimensional arrays. Section 12.5 wraps up the chapter by exploring the relationship between pointers and variable-length arrays, a C99 feature.

12.1 Pointer Arithmetic

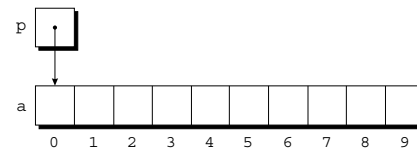
We saw in Section 11.5 that pointers can point to array elements. For example, suppose that `a` and `p` have been declared as follows:

```
int a[10], *p;
```

We can make p point to a[0] by writing

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

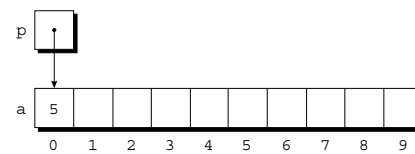
Graphically, here's what we've just done:



We can now access a[0] through p; for example, we can store the value 5 in a[0] by writing

```
*p = 5;
```

Here's our picture now:



Making a pointer p point to an element of an array a isn't particularly exciting. However, by performing ***pointer arithmetic*** (or ***address arithmetic***) on p, we can access the other elements of a. C supports three (and only three) forms of pointer arithmetic:

Adding an integer to a pointer
Subtracting an integer from a pointer
Subtracting one pointer from another

Let's take a close look at each of these operations. Our examples assume that the following declarations are in effect:

```
int a[10], *p, *q, i;
```

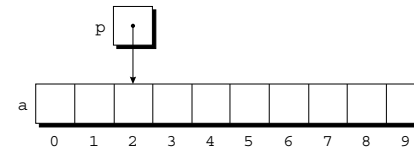
Adding an Integer to a Pointer

Adding an integer j to a pointer p yields a pointer to the element j places after the one that p points to. More precisely, if p points to the array element a[i], then p + j points to a[i+j] (provided, of course, that a[i+j] exists).

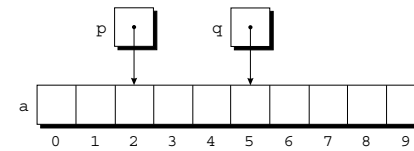
Q&A

The following example illustrates pointer addition; diagrams show the values of p and q at various points in the computation.

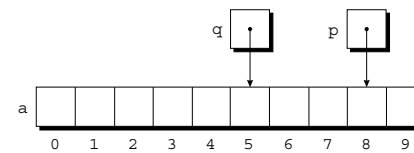
```
p = &a[2];
```




```
q = p + 3;
```



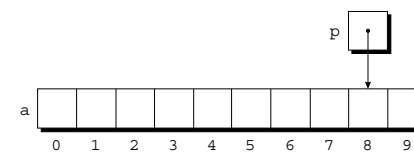

```
p += 6;
```



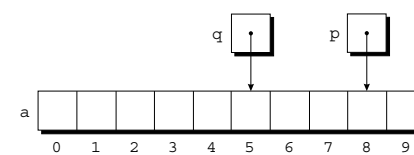
Subtracting an Integer from a Pointer

If `p` points to the array element `a[i]`, then `p - j` points to `a[i-j]`. For example:

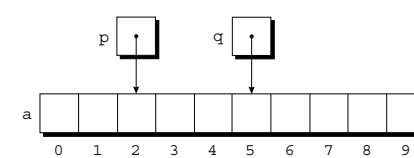
```
p = &a[8];
```




```
q = p - 3;
```



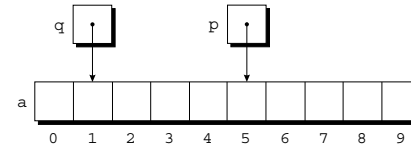

```
p -= 6;
```



Subtracting One Pointer from Another

When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers. Thus, if `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`. For example:

```
p = &a[5];
q = &a[1];
```




```
i = p - q;   /* i is 4 */
i = q - p;   /* i is -4 */
```

Performing arithmetic on a pointer that doesn't point to an array element causes undefined behavior. Furthermore, the effect of subtracting one pointer from another is undefined unless both point to elements of the *same* array.

