Chapter 4

Processing Character Data

So far we have considered only numeric processing, i.e. processing of numeric data represented as integer and floating point types. Humans also use computers to manipulate data that is not numeric such as the symbols used to represent alphabetic letters, digits, punctuation marks, etc. These symbols have a standard meaning to us, and we use them to represent (English) text. In the computer, the symbols used to store and process text are called **characters** and C provides a data type, `char`, for these objects. In addition, communication between humans and computers is in the form of character symbols; i.e. all data typed at a keyboard and written on a screen is a sequence of character symbols. The functions `scanf()` and `printf()` perform the tasks of converting between the internal form that the machine understands and the external form that humans understand.

In this chapter, we will discuss character processing showing how characters are represented in computers and the operations provided to manipulate character data. We will develop programs to process text to change it from lower case to upper case, separate text into individual words, count words and lines in text, and so forth. In the process, we will present several new control constructs of the C language, describe user interfaces in programs, and discuss input/output of character data.

4.1 A New Data Type: `char`

The complete set of characters that can be recognized by the computer is called the **character set** of the machine. As with numbers, the representation in the computer of each character in the set is done by assigning a unique bit pattern to each character. The typical character set consists of the following types of characters:

```
Alphabetic lower case: 'a',..., 'z'
Alphabetic upper case: 'A',..., 'Z'
Digit symbols        : '0',..., '9'
Punctuation          : '.', ',', ';', etc.
```

Character	Meaning
'\a'	alert (bell)
'\b'	backspace
'\f'	form feed
'\n'	newline
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\\'	backslash
'\''	single quote
'\"'	double quote
'\?'	question mark

Table 4.1: Escape Sequences

```
Space                : ' '
Special symbols      : '@', '#', '$', etc.
Control Characters   : newline, tab, bell or beep, etc.
```

For example, a digit symbol is character type data, so when we type *234* at the keyboard, we are typing a sequence of character symbols: `'2'`, followed by `'3'`, followed by `'4'`. The function `scanf()` takes this sequence and converts it to the internal form of the equivalent number, 234. Similarly, all writing on the screen is a sequence of characters so `printf()` takes the internal form of the number and converts it to a sequence of characters which are written onto the screen.

In C programs, variables may be declared to hold a single character data item by using the keyword `char` as the type specifier in the declaration statment:

```
char ch;
```

A character constant is written surrounded by single quotation marks, e.g. `'a'`, `'A'`, `'$'`, `'!'`, etc. Only *printable character* constants can be written in single quotes, not control characters, so writing of non-printable control character constants requires special handling. In C, the backslash character, \, is used as an *escape character* which signifies something special or different from the ordinary and is followed by one character to indicate the particular control character. We have already seen one such *control sequence* in our `printf()` statments; the newline character, `'\n'`. Other frequently used control character constants written with an escape sequence, include `'\t'` for tab, `'\a'` for bell, etc. Table 4.1 shows the escape sequences used in C. The newline, tab, and space characters are called **white space** characters, for obvious reasons.

Let us consider a simple task of reading characters typed at the keyboard and writing them to the screen. The task is to copy (or echo) the characters from the input to the output. We will continue this task until there is no more input, i.e. until the end of the input file.

```
/*   File: copy0.c
     Programmer:
     Date:
     This program reads a stream of characters, one character at
     a time, and echoes each to the output until EOF.
*/

#include <stdio.h>
main()
{    char ch;       /* declaration for a character object ch */
     int flag;         /* flag stores the number of items read by scanf() */

     printf("***Copy Program***\n\n");
     printf("Type text, terminate with EOF\n");
     flag = scanf("%c", &ch);         /* read the first char */
     while (flag != EOF) {            /* repeat while not EOF */
          printf("%c", ch);           /* print the last char read */
          flag = scanf("%c", &ch);    /* read the next char, update flag */
     }                                /* flag is EOF, ch may be unchanged */
}
```

Figure 4.1: Code for copy0.c

TASK

COPY0: Write out each character as it is read until the end of input file.

The algorithm can be stated simply as:

```
read the first character
while there are more characters to read
     write or print the previously read character;
     read the next character
```

The code for this program is shown in Figure 4.1.

The keyword `char` declares a variable, `ch`, of character data type. We also declare an integer variable, `flag`, to save the value returned by `scanf()`. Recall, the value returned is either the number of items read by `scanf()` or the value `EOF` defined in `stdio.h`. (We do not need to know the actual value of `EOF` to use it).

After the title is printed, a character is read by the statement:

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

The conversion specification for character type data is `%c`, so this `scanf()` reads a single character from the input. If it is not an end of file keystroke, the character read is stored into `ch`, and the value returned by `scanf()`, 1, is saved in `flag`. As long as the value of `flag` is not `EOF`, the loop is entered. The loop body first prints the value of `ch`, i.e. the last character read, and then, the assignment statement reads a new character and updates `flag`. The loop terminates when `flag` is `EOF`, i.e. when an end of file keystroke is detected. Remember, `scanf()` does not store the value, `EOF` into the object, `ch`. DO NOT TEST THE VALUE OF `ch` FOR `EOF`, TEST `flag`. A sample session is shown below:

```
***Copy Program***

Type text, terminate with EOF
Now is the time for all good men
Now is the time for all good men
To come to the aid of their country.
To come to the aid of their country.
^D
```

The sample session shows that as entire lines of characters are entered; they are printed. Each character typed is not immediately printed, since no input is received by the program until a newline character is typed by the user; i.e. the same buffering we saw for numeric data entry. When a newline is typed, the entire sequence of characters, including the newline, is placed in the keyboard buffer and `scanf()` then reads input from the buffer, one character at a time, up to and including the newline. In our loop, each character read is then printed. When the buffer is exhausted, the next line is placed in the buffer and read, and so on. So, `scanf()` is behaving just as it did for numeric data; each call reads one data item, in this case a character (`%c`). One notable difference between reading numeric data and character data is that when `scanf()` reads a character, leading white space characters are read, one character at a time, not skipped over as it is when reading numeric data.

4.1.1 The ASCII Character Set

Character data is represented in a computer by using standardized numeric codes which have been developed. The most widely accepted code is called the **American Standard Code for Information Interchange** (**ASCII**). The ASCII code associates an integer value for each symbol in the character set, such as letters, digits, punctuation marks, special characters, and control characters. Some implementations use other codes for representing characters, but we will use ASCII since it is the most widely used. The ASCII characters and their decimal code values are shown in Table 4.2. Of course, the internal machine representation of characters is in equivalent binary form.

ASCII value	Character	ASCII value	Character	ASCII value	Character
000	^@	043	+	086	V
001	^A	044	,	087	W
002	^B	045	-	088	X
003	^C	046	.	089	Y
004	^D	047	/	090	Z
005	^E	048	0	091	[
006	^F	049	1	092	\
007	^G	050	2	093	]
008	^H	051	3	094	^
009	^I	052	4	095	_
010	^J	053	5	096	‘
011	^K	054	6	097	a
012	^L	055	7	098	b
013	^M	056	8	099	c
014	^N	057	9	100	d
015	^O	158	:	101	e
016	^P	059	;	102	f
017	^Q	060	<	103	g
018	^R	061	=	104	h
019	^S	062	>	105	i
020	^T	063	?	106	j
021	^U	064	@	107	k
022	^V	065	A	108	l
023	^W	066	B	109	m
024	^X	067	C	110	n
025	^Y	068	D	111	o
026	^Z	069	E	112	p
027	^[	070	F	113	q
028	^\	071	G	114	r
029	^]	072	H	115	s
030	^^	073	I	116	t
031	^-	074	J	117	u
032	[space]	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	’	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

Table 4.2: ASCII Table

The ASCII table has 128 characters, with values from 0 through 127. Thus, 7 bits are sufficient to represent a character in ASCII; however, most computers typically reserve 1 byte, (8 bits), for an ASCII character. One byte allows a numeric range from 0 through 255 which leaves room for growth in the size of the character set, or for a sign bit. Consequently, a character data type may optionally represent signed values; however, for now, we will assume that character data types are unsigned, i.e. positive integer values, in the range 0—127.

Looking at the table, note that the decimal values 0 through 31, and 127, represent non-printable control characters. All other characters can be printed by the computer, i.e. displayed on the screen or printed on printers, and are called **printable characters**. All printable characters and many control characters can be input to the computer by typing the corresponding keys on the keyboard. The *character* column shows the key(s) that must be pressed. Only a single key is pressed for a printable character; however, control characters need either special keys on the keyboard or require the CTRL key pressed together with another key. In the table, a control key is shown by the symbol `^`. Thus, `^A` is control-A, i.e. the CTRL key kept pressed while pressing the key, A.

Notice that the character `'A'` has the code value of 65, `'B'` has the value 66, and so on. The important feature is the fact that the ASCII values of letters `'A'` through `'Z'` are in a contiguous increasing numeric sequence. The values of the lower case letters `'a'` through `'z'` are also in a contiguous increasing sequence starting at the code value 97. Similarly, the digit symbol characters `'0'` through `'9'` are also in an increasing contiguous sequence starting at the code value 48. As we shall see, this feature of the ASCII code is quite useful.

It must be emphasized that a digit symbol is a *character* type. Digit characters have code values that differ from their numeric equivalents: the code value of `'0'` is 48, that of `'1'` is 49, that of `'2'` is 50, and so forth. The table shows that the character with code value 0 is a control character, `^@`, called the **NULL character**. Do NOT confuse it with the digit symbol `'0'`. Remember, a digit *character* and the equivalent *number* have different representations.

Besides using single quotes, it is also possible to write character constants in terms of their ASCII values in a C program, using either their octal or their hexadecimal ASCII values. In writing character constants, the octal or hexadecimal value follows the escape character, \, as shown in Table 4.3. At most three octal digits or at most two hexadecimal digits are needed. Note, after the escape backslash, a leading zero should not be included in writing octal or hexadecimal numbers. The last example in Table 4.3, `'\0'`, is called the **NULL character**, whose ASCII value is zero. Once again, this is NOT the same character as the printable digit character, `'0'`, whose ASCII value is 48.

4.1.2 Operations on Characters

As we just saw, in C, characters have numeric values and, therefore, may be used in numeric expressions. It is the ASCII code value of a character that is used in these expressions. For example (referring to Table 4.2), the value of `'a'` is 97, and that of `'A'` is 65. So, the expression `'a' - 'A'` is evaluated as $97 - 65$, which is 32. As we shall see, this ability to do arithmetic with

Character Constants	Meaning
'\007', '\07', '\7'	character whose value is octal 7
'\101'	character whose octal value is 101, or whose decimal value is 65, i.e. 'A'
'\xB'	character with hex. value B, i.e. with decimal value 11.
'\0'	character whose value is zero; it is called the NULL character

Table 4.3: Escape sequences with Octal & Hexadecimal values

character data simplifies character processing. When a character variable or constant appears in an expression, it is replaced by its ASCII value of type integer. When a character cell is assigned an integer value, the value is interpreted to be an ASCII value. In other words, a character and its ASCII value are used interchangeably as required by the context. While a cast operator can be used, we do not need it to go from character type to integer type, and vice versa. Here are some expressions using character variables and constants.

```
ch = 97;                /* ch <--- ASCII value 97, i.e., 'a' */
ch = '\141';            /* ch <--- 'a'; octal 141 is decimal 97 */
ch = '\x61';            /* ch <--- 'a'; hexadecimal 61 is decimal 97 */
ch = 'a';               /* ch <--- 'a' */

ch = ch - 'a' + 'A';    /* ch <--- 'A' */

ch = 'd';
ch = ch - 'a' + 'A';    /* ch <--- 'D' */
ch = ch - 'A' + 'a';    /* ch <--- 'd' */
```

The first group of four statements merely assigns lower case 'a' to ch in four different ways: the first assigns a decimal ASCII value, the second assigns a character in octal form, the third assigns a character in hexadecimal form, the fourth assigns a character in a printable symbolic form. All of these statements have exactly the same effect.

The next statement, after the first group, assigns the value of an expression to ch. The right hand side of the assignment is:

```
ch - 'a' + 'A'
```

Since the value of ch is 'a' from the previous four statements, the above expression evaluates to the value of 'a' - 'a' + 'A', i.e. the value of 'A'. In other words, the right hand side expression converts lower case 'a' to its upper case version, 'A', which is then assigned to ch. Since the values of lower case letters are contiguous and increasing (as are those of upper case letters) 'a' is less than 'b', 'b' less than 'c', and so forth. Also, the offset value of each letter from the base of the alphabet is the same for lower case letters as it is for upper case letters. For example, 'd' - 'a' is the same as 'D' - 'A'. So, if ch is any lower case letter, then the expression

```
ch - 'a' + 'A'
```

results in the upper case version of ch. This is because the value of ch - 'a' is the offset of ch from the lower case base 'a'; adding that value to the upper case base 'A' results in the upper case version of ch. So for example, if ch is 'f' then the value of the above expression is 'F'. Similarly, if ch is an upper case letter, then the expression

```
ch - 'A' + 'a'
```

results in the lower case version of ch which may then be assigned to a variable.

Using this fact, the last group of three statements in the above set of statements first assigns a lower case letter 'd' to ch. Then the lower case value of ch is converted to its upper case version, and then back to lower case.

As we mentioned, all lower case and upper case letters have contiguous and increasing values. The same is true for digit characters. Such a contiguous ordering makes it easy to test if a given character, ch, is a lower case letter, an upper case letter, or a digit. For example, any lower case letter has a value that is greater than or equal that of 'a' AND less than or equal to that of 'z'. From this, we can write a C expression that is True if and only if ch is a lower case letter:

```
(ch >= 'a' && ch <= 'z')
```

Here is a code fragment that checks whether a character is a lower case letter, an upper case letter, a digit, etc.

```
if (ch >= 'a' && ch <= 'z')
     printf("%c is a lower case letter\n", ch);
else if (ch >= 'A' && ch <= 'Z')
     printf("%c is an upper case letter\n", ch);
else if (ch >= '0' && ch <= '9')
     printf("%c is a digit symbol\n", ch);
else
     printf("%c is neither a letter nor a digit\n");
```

Observe the multiway decision and branch: `if ... else if ... else if ... else`. Only one of the branches is executed. The first `if` expression checks if the value of `ch` is between the values of `'a'` and `'z'`, a lower case letter. Only if `ch` is not a lower case letter, does control proceed to the first `else if` part, which tests if `ch` is an upper case letter. Only if `ch` is not an upper case letter, does control proceed to the next `else if` part, which tests if `ch` is a digit. Finally, if `ch` is not a digit, the last `else` part is executed. Depending on the value of `ch`, only one of the paths is executed with its corresponding `printf()` statement.

