Module 1

1. Character Set

Character set includes a set of valid characters we can use in our program in different environments. In C, the character set used to represent characters in source code is based on the ASCII (American Standard Code for Information Interchange) character set. The characters in C are grouped into the following categories:

1. Alphabetic Characters (A-Z, a-z): These represent both uppercase and lowercase English letters. For example, 'A' to 'Z' and 'a' to 'z'.
2. Digits (0-9): These represent numeric characters. For example, '0' to '9'.
3. Special Characters: These include various special characters used in programming and text processing. Some common special characters include:
   - Arithmetic Operators: +, -, *, /, %
   - Relational Operators: <, >, ==, !=, <=, >=
   - Logical Operators: &&, ||, !
   - Punctuation: ;, :, ,, ., ?, !
   - Brackets and Parentheses: (,), [,], {, }
   - Quotes: ', "
   - Backslash: \
   - Ampersand: &
   - Dollar Sign: $
   - Hash or Pound Sign: #
4. Whitespace Characters: These include space (' '), tab ('\t'), newline ('\n'), carriage return ('\r'), and form feed ('\f'). These characters are used for formatting and layout in code and text. The compiler ignores white spaces unless they are a part of a string constant. White spaces may be used to separate words but are prohibited between the characters of keywords and identifiers.
5. Escape Sequences: C allows you to use escape sequences to represent characters that are difficult to type or invisible. An escape sequence in the C programming language consists of a backslash (\) and a character that stands in for a special character or control sequence. During the compilation process, the C compiler substitutes any escape sequences it comes across with the relevant character or control sequence. It enables the use of difficult-to-represent characters like newlines, tabs, quotations, and backslashes.
   It is composed of two or more characters starting with backslash \. For example, '\n' represents a newline character, and '\t' represents a tab character. Table. 1.1 shows common escape sequences in C.

6. Extended Characters: C also includes characters beyond the ASCII set, especially for international character encoding. These include characters with accents, diacritics, and symbols from various languages.

Escape Sequence	Meaning
\n	New Line
\t	Horizontal Tab
\b	BackSpace
\r	Carriage Return
\a	Audible bell
\'	Printing single quotation
\"	printing double quotation
\?	Question Mark Sequence
\\	Back Slash
\f	Form Feed
\v	Vertical Tab
\0	Null Value
\nnn	Print octal value
\xhh	Print Hexadecimal value

Table.1.1. Escape sequences

2. Tokens

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a C program the smallest individual units are known as C tokens. C has six types of tokens as shown in Fig. 2.1. C programs are written using these tokens and the syntax of the language.

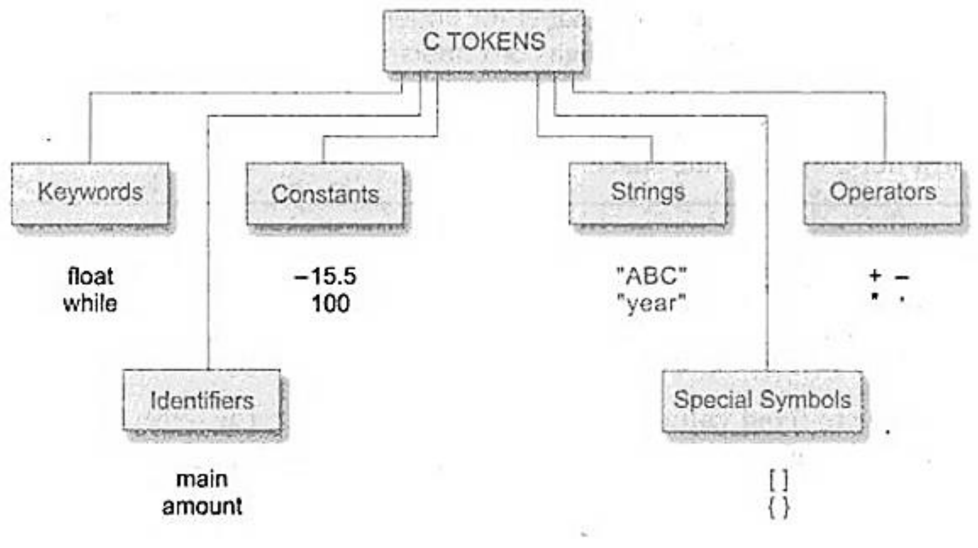
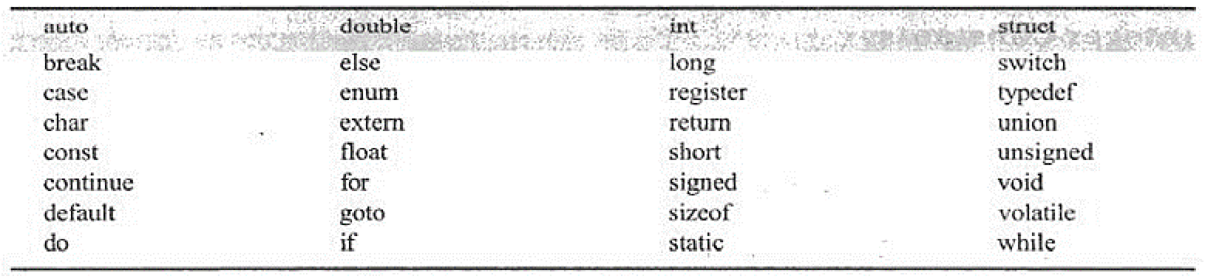


Fig.2.1. C tokens and examples

3. Keywords

Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings, and these meanings cannot be changed. Keywords serve as basic building blocks for program statements. The list of all keywords of C are listed in Table 3.1. All keywords must be written in lowercase.



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

Table 3.1: C keywords

4. Identifier

Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. They are used to uniquely identify the entity within the program. Both uppercase and lowercase letters are permitted, although lowercase letters are commonly used. The underscore character is also permitted in identifiers. It is usually used as a link between two words in long identifiers.

Rules for Identifiers

1. First character must be an alphabet (or underscore).
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword.
5. Must not contain white space.

Example of valid identifiers

- total
- sum
- average
- _m _
- sum_1 etc.

Example of invalid identifiers

- 2sum (starts with a numerical digit)
- **int** (reserved word)
- **char** (reserved word)
- m+n (special character, i.e., '+')

5. Data Types

C language is rich in its data types. It specifies the type of data that the variable can store like integer, character, floating, double, etc. Each data type requires different amounts of memory and has some specific operations which can be performed over it. The variety of data types available allow the programmer to select the type appropriate to the needs of the application as well as the machine.

C supports three classes of data types:

1. Primary (or fundamental) data types
2. Derived data types
3. User-defined data types

All C compilers support five fundamental data types, namely integer (int), character (char), floating point (float), double-precision floating point (double) and void. Many of them also offer extended data types such as long int and long double. Various data types and the terminology used to describe them are given in Fig. 5.1. The range of the basic four types are given in Table 5.1.

