Chapter 12

Strings

In this chapter, we begin to combine the previously studied data types into more complex units: strings and arrays of strings. A pointer type expression is used with `typedef` to define and name the type `string`, which is a fundamental type in C, but is not given a name by the standard. Strings combine the features of arrays, characters, and pointers discussed in the three preceding chapters. String input and output are presented, along with the most important functions from the standard `string` library.

Strings and arrays are combined to form a data structure called a *ragged array*, which is useful in many contexts. Examples given here include a two applications with menu-driven user interfaces.

12.0.1 Type Definitions

Variable names are like nouns in English; they are used to refer to objects. Type names are like adjectives; they are used to describe objects but are not objects themselves. A type is a pattern or rule for constructing future variables. Types such as `int`, `float`, and `double` have names defined by the C standard. These are all simple types, meaning that objects of these types have only one part. We have also studied arrays, which are aggregate objects.

Other important types that we use all the time, such as string, do not have standard names in C; they must be defined by type specifications. For example, although the C standard refers many times to strings, it does not define `string` as a type name; the actual type of a string in C is `char*`, or "pointer to character."

A `typedef` declaration enables us to attach a name to a type specification. Naming a type makes code easier to read and understand because type names permit the programmer to work on a conceptual plane instead of working at the level of implementation detail. If we want to use *string* as a type name in our program, we can write a `typedef` statement that defines `string` as a synonym for the character pointer expression:

```
typedef char* string;
```

A simple string literal:	`"George Washington\n"`
A two-part string literal:	`"was known for his unswerving honesty "`
	`"and became our first president."`
One with escape codes:	`"He said \"I cannot tell a lie\"."`

Figure 12.1. String literals.

We will use this type definition in many of the programs that follow[1]. When you include this definition in your program, you can use `string` as if it were an ordinary type name. We use it often in the programs in this chapter, especially when working with entire strings or arrays of strings. We use the basic type `char*` also, especially when working with pointers that scan or process a string one character at a time.

12.1 String Representation

We have used literal strings as formats and output messages in virtually every program presented so far but have never formally defined the type *string*. It is a necessary type and fundamental. C reference manuals talk about strings and the standard C libraries provide good support for working with strings. The functions in the `stdio` library make it possible to read and write strings in simple ways, and the functions in the `string` library let you manipulate strings easily. Within C, though, the implementation of a **string** is not simple; it is defined as a pointer to a null-terminated array of characters. The goal in this section is to learn certain fundamental ways to use and manipulate strings and to learn enough about their representation to avoid making common errors.

For instance, suppose we want a string that contains different messages at different times. There are two ways to do this. First, the pointer portion of the string can be switched from one literal array of letters to another. Second, it is possible to simply have an array of characters, each of which can be changed individually to form a new message. These two representations have very different properties and different applications, leading directly to programming choices, so we examine them more closely.

12.1.1 String Literals

A **string literal** is a series of characters enclosed in double quotes. String literals are used for output with `puts()`, as formats for `printf()` and `scanf()`, and as initializers. Ordinary letters and escape code characters such as `\n` may be written between the quotes, as shown in the first line of Figure 12.1. In addition to the characters between the quotes, the C translator adds a null character, `\0`, to the end of every string literal when it is compiled into a program. Therefore, the literal `"cat"` actually occupies 4 bytes of memory and the literal `"$"` occupies 2.

[1]You may wish to include it in your own personal file of C tools.

A comment in a string:	`"This /* is not a comment */ it is a string."`
A string in a comment:	`/* Here's how to put "a quote" in a comment. */`
Overlap error:	`"Here /* are some illegal" misnested delimiters. */`
Overlap error:	`/* We can't "misnest things */ this way, either."`

Figure 12.2. Quotes and comments.

Two-part literals. Often, prompts and formats are too long to fit on one line of code. In old versions of C, this constrained the way programs could be written. The ANSI C standard introduced the idea of *string merging* to fix this problem. With **string merging**, a long character string can be broken into two or more parts, each enclosed in quotes. These parts can be on the same line of code or on different lines, so long as there is nothing but whitespace or a comment between the closing quote of one part and the opening quote of the next. The compiler will join the parts into a single string with one null character at the end. An example of a two-line string literal is shown in the second part of Figure 12.1.

Quotation marks. You can even put a quote mark inside a string. To put a single quotation mark in a string, you can write it like any other character. To insert double quotes, you must write `\"`. Examples of quotes within a quoted string are shown in the last line of Figure 12.1.

Strings and comments. The relationship between comments and quoted strings can be puzzling. You can put a quoted phrase inside a comment and you can put something that looks like a comment inside a string. How can you tell which situation is which? The translator reads the lines of your source code from top to bottom and left to right. The symbol that occurs first determines whether the following code is interpreted as a comment or a string. If a begin-comment mark is found after an open quote, the comment text, enclosed between `/*` and `*/`, becomes part of the string. On the other hand, if the `/*` occurs first, followed by an open quote, the following string becomes part of the comment. In either case, if you attempt to put the closing marks out of order, you will generate a compiler error. Examples of these cases are shown in Figure 12.2.

The null string and the null character. The **null string** is a string with no characters in it except the null character. It is denoted in a program by two adjacent double quote marks, `""`. It is a legal string and often useful as a placeholder or initializer. The **null character** (eight 0 bits), or **null terminator**, plays a central role in all string processing. When you read a string with `scanf()`, the null character automatically is appended to it for you. When you print a string with `printf()` or `puts()`, the null character marks where to stop printing. The functions in the `string` library all use the null character to terminate their loops.

12.1.2 A String Is a Pointer to an Array

We have said that a string is a series of characters enclosed in double quotes and that a string is a pointer to an array of characters that ends with the null character. Actually, both these seemingly contradictory

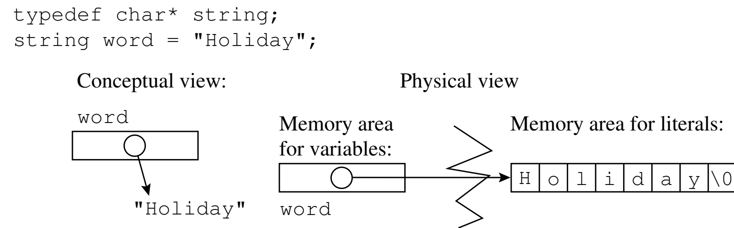


Figure 12.3. Implementation of a string variable.

statements are correct: When we write a string in a program using double quotes, it is translated by the compiler into a pointer to an array of characters.

String pointers. One way to create a string is to declare a variable of type `string` or `char*` and initialize it to point at a string literal.[2] Figure 12.3 illustrates a **string variable** named `word`, which contains the address of the first character in the sequence of letters that makes up the literal `"Holiday"`. The pointer variable and the letter sequence are stored in different parts of memory. The pointer is in the user's memory area and can be changed, as needed, to point to a different message. In contrast, the memory area for literals cannot be altered. C forbids us to change the letters of `"Holiday"` to make it into `"Ponyday"` or `"Holyman"`.

Using a pointer to get a string variable. We can use the pointer form of a string variable and **string assignment** to select one of a set of predefined messages and remember it for later processing and output. The string variable acts like a finger that can be switched back and forth among a set of possible literals.

For example, suppose you are writing a simple drill-and-practice program and the correct answer to a question, an integer, is stored in `target_answer` (an integer variable). Assume that the user's input has been read into `input_answer`. The output is supposed to be either the message "`Good job!`" when a correct answer is entered or "`Sorry!`" for a wrong answer. Let `response` be a string variable, as shown in Figure 12.4. We can select and remember the correct message by using an `if` statement and two pointer assignment statements, as shown.

12.1.3 Declare an Array to Get a String

To create a string whose individual letters can be changed, as when a word is entered from the keyboard, we must create a character array big enough to store the longest possible word. Figure 12.5 shows one way to do this with an array declaration.[3] Here, the variable `president` is a character array that can hold up to 17 letters and a null terminator, although only part of this space is used in this example.

[2]Dynamically allocated memory also can be used as an initializer. This advanced topic is left for Chapter 16.

[3]The use of `malloc()` for this task will be covered in Chapter 16.

```
if (input_answer == target_answer) response = "Good job!";
else response = "Sorry!";
```

After a correct answer, response points at "good" message:

target_answer: 48
input_answer: 48
response: → "Good job!\0" (not "Sorry!\0")

After an incorrect answer, response points at "bad" message:

target_answer: 48
input_answer: 56
response: → "Sorry!\0" (not "Good job!\0")

Figure 12.4. Selecting the appropriate answer.

An array of 18 `char`s containing the letters of a 15-character string and a null terminator. Two array slots remain empty.

```
char president [18] = "Abraham Lincoln";
```

Array name: president; Length: [18]; Initializer (optional): = "Abraham Lincoln"

president

A	b	r	a	h	a	m		L	i	n	c	o	l	n	\0		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Figure 12.5. An array of characters.

Initializers. A character array can be declared with or without an initializer. This initializer can be either a quoted string or a series of character literals, separated by commas, enclosed in braces. The array `president` in Figure 12.5 is initialized with a quoted string. The initializer also could be

```
= {'A','b','r','a','h','a','m',' ', 'L','i','n','c','o','l','n','\0'}
```

This kind of initializer is tedious and error prone to write, and you must remember to include the null character at the end of it. Because they are much more convenient to read and write, quoted strings (rather than lists of literal letters) usually are used to initialize `char` arrays. The programmer must remember, though, that every string ends in a null character, and there must be extra space in the array for the `\0`.

Using character arrays. If you define a variable to be an array of characters, you may process those characters using subscripts like any other array. Unlike the memory for a literal, this memory can be updated.

It also is true that an array name, when used in C with no subscripts, is translated as a pointer to the first slot of the array. Although an array is not the same as a string, this representation allows the name of a character array to be *used* as a string (a `char*`). Because of this peculiarity in the semantics of C, operations defined for strings also work with character arrays. Therefore, in Figure 12.5, we could write `puts( president )` and see `Abraham Lincoln` displayed.

However, an array that does not contain a null terminator must not be used in place of a string. The functions in the `string` library would accept it as a valid argument; the system would not identify a type or syntax error. But it would be a semantic error; the result is meaningless because C will know where the

Use type `char*` for these purposes:

- To point at one of a set of literal strings.
- As the type of a function parameter where the argument is either a `char*` or a `char` array.

Use a `char` array for these purposes:

- To hold keyboard input.
- Any time you wish to change some of the letters individually in the string.

Figure 12.6. Array vs. string.

These declarations create the objects diagrammed here:

```
char   ara[7] = "Monday"; /* 7 bytes total. */
string str    = "Sunday"; /* 11 bytes total, 4 for pointer, 7 for chars. */
```

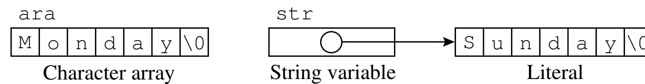


Figure 12.7. A character array and a string.

string starts but not where it ends. Everything in the memory locations following the string will be taken as part of it until a byte is encountered that happens to contain a 0.

12.1.4 Array vs. String

Typical uses of strings are summarized in Figure 12.6. Much (but not all) of the time, you can use strings without worrying about pointers, addresses, arrays, and null characters. But often confusion develops about the difference between a string and an array of characters. A common error is to declare a string variable as a `char*` and expect it to be able to store an array of characters. However, a string variable is only a pointer, which usually is just 4 bytes long. It cannot possibly hold an entire word like `"Hello"`. If we want a string to store alphabetic data, we need to allocate memory for an array of characters. These issues can all be addressed by understanding how strings are implemented in C.

Figure 12.7 shows declarations and diagrams for a character array and a string variable, both initialized by quoted strings. As you can see, the way storage is allocated for the two is quite different. When you use an array as a container for a string, space for the array is allocated with your other variables, and the letters of an initializing string (including the null) are copied into the array starting at slot 0. Any unused slots at the end of the array are left untouched and therefore contain garbage. In contrast, two separate storage

- The null character is 1 byte (type `char`) filled with 0 bits.
- The null pointer is a pointer filled with 0 bits (it points to address 0).
- The null string is a pointer to a null character. The address in the pointer is the nonzero address of some byte filled with 0 bits.

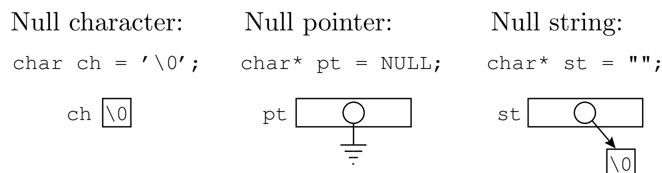


Figure 12.8. Three kinds of nothing.

areas are used when you use a string literal to initialize a pointer variable such as `str`. A string pointer typically occupies 4 bytes while the information itself occupies 1 byte for each character, plus another for the null character. The compiler stores the characters and the null terminator in an unchangeable storage area separate from the declared variables, and then it stores the address of the first character in the pointer variable.