Let us see how the expression

```
(ch >= 'a' && ch <= 'z')
```

is evaluated. First, the comparison `ch >= 'a'` is performed; then, `ch <= 'z'` is evaluated; finally, the results of the two sub-expressions are logically combined by the AND operator. Evaluation takes place in this order because the precedence of the binary relational operators (>=, <=, `==`, etc.) is higher than that of the binary logical operators (`&&`, ||). We could have used parentheses for clarity, but the precedence rules ensure the expression is evaluated as desired.

One very common error is to write the above expression analogous to mathematical expressions:

```
('a' <= ch <= 'z')
```

This would not be found to be an error by the compiler, but the effect will not be as expected. In the above expression, since the precedence of the operators is the same, they will be evaluated from left to right according to their associativity. The result of `'a' <= ch` will be either True or False, i.e. 1 or 0, which will then be compared with `'z'`. The result will be True since 1 or 0 is always less than `'z'` (ASCII value 122). So the value of the above expression will always be True regardless of the value of `ch` — not what we would expect.

Let's write a program using all this information. Our next task is to read characters until end of file and to print each one with its ASCII value and what we will call the *attributes* of the character. The attributes are a character's category, such as a lower case or an upper case letter, a digit, a punctuation, a control character, or a special symbol.

Task

ATTR: For each character input, print out its category and ASCII value in decimal, octal, and hexadecimal forms.

The algorithm requires a multiway decision for each character read. A character can only be in one category, so each character read will lead to the execution of one of the paths in a multiway decision. Here is the algorithm.

```
read the first character
```

```
repeat as long as end of file is not reached
     if the character is a lower case letter
          print the various character representations, and
          print that it is a lower case letter
     else if it is an upper case letter
          print the various character representations, and
          print that it is an upper case letter
     else if it is a digit
          print the various character representations, and
          print that it is a digit
          etc..
     read the next character
```

Notice we have abstracted the printing of the various representations of the character (as a character and its ASCII value in decimal, octal and hex) into a single step in the algorithm: `print the various character representations`, and we perform the same step in every branch of the algorithm. This is a classic situation calling for the use of a function: abstract the details of an operation and use that abstraction in multiple places. The code implementing the above algorithm is shown in Figure 4.2. We have *declared* a function `print_reps()` which is passed a single character argument and expect it to print the various representations of the character. We have used the function in the driver without knowing how `print_reps()` will perform its task.

We must now write the function `print_reps()`. The character's value is its ASCII value. When the character value is printed as a character with conversion specification `%c`, the symbol is printed; when printed as a decimal integer with conversion specification `%d`, the ASCII value is printed in decimal form. Conversion specification `%o` prints an integer value in octal form, and `%x` prints an integer value in hexadecimal form. We simply need a `printf()` call with these four conversion specifiers to print the character four times. The code for `print_reps()` is shown in Figure 4.3. The function simply prints its parameter as a character, a decimal integer, an octal integer, and a hexadecimal integer.

Sample Session:

```
***Character Attributes***

Type text, terminate with EOF
Aloha, ^A!
A, ASCII value decimal 65, octal 101, hexadecimal 41:  an upper case letter
l, ASCII value decimal 108, octal 154, hexadecimal 6c:  a lower case letter
o, ASCII value decimal 111, octal 157, hexadecimal 6f:  a lower case letter
h, ASCII value decimal 104, octal 150, hexadecimal 68:  a lower case letter
a, ASCII value decimal 97, octal 141, hexadecimal 61:  a lower case letter
,, ASCII value decimal 44, octal 54, hexadecimal 2c:  a punctuation symbol
^A, ASCII value decimal 1, octal 1, hexadecimal 1:  a control character
!, ASCII value decimal 33, octal 41, hexadecimal 21:  a punctuation symbol
```

```
/*   File: attr.c
     This program reads characters until end of file. It prints the
     attributes of each character including the ASCII value.
*/
#include <stdio.h>
int print_reps( char ch );

main()
{    char ch;
     int flag;

     printf("***Character Attributes***\n\n");
     printf("Type text, terminate with EOF \n");
     flag = scanf("%c", &ch);         /* read the first char */
     while (flag != EOF) {
          if (ch >= 'a' && ch <= 'z') {      /* lower case letter? */
               print_reps(ch);
               printf("lower case letter\n");
          }
          else if (ch >= 'A' && ch <= 'Z') {      /* upper case letter? */
               print_reps(ch);
               printf("an upper case letter\n");
          }
          else if (ch >= '0' && ch <= '9') { /* digit character? */
               print_reps(ch);
               printf("a digit symbol\n");
          }
          else if (ch == '.' || ch == ',' || ch == ';' || ch == ':' ||
                    ch == '?' || ch == '!') {     /* punctuation? */
               print_reps(ch);
               printf("a punctuation symbol\n");
          }
          else if (ch == ' ') {                  /* space? */
               print_reps(ch);
               printf("a space character\n");
          }
          else if (ch < 32 || ch == 127) {        /* control character? */
               print_reps(ch);
               printf("a control character\n");
          }
          else {                              /* must be a special symbol */
               print_reps(ch);
               printf("a special symbol\n");
          }
          flag = scanf("%c", &ch);         /* read the next char */
     }    /* end of while loop */
}         /* end of program */
```

Figure 4.2: Code for ASCII Attributes

```
/*   File: attr.c  --- continued
*/

int print_reps( char ch)
{
     printf("%c, ASCII value decimal %d, octal %o, hexadecimal %x: ",
                    ch,ch,ch,ch);
}
```

Figure 4.3: Printing character representations

```
, ASCII value decimal 10, octal 12, hexadecimal a:  a control character
^D
```

The last line printed refers to the newline character. Remember, every character including the newline is placed in the keyboard buffer for reading and, while `scanf()` skips over leading white space when reading a numeric data item, it does not do so when reading a character.

Can we improve this program? The driver (`main()`) shows all the details of character testing, beyond the logic of what is being performed here, so it may not be very readable. Perhaps we should define a set of macros to hide the details of the character testing expressions. For example, we might write a macro:

```
#define IS_LOWER(ch)     ((ch) >= 'a' && (ch) <= 'z')
```

Then the first if test in `main()` would be coded as:

```
if ( IS_LOWER(ch) )  {
...
```

which directly expresses the logic of the program. The remaining expressions can be recoded using macros similarly and this is left as an exercise at the end of the chapter.

One other thought may occur to us to further improve the program. Can we make the function `print_reps()` a little more abstract and have it print the various representations as well as the category? To do this we would have to give additional information to our new function, which we will call `print_category()`. We need to tell `print_category()` the character to print as well as its category. To pass the category, we assign a unique code to each category and pass the appropriate code value to `print_category()`. To avoid using "magic numbers" we define the following macros:

```
#define  LOWER     0
#define  UPPER     1
```

```
#define  DIGIT     2
#define  PUNCT     3
#define  SPACE     4
#define  CONTROL   5
#define  SPECIAL   6
```

Placing these defines (together with the comparison macros) in a header file, `category.h`, we can now recode the program as shown in Figure 4.4. The code for `print_category()` is also shown. Looking at this code, it may seem inefficient in that we are testing the category twice; once in `main()` using the character, and again in `print_category()` using the encoded parameter. Later in this chapter we will see another way to code the test in `print_category()` which is more efficient and even more readable. The contents of the header file, `category.h` is left as an exercise. The program shown in Figure 4.4 will behave exactly the same as as the code in Figure 4.2 producing the same sample session shown earlier.

4.1.3 Character I/O Using `getchar()` and `putchar()`

We have already seen how to read and print characters using our usual I/O built in functions, `scanf()` and `printf()`; i.e. the `%c` conversion specifier. We have also included the header file `stdio.h` in all our programs, because it contains the definition for `EOF`, and declares prototypes for these *formatted* I/O routines. In addition, `stdio.h` contains two other useful *routines*, `getchar()` and `putchar()`, which are simpler to use than the formatted routines for character I/O. We use the term *routine* for `getchar()` and `putchar()` because they are actually macros defined in `stdio.h` which use more general functions available in the standard library. (Often routines that are macros are loosely referred to as functions since their use in a program can appear like a function call, so we will usually refer to `getchar()` and `putchar()` as functions).

The function `getchar()` reads a single character from the standard input and returns the character value as the value of the function, but to accommodate a possible negative value for `EOF`, the type of the value returned is `int`. (Recall, `EOF` may be either 0 or -1 depending on implementation). So we could use `getchar()` to read a character and assign the returned value to an integer variable:

```
int c;

c = getchar();
```

If, after executing this statement, `c` equals `EOF`, we have reached the end of the input file; otherwise, `c` is the ASCII value of the next character in the input stream.

While `int` type can be used to store the ASCII value of a character, programs can become confusing to read — we expect that the `int` data type is used for numeric integer data and that `char` data type is used for character data. The problem is that `char` type, depending on implementation, may or may not allow negative values. To resolve this, C allows us to explicitly

```
/*   File: attr2.c
     This program reads characters until end of file. It prints the
     attributes of each character including the ASCII value.
*/
#include <stdio.h>
#include "category.h"

main()
{    char ch;
     int flag;

     printf("***Character Attributes***\n\n");
     printf("Type text, terminate with EOF \n");
     flag = scanf("%c", &ch);         /* read the first char */

     while (flag != EOF) {
          if( IS_LOWER(ch) ) print_category(LOWER, ch);
          else if( IS_UPPER(ch) ) print_category(UPPER, ch);
          else if( IS_DIGIT(ch) ) print_category(DIGIT, ch);
          else if( IS_PUNCT(ch) ) print_category(PUNCT, ch);
          else if( IS_SPACE(ch) ) print_category(SPACE, ch);
          else if( IS_CONTROL(ch) ) print_category(CONTROL, ch);
          else print_category(SPECIAL, ch);

          flag = scanf("%c", &ch);         /* read the next char */
     }    /* end of while loop */
}         /* end of program */

int print_category( int cat, char ch)
{
     printf("%c, ASCII value decimal %d, octal %o, hexadecimal %x: ",
                    ch,ch,ch,ch);
     if(      cat == LOWER )   printf("lower case letter\n");
     else if( cat == UPPER )   printf("an upper case letter\n");
     else if( cat == DIGIT )   printf("a digit symbol\n");
     else if( cat == PUNCT )   printf("a punctuation symbol\n");
     else if( cat == SPACE )   printf("a space character\n");
     else if( cat == CONTROL ) printf("a control character\n");
     else printf("a special symbol\n");
}
```

Figure 4.4: Alternate code for attributes program

declare a `signed char` data type for a variable, which can store negative values as well as positive ASCII values:

```
signed char c;

c = getchar();
```

An explicit `signed char` variable ensures that a character is stored in a character type object while allowing a possible negative value for `EOF`. The keyword `signed` is called a **type qualifier**.

A similar routine for character output is `putchar()`, which outputs its argument as a character to the standard output. Thus,

```
putchar(c);
```

outputs the ASCII character whose value is in `c` to the standard output. The argument of `putchar()` is expected to be an integer; however, the variable `c` may be either `char` type or `int` type (ASCII value) since the value of a `char` type is really an integer ASCII value.

Since both `getchar()` and `putchar()` are macros defined in `stdio.h`, any program that uses these functions must include the `stdio.h` header file in the program. Let us rewrite our copy program using these new character I/O routines instead of using `scanf()` and `printf()`. The new code is shown in Figure 4.5. Characters are read until `getchar()` returns `EOF`. Each character read is printed using `putchar()`. Sample output is shown below.

```
***File Copy Program***

Type text, EOF to quit
This is a test.
This is a test.
Now is the time for all good men
Now is the time for all good men
to come to the aid of their country.
to come to the aid of their country.
^D
```

The sample output shown here is for keyboard input so the effects of buffering the input is clearly seen: a line must be typed and entered before the characters become available in the input buffer for access by the program and then echoed to the screen.

Using `getchar()` and `putchar()` are simpler for character I/O because they do not require a format string as do `scanf()` and `printf()`. Also, `scanf()` stores a data item in an object whose address is given by its argument, whereas `getchar()` returns the value of the character read as its value. Both `scanf()` and `getchar()` return `EOF` as their value when they read an end of file marker in an input file.

```
/* File: copychr.c
     Program copies standard input to standard output.
*/
#include <stdio.h>

main()
{    signed char c;


     printf("***File Copy Program***\n\n");
     printf("Type text, EOF to quit\n");
     c = getchar();

     while (c != EOF) {
          putchar(c);
          c = getchar();
     }
}
```

Figure 4.5: Using `getchar()` and `putchar()`

4.1.4 Strings vs Characters

Frequently, we have needed to write constants that are not single characters but are sequences of characters. A sequence of zero or more characters is called a **string of characters** or simply a **string**. We have already used strings as arguments in function calls to `printf()` and `scanf()`. In C, there is no primitive data type for strings; however, as a convenience, string constants (also called **string literals**) may be written directly into a program using double quotes. The double quotes are not part of a string constant; they are merely used to delimit (define the limits,) of the string constant. (To include a double quote as part of a string, escape the double quote with the \ character).

```
"This is a string constant."
"This string constant includes newline character.\n"
"This string constant includes \" double quotes."
```

Escape sequences may of course be included in string constants. A string constant may even contain zero characters, i.e. an empty string:

```
""
```

Such a string is also called a **null string**.