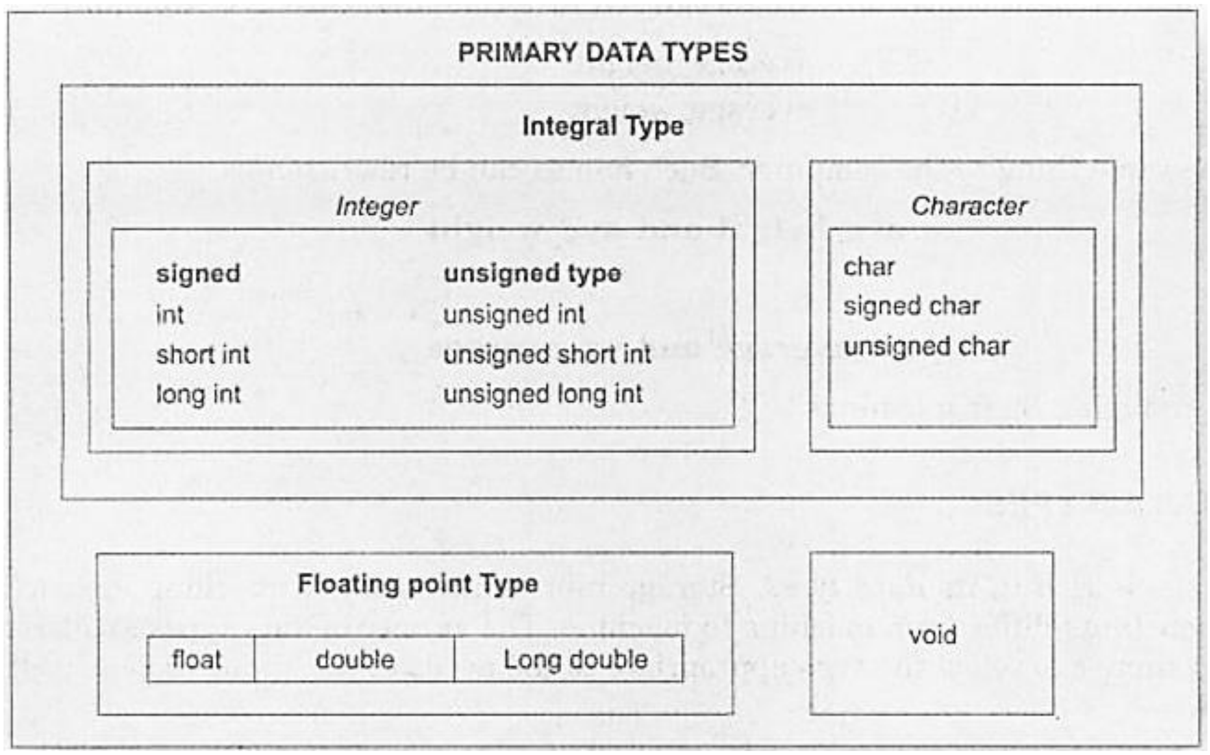
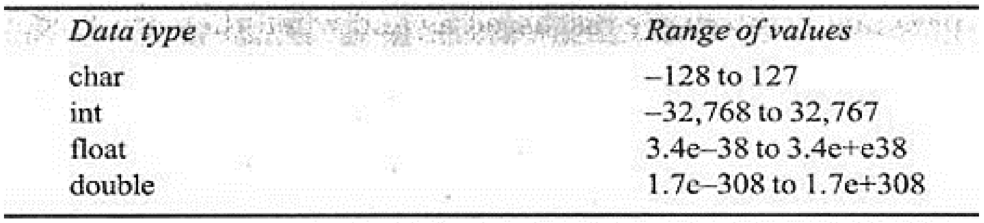


Fig 5.1. Primary data types in C



Data type	*Range of values*
char	–128 to 127
int	–32,768 to 32,767
float	3.4e–38 to 3.4e+e38
double	1.7e–308 to 1.7e+308

Table 5.1. Size and range of basic data types on 16-bit machines.

Derived data types are those datatypes which are derived from the primary or fundamental datatypes. The three derived data types in C are arrays, functions, and pointers.

- Array in C is a fixed-size collection of similar data items stored in contiguous memory locations. An array is capable of storing the collection of data of primitive, derived, and user-defined data types.
- A function is called a C language construct which consists of a function-body associated with a function-name. The function body gets executed when the function name is called within the program. In every program in C language, execution begins from the main function, which gets terminated after completing some operations which may include invoking other functions.
- A pointer in C language is a data type that stores the address where data is stored. Pointers store memory addresses of variables, functions, and even other pointers.

The data types that are defined by the user depending upon the use case of the programmer are termed **user-defined data types**. User-defined data types are created by combining the primary and derived datatypes. C offers a set of constructs that allow users to define their data types. To create a user-defined data type, the C programming language has the following constructs: structures (struct), union, type definitions (typedef), enumerations (enum).

- Structures are used to group items of different types into a single type. The "struct" keyword is used to define a structure.
- Unions are similar to structures in many ways except that all the members in the union are stored in the same memory location. Union is declared using the "union" keyword.
- Enumeration allows the user to create custom data types with a set of named integer constants. The "enum" keyword is used to declare an enumeration. It makes a program easy to read and maintain.
- Typedef is used to redefine the existing data type names. Basically, it is used to provide new names to the existing data types. The "typedef" keyword is used for this purpose.

5.1. Integer Types

The integer datatype in C is used to store the integer numbers (any number including positive, negative and zero without decimal part). Octal values, hexadecimal values, and decimal values can be stored in int data type in C. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. If we use a 16 bit word length, the size of the integer value is limited to the range-32768 to +32767 (that is, -215 to +215-1). A signed integer uses one bit for sign and 15 bits for the magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2,147,483,648 to 2,147,483,647.

In order to provide some control over the range of numbers and storage space, C has three classes of integer storage, namely **short int, int, and long int**, in both signed and unsigned forms. C defines these types so that they can be organized from the smallest to the largest, as shown in Fig. 5.1.1. For example, short int represents fairly small integer values and requires half the amount of storage as a regular int number uses. Unlike signed integers, unsigned integers use all the bits for the magnitude of the number and are always positive. Therefore, for a 16 bit machine, the range of unsigned integer numbers will be from 0 to 65,535.

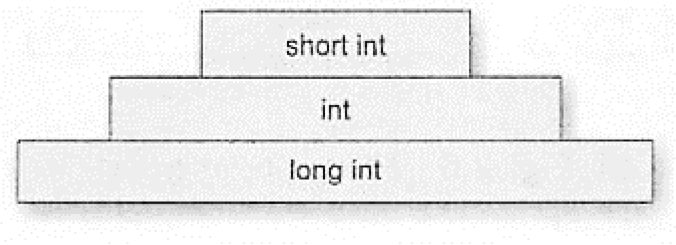


Fig 5.1.1. Integer Types

We declare long and unsigned integers to increase the range of values. The use of qualifier signed on integers is optional because the default declaration assumes a signed number. Table 5.1.2. shows all the allowed combinations of basic types and qualifiers and their size and range on a 16-bit machine.

Type	*Size (bits)*	*Range*
char or signed char	8	–128 to 127
unsigned char	8	0 to 255
int or signed int	16	–32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	–128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	–2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E – 38 to 3.4E + 38
double	64	1.7E – 308 to 1.7E + 308
long double	80	3.4E – 4932 to 1.1E + 4932

Table 5.1.2. Size and range of data types on a 16-bit machines.

5.2. Floating Point Types

Floating point (or real) numbers are stored in 32 bits (on all 16 bit and 32 bit machines), with 6 digits of precision. Floating point numbers are defined in C by the keyword **float**. When the accuracy provided by a float number is not sufficient, the type double can be used to define the number. A **double** data type number uses 64 bits giving a precision of 14 digits. These are known as double precision numbers. Double type represents the same data type that float represents, but with a greater precision. To extend the precision further, **long double** which uses 80 bits can be used. The relationship among floating types is illustrated in Fig. 5.2.1.

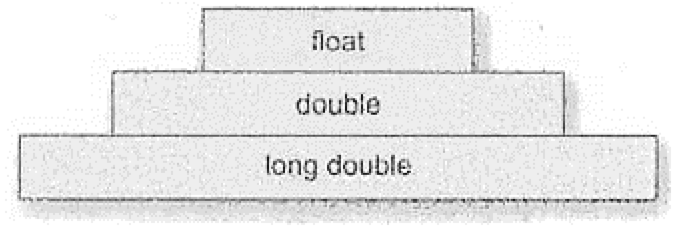


Fig 5.2.1. Floating-point types

5.3. Void Types

The void type has no values. This is usually used to specify the type of functions. The type of a function is said to be void when it does not return any value to the calling function. It can also play the role of a generic type, meaning that it can represent any of the other standard types.

