ELEC S212

Network Programming and Design

Unit 5

Readings

香港公開大學
THE OPEN UNIVERSITY OF HONG KONG

科技學院 School of Science and Technology

Readings

Copyright acknowledgements

Printed July 2018.
Scanned September 2018.

5.3 Kochan, S G (2005) *Programming in C*, 3rd edn, Indianapolis: Sam's Publishing, 440–42.

5.1 Summary of C Operators

Table A.5 summarizes the various operators in the C language. These operators are listed in order of decreasing precedence. Operators grouped together have the same precedence.

Table A.5 **Summary of C Operators**

Operator	Description	Associativity
`()`	Function call	
`[]`	Array element reference	
`->`	Pointer to structure member reference	Left to right
`.`	Structure member reference	
`-`	Unary minus	
`+`	Unary plus	
`++`	Increment	
`--`	Decrement	
`!`	Logical negation	
`~`	Ones complement	Right to left
`*`	Pointer reference (indirection)	
`&`	Address	
`sizeof`	Size of an object	
`(`*`type`*`)`	Type cast (conversion)	
`*`	Multiplication	
`/`	Division	Left to right
`%`	Modulus	
`+`	Addition	Left to right
`-`	Subtraction	
`<<`	Left shift	Left to right
`>>`	Right shift	
`<`	Less than	
`<=`	Less than or equal to	Left to right
`>`	Greater than	
`=>`	Greater than or equal to	

Table A.5 **Continued**

Operator	Description	Associativity
`==`	Equality	Left to right
`!=`	Inequality	
`&`	Bitwise AND	Left to right
`^`	Bitwise XOR	Left to right
`|`	Bitwise OR	Left to right
`&&`	Logical AND	Left to right
`||`	Logical OR	Left to right
`?:`	Conditional	Right to left
`=` `*= /= %=` `+= -= &=` `^= |=` `<<= >>=`	Assignment operators	Right to left
`,`	Comma operator	Right to left

As an example of how to use Table A.5, consider the following expression:

```
b | c & d * e
```

The multiplication operator has higher precedence than both the bitwise OR and bitwise AND operators because it appears above both of these in Table A.5. Similarly, the bitwise AND operator has higher precedence than the bitwise OR operator because the former appears above the latter in the table. Therefore, this expression is evaluated as

```
b | ( c & ( d * e ) )
```

Now consider the following expression:

```
b % c * d
```

Because the modulus and multiplication operator appear in the same grouping in Table A.5, they have the same precedence. The associativity listed for these operators is left to right, indicating that the expression is evaluated as

```
( b % c ) * d
```

As another example, the expression

```
++a->b
```

is evaluated as

```
++(a->b)
```

because the `->` operator has higher precedence than the `++` operator.

Finally, because the assignment operators group from right to left, the statement

```
a = b = 0;
```

is evaluated as

```
a = (b = 0);
```

which has the net result of setting the values of `a` and `b` to `0`. In the case of the expression

```
x[i] + ++i
```

it is not defined whether the compiler evaluates the left side of the plus operator or the right side first. Here, the way that it's done affects the result because the value of `i` might be incremented before `x[i]` is evaluated.

Another case in which the order of evaluation is not defined is in the following expression:

```
x[i] = ++i
```

In this situation, it is not defined whether the value of `i` is incremented before or after its value is used to index into `x`.

The order of evaluation of function arguments is also undefined. Therefore, in the function call

```
f (i, ++i);
```

`i` might be incremented first, thereby causing the same value to be sent as the two arguments to the function.

The C language guarantees that the `&&` and `||` operators are evaluated from left to right. Furthermore, in the case of `&&`, it is guaranteed that the second operand is not evaluated if the first is `0`; and in the case of `||`, it is guaranteed that the second operand is not evaluated if the first is nonzero. This fact is worth bearing in mind when forming expressions such as

```
if ( dataFlag  ||  checkData (myData) )
  ...
```

because, in this case, `checkData` is called only if the value of `dataFlag` is `0`. To take another example, if the array `a` is defined to contain `n` elements, the statement that begins

```
if (index >= 0  &&  index < n  &&  a[index] == 0))
  ...
```

references the element contained in the array only if `index` is a valid subscript in the array.

5.4 Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 64–65.

3.7 Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing at the top or bottom. The `break` statement provides an early exit from `for`, `while`, and `do`, just as from `switch`. A `break` causes the innermost enclosing loop or `switch` to be exited immediately.

The following function, `trim`, removes trailing blanks, tabs, and newlines from the end of a string, using a `break` to exit from a loop when the rightmost non-blank, non-tab, non-newline is found.

```
/* trim:  remove trailing blanks, tabs, newlines */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

`strlen` returns the length of the string. The `for` loop starts at the end and scans backwards looking for the first character that is not a blank or tab or newline. The loop is broken when one is found, or when `n` becomes negative (that is, when the entire string has been scanned). You should verify that this is correct behavior even when the string is empty or contains only white space characters.

The `continue` statement is related to `break`, but less often used; it causes the next iteration of the enclosing `for`, `while`, or `do` loop to begin. In the `while` and `do`, this means that the test part is executed immediately; in the `for`, control passes to the increment step. The `continue` statement applies only to loops, not to `switch`. A `continue` inside a `switch` inside a loop causes the next loop iteration.

As an example, this fragment processes only the non-negative elements in the array `a`; negative values are skipped.

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)   /* skip negative elements */
        continue;
    ...   /* do positive elements */
}
```

The `continue` statement is often used when the part of the loop that follows is complicated, so that reversing a test and indenting another level would nest the program too deeply.

5.5 Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 86–88.

4.10 Recursion

C functions may be used recursively; that is, a function may call itself either directly or indirectly. Consider printing a number as a character string. As we mentioned before, the digits are generated in the wrong order: low-order digits are available before high-order digits, but they have to be printed the other way around.

There are two solutions to this problem. One is to store the digits in an array as they are generated, then print them in the reverse order, as we did with `itoa` in Section 3.6. The alternative is a recursive solution, in which `printd` first calls itself to cope with any leading digits, then prints the trailing digit. Again, this version can fail on the largest negative number.

```
#include <stdio.h>

/* printd:  print n in decimal */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    putchar(n % 10 + '0');
}
```

When a function calls itself recursively, each invocation gets a fresh set of all the automatic variables, independent of the previous set. Thus in `printd(123)` the first `printd` receives the argument `n = 123`. It passes `12` to a second `printd`, which in turn passes `1` to a third. The third-level `printd` prints `1`, then returns to the second level. That `printd` prints `2`, then returns to the first level. That one prints 3 and terminates.

Another good example of recursion is quicksort, a sorting algorithm developed by C. A. R. Hoare in 1962. Given an array, one element is chosen and the others are partitioned into two subsets—those less than the partition element and those greater than or equal to it. The same process is then applied recursively to the two subsets. When a subset has fewer than two elements, it doesn't need any sorting; this stops the recursion.

Our version of quicksort is not the fastest possible, but it's one of the simplest. We use the middle element of each subarray for partitioning.

```
/* qsort:  sort v[left]...v[right] into increasing order */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right)    /* do nothing if array contains */
        return;           /* fewer than two elements */
    swap(v, left, (left + right)/2); /* move partition elem */
    last = left;                     /* to v[0] */
    for (i = left+1; i <= right; i++)   /* partition */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);          /* restore partition elem */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

We moved the swapping operation into a separate function `swap` because it occurs three times in `qsort`.

```
/* swap:  interchange v[i] and v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

The standard library includes a version of `qsort` that can sort objects of any type.

Recursion may provide no saving in storage, since somewhere a stack of the values being processed must be maintained. Nor will it be faster. But recursive code is more compact, and often much easier to write and understand than the non-recursive equivalent. Recursion is especially convenient for recursively defined data structures like trees; we will see a nice example in Section 6.5.

Exercise 4-12. Adapt the ideas of `printd` to write a recursive version of `itoa`; that is, convert an integer into a string by calling a recursive routine. □

Exercise 4-13. Write a recursive version of the function `reverse(s)`, which reverses the string `s` in place. □

5.8 Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 93–97.

CHAPTER 5: **Pointers and Arrays**

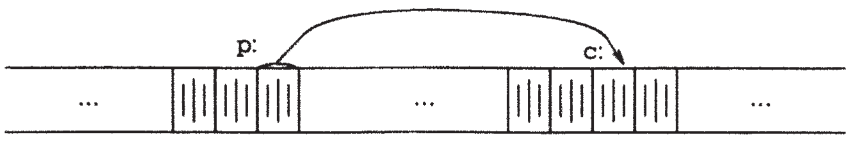
A pointer is a variable that contains the address of a variable. Pointers are much used in C, partly because they are sometimes the only way to express a computation, and partly because they usually lead to more compact and efficient code than can be obtained in other ways. Pointers and arrays are closely related; this chapter also explores this relationship and shows how to exploit it.

Pointers have been lumped with the `goto` statement as a marvelous way to create impossible-to-understand programs. This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected. With discipline, however, pointers can also be used to achieve clarity and simplicity. This is the aspect that we will try to illustrate.

The main change in ANSI C is to make explicit the rules about how pointers can be manipulated, in effect mandating what good programmers already practice and good compilers already enforce. In addition, the type `void *` (pointer to `void`) replaces `char *` as the proper type for a generic pointer.

5.1 Pointers and Addresses

Let us begin with a simplified picture of how memory is organized. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a `char`, a pair of one-byte cells can be treated as a `short` integer, and four adjacent bytes form a `long`. A pointer is a group of cells (often two or four) that can hold an address. So if `c` is a `char` and `p` is a pointer that points to it, we could represent the situation this way:



The unary operator `&` gives the address of an object, so the statement

```
p = &c;
```

assigns the address of c to the variable p, and p is said to "point to" c. The & operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

The unary operator * is the *indirection* or *dereferencing* operator; when applied to a pointer, it accesses the object the pointer points to. Suppose that x and y are integers and ip is a pointer to int. This artificial sequence shows how to declare a pointer and how to use & and *:

```
int x = 1, y = 2, z[10];
int *ip;            /* ip is a pointer to int */

ip = &x;            /* ip now points to x */
y = *ip;            /* y is now 1 */
*ip = 0;            /* x is now 0 */
ip = &z[0];         /* ip now points to z[0] */
```

The declarations of x, y, and z are what we've seen all along. The declaration of the pointer ip,

```
int *ip;
```

is intended as a mnemonic; it says that the expression *ip is an int. The syntax of the declaration for a variable mimics the syntax of expressions in which the variable might appear. This reasoning applies to function declarations as well. For example,

```
double *dp, atof(char *);
```

says that in an expression *dp and atof(s) have values of type double, and that the argument of atof is a pointer to char.

You should also note the implication that a pointer is constrained to point to a particular kind of object: every pointer points to a specific data type. (There is one exception: a "pointer to void" is used to hold any type of pointer but cannot be dereferenced itself. We'll come back to it in Section 5.11.)

If ip points to the integer x, then *ip can occur in any context where x could, so

```
*ip = *ip + 10;
```

increments *ip by 10.