Two adjacent strings are concatenated at compile time. Thus,

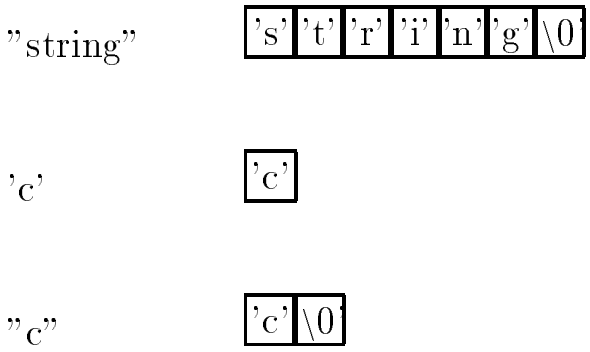

Figure 4.6: Strings

```
"John " "Doe"
```

are equivalent to:

```
"John Doe"
```

Whenever a string constant appears in a source program, the compiler stores the sequence of characters in contiguous memory locations and appends a `NULL` character to indicate the end of the string (see Figure 4.6). The compiler then replaces the string constant by the address where the characters are stored. Observe that a string of a single character is different from a character constant. Thus, `'c'` is a character constant; but, `"c"` is a string constant consisting of one character and the `NULL` character, as seen in the figure.

As we have said, a character constant takes on its ASCII value. The value of a string constant is the address where the string is stored. How this value can be used will be discussed in Chapter 6. For now, think of a string constant as a convenient representation, the exact nature of which will become clear later.

4.2 Sample Character Processing Functions

So far we have merely read and printed characters and determined their attributes. Character processing requires manipulation of input characters in meaningful ways. For example, we may wish to convert all lower case letters to upper case, all upper case letters to lower case, digit characters to their numeric equivalents, extract words, extract integers, and so forth. In this section we develop several programs which manipulate characters, beginning with simple example functions and continuing with programs for more complex text processing.

4.2.1 Converting Letter Characters

Our next task is to copy input characters to output as before except that all lower case letters are converted to upper case.

Task

COPY1: Copy input characters to output after converting lower case letters to upper case.

The algorithm is similar to COPY0, except that, before printing, each character it is converted to upper case, if necessary.

```
read the first character
repeat as long as NOT end of file
     convert character to upper case
     print the converted character
     read the next character
```

We will write a function, `uppercase()`, to convert a character. The function is given a character and if its argument is a lower case letter, `uppercase()` will return its upper case version; otherwise, it returns the argument character unchanged. The algorithm is:

```
if lower case convert to upper case,
otherwise, leave it unchanged;
```

The prototype for the function is:

```
char uppercase(char ch);
```

The code for the driver and the function is shown in Figure 4.7. The driver is straight forward; each character read is printed in its uppercase version. The while expression is:

```
((ch = getchar()) != EOF)
```

Here we have combined the operations of reading a character and testing for `EOF` into one expression. The innermost parentheses are evaluated first: `getchar()` reads a character and assigns the returned value to `ch`. The value of that expression, namely the value assigned to `ch`, is then compared with `EOF`. If it is not `EOF`, the loop executes, otherwise the loop terminates. The inner parentheses are essential. Without them, the expression is:

```
(ch = getchar() != EOF)
```

```
/*   File: copy1.c
     Programmer:
     Date:
     This program reads a stream of characters until end of file. Each
     character read is converted to its upper case version and printed
     out.
*/
#include <stdio.h>

#define IS_LOWER(c)    ((c) >= 'a' && (c) <= 'z')
#define TO_UPPER(c)    ((c) - 'a' + 'A')
char uppercase(char ch);

main()
{    signed char ch;

     printf("***Copy Program - Upper Case***\n\n");
     printf("Type text, terminate with EOF\n");

     while ((ch = getchar()) != EOF)
          putchar (uppercase(ch));  /* print value of uppercase(ch) */

}

/*   Function returns a lower case letter to an upper case. It returns
     all other characters unchanged.
*/
char uppercase(char c)
{
   if ( IS_LOWER(c) )       /* if c is a lower case letter */
     return TO_UPPER(c);           /* convert to upper case and return */
                                   /* otherwise, */
     return c;                     /* return c unchanged */
}
```

Figure 4.7: Code for upper case

Since the precedence of an assignment operator is the lowest, `getchar()` reads a character and the returned value is first compared to `EOF`:

```
getchar() != EOF
```

The value of this comparison expression, 0 or 1, is then assigned to `ch`: the wrong result is in `ch`. Of course, it is always best to use parentheses whenever there is the slightest doubt. Note, we have used the call to the function, `uppercase()`, as the argument for the routine, `putchar()`. The value returned from `uppercase()` is a character, which is then passed to `putchar()`.

The function, `uppercase()`, checks if `c` is a lower case letter (using the macro `IS_LOWER()`), in which case it returns the upper case version of `c`. We have used the macro `TO_UPPER()` for the expression to convert to upper case, making our program more readable. When the `return` statement is executed, control returns immediately to the calling function, thus, the code after the `return` statement is not executed. Therefore, in this case we do not need the `else` part of the `if` statement. In `uppercase()`, control progresses beyond the `if` statement only if `c` is not a lower case letter, where `uppercase()` returns `c` unchanged. A sample session is shown below:

```
***Copy Program - Upper Case***

Now is the time for all good men
NOW IS THE TIME FOR ALL GOOD MEN
To come to the aid of their country.
TO COME TO THE AID OF THEIR COUNTRY.
^D
```

4.2.2 Converting Digit Characters to Numbers

Next we discuss how digit symbols can be converted to their numeric equivalents and vice versa. As we have stated, the character `'0'` is not the integer, 0, `'1'` is not 1, etc. So it becomes necessary to convert digit characters to their numeric equivalent values, and vice versa. As we have seen, the digit values are contiguous and increasing; the value of `'0'` is 48, `'1'` is 49, and so forth. If we subtract the base value of `'0'`, i.e. 48, from the digit character, we can convert the digit character to its numeric equivalent; e.g. `'0' - '0'` is 0; `'1' - '0'` is 1; and so forth. Thus, if `ch` is a digit character, then its numeric equivalent is `ch - '0'`. Conversely, suppose `n` is a positive integer less than 10, (0, 1, 2, ..., 9). Then the corresponding digit character is `n + '0'`.

Using the sketch of an algorithm just described, we can write two functions that convert a digit character to its integer value, and an integer less than 10 to its character representation. These sound like operations that could be useful in a variety of programs, so we will put the functions in a file called `charutil.c`. These functions are the beginning of a library of character utility functions we will build. The code is shown in Figure 4.8. (We can also place the code for `uppercase()` from the previous example in this file as part ot the library). We have included the

```
/* File: chrutil.c */
/*   This file contains various utility functions for processing characters  */

#include <stdio.h>
#include "chrutil.h"

/*   Function converts ch to an integer if it is a digit. Otherwise, it
     prints an error message.
*/
int dig_to_int(char ch)
{
     if (IS_DIGIT(ch))
          return ch - '0';
     printf("ERROR:dig_to_int:  %c is not a digit\n", ch);
     return ERROR;
}

/*   Function converts a positive integer less than 10 to a corresponding
     digit character.
*/
char int_to_dig(int n)
{
     if (n >= 0 && n < 10)
          return n + '0';
     printf("ERROR:int_to_dig:  %d is not in the range 0 to 9\n", n);
     return NULL;
}
```

Figure 4.8: Code for Character Utilities

```
/*   File:  charutil.h  */
/*   This file contains macros and prototypes for character utilities  */

#define   ERROR     -1

#define   IS_DIGIT(c)   ((c) >= '0' && (c) <= '9')
#define   IS_LOWER(c)   ((c) >= 'a' && (c) <= 'z')

int dig_to_int(char ch);
char int_to_dig(int n);
char uppercase(char ch);
```

Figure 4.9: Header file for Character Utilities

file `charutil.h` where the necessary macros and prototypes are located. This header file is shown in Figure 4.9.

The function `dig_to_int()` is given a character and returns an integer, namely the value of `ch - '0'` if `ch` is a digit character. Otherwise, it prints an error message and returns the value `ERROR`. Since valid integer values of digits are from 0 to 9, a value of -1 is not normally expected as a return value so we can use it to signify an error. (Note, we use a macro, in `charutil.h`, to define this "magic number"). In `int_to_dig()`, given an integer, `n`, the returned value is a digit character, `n + '0'`, if `n` is between 0 and 9; otherwise, a message is printed and the NULL (ASCII value 0) character is returned to indicate an error. We do not use `ERROR` in this case because `int_to_dig()` returns a `char` type value, which may not allow negative values. As was the case for the function `uppercase()` above, in these two functions, we have not used an `else` part. If the condition is satisfied, a `return` statement is executed. The control proceeds beyond the `if` part only if the condition is false. Returning some error value is a good practice when writing utility functions as it makes the functions more general and *robust*, i.e. able to handle valid and invalid data.

Let us consider the task of reading and converting a sequence of digit characters to an equivalent integer. We might add such an operation to our library of character utilities and call it `getint()` (analogous to `getchar()`). We will assume that the input will be a sequence of digit characters, possibly preceded by white space, but not by a plus or minus sign. Further, we will assume that the conversion process will stop when a character other than a digit is read. Usually, the delimiter will be white space, but any non-digit character will also be assumed to delimit the integer being read.

The function, `getint()`, needs no arguments and returns an integer. It will read one character at a time and accumulate the value of the integer. Let us see how a correct integer is accumulated in a variable, `n`. Suppose the digits entered are '3' followed by '4'. When we read the first digit, `'3'`, and convert it to its integer value, we find that `n` is the number, 3. But we do not yet know if our integer is 3, or thirty something, or three hundred something, etc. So we read the next

character, and see that it is a digit character so we know our number is at least thirty something. The second digit is `'4'` which is converted to its integer value, 4. We cannot just add 4 to the previous value of `n` (3). Instead, we must add 4 to the previous value of 3 multiplied by 10 (the base — we are reading a decimal number). The new value of `n` is `n * 10 + 4`, or 34. Again, we do not know if the number being read is 34 or three hundred forty something, etc. If there were another digit entered, say `'5'`, the new value of `n` is obtained by adding its contribution to the previous value of `n` times 10, i.e.

```
n * 10 + dig_to_int('5')
```

which is 345. Thus, if the character read,`ch`, is a digit character, then `dig_to_int(ch)` is added to the previously accumulated value of `n` multiplied by 10. The multiplication by 10 is required because the new digit read is the current rightmost digit with positional weight of 1; so the weight of all previous digits must be multiplied by the base, 10. For each new character, the new accumulated value is obtained by:

```
n = n * 10 + dig_to_int(ch);
```

We can write this as an algorithm as follows:

```
initialize n to zero
read the first character
repeat while the character read is a digit
     accumulate the new value of n by adding
        n * 10 + the integer value of the digit character
     read the next character
return the result
```

The code for `getint()` is shown in Figure 4.10. We have used conditional compilation to test our implementation by including debug statements to print the value of each digit, `ch` and the accumulated value of `n` at each step. The loop is executed as long as the character read is a digit. The macro, `IS_DIGIT()`, expands to an expression which evaluates to True if and only if its argument is a digit. Could we have combined the reading of the character and testing into one expression for the `while`?

```
while( IS_DIGIT(ch = getchar()))
```

The answer is NO! Recall, `IS_DIGIT()` is a macro defined as:

```
#define   IS_DIGIT(c)   ((c) >= '0' && (c) <= '9')
```

so `IS_DIGIT(ch = getchar())` would expand to:

```
/*   File: charutil.c  - continued  */
/*   Function reads and converts a sequence of digit characters to an integer.  */

#define DEBUG

int getint()
{    int n;
     signed char ch;


     ch = getchar();                    /* read next char */
     while (IS_DIGIT(ch)) {             /* repeat as long as ch is a digit */
          n = n * 10 + dig_to_int(ch);  /* accumulate value in n */
#ifdef DEBUG
printf("debug:getint: ch = %c\n", ch);  /* debug statement */
printf("debug:getint: n = %d\n", n);    /* debug statement */
#endif
          ch = getchar();               /* read next char */
     }
     return n;                          /* return the result */

}
```

Figure 4.10: Code for `getint()`

```
((ch = getchar()) >= '0' && (ch = getchar()) <= '9')
```

While this is legal syntax (no compiler error would be generated), the function `getchar()` would be called twice when this expression is evaluated. The first character read will be compared with `'0'` and the second character read will be compared with `'9'` and be stored in the variable `ch`. The lesson here is be careful how you use macros.

Notice we have used the function, `dig_to_int()` in the loop. This is an example of our modular approach — we have already written a function to do the conversion, so we can just use it here, trusting that it works correctly. What if `dig_to_int` ever returns the `ERROR` condition? In this case, we know that that can never happen because if we are in the body of the loop, we know that `ch` is a digit character from the loop condition. We are simply not making use of the full generality of `dig_to_in()t`.

If, after adding the prototype for `getint()` to `charutil.h`:

```
int getint();
```

we compile the file `charutil.c`, we would get a load time error because there is no function `main()` in the file. Remember, every C program must have a `main()`. To test our program, we can write a short driver program which simply calls `getint()` and prints the result:

```
main()
{
     printf("***Test Digit Sequence to Integer***\n\n");
     printf("Type a sequence of digits\n");
     printf("Integer = %d\n", getint()); /* print value */
}
```

A sample session is shown below:

```
***Test Digit Sequence to Integer***

Type a sequence of digits
34
debug:getint:  ch = 3
debug:getint:  n = 16093
debug:getint:  ch = 4
debug:getint:  n = 160934
Integer = 160934
```

It is clear that something is wrong with the accumulated value of `n`. The first character '3' is read correctly; but the value of `n` is 16093. The only possibility is that `n` does not have a correct initial value; we have forgotten to initialize `n` to zero. A simple fix is to change the declaration of `n` in `getint()` to:

```
int n = 0;
```

A revised sample session is shown below.

```
***Test Digit Sequence to Integer***

Type a sequence of digits
3456
debug:getint:  ch = 3
debug:getint:  n = 3
debug:getint:  ch = 4
debug:getint:  n = 34
debug:getint:  ch = 5
debug:getint:  n = 345
debug:getint:  ch = 6
debug:getint:  n = 3456
Integer = 3456
```

The trace shows that the program works correctly. The value of `n` is accumulating correctly. It is 3 after the first character, 34 after the next, 345, after the next, and 3456 after the last character. At this point, we should test the program with other inputs until we are satisfied with the test results for all the diverse inputs. If during our testing we enter the input:

```
***Test Digit Sequence to Integer***

Type a sequence of digits
       123
Integer = 0
```

we get the wrong result and no debug output. Notice, we have added some white space at the beginning of the line. In this case, the first character read is white space, not a digit. So the loop is never entered, no debug statements are executed, and the initial value of `n`, 0, is returned. We have forgotten to handle the case where the integer is preceded by white space. Returning to our algorithm, we must skip over white space characters after the first character is read:

```
initialize n to zero
read the first character
skip leading white space
repeat while the character read is a digit
     accumulate the new value of n by adding
        n * 10 + the integer value of the digit character
     read the next character
return the result
```

This added step can be implemented with a simple `while` loop:

```
while (IS_WHITE_SPACE(ch)) ch = getchar();
```

For readability, we have used a macro, `IS_WHITE_SPACE()`, to test `ch`. We can define the macro in `charutil.h` as follows:

```
#define   IS_WHITE_SPACE(c)   ((c) == ' ' || (c) == '\t' || (c) == '\n')
```

Compiling and testing the program again yields the correct result.

