Chapter 8

Character Data

This chapter is concerned with the character data type. We show how characters are represented in a program and in the computer; how they can be read, written, and used in a program; and how they are related to integers.

8.1 Representation of Characters

A character is represented using a single byte (8 bits). We have shown how numeric values (integers and reals) are represented using the binary number system. Characters also are represented by bits. However, when we think of a text character, a binary number is not the first thing that comes to mind. There is no obvious way in which numbers or bit patterns correspond to the letters, digits, and special characters on a keyboard. So people have invented arbitrary codes to represent these characters. The most common of these is the ASCII (American Standard Code for Information Interchange) **code**, which is listed in a table in Appendix . Each ASCII character is represented by 7 bits,[1] which are stored on the rightmost side of a byte. You can think of the value of this byte either as a character or an integer.

The ASCII characters are listed in the appendix in numeric order, according to the value of their bit representations. We can find a character in the table and see its code or use a numeric code as an index into the table to determine the associated character. For historical reasons, the indexing number often is listed in two forms, decimal and hexadecimal.[2]

The ASCII codes from 33 to 126 represent printable characters; most of these are letters of the alphabet in upper or lower case. Note that each lower-case letter is 32 greater than the corresponding upper-case letter, for example, 'A' + 32 == 'a'. A single bit in the representation, called the *case bit*, makes this difference. This is illustrated in Figure 8.1

[1] International ASCII uses 8 bits to represent each character and Unicode uses 16 bits.

[2] The hexadecimal number system is discussed in Chapter 15 and in Appendix E.

In the ASCII code, upper and lower case letters differ by only one bit, the sixth when counting from the right. This bit has the binary value of 32.

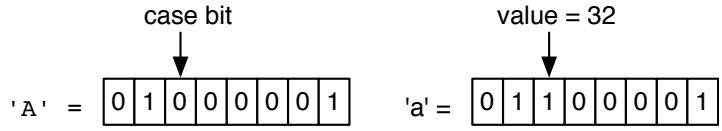

Figure 8.1. The case bit in ASCII.

Data type	Define Constant Limits	Range
`signed char`	`SCHAR_MIN..SCHAR_MAX`	$-128 \ldots 127$
`unsigned char`	`0..UCHAR_MAX`	$0 \ldots 255$
`char`	`CHAR_MIN..CHAR_MAX`	Same as `signed` or `unsigned char`

Figure 8.2. Character types.

8.1.1 Character Types in C

In reality, characters are just very short integers in C; the single byte of a character holds a number. Anything you can do with an integer, you can do with a character. Anything you can do with a character, you can do with 1 byte of an integer. *There is no difference between a character and an integer* except the number of bytes used and the format specifiers used to read and print the two types.

8.1.2 The Different Interpretations

Figure 8.2 lists the character types defined by the C standard. The type `signed char` is not used often, and when used, it is generally thought of as a very short `int`. The more common type, `unsigned char`, is useful primarily when a program must store large quantities of small positive integers in memory and conserve storage to avoid running out of it. In this case, the `unsigned char` actually is being used as a very short `unsigned int`. A program that does image processing is an example of such an application (see Chapter 18). The type `char` is used for most character-handling applications.

A character code such as ASCII uses a fixed number of bits to represent the letters of the alphabet, numerals, punctuation marks, special symbols, and control codes. ASCII uses 7 bits and therefore can represent 128 codes. The C standard permits type `char` to be defined either as signed or unsigned values; the compiler manufacturer makes that decision. Normally, it is of no concern to the programmer, since the index values for the ASCII table (0–127) are present in both forms, so there is not such a **portability** problem as there is with `int`. However, an international version of the ASCII code uses all 256 index values. The extra codes are used to represent additional letters and special symbols used in various European languages. In systems that use International ASCII, `char` is implemented as an unsigned type.

Character constants can be written symbolically or numerically. The symbolic form is preferable because it is portable; that is, it does not depend on the particular character code of the local computer.

Meaning	Symbol (Portable)	Decimal Index (ASCII Only)
The letter *A*	`'A'`	65
Blank space	`' '`	32
Newline	`'\n'`	10
Formfeed	`'\f'`	12
Null character	`'\0'`	0

Figure 8.3. Writing character constants.

Code	Meaning	Code	Meaning	Code	Meaning
`\0`	Null	`\a`	Attention	`\"`	Double quote
`\n`	Newline	`\b`	Backspace	`\'`	Single quote
`\r`	Return	`\f`	Formfeed	`\\`	Backslash (escape)
`\t`	Horizontal tab	`\v`	Vertical tab		

Figure 8.4. Useful predefined escape sequences.

8.1.3 Character Literals

Most often **character literals** are written in C using single quotes, like this: `'A'`. However, nothing in C is simple. The character literal inside the quotes can be written two ways, as shown in Figure 8.3:

- If it is an ordinary printable character, we write it directly in quoted form. Thus, the first letter of the alphabet is written as `'a'` (lower case) or `'A'` (upper case).
- Some characters are written with an **escape code** or escape sequence. This consists of the \ (escape) character followed by a code for the character itself. The predefined symbolic escape codes are listed in Figure 8.4, a few of which we already used in output formats.

Escape code characters are included in C for two different reasons: to resolve ambiguity and to provide visible symbols for invisible characters. Three of the escape codes, `\'`, `\\`, and `\"`, are used to resolve lexical and syntactic ambiguity. The backslash (also called *escape*), single quote, and double quote characters have special meaning in C, but we also need to be able to write and process them as ordinary characters. The escape character tells C that the following keystroke is to be treated as an ordinary character, not as an element of C syntax.