Null objects. The null character, the null pointer, and the null string (pictured in Figure 12.8) often are confused with each other, even though they are distinct objects and used in different situations. The null string and the null character have different types (pointer vs. `char`), have different sizes (4 bytes vs. 1 byte), and certainly are not interchangeable in any way. Both the null pointer and the null string are pointers, but the first is all 0 bits, the address of machine location 0, and the second is the nonzero address of a byte containing all 0 bits.

12.2 String I/O

Even though the string data type is not built into the C language, functions in the standard libraries perform input, output, and other useful operations with strings.

12.2.1 String Output

We have seen two ways to output a string literal: `puts()` and `printf()`. The `puts()` function is called with one string argument, which can be either a literal string, the name of a string variable, or the name of a character array. It prints the characters of its argument up to, but not including, the null terminator, then *adds* a newline (`\n`) character at the end. If the string already ends in `\n`, this effectively prints a blank line following the text. We can call `printf()` in the same way, with one string argument, and the result will be

the same except that a newline will not be added at the end. The first box in Figure 12.9 demonstrates calls on both these functions with single arguments.

String output with a format. We can also use `printf()` with a `%s` conversion specifier in its format to write a string. The string still must have a null terminator. Again, the corresponding string argument may be a literal, string variable, or `char` array. Examples include

```
string fname = "Henry"; \\
printf( "First name:\ \%s{\bk}n", fname ); \\
printf( "Last name:  \%s{\bk}n", "James" );
```

The output from these statements is

```
First name: Henry
Last name:  James
```

Further formatting is possible. A `%`*n*`s` field specifier can be used to write the string in a fixed-width column. If the string is shorter than `n` characters, spaces will be added to fill the field. When `n` is positive, the spaces are added on the left; that is, the string is **right justified** in a field of `n` columns. If `n` is negative, the string is left justified. If the string is longer than `n` characters, the entire string is printed and extends beyond the specified field width. In the following example, assume that `fname` is `"Andrew"`:

```
printf( "{\gt \gt}\%10s{\lt \lt}{\bk}n", fname );\\
printf( "{\gt \gt}\%-10s{\lt \lt}{\bk}n", fname );\\
```

Then the output would be

```
>>    Andrew<<
>>Andrew    <<
```

Notes on Figure 12.9. String literals and string output. In this program, we demonstrate a collection of string-printing possibilities.

First box: ways to print strings.

- On the first line, we use `puts()` to print the contents of a string variable.
- The second line calls `printf()` to print a literal string. When printing just one string with `printf()`, that string becomes the format, which always is printed. This line also shows how escape characters are used to print quotation marks in the output.
- The third line prints both a string and a character. The `%s` in the format prints the string `"Print this string and ring the bell."` and then goes to a new line; the `%c` prints the character code `\a`, which results in a beep with no visible output on our system.
- The output from this box is

```
String demo program.
        This prints single ' and double " quotes.
        Print this string and ring the bell.
```

```
#include <stdio.h>

int main( void )
{
    string s1 = "String demo program.";
    string s2 = "Print this string and ring the bell.\n";

    puts( s1 );
    printf( "\t This prints single ' and double \" quotes.\n" );
    printf( "\t %s\n%c", s2, '\a' );

    puts( "These two quoted sections " "form a single string.\n" );
    printf( "You can break a format string into pieces if you\n"
            "end each line and begin the next with quotes.\n"
            "You may indent the lines any way you wish.\n\n" );

    puts( "This\tstring\thas\ttab\tcharacters\tbetween\twords.\t" );
    puts( "Each word on the line above starts at a tab stop. \n" );
    puts( "This puts()\n\t will make 3 lines of output,\n\t indented.\n" );

    printf( " >>%-35s<< \n >>%35s<< \n",
            "left justify, -field width", "right justify, +field width" );
    printf( " >>%10s<< \n\n", "not enough room?  Field gets expanded" );
}
```

Figure 12.9. String literals and string output.

Second box: adjacent strings are united.

- The `puts()` shows that two strings, separated only by whitespace, become one string in the eyes of the compiler. This is a very useful technique for making programs more readable. Whitespace refers to any nonprinting character that is legal in a C source code file. This includes ordinary space, newline, vertical and horizontal tabs, formfeed, and comments. If we were to write a comma between the two strings, only the first one would be used as the format, the second one would cause a compiler error.
- The output from this box is

```
These two quoted sections form a single string.

You can break a format string into pieces if you
end each line and begin the next with quotes.
You may indent the lines any way you wish.
```

- When we have a long format in a `printf()` statement, we just break it into two parts, being sure to start

and end each part with a double quote mark. Where we break the format string and how we indent it have no effect on the output.

Third box: escape characters in strings.

- The output from this box is shown below a tab line so you can see the reason for the unusual spacing:

```
This    string  has     tab     characters      between words.
Each word on the line above starts at a tab stop.

This puts()
        will make 3 lines of output,
        indented.
```

- Note that the tab character does not insert a constant number of spaces, it moves the cursor to the next tab position. Where the tab stops occur is defined by the software (an eight-character stop was used here). In practice, it is a little tricky to get things lined up using tabs.
- The presence or absence of newline characters controls the vertical spacing. The number of lines of output does not correspond, in general, to the number of lines of code.

Fourth box: using field-width specifiers with strings.

- The output is

```
>>left justify, -field width          <<
>>         right justify, +field width<<
>>not enough room?  Field gets expanded<<
```

- Spaces are used to pad the string if it is shorter than the specified field width. Using a negative number as a field width causes the output to be left justified. A positive field width produces right justification.
- If the string is longer than the field width, the width specification is ignored and the entire string is printed.

12.2.2 String Input

String input is more complicated than string output. The string input functions in the standard I/O library offer a bewildering variety of options; only a few of the most useful are illustrated here. One thing common to all string input functions is that they take one string argument, which should be the name of a character array or a pointer to a character array. *Do not use a variable of type* `char*` *for this purpose unless you have previously initialized it to point at a character array.*

For example, the diagram in Figure 12.10 shows two before-and-after scenarios that cause trouble. The code fragment on the left is wrong because the pointer is uninitialized, so it points at some random memory location; attempting to store data there is likely to cause the program to crash. The code on the right is wrong because `C` does not permit a program to change a literal string value. The diagrams in Figure 12.11 show two ways to do the job right. That on the left uses the name of an array in the `scanf()` statement; that on the right uses a pointer initialized to point at an array.

Both these `scanf()` statements are errors:

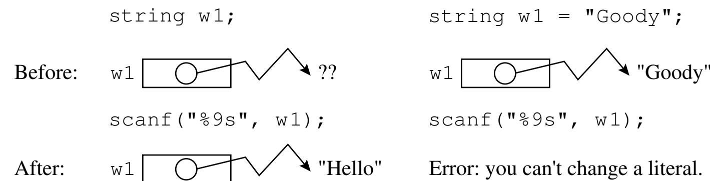


Figure 12.10. You cannot store an input string in a pointer.

Here are two correct and meaningful ways to read a string into memory:

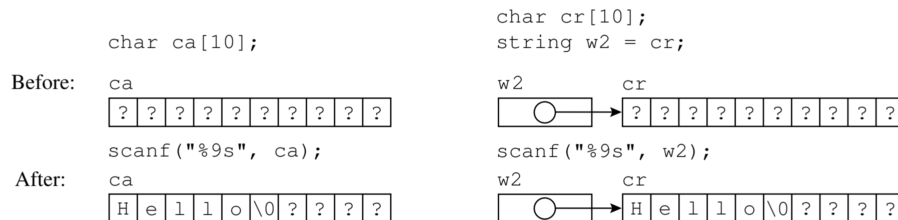


Figure 12.11. You need an array to store an input string.

Also, an array that receives string input must be long enough to contain all the information that will be read from the keyboard plus a null terminator. The input, once started by the user hitting the Enter key, continues until one of two conditions occurs:

1. The input function reads its own termination character. This varies, as described next, according to the input function.
2. The maximum number of characters is read, as specified by some field-width limit.

After the read operation is over, a null character will be appended to the end of the data.

Reading an entire line. The get-string function has the prototype `string gets ( char s[] )`. It reads characters (including leading whitespace) from the buffer into the array part of `s` until the first `\n` is read. The newline character terminates reading but is *not* stored in `s`. A null terminator then is stored in the array at the end of the input characters. The variable `s` must refer to an array of `char`s long enough to hold the entire string and the null character. Otherwise, the function will store the extra characters beyond the array bounds and wipe out the contents of adjacent memory variables. For example:

```
char course_name[20];
gets( course_name );    /* Error if input > 19 chars. */
```

Here, we ignore the return value of `gets()`. Normally this is just a pointer to the `char` array, but it is set to `NULL` when an input error occurs. For the time being, we continue to ignore such error-return codes; full error checking will be taken up in Chapter 14.

Reading a string with `scanf()`. For string input, we also may use `scanf()` with either a `%s` or `%ns` **conversion specifier**. In both cases, leading whitespace characters are skipped, then input characters are read and stored at the address given in the input list until more whitespace is encountered. The integer `n` in the `%ns` field specifier is the field-width limit. When `%ns` is used, the read operation will stop after reading `n` input characters, if whitespace was not found first. For example, `%9s` will stop the read operation after nine characters are read; the corresponding array variable must be at least 10 bytes long to allow enough space for the input plus the null terminator. Here are examples of calls on `scanf()` with and without a field-width limit:

```
char name[10];
scanf( "%s", name );    /* dangerous; overflow possible */
scanf( "%9s", name );    /* safe */
```

Note that there is no ampersand before the variable name; the `&` must be omitted because `name` is an array.

The brackets conversion. If the programmer wants some character other than whitespace to end the read operation with `scanf()`, a format of the form `%[^?]` or `%n[^?]` may be used instead of `%s` or `%ns` (*n* is still an integer denoting the field length). The desired termination character replaces the question mark and is written inside the brackets following the carat character, `^`.[4] Some examples:

```
scanf( "%29[^\n]", street );
scanf( "% 29[^,], %2[^\n]", city, state );
```

A complete input would be

```
122 E. 42nd St.
New York, NY
```

The first example stops the read operation after 29 characters have been read, sooner if a newline character is read (but it does not stop for other kinds of whitespace). The second example reads characters into the array `city` until a comma is read. It then discards the comma and reads the remaining characters into the array `state` until a newline is entered.

Combining input formats. A format for `scanf()` can have more than one string conversion specifier. Blanks can be included in the format to tell `scanf()` to ignore whitespace characters in the input. Anything else in an input format that is not part of a conversion specifier *must* be present in the input for correct operation. When that part of the format is interpreted, characters are read from the input buffer, compared

[4]A variety of effects can be implemented using square brackets. We present only the most useful here. The interested programmer should consult a standard reference manual for a complete description.

to the characters in the format, and discarded if they match. If they do not match, it is an input error and they are left in the buffer. This is another error condition that will be discussed in Chapter 14.

In the second example above, the program reads two strings (city and state) that are on one input line, separated by a comma and one or more spaces. The comma is detected first by the brackets specifier but not removed from the buffer. The comma and spaces then are read and removed from the buffer because of the `", "` in the format. Finally, the second string is read and stored. These comma and whitespace characters are discarded, not stored in a variable. Whenever a nonwhitespace character is used to stop an input operation, that character must be removed from the buffer by including it directly in the format or through some other method. Figure 12.12, in the next section, shows these calls on `scanf()` in the context of a complete program.

Using `gets()` vs. using `scanf()`. The basic differences between `gets()` and `scanf()` are the following:

- The `gets()` function reads an entire line up to the first newline character, whereas `scanf()` with `%`*n*`s` stops at the first whitespace character or after `n` characters have been read. We can make `scanf()` act more like `gets()` by using the `%[^\n]` conversion specifier. However, there is no way to make `gets()` limit its input to just `n` characters.
- The `scanf()` function skips leading whitespace characters, whereas `gets()` reads and stores any character, including whitespace, up to the first newline character. This is much like the difference between `getchar()` and `scanf()` with the `" %c"` specifier.

The `gets()` function typically is used to read entire lines of text of a known length. For more specialized processing, `scanf()` can be used to read single words, phrases delimited by a specified character, or fragments of text that are up to `n` characters long. If either input method is used and the array is too short to hold the entire data string, the input will overflow the storage area provided and overwrite adjacent memory variables. This is a serious security threat, and it is the reason that *unlimited string input operations should never be done.*

12.3 String Functions

The C standard supports a library of functions that process strings. In addition, programmers can define their own string functions. In this section, we first examine how strings are passed as arguments and used as parameters in functions. Then we explore the many built-in string functions and how they can be used in text processing.