The unary operators * and & bind more tightly than arithmetic operators, so the assignment

```
y = *ip + 1
```

takes whatever ip points at, adds 1, and assigns the result to y, while

```
*ip += 1
```

increments what ip points to, as do

```
++*ip
```

and

```
(*ip)++
```

The parentheses are necessary in this last example; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if iq is another pointer to int,

```
iq = ip
```

copies the contents of ip into iq, thus making iq point to whatever ip pointed to.

5.2 Pointers and Function Arguments

Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function. For instance, a sorting routine might exchange two out-of-order elements with a function called swap. It is not enough to write

```
swap(a, b);
```

where the swap function is defined as

```
void swap(int x, int y)  /* WRONG */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Because of call by value, swap can't affect the arguments a and b in the routine that called it. The function above only swaps *copies* of a and b.

The way to obtain the desired effect is for the calling program to pass *pointers* to the values to be changed:

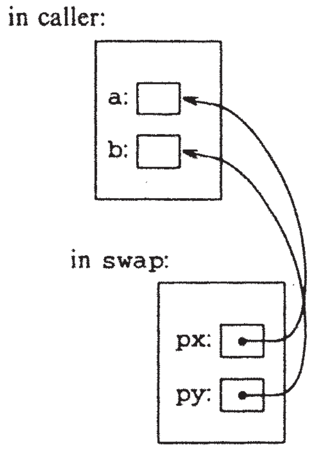
```
swap(&a, &b);
```

Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared to be pointers, and the operands are accessed indirectly through them.

```
void swap(int *px, int *py)  /* interchange *px and *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Pictorially:



Pointer arguments enable a function to access and change objects in the function that called it. As an example, consider a function `getint` that performs free-format input conversion by breaking a stream of characters into integer values, one integer per call. `getint` has to return the value it found and also signal end of file when there is no more input. These values have to be passed back by separate paths, for no matter what value is used for `EOF`, that could also be the value of an input integer.

One solution is to have `getint` return the end of file status as its function value, while using a pointer argument to store the converted integer back in the calling function. This is the scheme used by `scanf` as well; see Section 7.4.

The following loop fills an array with integers by calls to `getint`:

```
int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
    ;
```

Each call sets `array[n]` to the next integer found in the input and increments `n`. Notice that it is essential to pass the address of `array[n]` to `getint`. Otherwise there is no way for `getint` to communicate the converted integer back to the caller.

Our version of `getint` returns `EOF` for end of file, zero if the next input is not a number, and a positive value if the input contains a valid number.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint:  get next integer from input into *pn */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch()))   /* skip white space */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c);    /* it's not a number */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

Throughout getint, *pn is used as an ordinary int variable. We have also used getch and ungetch (described in Section 4.3) so the one extra character that must be read can be pushed back onto the input.

Exercise 5-1. As written, getint treats a + or - not followed by a digit as a valid representation of zero. Fix it to push such a character back on the input. □

Exercise 5-2. Write getfloat, the floating-point analog of getint. What type does getfloat return as its function value? □

5.9 Kochan, S G (2005) *Programming in C*, 3rd edn, Indianapolis: Sam’s Publishing, 259–72.

Pointers and Arrays

One of the most common uses of pointers in C is as pointers to arrays. The main reasons for using pointers to arrays are ones of notational convenience and of program efficiency. Pointers to arrays generally result in code that uses less memory and executes faster. The reason for this will become apparent through our discussions in this section.

If you have an array of 100 integers called `values`, you can define a pointer called `valuesPtr`, which can be used to access the integers contained in this array with the statement

```
int  *valuesPtr;
```

When you define a pointer that is used to point to the elements of an array, you don't designate the pointer as type "pointer to array"; rather, you designate the pointer as pointing to the type of element that is contained in the array.

If you have an array of characters called `text`, you could similarly define a pointer to be used to point to elements in `text` with the statement

```
char  *textPtr;
```

To set `valuesPtr` to point to the first element in the `values` array, you simply write

```
valuesPtr = values;
```

The address operator is not used in this case because the C compiler treats the appearance of an array name without a subscript as a pointer to the array. Therefore, simply specifying `values` without a subscript has the effect of producing a pointer to the first element of `values` (see Figure 11.9).

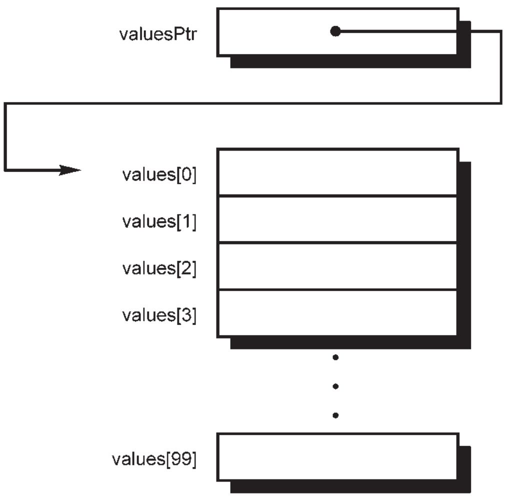


Figure 11.9 Pointer to an array element.

An equivalent way of producing a pointer to the start of `values` is to apply the address operator to the first element of the array. Thus, the statement

```
valuesPtr = &values[0];
```

can be used to serve the same purpose as placing a pointer to the first element of `values` in the pointer variable `valuesPtr`.

To set `textPtr` to point to the first character inside the `text` array, either the statement

```
textPtr = text;
```

or

```
textPtr = &text[0];
```

can be used. Whichever statement you choose to use is strictly a matter of taste.

The real power of using pointers to arrays comes into play when you want to sequence through the elements of an array. If `valuesPtr` is as previously defined and is set pointing to the first element of `values`, the expression

```
*valuesPtr
```

can be used to access the first integer of the `values` array, that is, `values[0]`. To reference `values[3]` through the `valuesPtr` variable, you can add 3 to `valuesPtr` and then apply the indirection operator:

```
*(valuesPtr + 3)
```

In general, the expression

```
*(valuesPtr + i)
```

can be used to access the value contained in `values[i]`.

So, to set `values[10]` to 27, you could obviously write the expression

```
values[10] = 27;
```

or, using `valuesPtr`, you could write

```
*(valuesPtr + 10) = 27;
```

To set `valuesPtr` to point to the second element of the `values` array, you can apply the address operator to `values[1]` and assign the result to `valuesPtr`:

```
valuesPtr = &values[1];
```

If `valuesPtr` points to `values[0]`, you can set it to point to `values[1]` by simply adding 1 to the value of `valuesPtr`:

```
valuesPtr += 1;
```

This is a perfectly valid expression in C and can be used for pointers to *any* data type.

So, in general, if `a` is an array of elements of type `x`, `px` is of type "pointer to `x`," and `i` and `n` are integer constants or variables, the statement

```
px = a;
```

sets `px` to point to the first element of `a`, and the expression

```
*(px + i)
```

subsequently references the value contained in `a[i]`. Furthermore, the statement

```
px += n;
```

sets px to point n elements farther in the array, *no matter what type of element is contained in the array*.

The increment and decrement operators ++ and -- are particularly handy when dealing with pointers. Applying the increment operator to a pointer has the same effect as adding one to the pointer, while applying the decrement operator has the same effect as subtracting one from the pointer. So, if textPtr is defined as a char pointer and is set pointing to the beginning of an array of chars called text, the statement

```
++textPtr;
```

sets textPtr pointing to the next character in text, which is text[1]. In a similar fashion, the statement

```
--textPtr;
```

sets textPtr pointing to the previous character in text, assuming, of course, that textPtr was not pointing to the beginning of text prior to the execution of this statement.

It is perfectly valid to compare two pointer variables in C. This is particularly useful when comparing two pointers in the same array. For example, you can test the pointer valuesPtr to see if it points past the end of an array containing 100 elements by comparing it to a pointer to the last element in the array. So, the expression

```
valuesPtr > &values[99]
```

is TRUE (nonzero) if valuesPtr is pointing past the last element in the values array, and is FALSE (zero) otherwise. Recall from previous discussions that you can replace the preceding expression with its equivalent

```
valuesPtr > values + 99
```

because values used without a subscript is a pointer to the beginning of the values array. (Remember, it's the same as writing &values[0].)

Program 11.11 illustrates pointers to arrays. The arraySum function calculates the sum of the elements contained in an array of integers.

Program 11.11 **Working with Pointers to Arrays**

```
// Function to sum the elements of an integer array

#include <stdio.h>

int  arraySum (int  array[], const int  n)
{
     int  sum = 0, *ptr;
     int  * const arrayEnd = array + n;

     for ( ptr = array;  ptr < arrayEnd;  ++ptr )
          sum += *ptr;
```

Program 11.11 **Continued**

```
     return sum;
}

int main (void)
{
     int  arraySum (int  array[], const int  n);
     int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

     printf ("The sum is %i\n", arraySum (values, 10));

     return 0;
}
```

Program 11.11 **Output**

```
The sum is 21
```

Inside the `arraySum` function, the constant integer pointer `arrayEnd` is defined and set pointing immediately after the last element of `array`. A `for` loop is then set up to sequence through the elements of `array`. The value of `ptr` is set to point to the beginning of `array` when the loop is entered. Each time through the loop, the element of `array` pointed to by `ptr` is added into `sum`. The value of `ptr` is then incremented by the `for` loop to set it pointing to the next element in `array`. When `ptr` points past the end of `array`, the `for` loop is exited, and the value of `sum` is returned to the calling routine.

A Slight Digression About Program Optimization

It is pointed out that the local variable `arrayEnd` was not actually needed by the function because you could have explicitly compared the value of `ptr` to the end of the array inside the `for` loop:

```
for ( ...; pointer <= array + n; ... )
```

The sole motivation for using `arrayEnd` was one of optimization. Each time through the `for` loop, the looping conditions are evaluated. Because the expression `array + n` is never changed from within the loop, its value is constant throughout the execution of the `for` loop. By evaluating it once *before* the loop is entered, you save the time that would otherwise be spent reevaluating this expression each time through the loop. Although there is virtually no savings in time for a 10-element array, especially if the `arraySum` function is called only once by the program, there could be a more substantial savings if this function were heavily used by a program for summing large-sized arrays, for example.

The other issue to be discussed about program optimization concerns the very use of pointers themselves in a program. In the `arraySum` function discussed earlier, the expression `*ptr` is used inside the `for` loop to access the elements in the array. Formerly, you would have written your `arraySum` function with a `for` loop that used an index variable, such as `i`, and then would have added the value of `array[i]` into `sum` inside the loop. In general, the process of indexing an array takes more time to execute than does the process of accessing the contents of a pointer. In fact, this is one of the main reasons why pointers are used to access the elements of an array—the code that is generated is generally more efficient. Of course, if access to the array is not generally sequential, pointers accomplish nothing, as far as this issue is concerned, because the expression `*(pointer + j)` takes just as long to execute as does the expression `array[j]`.

Is It an Array or Is It a Pointer?

Recall that to pass an array to a function, you simply specify the name of the array, as you did previously with the call to the `arraySum` function. You should also remember that to produce a pointer to an array, you need only specify the name of the array. This implies that in the call to the `arraySum` function, what was passed to the function was actually a *pointer* to the array `values`. This is precisely the case and explains why you are able to change the elements of an array from within a function.

