11

Character Strings and String Functions

You will learn about the following in this chapter:

- Functions:

  `gets()`, `gets_s()`, `fgets()`, `puts()`, `fputs()`, `strcat()`, `strncat()`, `strcmp()`, `strncmp()`, `strcpy()`, `strncpy()`, `sprintf()`, `strchr()`
- Creating and using strings
- Using several string and character functions from the C library and creating your own string functions
- Using command-line arguments

The character string is one of the most useful and important data types in C. You have been using character strings all along, but there still is much to learn about them. The C library provides a wide range of functions for reading and writing strings, copying strings, comparing strings, combining strings, searching strings, and more. This chapter will add these capabilities to your programming skills.

Representing Strings and String I/O

Of course, you already know the most basic fact: A *character string* is a `char` array terminated with a null character (`\0`). Therefore, what you've learned about arrays and pointers carries over to character strings. But because character strings are so commonly used, C provides many functions specifically designed to work with strings. This chapter discusses the nature of strings, how to declare and initialize strings, how to get them into and out of programs, and how to manipulate strings.

Let's look at a short program (see Listing 11.1) that illustrates some of the ways to represent strings in a program.

Listing 11.1 **The `strings1.c` Program**

```
//  strings1.c
#include <stdio.h>
#define MSG "I am a symbolic string constant."
#define MAXLENGTH 81
int main(void)
{
    char words[MAXLENGTH] = "I am a string in an array.";
    const char * pt1 = "Something is pointing at me.";
    puts("Here are some strings:");
    puts(MSG);
    puts(words);
    puts(pt1);
    words[8] = 'p';
    puts(words);

    return 0;
}
```

The `puts()` function, like `printf()`, belongs to the the `stdio.h` family of input/output functions. It only displays strings, and, unlike `printf()`, it automatically appends a newline to the string it displays. Here's the output:

```
Here are some strings:
I am an old-fashioned symbolic string constant.
I am a string in an array.
Something is pointing at me.
I am a spring in an array.
```

Rather than going through Listing 11.1 line-by-line, let's take a more encompassing approach. First, we will look at ways of defining a string within a program. Then we will see what is involved in reading a string into a program. Finally, we will study ways to output a string.

Defining Strings Within a Program

As you probably noticed when you read Listing 11.1, there are many ways to define a string. The principal ways are using string constants, using `char` arrays, and using `char` pointers. A program should make sure there is a place to store a string, and we will cover that topic, too.

Character String Literals (String Constants)

A *string literal*, also termed a *string constant*, is anything enclosed in double quotation marks. The enclosed characters, plus a terminating `\0` character automatically provided by the compiler, are stored in memory as a character string. So `"I am a symbolic string constant."`, `"I am a string in an array."`, `"Something is pointed at me."`, and `"Here are some strings:"` all are string literals.

Recall that, beginning with the ANSI C standard, C concatenates string literals if they are separated by nothing or by whitespace. For example,

```
char greeting[50] = "Hello, and"" how are"  " you"
                 " today!";
```

is equivalent to this:

```
char greeting[50] = "Hello, and how are you today!";
```

If you want to use a double quotation mark within a string, precede the quotation mark with a backslash, as follows:

```
printf("\"Run, Spot, run!\" exclaimed Dick.\n");
```

This produces the following output:

```
"Run, Spot, run!" exclaimed Dick.
```

Character string constants are placed in the *static storage* class, which means that if you use a string constant in a function, the string is stored just once and lasts for the duration of the program, even if the function is called several times. The entire quoted phrase acts as a pointer to where the string is stored. This action is analogous to the name of an array acting as a pointer to the array's location. If this is true, what kind of output should the program in Listing 11.2 produce?

Listing 11.2 **The `strptr.c` Program**

```
/* strptr.c -- strings as pointers */
#include <stdio.h>
int main(void)
{
    printf("%s, %p, %c\n", "We", "are", *"space farers");

    return 0;
}
```

The `%s` format should print the string `We`. The `%p` format produces an address. So if the phrase `"are"` is an address, then `%p` should print the address of the first character in the string. (Pre-ANSI implementations might have to use `%u` or `%lu` instead of `%p`.) Finally, `*"space farers"` should produce the value to which the address points, which should be the first character of the string `"space farers"`. Does this really happen? Well, here is the output:

```
We, 0x100000f61, s
```

Character String Arrays and Initialization

When you define a character string array, you must let the compiler know how much space is needed. One way is to specify an array size large enough to hold the string. The following declaration initializes the array `m1` to the characters of the indicated string:

```
const char m1[40] = "Limit yourself to one line's worth.";
```

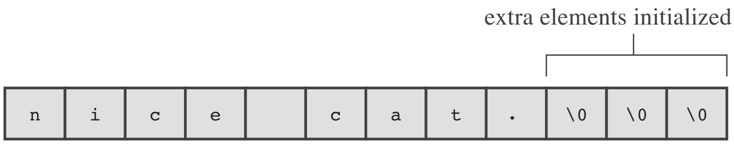
The `const` indicates the intent to not alter this string.

This form of initialization is short for the standard array initialization form:

```
const char m1[40] = {  'L',
'i', 'm', 'i', 't', ' ', 'y', 'o', 'u', 'r', 's', 'e', 'l',
'f', ' ', 't', 'o', ' ', 'o', 'n', 'e', ' ',
'l', 'i', 'n', 'e', '\", 's', ' ', 'w', 'o', 'r',
't', 'h', '.', '\0'
};
```

Note the closing null character. Without it, you have a character array, but not a string.

When you specify the array size, be sure that the number of elements is at least one more (that null character again) than the string length. Any unused elements are automatically initialized to `0` (which in `char` form is the null character, not the zero digit character). See Figure 11.1.



```
const char pets[12] = "nice cat.";
```

Figure 11.1 Initializing an array.

Often, it is convenient to let the compiler determine the array size; recall that if you omit the size in an initializing declaration, the compiler determines the size for you:

```
const char m2[] = "If you can't think of anything, fake it.";
```

Initializing character arrays is one case when it really does make sense to let the compiler determine the array size. That's because string-processing functions typically don't need to know the size of the array because they can simply look for the null character to mark the end.

Letting the compiler compute the size of the array works only if you initialize the array. If you create an array you intend to fill later, you need to specify the size when you declare it. When you do declare an array size, the array size must evaluate to an integer. Prior to the advent of variable length arrays (VLAs) with C99, the size had to be an integer constant, which includes the possibility of an expression formed from constant integer values.

```
int n = 8;
char cookies[1];    // valid
char cakes[2 + 5];  // valid, size is a constant expression
char pies[2*sizeof(long double) + 1];  // valid
char crumbs[n];     // invalid prior to C99, a VLA after C99
```

The name of a character array, like any array name, yields the address of the first element of the array. Therefore, the following holds:

```
char car[10] = "Tata";
car == &car[0] , *car == 'T', and *(car+1) == car[1] == 'a'
```

Indeed, you can use pointer notation to set up a string. For example, Listing 11.1 uses the following declaration:

```
const char * pt1 = "Something is pointing at me.";
```

This declaration is very nearly the same as this one:

```
const char ar1[] = "Something is pointing at me.";
```

The declarations amount to saying that both `pt1` and `ar1` are addresses of strings. In both cases, the quoted string itself determines the amount of storage set aside for the string. Nonetheless, the forms are not identical.

Array Versus Pointer

What is the difference, then, between an array and a pointer form? The array form (`ar1[]`) causes an array of 29 elements (one for each character plus one for the terminating `'\0'`) to be allocated in the computer memory. Each element is initialized to the corresponding character of the string literal. Typically, what happens is that the quoted string is stored in a data segment that is part of the executable file; when the program is loaded into memory, so is that string. The quoted string is said to be in *static memory*. But the memory for the array is allocated only after the program begins running. At that time, the quoted string is copied into the array. (Chapter 12, “Storage Classes, Linkage, and Memory Management,” will discuss memory management more fully.) Note that, at this time, there are two copies of the string. One is the string literal in static memory, and one is the string stored in the `ar1` array.

Hereafter, the compiler will recognize the name `ar1` as a synonym for the address of the first array element, `&ar1[0]`. One important point here is that in the array form, `ar1` is an address *constant*. You can’t change `ar1`, because that would mean changing the location (address) where the array is stored. You can use operations such as `ar1+1` to identify the next element in an array, but `++ar1` is not allowed. The increment operator can be used only with the names of variables (or, more generally, modifiable lvalues), not with constants.

The pointer form (`*pt1`) also causes 29 elements in static storage to be set aside for the string. In addition, once the program begins execution, it sets aside one more storage location for the pointer *variable* `pt1` and stores the address of the string in the pointer variable. This variable initially points to the first character of the string, but the value can be changed. Therefore, you can use the increment operator. For instance, `++pt1` would point to the second character (`o`).

A string literal is considered to be `const` data. Because `pt1` points to that data, it should be declared as pointing to `const` data. This doesn't mean you can't change the value of `pt1` (i.e., where it points), but it does mean you can't use `pt1` to change the data itself. If you copy a string literal to an array, on the other hand, you are free to change the data unless you choose to declare the array as `const`.

In short, initializing the array copies a string from static storage to the array, whereas initializing the pointer merely copies the address of the string. Listing 11.3 illustrates these points.

Listing 11.3 The `addresses.c` Program

```
//  addresses.c  -- addresses of strings
#define MSG "I'm special."

#include <stdio.h>
int main()
{
    char ar[] = MSG;
    const char *pt = MSG;
    printf("address of \"I'm special\": %p \n", "I'm special");
    printf("              address ar: %p\n", ar);
    printf("              address pt: %p\n", pt);
    printf("          address of MSG: %p\n", MSG);
    printf("address of \"I'm special\": %p \n", "I'm special");

    return 0;
}
```

Here's the output from one system:

```
address of "I'm special": 0x100000f0c
              address ar: 0x7fff5fbff8c7
              address pt: 0x100000ee0
          address of MSG: 0x100000ee0
address of "I'm special": 0x100000f0c
```

What does this show? First, `pt` and `MSG` are the same address, while `ar` is a different address, just as promised. Second, although the string literal `"I'm special."` occurs twice in the `printf()` statements, the compiler chose to use one storage location, but not the same address as `MSG`. The compiler has the freedom to store a literal that's used more than once in one or more locations. Another compiler might choose to represent all three occurrences of `"I'm special."` with a single storage location. Third, the part of memory used for static data is different from that used for dynamic memory, the memory used for `ar`. Not only are the values different, but this particular compiler even uses a different number of bits to represent the two kinds of memory.

Are the differences between array and pointer representations of strings important? Often they are not, but it depends on what you try to do. Let's look further into the matter.

Array and Pointer Differences

Let's examine the differences between initializing a character array to hold a string and initializing a pointer to point to a string. (By "pointing to a string," we really mean pointing to the first character of a string.) For example, consider these two declarations:

```
char heart[] = "I love Tillie!";
const char *head = "I love Millie!";
```

The chief difference is that the array name `heart` is a constant, but the pointer `head` is a variable. What practical difference does this make?

First, both can use array notation:

```
for (i = 0; i < 6; i++)
    putchar(heart[i]);
putchar('\n');
for (i = 0; i < 6; i++)
    putchar(head[i]));
putchar('\n');
```

This is the output:

```
I love
I love
```

Next, both can use pointer addition:

```
for (i = 0; i < 6; i++)
    putchar(*(heart + i));
putchar('\n');
for (i = 0; i < 6; i++)
    putchar(*(head + i));
putchar('\n');
```

Again, the output is as follows:

```
I love
I love
```

Only the pointer version, however, can use the increment operator:

```
while (*(head) != '\0')  /* stop at end of string           */
    putchar(*(head++));  /* print character, advance pointer */
```

This produces the following output:

```
I love Millie!
```

Suppose you want head to agree with heart. You can say

```
head = heart;  /* head now points to the array heart */
```

This makes the head pointer point to the first element of the heart array.

However, you cannot say

```
heart = head;  /* illegal construction */
```

The situation is analogous to x = 3; versus 3 = x;. The left side of the assignment statement must be a variable or, more generally, a modifiable *lvalue*, such as *p_int. Incidentally, head = heart; does not make the Millie string vanish; it just changes the address stored in head. Unless you've saved the address of "I love Millie!" elsewhere, however, you won't be able to access that string when head points to another location.

There is a way to alter the heart message—go to the individual array elements:

```
heart[7]= 'M';
```

or

```
*(heart + 7) = 'M';
```

The *elements* of an array are variables (unless the array was declared as const), but the *name* is not a variable.

Let's go back to a pointer initialization that doesn't use the const modifier:

```
char * word = "frame";
```

Can you use the pointer to change this string?

```
word[1] = 'l';  // allowed??
```

Your compiler may allow this, but, under the current C standard, the behavior for such an action is undefined. Such a statement could, for example, lead to memory access errors. The reason is that, as mentioned before, a compiler can choose to represent all identical string literals with a single copy in memory. For example, the following statements could all refer to a single memory location of string "Klingon":

```
char * p1 = "Klingon";
p1[0] = 'F';    // ok?
printf("Klingon");
printf(": Beware the %ss!\n", "Klingon");
```

That is, the compiler can replace each instance of "Klingon" with the same address. If the compiler uses this single-copy representation and allows changing p1[0] to 'F', that would affect all uses of the string, so statements printing the string literal "Klingon" would actually display "Flingon":

```
Flingon: Beware the Flingons!
```

In fact, in the past, several compilers did behave this rather confusing way, whereas others produced programs that abort. Therefore, the recommended practice for initializing a pointer to a string literal is to use the const modifier:

```
const char * pl = "Klingon";  // recommended usage
```

Initializing a non-const array with a string literal, however, poses no such problems, because the array gets a copy of the original string.