12.3.1 Strings as Parameters

The program example in Figure 12.12 demonstrates the proper syntax for a function prototype, definition, and call with a string parameter.

This program demonstrates a programmer-defined function with a string parameter. It also uses several of the string input and output functions in a program context.

```
#include <stdio.h>
#include <string.h>
void print_upper( string s );

int main( void )
{
    char first[15];
    string name = first;
    char street[30], city[30], state[3];

    printf( " Enter first name: " );
    scanf( "%14s", name );                /* read one word only */
    printf( " Enter street address: " );
    scanf( " %29[^\n]", street );         /* read to end of line */
    printf(" Enter city, state: " );      /* split line at comma */
    scanf( " %29[^,], %2[^\n]", city, state );

    putchar( '\n' );
    print_upper( name );                 /* Print name in all capitals.    */
    printf( "\n %s %s, ", street, city ); /* Print street, city as entered. */
    print_upper( state );                /* Print state in all capitals.   */
    puts( "\n" );
}

/* ---------------------------- Print letters of string in upper case. */
void print_upper( string s )
{
    int k;       /* loop counter */

    for (k = 0; s[k] != '\0'; ++k) putchar( toupper( s[k] ) );
}
```

Figure 12.12. A string parameter.

Notes on Figure 12.12. A string parameter.

First box: string input.

1. The safe methods of string input described in the previous section are used here in a complete program. In every case, a call on `scanf()` limits the number of characters read to one less than the length of the array that receives them, leaving one space for the null terminator.
2. In the first call, the argument to `scanf()` is a string variable initialized to point to a character array. It is the correct type for input using either the `%s` or `%[^?]` conversion specifiers.
3. In the other two calls, the argument is a character array. This also is a correct type for input using these specifiers.

Second box: character and string output.

1. Both `putchar()` and `puts()` are used to print a newline character. The arguments are different because these functions have different types of parameters. The argument for `putchar()` is the single character, `\n`, while the string `"\n"` is sent to `puts()`, which prints the newline character and adds another newline.
2. Inner boxes: calls on `print_upper()`. The programmer-defined function `print_upper()` is called twice to print upper-case versions of two of the input strings. In the first call, the argument is a string variable. In the second, it is a character array. Both are appropriate arguments when the parameter has type `string`. We also could call this function with a literal string.
3. Sample output from this program might be

```
Enter first name: Maggie
Enter street address: 180 River Rd.
Enter city, state: Haddon, Ct

MAGGIE
180 River Rd. Haddon, CT
```

Last box: the `print_upper()` ***function.***

1. The parameter here is type `string`. The corresponding argument can be the name of a character array, a literal string, or a string variable. It could also be the address of a character somewhere in the middle of a null-terminated sequence of characters.
2. Since a string is a pointer to an array of characters, those characters can be accessed by writing subscripts after the name of the string parameter, as done here. The `for` loop starts at the first character of the string, converts each character to upper case, and prints it. The original string remains unchanged.
3. Like most loops that process the characters of a string, this loop ends at the null terminator.

Walk down a string, from the beginning to the null terminator, counting characters as you go. Do not count the terminal null character.

```
int my_strlen( char * st )    /* One way to calculate the string length. */
{
    int k;
    for (k=0; st[k]; ++k);    /* A tight loop -- no body needed. */
    return k;
}
```

Figure 12.13. Computing the length of a string.

12.3.2 The String Library

The standard C **string library** contains many functions that manipulate strings. Some of them are beyond the scope of this book, but others are so basic that they deserve mention here:

- `strlen()` finds the length of a string.
- `strcmp()` compares two strings.
- `strncmp()` compares up to n characters of two strings.
- `strcpy()` copies a whole string.
- `strncpy()` copies up to n characters of a string. If no null terminator is among these n characters, do not add one.
- `strcat()` copies a string onto the end of another string.
- `strncat()` copies up to n characters of a string onto the end of another string. If no null terminator is among these n characters, add one at the end.
- `strchr()` finds the leftmost occurrence of a given character in a string.
- `strrchr()` finds the rightmost occurrence of a given character in a string.
- `strstr()` finds the leftmost occurrence of a given substring in a string.

We discuss how these functions can be used correctly in a variety of situations and, for some, show how they might be implemented.

The length of a string. Because a string is defined as a **two-part object** (recall the diagrams in Figure 12.7), we need different methods to find the sizes of each part. Figure 12.14 illustrates the use of both operations. The operator `sizeof` returns the size of the pointer part of a string, while the function `strlen()` returns the number of characters in the array part, not including the null terminator. (Therefore, the string length of the null (empty) string is 0.) Figure 12.13 shows how `strlen()` might be implemented. Figure 12.24 shows further uses of `strlen()` in a program.

```
#include <stdio.h>   /* contains definition of type string */
#include <string.h>  /* contains definition of type string */
#define N 5
int main( void )
{
    string w1 = "Annette";
    string w2 = "Ann";
    char w3[20] = "Zeke";

    printf( " sizeof w1 is %2i   string is \"%s\"\t strlen is %i\n",
             sizeof w1, w1, strlen( w1 ) );
    printf( " sizeof w2 is %2i   string is \"%s\"\t\t strlen is %i\n",
             sizeof w2, w2, strlen( w2 ) );
    printf( " sizeof w3 is %2i   string is \"%s\"\t strlen is %i\n\n",
             sizeof w3, w3, strlen( w3 ) );
}
```

The output of this program is

```
sizeof w1 is  4   string is "Annette"   strlen is 7
sizeof w2 is  4   string is "Ann"       strlen is 3
sizeof w3 is 20   string is "Zeke"      strlen is 4
```

Figure 12.14. The size of a string.

Notes on Figure 12.14. The size of a string.

1. The variables `w1` and `w2` are strings. When the program applies `sizeof` to a string, the result is the size of a pointer on the local system. So `sizeof w1 == sizeof w2` even though `w1` points at a longer name than `w2`.
2. In contrast, `w3` is not a string; it is a `char` array used as a container for a string. When the programmer prints `sizeof w3`, we see that `w3` occupies 20 bytes, which agrees with its declaration. This is true even though only 5 of those bytes contain letters from the string. (Remember that the null terminator takes up 1 byte, even though you do not write it or see it when you print the string.)
3. To find the number of letters in a string, we use `strlen()`, string length, not `sizeof`. The argument to `strlen()` can be a `string` variable, a string literal, or the name of a character array. The value returned is the number of characters in the array part of the string, up to but not including the null terminator.

Comparing two strings. Figure 12.15 shows what happens when we try to compare two strings or `char` arrays using `==`. Unfortunately, only the addresses get compared. So if two pointers point at the same array,

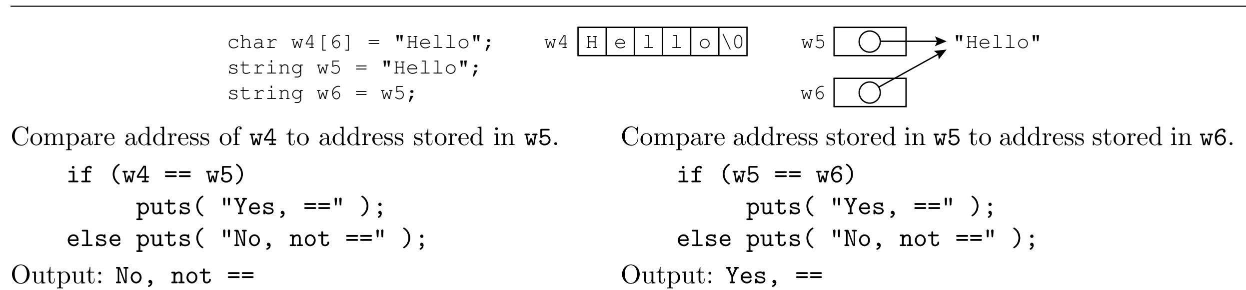


Figure 12.15. Do not use == with strings.

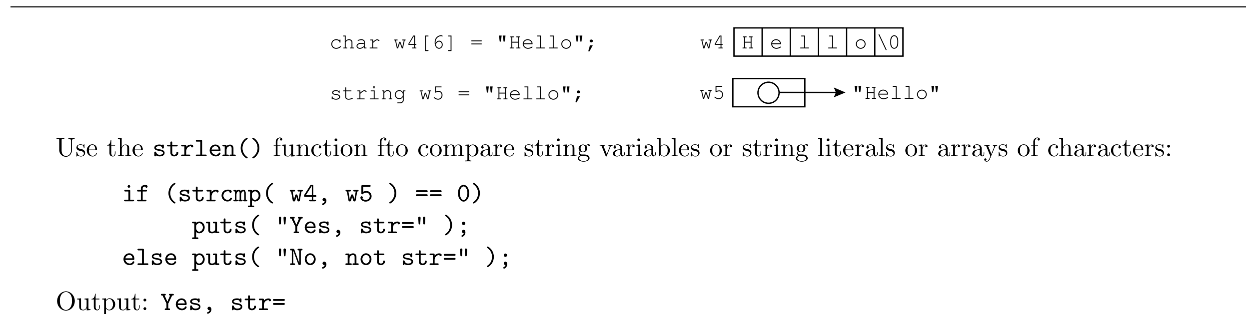


Figure 12.16. How to compare strings for alphabetic order.

as with `w5` and `w6`, they are equal; if they point at different arrays, as with `w4` and `w5`, they are not equal, even if the arrays contain the same characters.

To overcome this difficulty, the C `string` library contains the function `strcmp()`, which compares the actual characters, not the pointers. It takes two arguments that are strings (pointers) and does a letter-by-letter, alphabetic-order comparison. The return value is a negative number if the first string comes before the other alphabetically, a 0 if the strings are identical up through the null characters, and a positive number if the second string comes first. We can use `strcmp()` in the test portion of an `if` or `while` statement if we are careful. Because the intuitively opposite value of 0 is returned when the strings are the same, we must remember to compare the result of `strcmp()` to 0 in a test for equality. Figure 12.16 shows a proper test for equality and Figure 12.17 shows how this function could be implemented.

The other standard string comparison function, `strncmp()`, also deserves mention. A call takes three arguments and has the form `strncmp( s1, s2, n )`. The first two parameters are just like `strcmp()`; the third parameter is used to limit the number of letters that will be compared. For example, if we write `strncmp( "elephant", "elevator", 3 )`, the result will be 0, meaning that the strings are equal up to the $3rd$ character. If n is greater than the length of the strings, `strncmp()`, works identically to `strcmp()`.

Walk down two string at the same time. Leave loop if a pair of corresponding letters is unequal or if the end of one has been reached. Return a negative, zero, or positive result depending on whether the first string argument is less than, equal to, or greater than the second argument.

```
int my_strcmp( char * a, char * b )
{
    int k;
    for (k=0; a[k]; ++k){         /* Leave loop on at end of string */
        if (a[k] != b[k]) break;  /* Leave loop on inequality */
    }
    return a[k] - b[k];           /* =0 if at end of both strings. */
                                  /* Negative if a is lexically before b. */
                                  /* Positive if b comes before a. */
}
```

Figure 12.17. Possible implementation of `strcmp()`.

```
char line[] = "I found the cat in the fort.";
char * left, * right, * sub;
left  = strchr( line, 'n' );
right = strrchr( line, 'f' );
sub   = strstr( line, "the" );
```

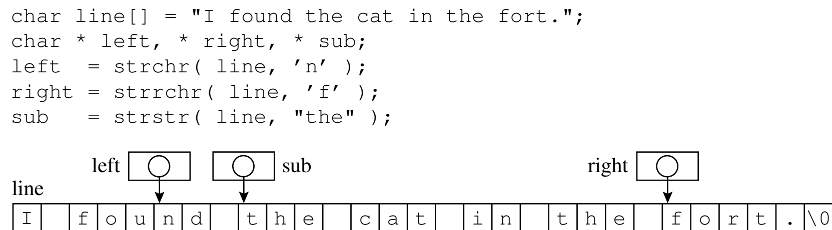


Figure 12.18. Searching for a character or a substring.

Character search. The function `strchr( string s1, char ch )` searches `s1` for the first (leftmost) occurrence of `ch` and returns a pointer to that occurrence. If the character does not appear at all, `NULL` is returned. The function `strrchr( string s1, char ch )` is similar, but it searches `s1` for the last (rightmost) occurrence of `ch` and returns a pointer to that occurrence. If the character does not appear at all, `NULL` is returned. Figure 12.18 shows how to call both these functions and interpret their results.

Substring search. The function `strstr( string s1, string s2 )` searches `s1` for the first (leftmost) occurrence of substring `s2` and returns a pointer to the beginning of that substring. If the substring does not appear at all, `NULL` is returned. Unlike searching for a character, there is no library function that searches for the last occurrence of a substring. Figure 12.18 gives an example of using `strstr()`.