But if it is indeed the case that a pointer to the array is passed to the function, then you might wonder why the formal parameter inside the function isn't declared to be a pointer. In other words, in the declaration of `array` in the `arraySum` function, why isn't the declaration

```
int  *array;
```

used? Shouldn't all references to an array from within a function be made using pointer variables?

To answer these questions, recall the previous discussion about pointers and arrays. As mentioned, if `valuesPtr` points to the same type of element as contained in an array called `values`, the expression `*(valuesPtr + i)` is in all ways equivalent to the expression `values[i]`, assuming that `valuesPtr` has been set to point to the beginning of `values`. What follows from this is that you also can use the expression `*(values + i)` to reference the `i`th element of the array `values`, and, in general, if `x` is an array of any type, the expression `x[i]` can always be equivalently expressed in C as `*(x + i)`.

As you can see, pointers and arrays are intimately related in C, and this is why you can declare `array` to be of type "array of `int`s" inside the `arraySum` function *or* to be of type "pointer to `int`." Either declaration works just fine in the preceding program—try it and see.

If you are going to be using index numbers to reference the elements of an array that is passed to a function, declare the corresponding formal parameter to be an array. This more correctly reflects the use of the array by the function. Similarly, if you are using the argument as a pointer to the array, declare it to be of type pointer.

Realizing now that you could have declared `array` to be an `int` pointer in the preceding program example, and then could have subsequently used it as such, you can eliminate the variable `ptr` from the function and use `array` instead, as shown in Program 11.12.

Program 11.12 **Summing the Elements of an Array**

```
// Function to sum the elements of an integer array  Ver. 2

#include <stdio.h>

int  arraySum (int  *array, const int  n)
{
     int  sum = 0;
     int  * const arrayEnd = array + n;

     for (  ; array < arrayEnd;  ++array )
          sum += *array;

     return sum;
}



int main (void)
{
     int  arraySum (int  *array, const int  n);
     int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };

     printf ("The sum is %i\n", arraySum (values, 10));

     return 0;
}
```

Program 11.12 **Output**

```
The sum is 21
```

The program is fairly self-explanatory. The first expression inside the `for` loop was omitted because no value had to be initialized before the loop was started. One point worth repeating is that when the `arraySum` function is called, a pointer to the `values` array is passed, where it is called `array` inside the function. Changes to the value of `array` (as opposed to the values referenced by `array`) do not in any way affect the contents of the `values` array. So, the increment operator that is applied to `array` is just incrementing a pointer to the array `values`, and not affecting its contents. (Of course, you know that you *can* change values in the array if you want to, simply by assigning values to the elements referenced by the pointer.)

Pointers to Character Strings

One of the most common applications of using a pointer to an array is as a pointer to a character string. The reasons are ones of notational convenience and efficiency. To show how easily pointers to character strings can be used, write a function called `copyString` to copy one string into another. If you write this function using normal array indexing methods, the function might be coded as follows:

```
void copyString (char  to[], char  from[])
{
     int  i;

     for ( i = 0;  from[i] != '\0';  ++i )
          to[i] = from[i];

     to[i] = '\0';
}
```

The `for` loop is exited before the null character is copied into the `to` array, thus explaining the need for the last statement in the function.

If you write `copyString` using pointers, you no longer need the index variable `i`. A pointer version is shown in Program 11.13.

Program 11.13 **Pointer Version of** `copyString`

```
#include <stdio.h>

void copyString (char  *to, char  *from)
{
     for (  ;  *from != '\0';  ++from, ++to )
          *to = *from;

     *to = '\0';
}

int main (void)
{
     void  copyString (char  *to, char  *from);
     char  string1[] = "A string to be copied.";
     char  string2[50];

     copyString (string2, string1);
     printf ("%s\n", string2);

     copyString (string2, "So is this.");
     printf ("%s\n", string2);

     return 0;
}
```

Program 11.13 **Output**

```
A string to be copied.
So is this.
```

The `copyString` function defines the two formal parameters `to` and `from` as character pointers and not as character arrays as was done in the previous version of `copyString`. This reflects how these two variables are used by the function.

A `for` loop is then entered (with no initial conditions) to copy the string pointed to by `from` into the string pointed to by `to`. Each time through the loop, the `from` and `to` pointers are each incremented by one. This sets the `from` pointer pointing to the next character that is to be copied from the source string and sets the `to` pointer pointing to the location in the destination string where the next character is to be stored.

When the `from` pointer points to the null character, the `for` loop is exited. The function then places the null character at the end of the destination string.

In the `main` routine, the `copyString` function is called twice, the first time to copy the contents of `string1` into `string2`, and the second time to copy the contents of the constant character string `"So is this."` into `string2`.

Constant Character Strings and Pointers

The fact that the call

```
copyString (string2, "So is this.");
```

works in the previous program implies that when a constant character string is passed as an argument to a function, what is actually passed is a pointer to that character string. Not only is this true in this case, but it also can be generalized by saying that *whenever* a constant character string is used in C, it is a pointer to that character string that is produced. So, if `textPtr` is declared to be a character pointer, as in

```
char  *textPtr;
```

then the statement

```
textPtr = "A character string.";
```

assigns to `textPtr` a *pointer* to the constant character string `"A character string."` Be careful to make the distinction here between character pointers and character arrays, as the type of assignment just shown is *not* valid with a character array. So, for example, if `text` is defined instead to be an array of `char`s, with a statement such as

```
char  text[80];
```

then you *could not* write a statement such as

```
text = "This is not valid.";
```

The *only* time that C lets you get away with performing this type of assignment to a character array is when initializing it, as in

```
char  text[80] =  "This is okay.";
```

Initializing the `text` array in this manner does not have the effect of storing a pointer to the character string `"This is okay."` inside `text`, but rather the actual characters themselves inside corresponding elements of the `text` array.

If `text` is a character pointer, initializing `text` with the statement

```
char  *text =  "This is okay.";
```

assigns to it a pointer to the character string `"This is okay."`

As another example of the distinction between character strings and character string pointers, the following sets up an array called `days`, which contains *pointers* to the names of the days of the week.

```
char *days[] =
    { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
      "Saturday" };
```

The array `days` is defined to contain seven entries, each a pointer to a character string. So `days[0]` contains a pointer to the character string `"Sunday"`, `days[1]` contains a pointer to the string `"Monday"`, and so on (see Figure 11.10). You could display the name of the third weekday, for example, with the following statement:

```
printf ("%s\n", days[3]);
```

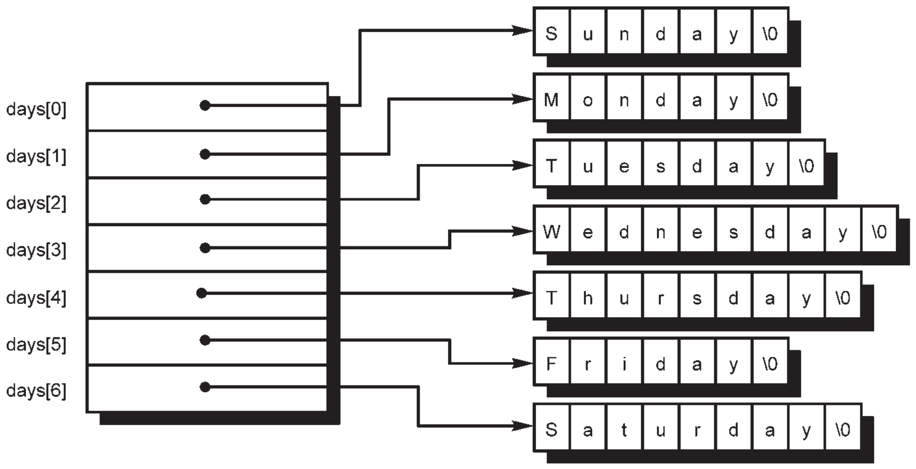


Figure 11.10 Array of pointers.

The Increment and Decrement Operators Revisited

Up to this point, whenever you used the increment or decrement operator, it was the only operator that appeared in the expression. When you write the expression `++x`, you know that this has the effect of adding 1 to the value of the variable `x`. And as you have just seen, if `x` is a pointer to an array, this has the effect of setting `x` to point to the next element of the array.

The increment and decrement operators can be used in expressions in which other operators also appear. In such cases, it becomes important to know more precisely how these operators work.

So far, when you used the increment and decrement operators, you always placed them *before* the variables that were being incremented or decremented. So, to increment a variable `i`, you simply wrote

```
++i;
```

Actually, it also is perfectly valid to place the increment operator *after* the variable, as follows:

```
i++;
```

Both expressions are perfectly valid and both achieve the same result—namely, of incrementing the value of `i`. In the first case, where the `++` is placed before its operand, the increment operation is more precisely identified as a *preincrement*. In the second case, where the `++` is placed after its operand, the operation is identified as a *postincrement*.

The same discussion applies to the decrement operator. So the statement

```
--i;
```

technically performs a *predecrement* of `i`, whereas the statement

```
i--;
```

performs a *postdecrement* of `i`. Both have the same net result of subtracting 1 from the value of `i`.

It is when the increment and decrement operators are used in more complex expressions that the distinction between the *pre-* and *post-* nature of these operators is realized.

Suppose you have two integers called `i` and `j`. If you set the value of `i` to `0` and then write the statement

```
j = ++i;
```

the value that gets assigned to `j` is `1`, and not `0` as you might expect. In the case of the preincrement operator, the variable is incremented *before* its value is used in the expression. So, in the preceding expression, the value of `i` is first incremented from `0` to `1` and then its value is assigned to `j`, as if the following two statements had been written instead:

```
++i;
j = i;
```

If you instead use the postincrement operator in the statement

```
j = i++;
```

then `i` is incremented *after* its value has been assigned to `j`. So, if `i` is `0` before the preceding statement is executed, `0` is assigned to `j` and *then* `i` is incremented by 1, as if the statements

```
j = i;
++i;
```

were used instead. As another example, if `i` is equal to 1, then the statement

```
x = a[--i];
```

has the effect of assigning the value of `a[0]` to `x` because the variable `i` is decremented before its value is used to index into `a`. The statement

```
x = a[i--];
```

used instead has the effect of assigning the value of `a[1]` to `x` because `i` is decremented after its value has been used to index into `a`.

As a third example of the distinction between the pre- and post- increment and decrement operators, the function call

```
printf ("%i\n", ++i);
```

increments `i` and then sends its value to the `printf` function, whereas the call

```
printf ("%i\n", i++);
```

increments `i` after its value has been sent to the function. So, if `i` is equal to 100, the first `printf` call displays 101, whereas the second `printf` call displays 100. In either case, the value of `i` is equal to 101 after the statement has executed.

As a final example on this topic before presenting Program 11.14, if `textPtr` is a character pointer, the expression

```
*(++textPtr)
```

first increments `textPtr` and then fetches the character it points to, whereas the expression

```
*(textPtr++)
```

fetches the character pointed to by `textPtr` before its value is incremented. In either case, the parentheses are not required because the `*` and `++` operators have equal precedence but associate from right to left.