In short, don't use a pointer to a string literal if you plan to alter the string.

Arrays of Character Strings

It is often convenient to have an array of character strings. Then you can use a subscript to access several different strings. Listing 11.4 shows two approaches: an array of pointers to strings and an array of char arrays.

Listing 11.4 **The arrchar.c Program**

```
//  arrchar.c -- array of pointers, array of strings
#include <stdio.h>
#define SLEN 40
#define LIM 5
int main(void)
{
    const char *mytalents[LIM] = {
        "Adding numbers swiftly",
        "Multiplying accurately", "Stashing data",
        "Following instructions to the letter",
        "Understanding the C language"
    };
    char yourtalents[LIM][SLEN] = {
        "Walking in a straight line",
        "Sleeping", "Watching television",
        "Mailing letters", "Reading email"
    };
    int i;

    puts("Let's compare talents.");
    printf ("%-36s  %-25s\n", "My Talents", "Your Talents");
    for (i = 0; i < LIM; i++)
        printf("%-36s  %-25s\n", mytalents[i], yourtalents[i]);
```

```
   printf("\nsizeof mytalents: %zd, sizeof yourtalents: %zd\n",
           sizeof(mytalents), sizeof(yourtalents));

    return 0;
}
```

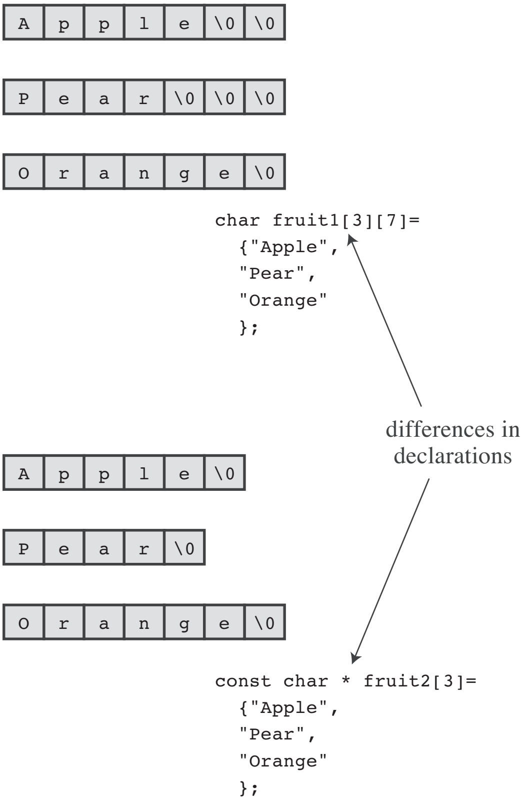
Here is the output:

```
Let's compare talents.
My Talents                            Your Talents
Adding numbers swiftly                Walking in a straight line
Multiplying accurately                Sleeping
Stashing data                         Watching television
Following instructions to the letter  Mailing letters
Understanding the C language          Reading email

sizeof mytalents: 40, sizeof yourtalents: 200
```

In some ways, `mytalents` and `yourtalents` are much alike. Each represents five strings. When used with one index, as in `mytalents[0]` and `yourtalents[0]`, the result is a single string. And, just as `mytalents[1][2]` is `'l'`, the third character of the second string represented by `mytalents`, `yourtalents[1][2]` is `'e'`, the third character of the second string represented by `yourtalents`. Both are initialized in the same fashion.

But there are differences, too. The `mytalents` array is an array of five pointers, taking up 40 bytes on our system. But `yourtalents` is an array of five arrays, each of 40 `char` values, occupying 200 bytes on our system. So `mytalents` is a different type from `yourtalents`, even though `mytalents[0]` and `yourtalents[0]` both are strings. The pointers in `mytalents` point to the locations of the string literals used for initialization, which are stored in static memory. The arrays in `yourtalents`, however, contain copies of the string literals, so each string is stored twice. Furthermore, the allocation of memory in the arrays is inefficient, for each element of `yourtalents` has to be the same size, and that size has to be at least large enough to hold the longest string.

One way of visualizing this difference is to think of `yourtalents` as a rectangular two-dimensional array, with each row being of the same length, 40 bytes, in this case. Next, think of `mytalents` as a ragged array, one in which the row length varies. Figure 11.2 shows the two kinds of arrays. (Actually, the strings pointed to by the `mytalents` array elements don't necessarily have to be stored consecutively in memory, but the figure does illustrate the difference in storage requirements.)



A	p	p	l	e	\0	\0
P	e	a	r	\0	\0	\0
O	r	a	n	g	e	\0

```
char fruit1[3][7]=
  {"Apple",
  "Pear",
  "Orange"
  };
```

A	p	p	l	e	\0	
P	e	a	r	\0		
O	r	a	n	g	e	\0

```
const char * fruit2[3]=
  {"Apple",
  "Pear",
  "Orange"
  };
```

Figure 11.2 Rectangular versus ragged array.

The upshot is that, if you want to use an array to represent a bunch of strings to be displayed, an array of pointers is more efficient than an array of character arrays. There is, however, a catch. Because the pointers in `mytalents` point to string literals, these strings shouldn't be altered. The contents of `yourtalents`, however, can be changed. So if you want to alter strings or set aside space for string input, don't use pointers to string literals.

Pointers and Strings

Perhaps you noticed an occasional reference to pointers in this discussion of strings. Most C operations for strings actually work with pointers. Consider, for example, the instructive program shown in Listing 11.5.

Listing 11.5 **The `p_and_s.c` Program**

```
/* p_and_s.c -- pointers and strings */
#include <stdio.h>
int main(void)
{
    const char * mesg = "Don't be a fool!";
    const char * copy;

    copy = mesg;
    printf("%s\n", copy);
    printf("mesg = %s; &mesg = %p; value = %p\n",
            mesg, &mesg, mesg);
    printf("copy = %s; &copy = %p; value = %p\n",
            copy, &copy, copy);

    return 0;
}
```

> **Note**
>
> Use `%u` or `%lu` instead of `%p` if your compiler doesn't support `%p`.

Looking at this program, you might think that it makes a copy of the string `"Don't be a fool!"`, and your first glance at the output might seem to confirm this guess:

```
Don't be a fool!
mesg = Don't be a fool!; &mesg = 0x0012ff48; value = 0x0040a000
copy = Don't be a fool!; &copy = 0x0012ff44; value = 0x0040a000
```

But study the `printf()` output more carefully. First, `mesg` and `copy` are printed as strings (`%s`). No surprises here; all the strings are `"Don't be a fool!"`.

The next item on each line is the address of the specified pointer. For this particular run, the two pointers `mesg` and `copy` are stored in locations `0x0012ff48` and `0x0012ff44`, respectively.

Now notice the final item, the one we called `value`. It is the value of the specified pointer. The value of the pointer is the address it contains. You can see that `mesg` points to location `0x0040a000`, and so does `copy`. Therefore, the string itself was never copied. All that `copy = mesg;` does is produce a second pointer pointing to the very same string.

Why all this pussyfooting around? Why not just copy the whole string? Well, ask yourself which is more efficient: copying one address or copying, say, 50 separate elements? Often, the address is all that is needed to get the job done. If you truly require a copy that is a duplicate, you can use the `strcpy()` or `strncpy()` function, discussed later in this chapter.

Now that we have discussed defining strings within a program, let's turn to strings provided by keyboard input.

String Input

If you want to read a string into a program, you must first set aside space to store the string and then use an input function to fetch the string.

Creating Space

The first order of business is setting up a place to put the string after it is read. As mentioned earlier, this means you need to allocate enough storage to hold whatever strings you expect to read. Don't expect the computer to count the string length as it is read and then allot space for it. The computer won't (unless you write a function to do so). For example, suppose you try something like this:

```
char *name;

scanf("%s", name);
```

It will probably get by the compiler, most likely with a warning, but when the name is read, the name might be written over data or code in your program, and it might cause a program abort. That's because `scanf()` copies information to the address given by the argument, and in this case, the argument is an uninitialized pointer; `name` might point anywhere. Most programmers regard this as highly humorous, but only in other people's programs.

The simplest course is to include an explicit array size in the declaration:

```
char name[81];
```

Now `name` is the address of an allocated block of 81 bytes. Another possibility is to use the C library functions that allocate memory, and we'll touch on those in Chapter 12.

After you have set aside space for the string, you can read the string. The C library supplies a trio of functions that can read strings: `scanf()`, `gets()`, and `fgets()`. The most commonly used one has been `gets()`, which we discuss first.

The Unfortunate `gets()` Function

Recall that, when reading a string, `scanf()` and the `%s` specifier read just a single word. Often it's useful if a program can read an entire line of input at a time instead of a single word. For many years, the `gets()` function has served that purpose. It's a simple function, easy to use. It reads an entire line up through the newline character, discards the newline character, stores the remaining characters, adding a null character to create a C string. It's often paired with `puts()`, which displays a string, adding a newline. Listing 11.6 presents a modest example.

Listing 11.6 **The `getsputs.c` Program**

```
/*  getsputs.c  -- using gets() and puts() */
#include <stdio.h>
```

```
#define STLEN 81
int main(void)
{
    char words[STLEN];


    puts("Enter a string, please.");
    gets(words);
    printf("Your string twice:\n");
    printf("%s\n", words);
    puts(words);
    puts("Done.");

    return 0;
}
```

Here's a sample run, or, at least what once would have been a sample run:

```
Enter a string, please.
I want to learn about string theory!
Your string twice:
I want to learn about string theory!
I want to learn about string theory!
Done.
```

Note that the entire line of input, aside from the newline, is stored in `words` and that `puts(words)` has the same effect as `printf("%s\n", words)`.

Next, here is a more contemporary sample run:

```
Enter a string, please.
warning: this program uses gets(), which is unsafe.
Oh, no!
Your string twice:
Oh, no!
Oh, no!
Done.
```

The compiler has taken the rather unusual action of inserting a warning into the program output! So this message gets displayed every time you or anyone else runs the program. Not all compilers will do this. Others may issue a warning during the compiling process, but that isn't quite as attention getting.

So what's the problem? The problem is that `gets()` doesn't check to see if the input line actually fits into the array. Given that its only argument here is `words`, `gets()` can't check. Recall that the name of an array is converted to the address of the first element. Thus `gets()` only knows where the array begins, not how many elements it has.

If the input string is too long, you get *buffer overflow*, meaning the excess characters overflow the designated target. The extra characters might just go into unused memory and cause no immediate problems, or they may overwrite other data in your program, but those certainly aren't the only possibilities. Here's a sample run for which `SLEN` was reset to 5 to make it easier to overflow the buffer:

```
Enter a string, please.
warning: this program uses gets(), which is unsafe.
I think I'll be just fine.
Your string twice:
I think I'll be just fine.
I think I'll be just fine.
Done.
Segmentation fault: 11
```

"Segmentation fault" doesn't sound healthy, and it isn't. On a Unix system, this message indicates the program attempted to access memory not allocated to it.

But C provides many paths for poor programming to lead to embarrassing and difficult to trace failures. Why, then, single out `gets()` for special mention? Probably because its unsafe behavior poses a security risk. In the past, people have taken advantage of system programming that uses `gets()` to insert and run code that compromised system security.

For a while, many in the C programming community have recommended banishing `gets()` from the programming vocabulary. The committee that created the C99 standard also published a rationale for the standard. This rationale acknowledged the problems with `gets()` and discouraged its use. However, it justified keeping `gets()` as part of the standard because it was a convenient function, in the right circumstances, and because it was part of much existing code.

The C11 committee, however, has taken a tougher view and has dropped `gets()` from the standard. However, a standard establishes what a compiler must support, not what it must not support. In practice, most compilers will continue to provide the function in the interests of backwards compatibility. But, as with the compiler we used, they don't have to be happy about it.

The Alternatives to `gets()`

The traditional alternative to `gets()` is `fgets()`, which has a slightly more complex interface and which handles input slightly differently. The C11 standard adds `gets_s()` to the mix. It's a bit more like `gets()` and is more easily substituted into existing code as a replacement. However, it's part of an optional extension to the `stdio.h` family of input/output functions, so C11 C compilers need not support it.

The `fgets()` Function (and `fputs()`)

The `fgets()` function meets the possible overflow problem by taking a second argument that limits the number of characters to be read. This function is designed for file input, which makes it a little more awkward to use. Here is how `fgets()` differs from `gets()`:

- It takes a second argument indicating the maximum number of characters to read. If this argument has the value `n`, `fgets()` reads up to `n-1` characters or through the newline character, whichever comes first.
- If `fgets()` reads the newline, it stores it in the string, unlike `gets()`, which discards it.
- It takes a third argument indicating which file to read. To read from the keyboard, use `stdin` (for *standard input*) as the argument; this identifier is defined in `stdio.h`.

Because the `fgets()` function includes the newline as part of the string (assuming the input line fits), it's often paired with `fputs()`, which works like `puts()`, except that it doesn't automatically append a newline. It takes a second argument to indicate which file to write to. For the computer monitor we can use `stdout` (for standard output) as an argument. Listing 11.7 illustrates how `fgets()` and `fputs()` behave.