Copying a string. What does it mean to copy a string? Do we copy the pointer or the contents of the array? Both. We really need two copy operations because sometimes we want one effect, sometimes the

```
char line{20];
strncpy( line, "Hotdog", 3 );     /* NO null terminator! */
strcpy( &line[3]. "diggety" );
strcat( line, " dog!" );
```

line after strncpy()

H	o	t																	

line after strcpy()

H	o	t		d	i	g	g	e	t	y	\0								

line after strcat()

H	o	t		d	i	g	g	e	t	y		d	o	g	!	\0			

Figure 12.19. Copy and concatenate.

other. Therefore, C has two kinds of operations: To copy the pointer, we use the assignment operator; to copy the contents of the array, we use `strcpy()` or `strncpy()`.

Figure 12.15 diagrams the result of using a pointer assignment; the expression `w6 = w5` copies the address from `w5` into `w6`. After the assignment, `w6` points at the same thing as `w5`. This technique is useful for output labeling. For example, in Figure 12.4 the value of `response` is set by assigning the address of the correct literal to it.

Copying the array portion of a string also is useful, especially when we need to extract data from an input buffer and move it into a more permanent storage area. The `strcpy()` function takes two string arguments, copies the contents of the second into the array referenced by the first, and puts a null terminator on the end of the copied string. It is necessary, first, to ensure that the receiving area is long enough to contain all the characters in the source string. This technique is illustrated in the second call of Figure 12.19. Here, the destination address is actually the middle of a character array.

The `strncpy()` function is similar to `strcpy()` but takes a third argument, an integer `n`, which indicates that copying should stop after `n` characters. Setting `n` equal to one less than the length of the receiving array guarantees that the copy operation will not overflow its bounds. Unfortunately, `strncpy()` does not automatically put a null terminator on the end of the copied string; that must be done as a separate operation unless the null character is copied by the operation. Or we can let a succeeding operation handle the termination, as happens in the example of Figure 12.19.

String concatenation. The function `strcat( string s1, string s2 )` appends `s2` onto the end of `s1`. Calling `strcat()` is equivalent to finding the end of a string, then doing a `strcpy()` operation starting at that address. If the position of the end of the string already is known, it actually is more efficient to use `strcpy()` rather than `strcat()` to append `s2`. In both cases, the array in which `s1` is stored must have enough space for the combined string. The `strcat()` function is used in the last call in the example of Figure 12.19 to complete the new string.

The function `strncat()` is like `strcat()`, except that the appending operation stops after at most `n`

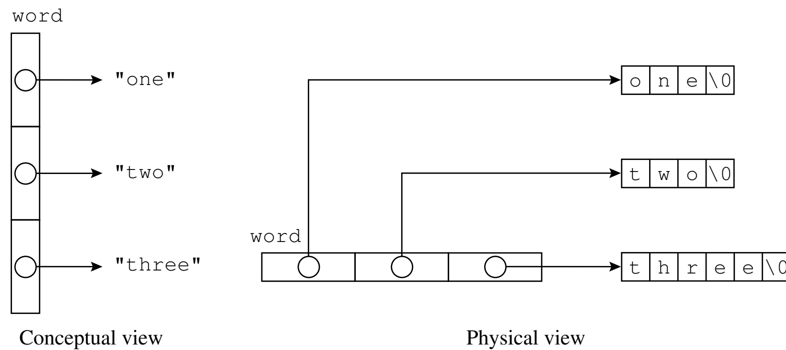


Figure 12.20. A ragged array.

characters. A null terminator *is* added to the end of the string. (Therefore, up to $n + 1$ characters may be written.) The careful programmer uses `strncpy()` or `strncat()`, rather than `strcpy()` or `strcat()`, and sets `n` to the number of storage slots remaining between the end of `s1` and the end of the array.

12.4 Arrays of Strings

An important aspect of C, and all modern languages, is that aggregate types such as strings and arrays can be compounded. Thus, we can build arrays of arrays and arrays of strings. An **array of strings**, sometimes called a **ragged array** because the ends of the strings generally do not line up in a neat column, is useful for storing a list of names or messages. This saves space over a list of uniform-sized arrays that are partially empty.

We can create a ragged array by declaring an array of strings and initializing it to a list of string literals.[5] For example, the following declaration creates the data structure shown in Figure 12.20:

```
string word[3] = \{ "one", "two", "three" \};
```

This particular form of a ragged array does not permit the contents of the entries to be changed. Making a ragged array of arrays that can be updated is explained in a later chapter. The program in Figures 12.21 and 12.25 demonstrate the use of ragged arrays in two different contexts.

12.4.1 The Menu Data Structure

In computer programs, a menu is a list of choices presented interactively to a computer user. Each choice corresponds to some action or process that the program can carry out and the user might wish to select. In

[5] In a later chapter, we show how to create a ragged array dynamically.

a modern windowing system, the user may make selections using a mouse. The other standard approach is to display an alphabetic or numeric code for each option and ask the user to enter the selected code from the keyboard. When execution of an option is complete, the menu is presented again for a new choice. Because they are easy to code and create an attractive interface, menus now are a component of programs in virtually every application area. Here, we present an easy way to set up your own menu-driven applications.

To use a menu, several things must occur:

- A title or some general instructions should be presented.
- Following that should be a list of possible choices and the input selection code for each. It is a good idea to include one option that means "take no action" or "quit."
- Then the user must be prompted to enter a choice, and the choice must be read and validated.
- Finally, the program must carry out the user's chosen intent.

Often, carrying out the chosen action means using supporting information specific to the choice. This information may be stored in a table implemented as a set of parallel arrays. In the menu display, each menu item can be shown next to its array subscript. The user is asked to choose a numbered item, and that number is used as a subscript for all of the arrays that form the table.

12.4.2 An Example: Selling Ice Cream

The program in Figure 12.21 illustrates the use of a menu of ice cream flavors and a parallel array of prices, as depicted at the top. A loop prints the menu. The loop variable is used as the subscript to print each menu item and is displayed as the selector for that item. When the user enters a choice, the number is used to subscript two **parallel arrays:** the `flavor` array and the `price` array. The program then prints a message using this information (and we imagine that ice cream and money change hands).

Notes on Figure 12.21. Selecting ice cream from a menu.

First box: creating a menu.

- The first string declared here is the greeting that will appear at the top of the menu.
- Second, we declare an array of strings named `flavor` and initialize it with the menu selections we want to offer.
- Another array, `price`, parallel to `flavor`, lists the price for a cone of each flavor. The position of each price in this array must be the same as the corresponding item in `flavor`.
- Making the menu longer or shorter is fairly easy: Just change the value of `CHOICES` at the top of the program, add or delete a string to or from the `flavor` array, and add or delete a corresponding cost to or from the `price` array.
- Since this is a relatively small program, the declarations have been placed in `main()`. Alternative placements are discussed as they occur in other examples in this chapter.

This program builds the parallel-array data structure diagrammed below, then calls the `menu()` function in Figure 12.22.

```
#include <stdio.h>
#include <string.h>
#define CHOICES 6
int menu ( string title, int max, string menu_list[] );

int main( void )
{
    int choice;                                      /* The menu selection. */
    string greeting = " I'm happy to serve you.  Our specials today are: ";
    string flavor[CHOICES] = { "Empty", "Vanilla", "Pistachio",
                               "Rocky Road", "Fudge Chip", "Lemon" };
    float price[CHOICES] = { 0.00, 1.00, 1.50, 1.35, 1.25, 1.20 };

    choice = menu( greeting, CHOICES, flavor );

    printf( "\n Here is your %s cone.\n That will be $%.2f.  ",
            flavor[choice], price[choice] );
    puts( "Thank you, come again." );
}
```

Figure 12.21. Selecting ice cream from a menu.

Second box: calling the* `menu()` *function.

- Since the `menu()` function, defined in Figure 12.22, will be used for all sorts of menus in various programs, the program must supply, as an argument, an appropriate application title to be displayed above the menu items.
- The second argument is the number of items in the menu. In C, the number of items in an array must be an argument to any function that processes the array; the function cannot determine this quantity from just the array itself.
- The third argument is the name of the array that contains the menu descriptions. These descriptions

This function is called from the main program in Figure 12.21.

```
int menu( string title, int max, string menu_list[] )
{
    int choice;
    int n = 0;               /* Loop counter for menu display. */
    printf( "\n %s\n ", title );
    for (n = 0; n < max; ++n)  printf( "\t %i. %s \n", n, menu_list[n] );

    printf( " Please enter your selection: ");
    for(;;) {                /* Prompt for and validate a menu selection. */
        scanf( "%i", &choice );
        if (choice >= 0 && choice < max) break;  /* Accept valid choice. */
        printf( " Please enter number between 0 and %i: ", max - 1 );
    }
    return choice;
}
```

Figure 12.22. A menu-handling function.

will be displayed for the user next to the selection codes.

- The return value from this function is the index of the menu item that the user selected. The `menu()` function returns only valid selections, so we can safely store the return value and use it as a subscript with no further checking.

Third box: the output.

- The variable `choice` is used to subscript both arrays. The program prints the chosen flavor and the price for that flavor using the validated menu selection as a subscript for the `flavor` and `price` arrays, respectively.
- Below the menu presented on the screen (shown later), the user will see his or her selection, as well as the results, displayed like this:

```
Please enter your selection: 3

Here is your Rocky Road cone.
That will be $1.35.  Thank you, come again.
```

Notes on Figure 12.22. A menu-handling function.

First box: the function header.

- The first parameter is a string that contains a title for the menu. This should be declared with the menu array, as in Figure 12.21.
- The second parameter is the number of choices in the menu. This number is used to control the loop that displays the menu and determine which index inputs are legal and which are out of bounds.
- The final parameter is an array of strings that contains the list of menu items.
- The return value will be a legal subscript for the arrays that contain the menu and price information.

Second box: displaying the menu.

- First, the program prints the title for the menu with appropriate vertical spacing.
- Then it uses a loop to display the menu. On each repetition, the loop displays the loop counter and one string. This creates a numbered list of items, where each number displayed is the array subscript of the corresponding item.
- The actual menu display follows. Note that choice `0` permits an escape from this menu without buying anything. It is a good idea to include such a "no operation" alternative. A "quit" option is not necessary in this menu since it is displayed only once by the program.

```
I'm happy to serve you.  Our specials today are:
        0. Empty
        1. Vanilla
        2. Pistachio
        3. Rocky Road
        4. Fudge Chip
        5. Lemon
```

Third box: the prompt, input, and menu selection validation.

- The original prompt is written outside the loop, since it will be displayed only once. If the first selection is invalid, the user will see an error prompt.
- This `for` loop is a typical data validation loop. It is very important to validate an input value before using it as a subscript. If that value is outside the range of legal subscripts, using it could cause the program to crash. The smallest legal subscript always is 0. The second argument to this function is the number of items in the menu array, which is one greater than the maximum legal subscript. The program compares the user's input to these bounds: If the selection is too big or too small, it displays an error prompt and asks for a new selection.
- A good user interface informs the user of the valid limits for a choice after he or she makes a mistake. Otherwise, the user might not understand what is wrong and have to figure it out by trial and error.
- When control leaves the loop, `choice` contains a number between `0` and `max-1`, which is a legal subscript for a menu with `max` slots. The function returns this choice. The calling program prints the chosen flavor and the price for that flavor using this number as a subscript for the `flavor` and `price` arrays.
- The following output demonstrates the validation process. The `menu()` function does not return to `main()` until the user makes a valid selection. After returning, `main()` uses the selection to print the chosen flavor and its price:

```
I'm happy to serve you.  Our specials today are:
        0. Empty
        1. Vanilla
        2. Pistachio
        3. Rocky Road
        4. Fudge Chip
        5. Lemon
Please enter your selection: -3
Please enter number between 0 and 5: 6
Please enter number between 0 and 5: 5

Here is your Lemon cone.
That will be $1.20.  Thank you, come again.
```

12.4.3 Subscript Validation

The `menu()` function in Figure 12.22 must validate the input because C does not guard against use of meaningless subscripts. To demonstrate the effects of using an invalid subscript, we removed the data validation loop from the third box, leaving only these two lines:

```
printf( " Please enter your selection:~" );\\
scanf( "\%i", \&choice );
```

Compiling and running the resulting program on a variety of invalid inputs produced the following output lines at the end of the program:

```
 Here is your POXO-@ .6D cone.

 Here is your MACHTYPE=m68k cone.

 Here is your (null pointer) cone.

 Here is your #9DO
$,R`nx[u cone.

 Here is your
 I'm happy to serve you.  Our specials today are:  cone.

 Segmentation fault

 Bus error
```

As you can see, C calculates a machine address based on any subscript given it and uses that address, even though it is not within the array bounds. The results normally are garbage and should be different for each illegal input. On some computer systems, the last two messages indicate errors that will cause the program to crash and may force the user to reboot the computer. A well-engineered program does not let an input error cause a system crash or garbage output. It validates the user's response to ensure that it is a legal choice, as in Figure 12.22.