The program may now be considered debugged, it meets the specification given in the task, so we can eliminate the definition for DEBUG and recompile the program. However, at this point we might also consider the utility and generality of our `getint()` function. What happens if the user does not enter digit characters? What happens at end of the file? Only after the program is tested for the "normal" case, should we consider these "abnormal" cases. The first step is to see what the function, as it is currently written, does when it encounters unexpected input.

Let's look at `EOF` first. If the user types end of file, `getchar()` will return `EOF`, which is not white space and is not a digit. So neither loop will be executed and `getint()` will return the initialized value of `n`, namely 0. This may seem reasonable; however, a program using this function cannot tell the difference between the user typing zero and typing end of file. Perhaps we would like `getint()` to indicate end of file by returning `EOF` as `getchar()` does. This is easy to add to our program; before returning `n` we add a statement:

```
if(ch == EOF) return EOF;
```

Of course, if the implementation defines `EOF` as zero, nothing has changed in the behavior of the function. On the other hand, if the implementation defines `EOF` as -1, we can legally enter 0 as input to the program; however, should not expect -1 as a legal value. (In our implementation we have not allowed any negative number, so `EOF` is a good choice for a return value at end of file).

Next, let us consider what happens if the user types a non-digit character. If the illegal character occurs after some digits have been processed, e.g.:

32r

a manual trace reveals that the function will convert the number, 32, and return. If `getint()` is called again, the character, `'r'` will have been read from the buffer so the next integer typed by the user will be read and converted. (Note, this is different than what `scanf()` would do under these circumstances). This is reasonable behavior for `getint()`, so we need make no changes to our code.

If no digits have been typed before an illegal character, e.g.:

r 32

again, the character, `'r'` is not white space and not a digit, so `getint()` will return 0. As before, a program calling `getint()` cannot tell if the user entered zero or an error. It would be better if we return an error condition in this case, but if we return `ERROR`, defined in `charutil.h`, we may not be able to tell the difference between this error and `EOF`. The best solution to this problem is to change the definition of `ERROR` to be -2 instead of -1. This does not affect any other functions that have used `ERROR` (such as `dig_to_int()`) since they only need a unique value to return as an error condition. We can simply change the `#define` in `charutil.h` and recompile (see Figure 4.11). Finally, we must determine how to detect this error in `getint()`. As described above, we must know whether or not we have begun converting an integer when the error occurred. We can do this with a variable, called a **flag**, which stores the *state* of the program. We have called this flag `got_digit` (see Figure 4.12), and declare and initialize it to `FALSE` in `getint()`. If we ever execute the digit loop body, we can set `got_digit` to `TRUE`. Before returning, if `got_digit` is `FALSE` we should return `ERROR`, otherwise we return `n`.

All of these changes are shown in Figures 4.11 and 4.12. Notice we have included the header file, `tfdef.h` from before in the file `charutil.c` to include the definitions of `TRUE` and `FALSE`. We have also modified the test driver to read integers from the input until end of file. (Only the modified versions of `getint()` and the test driver,`main()` are shown in Figure 4.12. The functions `dig_to_int()` and `int_to_dig()` remain unchanged in the file).

Our `getint()` function is now more general and robust (i.e. can handle errors). Of particular note here is the method we used in developing this function. We started by writing the algorithm and code to handle the normal case for input. We then considered what would happen in the abnormal case, and made adjustments to the code to handle them only when necessary. This approach to program development is good for utilities and complex programs: get the normal and easy cases working first; then modify the algorithm and code for unusual and complex cases. Sometimes this approach requires us to rewrite entire functions to handle unusual cases, but often little or no extra code is needed for these cases.

4.2.3 Counting Words

The next task we will consider is counting words in an input text file (a file of characters). A word is a sequence of characters separated by delimiters, namely, white space or punctuation. The first word may or may not be preceded by a delimiter and we will assume the last word is terminated by a delimiter.

Task

CNT: Count the number of characters, words, and lines in the input stream until end of file.

Counting characters and lines is simple: a counter, `chrs`, can be incremented every time a character is read, and a counter, `lns`, can be incremented every time a newline character is read.

```
/* File: chrutil.h */
/*   This file contains various  macros and prototypes for character processing */

#define   ERROR     -2

#define   IS_DIGIT(c)   ((c) >= '0' && (c) <= '9')
#define   IS_LOWER(c)   ((c) >= 'a' && (c) <= 'z')
#define   IS_WHITE_SPACE(c)   ((c) == ' ' || (c) == '\t' || (c) == '\n')

int dig_to_int(char ch);
char int_to_dig(int n);
char uppercase(char ch);
int getint();
```

Figure 4.11: Revised Character Utility Header File

Counting words requires us to know when a word starts and when it ends as we read the sequence of characters. For example, consider the sequence:

```
Lucky     luck
^    ^    ^   ^
```

We have shown the start and the end of a word by the symbol ^. There are several cases to consider:

- As long as no word has started yet AND the next character read is a delimiter, no new word has started.
- If no word has started AND the next character read is NOT a delimiter, then a new word has just started.
- If a word has started AND the next character is NOT a delimiter, then the word is continuing.
- If a word has started AND the character read is a delimiter, then a word has ended.

We can talk about the *state* of our text changing from "a word has not started" to "a word has started" and vice versa. We can use a variable, `inword`, as a flag to keep track of whether a word has started or not. It will be set to True if a word has started; otherwise, it will be set to False. If `inword` is False AND the character read is NOT a delimiter, then a word has started, and `inword` becomes True. If `inword` is True AND the new character read is a delimiter, then the word has ended and `inword` becomes False. With this information about the state, we can count a word either when it starts or when it ends. We choose the former, so each time the flag changes from False to True, we will increment the counter, `wds`. The algorithm is:

```
/* File: chrutil.c */
/*   This file contains various utility functions for processing characters  */

#include <stdio.h>
#include "tfdef.h"
#include "chrutil.h"

/*  Function reads the next integer from the input  */
int getint()
{    int n = 0;
     int got_dig = FALSE;
     signed char ch;


     ch = getchar();                    /* read next char                  */
     while (IS_WHITE_SPACE(ch))         /* skip white space                */
              ch = getchar();
     while (IS_DIGIT(ch)) {             /* repeat as long as ch is a digit */
          n = n * 10 + dig_to_int(ch);  /* accumulate value in n           */
          got_dig = TRUE;
#ifdef DEBUG
printf("debug:getint: ch = %c\n", ch);  /* debug statement */
printf("debug:getint: n = %d\n", n);    /* debug statement */
#endif
          ch = getchar();               /* read next char                  */
     }
     if(ch == EOF) return EOF;          /* test for end of file            */
     if(!got_dig)  return ERROR;        /* test for no digits read         */
     return n;                          /* otherwise return the result     */

}

/*  Dummy test driver for character utilities              */
/*   This driver will be removed after testing is complete  */
main()
{  int x;
     printf("***Test Digit Sequence to Integer***\n\n");
     printf("Type a sequence of digits\n");

     while((x = getint()) != EOF)
          printf("Integer = %d\n", x);       /* print value */
}
```

Figure 4.12: Revised Character Utility Code

```
initialize all counters to zero, set inword to False
while the character read, ch, is not EOF
     increment character count chrs
     if ch is a newline
          increment line count lns
     if NOT inword AND ch is NOT delimiter
          increment word count wds
          set inword to True
     else if inword and ch is delimiter
          set inword to False
print results
```

We first count characters and newlines. After that, only changes in the state, `inword`, need to be considered; otherwise we ignore the character and read in the next one. Each time the flag changes from False to True, we count a word. We will use a function `delimitp()` that checks if a character is a delimiter, i.e. if it is a white space or a punctuation. (The name `delimitp` stands for "delimit predicate" because it tests is its argument is a delimiter and returns True or False). White space and punctuation, in turn, will be tested by other functions. The code for the driver is shown in Figure 4.13.

After printing the program title, the counts are initialized:

```
lns = wds = chrs = 0;
```

Assignment operators associate from right to left so the rightmost operator is evaluated first; `chrs` is assigned 0, and the value of the assignment operation is 0. This value, 0, is then assigned to `wds`, and the value of that operation is 0. Finally, that value is assigned to `lns`, and the value of the whole expression is 0. Thus, the statement initializes all three variables to 0 as a concise way of writing three separate assignment statements.

The program driver follows the algorithm very closely. The function `delimitp()` is used to test if a character is a delimiter and is yet to be written. Otherwise, the program is identical to the algorithm. It counts every character, every newline, and every word each time the flag `inword` changes from False to True.

Source File Organization

We can add the source code for `delimitp()` to the source file `charutil.c` we have been building with character utility functions. In the last section we wrote a dummy driver in that file to test our utilities. Since we would like to use these utilities in many different programs, we should not have to keep copying a driver into this file. We will soon see how the code in `charutil.c` will be made a part of the above program without combining the two files into one (and without using the `#include` directive to include a code file). In our program file, `cnt.c`, we also include two header files besides `stdio.h`. These are: `tfdef.h` which defines symbolic constants `TRUE` and `FALSE`; and

```
/*   Program File: cnt.c
     Other Source Files: charutil.c
     Header Files: tfdef.h, charutil.h
     This program reads standard input characters and counts the number
     of lines, words, and characters. All characters are counted including
     the newline and other control characters, if any.
*/

#include <stdio.h>
#include "tfdef.h"
#include "charutil.h"

main()
{    signed char ch;
     int inword,         /* flag for in a word */
     lns, wds, chrs;     /* Counters for lines, words, chars. */

     printf("***Line, Word, Character Count Program***\n\n");
     printf("Type characters, EOF to quit\n");
     lns = wds = chrs = 0;    /* initialize counters to 0 */
     inword = FALSE;          /* set inword flag to False */

     while ((ch = getchar()) != EOF) {    /* repeat while not EOF */
          chrs = chrs + 1;               /* increment chrs */
          if (ch == '\n')                /* if newline char */
               lns = lns + 1;            /* increment lns */

          /* if not inword and not a delimiter  */
          if (!inword && !delimitp(ch)) { /* if not in word and not delim., */
               inword = TRUE;           /* set inword to True */
               wds = wds + 1;           /* increment wds */
          }

          else if (inword && delimitp(ch))   /* if in word and a delimiter*/
               inword = FALSE;               /* set inword to False */

     } /* end of while loop */
     printf("Lines = %d, Words = %d, Characters = %d\n",
               lns, wds, chrs);
} /* end of program */
```

Figure 4.13: Code for Count Words Driver

```
/* File: tfdef.h */
#define TRUE 1
#define FALSE 0

/*   File: charutil.h   - continued
     This file contains the prototype declarations for functions defined in
     charutil.c.
*/
int delimitp(char c);     /* Tests if c is a delimiter (white space, punct) */
int whitep(char c);       /* Tests if c is a white space */
int punctp(char c);       /* Tests if c is a punctuation */
```

Figure 4.14: Header Files for Word Count

`charutil.h` which declares prototypes for the functions defined in `charutil.c` and any related macros. Since we use these constants and functions in `main()`, we should include the header files at the head of our source file. Figure 4.14 shows the file `tfdef.h` and the additions to `charutil.h`.

The function `delimitp()` tests if a character is white space or punctuation. It uses two functions for its tests: `whitep()` which tests if a character is white space, and `punctp()` which tests if a character is punctuation. (We could have also implemented these as macros, but chose functions in this case). All these functions are added to the source file, `charutil.c` and are shown in Figure 4.15 This source file also includes `tfdef.h`, and `charutil.h` because the functions in the file use the symbolic constants `TRUE` and `FALSE` defined in `tfdef.h` and the prototypes for functions `whitep()` and `punctp()` declared in `charutil.h` are also needed in this file.

The source code for the functions is simple enough; `delimitp()` returns `TRUE` if the its parameter, `c`, is either white space or punctuation; `whitep()` returns `TRUE` if `c` is either a space, newline, or tab; and `punctp()` returns `TRUE` if `c` is one of the punctuation marks shown. All functions return `FALSE` if the primary test is not satisfied.