Listing 11.7 **The `fgets1.c` Program**

```
/*  fgets1.c  -- using fgets() and fputs() */
#include <stdio.h>
#define STLEN 14
int main(void)
{
    char words[STLEN];

    puts("Enter a string, please.");
    fgets(words, STLEN, stdin);
    printf("Your string twice (puts(), then fputs()):\n");
    puts(words);
    fputs(words, stdout);
    puts("Enter another string, please.");
    fgets(words, STLEN, stdin);
    printf("Your string twice (puts(), then fputs()):\n");
    puts(words);
    fputs(words, stdout);
    puts("Done.");

    return 0;
}
```

Here's a sample run:

```
Enter a string, please.
apple pie
```

```
Your string twice (puts(), then fputs()):
apple pie

apple pie
Enter another string, please.
strawberry shortcake
Your string twice (puts(), then fputs()):
strawberry sh
strawberry shDone.
```

The first input, `apple pie`, is short enough that `fgets()` reads the whole input line and stores `apple pie\n\0` in the array. So when `puts()` displays the string and adds its own newline to the output, it produces a blank output line after `apple pie`. Because `fputs()` doesn't add a newline, it doesn't produce a blank line.

The second input line, `strawberry shortcake`, exceeds the size limit, so `fgets()` reads the first 13 characters and stores `strawberry sh\0` in the array. Again, `puts()` adds a newline to the output and `fputs()` doesn't.

The `fgets()` function returns a pointer to `char`. If all goes well, it just returns the same address that was passed to it as the first argument. If the function encounters end-of-file, however, it returns a special pointer called the *null pointer*. This is a pointer guaranteed not to point to valid data so it can be used to indicate a special case. In code it can be represented by the digit `0` or, more commonly in C, by the macro `NULL`. (The function also returns `NULL` if there is some sort of read error.) Listing 11.8 shows a simple loop that reads and echoes text until `fgets()` encounters end-of-file or until it reads a blank line, indicated by the first character being a newline character.

Listing 11.8 **The `fgets2.c` Program**

```
/*  fgets2.c  -- using fgets() and fputs() */
#include <stdio.h>
#define STLEN 10
int main(void)
{
    char words[STLEN];

    puts("Enter strings (empty line to quit):");
    while (fgets(words, STLEN, stdin) != NULL && words[0] != '\n')
        fputs(words, stdout);
    puts("Done.");

    return 0;
}
```

Here's a sample run:

```
Enter strings (empty line to quit):
By the way, the gets() function
By the way, the gets() function
also returns a null pointer if it
also returns a null pointer if it
encounters end-of-file.
encounters end-of-file.

Done.
```

Interesting—even though `STLEN` is 10, the program seems to have no problem processing input lines much longer than that. What's happening is that, in this program, `fgets()` reads in input `STLEN - 1` (i.e., 9) characters at a time. So it begins by reading "By the wa", storing it as `By the wa\0`. Then `fputs()` displays this string and does not advance to the next output line. Next, `fgets()` resumes where it left off on the original input, that is, it reads "y, the ge" and stores it as `y, the ge\0`. Then `fputs()` displays it on the same line it used before. Then `fgets()` resumes reading the input, and so on, until all that's left is "tion\n"; `fgets()` stores `tion\n\0`, `fputs()` displays it, and the embedded newline character moves the cursor to the next line.

The system uses buffered I/O. This means the input is stored in temporary memory (the buffer) until the Return key is pressed; this adds a newline character to the input and sends the whole line on to `fgets()`. On output, `fputs()` sends characters to another buffer, and when a newline is sent, the buffer contents are sent on to the display.

The fact that `fgets()` stores the newline presents a problem and an opportunity. The problem is that you might not want the newline as part of the string you store. The opportunity is the presence or absence of a newline character in the stored string can be used to tell whether the whole line was read. If it wasn't, then you can decide what to do with the rest of the line.

First, how can you get rid of a newline? One way is to search the stored string for a newline and to replace it with a null character:

```
while (words[i] != '\n') // assuming \n in words
    i++;
words[i] = '\0';
```

Second, what if there are still characters left in the input line? One reasonable choice if the whole line doesn't fit into the destination array is to discard the part that doesn't fit:

```
while (getchar() != '\n')  // read but don't store
    continue;              // input including \n
```

Listing 11.9 adds a little more testing to these basic ideas to produce code that reads lines of inputs, removes the stored newlines, if any, and discards the part of a line that doesn't fit.

Listing 11.9 **The `fgets3.c` Program**

```
/*  fgets3.c  -- using fgets() */
#include <stdio.h>
#define STLEN 10
int main(void)
{
    char words[STLEN];
    int i;

    puts("Enter strings (empty line to quit):");
    while (fgets(words, STLEN, stdin) != NULL
                          && words[0] != '\n')
    {
        i = 0;
        while (words[i] != '\n' && words[i] != '\0')
            i++;
        if (words[i] == '\n')
            words[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
        puts(words);
    }
    puts("done");
    return 0;
}
```

The loop

```
while (words[i] != '\n' && words[i] != '\0')
    i++;
```

marches through the string until reaching a newline or null character, whichever comes first. If that character is a newline, the following `if` statement replaces it with a null character. Otherwise, the `else` part disposes of the rest of the input line. Here is sample run:

```
Enter strings (empty line to quit):
This
This
program seems
program s
unwilling to accept long lines.
unwilling
But it doesn't get stuck on long
But it do
```

```
lines either.
lines eit

done
```

Null and Null

Null character and null pointer both appear in Listing 11.9. Conceptually, these two nulls are different from one another. The null character, or `'\0'`, is the character used to mark the end of a C string. It's the character whose code is zero. Because that isn't the code of any character, it won't show up accidentally in some other part of the string.

The null pointer, or `NULL`, has a value that doesn't correspond to a valid address of data. It's often used by functions that otherwise return valid addresses to indicate some special occurrence, such as encountering end-of-file or failing to perform as expected.

So the null character is an integer type, while the null pointer is a pointer type. What sometimes causes confusion is that both can be represented numerically by the value 0. But, conceptually, they are different types of 0. Also, while the null character, being a character, is one byte, the null pointer, being an address, typically is four bytes.

The `gets_s()` Function

C11's optional `gets_s()` function, like `fgets()`, uses an argument to limit the number of characters read. Given the same definitions used in Listing 11.9, the following code would read a line of input into the `words` array providing the newline shows up in the first 9 characters of input:

```
gets_s(words, STLEN);
```

The three main differences from `fgets()` are these:

- `gets_s()` just reads from the standard input, so it doesn't need a third argument.
- If `gets_s()` does read a newline; it discards it rather than storing it.
- If `gets_s()` reads the maximum number of characters and fails to read a newline, it takes several steps. It sets the first character of the destination array to the null character. It reads and discards subsequent input until a newline or end-of-file is encountered. It returns the null pointer. It invokes an implementation-dependent "handler" function (or else one you've selected), which may cause the program to exit or abort.

The second feature means that, as long as the input line isn't too long, `gets_s()` behaves like `gets()`, making it easier to replace `gets()` with `gets_s()` rather than with `fgets()`. The third feature means there's a learning curve to using this function.

Let's compare the suitability of `gets()`, `fgets()`, and `gets_s()`. If the input line fits into the target storage, all three work fine. But `fgets()` does include the newline as part of the string, and you may need to provide code to replace it with a null character.

What if the input line doesn't fit? Then gets() isn't safe; it can corrupt your data and compromise security. The gets_s() function is safe, but, if you don't want the program to abort or otherwise exit, you'll need to learn how to write and register special "handlers." Also, if you manage to keep the program running, gets_s() disposes of the rest of the input line whether you want to or not. The fgets() function is the easiest to work with if the line doesn't fit, and it leaves more choices up to you. If you want the program to process the rest of the input line, you can, as Listing 11.8 shows. If, instead, you want to dispose of the rest of the input line, you can do that, too, as Listing 11.9 shows.

So gets_s(), when input fails to meet expectations, is less convenient and flexible than fgets(). Perhaps that's one reason that gets_s() is just an optional extension of the C library. And given that gets_s() is optional, using fgets() usually is the better choice.

The s_gets() Function

Listing 11.9 presented one way to use fgets(): Read a whole line and replace the newline character with a null character, or read the part of a line that fits and discard the rest—sort of a gets_s() function without the extra baggage. No standard function meets that description, but we can create one. It'll come in handy in later examples. Listing 11.10 shows one approach.

Listing 11.10 **The s_gets() Function**

```
char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)  // i.e., ret_val != NULL
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

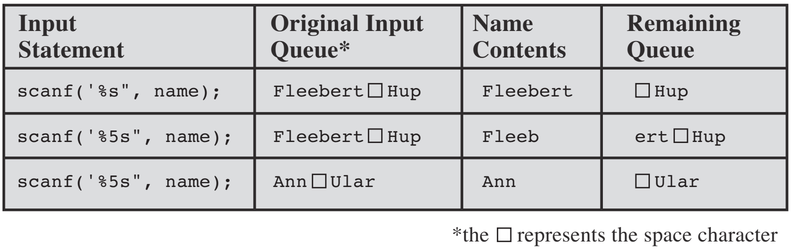
If fgets() returns NULL, indicating end-of-file or a read error, s_gets() skips the rest of the processing. Otherwise, it imitates Listing 11.9, replacing the newline character with a null character if the former is present in the string, and discarding the rest of the line otherwise. It then returns the same value fgets() returned. We'll use this function in later examples.

Perhaps you are wondering what's the rationale for discarding the rest of a too-long line. The problem is that if the remainder of the line is left in place, it becomes the input for the next read statement. This can, for example, cause the program to crash if the next read statement is looking for a type `double` value. Discarding the rest of the line keeps the read statements synchronized with the keyboard input.

Our `s_gets()` function isn't perfect. Its most serious flaw is that it is silent about encountering input that doesn't fit. It discards the extra input with neither the program nor the user being informed, thus closing off other options, such as having the user try again or finding more storage space. Another flaw is that it doesn't cope with misuse such as being passed a size of 1 or less. But it's good enough to serve as a `gets()` substitute for our examples.

The `scanf()` Function

Let's visit `scanf()` again. We've used `scanf()` with the `%s` format before to read a string. The chief difference between `scanf()` and `gets()` or `fgets()` lies in how they decide when they have reached the end of the string: `scanf()` is more of a "get word" than a "get string" function. The `gets()` function, as you've seen, takes in all the characters up to the first newline, as does `fgets()`, if the string is short enough. The `scanf()` function has two choices for terminating input. For either choice, the string starts at the first non-whitespace character encountered. If you use the `%s` format, the string runs up to (but not including) the next whitespace character (blank, tab, or newline). If you specify a field width, as in `%10s`, the `scanf()` collects up to 10 characters or up to the first whitespace character, whichever comes first (see Figure 11.3).



Input Statement	Original Input Queue*	Name Contents	Remaining Queue
`scanf('%s", name);`	`Fleebert□Hup`	`Fleebert`	`□Hup`
`scanf('%5s", name);`	`Fleebert□Hup`	`Fleeb`	`ert□Hup`
`scanf('%5s", name);`	`Ann□Ular`	`Ann`	`□Ular`

*the □ represents the space character

Figure 11.3 Field widths and `scanf()`.

Recall that the `scanf()` function returns an integer value that equals the number of items successfully read or returns `EOF` if it encounters the end of file.

Listing 11.11 illustrates how `scanf()` works when you specify a field width.

Listing 11.11 The `scan_str.c` Program

```
/* scan_str.c -- using scanf() */
#include <stdio.h>
```

```
int main(void)
{
    char name1[11], name2[11];
    int count;

    printf("Please enter 2 names.\n");
    count = scanf("%5s %10s",name1, name2);
    printf("I read the %d names %s and %s.\n",
           count, name1, name2);

    return 0;
}
```

Here are three runs:

```
Please enter 2 names.
Jesse Jukes
I read the 2 names Jesse and Jukes.
Please enter 2 names.
Liza Applebottham
I read the 2 names Liza and Applebotth.
Please enter 2 names.
Portensia Callowit
I read the 2 names Porte and nsia.
```

In the first example, both names fell within the allowed size limits. In the second example, only the first 10 characters of `Applebottham` were read because we used a `%10s` format. In the third example, the last four letters of `Portensia` went into `name2` because the second call to `scanf()` resumed reading input where the first ended; in this case, that was still inside the word `Portensia`.

Depending on the nature of the desired input, you may be better off using `fgets()` to read text from the keyboard. For example, `scanf()` wouldn't be that useful for entering the name of book or song, unless the name were a single word. The typical use for `scanf()` is reading and converting a mixture of data types in some standard form. For example, if each input line contains the name of a tool, the number in stock, and the cost of the item, you might use `scanf()`, or you might throw together a function of your own that does some entry error-checking. If you want to process input a word at a time, you can use `scanf()`.

The `scanf()` function has the same potential defect as `gets()`; it can create an overflow if the input word doesn't fit the destination. But you can use the field-width option in the `%s` specifier to prevent overflow.

String Output

Now let's move from string input to string output. Again, we will use library functions. C has three standard library functions for printing strings: `puts()`, `fputs()`, and `printf()`.

The `puts()` Function

The `puts()` function is very easy to use. Just give it the address of a string for an argument. Listing 11.12 illustrates some of the many ways to do this.

Listing 11.12 **The `put_out.c` Program**