The other escape characters are invisible; their purpose is to cause a side effect. For example, the *attention* code, `\a`, is used to alert the user that something exceptional has happened that needs attention. (Note that `'\a'` and `'a'` are very different; the first means "attention" and should cause most computers

to beep; the second is an ordinary letter.) A very important escape code is the *null character*, `\0`, which is used to mark the end of every character string and will be discussed further in Chapter 12.

One set of escape code characters that we use frequently are called **whitespace** *characters*; they affect the appearance of the text but leave no visible mark themselves. The list includes newline, `\n`; return, `\r`; horizontal tab, `\t`; vertical tab, `\v`; formfeed, `\f`; and the ordinary space character. The two tab characters insert horizontal spaces or vertical blank lines into the output (the precise number of horizontal or vertical spaces depends on the system). Whitespace characters often are treated as a group in C and handled specially in a variety of ways.[3] Note that whitespace characters are all invisible, but many invisible characters, including null and attention, are not classified as "whitespace".

8.2 Input and Output with Characters

Character input and output can be performed using the standard `scanf()` and `printf()` functions. In addition, other special functions exist just for characters. Some of these functions have subtle difficulties associated with them, which we discuss.

8.2.1 Character Input

The standard library functions for reading characters are

1. `getchar()`. This function has no parameters. It reads a single character of input and returns it as an `int`. The character is stored in the rightmost part of that `int`, and bytes to the left of the character are filled with **padding** bits. When the padded value is stored in a character variable, the padding is discarded. This process of adding and stripping off padding is automatic, and transparent, and can be ignored by beginning programmers. Normally, the value returned by `getchar()` is used in an assignment statement.

   Example: `ch = getchar();`

2. `scanf()` *with a* `"%c"` *conversion specifier.* In this format, there is *no space* between the opening quotation mark and the `%c` specifier. This will read the next input character, whether or not it is whitespace, and store it in the address provided. This version is equivalent to using the `getchar()` function.

   Example: `scanf( "%c", &ch );`

3. `scanf()` *with a* `" %c"` *conversion specifier.* In this format, there is a space in the format string before the `%c` specifier. The space causes `scanf()` to skip leading whitespace (if any exists) before reading a single nonwhitespace character and storing it in the address provided. This is similar to the manner in which other data types are scanned.

   Example: `scanf( " %c", &ch );`

[3] These ways will be explained as they become relevant to the text.

Keyboard input is buffered. Whether you are entering numeric or character data into a program, your input is not sent immediately to the program. Until you hit the Enter key, it is displayed on the screen but remains in a holding tank called the **keyboard buffer** so that you can inspect and change it, if necessary. It is not the case that as soon as you type a character the program will read it and begin processing. Some languages provide this feature, but C is not one of them.[4] After you hit Enter, your input moves to another area called the **input buffer** and becomes available to the program. The program will read as much or as little as called for by the `scanf()` or `getchar()` statement. Unread data remain in the input buffer and will be read by future calls on `scanf()` or `getchar()`.

Whitespace characters complicate input. When reading integers or floating-point numbers, `scanf()` skips over leading whitespace characters and starts reading with the first data character. However, with the `"%c"` specifier, leading whitespace is not ignored. If the first unread character is whitespace, that is what the system reads and returns. The function `getchar()` does the same thing. It reads a single character, which might be whitespace. Reading data in this manner leads to surprising behavior if the input contains unexpected whitespace characters such as `\n` and `\t`. Since these are not visible, it is easy to forget that they may be present.

In any text-processing program, it is frequently necessary to skip over indefinite amounts of whitespace. As mentioned previously, the behavior of `scanf()` can be changed by adding a single blank to the format: `" %c"`. The space inside the quotes and before the `%c` tells `scanf()` to skip over leading whitespace, if any exists. However, there is no way to force `getchar()` to skip over these invisible characters. For this reason, we usually use `scanf()` rather than `getchar()` to input single characters interactively.

8.2.2 Character Output

Character output is relatively straightforward. The `stdio` library provides two ways to display or print a single character:

1. `putchar()`. When only one character of output is needed, `putchar()` is the easiest way to do the job; we simply pass it the character we want to display. For example, to move the screen cursor to the beginning of the next line, we might say `putchar( '\n' );`. Note that the `putchar()` function does not automatically move the cursor to the next line as `puts()` does.
2. `printf()` *with a* `%c` *conversion specifier.* When printing a character mixed in with other kinds of data, we use `printf()` with the `%c` format specifier. Example:

   ```
   printf( "Child is %c, %i years old.", gender, age );
   ```

[4] Some old PC-based single-user C systems support the unbuffered character input functions `getch()` and `getche()`, in a library named `conio`. When using `getch()`, any character that the user typed would go directly to the program, rather than being held in the keyboard buffer until a newline was typed. This seems like a convenient function and it was popular with students. It is not supported by the standard because modern systems are multi-processing systems and cannot make a direct connection between any input device and any one process among the set that is running concurrently. We recommend against using any nonstandard feature because it is not portable.

```
#include <stdio.h>

int main( void )
{
    char ch;

    puts( "\n Demo: Printing the ASCII codes." );

    printf( "\n Please type a character then hit ENTER: " );
    ch = getchar();

    printf( " The ASCII code of %c is %i \n\n", ch, ch );
}
```

Figure 8.5. Printing the ASCII codes.

> Of course, a specific (nonvariable) character can be included in the format string itself. As with the other data types, it is possible to specify a field width between the `%` and the `c`, and the printed character will be right or left justified in the field area, depending on the sign of the width specifier.