Our *entire* program is now contained in the two source files `cnt.c` and `charutil.c` which must be compiled separately and linked together to create an executable code file. Commands to do so are implementation dependent; but on Unix systems, the shell command line is:

```
cc -o cnt cnt.c charutil.c
```

The command will compile `cnt.c` to the object file, `cnt.o`, then compile `charutil.c` to the object file, `charutil.o`, and finally link the two object files as well as any standard library functions into an executable file, `cnt` as directed by the `-o cnt` option. (If -o option is omitted, the executable file will be called `a.out`). For other systems, the commands are generally similar; for example, compilers for many personal computers also provide an integrated environment which allows one to edit, compile, and run programs. In such an environment, the programmer may be asked to prepare a project file listing all source files. Once a project file is prepared and the project

```
/*   File: charutil.c  - continued */
#include "tfdef.h"
#include "charutil.h"
/*   Function returns TRUE if c is a delimiter, i.e., it is a white space
     or a punctuation. Otherwise, it returns FALSE.
*/
int delimitp(char c)
{
     if (whitep(c) || punctp(c))
          return TRUE;
     return FALSE;
}

/* Function returns TRUE if c is white space; returns FALSE otherwise. */
int whitep(char c)
{
     if (c == '\n' || c == '\t' || c == ' ')
          return TRUE;
     return FALSE;
}

/* Function returns TRUE if c is a punctuation; returns FALSE otherwise. */
int punctp(char c)
{
     if (c == '.' || c == ',' || c == ';' || c == ':'
               || c == '?' || c == '!')
          return TRUE;
     return FALSE;
}
```

Figure 4.15: Code for Word Count Utility Functions

option activated, a simple command compiles the source files, links them into an executable file, and executes the program. Consult your implementation manuals for details. This technique of splitting the source code for an entire program into multiple files is called **serarate compilation** and is a good practice as programs grow larger.

Once the above two files, `cnt.c` and `charutil.c` are compiled and linked, the resulting program may then be executed producing a sample session as shown below:

```
***Line, Word, Character Count Program***

Type characters, EOF to quit
Now is the time for all good men
To come to the aid of their country.
^D
Lines = 2, Words = 16, Characters = 70
```

Henceforth, we will assume separate compilation of source code whenever it is spread over more then one file. Since `main()` is the program driver, we will refer to the source file that contains `main()` as the **program file**. Other source files needed for a complete program will be listed in the comment at the head of the program file. In the comment, we will also list header files needed for the program. Refer to `cnt.c` in Figure 4.13 for an example of a listing which enumerates all the files needed to build or create an executable program. (The file `stdio.h` is not listed since it is assumed to be present in all source files).

Header files typically include groups of related symbolic constant definitions and/or prototype declarations. Source files typically contain definitions of functions used by one or more program files. We will organize our code so that a source file contains the code for a related set of functions, and a header file with the same name contains prototype declarations for these functions, e.g. `charutil.c` and `charutil.h`. As we add source code for new functions to the source files, corresponding prototypes will be assumed to be added in the corresponding header files.

Separate compilation has several advantages. Program development can take place in separate modules, and each module can be separately compiled, tested, and debugged. Once debugged, a compiled module need not be recompiled but merely linked with other separately compiled modules. If changes are made in one of the source modules, only that source module needs recompiling and linking with other already compiled modules. Furthermore, compiled modules of useful functions can be used and reused as building blocks to create new and diverse programs. In summary, separate compilation saves compilation time during program development, allows development of compiled modules of useful functions that may be used in many diverse programs, and makes debugging easier by allowing incremental program development.

4.2.4 Extracting Words

The final task in this section extends the word count program to print each word in the input stream of characters.

Task

WDS: Read characters until end of file and keep a count of characters, lines, and words. Also, print each word in the input on a separate line.

The logic is very similar to that of the previous program, except that a character is printed if it is in a word, i.e. if `inword` is True. We will decide whether to print a character only after a possible state change of `inword` has taken place. That way when `inword` changes from False to True (the first character of a word has been found) the character is printed. When `inword` changes from from True to False (a delimiter has been found ending the word) it is not printed, instead we print a newline because each word is to be printed on a new line. So our algorithm is:

```
initialize counts to zero, set inword to False
while the character read, ch, is not EOF
     increment character count, chrs
     if ch is a newline
          increment line count, lns
     if NOT inword AND ch is NOT delimiter
          increment word count, wds
          set inword to True
     else if inword and ch is delimiter
          set inword to False
          print a newline
     if inword
          print ch
print results
```

and the code is shown in Figure 4.16. This code was generated by simply copying the file `cnt.c` and making the necessary changes as indicated in the algorithm. The program file is compiled and linked with `charutil.c`. and the following sample session is produced.

Sample Session:

```
***Word Program***

Type characters, EOF to quit
Now is the time for all good men
Now
is
the
time
for
all
good
men
```

```
/*   Program File: wds.c
     Other Source Files: charutil.c
     Header Files: tfdef.h, charutil.h
     This program reads standard input characters and prints each word on a
     separate line. It also counts the number of lines, words, and characters.
     All characters are counted including the newline and other control
     characters, if any.
*/

#include <stdio.h>
#include "tfdef.h"
#include "charutil.h"

main()
{    signed char ch;
     int inword,         /* flag for in a word                */
     lns, wds, chrs;     /* Counters for lines, words, chars. */

     printf("***Line, Word, Character Count Program***\n\n");
     printf("Type characters, EOF to quit\n");
     lns = wds = chrs = 0;    /* initialize counters to 0 */
     inword = FALSE;          /* set inword flag to False */

     while ((ch = getchar()) != EOF) {   /* repeat while not EOF */
          chrs = chrs + 1;               /* increment chrs       */
          if (ch == '\n')                /* if newline char      */
               lns = lns + 1;            /* increment lns        */

          /* if not inword and not a delimiter  */
          if (!inword && !delimitp(ch)) {   /* if not in word and not delim. */
               inword = TRUE;               /* set inword to True            */
               wds = wds + 1;               /* increment wds                 */
          }

          else if (inword && delimitp(ch)) {  /* if in word and a delimiter*/
               inword = FALSE;                /* set inword to False       */
               putchar('\n');                 /* end word with a newline   */
          }

          if (inword)                    /*  if in a word         */
               putchar(ch);              /*  print the character  */

     } /* end of while loop */
     printf("Lines = %d, Words = %d, Characters = %d\n",
               lns, wds, chrs);
} /* end of program */
```

Figure 4.16: Code fore extracting words

```
^D
Lines = 1, Words = 8, Characters = 33
```

In this section we have seen several sample programs for processing characters as well as some new programming techniques, in particular, splitting the source code for a program into files of related functions with separate compilation of each source code file. The executable program is then generated by linking the necessary object files. In the next section, we turn our attention to several new control constructs useful in character processing as well as in numeric programs.

4.3 New Control Constructs

Earlier in this chapter, we saw the use of a chain of `if...else if` constructs for a multiway decision. This is a common operation in programs so the C language provides an alternate multiway decision capability: the `switch` statement. In addition, two other control constructs are discussed in this section: the `break` and `continue` statements.

4.3.1 The `switch` Statement

In a `switch` statement, the value of an *integer valued expression* determines an alternate path to be executed. The syntax of the `switch` statement is:

switch (<expression>) <statement>

Typically, the <statement> is a compound statement with case labels.

```
switch (<expression>) {
        case <e1 >: <stmt1 >
        case <e2 >: <stmt2 >
        ...
        case <en−1 >: <stmtn−1 >
        default: <stmtn >
}
```

Each statement, except the last, starts with a case label which consists of the keyword `case` followed by a *constant expression*, followed by a colon. The constant expression, (whose value must be known at compile time) is called a **case expression**. An optional `default` label is also allowed after all the case labels. Executable statements appear after the labels as shown.

The semantics of the `switch` statement is as follows: The expression, <expression> is evaluated to an integer value, and control then passes to the first `case` label whose case expression value

matches the value of the switch expression. If no case expression value matches, control passes to the statement with the `default` label, if present. This control flow is shown in Figure 4.17. Labels play no role other than to serve as markers for transferring control to the appropriate statements. Once control passes to a labeled statement, the execution proceeds from that point and continues to process each of the subsequent statements until the end of the `switch` statement.

As an example, we use the `switch` statement to write a function that tests if a character is a vowel (the vowels are 'a', 'e', 'i', 'o', and 'u' in upper or lower case). If a character passed to this function, which we will call `vowelp()` (for vowel predicate), is one of the above vowels, the function returns True; otherwise, it returns False. We add the function to our file `charutil.c`, and the code is shown in Figure 4.18. If `c` matches any of the cases, control passes to the appropriate `case` label. For many of these cases, the <`stmt`> is empty, and the first non-empty statement is the `return TRUE` statement, which, when executed, immediately returns control to the calling function. If `c` is not a vowel, control passes to the `default` label, where the `return FALSE` statement is executed. While there is no particular advantage in doing so, the above function could be written with a `return` statement at every `case` label to return `TRUE`. The function `vowelp()` is much clearer and cleaner using the `switch` statement than it would have been using nested `if` statements or an `if` statement with a large, complex condition expression.

An Example: Encrypting Text

Remember, in a `switch` statement, control flow passes to the statement associated with the matching `case` label, and continues from there to all subsequent statements in the compound statement. Sometimes this is not the desired behavior. Consider the task of encrypting text in a very simple way, such as:

- Leave all characters except the letters unchanged.
- Encode each letter to be the next letter in a circular alphabet; i.e. `'a'` follows `'z'` and `'A'` follows `'Z'`.

We will use a function to print the next letter. The encrypt algorithm is simple enough:

```
read characters until end of file
     if a char is a letter
          print the next letter in the circular alphabet
     else
          print the character
```

Implementation is straight forward as shown in Figure 4.19. The program reads characters until end of file. Each character is tested to see if it is a letter using a function, `letterp()`. If it is a letter, `print_next()` is called to print the next character in the alphabet; otherwise, the character is printed as is. The function `letterp()` checks if a character passed as an argument is an alphabetic letter and returns True or False. The function is shown below and is added to our utility file, `charutil.c` (and its prototype is assumed to be added to the file `charutil.h`).

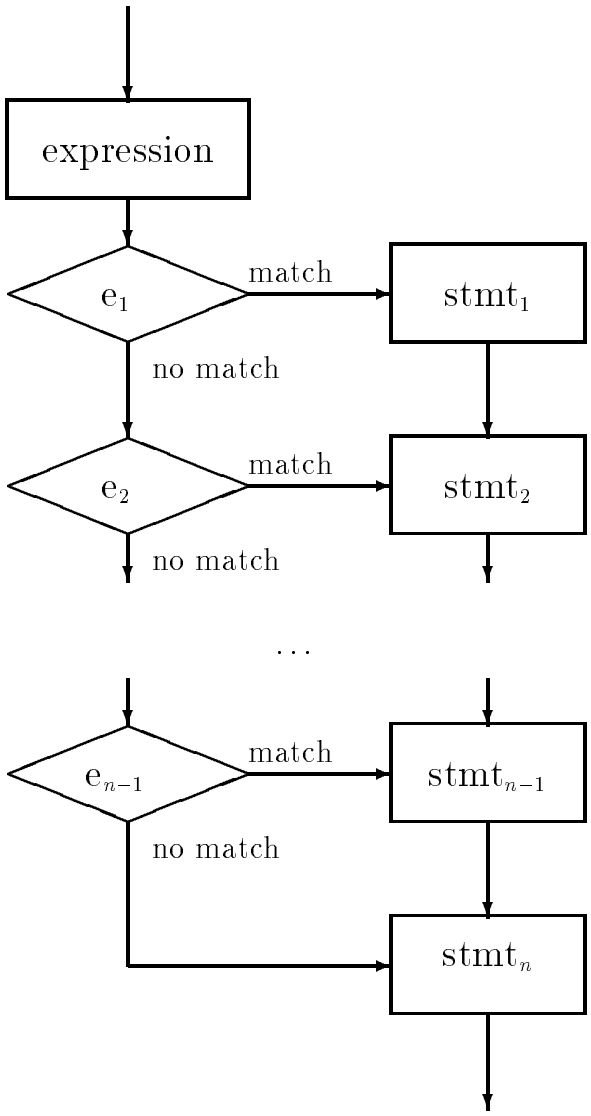

Figure 4.17: Control Flow for switch statement

```
/* File: charutil.c - continued */
/* File tfdef.h, which defines TRUE and FALSE, has already been
included in this file. */
/*   Function checks if c is a vowel. */
int vowelp(char c)
{
     switch(c) {
          case 'a':
          case 'A':
          case 'e':
          case 'E':
          case 'i':
          case 'I':
          case 'o':
          case 'O':
          case 'u':
          case 'U':  return TRUE;
          default:   return FALSE;
     }
}
```

Figure 4.18: Code for `vowelp()` Using a `switch` Statement

```
/*   File: encrypt.c
     Other Source Files: charutil.c
     Header Files: charutil.h
     This program encrypts text by converting each letter to the next letter
     in the alphabet. The last letter of the alphabet is changed to the first
     letter.
*/

#include <stdio.h>
#include "charutil.h"
void print_next(char c);

main()
{    signed char c;

     printf("***Text Encryption***\n\n");
     printf("Type text, EOF to quit\n");

     while ((c = getchar()) != EOF) {
          if (letterp(c))
               print_next(c);

          else
               putchar(c);
     }
}
```

Figure 4.19: Code for `encrypt.c`

```
/* File: charutil.c - continued */
/* Function tests if c is an alphabetic letter. */
int letterp(char c)
{
     if (IS_LOWER(c) || IS_UPPER(c))
          return TRUE;
     return FALSE;
}
```

It uses the macros `IS_LOWER()` and `IS_UPPER()`. We have already define `IS_LOWER()` in `charutil.h`; `IS_UPPER()` is similar:

```
     #define   IS_UPPER(c)   ((c) >= 'A' && (c) <= 'Z')
```

and is added to `charutil.h`.

Let us consider the function, `print_next()`, which is passed a single alphabetic letter as an argument. It should print an altered letter, that is the next letter in a circular alphabet. The altered letter is the next letter in the alphabet, unless the argument is the last letter in the alphabet. If the argument is `'z'` or `'Z'`, then the altered letter is the first letter of the alphabet, `'a'` or `'A'` respectively. There are two possible instances of the character `c` for which we must take special action, viz. when `c` is `'z'` or `c` is `'Z'`. The default case is any other letter, when the function should print `c + 1`, which is the ASCII value of the next letter.