Void is used in multiple ways as function return type, function arguments as void, and pointers to void.

- Function return type void
  void exit(int check);
- Function without any parameter can accept void.
  int print(void);
- Memory allocation function which returns a pointer to void.
  void *malloc (size_t size);

5.4. Character Types

A single character can be defined as a character(char) type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier signed or unsigned may be explicitly applied to char. While unsigned chars have values between 0 and 255, signed chars have values from-128 to 127.

6. Constants

Constants in C refer to fixed values that do not change during the execution of a program. C supports several types of constants as illustrated in Fig. 6.1.

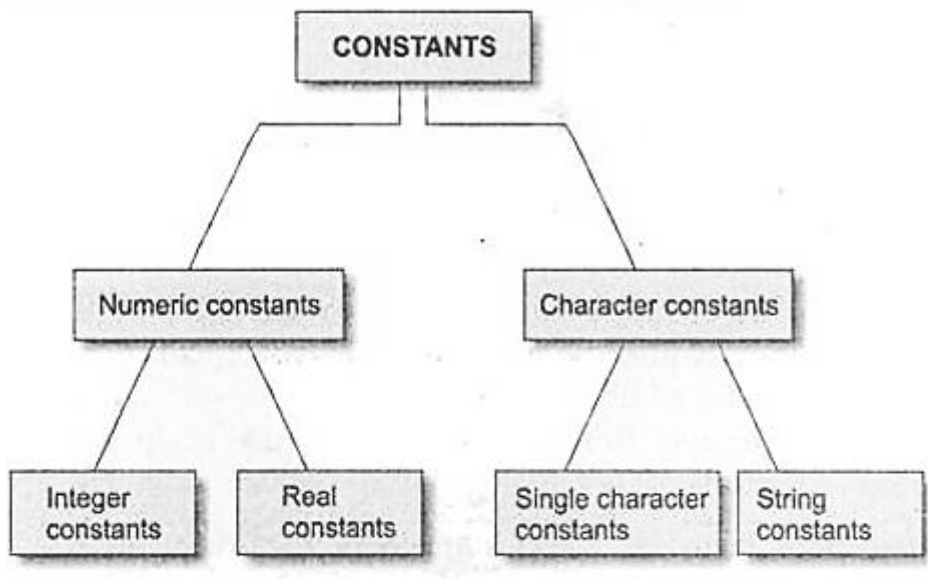


Figure 6.1. Basic types of C constants

6.1. Integer Constants

An integer constant refers to a sequence of digits. There are three types of integers, namely, decimal integer, octal integer and hexadecimal integer.

Decimal integers consist of a set of digits, 0 through 9, preceded by an optional - or + sign. Valid examples of decimal integer constants are:

123 -321 0 654321 +78

Embedded spaces, commas, and non-digit characters are not permitted between digits. For example,

15 750 20,000 $1000 are illegal numbers.

An octal integer constant consists of any combination of digits from the set 0 through 7, with a leading 0. Some examples of octal integer are:

037 0 0435 0551

A sequence of digits preceded by Ox or OX is considered as hexadecimal integer. They may also include alphabets A through F or a through f. The letter A through F represent the numbers 10 through 15. Following are the examples of valid hex integers:

0X2 0x9F 0Xbcd 0x

We rarely use octal and hexadecimal numbers in programming.

The largest integer value that can be stored is machine-dependent. It is 32767 on 16-bit machines and 2,147,483,647 on 32-bit machines. It is also possible to store larger integer constants on these machines by appending qualifiers such as U, L and UL to the constants.

Examples:

56789U or 56789u (unsigned integer)

98761234UL or 98761234ul (unsigned long integer)

9876543L or 98765431 (long integer)

The below program illustrates the use of integer constants on a 16-bit machine. The output shows that the integer values larger than 32767 are not properly stored on a 16-bit machine. However, when they are qualified as long integer (by appending L), the values are correctly stored.

Program

```
main()
{
printf("Integer values\n\n");
printf("%d %d %d\n", 32767,32767+1,32767+10);
printf("\n");
printf("Long integer values\n\n");
printf("%ld %ld %ld\n", 32767L,32767L+1L,32767L+10L);
}
```

Output

```
Integer values

32767  -32768   -32759

Long integer values

32767   32768    32777
```

6.2. Real Constants

Integer numbers are inadequate to represent quantities that vary continuously, such as distances, heights, temperatures, prices, and so on. These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called real (or floating point) constants. Further examples of real constants are:

0.0083 -0.75 435.36 +247.0

These numbers are shown in decimal notation, having a whole number followed by a decimal point and the fractional part. It is possible to omit digits before the decimal point, or digits after the decimal point. That is,

215 . 95 -.71 +.5

are all valid real numbers.

A real number may also be expressed in exponential (or scientific) notation. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10^2. The general form is:

mantissa e exponent

The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. The letter e separating the mantissa and the exponent can be written in either lowercase or uppercase. Since the exponent causes the decimal point to "float", this notation is said to represent a real number in floating point form. Examples of legal floating-point constants are:

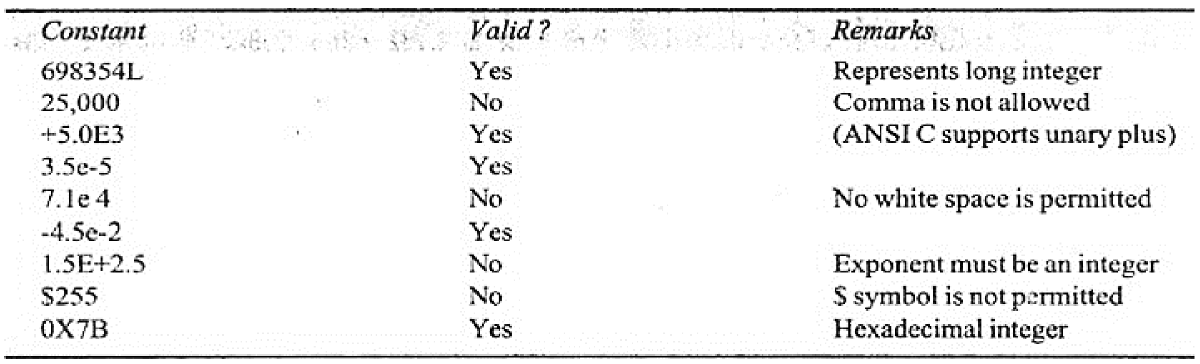
0.65e4 12e-2 1.5e+5 3.18E3 -1.2E-1

Embedded white space is not allowed.

Exponential notation is useful for representing numbers that are either very large or very small in magnitude. For example, 7500000000 may be written as 7.5E9 or 75E8. Similarly, - 0.000000368 is equivalent to -3.68E-7.

Floating-point constants are normally represented as double-precision quantities. However, the suffixes f or F may be used to force single-precision and 1 or L to extend double precision further.

Some examples of valid and invalid numeric constants are given in Table 6.2.1.



Constant	*Valid ?*	*Remarks*
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
$255	No	$ symbol is not permitted
0X7B	Yes	Hexadecimal integer

Table 6.2.1. Examples of numeric constants

6.3. Single Character Constants

A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks. Example of character constants are:

'5' 'X' ';' ' '

Note that the character constant '5' is not the same as the number 5. The last constant is a blank space.

```
printf("%d", 'a');
```

Character constants have integer values known as ASCII values. For example, the statement would print the number 97, the ASCII value of the letter a. Similarly, the statement

```
printf("%c", '97');
```

would output the letter 'a'. Since each character constant represents an integer value, it is also possible to perform arithmetic operations on character constants.