```
/* put_out.c -- using puts() */
#include <stdio.h>
#define DEF "I am a #defined string."
int main(void)
{
    char str1[80] = "An array was initialized to me.";
    const char * str2 = "A pointer was initialized to me.";

    puts("I'm an argument to puts().");
    puts(DEF);
    puts(str1);
    puts(str2);
    puts(&str1[5]);
    puts(str2+4);

    return 0;
}
```

The output is this:

```
I'm an argument to puts().
I am a #defined string.
An array was initialized to me.
A pointer was initialized to me.
ray was initialized to me.
inter was initialized to me.
```

As with previous examples, each string appears on its own line because `puts()` automatically appends a newline when it displays a string.

This example reminds you that phrases in double quotation marks are string constants and are treated as addresses. Also, the names of character array strings are treated as addresses. The expression `&str1[5]` is the address of the sixth element of the array `str1`. That element contains the character `'r'`, and that is what `puts()` uses for its starting point. Similarly, str2+4 points to the memory cell containing the 'i' of "pointer", and the printing starts there.

How does puts() know when to stop? It stops when it encounters the null character, so there had better be one. Don't emulate the program in Listing 11.13!

Listing 11.13 **The nono.c Program**

```
/* nono.c -- no! */
#include <stdio.h>
int main(void)
{
    char side_a[] = "Side A";
    char dont[] = {'W', 'O', 'W', '!' };
    char side_b[] = "Side B";

    puts(dont);   /* dont is not a string */

    return 0;
}
```

Because dont lacks a closing null character, it is not a string, so puts() won't know where to stop. It will just keep printing from memory following dont until it finds a null somewhere. To ensure that a null character is not too distant, the program stores dont between two true strings. Here's a sample run:

```
WOW!Side A
```

The particular compiler used here stored the side_a array after the dont array in memory, so puts() kept going until hitting the null character in side_a. You may get different results, depending on how your compiler arranges data in memory. What if the program had omitted the arrays side_a and side_b? There are usually lots of nulls in memory, and if you're lucky, puts() might find one soon, but don't count on it.

The fputs() Function

The fputs() function is the file-oriented version of puts(). The main differences are these:

- The fputs() function takes a second argument indicating the file to which to write. You can use stdout (for *standard output*), which is defined in stdio.h, as an argument to output to your display.
- Unlike puts(), fputs() does not automatically append a newline to the output.

Note that gets() discards a newline on input, but puts() adds a newline on output. On the other hand, fgets() stores the newline on input, and fputs() doesn't add a newline on output. Suppose you want to write a loop that reads a line and echoes it on the next line. You can do this:

```
char line[81];
while (gets(line))  // same as while (gets(line) != NULL)
    puts(line);
```

Recall that `gets()` returns the null pointer if it encounters end-of-file. The null pointer evaluates as zero, or false, so that terminates the loop. Or you can do this:

```
char line[81];
while (fgets(line, 81, stdin))
    fputs(line, stdout);
```

With the first loop, the string in the `line` array is displayed on a line of its own because `puts()` adds a newline. With the second loop, the string in the `line` array is displayed on a line of its own because `fgets()` stores a newline. Note that if you mix `fgets()` input with `puts()` output, you'd get two newlines displayed for each string. The point is that `puts()` is designed to work with `gets()`, and `fputs()` is designed to work with `fgets()`.

Of course we mention `gets()` only so that you'll know how it works if you run across it in code and not to encourage you to use it.

The `printf()` Function

We discussed `printf()` pretty thoroughly in Chapter 4, "Character Strings and Formatted Input/Output." Like `puts()`, it takes a string address as an argument. The `printf()` function is less convenient to use than `puts()`, but it is more versatile because it formats various data types.

One difference is that `printf()` does not automatically print each string on a new line. Instead, you must indicate where you want new lines. Therefore,

```
printf("%s\n", string);
```

has the same effect as

```
puts(string);
```

As you can see, the first form takes more typing. It also takes longer for the computer to execute (not that you would notice). On the other hand, `printf()` makes it simple to combine strings for one line of printing. For example, the following statement combines `Well,` with the user's name and a `#defined` character string, all on one line:

```
printf("Well, %s, %s\n", name, MSG);
```

The Do-It-Yourself Option

You aren't limited to the standard C library options for input and output. If you don't have these options or don't like them, you can prepare your own versions, building on `getchar()`

and putchar(). Suppose you want a function like puts() that doesn't automatically add a newline. Listing 11.14 shows one way to create it.

Listing 11.14 **The put1() Function**

```
/* put1.c -- prints a string  without adding \n */
#include <stdio.h>
void put1(const char * string) /* string not altered */
{
    while (*string != '\0')
        putchar(*string++);
}
```

The char pointer string initially points to the first element of the called argument. Because this function doesn't change the string, use the const modifier. After the contents of that element are printed, the pointer increments and points to the next element. This goes on until the pointer points to an element containing the null character. Remember, the higher precedence of ++ compared to * means that putchar(*string++) prints the value pointed to by string but increments string itself, not the character to which it points.

You can regard put1.c as a model for writing string-processing functions. Because each string has a null character marking its end, you don't have to pass a size to the function. Instead, the function processes each character in turn until it encounters the null character.

A somewhat longer way of writing the function is to use array notation:

```
int i = 0;
while (string[i]!= '\0')
        putchar(string[i++]);
```

This involves an additional variable for the index.

Many C programmers would use the following test for the while loop:

```
while (*string)
```

When string points to the null character, *string has the value 0, which terminates the loop. This approach certainly takes less typing than the previous version. If you are not familiar with C practice, it is less obvious. However, this idiom is widespread, and C programmers are expected to be familiar with it.

> **Note**
>
> Why does Listing 11.14 use const char * string rather than const char string[] as the formal argument? Technically, the two are equivalent, so either form will work. One reason to use bracket notation is to remind the user that the function processes an array. With strings, however, the actual argument can be the name of an array, a quoted string, or a variable that has been declared as type char *. Using const char * string reminds you that the actual argument isn't necessarily an array.

Suppose you want a function like `puts()` that also tells you how many characters are printed. As Listing 11.15 demonstrates, it's easy to add that feature.

Listing 11.15 **The `put2()` Function**

```
/* put2.c -- prints a string and counts characters */
#include <stdio.h>
int put2(const char * string)
{
    int count = 0;
    while (*string)         /* common idiom       */
    {
        putchar(*string++);
        count++;
    }
    putchar('\n');          /* newline not counted */

    return(count);
}
```

The following call prints the string `pizza`:

```
put1("pizza");
```

The next call also returns a character count that is assigned to `num` (in this case, the value `5`):

```
num = put2("pizza");
```

Listing 11.16 presents a driver using `put1()` and `put2()` and showing nested function calls.

Listing 11.16 **The `put_put.c` Program**

```
//put_put.c -- user-defined output functions
#include <stdio.h>
void put1(const char *);
int put2(const char *);

int main(void)
{
    put1("If I'd as much money");
    put1(" as I could spend,\n");
    printf("I count %d characters.\n",
           put2("I never would cry old chairs to mend."));

    return 0;
}

void put1(const char * string)
```

```
{
    while (*string)  /* same as *string != '\0' */
        putchar(*string++);
}

int put2(const char * string)
{
    int count = 0;
    while (*string)
    {
        putchar(*string++);
        count++;
    }
    putchar('\n');

    return(count);
}
```

Hmmm, we are using `printf()` to print the value of `put2()`, but in the act of finding the value of `put2()`, the computer first must execute that function, causing the string to be printed. Here's the output:

```
If I'd as much money as I could spend,
I never would cry old chairs to mend.
I count 37 characters.
```

String Functions

The C library supplies several string-handling functions; ANSI C uses the `string.h` header file to provide the prototypes. We'll look at some of the most useful and common ones: `strlen()`, `strcat()`, `strncat()`, `strcmp()`, `strncmp()`, `strcpy()`, and `strncpy()`. We'll also examine `sprintf()`, supported by the `stdio.h` header file. For a complete list of the `string.h` family of functions, see Reference Section V, "The Standard ANSI C Library with C99 Additions," in Appendix B.

The `strlen()` Function

The `strlen()` function, as you already know, finds the length of a string. It's used in the next example, a function that shortens lengthy strings:

```
void fit(char *string, unsigned int size)
{
    if (strlen(string) > size)
        string[size] = '\0';
}
```

This function does change the string, so the function header doesn't use `const` in declaring the formal parameter `string`.

Try the `fit()` function in the test program of Listing 11.17. Note that the code uses C's string literal concatenation feature.

Listing 11.17 **The `test_fit.c` Program**

```
/* test_fit.c -- try the string-shrinking function */
#include <stdio.h>
#include <string.h> /* contains string function prototypes */
void fit(char *, unsigned int);

int main(void)
{
    char mesg[] = "Things should be as simple as possible,"
    " but not simpler.";

    puts(mesg);
    fit(mesg,38);
    puts(mesg);
    puts("Let's look at some more of the string.");
    puts(mesg + 39);

    return 0;
}

void fit(char *string, unsigned int size)
{
    if (strlen(string) > size)
        string[size] = '\0';
}
```

The output is this:

```
Things should be as simple as possible, but not simpler.
Things should be as simple as possible
Let's look at some more of the string.
 but not simpler.
```

The `fit()` function placed a `'\0'` character in the 39th element of the array, replacing a comma. The `puts()` function stops at the first null character and ignores the rest of the array. However, the rest of the array is still there, as shown by the following call:

```
puts(mesg + 8);
```

The expression `mesg + 39` is the address of `mesq[39]`, which is a space character. So `puts()` displays that character and keeps going until it runs into the original null character. Figure 11.4 illustrates (with a shorter string) what's happening in this program.

(Variations of the quotation in the `mesg` array are attributed to Albert Einstein, but it appears more likely to be a representation of his philosophy than a direct quote.)

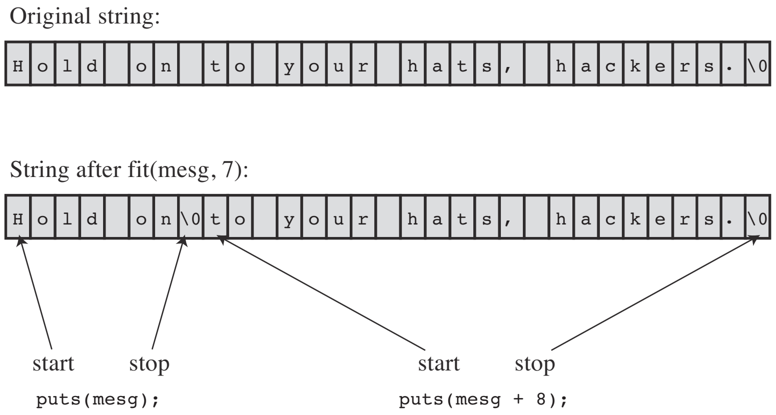

Original string:
H o l d   o n   t o   y o u r   h a t s ,   h a c k e r s . \0
String after fit(mesg, 7):
H o l d   o n \0 t o   y o u r   h a t s ,   h a c k e r s . \0
start
stop
puts(mesg);
start
stop
puts(mesg + 8);

Figure 11.4 The `puts()` function and the null character.

The `string.h` file contains function prototypes for the C family of string functions, which is why this example includes it.

> **Note**
>
> Some pre-ANSI systems use `strings.h` instead, and others might lack a string header file entirely.

The `strcat()` Function

The `strcat()` (for *string concatenation*) function takes two strings for arguments. A copy of the second string is tacked onto the end of the first, and this combined version becomes the new first string. The second string is not altered. The `strcat()` function is type `char *` (that is, a pointer-to-`char`). It returns the value of its first argument—the address of the first character of the string to which the second string is appended.

Listing 11.18 illustrates what `strcat()` can do. It also uses the `s_gets()` function we defined in Listing 11.10; recall that it uses `fgets()` to read a line, and then removes the newline character, if present.

Listing 11.18 **The `str_cat.c` Program**

```
/* str_cat.c -- joins two strings */
#include <stdio.h>
#include <string.h>  /* declares the strcat() function */
#define SIZE 80
char * s_gets(char * st, int n);
int main(void)
{
    char flower[SIZE];
    char addon[] = "s smell like old shoes.";

    puts("What is your favorite flower?");
    if (s_gets(flower, SIZE))
    {
        strcat(flower, addon);
        puts(flower);
        puts(addon);
    }
    else
        puts("End of file encountered!");
    puts("bye");


    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

This is a sample output:

```
What is your favorite flower?
wonderflower
wonderflowers smell like old shoes.
s smell like old shoes.
bye
```

The output illustrates that `flower` is altered while `addon` is not.

The `strncat()` Function

The `strcat()` function does not check to see whether the second string will fit in the first array. If you fail to allocate enough space for the first array, you will run into problems as excess characters overflow into adjacent memory locations. Of course, you can use `strlen()` to look before you leap, as shown in Listing 11.15. Note that it adds `1` to the combined lengths to allow space for the null character. Alternatively, you can use `strncat()`, which takes a second argument indicating the maximum number of characters to add. For example, `strncat(bugs, addon, 13)` will add the contents of the `addon` string to `bugs`, stopping when it reaches 13 additional characters or the null character, whichever comes first. Therefore, counting the null character (which is appended in either case), the `bugs` array should be large enough to hold the original string (not counting the null character), a maximum of 13 additional characters, and the terminal null character. Listing 11.19 uses this information to calculate a value for the `available` variable, which is used as the maximum number of additional characters allowed.