Since characters *are* integers, it is legal to read or print them as integers. When you read an integer that is the ASCII code of a letter and print that number using a `"%c"` format or `putchar()`, you see the letter. Conversely, when you read a letter and print it using a `"%i"` format, you see a number.[5] This technique is demonstrated in Figure 8.5.

Notes on Figure 8.5. Printing the ASCII codes.

First box: declaration. We declare a `char` variable. In the remaining code, we use it to perform both character and integer output.

Second box: character input. We read a character (one keystroke); its ASCII code is stored as a binary integer in the `char` variable. The character is not read until the Enter key is pressed.

Third box: output. We print the input character twice: first as a character, using `%c`; then as an integer, using `%i`. Sample program output looks like this:

```
Demo: Printing the ASCII codes.

Please type a character then hit ENTER: A
The ASCII code of A is 65
```

[5] It also is possible to print the hexadecimal form of the character's index by using a `%x` conversion specifier. See Chapter 15.

We demonstrate various ways to read and write single characters and how a whitespace character in the input can cause unexpected results.

```
#include <stdio.h>

int main( void )
{
    char input;
    char money = '$';
    char star  = '*';

    putchar( '\n' );  putchar( star );  putchar( 42 );   putchar( '\n' );
    printf( " Do you need %c (y/n)? ", money );

    scanf( "%c", &input );

    while (input == 'y') {
        printf( " Here is %c5.00\n", money );
        printf( "\n Do you need more (y/n)? " );
        input = getchar();   /* This code is wrong!  Use scanf( " %c", &input ) */
    }

    printf( " OK --- Bye now.  %c \n", '\a' );
}
```

Figure 8.6. Character input and output.

8.2.3 Using the I/OFunctions

The program in Figure 8.6 illustrates the use of the four character input and output functions and demonstrates how a simple program can produce very confusing output if the problem of whitespace in the input is not addressed.

Notes on Figure 8.6. Character input and output.

First box: character output.

- The function `putchar()` prints a single character. Its argument can be a character variable, a literal character, or an integer. If the argument is an `int`, the rightmost byte of its value is interpreted as an index for the ASCII code table, and the character in that position is printed. The command `putchar( 42 )` prints an asterisk because 42 is the ASCII code for `*`.
- We also can print a single character using `printf()` with `%c`; in this case, we print a dollar sign. If we try to print an integer with a `%c` conversion specifier, we see the character that corresponds to that integer's

index in the code table. For example, the output from `printf( "%c", 42 )` would be an `*`.

Second box: reading the first response.

- The line `scanf( "%c", ... )` reads a single character of input.
- When a program begins executing, the input buffer is empty. Normally, the user will not enter any input until prompted, so the user's response to the first prompt will be the only thing in the input buffer. The first character of that response will be read, while the newline character generated by the Enter key will remain in the buffer. We presume that the user will type `y`, causing control to enter the loop.

Third box: the loop.

- In this loop, we "give" $5.00 to the user and prompt for another response:

```
Here is $5.00

Do you need more (y/n)?
```

- This time we use `getchar()` to read the next input character. If it is `y`, we will stay in the loop; for any other input (including whitespace), we leave the loop.
- This logic seems simple enough, but *it does not work*. As shown in the following output, the user sees the second prompt but the program quits and says goodbye without giving that user a chance to enter anything. The reason for this problem is explained in the following section.
- The complete output for this run is

```
**
Do you need $ (y/n)? y
Here is $5.00

Do you need more (y/n)?  OK --- Bye now.
```

Fourth box: the closing message. The `%c` "prints" the escape character `\a`. On some systems, if the computer has a sound generator and the volume is turned up, you should hear a ding when you print the attention character, but you see nothing. On other systems, the output may be visible (for instance, a small box) but not audible.

Problems with `getchar()`. A common programming error was illustrated in Figure 8.6. After `scanf()`, the program initially performs as expected; if the input is `y`, it enters the loop and "gives" the user $5.00. Then the loop prompts the user to enter another `(y/n)` response. However, when the second prompt is displayed, the system does not even wait for the user to respond; it simply quits. Why?

When entering the answer to the first question (above the loop), the user types `y` and hits the Enter key. This puts the character `y` and a newline character into the input buffer. The newline character is necessary because, in most operating systems, the system does not send the keyboard input to the program until the user types a newline. The `scanf()` above the loop reads the `y` but leaves the `\n` in the buffer. At the end

of the first time around the loop, that `\n` still is sitting in the input buffer, unread. The loop prompts the user for a choice, but the program does not wait for a key to be hit because input already is waiting. The `getchar()` then reads the `\n`, emptying the buffer. Since `\n` is not equal to `y`, the loop ends and the program says goodbye.

Now, if the user had typed `yyy` followed by a newline instead of a single `y` at the first prompt, the characters waiting in the input buffer would have been `yy\n`. The input to the second prompt then would have been `y`, and the program would have given the user another $5.00 bill. Altogether, the user would get three bills before the program could read the newline and leave the loop, all with no further typing by the user. The resulting output would be

```
**
Do you need $ (y/n)? yyy
Here is $5.00

Do you need more (y/n)?  Here is $5.00
Do you need more (y/n)?  Here is $5.00

Do you need more (y/n)?  OK --- Bye now.
```

Using `scanf( "%c", &input )` in place of `input = getchar()` does not solve the problem, because `scanf()` with `"%c"` works the same way as `getchar()`. However, we can solve this whitespace problem by using a single space in the format for `scanf()`. Replace the call on `getchar()` in Figure 8.6 by this call on `scanf()`:

```
scanf( " %c", &input );
```

Note space in format

With this change, everything will work as intended: The program will query the user, wait for a response every time, and do the appropriate thing. A typical output would look like this:

```
**
Do you need $ (y/n)? y
Here is $5.00

Do you need more (y/n)? y
Here is $5.00

Do you need more (y/n)? n
OK --- Bye now.
```

If the user initially were to type `yyy`, the output would begin as shown earlier, but after three times through the loop, the user would have a chance to enter responses again.

```
void skip_ws( void ) {
    int ch;                          /* Most recently read character. */
    while (isspace( ch=getchar() ); /* Tight loop; exit when non-space is read. */
    ungetc( ch, stdin );             /* Put non-space character back into stream. */
}
```

Figure 8.7. A function for skipping whitespace.