6.4. String Constants

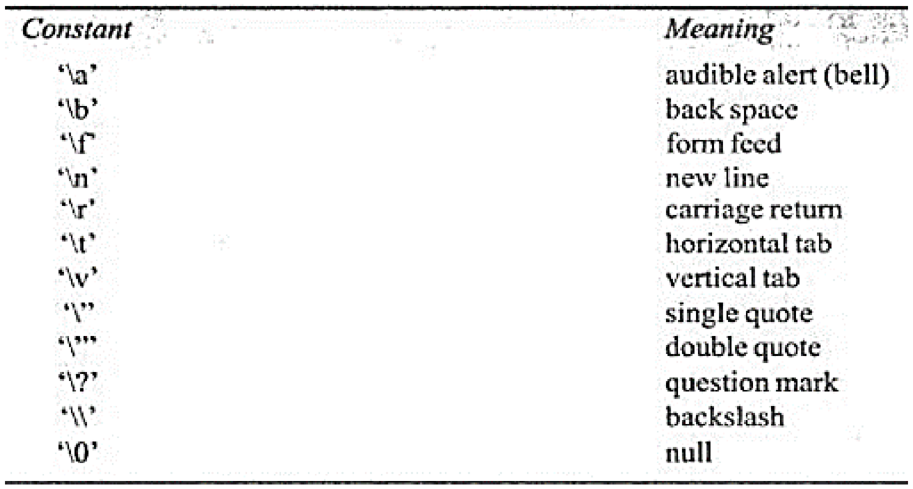
A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters and blank space. Examples are:

"Hello!" "1987" "WELL DONE" "?...!" "5+3" "X"

Remember that a character constant (e.g., 'X') is not equivalent to the single character string constant (e.g., "X"). Further, a single character string constant does not have an equivalent integer value while a character constant has an integer value. Character strings are often used in programs to build meaningful programs.

6.5. Backslash Character Constants

C supports some special backslash character constants that are used in output functions. For example, the symbol '\n' stands for newline character. A list of such backslash character constants is given in Table 6.5.1. Note that each one of them represents one character, although they consist of two characters. These characters combinations are known as escape sequences.



Constant	Meaning
'\a'	audible alert (bell)
'\b'	back space
'\f'	form feed
'\n'	new line
'\r'	carriage return
'\t'	horizontal tab
'\v'	vertical tab
'\''	single quote
'\"'	double quote
'\?'	question mark
'\\'	backslash
'\0'	null

Table 6.5.1. Backslash character constants

7. Variables

A variable is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution.

A variable name can be chosen by the programmer in a meaningful way so as to reflect its function or nature in the program. Some examples of such names are:

Average height Total Counter_1 class_strength

As mentioned earlier, variable names may consist of letters, digits, and the underscore(_) character, subject to the following conditions:

1. They must begin with a letter. Some systems permit underscore as the first character.
2. ANSI standard recognizes a length of 31 characters. However, length should not be normally more than eight characters, since only the first eight characters are treated as significant by many compilers.
3. Uppercase and lowercase are significant. That is, the variable Total is not the same as total or TOTAL.
4. It should not be a keyword.
5. White space is not allowed.

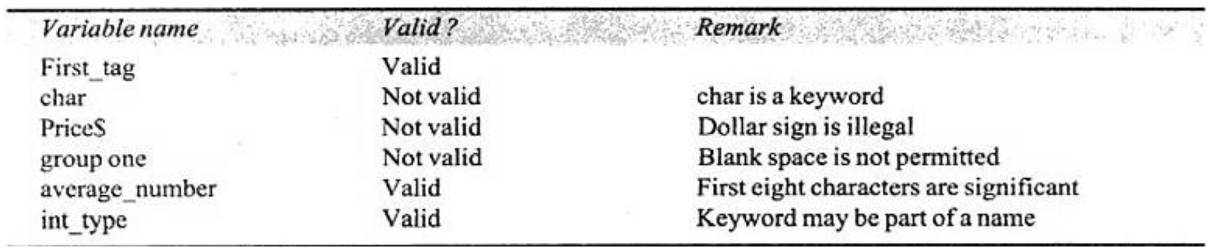
Some examples of valid variable names are:

John	Value	T_raise
Delhi	x1	ph_value
Mark	sum1	distance

Invalid examples include:

123	(area)
%	25th

Further examples of variable names and their correctness are given in Table 8.1.



Variable name	*Valid ?*	*Remark*
First_tag	Valid	
char	Not valid	char is a keyword
Price$	Not valid	Dollar sign is illegal
group one	Not valid	Blank space is not permitted
average_number	Valid	First eight characters are significant
int_type	Valid	Keyword may be part of a name

Table 8.1. Examples of variable names

8. Data Input and Output

Input means to feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

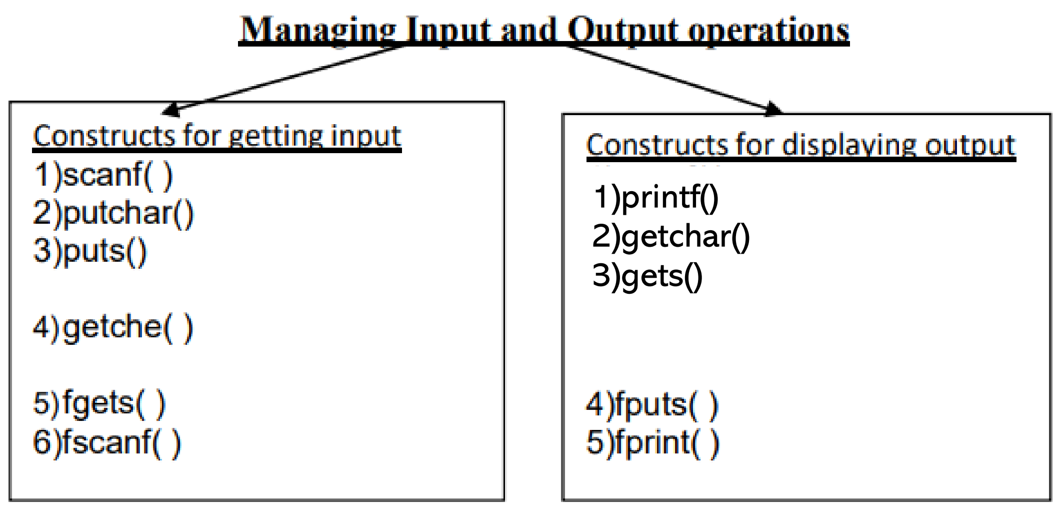
Output that means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output the data on the computer screen as well as you can save that data in text or binary files.

The Standard Files

C programming language treats all the devices as files. So, devices such as the display are addressed in the same way as files and following three file are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	**File Pointer**	**Device**
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

The file pointers are the means to access the file for reading and writing purpose. C language has standard libraries that allow input and output in a program. The stdio.h or standard input output library in C that has methods for input and output.



a. Single character input and output [getchar() and putchar()]

- ➢ input- getchar()
- ➢ output- putchar()

The **int getchar(void) function** reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. EOF is returned by the function in two cases:

- When the file end is reached
- When there is an error during the execution

Syntax:

```
int getchar(void);
```

getchar() function does not take any parameters.

The **int putchar(int c) function** accepts a character as parameter and writes it on the screen. It also returns the same character. The function returns EOF when some error occurs. This function puts only single character at a time.

Syntax:

```
int putchar(int ch)
```

Parameters:

This method accepts a mandatory parameter ch which is the character to be written to stdout.