We need a three way decision based on the value of a character `c`: is `c` the character `'z'`, or `'Z'`, or some other character? If it is `'z'` print `'a'`; else if it is `'Z'` print `'A'`; otherwise, print `c + 1`. We can easily implement this multiway decision using an `if ... else ...` construct.

```
     if (c == 'z')
          printf("%c", 'a');
     else if (c == 'Z')
          printf("%c", 'A');
     else
          printf("%c", c + 1);
```

Such multiway branches can also be implemented using the `switch` construct. Suppose we wrote:

```
     switch(c) {
          case 'z': printf("%c", 'a');
          case 'Z': printf("%c", 'A');
          default:  printf("%c", c + 1);
```

Will this do what we want? If `c` has the value `'z'`, the above `switch` statement would match the first `case` label and print `'a'`. However, by the semantics of `switch`, it would then print `'A'` followed by `'{'` (the character after `'z'` in the ASCII table) — not what we want. Can we salvage this approach to multiway branching?

```
/*   File: encrypt.c - continued */
/*   Prints the next higher letter to c. Alphabet is assumed circular. */
void print_next(char c)
{
     switch(c) {
          case 'z': printf("%c", 'a');
                    break;
          case 'Z': printf("%c", 'A');
                    break;
          default:  printf("%c", c + 1);
     }
}
```

Figure 4.20: Implementing `print_next()` Using a `switch` Statement

4.3.2 The `break` Statement

C provides a statement for circumstances like this; the `break` statement. A `break` can only be used within a `switch` statement or any looping statement (so far we have only seen `while`). Its syntax is very simple:

```
break;
```

The semantics of `break` are to immediately terminate the closest enclosing compound statement; either the `switch` or the loop.

To fix our problem above, Figure 4.20 shows an implementation of `print_next()` using a `switch` statement. Once control passes to a label, the control continues down the line of statements until the `break` statement is encountered. In the above case, if `c` is `'Z'`, then an `'A'` is printed and the switch statement is terminated. Similarly, if `c` is `'z'`, then `'a'` is printed and the control passes to the next statement after the `switch`. If there is no match, then the control passes to the `default` label, and a character with value `c + 1` is printed. The `switch` statement ends at this point anyway, so no `break` is required.

Here is a sample session with the program after `encrypt.c` and `charutil.c` are compiled and linked.

```
***Text Encryption***

Type text, EOF to quit
this is a test
uijt jt b uftu
^D
```

```
/*  Function prints a character, its decimal, octal, and hex value
    and its category, using a switch statement
*/
int print_category( int cat, char ch)
{
     printf("%c, ASCII value decimal %d, octal %o, hexadecimal %x: ",
                    ch,ch,ch,ch);
     switch(cat)   {
     case LOWER:   printf("lower case letter\n");
                   break;
     case UPPER:   printf("an upper case letter\n");
                   break;
     case DIGIT:   printf("a digit symbol\n");
                   break;
     case PUNCT:   printf("a punctuation symbol\n");
                   break;
     case SPACE:   printf("a space character\n");
                   break;
     case CONTROL: printf("a control character\n");
                   break;
     default:      printf("a special symbol\n");
     }
}
```

Figure 4.21: New Implementation of `print_category` using `switch`

This use of the `switch` statement with `break` statements in the various cases is a common and efficient way to implement a multiway branch in C. For example, we can now reimplement our `print_category()` function from Figure 4.4 as shown in Figure 4.21.

As mentioned above, the `break` statement can also be used to terminate a loop. Let us consider our previous word extraction task: reading text input and printing each word in the text (see Figure 4.16). However, now we will consider non-printable characters other than white space and the end of file marker as invalid. They will represent an error in the input and we will use a `break` statement to abort the program.

For this task, we will no longer count characters, words, and lines, simply extract words and print them, one per line. In our previous algorithm, each iteration of the loop processed one character and we used a flag variable, `inword` to carry information from one iteration to the next. For this program we will modify our algorithm so that each iteration of the loop will process one word. Each word is found by first skipping over leading delimiters, then, as long as we read printable, non-delimiter characters, we can print the word. The character terminating the word must be a delimiter unless it is a non-printable character or we have reached the end of file. In either of those cases, we abort the program, printing a message if a non-printable character was encountered. Otherwise, we print the newline terminating the word and process the next word.

Here is the revised algorithm with the code shown in Figure 4.22.

```
while there are more characters to read
     skip over leading delimiters (white space)
     while character is legal for a word
          print character
          read next character
     if EOF, terminate the program
     if character is non-printable,
          print a message and abort the program
     print a newline ending the word
```

The program uses two functions: `delimitp()` tests if the argument is a delimiter, and `illegal()` tests if the argument is not a legal character (printable or a delimiter). They are in the source file `charutil.c`; their prototypes are in `charutil.h`. We have already defined `delimitp()` (see Figure 4.15). We will soon write `illegal()`.

In the main loop, we skip over leading delimiters with a `while` loop, and then, as long legal "word" characters are read we print and read characters. If either of these loops terminates with `EOF`, the loop is terminated by a `break` statement and the program ends. (Note, if `EOF` is detected while skipping delimiters, the word processing loop will be executed zero times). If a non-printable, non-delimiter character is found, the program is aborted after a message is printed to that effect. Otherwise, the word is ended with a newline and the loop repeats.

Function `illegal()` is easy to write: legal characters are printable (in the ASCII range 32 through 126) or white space. Here is the function and its prototype.

```
/*   File: charutil.c - continued
     Header Files: tfdef.h, charutil.h
*/
/* Function tests if c is printable. */
int illegal(char c)
{
   if (IS_PRINT(c) || IS_WHITE_SPACE(c))
     return FALSE;
   return TRUE;
}

/*   File: charutil.h - continued */

#define  IS_PRINT(c)     ((c) >= 32 && (c) < 127)

int illegal(char c);      /* Tests if c is legal. */
```

We have also added the macro `IS_PRINT` to the header file. The program file `words.c` and the source file `charutil.c` can now be compiled and linked. A sample session when the program is

```
/*   File: words.c
     Other Source Files: charutil.c
     Header Files: tfdef.h, charutil.h
     This program reads text and extracts words until end of file. Only
     printable characters are allowed in a word. Upon encountering a control
     character, a message is printed and the program is aborted.
*/
#include <stdio.h>
#include "tfdef.h"
#include "charutil.h"  /* includes prototypes for delimitp(), printp() */

main()
{    signed char ch;

     printf("***Words: Non-Printable Character Aborts***\n\n");
     printf("Type text, EOF to quit\n");

     while ((ch = getchar()) != EOF) {  /* while characters remain to be read */

          while (delimitp(ch))          /* skip over leading delimiters       */
               ch = getchar();

          while (!delimitp(ch) && printp(ch)) {   /* process  a word    */
               putchar(ch);                       /* print ch          */
               ch = getchar();                    /* read the next char */
          }

          if (ch == EOF)                /* if end of file, terminate          */
               break;

          if (illegal(ch)) {/* if a control char, print msg and abort */
               printf("\nAborting - Control character present: ASCII %d\n",ch);
               break;
          }

          printf("\n");                  /* terminate word with newline       */
     }
}
```

Figure 4.22: Extracting Words Using `break`

executed is shown below.

```
***Words:  Non-Printable Character Aborts***

Type text, EOF to quit
Lucky you live H^Awaii^A
Lucky
you
live
H
Aborting - Control character present:  ASCII 1
```

The message shows that the program is abnormally terminated due to the presence of a control character.

It is also possible, though not advisable, to use a `break` statement to terminate an otherwise infinite loop. Consider the program fragment:

```
n = 0;
while (1) {
     n = n + 1;
     if (n > 3) break;
     printf("Hello, hello, hello\n");
}
printf("Print statement after the loop\n");
```

The loop condition is the constant 1, which is always True so the loop body will be repeatedly executed, `n` will be incremented, and the message printed, until `n` reaches 4. The condition `(n > 3)` will now be True, and the `break` statement will be executed. This will terminate the `while` loop, and control passes to the print statement after the loop. If the `if` statement containing the `break` statement were not present, the loop would execute indefinitely.

While it is possible to use a `break` statement to terminate an infinite loop, it is not a good practice because use of infinite loops makes program logic hard to understand. In a well structured program, all code should be written so that program logic is clear at each stage of the program. For example, a loop should be written so that the normal loop terminating condition is immediately clear. Otherwise, program reading requires wading through the detailed code to see how and when the loop is terminated. A `break` statement should be used to terminate a loop only in cases of special or unexpected events.

4.3.3 The `continue` Statement

A `continue` statement also changes the normal flow of control in a loop. When a `continue` statement is executed in a loop, the current iteration of the loop body is aborted; however, control

transfers to the loop condition test and normal loop processing continues, namely either a new iteration or a termination of the loop occurs based on the loop condition. As might be expected, the syntax of the `continue` statement is;

```
continue;
```

and the semantics are that statements in the loop body following the execution of the `continue` statement are not executed. Instead, control immediately transfers to the testing of the loop condition.

As an example, suppose we wish to write a loop to print out integers from 0 to 9, except for 5. We could use the `continue` statement as follows:

```
n = 0;
while (n < 10) {
     if (n == 5) {
          n = n + 1;
          continue;
     }
     printf("Next allowed number is %d\n", n);
     n = n + 1;
}
```

The loop executes normally except when `n` is 5. In that case, the `if` condition is True; `n` is incremented, and the `continue` statement is executed where control passes to the testing of the loop condition, `(n < 10)`. Loop execution continues normally from this point. Except for 5, all values from 0 through 9 will be printed.

We can modify our previous text encryption algorithm (Figure 4.19) to ignore illegal characters in its input. Recall, in that task we processed characters one at a time, encrypting letters and passing all other characters as read. In this case we might consider non-printable characters other than white space to be typing errors which should be ignored and omitted from the output.

The code for the revised program is shown in Figure 4.23. We have used the function, `illegal()`, from the previous program (it is in `charutil.c`) to detect illegal characters. When found, the `continue` statement will terminate the loop iteration, but continue processing the remaining characters in the input until `EOF`.

Sample Session:

```
***Text Encryption Ignoring Illegal Characters***

Type text, EOF to quit
Luck you live H^Awaii
Mvdl zpv mjjwf Ixbjj
```

```
/*   File: encrypt2.c
     Other Source Files: charutil.c
     Header Files: charutil.h
     This program encrypts text by converting each letter to the next letter
     in the alphabet. Illegal characters are ignored.
*/

#include <stdio.h>
#include "charutil.h"
void print_next(char c);

main()
{    signed char c;

     printf("***Text Encryption Ignoring Illegal Characters***\n\n");
     printf("Type text, EOF to quit\n");

     while ((c = getchar()) != EOF) {    /* while there are chars to process */

          if (illegal(c)) continue;           /* ignore illegal characters   */

          if (letterp(c))                     /* encrypt letters             */
               print_next(c);

          else
               putchar(c);                    /* print all others as is      */
     }
}
```

Figure 4.23: Code for Revised `encrypt.c`

```
/*   File: scan0.c
     This program shows problems with scanf() when wrong data is entered.
*/
#include <stdio.h>

main()
{    int cnt, n;

     printf("***Numeric and Character Data***\n\n");
     printf("Type integers, EOF to quit: ");

     cnt = 0;
     while ((scanf("%d", &n) != EOF) && (cnt < 4)) {
          printf("n = %d\n", n);

          cnt = cnt + 1;
          printf("Type an integer, EOF to quit: ");
     }
}
```

Figure 4.24: Code for Testing `scanf()`

```
     ^D
```

It should be noted that the use of `break` and `continue` statements is not strictly necessary. Proper structuring of the program, using appropriate loop and `if...else` constructs, can produce the same effect. The `break` and `continue` statements are best used for "unusual" conditions that would make program logic clearer.

4.4 Mixing Character and Numeric Input

We have seen how numeric data can be read with `scanf()` and character data with either `scanf()` or `getchar()`. Some difficulties can arise, however, when both numeric and character input is done within the same program. Several common errors in reading data can be corrected easily if the programmer understands exactly how data is read. In this section, we discuss problems in reading data and how they can be resolved.

The first problem occurs when `scanf()` attempts to read numeric data but the user enters the data incorrectly. (While the discussion applies to reading any numeric data, we will use integer data for our examples). Consider an example of a simple program that reads and prints integers as shown in Figure 4.24. In this program, `scanf()` reads an integer into the variable `n` (if possible) and returns a value which is compared with `EOF`. If `scanf()` has successfully read an integer,

the value returned is the number of conversions performed, namely 1, and the loop is executed. Otherwise, the value returned is expected to be `EOF` and the loop is terminated. The the first part of the `while` condition is:

```
(scanf("%d", &n) != EOF)
```

This expression both reads an item and compares the returned value with `EOF`, eliminating separate statements for initialization and update. The second part of the `while` condition ensures that the loop is executed at most 4 times. (The reason for this will become clear soon). The loop body prints the value read and keeps a count of the number of times the loop is executed. The program works fine as long as the user enters integers correctly. Here is a sample session that shows the problem when the user makes a typing error:

```
***Mistyped Numeric Data***

Type integers, EOF to quit:  23r
n = 23
Type an integer, EOF to quit:  n = 23
Type an integer, EOF to quit:  n = 23
Type an integer, EOF to quit:  n = 23
```

The user typed *23r*. These characters and the terminating newline go into the keyboard buffer, `scanf()` skips over any leading white space and reads characters that form an integer and converts them to the internal form for an integer. It stops reading when the first non-digit is encountered, in this case, the `'r'`. It stores the integer value, 23, in `n` and returns the number of items read, i.e. 1. The first integer, 23, is read correctly and printed , followed by a prompt to type in the next integer.

At this point, the program does not wait for the user to enter data; instead the loop repeatedly prints 23 and the prompt but does not read anything. The reason is that the next character in the keyboard buffer is still `'r'`. This is not a digit character so it does not belong in an integer; therefore, `scanf()` is unable to read an integer. Instead, `scanf()` simply returns the number of items read as 0 each time. Since `scanf()` is trying to read an integer, it can not read and discard the `'r'`. No more reading of integers is possible as long as `'r'` is the next character in the buffer. If the value of the constant `EOF` is -1 (not 0), an infinite loop results. (That is why we have included the test of `cnt` to terminate the loop after 4 iterations).