Now go back to the `copyString` function from Program 11.13 and rewrite it to incorporate the increment operations directly into the assignment statement.

Because the `to` and `from` pointers are incremented each time after the assignment statement inside the `for` loop is executed, they should be incorporated into the assignment statement as postincrement operations. The revised `for` loop of Program 11.13 then becomes

```
for ( ; *from != '\0'; )
     *to++ = *from++;
```

Execution of the assignment statement inside the loop proceeds as follows. The character pointed to by `from` is retrieved and then `from` is incremented to point to the next character in the source string. The referenced character is then stored inside the location pointed to by `to`, and then `to` is incremented to point to the next location in the destination string.

Study the preceding assignment statement until you fully understand its operation. Statements of this type are so commonly used in C programs, it's important that you understand it completely before continuing.

The preceding `for` statement hardly seems worthwhile because it has no initial expression and no looping expression. In fact, the logic would be better served when expressed in the form of a `while` loop. This has been done in Program 11.14. This program presents your new version of the `copyString` function. The `while` loop uses the fact that the null character is equal to the value `0`, as is commonly done by experienced C programmers.

Program 11.14 **Revised Version of the `copyString` Function**

```
// Function to copy one string to another. Pointer Ver. 2

#include <stdio.h>

void  copyString (char  *to, char  *from)
{
     while ( *from )
          *to++ = *from++;

     *to = '\0';
}



int main (void)
{
     void  copyString (char  *to, char  *from);
     char  string1[] = "A string to be copied.";
     char  string2[50];

     copyString (string2, string1);
     printf ("%s\n", string2);

     copyString (string2, "So is this.");
     printf ("%s\n", string2);

     return 0;
}
```

Program 11.14 **Output**

```
A string to be copied.
So is this.
```

5.11 Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 104–7.

5.5 Character Pointers and Functions

A *string constant*, written as

```
"I am a string"
```

is an array of characters. In the internal representation, the array is terminated with the null character `'\0'` so that programs can find the end. The length in storage is thus one more than the number of characters between the double quotes.

Perhaps the most common occurrence of string constants is as arguments to functions, as in

```
printf("hello, world\n");
```

When a character string like this appears in a program, access to it is through a character pointer; `printf` receives a pointer to the beginning of the character array. That is, a string constant is accessed by a pointer to its first element.

String constants need not be function arguments. If `pmessage` is declared as

```
char *pmessage;
```

then the statement

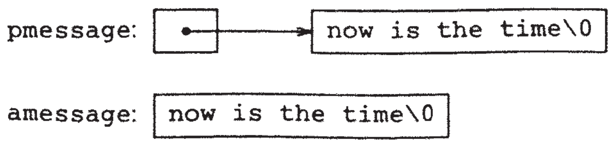
```
pmessage = "now is the time";
```

assigns to `pmessage` a pointer to the character array. This is *not* a string copy; only pointers are involved. C does not provide any operators for processing an entire string of characters as a unit.

There is an important difference between these definitions:

```
char amessage[] = "now is the time";   /* an array */
char *pmessage = "now is the time";    /* a pointer */
```

`amessage` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amessage` will always refer to the same storage. On the other hand, `pmessage` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.



We will illustrate more aspects of pointers and arrays by studying versions of two useful functions adapted from the standard library. The first function is `strcpy(s,t)`, which copies the string `t` to the string `s`. It would be nice just to say `s=t` but this copies the pointer, not the characters. To copy the characters, we need a loop. The array version is first:

```
/* strcpy:  copy t to s; array subscript version */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

For contrast, here is a version of strcpy with pointers:

```
/* strcpy:  copy t to s; pointer version 1 */
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Because arguments are passed by value, strcpy can use the parameters s and t in any way it pleases. Here they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the '\0' that terminates t has been copied to s.

In practice, strcpy would not be written as we showed it above. Experienced C programmers would prefer

```
/* strcpy:  copy t to s; pointer version 2 */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

This moves the increment of s and t into the test part of the loop. The value of *t++ is the character that t pointed to before t was incremented; the postfix ++ doesn't change t until after this character has been fetched. In the same way, the character is stored into the old s position before s is incremented. This character is also the value that is compared against '\0' to control the loop. The net effect is that characters are copied from t to s, up to and including the terminating '\0'.

As the final abbreviation, observe that a comparison against '\0' is redundant, since the question is merely whether the expression is zero. So the function would likely be written as

```
/* strcpy:  copy t to s; pointer version 3 */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Although this may seem cryptic at first sight, the notational convenience is considerable, and the idiom should be mastered, because you will see it frequently in C programs.

The `strcpy` in the standard library (`<string.h>`) returns the target string as its function value.

The second routine that we will examine is `strcmp(s,t)`, which compares the character strings `s` and `t`, and returns negative, zero or positive if `s` is lexicographically less than, equal to, or greater than `t`. The value is obtained by subtracting the characters at the first position where `s` and `t` disagree.

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

The pointer version of `strcmp`:

```
/* strcmp:  return <0 if s<t, 0 if s==t, >0 if s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
    return *s - *t;
}
```

Since `++` and `--` are either prefix or postfix operators, other combinations of `*` and `++` and `--` occur, although less frequently. For example,

```
*--p
```

decrements `p` before fetching the character that `p` points to. In fact, the pair of expressions

```
*p++ = val;     /* push val onto stack */
val = *--p;     /* pop top of stack into val */
```

are the standard idioms for pushing and popping a stack; see Section 4.3.

The header `<string.h>` contains declarations for the functions mentioned in this section, plus a variety of other string-handling functions from the standard library.

Exercise 5-3. Write a pointer version of the function `strcat` that we showed in Chapter 2: `strcat(s,t)` copies the string `t` to the end of `s`. □

Exercise 5-4. Write the function `strend(s,t)`, which returns 1 if the string `t` occurs at the end of the string `s`, and zero otherwise. □

Exercise 5-5. Write versions of the library functions `strncpy`, `strncat`, and `strncmp`, which operate on at most the first `n` characters of their argument strings. For example, `strncpy(s,t,n)` copies at most `n` characters of `t` to `s`. Full descriptions are in Appendix B. □

Exercise 5-6. Rewrite appropriate programs from earlier chapters and exercises with pointers instead of array indexing. Good possibilities include `getline` (Chapters 1 and 4), `atoi`, `itoa`, and their variants (Chapters 2, 3, and 4), `reverse` (Chapter 3), and `strindex` and `getop` (Chapter 4). □

5.12 Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 114–18.

5.10 Command-line Arguments

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When `main` is called, it is called with two arguments. The first (conventionally called `argc`, for argument count) is the number of command-line arguments the program was invoked with; the second (`argv`, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

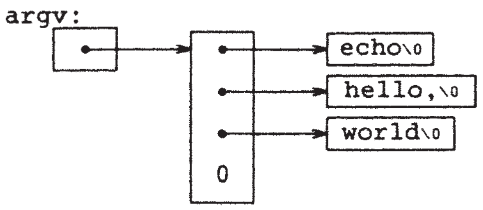
The simplest illustration is the program `echo`, which echoes its command-line arguments on a single line, separated by blanks. That is, the command

```
echo hello, world
```

prints the output

```
hello, world
```

By convention, argv[0] is the name by which the program was invoked, so argc is at least 1. If argc is 1, there are no command-line arguments after the program name. In the example above, argc is 3, and argv[0], argv[1], and argv[2] are "echo", "hello,", and "world" respectively. The first optional argument is argv[1] and the last is argv[argc-1]; additionally, the standard requires that argv[argc] be a null pointer.



The first version of echo treats argv as an array of character pointers:

```
#include <stdio.h>

/* echo command-line arguments; 1st version */
main(int argc, char *argv[])
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since argv is a pointer to an array of pointers, we can manipulate the pointer rather than index the array. This next variation is based on incrementing argv, which is a pointer to pointer to char, while argc is counted down:

```
#include <stdio.h>

/* echo command-line arguments; 2nd version */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Since argv is a pointer to the beginning of the array of argument strings, incrementing it by 1 (++argv) makes it point at the original argv[1] instead of argv[0]. Each successive increment moves it along to the next argument; *argv is then the pointer to that argument. At the same time, argc is decremented; when it becomes zero, there are no arguments left to print.

Alternatively, we could write the printf statement as

```
            printf((argc > 1) ? "%s " : "%s", *++argv);
```

This shows that the format argument of `printf` can be an expression too.

As a second example, let us make some enhancements to the pattern-finding program from Section 4.1. If you recall, we wired the search pattern deep into the program, an obviously unsatisfactory arrangement. Following the lead of the UNIX program `grep`, let us change the program so the pattern to be matched is specified by the first argument on the command line.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find:  print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
    return found;
}
```

The standard library function `strstr(s,t)` returns a pointer to the first occurrence of the string `t` in the string `s`, or `NULL` if there is none. It is declared in `<string.h>`.

The model can now be elaborated to illustrate further pointer constructions. Suppose we want to allow two optional arguments. One says "print all lines *except* those that match the pattern;" the second says "precede each printed line by its line number."

A common convention for C programs on UNIX systems is that an argument that begins with a minus sign introduces an optional flag or parameter. If we choose `-x` (for "except") to signal the inversion, and `-n` ("number") to request line numbering, then the command

```
find -x -n pattern
```

will print each line that doesn't match the pattern, preceded by its line number.

Optional arguments should be permitted in any order, and the rest of the program should be independent of the number of arguments that were present. Furthermore, it is convenient for users if option arguments can be combined, as in

```
    find -nx pattern
```

Here is the program:

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find:  print lines that match pattern from 1st arg */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: illegal option %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
    else
        while (getline(line, MAXLINE) > 0) {
            lineno++;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    printf("%ld:", lineno);
                printf("%s", line);
                found++;
            }
        }
    return found;
}
```

argc is decremented and argv is incremented before each optional argument. At the end of the loop, if there are no errors, argc tells how many arguments remain unprocessed and argv points to the first of these. Thus argc should be 1 and `*argv` should point at the pattern. Notice that `*++argv` is a pointer to an argument string, so `(*++argv)[0]` is its first character. (An alternate valid form would be `**++argv`.) Because `[]` binds tighter than `*` and `++`, the parentheses are necessary; without them the expression would be taken as `*++(argv[0])`. In fact, that is what we used in the inner loop, where the task is to walk along a specific argument string. In the inner loop, the expression `*++argv[0]` increments the pointer `argv[0]`!

It is rare that one uses pointer expressions more complicated than these; in such cases, breaking them into two or three steps will be more intuitive.