<u>Demo Program:</u>

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

```
int main() {
int c;
printf("Enter a value :");
c = getchar();
printf("\nYou entered: ");
putchar(c);
return 0;
}
```

Output

```
Enter a value : this is DS class

You entered: t
```

b. String input and output[gets()and puts()]

- Input--- gets (str)
- Output---puts (str)

The gets() function reads characters from the standard input (stdin) and stores them as a C string into str until a newline character or the end-of-file is reached. The function returns a pointer to the string where input is stored.

Syntax:

```
char *gets( char *str );
```

Parameters:

- str: Pointer to a block of memory (array of char) where the string read is copied as a C string.

The puts() function prints strings character by character until the NULL character is encountered. The puts() function prints the newline character at the end of the output string. On success, the puts() function returns a non-negative value. Otherwise, an End-Of-File (EOF) error is returned.

Syntax:

```
int puts(char* str);
```

Parameters:

- str: string to be printed.

Demo Program:

```
#include <stdio.h>
int main() {
char str[100];
printf("Enter a value :");
gets(str);
printf("\nYou entered: ");
puts(str);
return 0;
}
```

Output

```
Enter a value : this is DS class

You entered: this is DS class
```

c. scanf() and printf()

scanf() function is a predefined function in "stdio.h" header file. It can be used to read the input value from the keyword.

Syntax:

int scanf(const char *format, ...);

Parameters:

- int is the return type.
- format is a string that contains the format specifiers(s).
- "…" indicates that the function accepts a variable number of arguments.

That is,

scanf("control string", &variable1, &variable2, ...);

- & ampersand symbol is the address operator specifying the address of the variable
- control string holds the format of the data
- variable1, variable2, ... are the names of the variables that will hold the input value.

The scanf in C returns three types of values:

1. **>0:** The number of values converted and assigned successfully.
2. **0:** No value was assigned.

3. **<0:** Read error encountered or end-of-file(EOF) reached before any assignment was made.

For the control string, you can specify %s, %d, %c, %f, %ld, %lld etc., to print or read strings, integer, character, float, long integers, or long long integers respectively. There are many other formatting options available which can be used based on requirements.

Example 1:

```
int a;
float b;
scanf("%d%f",&a,&b);
```

Example 2:

```
double d;
char c;
long int l;
scanf("%c%lf%ld",&c&d&l);
```

printf function is a predefined function in "stdio.h" header file, by using this function, we can print the data or user defined message on console or monitor. While working with printf(), it can take any number of arguments but first argument must be within the double cotes (" ") and every argument should be separated with a comma (,). Whatever passed within the double quotes are printed as it is and if any format specifies are there, then the value is copied in that place. The printf() returns the number of characters printed after successful execution and if an error occurs, a negative value is returned.

Syntax:

```
printf ( "formatted_string", arguments_list);
```

Parameters:

- formatted_string: It is a string that specifies the data to be printed. It may also contain a format specifier to print the value of any variable such as a character and an integer.
- arguments_list: These are the variable names corresponding to the format specifier.

Demo Program:

```
#include <stdio.h> //This is needed to run printf() function.
int main()
```

```
{
printf("C Programming"); //displays the content inside quotation
return 0;
}
```

Output

```
C Programming
```

d. File string input and output using fgets( )and fputs( )

The fgets() function reads a line from the specified stream and stores it into the string pointed to by str. It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first. The fgets() function returns a pointer to the string where the input is stored.

Syntax:

```
char *fgets (char *str, int n, FILE *stream);
```

Parameters:

- str: Pointer to an array of chars where the string read is copied.
- n: Maximum number of characters to be copied into str (including the terminating null character).
- *stream: Pointer to a FILE object that identifies an input stream.

Example:

```
File*fp;
Str[80];
fgets(str,80,fp)
```

Demo program:

```
#include<stdio.h>
void main()
{
FILE *fp;
char str[80];
fp = fopen("file.txt","r"); // opens file in read mode (“r”)
while((fgets(str,80,fp))!=NULL)
printf("%s",str); //reads content from file
fclose(fp);
}
```

Data in the file and the output:

```
C is a general-purpose programming language.

It is developed by Dennis Ritchie.
```

The fputs() function is used to write string(array of characters) to the file. The fputs() function takes two arguments, first is the string to be written to the file and second is the file pointer where the string will be written. It returns 1 if the write operation was successful, otherwise, it returns 0.

Syntax:

```
fputs(char str[], FILE *fp);
```

Parameters:

str is a name of char array that we write in a file and fp is the file pointer.

Demo Program:

```
#include<stdio.h>
void main()
{
FILE *fp;
fp = fopen("proverb.txt", "w+"); //opening file in write mode
fputs("Cleanliness is next to godliness.", fp);
fputs("Better late than never.", fp);
fputs("The pen is mightier than the sword.", fp);
fclose(fp);
return(0);
}
```

Output:

```
Cleanliness is next to godliness.

Better late than never.

The pen is mightier than the sword.
```

e. fscanf() and fprintf() function

The fscanf() function is used to read mixed type(characters, strings and integers) from the file. The fscanf() function is similar to scanf() function except the first argument which is a file pointer that specifies the file to be read. It returns zero or EOF, if unsuccessful. Otherwise, it returns the number of items successfully assigned.

Syntax:

int fscanf(FILE *fp, const char * format);

fscanf reads from a file pointed by the FILE pointer (ptr), instead of reading from the input stream.

Demo program:

```
#include<stdio.h>
void main()
{
FILE *fp;
char ch;
int roll;
char name[25];
fp = fopen("file.txt","r");
printf("\n Reading from file...\n");
while((fscanf(fp,"%d%s",&rollno,&name))!=NULL)
printf("\n %d\t%s",rollno,name);//reading data
fclose(fp);
}
```

Output:

```
Reading from file...

6666 keith

7777 rose
```

The fprintf() function is used to write mixed type(characters, strings and integers) in the file. The fprintf() function is similar to printf() function except the first argument which is a file pointer specifies the filename to be written.

Syntax:

int fprintf(FILE *fptr, const char *str, ...);

fprintf writes to a file pointed by the FILE pointer (ptr), instead of writing to the output stream.

Demo Program:

```
#include<stdio.h>
void main()
{
FILE *fp;
int roll;
char name[25];
fp = fopen("file.txt","w");
scanf("%d",&roll);
scanf("%s",name);
fprintf(fp,"%d%s%",roll,name);
close(fp);
}
```

<u>*Output:*</u>

```
6666

John
```

9. Operators in C

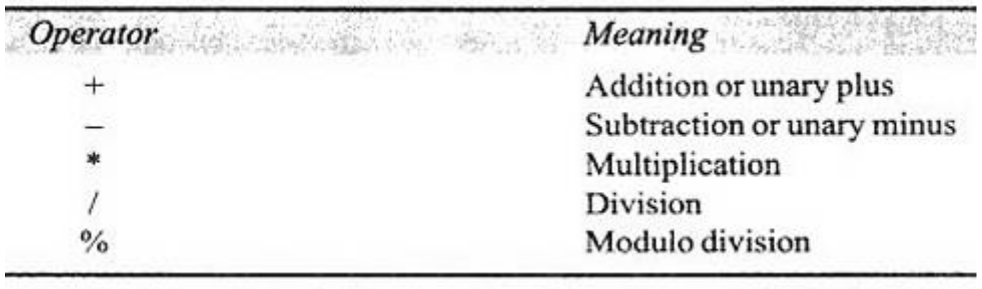
C supports a rich set of built-in operators, such as =, +, -, *, &, and <. An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in programs to manipulate data and variables. They usually form a part of the mathematical or logical expressions. C operators can be classified into a number of categories. They include:

a. Arithmetic operators
b. Relational operators
c. Logical operators
d. Assignment operators
e. Increment and decrement operators
f. Conditional operators
g. Bitwise operators
h. Special operators

An expression is a sequence of operands and operators that reduces to a single value. For example, 10+15 is an expression whose value is 25. The value can be any type other than void.

a. Arithmetic operators

C provides all the basic arithmetic operators. They are listed in Table 9.1. The operators +, -, * and / all work the same way as they do in other languages. These can operate on any built-in data type allowed in C. The unary minus operator, in effect, multiplies its single operand by -1. Therefore, a number preceded by a minus sign changes its sign.