Let us see how we can make the program more tolerant of errors. One solution to this problem is to check the value returned by `scanf()` and make sure it is the expected value, i.e. 1 in our case. If it is not, break out of the loop. The `while` loop can be written as:

```
while ((flag = scanf("%d", &n)) != EOF) {
     if (flag != 1) break;
     printf("n = %d\n", n);
```

```
     printf("Type an integer, EOF to quit\n");
}
```

In the `while` expression, the inner parentheses are evaluated first. The value returned by `scanf()` is assigned to `flag` which is the value that is then compared to `EOF`. If the value of the expression is not `EOF`, the loop is executed; otherwise, the loop is terminated. In the loop, we check if a data item was read correctly, i.e. if `flag` is 1. If not, we break out of the loop. The inner parentheses in the while expression are important; the while expression without them would be:

```
     (flag = scanf("%d", &n) != EOF)
```

Precedence of assignment operator is lower than that of the relational operator, ! =; so, the `scanf()` value is first compared with `EOF` and the result is True or False, i.e. 1 or 0. *This* value is then assigned to `flag`, NOT the value returned by `scanf()`.

The trouble with the above solution is that the program is aborted for a simple typing error. The next solution is to flush the buffer of all characters up to and including the first newline. A simple loop will take care of this:

```
while ((flag = scanf("%d", &n)) != EOF) {
     if (flag != 1)
          while (getchar() != '\n');
     else {
          printf("n = %d\n", n);
          printf("Type an integer, EOF to quit\n");
     }
}
```

If the value returned by `scanf()` when reading an integer is not 1, then the inner while loop is executed where, as long as a newline is not read, the condition is True and the body is executed. In this case, the loop body is an empty statement, so the condition will be tested again thus reading the next character. The loop continues until a newline is read. This is called **flushing the buffer**.

The trouble with this approach is that the user may have typed other useful data on the same line which will be flushed. The best solution is to flush only one character and try again. If unsuccessful, repeat the process until an item is read successfully. Figure 4.25 shows the revised program that will discard only those characters that do not belong in a numeric data item.

Sample Session:

```
     ***Mistyped Numeric Data:  Flush characters***

     Type integers, EOF to quit
```

```
/*   File: scan1.c
     This program shows how to handle mistyped numeric data by flushing
     erroneous characters.
*/
#include <stdio.h>

#define DEBUG

main()
{    char ch;
     int flag, n;

     printf("***Mistyped Numeric Data: Flush characters***\n\n");
     printf("Type integers, EOF to quit\n");

     while ((flag = scanf("%d", &n)) != EOF) {

          if (flag != 1) {
               ch = getchar();              /* flush one character */
#ifdef DEBUG
printf("debug:%c in input stream, discarding\n", ch);
#endif
          }

          else   printf("n = %d\n", n);

          printf("Type an integer, EOF to quit\n");
     }
}
```

Figure 4.25: Revised Code for Reading Integers

```
23rt        34
n = 23
Type an integer, EOF to quit
debug:r in input stream, discarding
Type an integer, EOF to quit
debug:t in input stream, discarding
Type an integer, EOF to quit
n = 34
Type an integer, EOF to quit
^D
```

The input contains several characters that do not belong in numeric data. Each of these is discarded in turn and another attempt is made to read an integer. If unable to read an integer, another character is discarded. This continues until it is possible to read an integer or the end of file is reached.

Even if the user types data as requested, other problems can occur with `scanf()`. The second problem occurs when an attempt is made to read a character after reading a numeric data item. Figure 4.26 shows an example which reads an integer and then asks the user if he/she wishes to continue. If the user types 'y', the next integer is read; otherwise, the loop is terminated. This program produces the following sample session:

```
***Numeric and Character Data***

Type an integer
23\n
n = 23
Do you wish to continue?  (Y/N): debug:
in input stream
```

The sample session shows that an integer input is read correctly and printed; the prompt to the user is then printed, but the program does not wait for the user to type the response. A newline is printed as the next character read, and the program terminates. The reason is that when the user types the integer followed by a RETURN, the digit characters followed by the terminating newline are placed in the keyboard buffer (we have shown the \n explicitly). The function `scanf()` reads the integer until it reaches the newline character, but leaves the newline in the buffer. This newline character is then read as the next input character into `c`. Its value is printed and the loop is terminated since the character read is not `'y'`.

A simple solution is to discard a single delimiting white space character after the numeric data is read. C provides a suppression conversion specifier that will read a data item of any type and discard it. Here are some examples:

```
scanf("%*c");            /* read and discard a character */
```

```
/*   File: mix0.c
     This program shows problems reading character data when it follows
     numeric data.
*/
#include <stdio.h>

#define  DEBUG

main()
{    char ch;
     int flag, n;

     printf("***Numeric and Character Data***\n\n");
     printf("Type an integer\n");

     while ((flag = scanf("%d", &n)) != EOF) {       /* continue until EOF */
          printf("n = %d\n", n);                     /* print n */

          printf("Do you wish to continue? (Y/N): ");  /* prompt */
          scanf("%c", &ch);                            /* read a character, */
#ifdef DEBUG
printf("debug:%c in input stream\n", ch);             /* type its value */
#endif

          if (ch == 'y')                               /* if char is 'y' */
               printf("Type an integer\n");            /* prompt */
          else                                         /* otherwise, */
               break;                                  /* terminate loop */
     }
}
```

Figure 4.26: Mixing Numeric and Character Data

```
scanf("%*d");            /* read and discard an integer */
scanf("%d%*c", &n);      /* read an integer and store it in n, */
                         /* then read and discard a character */
scanf("%*c%c", &ch);     /* read and discard a character, */
                         /* and read another, store it in ch, */
```

Figure 4.27 shows the revised program that discards one character after it reads an integer.

This program produces the following sample session:

```
***Numeric and Character Data***

Type an integer
23\n
n = 23
Do you wish to continue?  (Y/N): y\n
debug:y in input stream
Type an integer
34       \n
n = 34
Do you wish to continue?  (Y/N): debug:    in input stream
```

We have shown the terminating newline explicitly in the sample session input. The first integer is read and printed; one character is discarded and the next one read correctly as `'y'` and the loop repeats. The next integer is typed followed by some white space and then a newline. The character after the integer is a space which is discarded and the following character is read. The new character read is another space, and the program is terminated because it is not a `'y'`.

The solution is to flush the entire line of white space until a newline is reached. Then the next character should be the correct response. The revised program is shown in Figure 4.28 and the sample session is below:

```
***Numeric and Character Data***

Type an integer
23       \n
n = 23
Do you wish to continue?  (Y/N): y      \n
debug:y in input stream
Type an integer
34       \n
n = 34
Do you wish to continue?  (Y/N): n      \n
debug:n in input stream
```

```
/*   File: mix1.c
     This program shows how character data might be read correctly when it
     follows numeric data. It assumes only one white space character
     terminates numeric data. This character is suppressed.
*/
#include <stdio.h>

#define  DEBUG

main()
{    char ch;
     int flag, n;

     printf("***Numeric and Character Data***\n\n");
     printf("Type an integer\n");

     while ((flag = scanf("%d", &n)) != EOF) {
          printf("n = %d\n", n);

          printf("Do you wish to continue? (Y/N): ");
          scanf("%*c%c", &ch);     /* suppress a character, read another */
#ifdef DEBUG
printf("debug:%c in input stream\n", ch);
#endif

          if (ch == 'y')
               printf("Type an integer\n");
          else
               break;
     }
}
```

Figure 4.27: Revised Code for Mixing Data

```
/*   File: mix2.c
     This program shows how character data can be read correctly when it
     follows numeric data even if several white space characters follow
     numeric data.
*/
#include <stdio.h>

#define DEBUG

main()
{    char ch;
     int flag, n;

     printf("***Numeric and Character Data***\n\n");
     printf("Type an integer\n");

     while ((flag = scanf("%d", &n)) != EOF) {
          printf("n = %d\n", n);

          /* flush white space characters in a line; stop when newline read */
          while (getchar() != '\n');

          printf("Do you wish to continue? (Y/N): ");
          scanf("%c", &ch);
#ifdef DEBUG
printf("debug:%c in input stream\n", ch);
#endif

          if (ch == 'y')
               printf("Type an integer\n");
          else
               break;
     }
}
```

Figure 4.28: A Better Revision for Mixing Data

The first integer is read and printed, the keyboard buffer is flushed of all white space until the newline is read, and the next character is read to decide whether to continue or terminate the loop. The next character input is also terminated with white space; however, the next item to be read is a number and all leading white space will be skipped.

A final alternative might be to terminate the program only when the user types an `'n'`; accepting any other character as a `'y'`. This would be a little more forgiving of user errors in responding to the program. One should also be prepared for mistyping of numeric data as discussed above. A programmer should anticipate as many problems as possible, and should assume that a user may not be knowledgeable about things such as `EOF` keystrokes, will be apt to make mistakes, and will be easily frustrated with rigid programs.

4.5 Menu Driven Programs

Finally, we end this chapter by using what we have learned to improve the user interface to programs: we consider the case of a program driven by a *menu*. In a **menu driven program**, the user is given a set of choices of things to do (the menu) and then is asked to select a menu item. The driver then calls an appropriate function to perform the task selected by the menu item. A `switch` statement seems a natural one for handling the selection from the menu.

We will modify the simple version of our payroll program to make it menu driven. While a menu is not needed in this case, we use it to illustrate the concept. The menu items are: *get data*, *display data*, *modify data*, *calculate pay*, *print pay*, *help*, and *quit the program*. The user selects a menu item to execute a particular path; for example, new data is read only when the user selects the menu item, *get data*. On demand, the current data can be displayed so the user may make any desired changes. Pay is calculated only when the user is satisfied with the data.

Figure 4.29 shows the driver for this program. (The driver of any menu driven program will look similar to this). The program prints the menu and then reads a selection character. A `switch` is used to select the path desired by the user. The user may type a lower or an upper case letter; both cases are included by the `case` labels. Usually, the driver hides the details of processing individual selections, so we have implemented most selections as function calls. The only exception here is when the selection is *get data* where the actual statements to read the necessary data are included in the driver itself because to use a function, it would have to read several items and somehow return them. So far we only know how to write functions that return a single value. We will address this matter in Chapter 6.

Notice what happens if the user elects to quit the program: a standard library function, `exit()`, is called. This function is like a `return` statement, except that it terminates the entire program rather than return from a function. It may be passed a value which is returned to the *environment* in which the program runs. A value of 0 usually implies normal termination of a program; any other value implies abnormal termination.

After the appropriate function is called, we terminate the selected case with a `break` statement to end the `switch` statement . The control then passes to the statement after the `switch` state-

```
/*   File: menu.c
     An example of a menu driven program. The main() driver prints the menu,
     reads the selected item, and performs an appropriate task.  */
#include <stdio.h>
#include "payroll.h"

main()
{    signed char c;
     int id;
     float hours_worked, rate_of_pay, pay;

     printf("***Pay Calculation: Menu Driven***\n\n"); /* print title */
     print_menu();        /* Display the menu to the user             */
     while ((c = getchar()) != EOF) {        /* get user selection    */
          switch(c) {                 /* select an appropriate path   */
               case 'g':       /* should be a function get_data()     */
               case 'G': printf("Id number: ");
                         scanf("%d", &id);
                         printf("Type Hours worked and rate of pay\n");
                         scanf("%f %f", &hours_worked, &rate_of_pay);
                         break;
               case 'd':
               case 'D': display_data(id, hours_worked, rate_of_pay);
                         break;
               case 'm':
               case 'M': modify_data();
                         break;
               case 'c':
               case 'C': pay = calc_pay(hours_worked, rate_of_pay);
                         break;
               case 'p':
               case 'P': display_data(id, hours_worked, rate_of_pay);
                         print_pay(pay);
                         break;
               case 'h':
               case 'H': print_menu();
                         break;
               case 'q':
               case 'Q': exit(0);
               default:  printf("Invalid selection\n");
                         print_menu();
          } /* end of switch */
          while ((c = getchar()) != '\n');    /* flush the buffer */
     } /* end of while loop */
} /* end of program */
```

Figure 4.29: Code for menu driven program

ment, namely flushing the buffer. Let us see what would happen if this flush were not present. The user selects an item by typing a character and must terminate the input with a newline. The keyboard buffer will retain all characters typed by the user, including the newline. So if the user types:

$d\backslash n$

(showing the newline explicitly), the program would read the character, `'d'`, select the appropriate case in the `switch` statement and execute the path which displays data. When the `break` ends the `switch`, control returns to the `while` expression which reads the next character in the buffer: the newline. Since newline is not one of the listed cases, the `switch` will choose the `default` case and print an error message to the user. Thus, flushing the keyboard buffer always obtains a new selection. In fact, even if the user typed more than a single character to select a menu item (such as an entire word), the buffer will be flushed of all remaining characters after the first.

As we have mentioned before, a large program should be developed incrementally, i.e. in small steps. The overall program logic consisting of major sub-tasks is designed first without the need to know the details of how these sub-tasks will be performed. Menu driven programs are particularly well suited for incremental development. Once the driver is written, "dummy" functions (sometimes called **stubs**) can be written for each task which may do nothing but print a debug message to the screen. Then each sub-task is implemented and tested one at a time. Only after some of the basic sub-tasks are implemented and tested, should others be implemented. At any given time during program development, many sub-task functions may not yet be implemented. For example, we may first implement only *get data*, *print data*, and *help* (*help* is easy to implement; it just prints the menu). Other sub-tasks may be delayed for later implementation. Figure 4.30 shows example implementations of the functions used in the above driver. These are in skeleton form and can be modified as needed without changing the program driver. It should be noted that the linker will require that *all* functions used in the driver be defined. The stubs satisfy the linker without having to write the complete function until later.

The use of a menu in this example is not very practical. It is merely for illustration of the technique. The menu is normally printed only once, so if the user forgets the menu items, he/she may ask for help, in which case the menu is printed again. Also, if the user types any erroneous character, the default case prints an appropriate message and prints the menu.

4.6 Common Errors

1. Errors in program logic: The program does not produce the expected results during testing. Use conditional compilation to introduce debug statements.

2. The value of `getchar()` is assigned to a `char` type. It should be assigned to a `signed char` type if it is to be checked for a possibly negative value of `EOF`.