Listing 11.19 **The `join_chk.c` Program**

```
/* join_chk.c -- joins two strings, check size first */
#include <stdio.h>
#include <string.h>
#define SIZE 30
#define BUGSIZE 13
char * s_gets(char * st, int n);
int main(void)
{
    char flower[SIZE];
    char addon[] = "s smell like old shoes.";
    char bug[BUGSIZE];
    int available;

    puts("What is your favorite flower?");
    s_gets(flower, SIZE);
    if ((strlen(addon) + strlen(flower) + 1) <= SIZE)
        strcat(flower, addon);
    puts(flower);
    puts("What is your favorite bug?");
    s_gets(bug, BUGSIZE);
    available = BUGSIZE - strlen(bug) - 1;
    strncat(bug, addon, available);
    puts(bug);

    return 0;
}
char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Here is a sample run:

```
What is your favorite flower?
Rose
Roses smell like old shoes.
What is your favorite bug?
Aphid
Aphids smell
```

You may have noticed that `strcat()`, like `gets()`, can lead to buffer overflows. Why, then, doesn't the C11 standard dump `strcat()` and just offer `strncat()`? One reason may be that `gets()` exposes a program to dangers from those who use the program, while `strcat()` exposes the program to the dangers of a careless programmer. You can't control what some user will do in the future, but you can control what goes in your program. The C philosophy of trust the programmer brings with it the responsibility of recognizing when you can use `strcat()` safely.

The `strcmp()` Function

Suppose you want to compare someone's response to a stored string, as shown in Listing 11.20.

Listing 11.20 **The `nogo.c` Program**

```
/* nogo.c -- will this work? */
#include <stdio.h>
#define ANSWER "Grant"
#define SIZE 40
char * s_gets(char * st, int n);

int main(void)
{
    char try[SIZE];

    puts("Who is buried in Grant's tomb?");
    s_gets(try, SIZE);
    while (try != ANSWER)
    {
        puts("No, that's wrong. Try again.");
        s_gets(try, SIZE);
    }
    puts("That's right!");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

As nice as this program might look, it will not work correctly. ANSWER and try really are pointers, so the comparison try != ANSWER doesn't check to see whether the two strings are the same. Rather, it checks to see whether the two strings have the same address. Because ANSWER and try are stored in different locations, the two addresses are never the same, and the user is forever told that he or she is wrong. Such programs tend to discourage people.

What you need is a function that compares string *contents*, not string *addresses*. You could devise one, but the job has been done for you with strcmp() (for *string comparison*). This function does for strings what relational operators do for numbers. In particular, it returns 0 if its two string arguments are the same and nonzero otherwise. The revised program is shown in Listing 11.21.

Listing 11.21 **The compare.c Program**

```
/* compare.c -- this will work */
#include <stdio.h>
#include <string.h>   // declares strcmp()

#define ANSWER "Grant"
#define SIZE 40
char * s_gets(char * st, int n);

int main(void)
{
    char try[SIZE];

    puts("Who is buried in Grant's tomb?");
    s_gets(try, SIZE);
    while (strcmp(try,ANSWER) != 0)
    {
        puts("No, that's wrong. Try again.");
        s_gets(try, SIZE);
    }
    puts("That's right!");

    return 0;
}


char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

> **Note**
>
> Because any nonzero value is "true," most experienced C programmers would abbreviate the `while` statement to `while (strcmp(try,ANSWER))`.

One of the nice features of `strcmp()` is that it compares strings, not arrays. Although the array `try` occupies 40 memory cells and `"Grant"` only six (one for the null character), the comparison looks only at the part of `try` up to its first null character. Therefore, `strcmp()` can be used to compare strings stored in arrays of different sizes.

What if the user answers `"GRANT"` or `"grant"` or `"Ulysses S. Grant"`? The user is told that he or she is wrong. To make a friendlier program, you have to anticipate all possible correct answers. There are some tricks you can use. For example, you can use `#define` to define the answer as `"GRANT"` and write a function that converts all input to uppercase. That eliminates the problem of capitalization, but you still have the other forms to worry about, as well as the fact that his wife Julia is entombed there, too. We leave these concerns as exercises for you.

The `strcmp()` Return Value

What value does `strcmp()` return if the strings are not the same? Listing 11.22 shows an example.

Listing 11.22 **The `compback.c` Program**

```
/* compback.c -- strcmp returns */
#include <stdio.h>
#include <string.h>
int main(void)
{

    printf("strcmp(\"A\", \"A\") is ");
    printf("%d\n", strcmp("A", "A"));

    printf("strcmp(\"A\", \"B\") is ");
    printf("%d\n", strcmp("A", "B"));
    printf("strcmp(\"B\", \"A\") is ");
    printf("%d\n", strcmp("B", "A"));

    printf("strcmp(\"C\", \"A\") is ");
    printf("%d\n", strcmp("C", "A"));

    printf("strcmp(\"Z\", \"a\") is ");
    printf("%d\n", strcmp("Z", "a"));

    printf("strcmp(\"apples\", \"apple\") is ");
    printf("%d\n", strcmp("apples", "apple"));

    return 0;
}
```

Here is the output on one system:

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 1
strcmp("Z", "a") is -1
strcmp("apples", "apple") is 1
```

Comparing "A" to itself returns 0. Comparing "A" to "B" returns -1, and reversing the comparison returns 1. These results suggest that strcmp() returns a negative number if the first string precedes the second alphabetically and that it returns a positive number if the order is the other way. Therefore, comparing "C" to "A" gives a 1. Other systems might return 2—the difference in ASCII code values. The ANSI standard says that strcmp() returns a negative number if the first string comes before the second alphabetically, returns 0 if they are the same, and returns a positive number if the first string follows the second alphabetically. The exact numerical values, however, are left open to the implementation. Here, for example, is the output for another implementation, one that returns the difference between the character codes:

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 2
strcmp("Z", "a") is -7
strcmp("apples", "apple") is 115
```

What if the initial characters are identical? In general, strcmp() moves along until it finds the first pair of disagreeing characters. It then returns the corresponding code. For instance, in the very last example, "apples" and "apple" agree until the final s of the first string. This matches up with the sixth character in "apple", which is the null character, ASCII 0. Because the null character is the very first character in the ASCII sequence, s comes after it, and the function returns a positive value.

The last comparison points out that strcmp() compares all characters, not just letters, so instead of saying the comparison is alphabetic, we should say that strcmp() goes by the machine *collating sequence*. That means characters are compared according to their numeric representation, typically the ASCII values. In ASCII, the codes for uppercase letters precede those for lowercase letters. Therefore, strcmp("Z", "a") is negative.

Most often, you won't care about the exact value returned. You just want to know if it is zero or nonzero—that is, whether there is a match or not—or you might be trying to sort the strings alphabetically, in which case you want to know if the comparison is positive, negative, or zero.

> **Note**
>
> The strcmp() function is for comparing *strings*, not *characters*. So you can use arguments such as "apples" and "A", but you cannot use character arguments, such as 'A'. However, recall that the char type is an integer type, so you can use the relational operators for character comparisons. Suppose word is a string stored in an array of char and that ch is a char variable. Then the following statements are valid:
>
> ```
> if (strcmp(word, "quit") == 0)  // use strcmp() for strings
>     puts("Bye!");
> if (ch == 'q')                  // use == for chars
>     puts("Bye!");
> ```
>
> However, don't use ch or 'q' as arguments for strcmp().

Listing 11.23 uses the strcmp() function for checking to see whether a program should stop reading input.

Listing 11.23 **The quit_chk.c Program**

```
/* quit_chk.c -- beginning of some program */
#include <stdio.h>
#include <string.h>
#define SIZE 80
#define LIM 10
#define STOP "quit"
char * s_gets(char * st, int n);

int main(void)
{
    char input[LIM][SIZE];
    int ct = 0;

    printf("Enter up to %d lines (type quit to quit):\n", LIM);
    while (ct < LIM && s_gets(input[ct], SIZE) != NULL &&
```

```
            strcmp(input[ct],STOP) != 0)
    {
        ct++;
    }
    printf("%d strings entered\n", ct);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

This program quits reading input when it encounters an `EOF` character (`s_gets()` returns `NULL` in that case), when you enter the word *quit,* or when you reach the limit `LIM`.

Incidentally, sometimes it is more convenient to terminate input by entering an empty line—that is, by pressing the Enter key or Return key without entering anything else. To do so, you can modify the `while` loop control statement so that it looks like this:

```
while (ct < LIM && s_gets(input[ct], SIZE) != NULL
                && input[ct][0] != '\0')
```

Here, `input[ct]` is the string just entered and `input[ct][0]` is the first character of that string. If the user enters an empty line, `s_gets()` places the null character in the first element, so the expression

```
input[ct][0] != '\0'
```

tests for an empty input line.

The strncmp() Variation

The strcmp() function compares strings until it finds corresponding characters that differ, which could take the search to the end of one of the strings. The strncmp() function compares the strings until they differ or until it has compared a number of characters specified by a third argument. For example, if you wanted to search for strings that begin with "astro", you could limit the search to the first five characters. Listing 11.24 shows how.

Listing 11.24 **The starsrch.c Program**

```
/* starsrch.c -- use strncmp() */
#include <stdio.h>
#include <string.h>
#define LISTSIZE 6
int main()
{
    const char * list[LISTSIZE] =
    {
        "astronomy", "astounding",
        "astrophysics", "ostracize",
        "asterism", "astrophobia"
    };
    int count = 0;
    int i;

    for (i = 0; i < LISTSIZE; i++)
        if (strncmp(list[i],"astro", 5) == 0)
        {
            printf("Found: %s\n", list[i]);
            count++;
        }
    printf("The list contained %d words beginning"
           " with astro.\n", count);

    return 0;
}
```

Here is the output:

```
Found: astronomy
Found: astrophysics
Found: astrophobia
The list contained 3 words beginning with astro.
```

The strcpy() and strncpy() Functions

We've said that if pts1 and pts2 are both pointers to strings, the expression

```
pts2 = pts1;
```

copies only the address of a string, not the string itself. Suppose, though, that you do want to copy a string. Then you can use the strcpy() function. Listing 11.25 asks the user to enter words beginning with q. The program copies the input into a temporary array, and if the first letter is a q, the program uses strcpy() to copy the string from the temporary array to a permanent destination. The strcpy() function is the string equivalent of the assignment operator.

Listing 11.25 **The copy1.c Program**

```
/* copy1.c -- strcpy() demo */
#include <stdio.h>
#include <string.h>  // declares strcpy()
#define SIZE 40
#define LIM 5
char * s_gets(char * st, int n);

int main(void)
{
    char qwords[LIM][SIZE];
    char temp[SIZE];
    int i = 0;

    printf("Enter %d words beginning with q:\n", LIM);
    while (i < LIM && s_gets(temp, SIZE))
    {
        if (temp[0] != 'q')
            printf("%s doesn't begin with q!\n", temp);
        else
        {
            strcpy(qwords[i], temp);
            i++;
        }
    }
    puts("Here are the words accepted:");
    for (i = 0; i < LIM; i++)
        puts(qwords[i]);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Here is a sample run:

```
Enter 5 words beginning with q:
quackery
quasar
quilt
quotient
no more
no more doesn't begin with q!
quiz
Here are the words accepted:
quackery
quasar
quilt
quotient
quiz
```

Note that the counter `i` is incremented only when the word entered passes the *q* test. Also note that the program uses a character-based test:

```
if (temp[0] != 'q')
```

That is, is the first character in the `temp` array not a *q*? Another possibility is using a string-based test:

```
if (strncmp(temp, "q", 1) != 0)
```

That is, are the strings `temp` and `"q"` different from each other in the first element?

Note that the string pointed to by the second argument (`temp`) is copied into the array pointed to by the first argument (`qword[i]`). The copy is called the *target*, and the original string is called the *source*. You can remember the order of the arguments by noting that it is the same as the order in an assignment statement (the target string is on the left):

```
char target[20];
int x;
x = 50;                    /* assignment for numbers */
strcpy(target, "Hi ho!");  /* assignment for strings */
target = "So long";        /* syntax error           */
```

It is your responsibility to make sure the destination array has enough room to copy the source. The following is asking for trouble:

```
char * str;
strcpy(str, "The C of Tranquility"); // a problem
```

The function will copy the string `"The C of Tranquility"` to the address specified by `str`, but `str` is uninitialized, so the copy might wind up anywhere!

In short, `strcpy()` takes two string pointers as arguments. The second pointer, which points to the original string, can be a declared pointer, an array name, or a string constant. The first pointer, which points to the copy, should point to a data object, such as an array, roomy enough to hold the string. Remember, declaring an array allocates storage space for data; declaring a pointer only allocates storage space for one address.

Further `strcpy()` Properties

The `strcpy()` function has two more properties that you might find useful. First, it is type `char *`. It returns the value of its first argument—the address of a character. Second, the first argument need not point to the beginning of an array; this lets you copy just part of an array. Listing 11.26 illustrates both these points.

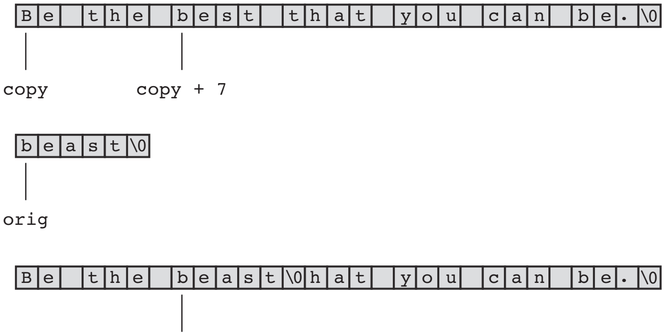
Listing 11.26 **The `copy2.c` Program**