Operator	Meaning
+	Addition or unary plus
–	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

Table 9.1. Arithmetic Operators

Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division. Examples of use of arithmetic operators are:

```
a – b        a + b
a * b        a/b
a% b         - a * b
```

Here a and b are variables and are known as operands. The modulo division operator % cannot be used on floating point data.

Integer Arithmetic

When both the operands in a single arithmetic expression such as a+b are integers, the expression is called an integer expression, and the operation is called integer arithmetic.

Integer arithmetic always yields an integer value. The largest integer value depends on the machine. In the above examples, if a and b are integers, then for a = 14 and b = 4 we have the following results:

```
a - b = 10
a + b = 18
a^ * b = 56
a/b   = 3 (decimal part truncated)
a%b =  2 (remainder of division)
```

During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of truncation is implementation dependent. That is, 6/7 = 0 and - 6/- 7 = 0 but - 6/7 may be zero or-1. (machine dependent)

Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). That is

(- 14)% 3 = - 2

(- 14)% - 3 = - 2

14% - 3 = 2

Example 9.1. The following program shows the use of integer arithmetic to convert a given number of days into months and days.

Program

```
main ()
{
int months, days;
printf("Enter days\n"):
scanf("%d", &days):
months = days/30;
days = days%30;
printf("Months = %d Days = %d", months, days);
}
```

Output

```
Enter days
265
Months = 8 Days = 25
Enter days
364
Months = 12 Days = 4
Enter days
45
Months = 1 Days = 15
```

The variables months and days are declared as integers. Therefore, the statement

months = days/30;

truncates the decimal part and assigns the integer part to months. Similarly, the statement

```
days = days%30;
```

assigns the remainder part of the division to days. Thus, the given number of days is converted into an equivalent number of months and days and the result is printed as shown in the output.

Real Arithmetic

An arithmetic operation involving only real operands is called real arithmetic. A real operand may assume values either in decimal or exponential notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result. If x, y, and z are floats, then we will have:

```
x=6.0/7.0 = 0.857143

y=1.0/3.0 = 0.333333

z=-2.0/3.0 = -0.666667
```

The operator % cannot be used with real operands.

Mixed-mode Arithmetic

When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression. If either operand is of the real type, then only the real operation is performed, and the result is always a real number. Thus,

```
          15/10.0 = 1.5

Whereas,  15/10 = 1
```

b. Relational operators

We often compare two quantities and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on. These comparisons can be done with the help of relational operators. An expression such as

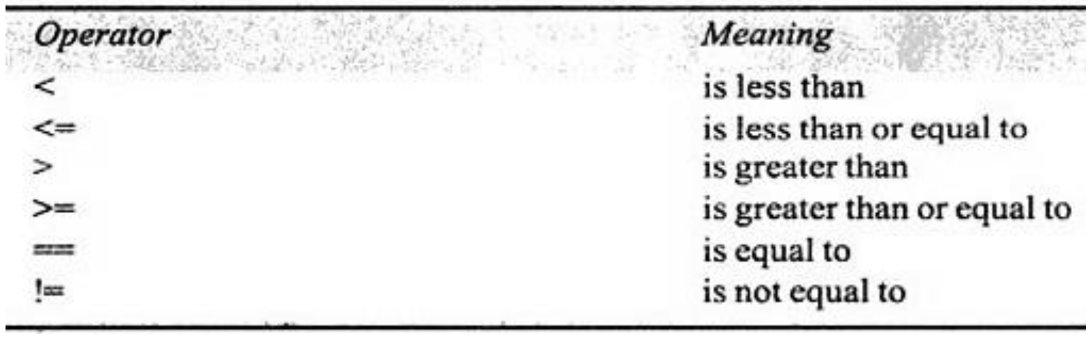
```
a < b or 1 < 20
```

containing a relational operator is termed as a relational expression. The value of a relational expression is either one or zero. It is one if the specified relation is true and zero if the relation is false. For example, 10 < 20 is true

but

20 < 10 is false

C supports six relational operators in all. These operators and their meanings are shown in Table 9.2.



Operator	*Meaning*
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Table 9.2. Relational operators

A simple relational expression contains only one relational operator and takes the following form:

ae-1 relational operator ae-2

ae-1 and ae-2 are arithmetic expressions, which may be simple constants, variables or combination of them. Given below are some examples of simple relational expressions and their values:

4.5 <= 10	TRUE
4.5 < -10	FALSE
-35 >= 0	FALSE
10 < 7+5	TRUE
a+b = c+d	TRUE only if the sum of values of a and b is equal to the sum of values of c and d.

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators. Relational expressions are used in decision statements such as if and while to decide the course of action of a running program.

Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

> is complement of <=

< is complement of >=

== is complement of !=

We can simplify an expression involving the not and the less than operators using the complements as shown below:

Actual one	Simplified one
!(x < y)	x >= y
!(x < y)	x >= y
!(x != y)	x == y
!(x <= y)	x > y
!(x >= y)	x < y
!(x == y)	x != y

c. Logical operators

In addition to the relational operators, C has the following three logical operators.

- && - logical AND
- || - logical OR
- ! - logical NOT

The logical operators && and || are used when we want to test more than one condition and make decisions. An example is:

```
a>b && x=10
```

An expression of this kind, which combines two or more relational expressions, is termed as a logical expression or a compound relational expression. Like the simple relational expressions, a logical expression also yields a value of one or zero, according to the truth table shown in Table 9.3. The logical expression given above is true only if a > b is true and x=10 is true. If either (or both) of them is false, the expression is false.

Some examples of the usage of logical expressions are:

1. if (age > 55 && salary < 1000)
2. if (number <0 || number > 100)

NOTE: Relative precedence of the relational and logical operators is as follows:

Highest	!
	> >= < <=
	= = ! =
	&&
Lowest	||

It is important to remember this when we use these operators in compound expressions.

op-1	*op-2*	*Value of the expression*	
		op-1 && op-2	*op-1 || op-2*
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Table 9.3. Truth table

d. Assignment operators

Assignment operators are used to assign the result of an expression to a variable. The usual assignment operator is '='. In addition, C has a set of 'shorthand' assignment operators of the form

v op = exp

Where is a variable, exp is an expression and op is a C binary arithmetic operator. The operator op is known as the shorthand assignment operator.

The assignment statement

```
v op= exp;
```

is equivalent to

```
v = v op (exp);
```

with v evaluated only once. Consider an example

```
x += y + 1;
```

This is same as the statement

```
x = x + (y + 1);
```

The shorthand operator += means 'add y + 1 to x' or 'increment x by y + 1' For y = 2 the above statement becomes

```
x +=3
```

and when this statement is executed, 3 is added to x. If the old value of x is, say 5, then the new value of x is 8. Some of the commonly used shorthand assignment operators are illustrated in Table 9.4.

Statement with simple assignment operator	**Statement with shorthand operator**
a = a + 1	a += 1
a = a - 1	a -= 1
a = a * (n+1)	a *= n+1
a = a / (n+1)	a /= n+1
a = a % b	a %= b

Table 9.4. Shorthand Assignment Operators

The use of shorthand assignment operators has three advantages:

1. What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

These advantages may be appreciated if we consider a slightly more involved statement like

```
value(5 * j-2) = va1ue(5 * j-2) + delta;
```

With the help of the += operator, this can be written as follows:

```
value (5 * j-2) += delta;
```

It is easier to read and understand and is more efficient because the expression 5 * j - 2 is evaluated only once.