3. The keyboard buffer is not flushed of erroneous or unnecessary characters as explained in Section 4.4.

```
/* File: payroll.c  */
/* Prints the menu. */
void print_menu(void)
{    /* print the menu */
     printf("Select:\n");
     printf("\tG(et Data\n");
     printf("\tD(isplay Data\n");
     printf("\tM(odify Data\n");
     printf("\tC(alculate Pay\n");
     printf("\tP(rint Pay\n");
     printf("\tH(elp\n");
     printf("\tQ(uit\n");
}

/* Displays input data, Id number, hours worked, and rate of pay. */
void display_data(int id, float hrs, float rate)
{
     printf("Id Number %d\n", id);
     printf("Hours worked %f\n", hrs);
     printf("Rate of pay %f\n", rate);
}

/* Calculates pay as hrs * rate */
/*  a very simple version of calc_pay.  Out previous implementation
    could be used here instead.
*/
float calc_pay(float hrs, float rate)
{
     return hrs * rate;
}

/* Modifies input data. */
void modify_data(void)
{
     printf("Modify Data not implemented yet\n");
}

/* Prints pay */
void print_pay(float pay)
{
     printf("Total pay = %f\n", pay);
}
```

Figure 4.30: Menu Driven Functions

4. Improper use of relational operators:

```
if ('a' <= ch <= 'z')    /* should be ('a' <= ch && ch <= 'z') */
     ...
```

   The operators are evaluated left to right: `'a' <= ch` is either True or False, i.e. 1 or 0. This value is compared with `'z'` and the result is always True.

5. An attempt is made to read past the end of the input file. If the standard input is the keyboard, it may or may not be possible to read input once the end of file keystroke is pressed. If the standard input is redirected, it is NOT possible to read beyond the end of file.

6. A `break` statement is not used in a `switch` statement. When a case expression matches the switch expression, control passes to that case label and control flow continues until the end of the `switch` statement. The only way to terminate the flow is with a `break` statement. Here is an example:

```
char find_next(char c)
{    char next;

     switch(c) {
          case 'z': next = 'a';
          default: next = c + 1;
     }
     return next;
}
```

   Suppose `c` is `'z'`. The variable `next` is assigned an `'a'` and control passes to the next statement which assigns `c + 1` to `next`. In fact, the function always returns `c + 1` no matter what `c` is.

7. Errors in defining macros. Define macros carefully with parentheses around macro formal parameters. If the actual argument in a macro call is an expression, it will be expanded correctly only if the macro is defined with parentheses around formal parameters.

8. A header file is not included in each of the source files that use the prototypes and/or macros defined in it.

9. Repeated inclusion of a header file in a source file. If the header file contains defines, there is no harm done. BUT, if the header file contains function prototypes, repeated inclusion is an attempt to redeclare functions, a compiler error.

10. Failure to set environment parameters, such as the standard include file directory, standard library directory, and so forth. Most systems may already have the environment properly set, but that may not be true in personal computers. If necessary, make sure the environment is set correctly. Also, make sure that the compile and link commands correctly specify all the source files.

4.7 Summary

In this chapter we have introduced a new data type, `char`, used to represent textual data in the computer. Characters are represented using a standard encoding, or assignment of a bit pattern to each character in the set. This encoding is called **ASCII** and includes representations of several classes of characters such as alphabetic characters (letters, both upper and lower case), digit characters, punctuation, space, other special symbols, and control characters. We have seen how character variables can be declared using the `char` keyword as the type specifier in a declaration statement, and how character constants are expressed in the program, namely by enclosing them in single quotes, e.g. `'a'`. The ASCII value of a character can be treated as an integer value, so we can do arithmetic operations using character variables and constants. For example, we have discussed how characters can be tested using relational operators to determine their class, how characters can be converted, for example from upper to lower case, or from a digit to its corresponding integer value.

We have also discussed character Input/Output using `scanf()` and `printf()` with the `%c` conversion specifier, or the `getchar()` and `putchar()` routines defined in `stdio.h`. We have used these routines and operations to write several example programs for processing characters and discussed the organization of program code into separate source files. This later technique allows us to develop our own libraries of utility functions which can be linked to various programs, further supporting our modular programming style.

In this chapter we have also introduced several new control constructs available in the C language. These include the `switch` statement:

switch (<expression>) <statement>

where the <statement> is usually a compound statement with case labels.

switch (<expression>) {
    case $<e_1>$: $<stmt_1>$
    case $<e_2>$: $<stmt_2>$
    ...
    case $<e_{n-1}>$: $<stmt_{n-1}>$
    default: $<stmt_n>$
}

The semantics of this statement are that the <expression> is evaluated to an integer type value and the `case` labels are searched for the first label that matches this value. If no match is found, the optional `default` label is considered to match any value. Control flow transfers to the statement associated with this label and proceeds to successive statements in the switch body. We can control which statements are executed further by using `return` or `break` statements with the `switch` body.

The syntax of the `break` statement is simply:

```
break;
```

and it may be used only within `switch` or loop bodies with the semantics of immediately terminating the execution of the body. In loops, the `break` statement is best used to terminate a loop under unusual or error conditions. A similar control construct available for loops is the `continue` statement:

```
continue;
```

which immediately terminates the current *iteration* of the loop but returns to the loop condition test to determine if the loop body is to be executed again.

We have also discussed some of the difficulties that can be encountered when mixing numeric and character data on input. These difficulties are due to the fact that numeric conversion specifiers (`%d` or `%f`) are "tolerant" of white space, i.e. will skip leading white space in the input buffer to find numeric characters to be read and converted, while character input (using `%c` or `getchar()`) is not. For character input, the next character, whatever it is, is read. In addition, numeric conversions will stop at the first non-numeric character detected in the input, leaving it in the buffer. We have shown several ways of handling this behavior to make the input tolerant of user errors in Section 4.4.

Finally, we used the features of the language discussed in this chapter to implement a common style of user interface: menu driven programs. Such a style of program also facilitates good top down, modular design in the coding and testing of our programs.

4.8 Exercises

1. What is the value of each of the following expressions:

```
ch = 'd';

(a)   ((ch >= 'a') && (ch <= 'z'))
(b)   ((ch > 'A') && (ch < 'Z'))
(c)   ((ch >= 'A') && (ch <= 'Z'))
(d)   ch = ch -'a' + 'A';
(e)   ch = ch - 'A' + 'a';
```

2. What will be the output of the following:

```
char ch;
int d;

ch = 'd';
d = 65;
printf("ch = %c, value = %d\n", ch, ch);
printf("d = %d, d = %c\n", d, d);
```

3. Write the header file `category.h` discussed in section 4.1.2. Write the macros `IS_UPPER()`, `IS_DIGIT()`, `IS_PUNCT()`, `IS_SPACE()`, `IS_CONTROL()`.

4. Write a code fragment to test:

   - if a character is printable but not alphabetic
   - if a character is alphabetic but not above 'M' or 'm'
   - if a character is printable but not a digit

5. Write separate loops to print out the ASCII characters and their values in the ranges:

```
'a' to 'z',
'A' to 'Z',
'0' to '9'.
```

6. Are these the same: 'a' and "a"? What is the difference between them?

7. What will be the output of the source code:

```
#define SQ(x) ((x) * (x))
#define CUBE(x)  ((x) * (x) * (x))
#define DIGITP(c) ((c) >= '0' && (c) <= '9')

char c = '3';
```

```
if (DIGITP(c))
     printf("%d\n", CUBE(c - '0'));
else
     printf("%d\n", SQ(c - '0'));
```

8. Find the errors in the following code that was written to read characters until end of file.

```
char c;

while (c = getchar())
     putchar(c);
```

9. What will be the output of the following program?

```
#include <stdio.h>
main()
{    int n, sum;
     char ch;

     ch = 'Y';
     sum = 0;
     scanf("%d", &n);
     while (ch != 'N') {
          sum = sum + n;
          printf("More numbers? (Y/N) ");
          scanf("%c", &ch);
          scanf("%d", &n);
     }
}
```

10. What happens if `scanf()` is in a loop to read integers and a letter is typed?

11. What happens if `scanf()` reads an integer and then attempts to read a character?

12. Use a switch statement to test if a digit symbol is an even digit symbol.

13. Write a single loop that reads and prints all integers as long as they are between 1 and 100 with the following restrictions: If an input integer is divisible by 7 terminate the loop with a `break` statement; if an input integer is divisible by 6, do not print it but continue the loop with a `continue` statement.

4.9 Problems

1. First use graph paper to plan out and then write a program that prints the following message centered within a box whose borders are made up of the character *.

```
Happy New Year
```

2. Write a program to print a character corresponding to an ASCII value or vice versa, as specified by the user, until the user quits. If the character is not printable, print a message.

3. Write a function that takes one character argument and returns the following: if the argument is a letter, it returns the position of the letter in the alphabet; otherwise, it returns `FAIL`, whose value is -1. For example, if the argument is `'A'`, it returns 0; if the argument is `'d'`, it returns 3, and so forth. Define and use macros to test if a character is a lower case letter or an upper case letter.

4. Use a switch statement to write a function that returns `TRUE` if a character is a consonant and returns `FALSE` otherwise.

5. Use a switch statement to write a function that returns `TRUE` if a digit character represents an odd digit value. If the character is not an odd digit, the function returns `FALSE`.

6. Write a program to count the occurrence of a specified character in the input stream.

7. Write a program that reads in characters until end of file. The program should count and print the number of characters, printable characters, vowels, digits, and consonants in the input. Use functions to check whether a character is a vowel, a consonant, or a printable character. Define and use macros to test if a character is a digit or a letter.

8. Modify the program in Chapter 2 to find prime numbers so that the inner loop is terminated by a `break` statement when a number is found not to be prime.

9. Write a function that takes two arguments, `replicate(int n, char c);`, and prints the character, `c`, a number, `n`, times.

10. Use `replicate()` to write a function, `drawrect()`, that draws a rectangle of length, `g`, and width, `w`. The dimensions are in terms of character spaces. The rectangle top left corner is at top, `t`, and left, `l`. The arguments, `g`, `w`, `t`, and `l` are integers, where `t` and `l` determine the top left corner of the rectangle, and the length of the rectangle should be along the horizontal. Use `'*'` to draw your lines. Write a program that repeatedly draws rectangles until length and width specified by the user are both zero.

11. Repeat 10, but modify `drawrect()` to `fillrect()` that draws a rectangle filled in with a specified fill character.

12. Write a function that draws a horizontal line proportional to a specified integer between the values of 0 and 50. Use the function in a program to draw a bar chart, where the bars are horizontal and in proportion to a sequence of numbers read.

13. Write a function to encode text as follows:

    a. If the first character of a line is an upper case letter, then encode the first character to one that is 1 position higher in a circular alphabet. Move the rest of the characters in the line up by 1 position in a circular printable part of the ASCII character set.

    b. If the first character of a line is a lower case letter, then move the first character down by 2 positions in a circular alphabet. Move the rest of the characters in the line down by 2 positions in a circular printable part of the ASCII character set.

    c. If the first character of a line is white space, then terminate the input.

    d. Otherwise, if the first character of a line is not a letter, then move all characters in the line down by 1 position in a circular printable part of the ASCII character set.

14. Write a function to decode text that was encoded as per Problem 13.

15. Write a menu-driven program that combines Problems 13 and 14 to encode or decode text as required by the user. The input for encoding or decoding is terminated when the first character of a line is a space. The commands are: *encode*, *decode*, *help*, and *quit*.

16. Write a function that takes three arguments, two float numbers and one arithmetic operator character. It returns the result of applying the operator to the two numbers. Using the function, write a program that repeatedly reads a float number, followed by an arithmetic operator, followed by a float number; each time it prints out the result of applying the operator to the numbers.

17. Modify the program in Problem 16 to allow further inputs of a sequence of an operator followed by a number. Each new operator is to be applied to the result from the previous operation and the new number entered. The input is terminated by a newline. Print only the final result.

18. Read and convert a sequence of digits to its equivalent integer. Any leading white space should be skipped. The conversion should include digit characters until a non-digit character is encountered. Modify the program so it can read and convert a sequence of digit characters preceded by a sign, `+` or `-`.

19. Write a program that converts the input sequence of digit characters, possibly followed by a decimal point, followed by a sequence of digits, to a float number. The leading white space is skipped and the input is terminated when a character not admissible in a float number is encountered.

20. Modify the above program to include a possible leading sign character.

21. Write a function that takes a possibly signed integer as an argument, and converts it to a sequence of characters.

22. Write a program that takes a possibly signed floating point number and converts it to a sequence of characters with 4 digits after the decimal point.

23. Modify the word extraction program, `wds.c`, in Figure 4.16. It should count words with exactly four characters and words with five characters. Assume the input consists of only valid characters and white space.

24. Write a program that reads in characters until end of file. The program should identify each *token*, i.e. a word after skipping white space. The only valid token types are: *integer* and *invalid*. White space delimits words but is otherwise ignored. An integer token is a word that starts with a digit and is followed by digits and terminates when a non-digit character is encountered. An invalid token is made up of any other single character that does not belong to an integer. Print each token as it is encountered as well as its type. Here is a sample session:

```
Type text, EOF to quit:  3456 a23b
3456 integer
a invalid
23 integer
b invalid
Type text, EOF to quit:  ^D
```

25. Modify the program in Problem 24 so it also allows an identifier as a valid token. An identifier starts with a letter and may be followed by a sequence of letters and/or digits.

26. Modify the program in Problem 25 so that tokens representing float numbers are also allowed. A float token must start with a digit, may be followed by a sequence of digits, followed by a decimal point, followed by zero or more digits. Here is a sample session:

```
Type text, EOF to quit:  The ID Number is 123, not 123.
The Identifier
ID Identifier
Number Identifier
is Identifier
123 Integer
, Invalid
not Identifier
123.  Float

Type text, EOF to quit:  pay = 1.5 * hours * rate;
pay Identifier
= Invalid
1.5 Float
* Invalid
hours Identifier
* Invalid
rate Identifier
; Invalid
Type text, EOF to quit:  ^D
```

Hint: Skip leading delimiters; test the first non-delimiter, and build a word of the appropriate type. An integer and a float are distinguished by the presence of a decimal point.