Comparing Pointers

We can compare pointers using the relational operators (`<`, `<=`, `>`, `>=`) and the equality operators (`==` and `!=`). Using the relational operators to compare two pointers is meaningful only when both point to elements of the same array. The outcome of the comparison depends on the relative positions of the two elements in the array. For example, after the assignments

```
p = &a[5];
q = &a[1];
```

the value of `p <= q` is 0 and the value of `p >= q` is 1.

C99 Pointers to Compound Literals

compound literals ➤9.3

It's legal for a pointer to point to an element within an array created by a compound literal. A compound literal, you may recall, is a C99 feature that can be used to create an array with no name.

Consider the following example:

```
int *p = (int []){3, 0, 3, 4, 1};
```

`p` points to the first element of a five-element array containing the integers 3, 0, 3, 4, and 1. Using a compound literal saves us the trouble of first declaring an array variable and then making `p` point to the first element of that array:

```
int a[] = {3, 0, 3, 4, 1};
int *p = &a[0];
```

12.2 Using Pointers for Array Processing

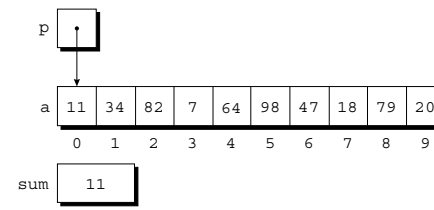
Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable. The following program fragment, which sums the elements of an array `a`, illustrates the technique. In this example, the pointer variable

`p` initially points to `a[0]`. Each time through the loop, `p` is incremented; as a result, it points to `a[1]`, then `a[2]`, and so forth. The loop terminates when `p` steps past the last element of `a`.

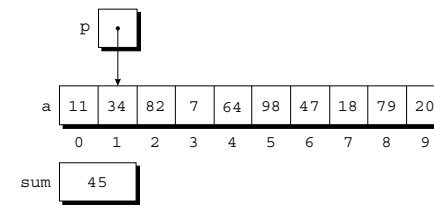
```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

The following figures show the contents of `a`, `sum`, and `p` at the end of the first three loop iterations (before `p` has been incremented).

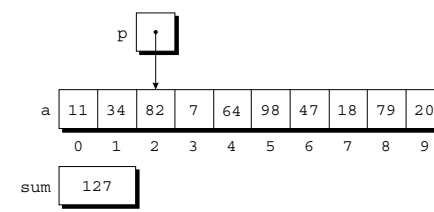
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



The condition `p < &a[N]` in the `for` statement deserves special mention. Strange as it may seem, it's legal to apply the address operator to `a[N]`, even though this element doesn't exist (`a` is indexed from 0 to `N` – 1). Using `a[N]` in this fashion is perfectly safe, since the loop doesn't attempt to examine its value. The body of the loop will be executed with `p` equal to `&a[0]`, `&a[1]`, …, `&a[N-1]`, but when `p` is equal to `&a[N]`, the loop terminates.

We could just as easily have written the loop without pointers, of course, using subscripting instead. The argument most often cited in support of pointer arithmetic is that it can save execution time. However, that depends on the implementation—some C compilers actually produce better code for loops that rely on subscripting.

Q&A

Combining the * and ++ Operators

C programmers often combine the * (indirection) and ++ operators in statements that process array elements. Consider the simple case of storing a value into an array element and then advancing to the next element. Using array subscripting, we might write

```
a[i++] = j;
```

If p is pointing to an array element, the corresponding statement would be

```
*p++ = j;
```

Because the postfix version of ++ takes precedence over *, the compiler sees this as

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

The value of p++ is p. (Since we're using the postfix version of ++, p won't be incremented until after the expression has been evaluated.) Thus, the value of *(p++) will be *p—the object to which p is pointing.

Of course, *p++ isn't the only legal combination of * and ++. We could write (*p)++, for example, which returns the value of the object that p points to, and then increments that object (p itself is unchanged). If you find this confusing, the following table may help:

Expression	*Meaning*
*p++ or *(p++)	Value of expression is *p before increment; increment p later
(*p)++	Value of expression is *p before increment; increment *p later
*++p or *(++p)	Increment p first; value of expression is *p after increment
++*p or ++(*p)	Increment *p first; value of expression is *p after increment

All four combinations appear in programs, although some are far more common than others. The one we'll see most frequently is *p++, which is handy in loops. Instead of writing

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

to sum the elements of the array a, we could write