This function displays a menu, then reads, validates, and returns a nonwhitespace character.

```
char menu_c( string title, int n, const string menu[], string valid )
{
    int k;
    char ch;

    printf("\n%s\n\n", title);
    for(;;) {
        for( k=0; k<n; ++k ) printf("\t%s \n", menu[k]);
        printf("\n Enter code of desired item: ");
        scanf(" %c", &ch);
        if (strchr( valid, ch )) break;

        while (getchar() != '\n');      /* Discard rest of input line */
        puts("Illegal choice or input error; try again.");
    }
    return ch;
}
```

Figure 12.23. The `menu_c()` function.

12.4.4 The `menu_c()` Function

The process of displaying the menu and eliciting a selection is much the same for any menu-based program: the content of the menu changes from application to application but the processing method remains the same. There is one major variation: some menus are character-based and the menu choice is type `char`, (not type `int`) and are processed by a `switch` (not by using subscript). The commonality from application to application enables us to write two general-purpose menu-handling function, one for integer choices `menu()`, above, and one for characters, `menu_c()`, below.

In Figure 12.23, we show a character-based menu function, `menu_c()`. Like `menu()`, it receives a menu title, the number of menu items, and the menu as arguments, where each menu string now consists of a selection code character and a phrase describing the menu item. The fourth argument is a string consisting of all the letters that are legal menu choices, and is used to validate the selection.

The function displays the strings from the menu array, one per line, then reads and returns the menu choice in the form of a single nonwhitespace character.[6] This function will be used in the program of Figure 12.29 in the next section.

[6]In contrast, `menu()` reads and returns an integer response that can be used directly as a table subscript.

Notes on Figure 12.23. The `menu_c()` function. We expect the third parameter (the menu) to have a phrase describing each choice and, with that phrase, a letter to type. This function displays a title and a menu and prompts for a response in the same manner as `menu()`, in Figure 12.22. After validation, the response is returned. Only the method of input validation is different.

First box: Reading and validating the choice.

- The `printf()` prompts the user to make a selection. It lets the user know that a character (not a number) is expected this time.
- We use `scanf()` with `" %c"` to read the user's response. The space before the `%c` is important because the input stream probably contains a newline character from some prior input step.[7] The program must skip over that character to reach the user's current input.
- To validate the user's choice, we compare it to the list of valid choices that were supplied as the fourth parameter. The `strchr()` function makes this very easy: if the user's choice is in the list, `strchr()` will return a pointer to that letter, If it is `not` in the list, `strchr()` will return a `NULL` pointer. Thus, a non-null return value indicates a valid choice. We use the `if` statement directly to test this value. If the choice is valid, control leaves the loop. Otherwise, we move on to the error handling code.

Second box: Error handling.

- This is a one-line loop with no loop body; the semicolon is not a mistake. It reads a character and tests it, reads and tests, until the newline is found. The purpose is to skip over any extra characters that the user might have typed and that might be still in the input stream. Cleaning out the input stream is often done after an error to help the user become synchronized with the action of the program and what should happen next.
- We announce clearly that an error was made. Control will return to the top of the loop and display the menu again. The only way to leave the loop is to type a valid choice.

12.5 String Processing Applications

In this section, we examine two applications that use many of the string processing functions presented in the chapter.

12.5.1 Password Validation

The following program uses some of the string functions in a practical context; namely, requiring the user to enter a password to access a sensitive database on the computer. In this case, the database is information about students in one class at an elementary school.

[7] Review the discussion of these issues concerning problems with `getchar()` following Figure 8.6.

Create a personalized form letter to send to each members of the class.

```
#include <stdio.h>
#include <string.h>
typedef char* string;

#define PASSW "StaySafe"
#define BUFLEN 80
void writeNotices( void );
void compose( int who, int what, string ills );
int menu_i( string title, int n, const string menu[] );

/* ---------------------------------------------------------------- */
int main( void )
{
    char word[ BUFLEN ];
    puts( "\n Send Illness Notices" );
    printf( "\n Please enter the password: " );
    scanf( "%79[^\n]", word );
    if (strcmp( word, PASSW )) {    /* Non-zero response means non-match. */
        printf( " %s is not the password; No Entry! \n", word );
    else
        writeNotices();            /* Send notices to the entire class. */
    return 0;
}
```

Figure 12.24. Password validation: Comparing and copying strings.

The illness notice is a form letter consisting of a series of sentence fragments with blanks to be filled in to make it complete. The data to use are defined as a pair of parallel arrays giving information about each illnesses, an array of symptoms, and a array of information about each student in the class.

Figure 12.24 implements the password validation phase of this program. Figure 12.25 contains arrays of data about illnesses and symptoms. Finally, Figure 12.27 composes the letter using this text, the user's selections, and functions from the standard `string` library.

Notes on Figure 12.24. Comparing and copying strings.

First box: Using strings.

- We include the header for the string library because we will use `strlen(), strcmp(), strcpy()`, and `strchr()`.
- We write this `typedef` declaration so that we can use `string` as a type name.

Second box: Using a password.

- In a real application, the owner of the application would be able to change the password. However, we are trying to keep this program simple and so we are using `#define` to create the password. The program would need to be recompiled to change it.
- `BUFLEN` is used to define the variable `word`, which will store the user's password input. This limits passwords to 80 characters, an arbitrary length that is much longer than we expect to need. 80 characters is a common limit for the length of an input line. This usage can be traced back to the time when punched cards were used for computer input. A punched card had 80 columns.

Third box: Checking the password.

- We prompt the user to enter a password. In a real application, this would be done without showing the password on the screen. In this program, however, we let the password stay on the screen for simplicity, and because we would need to go outside standard C to mask the password input.
- This call on `scanf()` will read all the characters typed, up to but not including the end-of-line character, and store them in `word`. A maximum of 79 characters will be read, to avoid overflowing the buffer. If more characters are typed, the extra ones will remain, unread, in the input stream.

Fourth box: the comparison.

- `strcmp()` takes two string arguments of any variety. Here, the first argument is a character array, the second is a literal string.
- If the two strings are unequal, `strcmp()` will return a non-zero value and and an error comment will be printed.
- If the two strings are equal, the `else` clause will be selected. The user now has gained entry into the protected part of the program, which calls the `writeNotices()` function to compose and print a form letter to be sent to all students in the class.
- Here is a sample of beginning output that resulted from entering an incorrect password:

```
Send Illness Notices

Please enter the password: Jam
Jam is not the password; No Entry!
```

12.5.2 Table-driven Programming

When writing a personalized form letter, we need data about each recipient and about variable elements in the message. In this program, that data is supplied in the form of tables of constant strings. In a later chapter, we will see how to put the data into data files instead of compiling it as part of the program. In both cases, we are using a technique called *table-driven programming*. The behavior of such a program is controlled by the amount of the data in the table, but does not depend at all on the actual data.

This header file is named `ills.h`. It is included by the program in Figure 12.28. It defines a list of communicable diseases that must be reported to parents. The incubation period for each disease is in a parallel array, and the major symptoms of these illnesses form a second data structure.

```
#define ILLS 7           /* Number of illnesses in the list */
const string illness[] = { "chicken pox","flu","head lice",
                           "measles","mumps","scabies","strep throat"
                         };
const string incubate[] = { "2 to 3 weeks","1 to 4 days","1 to 2 weeks",
                            "10 days","2 to 3 weeks","four weeks","one week"
                          };
```

```
#define SYMPTOMS 9       /* Number of symptoms in the list */
const string symptom[] = { "done", "cough, ", "diarrhea, ",
                           "fever, ", "headache, ", "itching, ",
                           "rash, ","sore throat, ","swollen face, "
                         };
```

Figure 12.25. The list of health problems.

Notes on Figures 12.25. The list of health problems.

First box: Illnesses.
This is a parallel-array data structure. We list 7 childhood illnesses and, for each, the incubation period. These strings will be printed as part of the form letter sent to parents.

Second box: Symptoms.
The second box contains a list of possible symptoms. The number and combination of symptoms for each illness is different. When the message is constructed, a series of symptoms is copied into one long string and printed.

Notes on Figure 12.26. The class list.

Second box: The class. We define an array of strings, with one string per student. Each string holds three pieces of information: the child's name, gender, and parent's name. The comma after the first name enables us to parse the string and identify all the fields.

Notes on Figure 12.28. Composing the letter.

First box: the header file.
We must include the header files that contain the class list and the information on illnesses. The `#include` statements are written above this function because the main program does not use these data arrays. Separate files were used for these declarations because of the large number and the changeable

This header file is named `myclass.h`. It is included by the program in Figure 12.28. It provides information about one teacher and all of the students in his or her class. Listed for each student is the name, gender, and name of that student's legal guardian.

```
#define TEACHER "Mr. David Schein"
#define CLASS   "Third Grade"
```

```
#define CLASS_SIZE 7          /* Number of children in my class */
const string student[] = {
    "Ann Wang,          F Ms. Cho",
    "Charles Baker,     M Mr. & Mrs. Baker",
    "Desiree Smith,     F Mr. Smith",
    "Harold Pinter,     M Mrs. Kilmer",
    "James Hendrix,     M Mr. & Mrs. Hendrix",
    "Leila Strassen,    F Ms. Henson",
    "Nora Janssen,      F Mrs. Janssen",
};
```

Figure 12.26. The class list.

nature of the database they represent. By using a separate file, we avoid the danger of accidentally changing the source code when intending to change only the database.

Second box: Using a menu.

When a illness letter must be sent home with the children, the first step in composing it is to select one of the illnesses from the school's list. This is easy to do using the `menu()` function which displays a prompt, then reads and validates the input.

Third box: Compiling a list of symptoms.

- The goal here is to compile a list of symptoms for the selected illness. Later, this list will be printed as part of the letter to the parents. We start with only a tab character in the array named `symptomlist`, then use a `for` loop to add symptoms one at a time to the end of the list. At the end of the loop, we terminate the list.
- First inner box: The same list of symptoms will be displayed each time around the loop, and the user will be able to choose one or to quit.
- Second inner box: Adding to the end of a string. After a symptom is selected, it is added to the end of the symptom list, which was initialized to just a tab character. Each symptom in the table includes a comma and a trailing space so that it will be easy to make a readable string by concatenating them together.
- We could use `strcat()` to do this job. However, that is very inefficient. Each time `strcat()` is used, the function scans the entire list from the beginning to the end. As the list becomes longer, it scans and rescans the same things.
- It is much more direct to use `strcpy()` or `strncpy()` to append the new word to the end of the list. However, doing so requires us to keep track of the current length of the list. For this purpose, we use

This function is called from `main()` in Figure 12.24, which asks for and validates a password before it allows entry to this part of the program. It calls the function in Figure 12.25 to print each individual letter. The include files are given in Figures 12.25 and 12.26.

```
#include "ills.h"
#include "class.h"

void writeNotice( void )
{
    char symptomList[ BUFLEN ] = "\t";
    int n, what;
    int choice, lenSymptom, lenList;
    char* end;

    what = menu_i( "Select an illness:", ILLS, illness );

    for (end=symptomList, lenList = 1; lenList < BUFLEN; ){
        choice = menu_i("Choose a symptom:", SYMPTOMS, symptom);
        if (choice == 0) break;

        lenSymptom = strlen( symptom[choice] );
        strncpy( &symptomList[lenList], symptom[ choice ], BUFLEN-lenList );
        lenList += lenSymptom;
    }
    symptomList[ lenList ] = '\0';

    for (n=0; n<CLASS_SIZE; ++n) compose( n, what, symptomList );
}
```

Figure 12.27. Preparing the form letter.

`strlen()` to calculate the length of the word being added and (eventually) add this number to `lenList` to be used the next time around the loop.

- We use the current list length, also, to avoid buffer overflow. At any given time, the space left in the buffer is `BUFLEN - lenList`. When we copy the new symptom onto the end of the symptom list, we never copy more than this many characters. It is *extremely important* to be careful of any use of `strcat()` or `strcpy()`, and to ensure that the resulting string actually will fit into the space allotted *before* the copy operation is performed.
- Because `strncpy()` does not always put a terminator on the end of the string, we must do this job after leaving the `for` loop. The variable `lenList` stores the number of meaningful characters that have been stored in the array and is the subscript of the first unused array position.