```
/* copy2.c -- strcpy() demo */
#include <stdio.h>
#include <string.h>    // declares strcpy()
#define WORDS  "beast"
#define SIZE 40

int main(void)
{
    const char * orig = WORDS;
    char copy[SIZE] = "Be the best that you can be.";
    char * ps;

    puts(orig);
    puts(copy);
    ps = strcpy(copy + 7, orig);
    puts(copy);
    puts(ps);

    return 0;
}
```

Here is the output:

```
beast
Be the best that you can be.
Be the beast
beast
```

Note that `strcpy()` copies the null character from the source string. In this example, the null character overwrites the first *t* in `that` in `copy` so that the new string ends with `beast` (see Figure 11.5). Also note that `ps` points to the eighth element (index of 7) of `copy` because the first argument is `copy + 7`. Therefore, `puts(ps)` prints the string starting at that point.


B e   t h e   b e s t   t h a t   y o u   c a n   b e . \0
copy
copy + 7
b e a s t \0
orig
B e   t h e   b e a s t \0 h a t   y o u   c a n   b e . \0

the command
`strcpy ( copy + 7, orig );`
means "copy string from orig to here"

Figure 11.5 The `strcpy()` function uses pointers.

The Careful Choice: `strncpy()`

The `strcpy()` function shares a problem with `strcat()`—neither checks to see whether the source string actually fits in the target string. The safer way to copy strings is to use `strncpy()`. It takes a third argument, which is the maximum number of characters to copy. Listing 11.27 is a rewrite of Listing 11.25, using `strncpy()` instead of `strcpy()`. To illustrate what happens if the source string is too large, it uses a rather small size (seven elements, six characters) for the target strings.

Listing 11.27 **The** `copy3.c` **Program**

```
/* copy3.c -- strncpy() demo */
#include <stdio.h>
#include <string.h>  /* declares strncpy() */
#define SIZE 40
#define TARGSIZE 7
#define LIM 5
char * s_gets(char * st, int n);

int main(void)
{
    char qwords[LIM][TARGSIZE];
    char temp[SIZE];
    int i = 0;

    printf("Enter %d words beginning with q:\n", LIM);
    while (i < LIM && s_gets(temp, SIZE))
    {
        if (temp[0] != 'q')
            printf("%s doesn't begin with q!\n", temp);
        else
        {
            strncpy(qwords[i], temp, TARGSIZE - 1);
            qwords[i][TARGSIZE - 1] = '\0';
            i++;
        }
    }
    puts("Here are the words accepted:");
    for (i = 0; i < LIM; i++)
        puts(qwords[i]);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Here is a sample run:

```
Enter 5 words beginning with q:
quack
quadratic
quisling
quota
quagga
Here are the words accepted:
quack
quadra
quisli
quota
quagga
```

The function call `strncpy(target, source, n)` copies up to `n` characters or up through the null character (whichever comes first) from `source` to `target`. Therefore, if the number of characters in `source` is less than `n`, the entire string is copied, including the null character. The function never copies more than `n` characters, so if it reaches the limit before reaching the end of the source string, no null character is added. As a result, the final product may or may not have a null character. For this reason, the program sets `n` to one less than the size of the target array and then sets the final element in the array to the null character:

```
strncpy(qwords[i], temp, TARGSIZE - 1);
qwords[i][TARGSIZE - 1] = '\0';
```

This ensures that you've stored a string. If the source string actually fits, the null character copied with it marks the true end of the string. If the source string doesn't fit, this final null character marks the end of the string.

The `sprintf()` Function

The `sprintf()` function is declared in `stdio.h` instead of `string.h`. It works like `printf()`, but it writes to a string instead of writing to a display. Therefore, it provides a way to combine several elements into a single string. The first argument to `sprintf()` is the address of the target string. The remaining arguments are the same as for `printf()`—a conversion specification string followed by a list of items to be written.

Listing 11.28 uses `sprintf()` to combine three items (two strings and a number) into a single string. Note that it uses `sprintf()` the same way you would use `printf()`, except that the resulting string is stored in the array `formal` instead of being displayed onscreen.

Listing 11.28 **The `format.c` Program**

```
/* format.c -- format a string */
#include <stdio.h>
#define MAX 20
char * s_gets(char * st, int n);

int main(void)
{
    char first[MAX];
    char last[MAX];
    char formal[2 * MAX + 10];
    double prize;

    puts("Enter your first name:");
    s_gets(first, MAX);
    puts("Enter your last name:");
    s_gets(last, MAX);
    puts("Enter your prize money:");
    scanf("%lf", &prize);
    sprintf(formal, "%s, %-19s: $%6.2f\n", last, first, prize);
    puts(formal);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Here's a sample run:

```
Enter your first name:
Annie
Enter your last name:
von Wurstkasse
Enter your prize money:
25000
von Wurstkasse, Annie              : $25000.00
```

The `sprintf()` command took the input and formatted it into a standard form, which it then stored in the string `formal`.

Other String Functions

The ANSI C library has more than 20 string-handling functions, and the following list summarizes some of the more commonly used ones:

- `char *strcpy(char * restrict s1, const char * restrict s2);`

  This function copies the string (including the null character) pointed to by `s2` to the location pointed to by `s1`. The return value is `s1`.

- `char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`

  This function copies to the location pointed to by `s1` no more than `n` characters from the string pointed to by `s2`. The return value is `s1`. No characters after a null character are copied and, if the source string is shorter than `n` characters, the target string is padded with null characters. If the source string has `n` or more characters, no null character is copied. The return value is `s1`.

- `char *strcat(char * restrict s1, const char * restrict s2);`

  The string pointed to by `s2` is copied to the end of the string pointed to by `s1`. The first character of the `s2` string is copied over the null character of the `s1` string. The return value is `s1`.

- `char *strncat(char * restrict s1, const char * restrict s2, size_t n);`

  No more than the first `n` characters of the `s2` string are appended to the `s1` string, with the first character of the `s2` string being copied over the null character of the `s1` string. The null character and any characters following it in the `s2` string are not copied, and a null character is appended to the result. The return value is `s1`.

- `int strcmp(const char * s1, const char * s2);`

  This function returns a positive value if the `s1` string follows the `s2` string in the machine collating sequence, the value `0` if the two strings are identical, and a negative value if the first string precedes the second string in the machine collating sequence.

- `int strncmp(const char * s1, const char * s2, size_t n);`

  This function works like `strcmp()`, except that the comparison stops after `n` characters or when the first null character is encountered, whichever comes first.

- `char *strchr(const char * s, int c);`

  This function returns a pointer to the first location in the string `s` that holds the character `c`. (The terminating null character is part of the string, so it can be searched for.) The function returns the null pointer if the character is not found.

- `char *strpbrk(const char * s1, const char * s2);`

  This function returns a pointer to the first location in the string `s1` that holds any character found in the `s2` string. The function returns the null pointer if no character is found.

- `char *strrchr(const char * s, int c);`

  This function returns a pointer to the last occurrence of the character `c` in the string `s`. (The terminating null character is part of the string, so it can be searched for.) The function returns the null pointer if the character is not found.

- `char *strstr(const char * s1, const char * s2);`

  This function returns a pointer to the first occurrence of string `s2` in string `s1`. The function returns the null pointer if the string is not found.

- `size_t strlen(const char * s);`

  This function returns the number of characters, not including the terminating null character, found in the string `s`.

Note that these prototypes use the keyword `const` to indicate which strings are not altered by a function. For example, consider the following:

```
char *strcpy(char * restrict s1, const char * restrict s2);
```

It means `s2` points to a string that can't be changed, at least not by the `strcpy()` function, but `s1` points to a string that can be changed. This makes sense, because `s1` is the target string, which gets altered, and `s2` is the source string, which should be left unchanged.

The keyword `restrict`, discussed in Chapter 12, indicates restrictions on how the function arguments should be used, for example, not copying a string into itself.

The `size_t` type, as discussed in Chapter 5, "Operators, Expressions, and Statements," is whatever type the `sizeof` operator returns. C states that the `sizeof` operator returns an integer type, but it doesn't specify which integer type, so `size_t` can be `unsigned int` on one system and `unsigned long` on another. The `string.h` file defines `size_t` for a particular system or else refers to another header file having the definition.

As mentioned earlier, Reference Section V lists all the functions in the `string.h` family. Many implementations provide additional functions beyond those required by the ANSI standard. You should check the documentation for your implementation to see what is available.

Let's look at a simple use of one of these functions. Earlier we saw that `fgets()`, when it reads a line of input, stores the newline in the destination string. Our `s_gets()` function used a `while` loop to detect that newline character, but we can use `strchr()` instead. First, use `strchr()` to find the newline, if any. If the function finds the newline, it returns the address of the newline, and you then can place a null character at that address:

```
char line[80];
char * find;

fgets(line, 80, stdin);
find = strchr(line, '\n');   // look for newline
if (find)                    // if the address is not NULL,
    *find = '\0';            // place a null character there
```

If `strchr()` fails to find a newline, `fgets()` ran into the size limit before reaching the end of the line. You can add an `else`, as we did in `s_gets()`, to the `if` to process that circumstance.

Next, let's look at a full program that handles strings.

A String Example: Sorting Strings

Let's tackle the practical problem of sorting strings alphabetically. This task can show up in preparing name lists, in making up an index, and in many other situations. One of the main tools in such a program is `strcmp()` because it can be used to determine the order of two strings. The general plan will be to read an array of strings, sort them, and print them. Earlier, we presented a scheme for reading strings, and we will start the program that way. Printing the strings is no problem. We'll use a standard sorting algorithm that we'll explain later. We will also do one slightly tricky thing; see whether you can spot it. Listing 11.29 presents the program.

Listing 11.29 **The `sort_str.c` Program**

```
/* sort_str.c -- reads in strings and sorts them */
#include <stdio.h>
#include <string.h>
#define SIZE 81        /* string length limit, including \0  */
#define LIM 20         /* maximum number of lines to be read */
#define HALT ""        /* null string to stop input          */
void stsrt(char *strings[], int num);/* string-sort function */
char * s_gets(char * st, int n);

int main(void)
{
    char input[LIM][SIZE];     /* array to store input       */
    char *ptstr[LIM];          /* array of pointer variables */
    int ct = 0;                /* input count                */
    int k;                     /* output count               */

    printf("Input up to %d lines, and I will sort them.\n",LIM);
    printf("To stop, press the Enter key at a line's start.\n");
    while (ct < LIM && s_gets(input[ct], SIZE) != NULL
           && input[ct][0] != '\0')
    {
        ptstr[ct] = input[ct];  /* set ptrs to strings        */
        ct++;
    }
    stsrt(ptstr, ct);          /* string sorter              */
    puts("\nHere's the sorted list:\n");
    for (k = 0; k < ct; k++)
        puts(ptstr[k]) ;       /* sorted pointers            */

    return 0;
}

/* string-pointer-sorting function */
void stsrt(char *strings[], int num)
{
    char *temp;
    int top, seek;

    for (top = 0; top < num-1; top++)
        for (seek = top + 1; seek < num; seek++)
            if (strcmp(strings[top],strings[seek]) > 0)
            {
                temp = strings[top];
                strings[top] = strings[seek];
                strings[seek] = temp;
            }
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

We fed Listing 11.29 an obscure nursery rhyme to test it:

```
Input up to 20 lines, and I will sort them.
To stop, press the Enter key at a line's start.
O that I was where I would be,
Then would I be where I am not;
But where I am I must be,
And where I would be I can not.

Here's the sorted list:

And where I would be I can not.
But where I am I must be,
O that I was where I would be,
Then would I be where I am not;
```

Hmm, the nursery rhyme doesn't seem to suffer much from being alphabetized.

Sorting Pointers Instead of Strings

The tricky part of the program is that instead of rearranging the strings themselves, we just rearranged *pointers* to the strings. Let's see what that means. Originally, ptrst[0] is set to input[0], and so on. That means the pointer ptrst[i] points to the first character in the array input[i]. Each input[i] is an array of 81 elements, and each ptrst[i] is a single variable. The sorting procedure rearranges ptrst, leaving input untouched. If, for example, input[1] comes before input[0] alphabetically, the program switches ptrsts, causing ptrst[0] to point to the beginning of input[1] and causing ptrst[1] to point to the beginning of input[0]. This is much easier than using, say, strcpy() to interchange the contents of the two input strings. See Figure 11.6 for another view of this process. It also has the advantage of preserving the original order in the input array.