```
p = &a[0];
while (p < &a[N])
  sum += *p++;
```

The * and -- operators mix in the same way as * and ++. For an application that combines * and --, let's return to the stack example of Section 10.2. The original version of the stack relied on an integer variable named top to keep track of the "top-of-stack" position in the contents array. Let's replace top by a pointer variable that points initially to element 0 of the contents array:

```
int *top_ptr = &contents[0];
```

Here are the new `push` and `pop` functions (updating the other stack functions is left as an exercise):

```
void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    *top_ptr++ = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return *--top_ptr;
}
```

Note that I've written `*--top_ptr`, not `*top_ptr--`, since I want `pop` to decrement `top_ptr` *before* fetching the value to which it points.

12.3 Using an Array Name as a Pointer

Pointer arithmetic is one way in which arrays and pointers are related, but it's not the only connection between the two. Here's another key relationship: *The name of an array can be used as a pointer to the first element in the array.* This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.

For example, suppose that `a` is declared as follows:

```
int a[10];
```

Using `a` as a pointer to the first element in the array, we can modify `a[0]`:

```
*a = 7;   /* stores 7 in a[0] */
```

We can modify `a[1]` through the pointer `a + 1`:

```
*(a+1) = 12;   /* stores 12 in a[1] */
```

In general, `a + i` is the same as `&a[i]` (both represent a pointer to element `i` of `a`) and `*(a+i)` is equivalent to `a[i]` (both represent element `i` itself). In other words, array subscripting can be viewed as a form of pointer arithmetic.

The fact that an array name can serve as a pointer makes it easier to write loops that step through an array. Consider the following loop from Section 12.2:

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

To simplify the loop, we can replace `&a[0]` by `a` and `&a[N]` by `a + N`:

idiom
```
for (p = a; p < a + N; p++)
  sum += *p;
```

Although an array name can be used as a pointer, it's not possible to assign it a new value. Attempting to make it point elsewhere is an error:

```
while (*a != 0)
  a++;            /*** WRONG ***/
```

This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:

```
p = a;
while (*p != 0)
  p++;
```

PROGRAM

Reversing a Series of Numbers (Revisited)

The `reverse.c` program of Section 8.1 reads 10 numbers, then writes the numbers in reverse order. As the program reads the numbers, it stores them in an array. Once all the numbers are read, the program steps through the array backwards as it prints the numbers.

The original program used subscripting to access elements of the array. Here's a new version in which I've replaced subscripting with pointer arithmetic.

reverse3.c
```
/* Reverses a series of numbers (pointer version) */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], *p;

  printf("Enter %d numbers: ", N);
  for (p = a; p < a + N; p++)
    scanf("%d", p);

  printf("In reverse order:");
  for (p = a + N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");

  return 0;
}
```

In the original program, an integer variable `i` kept track of the current position within the array. The new version replaces `i` with `p`, a pointer variable. The num-

bers are still stored in an array; we're simply using a different technique to keep track of where we are in the array.

Note that the second argument to `scanf` is `p`, not `&p`. Since `p` points to an array element, it's a satisfactory argument for `scanf`; `&p`, on the other hand, would be a pointer to a pointer to an array element.

Array Arguments (Revisited)

When passed to a function, an array name is always treated as a pointer. Consider the following function, which returns the largest element in an array of integers:

```
int find_largest(int a[], int n)
{
  int i, max;

  max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
```

Suppose that we call `find_largest` as follows:

```
largest = find_largest(b, N);
```

This call causes a pointer to the first element of `b` to be assigned to `a`; the array itself isn't copied.

The fact that an array argument is treated as a pointer has some important consequences:

- When an ordinary variable is passed to a function, its value is copied; any changes to the corresponding parameter don't affect the variable. In contrast, an array used as an argument isn't protected against change, since no copy is made of the array itself. For example, the following function (which we first saw in Section 9.3) modifies an array by storing zero into each of its elements:

  ```
  void store_zeros(int a[], int n)
  {
    int i;

    for (i = 0; i < n; i++)
      a[i] = 0;
  }
  ```

  To indicate that an array parameter won't be changed, we can include the word `const` in its declaration:

  ```
  int find_largest(const int a[], int n)
  {
    …
  }
  ```

If `const` is present, the compiler will check that no assignment to an element of `a` appears in the body of `find_largest`.

- The time required to pass an array to a function doesn't depend on the size of the array. There's no penalty for passing a large array, since no copy of the array is made.
- An array parameter can be declared as a pointer if desired. For example, `find_largest` could be defined as follows:

```
int find_largest(int *a, int n)
{
  …
}
```

Declaring `a` to be a pointer is equivalent to declaring it to be an array; the compiler treats the declarations as though they were identical.

Q&A

Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, the same isn't true for a *variable*. The declaration

```
int a[10];
```

causes the compiler to set aside space for 10 integers. In contrast, the declaration

```
int *a;
```

causes the compiler to allocate space for a pointer variable. In the latter case, `a` is not an array; attempting to use it as an array can have disastrous results. For example, the assignment

```
*a = 0;   /*** WRONG ***/
```

will store 0 where `a` is pointing. Since we don't know where `a` is pointing, the effect on the program is undefined.

- A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements. Suppose that we want `find_largest` to locate the largest element in some portion of an array `b`, say elements `b[5]`, …, `b[14]`. When we call `find_largest`, we'll pass it the address of `b[5]` and the number `10`, indicating that we want `find_largest` to examine 10 array elements, starting at `b[5]`:

```
largest = find_largest(&b[5], 10);
```

Using a Pointer as an Array Name

If we can use an array name as a pointer, will C allow us to subscript a pointer as though it were an array name? By now, you'd probably expect the answer to be yes, and you'd be right. Here's an example:

```
#define N 10
…
int a[N], i, sum = 0, *p = a;
…
for (i = 0; i < N; i++)
  sum += p[i];
```

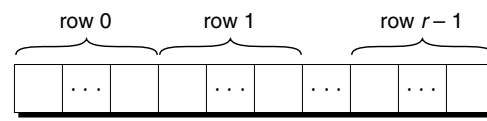
The compiler treats `p[i]` as `*(p+i)`, which is a perfectly legal use of pointer arithmetic. Although the ability to subscript a pointer may seem to be little more than a curiosity, we'll see in Section 17.3 that it's actually quite useful.

12.4 Pointers and Multidimensional Arrays

Just as pointers can point to elements of one-dimensional arrays, they can also point to elements of multidimensional arrays. In this section, we'll explore common techniques for using pointers to process the elements of multidimensional arrays. For simplicity, I'll stick to two-dimensional arrays, but everything we'll do applies equally to higher-dimensional arrays.

Processing the Elements of a Multidimensional Array

We saw in Section 8.2 that C stores two-dimensional arrays in row-major order; in other words, the elements of row 0 come first, followed by the elements of row 1, and so forth. An array with *r* rows would have the following appearance:



We can take advantage of this layout when working with pointers. If we make a pointer `p` point to the first element in a two-dimensional array (the element in row 0, column 0), we can visit every element in the array by incrementing `p` repeatedly.

As an example, let's look at the problem of initializing all elements of a two-dimensional array to zero. Suppose that the array has been declared as follows:

```
int a[NUM_ROWS][NUM_COLS];
```

The obvious technique would be to use nested `for` loops:

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
  for (col = 0; col < NUM_COLS; col++)
    a[row][col] = 0;
```