8.2.4 Other ways to skip whitespace.

Skipping whitespace by inserting a space into a format specifier is a little-known technique, even though it is easy to do and easy to understand. A common mistake among programmers is to use the C function `fflush()` to do this task. The C standard sates that *the function* `fflush()` *is defined only for output streams*, not for input. Sometimes it seems to work for input, but it does so for indirect and subtle reasons, and only in some implementations. A correct alternative technique for skipping whitespace is to use a simple loop, as shown in Figure 8.7.

Notes on Figure 8.7. A function for skipping whitespace during keyboard input.

Line 1: The variable declaration. We declare an `int` variable to store the character being read. Although this seems wrong, remember that the function `getchar()` returns an `int`, not a `char`.

Line 2: Reading and testing the data.

- The function `isspace()` is explained in Section 8.3.5. Briefly, it returns `true` if its argument is a whitespace character, `false` otherwise.
- This line is a "tight loop"; it has no body at all and does nothing except read and test, read and test, until the input is non-whitespace.
- This loop will work correctly for any combination and any number (zero or more) of whitespace characters. It will just keep reading until the user types something else. (Remember that many invisible characters are not classified as "whitespace".)
- We store the input character in a variable because we will use the final character read after the end of the loop.

Line 3: Get and give back. We leave the `getchar()` loop when a non-whitespace character is found. However, that is too late! That character is real input and cannot be processed in this function. The easiest remedy is to put that character back into the input stream so that it can later be read by the proper part of the program. Happily, C supplies a function for this task: `ungetc()`. The arguments in this call are the character that must be put back into the input stream, and the name of the input stream. This function will be explained more fully in the chapter about streams and files, Chapter 14.

A useful tool. This is a function that will often be useful in programs that analyze character input. Copy the definition into your personal file of C-tools.

8.3 Operations on Characters

8.3.1 Characters Are Very Short Integers

The basic operations defined for characters are the same operations that are defined for integers because, technically, characters *are* integers in the range $0 \ldots 255$ or $-128 \ldots 127$. Some kinds of data (such as digital images) are composed of a very large number of very small integers. In such cases, it is useful to minimize the amount of storage occupied by the data, so the data are stored as type `char` or `unsigned char` rather than `short int` or `int`. In such applications, it is important that all the integer operations can be applied to variables of type `char`.

The common use of type `char`, however, is to represent characters. Even then, many integer operators are useful; These are summarized in Figure 8.8. A little caution is warranted here; some integer operations are legal but not useful with characters. For example, it makes no sense to multiply or divide one character by another. Unfortunately, useless or not, the compiler will not identify such expressions as errors. In addition to the basic integer operators, there is also a set of functions in the `ctype` library that can manipulate character values. We now examine some of the library functions and take a closer look at the operators in Figure 8.8.

8.3.2 Assignment

We have seen that the values of both character and integer variables can be assigned to a `char` variable. As long as the integer value is not too big, everything will be fine (large values lead to overflow). Automatic coercion will shorten the integer and a single byte will be assigned. Literal characters also can be assigned to `char` variables. The columns of Figure 8.3 show two different values that will assign the same character code to a variable.

8.3.3 Comparing Characters

The operators `==` and `!=` are used to test whether two characters are equal. These operators are straightforward and portable; that is, they work identically on all systems. The other four comparison operators, `<`, `>`, `<=`, and `>=`, also are useful for characters, but their results can vary because they depend on the local computer system.

ASCII and International ASCII are the two codes used most commonly in personal computers today, but some systems use different underlying character codes. The particular code in use on the local system determines the "alphabetical order" on that system. (The technical terms for alphabetical order are **collating sequence** and **lexical order**.) Numerals and letters of the English alphabet are arranged in the usual order in most codes, but the special symbols may be arranged in arbitrary and incompatible ways. Therefore, two dissimilar machines might produce different results for some character comparisons.

Operation	Meaning and Use
`char c1, c2;`	Declare two character variables
`c1 = c2;`	Copy the value of `c2` into `c1`.
`c1 == c2`	Do `c1` and `c2` contain the same letter?
`c1 != c2`	Do `c1` and `c2` contain different letters?
`c1 < c2`	Does `c1` come before `c2` in alphabetical order? The `<=`, `>`, and `>=` operators also are defined for characters.
`c1 + 1`	The letter that follows the value of `c1` in the alphabet.
`c1 - 1`	The letter that precedes the value of `c1` in the alphabet.
`++c2;`	Change `c2` from its current value to the next letter in the alphabet. All four increment and decrement operators are defined for characters.
`c2 - c1`	Assuming that `c2 > c1`, this is the number of letters in the alphabet between `c1` and `c2`. If `c2` is the ASCII code for a base-10 digit, then `c2 - '0'` is the numeric value of that digit.

Commonly used character operations are listed here. The phrase *alphabetical order* used here means "the order defined by the ASCII code or whatever code is in use on the local hardware." This normally is an extension of ordinary alphabetical order to include all of the characters in the local character set.

Figure 8.8. Character operations.