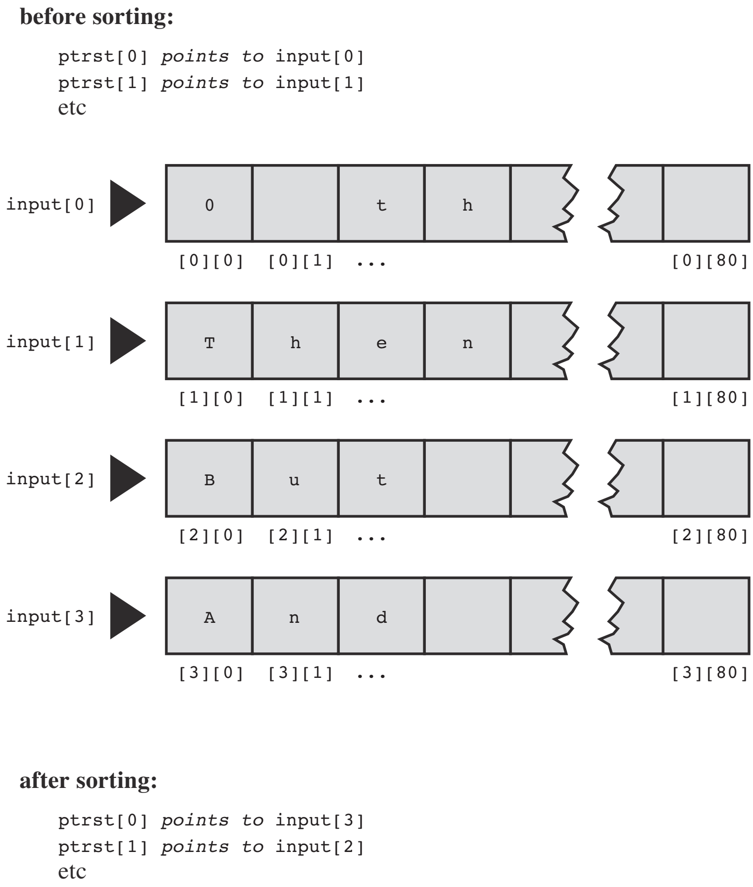


Figure 11.6 Sorting string pointers.

The Selection Sort Algorithm

To sort the pointers, we use the *selection sort* algorithm. The idea is to use a `for` loop to compare each element in turn with the first element. If the compared element precedes the current first element, the program swaps the two. By the time the program reaches the end of the loop, the first element contains a pointer to whichever string is first in the machine collating sequence. Then the outer `for` loop repeats the process, this time starting with the second element of `input`. When the inner loop completes, the pointer to the second-ranking string ends up in the second element of `ptrst`. The process continues until all the elements have been sorted.

Now let's take a more detailed look at the selection sort. Here is an outline in pseudocode:

```
for n = first to n = next-to-last element,
     find largest remaining number and place it in the nth element
```

The plan works like this: First, start with `n = 0`. Scan the entire array, find the largest number, and swap it with the first element. Next, set `n = 1` and then scan all but the first element of the array. Find the largest remaining number and swap it with the second element. Continue this process until reaching the next-to-last element. Now only two elements are left. Compare them and place the larger in the next-to-last position. This leaves the smallest element of all in the final position.

It looks like a `for` loop task, but we still have to describe the "find and place" process in more detail. One way to select the largest remaining value is to compare the first and second elements of the remaining array. If the second is larger, swap the two values. Now compare the first element with the third. If the third is larger, swap those two. Each swap moves a larger element to the top. Continue this way until you have compared the first with the last element. When you finish, the largest value is now in the first element of the remaining array. You have sorted the array for the first element, but the rest of the array is in a jumble. Here is the procedure in pseudocode:

```
for n - second element to last element,
  compare nth element with first element; if nth is greater, swap values
```

This process looks like another `for` loop. It will be nested in the first `for` loop. The outer loop indicates which array element is to be filled, and the inner loop finds the value to put there. Putting the two parts of the pseudocode together and translating them into C, we get the function in Listing 11.29. Incidentally, the C library includes a more advanced sorting function called `qsort()`. Among other things, it uses a pointer to a function to make the sorting comparison. Chapter 16, "The C Preprocessor and the C Library," gives examples of its use.

The `ctype.h` Character Functions and Strings

Chapter 7, "C Control Statements: Branching and Jumps," introduced the `ctype.h` family of character-related functions. These functions can't be applied to a string as a whole, but they can be applied to the individual characters in a string. Listing 11.30, for example, defines a function that applies the `toupper()` function to each character in a string, thus converting the whole string to uppercase. It also defines a function that uses `ispunct()` to count the number of punctuation characters in a string. Finally, the program uses `strchr()`, as described earlier, to handle the newline, if any, in the string read by `fgets()`.

Listing 11.30 **The `mod_str.c` Program**

```
/* mod_str.c -- modifies a string */
#include <stdio.h>
#include <string.h>
```

```
#include <ctype.h>
#define LIMIT 81
void ToUpper(char *);
int PunctCount(const char *);

int main(void)
{
    char line[LIMIT];
    char * find;

    puts("Please enter a line:");
    fgets(line, LIMIT, stdin);
    find = strchr(line, '\n');   // look for newline
    if (find)                    // if the address is not NULL,
        *find = '\0';            // place a null character there
    ToUpper(line);
    puts(line);
    printf("That line has %d punctuation characters.\n",
           PunctCount(line));

    return 0;
}

void ToUpper(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
}

int PunctCount(const char * str)
{
    int ct = 0;
    while (*str)
    {
        if (ispunct(*str))
            ct++;
        str++;
    }

    return ct;
}
```

The `while (*str)` loop processes each character in the string pointed to by `str` until the null character is reached. At that point, the value of `*str` becomes 0 (the code for the null character), or false, and the loop terminates. Here is a sample run:

```
Please enter a line:
Me? You talkin' to me? Get outta here!
ME? YOU TALKIN' TO ME? GET OUTTA HERE!
That line has 4 punctuation characters.
```

The `ToUpper()` function applies `toupper()` to each character in a string. (The fact that C distinguishes between uppercase and lowercase makes these two function names different from one another.) As defined by ANSI C, the `toupper()` function alters only characters that are lowercase. However, very old implementations of C don't do that check automatically, so old code normally does something like this:

```
if (islower(*str))         /* pre-ANSI C -- check before converting */
    *str = toupper(*str);
```

Incidentally, the `ctype.h` functions are usually implemented as *macros*. These are C preprocessor constructions that act much like functions but have some important differences. We'll cover macros in Chapter 16.

This program used a combination of `fgets()` and `strchr()` to read a line of input and replace the newline with a null character. The main difference between this approach and using `s_gets()` is that the latter disposes of the rest of the input line, if any, preparing the program for the next input statement. In this case, there is only one input statement, so that extra step isn't needed.

Next, let's try to fill an old emptiness in our lives, namely, the void between the parentheses in `main()`.

Command-Line Arguments

Before the modern graphical interface, there was the command-line interface. DOS and Unix are examples, and Linux terminal provides a Unix-like command-line environment. The *command line* is the line you type to run your program in a command-line environment. Suppose you have a program in a file named `fuss`. Then the command line to run it might look like this in Unix:

```
$ fuss
```

Or it might look like this in the Windows Command Prompt mode:

```
C> fuss
```

Command-line arguments are additional items on the same line. Here's an example:

```
$ fuss -r Ginger
```

A C program can read those additional items for its own use (see Figure 11.7).

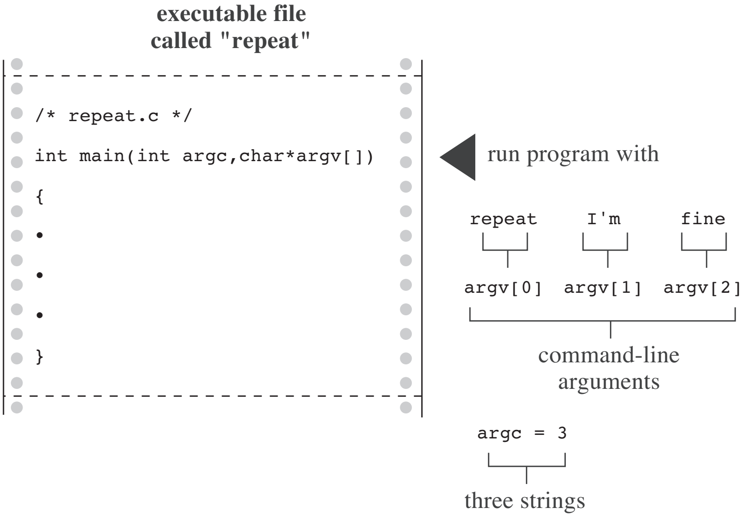


Figure 11.7 Command-line arguments.

A C program reads these items by using arguments to `main()`. Listing 11.31 shows a typical example.

Listing 11.31 **The `repeat.c` Program**

```
/* repeat.c -- main() with arguments */
#include <stdio.h>
int main(int argc, char *argv[])
{
    int count;

    printf("The command line has %d arguments:\n", argc - 1);
    for (count = 1; count < argc; count++)
        printf("%d: %s\n", count, argv[count]);
    printf("\n");

    return 0;
}
```

Compile this program into an executable file called `repeat`. Here is what happens when you run it from a command line:

```
C>repeat Resistance is futile
```

```
The command line has 3 arguments:
1: Resistance
2: is
3: futile
```

You can see why it is called `repeat`, but you might wonder how it works. We'll explain now.

C compilers allow `main()` to have no arguments or else to have two arguments. (Some implementations allow additional arguments, but that would be an extension of the standard.) With two arguments, the first argument is the number of strings in the command line. By tradition (but not by necessity), this `int` argument is called `argc` for *argument count*. The system uses spaces to tell when one string ends and the next begins. Therefore, the `repeat` example has four strings, including the command name, and the `fuss` example has three. The program stores the command line strings in memory and stores the address of each string in an array of pointers. The address of this array is stored in the second argument. By convention, this pointer to pointers is called `argv`, for *argument values*. When possible (some operating systems don't allow this), `argv[0]` is assigned the name of the program itself. Then `argv[1]` is assigned the first following string, and so on. For our example, we have the following relationships:

> `argv[0]` points to `repeat` (for most systems)
>
> `argv[1]` points to `Resistance`
>
> `argv[2]` points to `is`
>
> `argv[3]` points to `futile`

The program in Listing 11.31 uses a `for` loop to print each string in turn. Recall that the `%s` specifier for `printf()` expects the address of a string to be provided as an argument. Each element—`argv[0]`, `argv[1]`, and so on—is just such an address.

The form is the same as for any other function having formal arguments. Many programmers use a different declaration for `argv`:

```
int main(int argc, char **argv)
```

This alternative declaration for `argv` really is equivalent to `char *argv[]`. It says that `argv` is a pointer to a pointer to `char`. The example comes down to the same thing. It had an array with seven elements. The name of the array is a pointer to the first element, so `argv` points to `argv[0]`, and `argv[0]` is a pointer to `char`. Hence, even with the original definition, `argv` is a pointer to a pointer to `char`. You can use either form, but we think that the first more clearly suggests that `argv` represents a set of strings.

Incidentally, many environments, including Unix and DOS, allow the use of quotation marks to lump several words into a single argument. For example, the command

```
repeat "I am hungry" now
```

would assign the string `"I am hungry"` to `argv[1]` and the string `"now"` to `argv[2]`.

Command-Line Arguments in Integrated Environments

Integrated Windows environments, such as Apple's Xcode, Microsoft Visual C++, and Embarcadero C++ Builder, don't use command lines to run programs. However, some have a project dialog box that enables you to specify a command-line argument for a particular project. In other cases, you may be able to compile the program in the IDE and then open an MS-DOS window to run the program in command-line mode. But it's simpler if your system has the option of running a command-line compiler such as GCC.

Command-Line Arguments with the Macintosh

If you are using Xcode 4.6 (or similar version), you can provide command-line arguments by going to the Products menu and selecting Scheme, Edit Scheme, Run. Then select the Arguments tab and enter arguments in the Arguments Pass on Launch.

Or you can enter the Mac's Terminal mode and the world of command-line Unix. Then you can either locate the directory (Unix for folder) containing the executable code for your program, or, if you have downloaded the command-line tools, use `gcc` or `clang` to compile the program.

String-to-Number Conversions

Numbers can be stored either as strings or in numeric form. Storing a number as a string means storing the digit characters. For example, the number 213 can be stored in a character string array as the digits `'2'`, `'1'`, `'3'`, `'\0'`. Storing 213 in numeric form means storing it as, say, an `int`.

C requires numeric forms for numeric operations, such as addition and comparison, but displaying numbers on your screen requires a string form because a screen displays characters. The `printf()` and `sprintf()` functions, through their `%d` and other specifiers, convert numeric forms to string forms, and `scanf()` can convert input strings into numeric forms.. C also has functions whose sole purpose is to convert string forms to numeric forms.

Suppose, for example, that you want a program to use a numeric command-line argument. Unfortunately, command-line arguments are read as strings. Therefore, to use the numeric value, you must first convert the string to a number. If the number is an integer, you can use the `atoi()` function (for *alphanumeric to integer*). It takes a string as an argument and returns the corresponding integer value. Listing 11.32 shows a sample use.

Listing 11.32 **The `hello.c` Program**

```
/* hello.c -- converts command-line argument to number */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
```

```
{
    int i, times;

    if (argc < 2 || (times = atoi(argv[1])) < 1)
        printf("Usage: %s positive-number\n", argv[0]);
    else
        for (i = 0; i < times; i++)
            puts("Hello, good looking!");

    return 0;
}
```

Here's a sample run:

```
$ hello 3
Hello, good looking!
Hello, good looking!
Hello, good looking!
```

The `$` is a Unix and Linux prompt. (Some Unix systems use `%`.)The command-line argument of `3` was stored as the string `3\0`. The `atoi()` function converted this string to the integer value `3`, which was assigned to `times`. This then determined the number of `for` loop cycles executed.