But if we view a as a one-dimensional array of integers (which is how it's stored), we can replace the pair of loops by a single loop:

```
int *p;
…
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```

The loop begins with p pointing to a[0][0]. Successive increments of p make it point to a[0][1], a[0][2], a[0][3], and so on. When p reaches a[0][NUM_COLS-1] (the last element in row 0), incrementing it again makes p point to a[1][0], the first element in row 1. The process continues until p goes past a[NUM_ROWS-1][NUM_COLS-1], the last element in the array.

Q&A

Although treating a two-dimensional array as one-dimensional may seem like cheating, it works with most C compilers. Whether it's a good idea to do so is another matter. Techniques like this one definitely hurt program readability, but—at least with some older compilers—produce a compensating increase in efficiency. With many modern compilers, though, there's often little or no speed advantage.

Processing the Rows of a Multidimensional Array

What about processing the elements in just one *row* of a two-dimensional array? Again, we have the option of using a pointer variable p. To visit the elements of row i, we'd initialize p to point to element 0 in row i in the array a:

```
p = &a[i][0];
```

Or we could simply write

```
p = a[i];
```

since, for any two-dimensional array a, the expression a[i] is a pointer to the first element in row i. To see why this works, recall the magic formula that relates array subscripting to pointer arithmetic: for any array a, the expression a[i] is equivalent to *(a + i). Thus, &a[i][0] is the same as &(*(a[i] + 0)), which is equivalent to &*a[i], which is the same as a[i], since the & and * operators cancel. We'll use this simplification in the following loop, which clears row i of the array a:

```
int a[NUM_ROWS][NUM_COLS], *p, i;
…
for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

Since a[i] is a pointer to row i of the array a, we can pass a[i] to a function that's expecting a one-dimensional array as its argument. In other words, a function that's designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array. As a result, functions such as

`find_largest` and `store_zeros` are more versatile than you might expect. Consider `find_largest`, which we originally designed to find the largest element of a one-dimensional array. We can just as easily use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:

```
largest = find_largest(a[i], NUM_COLS);
```

Processing the Columns of a Multidimensional Array

Processing the elements in a *column* of a two-dimensional array isn't as easy, because arrays are stored by row, not by column. Here's a loop that clears column `i` of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;
…
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

I've declared `p` to be a pointer to an array of length `NUM_COLS` whose elements are integers. The parentheses around `*p` in `(*p)[NUM_COLS]` are required; without them, the compiler would treat `p` as an array of pointers instead of a pointer to an array. The expression `p++` advances `p` to the beginning of the next row. In the expression `(*p)[i]`, `*p` represents an entire row of `a`, so `(*p)[i]` selects the element in column `i` of that row. The parentheses in `(*p)[i]` are essential, because the compiler would interpret `*p[i]` as `*(p[i])`.

Using the Name of a Multidimensional Array as a Pointer

Just as the name of a one-dimensional array can be used as a pointer, so can the name of *any* array, regardless of how many dimensions it has. Some care is required, though. Consider the following array:

```
int a[NUM_ROWS][NUM_COLS];
```

`a` is *not* a pointer to `a[0][0]`; instead, it's a pointer to `a[0]`. This makes more sense if we look at it from the standpoint of C, which regards `a` not as a two-dimensional array but as a one-dimensional array whose elements are one-dimensional arrays. When used as a pointer, `a` has type `int (*)[NUM_COLS]` (pointer to an integer array of length `NUM_COLS`).

Knowing that `a` points to `a[0]` is useful for simplifying loops that process the elements of a two-dimensional array. For example, instead of writing

```
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

to clear column `i` of the array `a`, we can write

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

Another situation in which this knowledge comes in handy is when we want to "trick" a function into thinking that a multidimensional array is really one-dimensional. For example, consider how we might use `find_largest` to find the largest element in `a`. As the first argument to `find_largest`, let's try passing `a` (the address of the array); as the second, we'll pass `NUM_ROWS * NUM_COLS` (the total number of elements in `a`):

```
largest = find_largest(a, NUM_ROWS * NUM_COLS);   /* WRONG */
```

Unfortunately, the compiler will object to this statement, because the type of `a` is `int (*)[NUM_COLS]` but `find_largest` is expecting an argument of type `int *`. The correct call is

```
largest = find_largest(a[0], NUM_ROWS * NUM_COLS);
```

Q&A

`a[0]` points to element 0 in row 0, and it has type `int *` (after conversion by the compiler), so the latter call will work correctly.

12.5 Pointers and Variable-Length Arrays (C99)

variable-length arrays ➤8.3

Pointers are allowed to point to elements of variable-length arrays (VLAs), a feature of C99. An ordinary pointer variable would be used to point to an element of a one-dimensional VLA:

```
void f(int n)
{
  int a[n], *p;
  p = a;
  …
}
```

When the VLA has more than one dimension, the type of the pointer depends on the length of each dimension except for the first. Let's look at the two-dimensional case:

```
void f(int m, int n)
{
  int a[m][n], (*p)[n];
  p = a;
  …
}
```

Since the type of `p` depends on `n`, which isn't constant, `p` is said to have a ***variably modified type.*** Note that the validity of an assignment such as `p = a` can't always be determined by the compiler. For example, the following code will compile but is correct only if `m` and `n` are equal:

```
int a[m][n], (*p)[m];
p = a;
```

If `m` ≠ `n`, any subsequent use of `p` will cause undefined behavior.

Variably modified types are subject to certain restrictions, just as variable-length arrays are. The most important restriction is that the declaration of a variably modified type must be inside the body of a function or in a function prototype.

Pointer arithmetic works with VLAs just as it does for ordinary arrays. Returning to the example of Section 12.4 that clears a single column of a two-dimensional array `a`, let's declare `a` as a VLA this time:

```
int a[m][n];
```

A pointer capable of pointing to a row of `a` would be declared as follows:

```
int (*p)[n];
```

The loop that clears column `i` is almost identical to the one we used in Section 12.4:

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

Q & A

Q: I don't understand pointer arithmetic. If a pointer is an address, does that mean that an expression like `p + j` adds `j` to the address stored in `p`? [p. 258]

A: No. Integers used in pointer arithmetic are scaled depending on the type of the pointer. If `p` is of type `int *`, for example, then `p + j` typically adds 4 × `j` to `p`, assuming that `int` values are stored using 4 bytes. But if `p` has type `double *`, then `p + j` will probably add 8 × `j` to `p`, since `double` values are usually 8 bytes long.

Q: When writing a loop to process an array, is it better to use array subscripting or pointer arithmetic? [p. 261]

A: There's no easy answer to this question, since it depends on the machine you're using and the compiler itself. In the early days of C on the PDP-11, pointer arithmetic yielded a faster program. On today's machines, using today's compilers, array subscripting is often just as good, and sometimes even better. The bottom line: Learn both ways and then use whichever is more natural for the kind of program you're writing.

***Q: I read somewhere that `i[a]` is the same as `a[i]`. Is this true?**

A: Yes, it is, oddly enough. The compiler treats `i[a]` as `*(i + a)`, which is the same as `*(a + i)`. (Pointer addition, like ordinary addition, is commutative.) But `*(a + i)` is equivalent to `a[i]`. Q.E.D. But please don't use `i[a]` in programs unless you're planning to enter the next Obfuscated C contest.

Q: Why is `*a` the same as `a[]` in a parameter declaration? [p. 266]

A: Both indicate that the argument is expected to be a pointer. The same operations on `a` are possible in both cases (pointer arithmetic and array subscripting, in particular). And, in both cases, `a` itself can be assigned a new value within the function. (Although C allows us to use the name of an array *variable* only as a "constant pointer," there's no such restriction on the name of an array *parameter*.)

Q: Is it better style to declare an array parameter as `*a` or `a[]`?

A: That's a tough one. From one standpoint, `a[]` is the obvious choice, since `*a` is ambiguous (does the function want an array of objects or a pointer to a single object?). On the other hand, many programmers argue that declaring the parameter as `*a` is more accurate, since it reminds us that only a pointer is passed, not a copy of the array. Others switch between `*a` and `a[]`, depending on whether the function uses pointer arithmetic or subscripting to access the elements of the array. (That's the approach I'll use.) In practice, `*a` is more common than `a[]`, so you'd better get used to it. For what it's worth, Dennis Ritchie now refers to the `a[]` notation as "a living fossil" that "serves as much to confuse the learner as to alert the reader."

Q: We've seen that arrays and pointers are closely related in C. Would it be accurate to say that they're interchangeable?

A: No. It's true that array *parameters* are interchangeable with pointer parameters, but array *variables* aren't the same as pointer variables. Technically, the name of an array isn't a pointer; rather, the C compiler *converts* it to a pointer when necessary. To see this difference more clearly, consider what happens when we apply the `sizeof` operator to an array `a`. The value of `sizeof(a)` is the total number of bytes in the array—the size of each element multiplied by the number of elements. But if `p` is a pointer variable, `sizeof(p)` is the number of bytes required to store a pointer value.

Q: You said that treating a two-dimensional array as one-dimensional works with "most" C compilers. Doesn't it work with all compilers? [p. 268]

A: No. Some modern "bounds-checking" compilers track not only the type of a pointer, but—when it points to an array—also the length of the array. For example, suppose that `p` is assigned a pointer to `a[0][0]`. Technically, `p` points to the first element of `a[0]`, a one-dimensional array. If we increment `p` repeatedly in an effort to visit all the elements of `a`, we'll go out of bounds once `p` goes past the last element of `a[0]`. A compiler that performs bounds-checking may insert code to check that `p` is used only to access elements in the array pointed to by `a[0]`; an attempt to increment `p` past the end of this array would be detected as an error.

Q: If `a` is a two-dimensional array, why can we pass `a[0]`—but not `a` itself—to `find_largest`? Don't both `a` and `a[0]` point to the same place (the beginning of the array)? [p. 270]

A: They do, as a matter of fact—both point to element `a[0][0]`. The problem is that

`a` has the wrong type. When used as an argument, it's a pointer to an array, but `find_largest` is expecting a pointer to an integer. However, `a[0]` has type `int *`, so it's an acceptable argument for `find_largest`. This concern about types is actually good; if C weren't so picky, we could make all kinds of horrible pointer mistakes without the compiler noticing.

Exercises

Section 12.1

1. Suppose that the following declarations are in effect:

```
int a[] = {5, 15, 34, 54, 14, 2, 52, 72};
int *p = &a[1], *q = &a[5];
```

(a) What is the value of `*(p+3)`?
(b) What is the value of `*(q-3)`?
(c) What is the value of `q - p`?
(d) Is the condition `p < q` true or false?
(e) Is the condition `*p < *q` true or false?

W *2. Suppose that `high`, `low`, and `middle` are all pointer variables of the same type, and that `low` and `high` point to elements of an array. Why is the following statement illegal, and how could it be fixed?

```
middle = (low + high) / 2;
```

Section 12.2

3. What will be the contents of the `a` array after the following statements are executed?

```
#define N 10

int a[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *p = &a[0], *q = &a[N-1], temp;

while (p < q) {
  temp = *p;
  *p++ = *q;
  *q-- = temp;
}
```

W 4. Rewrite the `make_empty`, `is_empty`, and `is_full` functions of Section 10.2 to use the pointer variable `top_ptr` instead of the integer variable `top`.

Section 12.3

5. Suppose that `a` is a one-dimensional array and `p` is a pointer variable. Assuming that the assignment `p = a` has just been performed, which of the following expressions are illegal because of mismatched types? Of the remaining expressions, which are true (have a nonzero value)?

(a) `p == a[0]`
(b) `p == &a[0]`
(c) `*p == a[0]`
(d) `p[0] == a[0]`

W 6. Rewrite the following function to use pointer arithmetic instead of array subscripting. (In other words, eliminate the variable `i` and all uses of the `[]` operator.) Make as few changes as possible.