Example 9.2. The following program prints a sequence of squares of numbers. Note the use of the shorthand operator *=

The program attempts to print a sequence of squares of numbers starting from 2. The statement

```
a *= a;
```

which is identical to

a = a * a;

replaces the current value of a by its square. When the value of a becomes equal or greater than N (=100) the while is terminated. Note that the output contains only three values 2, 4 and 16.

<u>*Program*</u>

```
#define N 100
#define A 2
main()
{
int a;
a = A;
while(a< N)
{
printf("%d\n", a);
a *= a;
}
}
```

<u>*Output*</u>

```
2
4
16
```

e. Auto increment and decrement operators

C allows two very useful operators not generally found in other languages. These are the increment and decrement operators: ++ and --

The operator ++ adds 1 to the operand, while -- subtracts 1. Both are unary operators and takes the following form:

++m; or m++;

--m; or m--;

++m; is equivalent to m = m+1; (or m += 1;)

--m; is equivalent to m = m - 1; (or m -= 1;)

We use the increment and decrement statements in for and while loops extensively.

While ++m and m++ mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement. Consider the following:

```
m = 5;
y = ++m;
```

In this case, the value of y and m would be 6. Suppose, if we rewrite the above statements as

```
m = 5;
y = m++;
```

then, the value of y would be 5 and m would be 6. A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left. On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand. Similar is the case, when we use ++(or --) in subscripted variables. That is, the statement

```
a [i++]=10;
```

is equivalent to

```
a[i] = 10;
i = i + 1;
```

The increment and decrement operators can be used in complex statements. Example:

```
m = n++  - j + 10;
```

Old value of n is used in evaluating the expression. n is incremented after the evaluation. Some compilers require a space on either side of n++ or ++n.

Rules for ++ and -- Operators

- Increment and decrement operators are unary operators, and they require variable as their operands.
- When postfix ++ (or --) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++ (or--) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.

- The precedence and associatively of ++ and -- operators are the same as those of unary + and unary -.

f. Conditional operators

A ternary operator pair "? :" is available in C to construct conditional expressions of the form:

exp1? exp2: exp3

where exp1, exp2, and exp3 are expressions.

The operator ? : works as follows: exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either exp2 or exp3) is evaluated. For example, consider the following statements.

```
a = 10;

b = 15;

x= (a > b) ? a : b;
```

In this example, x will be assigned the value of b. This can be achieved using the if…else statements as follows:

```
if (a > b)
   x = a;
else
   x = b;
```

g. Comma operators

The comma operator can be used to link the related expressions together. A comma-linked list of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression. For example, the statement

```
value = (x = 10, y = 5, x+y);
```

first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 (i.e. 10 + 5) to value. Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

In for loops:

```
for (n =1, m = 10, n <= m; n++, m++)
```

In while loops:

```
while (c = getchar(), c != '10')
```

Exchanging values:

```
t = x, x = y, y = t;
```

10. Precedence of operators

An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C

High priority * / %

Low priority + -

The basic evaluation procedure includes 'two' left-to-right passes through the expression. During the first pass, the high priority operators (if any) are applied as they are encountered. During the second pass, the low priority operators (if any) are applied as they are encountered. Consider the following evaluation statement that has been used in the program

x = a - b / 3 + c * 2 - 1

When a = 9, b = 12, and c = 3 the statement becomes

x = 9 - 12 / 3 + 3 * 2 - 1

and is evaluated as follows

First pass

Step 1: x = 9 - 4 + 3 * 2 - 1

Step 2: x = 9 - 4 + 6 - 1

Second pass

Step 3: x = 5 + 6 - 1

Step 4: 11-1

Step 5: x = 10

These steps are illustrated in Fig. 10.1. The numbers inside parentheses refer to step numbers

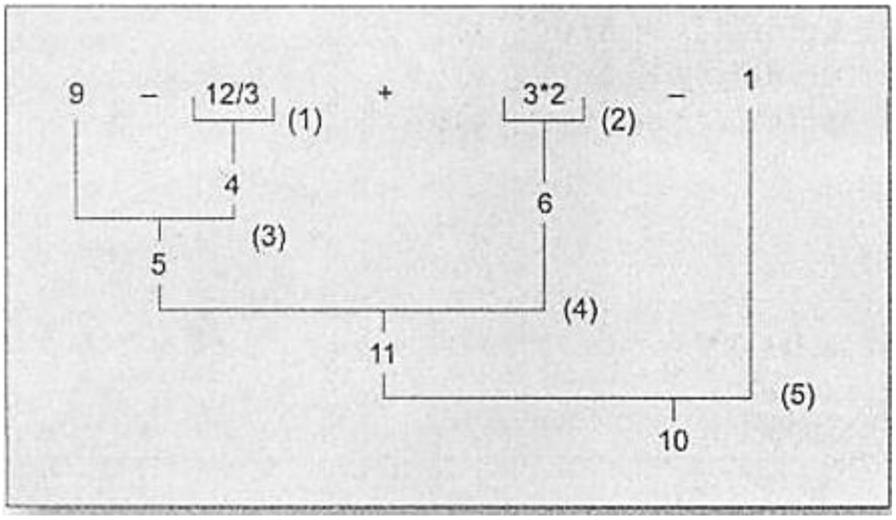


Figure 10.1. Illustration of hierarchy of operations

However, the order of evaluation can be changed by introducing parentheses into an expression, Consider the same expression with parentheses as shown below:

9 - 12 / (3 + 3) * (2 - 1)

Whenever parentheses are used, the expressions within parentheses assume highest priority. If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the now steps.

First pass

Step 1: 9 -12 / 6 * (2 - 1)

Step 2: 9 -12 / 6 * 1

Second pass

Step 3: 9 - 2 * 1

Step 4: 9 - 2

Third pass

Step 5: 7

This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remains the same as 5 (i.e. equal to the number of arithmetic operators).

Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing parenthesis. For example,

9 - (12 / (3 + 3) * 2) - 1 - 4

whereas

9 - ((12 / 3) + 3 * 2) -1 = -2

While parentheses allow us to change the order of priority, we may also use them to improve understandability of the program.

11. Expressions

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language. C can handle any complex mathematical expressions. Some of the examples of C expressions are shown in Table 11.1. C does not have an operator for exponentiation.

Algebraic Expression	C Expression
a x b - c	a * b - c
(m + n) (x + y)	(m + n) * (x + y)
$\left(\frac{ab}{c}\right)$	a * b/c
$3x^2 + 2x + 1$	3 * x * x + 2 * x + 1
$\left(\frac{x}{y}\right) + c$	x/y + c

Table 11.1. Expressions

a. Evaluation of expressions

Expressions are evaluated using an assignment statement of the form:

variable = expression;

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left- hand side. All variables used in the expression must be assigned values before evaluation is attempted. Examples of evaluation statements are

```
x = a * b - c;

y = b / c * a;

z = a - b / c + d;
```

The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables a, b, c, and d must be defined before they are used in the expressions.

Example 11.1. The following program illustrates the use of variables in expressions and their evaluation. Output of the program also illustrates the effect of presence of parentheses in expressions.

Program

```
main()
{
float a, b, c, x, y, z;
a = 9;
b = 12;
c = 3;
x = a - b / 3 + c * 2 - 1;
y = a - b / (3 + c) * (2 - 1);
z = a - (b / (3 + c) * 2) - 1 ;
printf("x = %f\n", x);
printf("y = %f\n", y);
printf("z = %f\n", z);
}
```

Output

```
x = 10.000000
y = 7.000000
z = 4.000000
```

Rules for Evaluation of Expression

- First, parenthesized sub expression from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

12. Type conversion in expressions

Implicit Type Conversion

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as implicit type conversion.