Exercise 5-10. Write the program `expr`, which evaluates a reverse Polish expression from the command line, where each operator or operand is a separate argument. For example,

```
expr  2  3  4  +  *
```

evaluates 2 × (3+4). □

Exercise 5-11. Modify the programs `entab` and `detab` (written as exercises in Chapter 1) to accept a list of tab stops as arguments. Use the default tab settings if there are no arguments. □

Exercise 5-12. Extend `entab` and `detab` to accept the shorthand

```
entab -m +n
```

to mean tab stops every *n* columns, starting at column *m*. Choose convenient (for the user) default behavior. □

Exercise 5-13. Write the program `tail`, which prints the last *n* lines of its input. By default, *n* is 10, let us say, but it can be changed by an optional argument, so that

```
tail -n
```

prints the last *n* lines. The program should behave rationally no matter how unreasonable the input or the value of *n*. Write the program so it makes the best use of available storage; lines should be stored as in the sorting program of Section 5.6, not in a two-dimensional array of fixed size. □

5.13 Kochan, S G (2005) *Programming in C*, 3rd edn, Indianapolis: Sam's Publishing, 166–77, 180–85, 240–43 and 325–27.

A Structure for Storing the Date

You can define a structure called `date` in the C language that consists of three components that represent the month, day, and year. The syntax for such a definition is rather straightforward, as follows:

```
struct  date
{
    int  month;
    int  day;
    int  year;
};
```

The `date` structure just defined contains three integer *members* called `month`, `day`, and `year`. The definition of `date` in a sense defines a new type in the language in that variables can subsequently be declared to be of type `struct date`, as in the declaration

```
struct date  today;
```

You can also declare a variable `purchaseDate` to be of the same type by a separate declaration, such as

```
struct date  purchaseDate;
```

Or, you can simply include the two declarations on the same line, as in

```
struct date  today, purchaseDate;
```

Unlike variables of type `int`, `float`, or `char`, a special syntax is needed when dealing with structure variables. A member of a structure is accessed by specifying the variable name, followed by a period, and then the member name. For example, to set the value of the `day` in the variable `today` to `25`, you write

```
today.day = 25;
```

Note that there are no spaces permitted between the variable name, the period, and the member name. To set the `year` in `today` to `2004`, the expression

```
today.year = 2004;
```

can be used. Finally, to test the value of `month` to see if it is equal to `12`, a statement such as

```
if  ( today.month == 12 )
   nextMonth = 1;
```

does the trick.

Try to determine the effect of the following statement.

```
if  ( today.month == 1  &&  today.day == 1 )
  printf ("Happy New Year!!!\n");
```

Program 9.1 incorporates the preceding discussions into an actual C program.

Program 9.1 **Illustrating a Structure**

```
// Program to illustrate a structure

#include <stdio.h>

int main (void)
{
    struct  date
    {
        int  month;
        int  day;
        int  year;
    };

    struct date  today;

    today.month = 9;
    today.day = 25;
    today.year = 2004;

    printf ("Today's date is %i/%i/%.2i.\n", today.month, today.day,
              today.year % 100);

    return 0;
}
```

Program 9.1 **Output**

```
Today's date is 9/25/04.
```

The first statement inside `main` defines the structure called `date` to consist of three integer members called `month`, `day`, and `year`. In the second statement, the variable `today` is declared to be of type `struct date`. The first statement simply defines what a `date` structure looks like to the C compiler and causes no storage to be reserved inside the computer. The second statement declares a variable to be of type `struct date` and, therefore, *does* cause memory to be reserved for storing the three integer values of the variable `today`. Be certain you understand the difference between defining a structure and declaring variables of the particular structure type.

After `today` has been declared, the program then proceeds to assign values to each of the three members of `today`, as depicted in Figure 9.1.

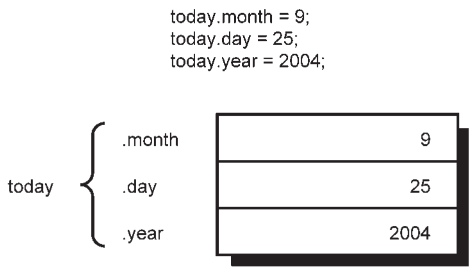


Figure 9.1. Assigning values to a structure variable.

After the assignments have been made, the values contained inside the structure are displayed by an appropriate `printf` call. The remainder of `today.year` divided by 100 is calculated prior to being passed to the `printf` function so that just 04 is displayed for the year. Recall that the format characters `%.2i` are used to specify that two integer digits are to be displayed with zero fill. This ensures that you get the proper display for the last two digits of the year.

Using Structures in Expressions

When it comes to the evaluation of expressions, structure members follow the same rules as ordinary variables do in the C language. So division of an integer structure member by another integer is performed as an integer division, as in

```
century = today.year / 100 + 1;
```

Suppose you want to write a simple program that accepts today's date as input and displays tomorrow's date to the user. Now, at first glance, this seems a perfectly simple task to perform. You can ask the user to enter today's date and then proceed to calculate tomorrow's date by a series of statements, such as

```
tomorrow.month = today.month;
tomorrow.day   = today.day + 1;
tomorrow.year  = today.year;
```

Of course, the preceding statements work just fine for the majority of dates, but the following two cases are not properly handled:

1. If today's date falls at the end of a month.
2. If today's date falls at the end of a year (that is, if today's date is December 31).

One way to determine easily if today's date falls at the end of a month is to set up an array of integers that corresponds to the number of days in each month. A lookup inside the array for a particular month then gives the number of days in that month. So the statement

```
int  daysPerMonth[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

defines an array called `daysPerMonth` containing 12 integer elements. For each month `i`, the value contained in `daysPerMonth[i - 1]` corresponds to the number of days in that particular month. Therefore, the number of days in April, which is the fourth month of the year, is given by `daysPerMonth[3]`, which is equal to 30. (You could define the array to contain 13 elements, with `daysPerMonth[i]` corresponding to the number of days in month `i`. Access into the array could then be made directly based on the month number, rather than on the month number minus 1. The decision of whether to use 12 or 13 elements in this case is strictly a matter of personal preference.)

If it is determined that today's date falls at the end of the month, you can calculate tomorrow's date by simply adding 1 to the month number and setting the value of the day equal to `1`.

To solve the second problem mentioned earlier, you must determine if today's date is at the end of a month and if the month is 12. If this is the case, then tomorrow's day and month must be set equal to `1` and the year appropriately incremented by 1.

Program 9.2 asks the user to enter today's date, calculates tomorrow's date, and displays the results.

Program 9.2 **Determining Tomorrow's Date**

```
// Program to determine tomorrow's date

#include <stdio.h>

int main (void)
{
    struct  date
    {
         int  month;
         int  day;
         int  year;
     };

     struct date  today, tomorrow;

     const int  daysPerMonth[12] = { 31, 28, 31, 30, 31, 30,
                                     31, 31, 30, 31, 30, 31 };

     printf ("Enter today's date (mm dd yyyy): ");
     scanf ("%i%i%i", &today.month, &today.day, &today.year);

     if  ( today.day != daysPerMonth[today.month - 1] ) {
         tomorrow.day = today.day + 1;
         tomorrow.month = today.month;
         tomorrow.year = today.year;
     }
```

Program 9.2 **Continued**

```
    else if ( today.month == 12 ) {    // end of year
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else {                             // end of month
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }

    printf ("Tomorrow's date is %i/%i/%.2i.\n", tomorrow.month,
           tomorrow.day, tomorrow.year % 100);

    return 0;
}
```

Program 9.2 **Output**

```
Enter today's date (mm dd yyyy): 12 17 2004
Tomorrow's date is 12/18/04.
```

Program 9.2 **Output (Rerun)**

```
Enter today's date (mm dd yyyy): 12 31 2005
Tomorrow's date is 1/1/06.
```

Program 9.2 **Output (Second Rerun)**

```
Enter today's date (mm dd yyyy): 2 28 2004
Tomorrow's date is 3/1/04.
```

If you look at the program's output, you quickly notice that there seems to be a mistake somewhere: The day after February 28, 2004 is listed as March 1, 2004 and *not* as February 29, 2004. The program forgot about leap years! You fix this problem in the following section. First, you need to analyze the program and its logic.

After the `date` structure is defined, two variables of type `struct date`, `today` and `tomorrow`, are declared. The program then asks the user to enter today's date. The three integer values that are entered are stored into `today.month`, `today.day`, and `today.year`, respectively. Next, a test is made to determine if the day is at the end of the month, by comparing `today.day` to `daysPerMonth[today.month - 1]`. If it is not the end of the month, tomorrow's date is calculated by simply adding 1 to the day and setting tomorrow's month and year equal to today's month and year.

If today's date does fall at the end of the month, another test is made to determine if it is the end of the year. If the month equals 12, meaning that today's date is December 31, tomorrow's date is set equal to January 1 of the next year. If the month does not equal 12, tomorrow's date is set to the first day of the following month (of the same year).

After tomorrow's date has been calculated, the values are displayed to the user with an appropriate `printf` statement call, and program execution is complete.

Functions and Structures

Now, you can return to the problem that was discovered in the previous program. Your program thinks that February always has 28 days, so naturally when you ask it for the day after February 28, it always displays March 1 as the answer. You need to make a special test for the case of a leap year. If the year is a leap year, and the month is February, the number of days in that month is 29. Otherwise, the normal lookup inside the `daysPerMonth` array can be made.

A good way to incorporate the required changes into Program 9.2 is to develop a function called `numberOfDays` to determine the number of days in a month. The function would perform the leap year test and the lookup inside the `daysPerMonth` array as required. Inside the `main` routine, all that has to be changed is the `if` statement, which compares the value of `today.day` to `daysPerMonth[today.month - 1]`. Instead, you could now compare the value of `today.day` to the value returned by your `numberOfDays` function.

Study Program 9.3 carefully to determine what is being passed to the `numberOfDays` function as an argument.

Program 9.3 **Revising the Program to Determine Tomorrow's Date**

```
// Program to determine tomorrow's date

#include <stdio.h>
#include <stdbool.h>

struct  date
{
    int  month;
    int  day;
    int  year;
};

int main (void)
{
```

Program 9.3 **Continued**

```
     struct date  today, tomorrow;
     int  numberOfDays (struct date d);

     printf ("Enter today's date (mm dd yyyy): ");
     scanf ("%i%i%i", &today.month, &today.day, &today.year);

     if  ( today.day != numberOfDays (today) ) {
         tomorrow.day = today.day + 1;
         tomorrow.month = today.month;
         tomorrow.year = today.year;
     }
     else if ( today.month == 12 ) {     // end of year
         tomorrow.day = 1;
         tomorrow.month = 1;
         tomorrow.year = today.year + 1;
     }
     else {                              // end of month
          tomorrow.day = 1;
          tomorrow.month = today.month + 1;
          tomorrow.year = today.year;
     }

     printf ("Tomorrow's date is %i/%i/%.2i.\n",tomorrow.month,
                 tomorrow.day, tomorrow.year % 100);

     return 0;
}

// Function to find the number of days in a month

int  numberOfDays  (struct date  d)
{
    int   days;
    bool  isLeapYear (struct date  d);
    const int   daysPerMonth[12] =
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    if ( isLeapYear (d) == true &&  d.month == 2 )
         days = 29;
    else
         days = daysPerMonth[d.month - 1];

    return days;
}
```

Program 9.3 **Continued**

```
// Function to determine if it's a leap year

bool  isLeapYear (struct date  d)
{
    bool  leapYearFlag;

    if ( (d.year % 4 == 0  &&  d.year % 100 != 0)  ||
                d.year % 400 == 0 )
        leapYearFlag = true;   // It's a leap year
    else
        leapYearFlag = false;  // Not a leap year

    return leapYearFlag;
}
```

Program 9.3 **Output**

```
Enter today's date (mm dd yyyy): 2 28 2004
Tomorrow's date is 2/29/04.
```

Program 9.3 **Output (Rerun)**