```
int sum_array(const int a[], int n)
{
  int i, sum;

  sum = 0;
  for (i = 0; i < n; i++)
    sum += a[i];
  return sum;
}
```

7. Write the following function:

   ```
   bool search(const int a[], int n, int key);
   ```

   `a` is an array to be searched, `n` is the number of elements in the array, and `key` is the search key. `search` should return `true` if `key` matches some element of `a`, and `false` if it doesn't. Use pointer arithmetic—not subscripting—to visit array elements.

8. Rewrite the following function to use pointer arithmetic instead of array subscripting. (In other words, eliminate the variable `i` and all uses of the `[]` operator.) Make as few changes as possible.

   ```
   void store_zeros(int a[], int n)
   {
     int i;

     for (i = 0; i < n; i++)
       a[i] = 0;
   }
   ```

9. Write the following function:

   ```
   double inner_product(const double *a, const double *b,
                        int n);
   ```

   `a` and `b` both point to arrays of length `n`. The function should return `a[0] * b[0] + a[1] * b[1] + ... + a[n-1] * b[n-1]`. Use pointer arithmetic—not subscripting—to visit array elements.

10. Modify the `find_middle` function of Section 11.5 so that it uses pointer arithmetic to calculate the return value.

11. Modify the `find_largest` function so that it uses pointer arithmetic—not subscripting—to visit array elements.

12. Write the following function:

    ```
    void find_two_largest(const int *a, int n, int *largest,
                          int *second_largest);
    ```

    `a` points to an array of length `n`. The function searches the array for its largest and second-largest elements, storing them in the variables pointed to by `largest` and `second_largest`, respectively. Use pointer arithmetic—not subscripting—to visit array elements.

Section 12.4

W 13. Section 8.2 had a program fragment in which two nested `for` loops initialized the array `ident` for use as an identity matrix. Rewrite this code, using a single pointer to step through the array one element at a time. *Hint:* Since we won't be using `row` and `col` index variables, it won't be easy to tell where to store 1. Instead, we can use the fact that the first element of the array should be 1, the next `N` elements should be 0, the next element should

be 1, and so forth. Use a variable to keep track of how many consecutive 0s have been stored; when the count reaches `N`, it's time to store 1.

14. Assume that the following array contains a week's worth of hourly temperature readings, with each row containing the readings for one day:

```
int temperatures[7][24];
```

Write a statement that uses the `search` function (see Exercise 7) to search the entire `temperatures` array for the value 32.

W 15. Write a loop that prints all temperature readings stored in row `i` of the `temperatures` array (see Exercise 14). Use a pointer to visit each element of the row.

16. Write a loop that prints the highest temperature in the `temperatures` array (see Exercise 14) for each day of the week. The loop body should call the `find_largest` function, passing it one row of the array at a time.

17. Rewrite the following function to use pointer arithmetic instead of array subscripting. (In other words, eliminate the variables `i` and `j` and all uses of the `[]` operator.) Use a single loop instead of nested loops.

```
int sum_two_dimensional_array(const int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```

18. Write the `evaluate_position` function described in Exercise 13 of Chapter 9. Use pointer arithmetic—not subscripting—to visit array elements. Use a single loop instead of nested loops.

Programming Projects

W 1. (a) Write a program that reads a message, then prints the reversal of the message:

```
Enter a message: Don't get mad, get even.
Reversal is: .neve teg ,dam teg t'noD
```

Hint: Read the message one character at a time (using `getchar`) and store the characters in an array. Stop reading when the array is full or the character read is `'\n'`.

(b) Revise the program to use a pointer instead of an integer to keep track of the current position in the array.

2. (a) Write a program that reads a message, then checks whether it's a palindrome (the letters in the message are the same from left to right as from right to left):

```
Enter a message: He lived as a devil, eh?
Palindrome

Enter a message: Madam, I am Adam.
Not a palindrome
```

Ignore all characters that aren't letters. Use integer variables to keep track of positions in the array.

(b) Revise the program to use pointers instead of integers to keep track of positions in the array.

W 3. Simplify Programming Project 1(b) by taking advantage of the fact that an array name can be used as a pointer.

4. Simplify Programming Project 2(b) by taking advantage of the fact that an array name can be used as a pointer.

5. Modify Programming Project 14 from Chapter 8 so that it uses a pointer instead of an integer to keep track of the current position in the array that contains the sentence.

6. Modify the `qsort.c` program of Section 9.6 so that `low`, `high`, and `middle` are pointers to array elements rather than integers. The `split` function will need to return a pointer, not an integer.

7. Modify the `maxmin.c` program of Section 11.4 so that the `max_min` function uses a pointer instead of an integer to keep track of the current position in the array.