8.3.4 Character Arithmetic

The operators `+`, `-`, `++`, and `--` are used in **character arithmetic** to compute the next (or prior) letters of the alphabet. The operator `-` can be used to determine how far apart two letters are in the alphabet. All these operations are useful for text processing programs and all depend on the collating sequence of the machine.

8.3.5 Other Character Functions

One of the standard C libraries is the character processing library, whose header file is `ctype.h`. This library contains a group of functions essential to a system programmer, including ones to test whether a character is in a particular set, such as the alphabet, as well as certain transformation routines. Several of these functions are frequently useful, even in simple programs:

1. `isalpha()`. This function takes one argument, a character. If the character is an alphabetic character (A ... Z or a ... z), the value `true` (1) is returned; otherwise, `false` (0) is returned.
2. `islower()`. This function takes one argument, a character. If the character is a lower-case alphabetic character (a ... z), the value `true` (1) is returned; otherwise, `false` (0) is returned.

3. `isupper()`. This function takes one argument, a character. If the character is an upper-case alphabetic character (A ... Z), the value `true` (1) is returned; otherwise, `false` (0) is returned.
4. `isdigit()`. This function takes one argument, a character. If it is a digit (0 ... 9), `true` (1) is returned; otherwise, `false` (0) is returned.
5. `isspace()`. This function takes one argument, a character. If the character is a whitespace character (space, newline, return, formfeed, horizontal tab, or vertical tab), `true` is returned; otherwise, `false` is returned. We use `isspace()` to help analyze input data so that the results will be the same whether the user types spaces, newlines, or tab characters.
6. `tolower()`. This function takes one argument, a character. If the character is an upper-case alphabetic character (between A and Z), the return value is the corresponding lower-case character (between a and z). If the argument is anything else, it is returned unchanged. This function sets the case bit to 1.
7. `toupper()`. This function is the opposite of `tolower()`. If the argument is a lower-case character, the return value is the corresponding upper-case character. If the argument is anything else, it is returned unchanged. This function sets the case bit to 0.

When an input buffer might contain an unpredictable sequence of input items, a program can use the functions `isalpha()`, `islower()`, `isupper()`, `isdigit()`, and `isspace()` to analyze each input character and handle it appropriately. This makes it easier to build a well-designed user interface.

The functions `toupper()` and `tolower()` often are used in conjunction with testing character input, so that it does not matter whether the user types an upper-case or lower-case character. The programmer chooses one case to use for processing (often, it does not matter which) and converts all input characters to that case. By doing so, the logic of the program can be simplified. For example, suppose that a program needs to read a character into a variable named `ch` and use it to select one action from a set of actions. We could write the code like this:

```
scanf( " %c", &ch );

if (ch == 'x' || ch == 'X') {       /* X actions here */ }
else if (ch == 'y' || ch == 'Y') { /* Y actions here */ }
else if (ch == 'z' || ch == 'Z') { /* Z actions here */ }
else {                               /* default actions */ }
```

By using a case-shift function, we can eliminate one comparison in each test:

```
scanf( " %c", &ch );
ch = tolower( ch );

if (ch == 'x') {       /* X actions go here */  }
else if (ch == 'y') {  /* Y actions go here */  }
else if (ch == 'z') {  /* Z actions go here */  }
else {                 /* default actions go here */  }
```

8.4 Character Application: An Improved Processing Loop

This example combines the use of `scanf( " %c", ... )` with `toupper()`, `tolower()`, and a `switch` with character case-labels.

In Chapter 6, Figure 6.11, we demonstrate how a process can be repeated using a query loop until the user chooses to quit. Using the codes 1 to continue and 0 to quit is adequate, but it is not good human engineering and not customary. Now that we have shown how to read a single input character and how to force that character into a particular case, it is possible to improve the human interface by giving the more usual prompt: `Do you want to continue (y/n)?` and accepting either an upper-case or lower-case answer. The next program implements this improved interface.

The `work()` function, in Figure 8.10, is a simple application that computes the area of a regular polygon or circle. It prompts the user to select the kind of polygon from a menu and uses `tolower()` with a `switch` and character case labels to process that choice.

Notes on Figure 8.9. Improving the workmaster.

First box: the `#include` ***statements.*** In addition to `<stdio.h>`, we must include `<ctype.h>` because we are using the character-function library.

Second box: three functions. These functions perform the area computations for three different shaped figures. They are called from the `work()` function in Figure 8.10.

Third box: the `char` ***variable.*** We use a character variable rather than an integer to store the user's quit-or-continue response.

Fourth box: the repetition loop. We change the `do...while()` termination condition to test for the letter `'N'` instead of the number 0. This is more natural for the user. However, to make this test work reliably, we need to change two things in the inner box. Either response, `'N'` or `or 'n'` will cause the loop to end; any other response will permit it to continue.

Inner box: reading input. To read the input, we use a `scanf()` format that will skip whitespace in the input buffer. This is important in a loop that controls a `work()` function, because the input operations performed by that function usually leave whitespace (at least one newline character) in the input buffer.