This function is called from `WriteNotice()` in Figure 12.28. It parses the data for one student, then prints a notice for that student's parent.

```
void compose( int who, int what, string ills )
{
    char child[ strlen( student[who] ) + 1 ];
    char * cursor, * parent, * gender;
    int choice;

    strcpy (child, student[who] );          /* Copy data before modification. */
    cursor = strchr( child, ',' );          /* Find the first comma. */
    *cursor++ = '\0';                       /* Store nul, point at next char. */

    while (isspace( *cursor )) ++cursor;    /* Skip whitespace. */
    if (*cursor++ == 'F') gender = "her"; else gender = "him";

    while (isspace( *cursor )) ++cursor;    /* Skip whitespace. */
    parent = cursor;                        /* First letter of parent name. */

    /* Parent, child, and illness have now been identified.  Print the letter. */
    puts( "\n\n Spring Glen Elementary School\n Communicable Disease Notice\n" );

    printf( " Dear %s:\n\n Your child, %s,", parent, child );
    printf( " has been exposed to %s at\n school. ", illness[ what ] );
    printf( " Please watch %s carefully for %s", gender, incubate[ what ] );
    printf( " and consult\n your doctor if you notice symptoms. ");
    printf( " Typical symptoms are:\n %sand general illness.", ills );
    printf( "\n\n Sincerely,\n %s, %s \n\n\f", TEACHER, CLASS );
}
```

Figure 12.28. Composing the letters.

Fourth box: Printing all the letters.

All of the information we need to write the form letter has now been collected and made ready. The only thing that remains is to write and address the letters. This `for` loop calls the `compose()` function once for each child in the class. The function prints the letter.

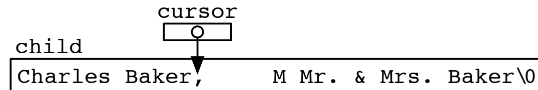
Notes on Figure 12.28. Composing the letters. This function does string processing, that is, it starts with a string, then analyzes it, divides it into substrings, and uses the parts.

First box: Space to work.

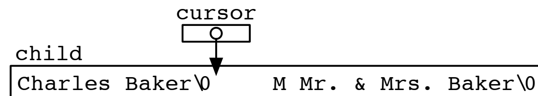
We define an array to big enough to hold a copy of the third parameter. This is necessary because the string will be changed during this process. When we are done, the three character pointers will be pointing

to the beginning of each field needed to print the form letter.

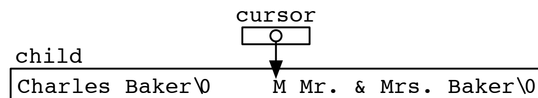
Second box: Isolating the child's name. We begin by copying the string parameter into the local buffer. Then we use `strchr()` to find the comma at the end of the child's name:



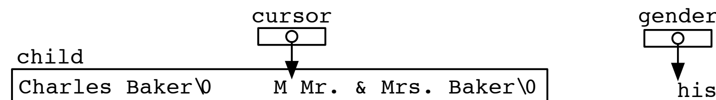
We replace the comma by a null terminator, then immediately move the cursor to the next array slot. This separates the name from the rest of the string.



Third box: Finding the gender. Now we use a tight loop to skip whitespace and position the cursor at the next input character:

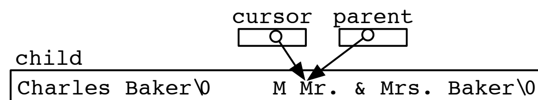


This character is `'M'`, so we set the `gender` pointer to point at the male pronoun `"him"` and immediately move the cursor to the next array slot.



Fourth box: Finding the parent's information.

Again, we use a tight loop to move the cursor past any whitespace. This will work properly whether there is no whitspace, one space, or many. Finally, we set the `parent` pointer to point at the first letter of the parents' name.



Fifth box: printing the letter.

The child's id number, illness, and symptoms are parameters to this function. The class data and illness data are global constant tables. The teacher's information is `#defined` globally. And we We have now positioned pointers to each of the other strings that will become part of the output letter. All that remains is a series of `printf()` statements that use the tables, constants, and strings we have set up.

12.5.3 Menu Processing and String Parsing

The `switch` statement was introduced back in Section 6.3 as a method of decision making. Putting a `switch` inside a loop is an important control pattern used in menu processing, especially when the menu choices are characters rather than numbers. The loop is used to display the menu repeatedly, until the user wishes to quit, while the `switch` is used to process each selection. One option on the menu should be to quit, that is, to leave the loop.

An example of this control structure is given in Figures 12.29 and 12.30. The program in Figure 12.29 presents a menu and processes selections until the user types the code `q` or `Q`. It illustrates the use of a `for` loop with a `switch` statement inside it to process the selections, a `continue` statement to handle errors, and a `break` statement to quit. This program also shows a use of `strchr()` and `strrchr()` in context.

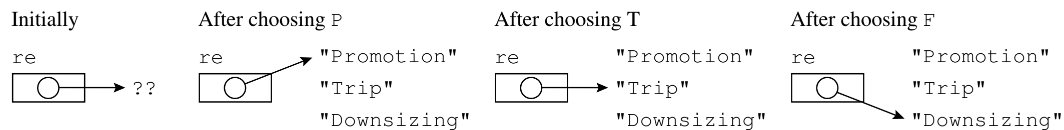
Notes on Figures 12.29 and 12.30. Magic memo maker and its flow diagram.

First box: the menu.

- As usual, a menu consists of an integer, `N`, and a list of `N` strings to be displayed for the user. This menu is designed to be used with `menu_c()`, so each string incorporates both a single-letter code and a brief description of the option.
- To add another menu option, we must increase `N`, add a string to the array, and add another case in the switch statement (second box). To make such modifications easier, the declarations have been placed at the top of the program. The `const` qualifier is used to protect them from accidental destruction, since they are global.

Second box: the menu loop. This menu-processing loop is typical and can be adapted to a wide variety of applications. Its basic actions are

- The first inner box reads the user's choice. We use `menu_c()` to display a menu, return a selection, and break out of the `for` loop if the user chooses the "Quit" option.
- The second inner box validates the choice and sets the variables. We use a `switch` to process other possible selections. For each legal choice, a case should be available that will call a function to process that case or set up the variables needed to process the case later. Here, we set the value of the string `re` for the memo generator:



The `break` at the end of each valid case takes control to the processing statements at the bottom of the loop.

This program calls the `compose()` function from Figure 12.31. Its control flow is diagrammed in Figure 12.30.

```
#include <stdio.h>
#include <string.h>
void compose( string name, string re, string event );  /* Modifies name. */

#define N 4    /* Number of memo menu choices */
const string menu[N] = {"P  Promote", "T  Trip", "F  Fire", "Q  Done"};

int main( void )
{
    char memo;         /* Menu choice.  */
    string re, event;  /* Subject and main text of memo. */
    char name[52];     /* Employee's complete name. */

    puts( "\n Magic Memo Maker" );

    for(;;) {
        memo = toupper( menu_c( " Next memo:", N, menu ) );
        if (memo == 'Q') break;    /* Leave for loop and end program. */

        switch (memo) {
          case 'P': re = "Promotion";
                    event = "You are being promoted to assistant manager.";
                    break;
          case 'T': re = "Trip";
                    event = "You are tops this year "
                            "and have won a trip to Hawaii.";
                    break;
          case 'F': re = "Downsizing";
                    event = "You're fired.\n "
                            "Pick up your final paycheck from personnel.";
                    break;
          default:  puts( "Illegal menu choice" );
                    continue;
        }

        printf( " Enter name: " );
        scanf( " %51[^\n]", name );
        compose( name, re, event );
    }

    return 0;
}
```

Figure 12.29. Magic memo maker.

This is a flow diagram of the program in Figure 12.29.

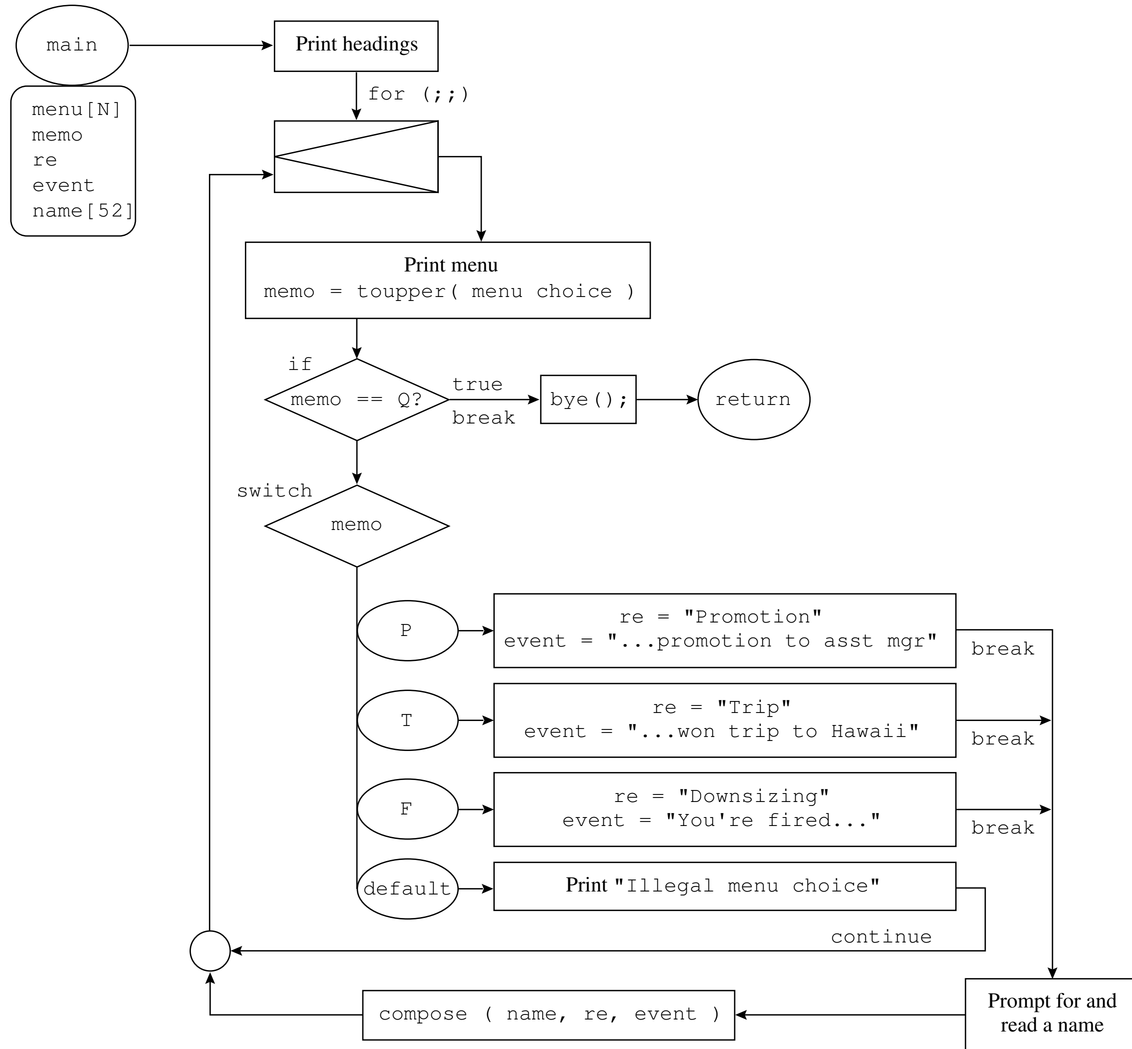


Figure 12.30. Flow diagram with a menu loop.

- The default case should print an error comment and `continue`. By going directly back to the loop start, the program avoids the processing actions that come at the end of the loop body. In the flow diagram (Figure 12.30), note how the arrow for `continue` returns to the top of the loop for the menu display rather than going through the process boxes.
- In the third inner box the request is processed. The common actions for generating the memo are performed after the end of the `switch`. When control reaches this point, invalid menu choices have been eliminated and valid ones fully set up. In this program, we call the `compose()` function to process the request.

The output from the main program begins like this:

```
Magic Memo Maker
Next memo:

        P  Promote
        T  Trip
        F  Fire
        Q  Done

Enter letter of desired item: T
Enter name: Harvey P. Rabbit, Jr.
```

In Figure 12.31, the memo is composed and processed.

Notes on Figure 12.31. Composing and printing the memo.

First box: memo header.

- The program prints a memo header that contains the employee's entire name, the boss's name, and the subject of the memo.
- The boss's name is supplied by a `#define` command. The other information is passed as arguments from `main()`.

Second box: finding the end of the last name.

