According to most C literature, the valid range for floats is 10^{-38} to 10^{38}. But, how is such an odd range used? Well, the answer lies in the IEEE representation. Since the exponent of a float in IEEE format is stored with a positive bias of 127, the smallest positive value that can be stored in a float variable is 2^{-127}, which is approximately 1.175×10^{-38}. The largest positive value is 2^{128}, which is about 3.4×10^{38}. Similarly, for a double variable the smallest possible value is 2^{-1023}, which is approximately 2.23×10^{-308}. The largest positive value that can be held in a double variable is 2^{1024}, which is approximately 1.8×10^{308}.

There is one more quirk. After obtaining the IEEE format for a float, when the time comes to actually store it in memory, it is stored in the reverse order. That is, if the user calls the four-byte IEEE form as ABCD, then while storing in memory it is stored in the form DCBA. This can be understood with an example. Suppose the floating point number in question is 5.375. Its IEEE representation is 01000000101011000000000000000 0000. Expressed in Hex this is 40 AC 00 00. While storing this in memory, it is stored as 00 00 AC 40.

The representation of a long double (10-byte entity) is also similar. The only difference is that unlike float and double, the most significant bit of the normalized form is specifically stored. In a long double, 1 bit is occupied by the sign, 15 bits by the biased exponent (bias value 16383), and 64 bits by the mantissa.

2.12 TOKEN

Tokens are the basic lexical building blocks of source code. In other words, one or more symbols understood by the compiler that help it interpret your code. Characters are combined into tokens according to the rules of the programming language. The compiler checks that the tokens can be formed into legal strings according to the syntax of the language.There are five classes of tokens: *identifiers, reserved words, operators, separators,* and *constants*.

An ***identifier*** is a sequence of characters invented by the programmer to identify or name a specific object and name is formed by a sequence of letters, digits, and underscores.

Keywords are explicitly reserved words that have a strict meaning as individual tokens to the compiler. They cannot be redefined or used in other contexts. Use of variable names with the same name as any of the keywords will cause a compiler error.

Operators are tokens used to indicate an action to be taken (usually arithmetic operations, logical operations, bit operations, and assignment operations). Operators can be simple operators (a single character token) or compound operators (two or more character tokens).

Separators are tokens used to separate other tokens. Two common kinds of separators are indicators of an end of an instruction and separators used for grouping.

A ***constant*** is an entity that doesn't change.

Say we have the following piece of code,

```
if(x<5)

   x = x + 2;
else
  x = x + 10;
```

Here the tokens that will be generated are

```
Keywords   : if , else
Identifier : x
Constants  : 2, 10,5
Operators  : +,=
Separator  : ;
```

2.12.1 Identifier

An identifier or name is a sequence of characters invented by the programmer to identify or name a specific object. In C, variables, arrays, functions, and labels are named. Describing them may help to learn something about the character of the language since they are elements that C permits the programmer to define and manipulate. Some rules must be kept in mind when naming identifiers. These are stated as follows.

1. The first character must be an alphabetic character (lower-case or capital letters) or an underscore '_'.
2. All characters must be alphabetic characters, digits, or underscores.
3. The first 31 characters of the identifier are significant. Identifiers that share the same first 31 characters may be indistinguishable from each other.
4. Cannot duplicate a key word. A keyword word is one which has special meaning to C.

Some examples of proper identifiers are employee_number, box_4_weight, monthly_pay, interest_per_annum, job_number, and tool_4.

Some examples of incorrect identifiers are 230_item, #pulse_rate, total~amount, /profit margin, and ~cost_per_item.

2.12.2 Keywords

Keywords are the vocabulary of C. Because they are special to C, one can't use them for variable names.

There are 32 words defined as keywords in C. These have predefined uses and cannot be used for any other purpose in a C program. They are used by the compiler to compile the program. They are always written in lowercase letters. A complete list of these keywords is given in Table 2.7.

Table 2.7 Keywords in C

auto	double	int	struct
break	else	long	witch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Several keywords were added in C89: *const, enum, signed, void* and *volatile*. New in C99 are the keywords *inline, restrict, _Bool, _Complex* and *_Imaginary*.