Innermost box: case conversion. Even when instructions call for a `Y` or `N` response, many users will often type `y` or `n` instead. Good human engineering dictates that the program should accept upper-case and lower-case letters interchangeably. We achieve this by using `toupper()` to force the response into upper case. This permits us to make a simple check for an upper-case response instead of the more complex test for either a lower-case or upper-case letter. To achieve the same result without `toupper()`, we would have to write the loop test as

```
while (more != 'n' && more != 'N').
```

We improve the user interface of the main program from Figure 6.11 by permitting a `y/n` or `Y/N` response to the question, "Do you want to continue?" This program calls the `work()` function in Figure 8.10.

```
#include <stdio.h>
#include <ctype.h>    /* For toupper() and tolower(). */

void work( void );
double circle_area( double diam ) { return 3.1416 * diam * diam / 4; }
double square_area( double side ){ return side * side; }
double triangle_area( double side ) { return side * side / 4.0 * sqrt( 3 ); }

int main( void )
{
    char more;              /* repeat-or-stop switch */

    puts( "\n Calculate the area of a regular figure." );

    do { work();
         puts( "\n Do you want to continue (Y/N)? " );
         scanf( " %c", &more );
         more = toupper( more ) ;
    } while (more != 'N');
    return 0;
}
```

Figure 8.9. Improving the workmaster.

Notes on Figure 8.10. Using characters in a switch.

First box: the menu. We display a simple menu and prompt the user for a choice. When we read the input character, we use a space in the format to skip over the carriage return character left in the input stream by `main()`.

Second box: using `tolower()` ***in a*** `switch`***.*** We use `tolower()` here so that the user can enter either upper-case or lower-case choices. This line could be written thus: `switch (ch)` . However, if it were

This function is called from `main()` in Figure 8.9. Code from both Figures should be in the same file.

```
void work( void )
{
    char ch;      /* Length of one side or of the diameter */
    double x;     /* Length of one side or of the diameter */
    double area;  /* Area of the figure */

    printf (" Enter the code for the shape you wish to calculate:  \n" );
    printf (" C Circle\n S Square\n T Equilateral triangle\n > " );
    scanf( " %c", &ch );
    switch (tolower( ch ))
    {
      case 'c':
        printf( " Enter the diameter of the circle:  " );
        scanf( "%lg", &x );
        area = circle_area( x );
        printf( " The area of this circle = %.2f\n", area );
        break;
      case 's':
        printf( " Enter the length of one side of the square:  " );
        scanf( "%lg", &x );
        area = square_area( x );
        printf( " The area of this square = %.2f\n", area );
        break;
      case 't':
        printf( " Enter the length of one side of the triangle:  " );
        scanf( "%lg", &x );
        area = triangle_area( x );
        printf( " The area of this triangle = %.2f\n", area );
        break;
      default:  printf( "%c is not a meaningful choice.  Try again.", ch );
    }
    return 0;
}
```

Figure 8.10. Using characters in a switch.

written without the call on `tolower()`, the following case labels would need to be more complex.

Third box: the* `case` *label. If the call on `tolower()` were omitted in the second box, we would need to write two case labels for each case, like this: `case 'c': case 'C':`

Fourth box: the* `case` *actions. We perform all of the actions needed for a circle: input, calculation using the appropriate function, and output. Of course, a `break` statement must end the sequence. It is good style to use functions to do much or most of the work for each case, so that the entire `switch` statement fits on one computer screen.

Fourth box: the default. This case traps illegal menu choices. You can see the result in the last block of output, below.

Output. A sample of the output follows. Note that whitespace and case differences are ignored, and that the invalid response to the second query causes the process to continue, not quit.

```
Calculate the area of a regular figure.
Enter the code for the shape you wish to calculate:
 C  Circle
 S  Square
 T  Equilateral triangle
 > c
   Enter the diameter of the circle: 10
   The area of this circle = 78.54

Do you want to continue (Y/N)? :      y
Enter the code for the shape you wish to calculate:
 C  Circle
 S  Square
 T  Equilateral triangle
 > t
   Enter the length of one side of the triangle: 3.5
   The area of this triangle = 5.30

Do you want to continue (Y/N)? : t
Enter the code for the shape you wish to calculate:
 C  Circle
 S  Square
 T  Equilateral triangle
 > s
   Enter the length of one side of the square: 10
   The area of this square = 100.00

Do you want to continue (Y/N)? : y
Enter the code for the shape you wish to calculate:
 C  Circle
 S  Square
 T  Equilateral triangle
 > w
   w is not a meaningful choice.  Try again.

Do you want to continue (Y/N)? : n
```

8.5 What You Should Remember

8.5.1 Major Concepts

- ASCII is a character code. It uses 7 bits to represent the set of 128 characters that are part of the code. International ASCII is an 8-bit code that represents 256 characters. These codes are used with the `char` data type.
- Character literals are written between single quotemarks.
- Escape codes are used to write literals for invisible characters.
- There are several whitespace characters, including space, horizontal and vertical tabs, and newline.

8.5.2 Programming Style

Escape codes. Most ASCII characters are printable characters; that is, they leave a visible mark when displayed on a video screen or a printer. These characters correspond to keys on a typical computer keyboard. Some keys, such as the space bar, the Tab key, and the Enter key, do not represent printable characters but are used for their effect on the printed text. These, called *whitespace* characters, are represented in a C program by symbolic escape codes. There also are nonprintable ASCII characters that have no symbolic escape codes; they are used infrequently but may be referenced, if necessary, by using the underlying value. To be sure that your program is portable, use only the literal form of a character or a symbolic code.

Avoiding errors. Use character processing for a better human interface, like that in the revised `work()` function. Use functions like `toupper()` and `tolower()` to handle both upper-case and lower-case responses.

8.5.3 Sticky Points and Common Errors

`char` ***vs.*** `int`**.** Technically, characters are very short integers in C. Conceptually, though, they are a separate type with separate operations and different methods for input and output. Be sure not to do meaningless operations like multiplying two characters and avoid potentially nonportable operations like `<`. Every C implementation uses the character code built into the underlying hardware. For most modern machines, that code is either International ASCII or ASCII, which is given in Appendix A.