```
Enter today's date (mm dd yyyy): 2 28 2005
Tomorrow's date is 3/1/05.
```

The first thing that catches your eye in the preceding program is the fact that the definition of the `date` structure appears first and outside of any function. This makes the definition known throughout the file. Structure definitions behave very much like variables—if a structure is defined within a particular function, only that function knows of its existence. This is a *local* structure definition. If you define the structure outside of any function, that definition is *global*. A global structure definition allows any variables that are subsequently defined in the program (either inside or outside of a function) to be declared to be of that structure type.

Inside the `main` routine, the prototype declaration

```
int  numberOfDays (struct date d);
```

informs the C compiler that the `numberOfDays` function returns an integer value and takes a single argument of type `struct date`.

Instead of comparing the value of `today.day` against the value `daysPerMonth[today.month - 1]`, as was done in the preceding example, the statement

```
if  ( today.day != numberOfDays (today) )
```

is used. As you can see from the function call, you are specifying that the structure `today` is to be passed as an argument. Inside the `numberOfDays` function, the appropriate declaration must be made to inform the system that a structure is expected as an argument:

```
int  numberOfDays  (struct date  d)
```

As with ordinary variables, and unlike arrays, any changes made by the function to the values contained in a structure argument have no effect on the original structure. They affect only the copy of the structure that is created when the function is called.

The `numberOfDays` function begins by determining if it is a leap year and if the month is February. The former determination is made by calling another function called `isLeapYear`. You learn about this function shortly. From reading the `if` statement

```
if ( isLeapYear (d) == true   &&  d.month == 2 )
```

you can assume that the `isLeapYear` function returns `true` if it is a leap year and returns `false` if it is not a leap year. This is directly in line with our discussions of Boolean variables back in Chapter 6, "Making Decisions." Recall that the standard header file `<stdbool.h>` defines the values `bool`, `true`, and `false` for you, which is why this file is included at the beginning of Program 9.3.

An interesting point to be made about the previous `if` statement concerns the choice of the function name `isLeapYear`. This name makes the `if` statement extremely readable and implies that the function is returning some kind of yes/no answer.

Getting back to the program, if the determination is made that it is February of a leap year, the value of the variable `days` is set to 29; otherwise, the value of `days` is found by indexing the `daysPerMonth` array with the appropriate month. The value of `days` is then returned to the `main` routine, where execution is continued as in Program 9.2.

The `isLeapYear` function is straightforward enough—it simply tests the year contained in the `date` structure given as its argument and returns `true` if it is a leap year and `false` if it is not.

As an exercise in producing a better-structured program, take the entire process of determining tomorrow's date and relegate it to a separate function. You can call the new function `dateUpdate` and have it take as its argument today's date. The function then calculates tomorrow's date and *returns* the new date back to us. Program 9.4 illustrates how this can be done in C.

Program 9.4 **Revising the Program to Determine Tomorrow's Date, Version 2**

```
// Program to determine tomorrow's date

#include <stdio.h>
#include <stdbool.h>

struct  date
{
    int  month;
    int  day;
```

Program 9.4 **Continued**

```
    int  year;
};

// Function to calculate tomorrow's date

struct date  dateUpdate (struct date  today)
{
    struct date  tomorrow;
    int  numberOfDays (struct date  d);

    if ( today.day != numberOfDays (today) ) {
        tomorrow.day = today.day + 1;
        tomorrow.month = today.month;
        tomorrow.year = today.year;
    }
    else if ( today.month == 12 )  {   // end of year
        tomorrow.day = 1;
        tomorrow.month = 1;
        tomorrow.year = today.year + 1;
    }
    else {                             // end of month
        tomorrow.day = 1;
        tomorrow.month = today.month + 1;
        tomorrow.year = today.year;
    }

    return tomorrow;
}

// Function to find the number of days in a month

int  numberOfDays  (struct date  d)
{
     int  days;
     bool isLeapYear (struct date  d);
     const int  daysPerMonth[12] =
       { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

     if ( isLeapYear   &&  d.month == 2 )
         days = 29;
     else
         days = daysPerMonth[d.month - 1];

     return days;
}
```

Program 9.4 **Continued**

```
// Function to determine if it's a leap year

bool  isLeapYear (struct date  d)
{
    bool  leapYearFlag;

    if ( (d.year % 4 == 0  &&  d.year % 100 != 0)  ||
                  d.year % 400 == 0 )
        leapYearFlag = true;   // It's a leap year
    else
        leapYearFlag = false;  // Not a leap year

    return leapYearFlag;
}

int main (void)
{
     struct date  dateUpdate (struct date  today);
     struct date  thisDay, nextDay;

     printf ("Enter today's date (mm dd yyyy): ");
     scanf ("%i%i%i", &thisDay.month, &thisDay.day,
                &thisDay.year);

     nextDay = dateUpdate (thisDay);

     printf ("Tomorrow's date is %i/%i/%.2i.\n",nextDay.month,
                nextDay.day, nextDay.year % 100);

     return 0;
}
```

Program 9.4 **Output**

```
Enter today's date (mm dd yyyy): 2 28 2008
Tomorrow's date is 2/29/08.
```

Program 9.4 **Output (Rerun)**

```
Enter today's date (mm dd yyyy): 2 22 2005
Tomorrow's date is 2/23/05.
```

Inside `main`, the statement

```
next_date = dateUpdate (thisDay);
```

illustrates the ability to pass a structure to a function and to return one as well. The `dateUpdate` function has the appropriate declaration to indicate that the function returns a value of type `struct date`. Inside the function is the same code that was included in the `main` routine of Program 9.3. The functions `numberOfDays` and `isLeapYear` remain unchanged from that program.

Make certain that you understand the hierarchy of function calls in the preceding program: The `main` function calls `dateUpdate`, which in turn calls `numberOfDays`, which itself calls the function `isLeapYear`.

Initializing Structures

Initializing structures is similar to initializing arrays—the elements are simply listed inside a pair of braces, with each element separated by a comma.

To initialize the `date` structure variable `today` to July 2, 2005, the statement

```
struct date  today = { 7, 2, 2005 };
```

can be used. The statement

```
struct time  this_time = { 3, 29, 55 };
```

defines the `struct time` variable `this_time` and sets its value to 3:29:55 a.m. As with other variables, if `this_time` is a local structure variable, it is initialized each time the function is entered. If the structure variable is made static (by placing the keyword `static` in front of it), it is only initialized once at the start of program execution. In either case, the initial values listed inside the curly braces must be constant expressions.

As with the initialization of an array, fewer values might be listed than are contained in the structure. So the statement

```
struct time  time1 = { 12, 10 };
```

sets `time1.hour` to `12` and `time1.minutes` to `10` but gives no initial value to `time1.seconds`. In such a case, its default initial value is undefined.

You can also specify the member names in the initialization list. In that case, the general format is

```
.member = value
```

This method enables you to initialize the members in any order, or to only initialize specified members. For example,

```
struct time time1 = { .hour = 12, .minutes = 10 };
```

sets the `time1` variable to the same initial values as shown in the previous example. The statement

```
struct date today = { .year = 2004 };
```

sets just the year member of the date structure variable `today` to `2004`.

Compound Literals

You can assign one or more values to a structure in a single statement using what is know as *compound literals*. For example, assuming that `today` has been previously declared as a `struct date` variable, the assignment of the members of `today` as shown in Program 9.1 can also be done in a single statement as follows:

```
today = (struct date) { 9, 25, 2004 };
```

Note that this statement can appear anywhere in the program; it is not a declaration statement. The type cast operator is used to tell the compiler the type of the expression, which in this case is `struct date`, and is followed by the list of values that are to be assigned to the members of the structure, in order. These values are listed in the same way as if you were initializing a structure variable.

You can also specify values using the *`.member`* notation like this:

```
today = (struct date) { .month = 9, .day = 25, .year = 2004 };
```

The advantage of using this approach is that the arguments can appear in any order. Without explicitly specifying the member names, they must be supplied in the order in which they are defined in the structure.

The following example shows the `dateUpdate` function from Program 9.4 rewritten to take advantage of compound literals:

```
// Function to calculate tomorrow's date - using compound literals

struct date  dateUpdate (struct date  today)
{
    struct date  tomorrow;
    int  numberOfDays (struct date  d);

    if ( today.day != numberOfDays (today) )
        tomorrow = (struct date) { today.month, today.day + 1, today.year };
    else if ( today.month == 12 )      // end of year
        tomorrow = (struct date) { 1, 1, today.year + 1 };
    else                               // end of month
        tomorrow = (struct date) { today.month + 1, 1, today.year };


    return tomorrow;
}
```

Whether you decide to use compound literals in your programs is up to you. In this case, the use of compound literals makes the `dateUpdate` function easier to read.

Compound literals can be used in other places where a valid structure expression is allowed. This is a perfectly valid, albeit totally impractical example of such a use:

```
nextDay =  dateUpdate ((struct date) { 5, 11, 2004} );
```

The `dateUpdate` function expects an argument of type `struct date`, which is precisely the type of compound literal that is supplied as the argument to the function.

Arrays of Structures

You have seen how useful the structure is in enabling you to logically group related elements together. With the `time` structure, for instance, it is only necessary to keep track of one variable, instead of three, for each time that is used by the program. So, to handle 10 different times in a program, you only have to keep track of 10 different variables, instead of 30.

An even better method for handling the 10 different times involves the combination of two powerful features of the C programming language: structures and arrays. C does not limit you to storing simple data types inside an array; it is perfectly valid to define an *array of structures*. For example,

```
struct time  experiments[10];
```

defines an array called `experiments`, which consists of 10 elements. Each element inside the array is defined to be of type `struct time`. Similarly, the definition

```
struct date  birthdays[15];
```

defines the array `birthdays` to contain 15 elements of type `struct date`. Referencing a particular structure element inside the array is quite natural. To set the second birthday inside the `birthdays` array to August 8, 1986, the sequence of statements

```
birthdays[1].month = 8;
birthdays[1].day   = 8;
birthdays[1].year  = 1986;
```

works just fine. To pass the entire `time` structure contained in `experiments[4]` to a function called `checkTime`, the array element is specified:

```
checkTime (experiments[4]);
```

As is to be expected, the `checkTime` function declaration must specify that an argument of type `struct time` is expected:

```
void checkTime (struct time  t0)
{
     .
     .
     .
}
```

Initialization of arrays containing structures is similar to initialization of multidimensional arrays. So the statement

```
struct time  runTime [5] =
     {  {12, 0, 0},  {12, 30, 0},  {13, 15, 0} };
```

sets the first three times in the array `runTime` to 12:00:00, 12:30:00, and 13:15:00. The inner pairs of braces are optional, meaning that the preceding statement can be equivalently expressed as

```
struct time  runTime[5] =
     { 12, 0, 0, 12, 30, 0, 13, 15, 0 };
```

The following statement

```
struct time runTime[5] =
     { [2] = {12, 0, 0} };
```

initializes just the third element of the array to the specified value, whereas the statement