- A name can have up to four parts: first name, middle initial (optional), last name, and a comma followed by a title (optional).
- For the salutation, we wish to print only the employee's last name, omitting the first name, middle initial, and any titles such as Jr. or V.P. Since the initial and titles are optional, it is a nontrivial task to *locate* the last name and separate it from the rest of the name.
- The last name is followed by either a comma or the null terminator that ends the string. We search the name for a comma thus:

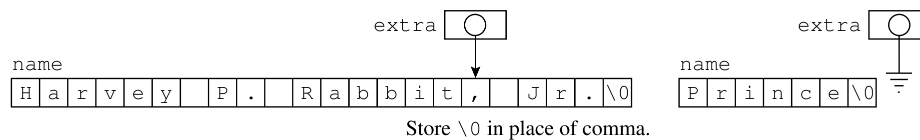
  ```
  extra = strchr( name, ',' );
  if (extra != NULL) *extra = '\0';
  ```

  If the result of `strchr()` is `NULL`, we know that the last name is the last thing in the string. Otherwise, the result of `strchr()` is a pointer to the comma. The following diagram shows how the `extra` pointer would be positioned for two sample inputs, with and without "extra" material at the end:

This function is called from Figure 12.29. As a side effect, the first argument, `name`, is modified.

```
#define BOSS  "Leland Power"

void compose( string name, string re, string event )
{
    string extra, last;             /* Pointers used for parsing name. */
    printf( "\n\n To: %s\n", name );
    printf( " From: The Big Boss\n" );
    printf( " Re: %s\n\n", re );

    extra = strchr( name, ',' );       /* Find end of last name. */
    if (extra != NULL) *extra = '\0';  /* Mark end of last name. */

    last = strrchr( name, ' ' );       /* Find beginning of last name. */
    if (last == NULL) last = name;
    else ++last;

    printf( " Dear M. %s:\n %s\n\n -- %s\n\n", last, event, BOSS );
}
```

Figure 12.31. Composing and printing the memo.

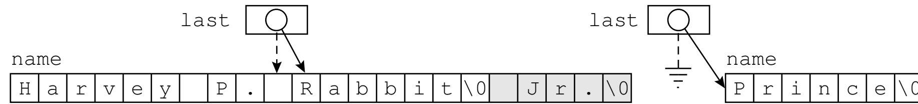


We replace the comma, if it exists, with a null character, using `*extra = '\0'`. We now know for certain that the last name is followed by a null character. This has the side effect of changing the value of the first argument of `compose()` by shortening the string. The part cut off is colored gray in the next diagram. This is an efficient but generally risky technique. Documentation must be provided to warn programmers not to call the function with constants, literal strings, or strings that will be used again. For a safe alternative, see Figure 12.27, where the string is first copied into a local buffer. The local copy then is modified; the original remains unchanged.

Third box: the beginning of the last name.

- To find the beginning of the last name, use `strrchr(name, ' ')` to search the (newly shortened) string for the rightmost space. The result is indicated by a dashed arrow in the diagram that follows.

```
last = strrchr( name, ' ' );
if (last == NULL) last = name;
else ++ last;
```



- If a space is found, as in the diagram on the left, the last name will be between that space and the (newly written) null terminator. The program then increments the pointer one slot so that it points at the first letter of the name.
- If the name has only one part, like Midori or Prince, the result of `strrchr()`, which is stored in `last`, will be `NULL`. In that case, the last name is the first and only thing in the string, so we set `last = name`.

Fourth box: printing the memo. The program now has located the last name and is ready to print the rest of the memo. The event (an input parameter) and the boss's name (given by a `#define` command) are used. Here is a sample output memo:

```
To: Harvey P. Rabbit, Jr.
From: The Big Boss
Re: Trip

Dear M. Rabbit:
You are tops this year and have won a trip to Hawaii.

-- Leland Power
```

12.6 What You Should Remember

12.6.1 Major Concepts

- The type `string` is not a built-in data type, but C has functions to support many of the basic operations that exist for built-in types.
- The standard `string` library, whose header file is `string.h`, contains many useful functions for operating on strings. In this chapter, we note the following:
  - To find the number of characters in a string, use `strlen()`. (Note that `sizeof` gives the size of the pointer part.)
  - To search a string for a given character or substring, use `strchr()`, `strrchr()`, and `strstr()`.
  - To compare two strings for alphabetic order, use `strcmp()` and `strncmp()`. (Note that `==` tells only whether two string pointers point at the same slot in an array.)

  - To copy a string or a part of a string, use `strcpy()`, `strncpy()`, `strcat()`, and `strncat()`. (Note that `=` copies only the pointer part of the string and cannot be used to copy an array.)

- A character array may be initialized with a string literal. The length of the array must be at least one longer than the number of letters in the string to allow for the null terminator.
- Literal strings, string variables, and character arrays all are called *strings*. However, these objects have very different properties and are not fully interchangeable.
- In memory, a string is represented as a pointer to a null-terminated array of characters.
- A string variable is a pointer and must be set to point at some referent before it can be used. The referent can be either a quoted string or a character array.
- The operators that work on the pointer part of a string are different from the functions that do analogous jobs on the array part.
- An array of strings, or ragged array, is a compound data structure that often saves memory space over an array of fixed-length strings. One common use is to represent a menu.
- An essential part of interactive programs is the display of a menu to provide choices to the user. A `switch` statement embedded in a loop is a common control structure for processing menus. The menu display and choice selection continue until the user picks a termination option.

12.6.2 Programming Style

- A loop that processes all of the characters in a string typically is a sentinel loop that stops at the null terminator. Many string functions operate in this manner.
- Use the "zero" literal of the correct type. Use `NULL` for pointers, `\0` for characters, and `""` for strings. Reserve `0` for use as an integer.
- Use the `typedef` name `string` instead of `char*` for variables. When you use this type declaration, you can use `string` as an ordinary type name.
- You can use a string literal to initialize a character array or a string variable. For the array, this is preferable to a sequence of individually quoted characters.
- It is not possible to change the contents of a string literal.
- Adjacent string literals will be merged. This is helpful in breaking up long output formats in a `printf()` statement.
- Use `char[]` as a parameter type when the argument is a character array that may be modified. Use `char*` when the argument is a literal or a string that should not be modified.
- Declare a `char` array to hold text input or for character manipulation and take precautions that the operations will not go past the last array slot. Do not attempt to store input in a `char*` variable.
- Using the brackets specifier makes `scanf()` quite versatile, allowing you to control the number of characters read as well as the character or characters that will terminate the input operation.

- A menu-driven program provides a convenient, clear user interface. To use a menu, a list of options must be displayed where each option is associated with a code. The user is instructed to select and key in a code, which then is used to control the program's actions. Having one option on the menu to either quit or do nothing is common practice. User selections should be validated.
- One or more data arrays parallel to a menu array can be used to make calculations based on a table of data. Each array in the set represents one column in the table that relates to one data property. If integer selections are used, the items in the arrays can be indexed directly using the choice value.
- The process of using a menu is much the same for all menu applications. We introduced two menu functions that automate the process, `menu()` for numeric codes and `menu_c()` for alphabetic codes. The ragged arrays used for menus should be declared as `const` if they are defined globally.

12.6.3 Sticky Points and Common Errors

These errors are based on a misunderstanding of the nature of strings.

Ampersand errors. When using `scanf()` with `%s` to read a string, the argument must be an array of characters. Do *not* use an ampersand with the array name, because all arrays are passed by address. A character pointer (a string) can be used as an argument to `scanf()`, but it first must be initialized to point at a character array. Do *not* use an ampersand with this pointer because a pointer already is an address, and `scanf()` does not want the address of an address.

String input overflow. When reading a string as input, limit the length of the input to one less than the number of bytes available in the array that will store the data. (One byte is needed for the null character that terminates the string.) C provides some input operations, such as `gets()`, that should not be used in cases of unknown input length because there is no way to limit the amount of data read. If an input operation overflows the storage, it will destroy other data or crash the program.

No room for the null character. When you allocate an array to store string data, be sure to leave space for the null terminator. Remember that most of the library functions that read and copy strings store a null character at the end of the data.

A pointer cannot store character data. Be careful to distinguish between a character array and a variable of type `char*`. The first can store a whole sequence of characters; the second cannot. A `char*` variable can only be used to point at a literal or a character array that has already been created.

Subscript validation. If the choice for a menu is not validated and if it is used to index an array, then an erroneous choice will result in accessing a location outside of the array bounds. This may cause almost any kind of error, from garbage in the output to an immediate crash.

The correct string operator. Strings can be considered in memory as two-part objects. The operations used on each part are different:

- To find the size of the pointer part, use `sizeof`; for the array part use `strlen()`.

- Remember to use the `==` operator when comparing the pointer parts, but use `strcmp()` to compare the array parts. When using `strcmp()` remember to compare the result to 0 when testing for equality.
- To copy the pointer part of a string, use `=`; to copy the array part, use `strcpy()` or `strncpy()`. Make sure the destination array is long enough to hold the result.
- The following table summarizes the syntax for initializing, assigning, or copying a string:

Operation	With a char Array	With a char*
Initialization	`char word[10] = "Hello";`	`char* message = "circle";`

Assignment	Does not copy the letters
	Cannot do `word = "Hello"`
		`message = "square";`
		
		`char* message = word;`

String copy	`strcpy( word, "Bye" );`	`strcpy( message, "Thanks" );`
String comparison	`strcmp( word, "Bye" )`	`message = word`

Quoting. A string containing one character is not the same as a single character. The string also contains a null terminator. Be sure to write single quotes when you want a character and double quotes when you want a string. The double quotes tell C to append a null terminator. You can embed quotation marks within a string, but you must use an escape character. Also, be careful to end all strings properly, or you may find an unwanted comment in the output.

12.6.4 New and Revisited Vocabulary

These are the most important terms and concepts presented or discussed in this chapter:

string	null string	parallel arrays
string literal	`NULL` pointer	C `string` library
string merging	conversion specifier	array of strings
two-part object	right and left justification	ragged array
type expression	string variable	menu functions
null terminator	string assignment	buffer overflow
null character	string comparison	menu selection validation
	string concatenation	

The following keywords and C library functions are discussed in this chapter.

`\"` and `\0`	`gets()`	`strncat()`
`char *`	`puts()`	`strchr()`
`typedef`	`strlen()`	`strrchr()`
`char []`	`strcmp()`	`strstr()`
`printf() %s` conversion	`strncmp()`	`sizeof` a string
`scanf() %s` conversion	`strcpy()`	`==` (on strings)
`scanf %[^?]` conversion	`strncpy()`	`=` (on strings)
	`strcat()`	

12.6.5 Where to Find More Information

- To create a string whose individual letters can be changed, as when a word is entered from the keyboard, we must create a character array big enough to store the longest possible word. The use of `malloc()` for this task will be covered in Chapter 16.
- Dynamically allocated memory can be used to initializer a string variable. This is explored in Chapter 16.
- In Chapter 16, Figure 16.3 we show how to create a ragged array dynamically.
- Pointers are introduced in Chapter 11 and the use of pointers to process arrays is explained in Chapter 17.
- A variety of effects can be implemented using the `scanf()` brackets conversion. A few more examples can be found on the text website. The interested programmer should consult a standard reference manual for a complete description.
- Chapter 8 discusses whitespace in the input stream, the problems it can cause, and how to use `scanf()` with `%c` to handle these problems.
- We recommend that you create a personalized file of tools containing useful definitions and functions that you have written or adapted. In this chapter, the `typedef` for strings and the two menu functions belong in that category. The file "my_tools" on the text website contains these functions and others that use or process strings. Among these are the functions `ocklock()`, `date()`, and `when()`, that make it easy to print out the date and time.

12.7 Exercises

12.7.1 Self-Test Exercises

1. (a) Given the following declarations and `printf()` statements, what is printed?

```
string s[4] = {"hello", "help", "save me", "groan"};
char t[] = "help";
string p = s[3];

printf( "%4s?\n", s[0] );
printf( "%s!  ", p );
printf( "%10s ", s[2] );
printf( "%i", t == s[1] );
printf( "%c!\n", s[0][0] );
```

   (b) The following statement will compile with no errors, but it will not work properly. What is wrong?

```
char t[5] = "wait";
strcpy( t, "finished" );
```

2. You wish to read either a single word or a name consisting of two or three words separated by spaces. Given the declarations shown, find and fix the errors in the following `scanf()` statements.

```
char word[10], name[20];
string w = word;     /* Make w point at first char in word */
string t;
```

a. `scanf( "%s", t );           /* Read a single word. */`
b. `scanf( "%9s", &word );      /* Read a single word. */`
c. `scanf( "%19s", name );      /* Read first, middle, last. */`
d. `scanf( "%9s", &w );         /* Read a single word. */`
e. `scanf( "%[^\n]", name );    /* Read entire name. */`

3. Using these variables,

```
string message;
int v;
```

write an `if` statement that will select the string `"good"` if the value of `v` is between 1 and 10, and the string `"bad"` otherwise. Then write a single `printf()` statement that will print the selected message.