Table 2.8 Full set of keywords upto C99

auto	enum	restrict	unsigned
break	extern	return	void
case	float	short	volatile
char	for	signed	while
const	goto	sizeof	_Bool
continue	if	static	_Complex
default	inline	struct	_Imaginary
do	int	switch	
double	long	typedef	
else	register	union	

Note that compiler vendors (like Microsoft, Borland, etc.) provide their own keywords apart from the ones mentioned above. These include extended keywords like **near**, **far**, **asm**, etc. Though it has been suggested by the ANSI committee that every such compiler specific keyword should be preceded by two underscores (as in **__asm**), not every vendor follows this rule.

2.12.3 Constant

A constant is an explicit data value written by the programmer. Thus, it is a value known to the compiler at compiling time. The compiler may deal with this value in any of several ways, depending on the type of constant and its context. For example, the binary equivalent of the constant may be inserted directly into the output code stream. The value of the constant may be stored in a special data area in memory. The compiler may decide to use the constant's value for its own immediate purpose, e.g., to determine how much storage it should allocate to a data array.

C permits integer constants, floating-point constants, character constants, and string constants. Figure 2.16 depicts the types of constants that C allows. An integer constant consists of a sequence of digits. It is normally interpreted as a decimal value. Thus, 1, 25, and 23456 are all decimal integer constants.

A literal integer (e.g., 1984) is always assumed to be of type `int`, unless it has an '`L`' or '`l`' suffix, in which case it is treated as a `long`. Also, a literal integer can be specified to be unsigned using the suffix `U` or `u`. For example,

```
1984L  1984l  1984U  1984u  1984LU  1984ul
```

Literal integers can be expressed in decimal, octal, and hexadecimal notations. The decimal notation is the one that has been used so far. An integer is taken to be octal if it is preceded by a zero (`0`), and hexadecimal if it is preceded by a `0x` or `0X`. For example,

```
92    /* decimal         */
0134  /* equivalent octal   */
0x5C  /* equivalent hexadecimal */
```

Points to Note

In ANSI C, a decimal integer constant is treated as an `unsigned long` if its magnitude exceeds that of the `signed long`. An octal or hexadecimal integer that exceeds the limit of `int` is taken to be `unsigned`; if it exceeds this limit, it is taken to be long; and if it exceeds this limit, it is treated as an `unsigned long`. An integer constant is regarded as `unsigned` if its value is followed by the letter '`u`' or '`U`', e.g., `0x9999u`; it is regarded as `unsigned long` if its value is followed by '`u`' or '`U`' and '`l`' or '`L`', e.g., `0xFFFFFFFFul`.

A floating-point constant consists of an integer part, a decimal point, a fractional part, and an exponent field containing an `e` or `E` followed by an integer. Both integer

and fractional parts are digit sequences. Certain portions of this format may be missing as long as the resulting number is distinguishable from a simple integer. For example, either the decimal point or the fractional part, but not both, may be absent. A literal real (e.g., 0.06) is always assumed to be of type `double`, unless it has an '`F`' or '`f`' suffix, in which case it is treated as a `float`, or an '`L`' or '`l`' suffix, in which case it is treated as a `long double`. The latter uses more bytes than a `double` for better accuracy (e.g., 10 bytes on the programmer's PC). For example,

```
0.06F   0.06f   3.141592654L    3.1415926541
```

In addition to the decimal notation used so far, literal reals may also be expressed in *scientific* notation. For example, 0.002164 may be written in scientific notation as

```
2.164E-3    or   2.164e-3
```

The letter `E` (or `e`) stands for *exponent*. The scientific notation is interpreted as follows.

$2.164E\text{-}3 = 2.164 \times 10^{-3}$

The following are examples of *long long*:

```
12345LL
12345ll
```

The following are examples of *unsigned long long*:

```
123456ULL
123456ull
```

A character constant normally consists of a single character enclosed in single quotes. Thus, for example, `'b'` and `'$'` are both character constants. Each takes on the numeric value of its character in the machine's character set. Unless stated otherwise, it will henceforth be assumed that the ASCII code is used. This table is provided in Appendix A. Thus, for example, writing down the character constant 'A' is equivalent to writing down the hex value 41 or the octal value 101. The 'A' form is preferable, of course, first, because its meaning is unmistakable, and second, because it is independent of the actual character set of the machine.

In C, certain special characters, in particular, non-printing control characters are represented by special, so-called escape character sequences, each of which begins with the special backslash (\) escape character. Most of these escape codes are designed to make visible, on paper, any of those characters whose receipt by a printer or terminal causes a special, non-printing control action.