```
static struct time runTime[5] = { [1].hour = 12, [1].minutes = 30 };
```

sets just the hours and minutes of the second element of the `runTime` array to `12` and `30`, respectively.

Program 9.6 sets up an array of time structures called `testTimes`. The program then calls your `timeUpdate` function from Program 9.5. To conserve space, the `timeUpdate` function is not included in this program listing. However, a comment statement is inserted to indicate where in the program the function could be included.

In Program 9.6, an array called `testTimes` is defined to contain five different times. The elements in this array are assigned initial values that represent the times 11:59:59, 12:00:00, 1:29:59, 23:59:59, and 19:12:27, respectively. Figure 9.2 can help you to understand what the `testTimes` array actually looks like inside the computer's memory. A particular `time` structure stored in the `testTimes` array is accessed by using the appropriate index number 0–4. A particular member (`hour`, `minutes`, or `seconds`) is then accessed by appending a period followed by the member name.

For each element in the `testTimes` array, Program 9.6 displays the time as represented by that element, calls the `timeUpdate` function from Program 9.5, and then displays the updated time.

Program 9.6 **Illustrating Arrays of Structures**

```
//  Program to illustrate arrays of structures

#include <stdio.h>

struct  time
{
```

Program 9.6 **Continued**

```
    int  hour;
    int  minutes;
    int  seconds;
};

int main (void)
{
    struct time  timeUpdate (struct time  now);
    struct time  testTimes[5] =
        {  { 11, 59, 59 }, { 12, 0, 0 }, { 1, 29, 59 },
           { 23, 59, 59 }, { 19, 12, 27 }};
    int  i;

    for ( i = 0;  i < 5;  ++i )  {
        printf ("Time is %.2i:%.2i:%.2i", testTimes[i].hour,
            testTimes[i].minutes, testTimes[i].seconds);

        testTimes[i] = timeUpdate (testTimes[i]);

        printf (" ...one second later it's %.2i:%.2i:%.2i\n",
            testTimes[i].hour, testTimes[i].minutes, testTimes[i].seconds);
    }

    return 0;
}

//  ***** Include the timeUpdate function here  *****
```

Program 9.6 **Output**

```
Time is 11:59:59 ...one second later it's 12:00:00
Time is 12:00:00 ...one second later it's 12:00:01
Time is 01:29:59 ...one second later it's 01:30:00
Time is 23:59:59 ...one second later it's 00:00:00
Time is 19:12:27 ...one second later it's 19:12:28
```

The concept of an array of structures is a very powerful and important one in C. Make certain you understand it fully before you move on.

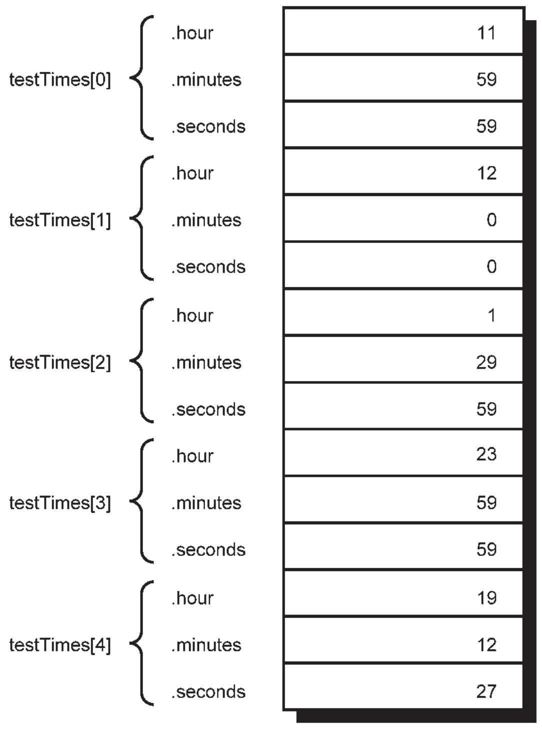


Figure 9.2 The array `testTimes` in memory.

Working with Pointers and Structures

You have seen how a pointer can be defined to point to a basic data type, such as an `int` or a `char`. But pointers can also be defined to point to structures. In Chapter 9, "Working with Structures," you defined your `date` structure as follows:

```
struct date
{
     int  month;
     int  day;
     int  year;
};
```

Just as you defined variables to be of type `struct date`, as in

```
struct date   todaysDate;
```

so can you define a variable to be a pointer to a `struct date` variable:

```
struct date  *datePtr;
```

The variable `datePtr`, as just defined, then can be used in the expected fashion. For example, you can set it to point to `todaysDate` with the assignment statement

```
datePtr = &todaysDate;
```

After such an assignment has been made, you then can indirectly access any of the members of the `date` structure pointed to by `datePtr` in the following way:

```
(*datePtr).day = 21;
```

This statement has the effect of setting the day of the `date` structure pointed to by `datePtr` to 21. The parentheses are required because the structure member operator `.` has higher precedence than the indirection operator `*`.

To test the value of `month` stored in the `date` structure pointed to by `datePtr`, a statement such as

```
if  ( (*datePtr).month == 12  )
          ...
```

can be used.

Pointers to structures are so often used in C that a special operator exists in the language. The structure pointer operator `->`, which is the dash followed by the greater than sign, permits expressions that would otherwise be written as

```
(*x).y
```

to be more clearly expressed as

```
x->y
```

So, the previous `if` statement can be conveniently written as

```
if  ( datePtr->month == 12 )
     ...
```

Program 9.1, the first program that illustrated structures, was rewritten using the concept of structure pointers, as shown in Program 11.4.

Program 11.4 **Using Pointers to Structures**

```
//  Program to illustrate structure pointers

#include <stdio.h>

int main (void)
{
     struct date
     {
          int  month;
          int  day;
          int  year;
     };

     struct date  today, *datePtr;

     datePtr = &today;

     datePtr->month = 9;
     datePtr->day = 25;
     datePtr->year = 2004;

     printf ("Today's date is %i/%i/%.2i.\n",
               datePtr->month, datePtr->day, datePtr->year % 100);

     return 0;
}
```

Program 11.4 **Output**

```
Today's date is 9/25/04.
```

Figure 11.2 depicts how the variables `today` and `datePtr` would look after all of the assignment statements from the preceding program have been executed.

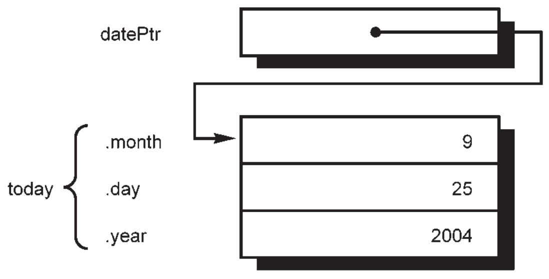


Figure 11.2 Pointer to a structure.

Once again, it should be pointed out that there is no real motivation shown here as to why you should even bother using a structure pointer when it seems as though you can get along just fine without it (as you did in Program 9.1). You will discover the motivation shortly.

The `typedef` Statement

C provides a capability that enables you to assign an alternate name to a data type. This is done with a statement known as `typedef`. The statement

```
typedef  int  Counter;
```

defines the name `Counter` to be equivalent to the C data type `int`. Variables can subsequently be declared to be of type `Counter`, as in the following statement:

```
Counter  j, n;
```

The C compiler actually treats the declaration of the variables `j` and `n`, shown in the preceding code, as normal integer variables. The main advantage of the use of the `typedef` in this case is in the added readability that it lends to the definition of the variables. It is clear from the definition of `j` and `n` what the intended purpose of these variables is in the program. Declaring them to be of type `int` in the traditional fashion would not have made the intended use of these variables at all clear. Of course, choosing more meaningful variable names would have helped as well!

In many instances, a `typedef` statement can be equivalently substituted by the appropriate `#define` statement. For example, you could have instead used the statement

```
#define  Counter  int
```

to achieve the same results as the preceding statement. However, because the `typedef` is handled by the C compiler proper, and not by the preprocessor, the `typedef` statement provides more flexibility than does the `#define` when it comes to assigning names to derived data types. For example, the following `typedef` statement:

```
typedef  char  Linebuf [81];
```

defines a type called `Linebuf`, which is an array of 81 characters. Subsequently declaring variables to be of type `Linebuf`, as in

```
Linebuf  text, inputLine;
```

has the effect of defining the variables `text` and `inputLine` to be arrays containing 81 characters. This is equivalent to the following declaration:

```
char  text[81], inputLine[81];
```

Note that, in this case, `Linebuf` could *not* have been equivalently defined with a `#define` preprocessor statement.

The following `typedef` defines a type name `StringPtr` to be a `char` pointer:

```
typedef  char *StringPtr;
```

Variables subsequently declared to be of type `StringPtr`, as in

```
StringPtr  buffer;
```

are treated as character pointers by the C compiler.

To define a new type name with `typedef`, follow these steps:

1. Write the statement as if a variable of the desired type were being declared.
2. Where the name of the declared variable would normally appear, substitute the new type name.
3. In front of everything, place the keyword `typedef`.

As an example of this procedure, to define a type called `Date` to be a structure containing three integer members called `month`, `day`, and `year`, you write out the structure definition, substituting the name `Date` where the variable name would normally appear (before the last semicolon). Before everything, you place the keyword `typedef`:

```
typedef  struct
         {
              int   month;
              int   day;
              int   year;
         } Date;
```

With this `typedef` in place, you can subsequently declare variables to be of type `Date`, as in

```
Date  birthdays[100];
```

This defines `birthdays` to be an array containing 100 `Date` structures.

When working on programs in which the source code is contained in more than one file (as described in Chapter 15, "Working with Larger Programs"), it's a good idea to place the common `typedefs` into a separate file that can be included into each source file with an `#include` statement.

As another example, suppose you're working on a graphics package that needs to deal with drawing lines, circles, and so on. You probably will be working very heavily with the coordinate system. Here's a `typedef` statement that defines a type named `Point`, where a `Point` is a structure containing two `float` members `x` and `y`:

```
typedef  struct
{
    float  x;
    float  y;
} Point;
```

You can now proceed to develop your graphics library, taking advantage of this `Point` type. For example, the declaration

```
Point  origin = { 0.0, 0.0 }, currentPoint;
```

defines `origin` and `currentPoint` to be of type `Point` and sets the `x` and `y` members of `origin` to 0.0.

Here's a function called `distance` that calculates the distance between two points.