Character input. Whitespace can be a confusing factor when doing character input. The `scanf()` input conversion process for numeric types automatically skips leading whitespace and starts storing data only when a nonwhitespace character is read. However, `getchar()` returns the first character, no matter what it is; and `scanf()` with a `"%c"` does the same thing. To skip leading whitespace, you must use `scanf()` with a `" %c"` specifier (a space inside the format and before the percent sign). If this space is omitted, the program is likely to read whitespace and try to interpret it as data, which usually leads to trouble. Therefore, a programmer must have a clear idea of what he or she wishes to do (read whitespace or skip it) and choose the appropriate input mechanism for the task.

8.5.4 New and Revisited Vocabulary

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

character literal	lexical order	padding
escape code	collating sequence	character arithmetic
portability	input buffer	improved `work()` function
ASCII code	whitespace	`switch` with `char` cases

The following C keywords, functions, and symbols are discussed in this chapter:

`\n` (newline)	`getchar()`	`isalpha()`
`\r` (return)	`putchar()`	`isupper()`
`\b` (backspace)	`printf()`	`islower()`
`\t` (horizontal tab)	`scanf()`	`isspace()`
`\v` (vertical tab)	`ungetc()`	`isdigit()`
`\f` (formfeed)	`"%c"` conversion	`tolower()`
`\a` (attention)	`" %c"` conversion	`toupper()`
`\0` (null character)	`ctype.h`	

8.6 Exercises

8.6.1 Self-Test Exercises

1. Explain the difference between `'6'` and `6`.
2. What is a whitespace character? List three of them. What is an escape code character? List three of them.
3. Show the output produced by the following program. Be careful about spacing.

```
#include <stdio.h>
int main( void )
{
    int k;
    for (k = 1; k <= 5; k += 2) {
        printf( "    %i:", k );
        putchar( '0'+k );
    }
    putchar( '\n' );
}
```

4. What will be stored in `k`, `c`, or `b` by the following sets of assignments? Use these variables:

```
int k;    char c, d;   int b;
```

(a) `d = 'b';   c = d+1;`

(b) `d = 'b';  c = d--;`
(c) `d = 'E';  c = toupper( d );`
(d) `d = '7';  k = d - '0';`
(e) `b = isalpha( '@' );`
(f) `b = 'A' == 'a';`

5. What will be stored in each of the variables by the input statements on the right, given the line of input on the left. If the combination is an error, say so. Use these variables:

```
int k;   char d;
```

(a) `a` `scanf( "%c", &d);`
(b) `66` `scanf( "%c", &d);`
(c) `70 C` `scanf( "%i%c", &k, &d);`
(d) `F` `scanf( "%i", &k);`
(e) `go!` `d = getchar();`
(f) `\n` `scanf( "%c", &d);`

6. What is the output from the following program if the user enters `Z` after the input prompt?

```
#include <stdio.h>
int main( void )
{
    char ch;
    printf( "\n Type a character and hit ENTER: " );
    ch = getchar();
    printf( "%3i %c \n ", ch, ch );
    putchar( ch ); putchar( '\n' );
}
```

8.6.2 Using Pencil and Paper

1. What will be stored in `k`, `c`, or `b` by the following sets of assignments? Use these variables:

```
int b, k;    char c, d;
```

(a) `d ='A';  k = d;`
(b) `d ='c';  c = toupper( d );`
(c) `d ='@';  c = tolower( d );`
(d) `k = 66;  c = k-1;`
(e) `b = isupper( 'A' );`

(f) `b = 'A' < 'a';`

2. What will be stored in each of the variables by the input statements on the right, given the line of input on the left. If the combination is an error, say so. Use these variables:

```
int k, m;  char c, d;
```

(a) `a` `scanf( "%c", &k);`
(b) `126` `scanf( "%c", &d);`
(c) `70 D` `scanf( "%i %c", &k, &d);`
(d) `70 71` `scanf( "%i%i", &k, &m);`
(e) `U2` `scanf( "%c%i", &d, &k);`
(f) `I O` `c = getchar();  d = getchar();`

3. Without running the following program, show what the output will be. Use your ASCII table.

```
#include <stdio.h>

int main( void )
{
    int upper = 65;
    int lower = upper + 32;
    int limit = 26;
    int step = 0;

    puts( " Do you read me?" );
    while ( step < limit ) {
        printf( "%2i.   %c  %c\n", step, upper, lower );
        ++upper;
        ++lower;
        ++step;
    }
    printf( "=========\n" );
}
```

4. What is the output from the following program if the user enters 80 after the input prompt?

```
#include <stdio.h>
int main( void )
{
   int k;
   printf( "\n Enter a number 65 ... 126: " );
   scanf ( "%i", &k );
   printf( " %3i %c \n", k, k );
   putchar( k ); putchar( '\n' );
}
```

5. Write a code fragment to compare two character variables and print `true` if both are alphabetic and they are the same letter, except for possible case differences. Print `false` otherwise.

8.6.3 Using the Computer

1. Character manipulation practice.

   Write a program to prompt the user once for a series of characters. Read the characters one at a time using `scanf( " %c", ...)` in a loop. Generate the following output, based on the input, one response per line:

   (a) Echo the input character.
   (b) Call `isalpha()` to find out whether it is alphabetic.
   (c) If so, call `toupper()` to convert it to an upper-case letter and print the result. If not, print an error comment.
   (d) If the character is a period, print a statement to that effect and quit.