Character constants can also be defined via their octal ASCII codes. The octal value of the character, which may be found from the table in Appendix A, is preceded by a backslash and enclosed in single quotes.

```
char terminal_bell = '\07';
                /* 7 = octal ASCII code for beep */

char backspace = '\010';
                /* 10 = octal code for backspace */
```

For ANSI C compilers, character constants may be defined by hex digits instead of octals. Hex digits are preceded by `x`, unlike `0` in the case of octals. Thus, in ANSI C

```
char backspace = '\xA';
```

is an acceptable alternative declaration to

```
char backspace = '\010';
```

Any number of digits may be written but the value stored is undefined if the resulting character value exceeds the limit of `char`.

On an ASCII machine both `'\b'` and `'\010'` are equivalent representations. Each will print the backspace character. But the latter form, the ASCII octal equivalent of `'\b'`, will not work on an EBCDIC machine, typically an IBM mainframe, where the collating sequence of the characters (i.e., their gradation or numerical ordering) is different. In the interests of portability it is therefore preferable to write `'\b'` for the backspace character rather than its octal code. Then the program will work as faultlessly on an EBCDIC machine as it will on an ASCII.

Note that the character constant '`a`' is not the same as the string `"a"`. A string is really an array of characters that is a bunch of characters stored in consecutive memory locations, the last location containing the null character; so the string `"a"` really contains two `chars`, an '`a`' immediately followed by '`\0`'. It is important to realize that the null character is not the same as the decimal digit 0, the ASCII value of which is 00110000.

A string constant is a sequence of characters enclosed in double quotes. Whenever the C compiler encounters a string constant, it stores the character sequence in an available data area in memory. It also records the address of the first character and appends to the stored sequence an additional character, the null character '`\0`', to mark the end of the string.

The length of a character string is the number of characters in it (again, excluding the surrounding double quotes). Thus, the string "`messagen`" has a length of eight. The actual number of stored characters is one more as a null character is added.

The characters of a string may be specified using any of the notations for specifying literal characters. For example,

```
"Name\tAddress\tTelephone" /* tab-separated words*/
"ASCII character 65: \101"/* 'A' specified as '101' */
```

A long string may extend beyond a single line, in which case each of the preceding lines should be terminated by a backslash. For example,

```
"Example to show \
the use of backslash for \
writing a long string"
```

The backslash in this context means that the rest of the string is continued on the next line. The preceding string is equivalent to the single-line string

```
"Example to show the use of backslash for writing
a long string"
```

Points to Note

A common programming error results from confusing a single-character string (e.g., "A") with a single character (e.g., 'A'). These two are *not* equivalent. The former consists of two bytes (the character 'A' followed by the character '\0'), whereas the latter consists of a single byte.

The shortest possible string is the null string (" "). It simply consists of the null character. Table 2.9 summarizes the different constants.

Table 2.9 Specifications of different constants

Type	Specification	Example
Decimal	nil	50
Hexadecimal	Preceded by 0x or 0X	0x10
Octal	Begins with 0	010
Floating constant	Ends with f/F	123.0f
Character	Enclosed within single quote	'A' 'o'
String	Enclosed within double quote	"welcome"
Unsigned integer	Ends with U/u	37 u
Long	Ends with L/l	37 L
Unsigned long	Ends with UL/w	37 UL

C89 added the suffixes **U** and **u** to specify unsigned numbers. C99 adds **LL** to specify **long long**.

More than one \n can be used within a string enabling multi-line output to be produced with a single use of the printf() function. Here's an example.

```
int main()
{
  printf("This sentence will \n be printed\nin\
    multi-line \n");
  return 0;
}
```

When the program was compiled and run it produced the following output.

```
This sentence will
be printed
in multi-line
```

However if the string is too long to fit on a single line then it is possible to spread a string over several lines by escaping the actual new-line character at the end of a line by preceding it with a backslash. The string may then be continued on the next line as shown in the following program:

```
int main()
{
  printf("hello,\
   world\n");
  return 0;
}
```

The output is

```
hello, world
```

The indenting spaces at the start of the string continuation being taken as part of the string. A better approach is to use *string concatenation* which means that two strings which are only separated by *whitespaces* are regarded by the compiler as a single string. Space, newline, tab character and comment are collectively known as *whitespace*. The use of string concatenation is shown by the following example.

```
int main()
{
  printf("hello," "world\n");
  return 0;
}
```

2.12.4 Assignment

In the example (i) above we have used a statement :

```
int a=2,b=3;
```

Here both a and b assigned a value.

The assignment operator is the single equal to sign (=).