If you run the program without a command-line argument, the `argc < 2` test aborts the program and prints a usage message. The same thing happens if `times` is 0 or negative. C's order-of-evaluation rule for logical operators guarantees that if `argc < 2`, `atoi(argv[1])` is not evaluated.

The `atoi()` function still works if the string only begins with an integer. In that case, it converts characters until it encounters something that is not part of an integer. For example, `atoi("42regular")` returns the integer `42`. What if the command line is something like `hello what`? On the implementations we've used, the `atoi()` function returns a value of `0` if its argument is not recognizable as a number. However, the C standard says the behavior in that case is undefined. The `strtol()` function, discussed shortly, provides error checking that is more reliable.

We include the `stdlib.h` header because, since ANSI C, it contains the function declaration for `atoi()`. That header file also includes declarations for `atof()` and `atol()`. The `atof()` function converts a string to a type `double` value, and the `atol()` function converts a string to a type `long` value. They work analogously to `atoi()`, so they are type `double` and `long`, respectively.

ANSI C has supplied more sophisticated versions of these functions: `strtol()` converts a string to a `long`, `strtoul()` converts a string to an `unsigned long`, and `strtod()` converts a string to `double`. The more sophisticated aspect is that the functions identify and report the first character in the string that is not part of a number. Also, `strtol()` and `strtoul()` allow you to specify a number base.

Let's look at an example involving strtol(). Its prototype is as follows:

```
long strtol(const char * restrict nptr, char ** restrict endptr, int base);
```

Here, nptr is a pointer to the string you want to convert, endptr is the address of a pointer that gets set to the address of the character terminating the input number, and base is the number base the number is written in. An example, given in Listing 11.33, makes this clearer.

Listing 11.33 **The strcnvt.c Program**

```
/* strcnvt.c -- try strtol()  */
#include <stdio.h>
#include <stdlib.h>
#define LIM 30
char * s_gets(char * st, int n);

int main()
{
    char number[LIM];
    char * end;
    long value;

    puts("Enter a number (empty line to quit):");
    while(s_gets(number, LIM) && number[0] != '\0')
    {
        value = strtol(number, &end, 10);  /* base 10 */
        printf("base 10 input, base 10 output: %ld, stopped at %s (%d)\n",
               value, end, *end);
        value = strtol(number, &end, 16);  /* base 16 */
        printf("base 16 input, base 10 output: %ld, stopped at %s (%d)\n",
               value, end, *end);
        puts("Next number:");
    }
    puts("Bye!\n");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
```

```
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else // must have words[i] == '\0'
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Here is some sample output:

```
Enter a number (empty line to quit):
10
base 10 input, base 10 output: 10, stopped at  (0)
base 16 input, base 10 output: 16, stopped at  (0)
Next number:
10atom
base 10 input, base 10 output: 10, stopped at atom (97)
base 16 input, base 10 output: 266, stopped at tom (116)
Next number:

Bye!
```

First, note that the string `"10"` is converted to the number 10 when `base` is 10 and to 16 when `base` is 16. Also note that if `end` points to a character, `*end` is a character. Therefore, the first conversion ended when the null character was reached, so `end` pointed to the null character. Printing `end` displays an empty string, and printing `*end` with the `%d` format displays the ASCII code for the null character.

For the second input string (base-10 interpretation), `end` is given the address of the `'a'` character. So printing `end` displays the string `"atom"`, and printing `*end` displays the ASCII code for the `'a'` character. When the base is changed to 16, however, the `'a'` character is recognized as a valid hexadecimal digit, and the function converts the hexadecimal number `10a` to `266`, base 10.

The `strtol()` function goes up to base 36, using the letters through `'z'` as digits. The `strtoul()` function does the same, but converts unsigned values. The `strtod()` function does only base 10, so it uses just two arguments.

Many implementations have `itoa()` and `ftoa()` functions for converting integers and floating-point values to strings. However, they are not part of the standard C library; use `sprintf()`, instead, for greater compatibility.

Key Concepts

Many programs deal with text data. A program may ask you to enter your name, a list of corporations, an address, the botanical name for a type of fern, the cast of a musical, or...well, because we interact with the world using words, there's really no end to examples using text. And strings are the means a C program uses to handle strings.

A C *string*—whether it be identified by a character array, a pointer, or a string literal—is stored as a series of bytes containing character codes, and the sequence is terminated by the null character. C recognizes the usefulness of strings by providing a library of functions for manipulating them, searching them, and analyzing them. In particular, keep in mind that you should use `strcmp()` instead of relational operators when comparing strings, and you should use `strcpy()` or `strncpy()` instead of the assignment operator to assign a string to a character array.

Summary

A C *string* is a series of `char`s terminated by the null character, `'\0'`. A string can be stored in a character array. A string can also be represented with a *string constant*, in which the characters, aside from the null character, are enclosed in double quotation marks. The compiler supplies the null character. Therefore, `"joy"` is stored as the four characters `j`, `o`, `y`, and `\0`. The length of a string, as measured by `strlen()`, doesn't count the null character.

String constants, also known as *string literals*, can be used to initialize character arrays. The array size should be at least one greater than the string length to accommodate the terminating null character. String constants can also be used to initialize pointers of type pointer-to-`char`.

Functions use pointers to the first character of a string to identify on which string to act. Typically, the corresponding actual argument is an array name, a pointer variable, or a quoted string. In each case, the address of the first character is passed. In general, it is not necessary to pass the length of the string, because the function can use the terminating null character to locate the end of a string.

The `fgets()` function fetches a line of input, and the `puts()` and `fputs()` functions display a line of output. They are part of the `stdio.h` family of functions, as once was the now disgraced and abandoned function `gets()`.

The C library includes several *string-handling* functions. Under ANSI C, these functions are declared in the `string.h` file. The library also has several *character-processing* functions; they are declared in the `ctype.h` file.

You can give a program access to *command-line arguments* by providing the proper two formal variables to the `main()` function. The first argument, traditionally called `argc`, is an `int` and is assigned the count of command-line words. The second argument, traditionally called `argv`, is a pointer to an array of pointers to `char`. Each pointer-to-`char` points to one of the command-line argument strings, with `argv[0]` pointing to the command name, `argv[1]` pointing to the first command-line argument, and so on.

The atoi(), atol(), and atof() functions convert string representations of numbers to type int, long, and double forms, respectively. The strtol(), strtoul(), and strtod() functions convert string representations of numbers to type long, unsigned long, and double forms, respectively.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. What's wrong with this attempted declaration of a character string?

```
int main(void)
{
   char name[] = {'F', 'e', 's', 's' };
  ...
}
```

2. What will this program print?

```
#include <stdio.h>
int main(void)
{
   char note[] = "See you at the snack bar.";
   char *ptr;

   ptr = note;
   puts(ptr);
   puts(++ptr);
   note[7] = '\0';
   puts(note);
   puts(++ptr);
   return 0;
}
```

3. What will this program print?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char food[] = "Yummy";
   char *ptr;

   ptr = food + strlen(food);
   while (--ptr >= food)
```

```
        puts(ptr);
   return 0;
}
```

4. What will the following program print?

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char goldwyn[40] = "art of it all ";
    char samuel[40] = "I read p";
    const char * quote = "the way through.";

    strcat(goldwyn, quote);
    strcat( samuel, goldwyn);
    puts(samuel);
    return 0;
}
```

5. The following provides practice with strings, loops, pointers, and pointer incrementing. First, suppose you have this function definition:

```
#include <stdio.h>
char *pr (char *str)
{
  char *pc;

  pc = str;
  while (*pc)
     putchar(*pc++);
  do {
     putchar(*--pc);
     } while (pc - str);
  return (pc);
}
```

Consider the following function call:

```
x = pr("Ho Ho Ho!");
```

a. What is printed?

b. What type should `x` be?

c. What value does `x` get?

d. What does the expression `*--pc` mean, and how is it different from `--*pc`?

e. What would be printed if `*--pc` were replaced with `*pc--`?

f. What do the two `while` expressions test for?

g. What happens if `pr()` is supplied with a null string as an argument?

h. What must be done in the calling function so that `pr()` can be used as shown?

6. Assume this declaration:

```
char sign = '$';
```

How many bytes of memory does `sign` use? What about `'$'`? What about `"$"`?

7. What does the following program print?

```
#include <stdio.h>
#include <string.h>
#define M1   "How are ya, sweetie? "
char M2[40] = "Beat the clock.";
char * M3  = "chat";
int main(void)
{
     char words[80];
     printf(M1);
     puts(M1);
     puts(M2);
     puts(M2 + 1);
     strcpy(words,M2);
     strcat(words, " Win a toy.");
     puts(words);
     words[4] = '\0';
     puts(words);
     while (*M3)
        puts(M3++);
     puts(--M3);
     puts(--M3);
     M3 = M1;
     puts(M3);
     return 0;
}
```

8. What does the following program print?

```
#include <stdio.h>
int main(void)
{
    char str1[] = "gawsie";     // plump and cheerful
    char str2[] = "bletonism";
```

```
        char *ps;
        int i = 0;

        for (ps = str1; *ps != '\0'; ps++) {
             if ( *ps == 'a' || *ps == 'e')
                    putchar(*ps);
             else
                    (*ps)--;
             putchar(*ps);
            }
        putchar('\n');
        while (str2[i] != '\0' ) {
           printf("%c", i % 3 ? str2[i] : '*');
           ++i;
           }
        return 0;
    }
```

9. The `s_gets()` function defined in this chapter can be written in pointer notation instead of array notation so as to eliminate the variable `i`. Do so.

10. The `strlen()` function takes a pointer to a string as an argument and returns the length of the string. Write your own version of this function.

11. The `s_gets()` function defined in this chapter can be written using `strchr()` instead of a `while` loop to find the newline. Do so.

12. Design a function that takes a string pointer as an argument and returns a pointer to the first space character in the string on or after the pointed-to position. Have it return a null pointer if it finds no spaces.

13. Rewrite Listing 11.21 using `ctype.h` functions so that the program recognizes a correct answer regardless of the user's choice of uppercase or lowercase.

Programming Exercises

1. Design and test a function that fetches the next `n` characters from input (including blanks, tabs, and newlines), storing the results in an array whose address is passed as an argument.

2. Modify and test the function in exercise 1 so that it stops after `n` characters or after the first blank, tab, or newline, whichever comes first. (Don't just use `scanf()`.)

3. Design and test a function that reads the first word from a line of input into an array and discards the rest of the line. It should skip over leading whitespace. Define a word as a sequence of characters with no blanks, tabs, or newlines in it. Use `getchar()`, not

4. Design and test a function like that described in Programming Exercise 3 except that it accepts a second parameter specifying the maximum number of characters that can be read.

5. Design and test a function that searches the string specified by the first function parameter for the first occurrence of a character specified by the second function parameter. Have the function return a pointer to the character if successful, and a null if the character is not found in the string. (This duplicates the way that the library `strchr()` function works.) Test the function in a complete program that uses a loop to provide input values for feeding to the function.

6. Write a function called `is_within()` that takes a character and a string pointer as its two function parameters. Have the function return a nonzero value (true) if the character is in the string and zero (false) otherwise. Test the function in a complete program that uses a loop to provide input values for feeding to the function.

7. The `strncpy(s1,s2,n)` function copies exactly `n` characters from `s2` to `s1`, truncating `s2` or padding it with extra null characters as necessary. The target string may not be null-terminated if the length of `s2` is `n` or more. The function returns `s1`. Write your own version of this function; call it `mystrncpy()`. Test the function in a complete program that uses a loop to provide input values for feeding to the function.

8. Write a function called `string_in()` that takes two string pointers as arguments. If the second string is contained in the first string, have the function return the address at which the contained string begins. For instance, `string_in("hats", "at")` would return the address of the `a` in `hats`. Otherwise, have the function return the null pointer. Test the function in a complete program that uses a loop to provide input values for feeding to the function.

9. Write a function that replaces the contents of a string with the string reversed. Test the function in a complete program that uses a loop to provide input values for feeding to the function.

10. Write a function that takes a string as an argument and removes the spaces from the string. Test it in a program that uses a loop to read lines until you enter an empty line. The program should apply the function to each input string and display the result.

11. Write a program that reads in up to 10 strings or to `EOF`, whichever comes first. Have it offer the user a menu with five choices: print the original list of strings, print the strings

in ASCII collating sequence, print the strings in order of increasing length, print the strings in order of the length of the first word in the string, and quit. Have the menu recycle until the user enters the quit request. The program, of course, should actually perform the promised tasks.

12. Write a program that reads input up to `EOF` and reports the number of words, the number of uppercase letters, the number of lowercase letters, the number of punctuation characters, and the number of digits. Use the `ctype.h` family of functions.

13. Write a program that echoes the command-line arguments in reverse word order. That is, if the command-line arguments are `see you later`, the program should print `later you see`.

14. Write a power-law program that works on a command-line basis. The first command-line argument should be the type `double` number to be raised to a certain power, and the second argument should be the integer power.

15. Use the character classification functions to prepare an implementation of `atoi()`; have this version return the value of 0 if the input string is not a pure number.

16. Write a program that reads input until end-of-file and echoes it to the display. Have the program recognize and implement the following command-line arguments:

`-p`	Print input as is
`-u`	Map input to all uppercase
`-l`	Map input to all lowercase

Also, if there are no command-line arguments, let the program behave as if the `–p` argument had been used.