```
#include <math.h>

double  distance (Point p1, Point p2)
{
   double  diffx, diffy;

   diffx = p1.x - p2.x;
   diffy = p1.y - p2.y;

   return sqrt (diffx * diffx + diffy * diffy);
}
```

As previously noted, `sqrt` is the square root function from the standard library. It is declared in the system header file `math.h`, thus the reason for the `#include`.

Remember, the `typedef` statement does not actually define a new type—only a new type name. So the `Counter` variables `j` and `n`, as defined in the beginning of this section, would in all respects be treated as normal `int` variables by the C compiler.

5.14 Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 151–53.

CHAPTER 7: Input and Output

Input and output facilities are not part of the C language itself, so we have not emphasized them in our presentation thus far. Nonetheless, programs interact with their environment in much more complicated ways than those we have shown before. In this chapter we will describe the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs. We will concentrate on input and output.

The ANSI standard defines these library functions precisely, so that they can exist in compatible form on any system where C exists. Programs that confine their system interactions to facilities provided by the standard library can be moved from one system to another without change.

The properties of library functions are specified in more than a dozen headers; we have already seen several of these, including `<stdio.h>`, `<string.h>`, and `<ctype.h>`. We will not present the entire library here, since we are more interested in writing C programs that use it. The library is described in detail in Appendix B.

7.1 Standard Input and Output

As we said in Chapter 1, the library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character. If the system doesn't operate that way, the library does whatever is necessary to make it appear as if it does. For instance, the library might convert carriage return and linefeed to newline on input and back again on output.

The simplest input mechanism is to read one character at a time from the *standard input*, normally the keyboard, with `getchar`:

```
int getchar(void)
```

`getchar` returns the next input character each time it is called, or `EOF` when it encounters end of file. The symbolic constant `EOF` is defined in `<stdio.h>`.

The value is typically -1, but tests should be written in terms of EOF so as to be independent of the specific value.

In many environments, a file may be substituted for the keyboard by using the < convention for input redirection: if a program prog uses getchar, then the command line

```
prog <infile
```

causes prog to read characters from infile instead. The switching of the input is done in such a way that prog itself is oblivious to the change; in particular, the string "<infile" is not included in the command-line arguments in argv. Input switching is also invisible if the input comes from another program via a pipe mechanism: on some systems, the command line

```
otherprog | prog
```

runs the two programs otherprog and prog, and pipes the standard output of otherprog into the standard input for prog.

The function

```
int putchar(int)
```

is used for output: putchar(c) puts the character c on the *standard output*, which is by default the screen. putchar returns the character written, or EOF if an error occurs. Again, output can usually be directed to a file with >*filename*: if prog uses putchar,

```
prog >outfile
```

will write the standard output to outfile instead. If pipes are supported,

```
prog | anotherprog
```

puts the standard output of prog into the standard input of anotherprog.

Output produced by printf also finds its way to the standard output. Calls to putchar and printf may be interleaved—output appears in the order in which the calls were made.

Each source file that refers to an input/output library function must contain the line

```
#include <stdio.h>
```

before the first reference. When the name is bracketed by < and > a search is made for the header in a standard set of places (for example, on UNIX systems, typically in the directory /usr/include).

Many programs read only one input stream and write only one output stream; for such programs, input and output with getchar, putchar, and printf may be entirely adequate, and is certainly enough to get started. This is particularly true if redirection is used to connect the output of one program to the input of the next. For example, consider the program lower, which converts its input to lower case:

```
#include <stdio.h>
#include <ctype.h>

main()  /* lower: convert input to lower case */
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

The function `tolower` is defined in `<ctype.h>`; it converts an upper case letter to lower case, and returns other characters untouched. As we mentioned earlier, "functions" like `getchar` and `putchar` in `<stdio.h>` and `tolower` in `<ctype.h>` are often macros, thus avoiding the overhead of a function call per character. We will show how this is done in Section 8.5. Regardless of how the `<ctype.h>` functions are implemented on a given machine, programs that use them are shielded from knowledge of the character set.

Exercise 7-1. Write a program that converts upper case to lower or lower case to upper, depending on the name it is invoked with, as found in `argv[0]`. □

5.15

Kernighan, B W and Ritchie, D M (1988) *The C Programming Language*, 2nd edn, Prentice Hall, 153–55 and 157–59.

7.2 Formatted Output—Printf

The output function `printf` translates internal values to characters. We have used `printf` informally in previous chapters. The description here covers most typical uses but is not complete; for the full story, see Appendix B.

```
int printf(char *format, arg1, arg2, ...)
```

`printf` converts, formats, and prints its arguments on the standard output under control of the `format`. It returns the number of characters printed.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to `printf`. Each conversion specification begins with a `%` and ends with a conversion character. Between the `%` and the conversion character there may be, in order:

- A minus sign, which specifies left adjustment of the converted argument.
- A number that specifies the minimum field width. The converted argument will be printed in a field at least this wide. If necessary it will be padded on the left (or right, if left adjustment is called for) to make up the field width.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits after the decimal point of a floating-point value, or the minimum number of digits for an integer.

• An h if the integer is to be printed as a short, or l (letter ell) if as a long.

Conversion characters are shown in Table 7-1. If the character after the % is not a conversion specification, the behavior is undefined.

TABLE 7-1. BASIC PRINTF CONVERSIONS

CHARACTER	ARGUMENT TYPE; PRINTED AS
d, i	int; decimal number.
o	int; unsigned octal number (without a leading zero).
x, X	int; unsigned hexadecimal number (without a leading 0x or 0X), using abcdef or ABCDEF for 10, ..., 15.
u	int; unsigned decimal number.
c	int; single character.
s	char *; print characters from the string until a '\0' or the number of characters given by the precision.
f	double; [-]*m.dddddd*, where the number of *d*'s is given by the precision (default 6).
e, E	double; [-]*m.dddddd*e±*xx* or [-]*m.dddddd*E±*xx*, where the number of *d*'s is given by the precision (default 6).
g, G	double; use %e or %E if the exponent is less than −4 or greater than or equal to the precision; otherwise use %f. Trailing zeros and a trailing decimal point are not printed.
p	void *; pointer (implementation-dependent representation).
%	no argument is converted; print a %.

A width or precision may be specified as *, in which case the value is computed by converting the next argument (which must be an int). For example, to print at most max characters from a string s,

```
printf("%.*s", max, s);
```

Most of the format conversions have been illustrated in earlier chapters. One exception is precision as it relates to strings. The following table shows the effect of a variety of specifications in printing "hello, world" (12 characters). We have put colons around each field so you can see its extent.

```
:%s:          :hello, world:
:%10s:        :hello, world:
:%.10s:       :hello, wor:
:%-10s:       :hello, world:
:%.15s:       :hello, world:
:%-15s:       :hello, world   :
:%15.10s:     :     hello, wor:
:%-15.10s:    :hello, wor     :
```

A warning: printf uses its first argument to decide how many arguments follow and what their types are. It will get confused, and you will get wrong answers, if there are not enough arguments or if they are the wrong type. You should also be aware of the difference between these two calls:

```
printf(s);          /* FAILS if s contains % */
printf("%s", s);    /* SAFE */
```

The function `sprintf` does the same conversions as `printf` does, but stores the output in a string:

```
int sprintf(char *string, char *format, arg1, arg2, ...)
```

`sprintf` formats the arguments in *arg*$_1$, *arg*$_2$, etc., according to `format` as before, but places the result in `string` instead of on the standard output; `string` must be big enough to receive the result.

Exercise 7-2. Write a program that will print arbitrary input in a sensible way. As a minimum, it should print non-graphic characters in octal or hexadecimal according to local custom, and break long text lines. □

7.4 Formatted Input—Scanf

The function `scanf` is the input analog of `printf`, providing many of the same conversion facilities in the opposite direction.

```
int scanf(char *format, ...)
```

`scanf` reads characters from the standard input, interprets them according to the specification in `format`, and stores the results through the remaining arguments. The format argument is described below; the other arguments, *each of which must be a pointer*, indicate where the corresponding converted input should be stored. As with `printf`, this section is a summary of the most useful features, not an exhaustive list.

`scanf` stops when it exhausts its format string, or when some input fails to match the control specification. It returns as its value the number of successfully matched and assigned input items. This can be used to decide how many items were found. On end of file, `EOF` is returned; note that this is different from 0, which means that the next input character does not match the first specification in the format string. The next call to `scanf` resumes searching immediately after the last character already converted.

There is also a function `sscanf` that reads from a string instead of the standard input:

int sscanf(char *string, char *format, arg_1, arg_2, ...)

It scans the `string` according to the format in `format`, and stores the resulting values through arg_1, arg_2, etc. These arguments must be pointers.

The format string usually contains conversion specifications, which are used to control conversion of input. The format string may contain:

- Blanks or tabs, which are ignored.
- Ordinary characters (not `%`), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of the character `%`, an optional assignment suppression character `*`, an optional number specifying a maximum field width, an optional `h`, `l`, or `L` indicating the width of the target, and a conversion character.

A conversion specification directs the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by the `*` character, however, the input field is skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that `scanf` will read across line boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer, as required by the call-by-value semantics of C. Conversion characters are shown in Table 7-2.

TABLE 7-2. BASIC SCANF CONVERSIONS

CHARACTER	INPUT DATA; ARGUMENT TYPE
d	decimal integer; int *.
i	integer; int *. The integer may be in octal (leading 0) or hexadecimal (leading 0x or 0X).
o	octal integer (with or without leading zero); int *.
u	unsigned decimal integer; unsigned int *.
x	hexadecimal integer (with or without leading 0x or 0X); int *.
c	characters; char *. The next input characters (default 1) are placed at the indicated spot. The normal skip over white space is suppressed; to read the next non-white space character, use %1s.
s	character string (not quoted); char *, pointing to an array of characters large enough for the string and a terminating '\0' that will be added.
e, f, g	floating-point number with optional sign, optional decimal point and optional exponent; float *.
%	literal %; no assignment is made.

The conversion characters d, i, o, u, and x may be preceded by h to indicate that a pointer to short rather than int appears in the argument list, or by l (letter ell) to indicate that a pointer to long appears in the argument list. Similarly, the conversion characters e, f, and g may be preceded by l to indicate that a pointer to double rather than float is in the argument list.

As a first example, the rudimentary calculator of Chapter 4 can be written with scanf to do the input conversion:

```
#include    <stdio.h>

main()  /* rudimentary calculator */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Suppose we want to read input lines that contain dates of the form

```
25 Dec 1988
```

The scanf statement is

```
    int day, year;
    char monthname[20];

    scanf("%d %s %d", &day, monthname, &year);
```

No & is used with monthname, since an array name is a pointer.

Literal characters can appear in the scanf format string; they must match the same characters in the input. So we could read dates of the form mm/dd/yy with this scanf statement:

```
    int day, month, year;

    scanf("%d/%d/%d", &month, &day, &year);
```

scanf ignores blanks and tabs in its format string. Furthermore, it skips over white space (blanks, tabs, newlines, etc.) as it looks for input values. To read input whose format is not fixed, it is often best to read a line at a time, then pick it apart with sscanf. For example, suppose we want to read lines that might contain a date in either of the forms above. Then we could write

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line);    /* 25 Dec 1988 form */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line);    /* mm/dd/yy form */
    else
        printf("invalid: %s\n", line);  /* invalid form */
}
```

Calls to scanf can be mixed with calls to other input functions. The next call to any input function will begin by reading the first character not read by scanf.

A final warning: the arguments to scanf and sscanf *must* be pointers. By far the most common error is writing

```
    scanf("%d", n);
```

instead of

```
    scanf("%d", &n);
```

This error is not generally detected at compile time.

Exercise 7-4. Write a private version of scanf analogous to minprintf from the previous section. □

Exercise 7-5. Rewrite the postfix calculator of Chapter 4 to use scanf and/or sscanf to do the input and number conversion. □