2. Volumes.

   Write a program to calculate volumes. Start by writing three double→double functions for these three geometric shapes:

   (a) `cylinder()`. The single argument, d, is the height of a cylinder and also the diameter of its circular base. Calculate and return its volume. The formula is: $volume = \pi \times d^3/4$
   (b) `cube()`. The argument is the length, s, of one side of a cubical box. Calculate and return its volume. The formula is: $volume = s^3$
   (c) `sphere()`. The argument is the diameter, d, of a sphere. Calculate and return its volume. The formula is: $volume = (4/3) \times \pi \times d^3/8$

   Write a program that will permit the user to compute the area of several shapes. Use a menu and a switch to process the menu selections. Include a menu item "Q Quit", and use this instead of a uery loop to end the program.

   Read the letter in such a way that whitespace does not matter. Test the input in such a way that upper-case and lower-case differences do not matter. If the letter is `q`, terminate the program. Otherwise, read and validate a real number that represents the size of the figure. If this length is 0 or negative, print an error comment. If it is positive, call the appropriate area function and print the answer it returns. If the letter entered is not `c`, `s`, `t`, or `q`, print an appropriate error message.

3. Palindromes.

   This program will test whether a sentence is a palindrome; that is, whether it has the same letters when read forward and backward. First, prompt the user to enter a sentence. Read the characters one at a time using `getchar()` until a period appears. As they are read,

   (a) Echo the input character.
   (b) Call `tolower()` to convert each character to lower case.

(c) Count the number of characters read (excluding the period).

(d) Store the converted character in the next available slot in an array.

When a period appears, start from both ends of the array and compare the letters. Compare the first to the last, the second to the second-last, and so forth. If any pair fails to match, leave the loop and announce that the sentence is not a palindrome. If you get to the middle, stop, and announce that the input is a palindrome. Assume that the input will be no more than 80 characters long.

4. Ascending or descending.

   Your program should read three numbers and a letter. If the letter is , 'A' or 'a', output the numbers in ascending order. If it is 'D or 'd', output the numbers in descending order. For any other letter, give an error message and output them in the opposite order that they were read in..

5. Vowels.

   Prompt the user to enter a sentence, then hit newline. Read the sentence one character at a time, until you come to the newline character. Count the total number of keystrokes entered, the number of alphabetic characters, and the number of vowels (`'a', 'e', 'i', 'o',` and `'u'`). Output these three counts.

6. Tooth fairy time.

   This program will "pronounce" an ordinary sentence with a lisp. Prompt the user to enter a sentence. Read the characters, one at a time, until a period appears. As they are read, convert everything to lower case and test for occurrences of the character `'s'` and the pair `"ss"`. Replace each `'s'` or `"ss"` by the letters `'t'` and `'h'`. Print the converted message using `putchar()`. For example, given the sentence `I see his house`, the output would be `I thee hith houthe`.

7. Building numbers.

   Write a program that will use a work function to input and process several data sets. Each data set will consist of three input characters if they are all valid base-10 digits, you will convert them to an integer and print it. Otherwise, print an error comment.

   - Use `isdigit()` to test for valid inputs.
   - Convert the ASCII code for a digit to its corresponding numeric value by subtracting the character `'0'`. For example, if `ch` contains the character `'7'`, then `ch-'0'` is the integer value 7.
   - If the numeric values of your inputs are stored in the variables a, b, and c, then the answer is $a * 100 + b * 10 + c$

8. A tall story.

   Develop a program for a baker who makes wedding cakes. These cakes have multiple layers, and the layers have different shapes. The top layer always is circular, with a diameter of 6 inches. The next layer down is square, each side 7.5 inches long. The third layer would be circular again, with a diameter of 8 inches; and the fourth layer is a 9.5-inch square. The shapes continue to alternate in this pattern and get bigger until the bottom layer is reached. In addition, each layer is 2 inches thick, so the area of its side is 2" $\times$ the perimeter of the layer. The baker wants to know how much frosting to make for the cake to frost the entire top and side of every layer. Write a program that will read in the number of layers

desired for a cake, and then print the total square inches of cake to be covered with frosting. Break up your program as follows:

(a) Write a function called `surface_area()`. This function has two parameters. One is a character with the value `C` for circle or `S` for square. The other is an integer that represents either the diameter for a circle or the length of a side of a square. This function will compute the sum of the top area and the side area of either a circular or a square layer, depending on the character value.

(b) Write a main program that first will read in the number of layers of the cake. Then it will call the `surface_area()` function for each layer of the cake and total the areas. Finally, print the total.

9. Temperatures.

   Write a program and three functions that will convert temperatures from Fahrenheit to Celsius or vice versa. The main program should implement a work loop and use the improved interface of Figure 8.9. The `work()` function should prompt the user to enter a temperature, which consists of a number followed by optional whitespace and a letter (`F` or `f` for Fahrenheit, `C` or `c` for Celsius). Appropriate inputs might be `125.2F` and `-72 c`. Read the number and the letter, test the letter, and call the appropriate conversion function, described here. Test the converted answer that is returned and print an error comment if the return value is -500. Otherwise, echo the input temperature and print the answer with the correct unit, `F` or `C`. (Note that there is no difficulty reading an input in the form `125F`; `scanf()` stops reading the digits of the number when it gets to the letter and the letter then can be read by a `%c` specifier.)

   Write two functions: `Fahr_2_Cels()` converts a Fahrenheit argument to Celsius and returns the converted temperature; `Cels_2_Fahr()` converts a Celsius argument to Fahrenheit and returns it. Both functions must test the input to detect temperatures below absolute 0 (-273.15°C and -459.67°F). If an input is out of range, each function should return the special value -500.

10. Try it, I dare you!

    Write a program that will display the ASCII code in a table that looks like the one in Appendix . Be sure not to try to print the unprintable characters directly. Omit the column that lists the hexadecimal codes.