4. Given the declarations shown, find and fix the errors in the following statements.

```
char name[12] = "";
char word[10] = "harmony";
string w = word; /* Make w point at first char in word~*/
string t;
```

(a) `if !  (strcmp( t, w )) puts( "Same words here." );`
(b) `t = '\0';`
(c) `name = word;`
(d) `strncpy( word, name, 10 );`
(e) `if (strcmp( name, word )) puts( "Same words here." );`
(f) `strcpy( w, &name );`

5. Trace the program that follows and show the output produced. Also, make a diagram (or several) of the message string. Use it to show the referent of each pointer after a string operation and the change in the letters of the message after each assignment.

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

int main( void )
{
    char mess[] = "Yes, I would truly love to see you on Friday.";
    string p1, p2, p3, p4, p5;

    p1 = strstr( mess, "to" );  *p1 = 'd';
    p2 = strchr( p1, 'y' );
    p3 = strchr( p2, ' ' );     *p3 = p1[2] = '\0';

    p3 = strrchr( mess, 'l' );  *(p1-1) = *(p3-1) = '\0';
    p4 = strchr( mess, ' ' );   *p4 = '\0';  p4++;
    p5 = strchr( p4, 't' );

    printf( "%c %s %s\n", *p4, p3, p2 );
    printf( "%s %c %s\n", mess, *p4, p1 );

    strcpy( mess, "Drew" );
    printf( "%c %s %s %s %s!\n", *p4, p3, p2, p5, mess );
}
```

6. Given the declarations shown for exercise 4, determine whether each of the following statements is true or false. If false, supply the correct answer.

   (a) `sizeof w` is 4.
   (b) `sizeof name` is 12.
   (c) `sizeof word` is 8.
   (d) `strlen( w )` is 10.
   (e) `strlen( name )` is 0.
   (f) `strlen( word )` is 10.

7. You are writing an interactive program to be used by the Baloo Balloon telephone order service. Right now, you are implementing only the user interface. The operator will answer the phone and take care of customers' needs. The program eventually will have functions for taking orders, canceling orders, checking on standing orders, and making complaints. At the end of the day, the operator must shut down the program before going home.

   (a) Define an array of strings to implement a menu for the operator's use.
   (b) Write a loop containing a call on a menu function to display the menu and a `switch` statement to process the menu selection. Assume that you will have functions to do the four tasks listed above. Be sure that invalid responses are handled properly, either here or by the menu function.
   (c) Draw a flowchart of your loop.

12.7.2 Using Pencil and Paper

1. You are writing an interactive program to search a database. At any time, the user will have several options: quit, ask for help, or perform a search based on title, author, or subject.

   (a) Define an array of strings to implement a menu for this program.
   (b) Write a call on a menu function to display the menu and receive a response.
   (c) Write a `switch` statement to process the menu selection. Pretend that there are functions you can call to process each case. Be sure that invalid responses are handled properly, either here or by the menu function.

2. The following are a set of declarations, a list of tasks (on the left), and input lines (on the right). Write a single `scanf()` statement that can perform each task correctly if given the corresponding input.

```
char sentence[80], w1[10], w2[10], c1, c2;
```

Task	Input
(a) Read entire line into `sentence`.	The house, I know, is red.
(b) Read the first word into `w1`, safely.	Paderewski's dog is black.
(c) Using different conversion specifiers, read the first two words into `w1` and `w2`.	Ho! Where did you come from?
(d) Read all but the last word into `sentence`.	Happy day, I can get zzzzz.
(e) Read the initials into `c1` and `c2` and the name into `w1`.	T. P. Hammond

3. Trace the program that follows. After each of the three groups of statements, diagram the array and the positions of the pointers `p1`, `p2`, and `p3` relative to the message array.

```
#include <stdio.h>
#include <string.h>
int main( void )
{
    char mess[] = "Murphy Brown had a baby "
                  "and the President had a cow.";
    string p1, p2, p3;

    p1 = strstr( mess, "ow" );
    p2 = strstr( p1, "ow" );
    p3 = strstr( p1+1, "ow"  );

    p1 = strchr( mess, ' ' );
    p2 = strchr( p1, 'r' );
    p3 = strrchr(p1, 'r' );

    p1 = strstr( mess, " and" ) ;
    *p1 = '\0';
    p2 = strrchr( mess, ' ' );
    strcpy( p2, " zoo" );
}
```

4. What is the `sizeof` each of the following:

   (a) `char a[] = { 'Z', 'e', 'r', 'o' };`
   (b) `char a[] = "One";`
   (c) `char a[6] = "Two";`
   (d) `string c = "Three";`
   (e) `string d[3];`

5. Show the output produced by the following program:

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

int main( void )
{
    int k;
    string p;
    char line[80] = "";
    string word[] = {" attention","like","sleep","eat","dogs "};

    for (k = 4; k >= 0; --k) {
        strcat( line, word[k] );
        if (k < 4 && k > 1) strcat( line, " and " );
    }
    printf( "%s.\n", line );

    strcpy( line, "I want to eat horses." );
    strncpy( &line[10], "seek", 3 );
    p = strrchr( line, 'r' );
    *p = 'u';
    puts( line );
}
```

6. What is the output from the following program?

```
#include <stdio.h>
#include <string.h>
void stutter( char w[], int n );

int main( void )
{
    string s = "Hello";
    char word[10] = "Goodbye";
    puts( "\n---------------" );
    stutter( s, 3 );
    stutter( word, strlen( s ) );
    puts( "\n---------------" );
}

/* ------------------------------------- */
void stutter( char w[], int n )
{
    int k;
    for(k = 0; k < n; ++k) printf( "%s", w );
}
```

7. Given the variables `action` and `task` declared in the following, determine whether each of the listed operations is legal and meaningful.

```
char t[10] = "wait";
string action = t;
char* task = "mail"
```

   (a) `action = "go";`
   (b) `*action = 'b';`
   (c) `scanf( "%9s", action );`
   (d) `printf( "%s\n", action );`
   (e) `task = "go";`
   (f) `*task = 'b';`
   (g) `scanf( "%9s", task );`
   (h) `printf( "%s\n", task );`

8. Rewrite the `find_max()` function in Figure 10.23 to have an array of strings as its data parameter. Have it find the minimum string; that is, the one that comes first in alphabetical order. Use `strcmp()`. Return the subscript of that string.

12.7.3 Using the Computer

1. A better banner.

   (a) Write a function that prints a personalized banner for your output. Include your name , course number, assignment number, assignment name, and the date. The new function should take the

lab number and title as parameters. Use a `#define` statements for your name and course number. Use the `when()` function to get the date and time from the system clock. The definition and documentation for this function can be found on the text website in the section named Input and Output. The prototype for your function should be

```
void  banner( int lab_no, string title );
```

Your banner should look like this:

```
-----------------------------------------------------
            Mary Jane Talmadge
            CS 110
            Lab 8: My Banner
            Mon Aug  9 2005  18:19:23
  -----------------------------------------------------
```

(b) Put your `banner()` function, its `#define` statements, and the `when` function into your personal `my_tools.c` file. Compile this file as part of your project when you compile your other programs. Call your banner function at the beginning of each program you write from now on.

(c) Put the prototype for `banner()` into your tools header file, `my_tools.h`.

(d) Test your function by calling it from some program you have already debugged.

2. A string search.

   Write a function, `rightmost()`, that has two input parameters (a character and a string) and one output parameter (an integer). Search the argument string for copies of the specified character and count the number of occurrences. If the number is nonzero, return the subscript of the rightmost occurrence through the output parameter. Return the number of occurrences as the value of the function. Write a main program with a query loop to permit you to test the `rightmost()` function.

3. Which is longer?

   (a) Write a function, `lencmp()`, that has two string parameters. It should return a positive number if the first string is longer than the second, 0 if they are the same length, and a negative value if the second is longer.

   (b) Write a function, `longer()`, that has two `string` parameters and returns a value of type `string`. It should call `lencmp()` to compare the lengths of the two strings and return a pointer to the longer one.

   (c) Write a main program with a query loop to permit you to test these two functions.

4. Substitution.

   Write a function, `substitute()`, that has three parameters: two characters and a string. It should search the string for copies of the first character, and change them all to the second character. Return the number of times a substitution was made. This function might be part of a larger hangman or cryptogram program.

   Write a main program with a query loop to permit you to replace several letters in the same sentence. For each data set, prompt the user to enter two characters, the letter to find, and its replacement.

5. Metric to English.

   The following table lists the factors for converting a variety of metric measurement units to the English equivalents. To perform the opposite conversion, you simply divide (rather than multiply) by the given factor. Implement this table as three global arrays of constant strings with a parallel array of constant `double`s.

To Convert	**From**	**To**	**Multiply By**
Acceleration	m/s^2	ft/s^2	3.281
Area	m^2	ft^2	10.76
Density	kg/m^3	lbm/ft^3	0.06243
Energy	J	Btu	9.478E−4
Force	N	lbf	0.2248
Length	m	ft	3.281
Length	m	mile	6.214E−4
Mass	kg	lbm	2.205
Power	W	hp	1.341E−3
Pressure	N/m^2	lb/in^2	1.450E−4
Velocity	m/s	ft/s	3.281
Velocity	m/s	mph	2.237

   Write a program that will present a menu of possible conversions to the user and permit selection of one. Then ask whether he or she wishes to convert from metric to English units or from English to metric. Finally, prompt the user to enter a measurement and convert it. Continue presenting the menu until the user selects the "Quit" option.

6. Cost estimate.

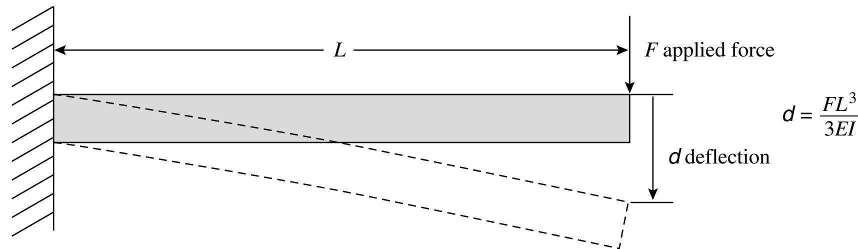
   (a) Define a pair of parallel arrays that will be used for estimating the cost of a repair job. The first array should contain the names of various parts; the second array, the price of each part. For example, these arrays might contain data like this:

```
{ "muffler", "oil filter", "spark plugs", ... }
{ 39.95, 4.95, 6.00, ... }
```

   (b) Write a program that will present a menu of parts for the user, then permit him or her to select any number of parts from the menu. After selecting a part, the program should prompt for the quantity of that part (one or more) that will be used for the repair, calculate the price for those parts, and add it to a total price. When the user selects "Done" from the menu, print the total price of all parts needed, with a sales tax of 6%.

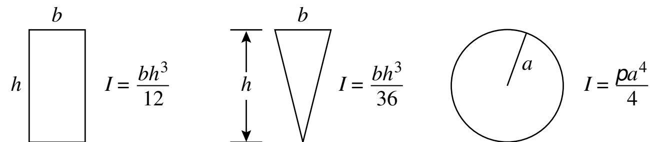
7. Deflection of a Cantilever Beam for Different Cross Sections and Materials.

   Write a program that will calculate the deflection of the end of a cantilevered steel beam projecting horizontally from a vertical wall as follows. The beam is a solid having a cross section in the shape of a rectangle, a circle, or an equilateral triangle. An analysis of the beam based on Hooke's law of stress and strain gives an equation for δ (delta), the deflection of the end of the beam due to a vertical downward force, F:



where F is the applied force (N), L is the beam length (m), E is Young's modulus (a property of the material, N/m^2), and I is the moment of inertia (a property of the geometry of the cross-sectional area, m^4).

(a) Write three functions, one for each of the cross-sectional areas, as shown. Each function should prompt for the values of `b`, `h`, or `a`, as appropriate, and validate that the inputs are positive. Return the moment of inertia to the main program. Use the formulas for moments of inertia shown adjacent to each shape.



(b) Use an array to store Young's modulus, E, for the following materials:

Material	$(E \times \mathbf{10}^9$ $\mathbf{N/m^2})$
Steel (cold rolled)	204
Brass (70–30)	109
Aluminum alloy	68.9
Molybdenum	345

(c) The main program should ask the user to choose from one menu for materials, then from a second menu for the cross-section shape (`r=rectangle`, `t=triangle`, or `c=circle`). Use a numeric menu for the first choice, a character menu for the second. Last, prompt the user for the length and applied force (both must be greater than 0.0). Use validation loops for all inputs. Use a `switch` statement to call the appropriate cross-section function based on the second menu choice and compute the deflection amount based on a value of E from the first choice, I computed for the second choice, and the input values of F and L. Finally, print the resulting value of δ.