During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the higher' type before the operation proceeds. The result is of the higher type. A typical type conversion process is illustrated in Fig 12.1. For example:

int i, x;
float f;
double d;
long int l;
x = 1 / i + i * f - d

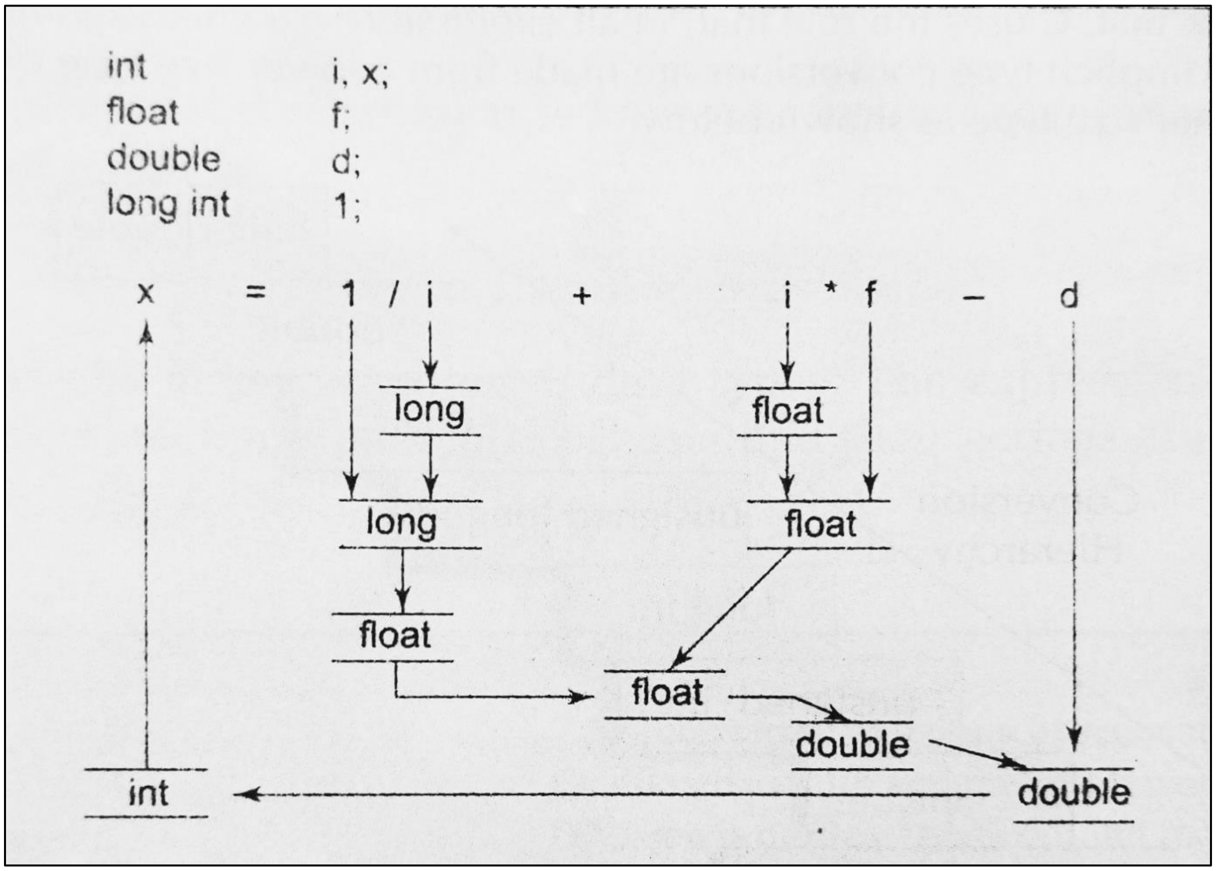


Figure 12.1. Process of implicit type conversion

Given below is the sequence of rules that are applied while evaluating expressions.

All short and char are automatically converted to int; then

1. if one of the operands is long double, the other will be converted to long double and the result will be long double;
2. else, if one of the operands is double, the other will be converted to double and the result will be double;
3. else, if one of the operands is float, the other will be converted to float and the result will be float;
4. else, if one of the operands is unsigned long int, the other will be converted to un-signed long int and the result will be unsigned long int;
5. else, if one of the operands is long int and the other is unsigned int, then
   (a) if unsigned int can be converted to long int, the unsigned int operand will be converted as such and the result will be long int;
   (b) else, both operands will be converted to unsigned long int and the result will be unsigned long int;
6. else, if one of the operands is long int, the other will be converted to long int and the result will be long int;
7. else, if one of the operands is unsigned int, the other will be converted to unsigned int and the result will be unsigned int.

Note that some versions of C automatically convert all floating-point operands to double precision.

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

1. float to int causes truncation of the fractional part.
2. double to float causes rounding of digits.
3. long int to int causes dropping of the excess higher order bits.

Explicit Conversion

C performs type conversion automatically as discussed in the above section. However, there are instances when we want to force a type conversion in a way that is different from the automatic conversion. Consider, for example, the calculation of ratio of females to males in a town.

ratio = female_number/male_number

Since female_number and male_number are declared as integers in the program, the decimal part of the result of the division would be lost and ratio would represent a wrong figure. This problem can be solved by converting locally one of the variables to the floating point as shown below:

ratio = (float) female_number/male_number

The operator (float) converts the female_number to floating point for the purpose of evaluation of the expression. Then using the rule of automatic conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator (float) affect the value of the variable female number. And also, the type of female number remains as int in the other parts of the program.

The process of such a local conversion is known as explicit conversion or casting a value. The general form of a cast is:

(type-name) expression

where type-name is one of the standard C data types. The expression may be a constant, variable or an expression. Some examples of casts and their actions are shown in Table 12.1.

Example	Action
x = (int) 7.5	7.5 is converted to integer by truncation.
a = (int) 21.3 / (int) 4.5	Evaluated as 21/4 and the result would be 5.
b = (double) sum / n	Division is done in floating point mode.
y = (int) (a + b)	The result of a + b is converted to integer.
z = (int) a + b	A is converted to integer and then added to b.
p = cos((double) x)	Converts x to double before using it.

Table 12.1. Use of Casts

Casting can be used to round-off a given value. Consider the following statement:

x = (int) (y+0.5);

If y is 27.6, y+0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to x. Of course, the expression, being cast is not changed.

Example 12.1. Program using a cast to evaluate the equation sum = $\sum_{i=1}^{n} \frac{1}{i}$

Program

```
main()
{
float sum;
int n;
sum = 0;
for(n = 1; n <= 10; ++n)
{
      sum = sum + 1 / (float)n;
      printf("%2d %6.4f\n", n, sum);
}
}
```

Output

```
1  1.0000
2  1.5000
3  1.8333
4  2.0833
5  2.2833
 6 2.4500
7  2.5929
8  2.7179
9  2.8290
10 2.9290
```

a. Precedence and Associativity

Operators in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence, and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from 'left to right' or from 'right to left', depending on the level. This is known as the associativity property of an operator. Table 12.2 provides a complete list of operators, their precedence levels, and their rules of association. The groups are listed in the order of decreasing precedence. Rank 1 indicates the highest precedence level and 15 the lowest.

Consider the following conditional statement:

if (x == 10 + 15 && y < 10)

The precedence rules say that the addition operator has a higher priority than the logical operator (&&) and the relational operators (== and <). Therefore, the addition of 10 and 15 is executed first. This is equivalent to:

if (x == 25 && y < 10)

The next step is to determine whether x is equal to 25 and y is less than 10. If we assume a value of 20 for x and 5 for y, then

x == 25 is FALSE (0)

y < 10 is TRUE (1)

Note that since the operator < enjoys a higher priority compared to ==, y < 10 is tested first and then x == 25 is tested.

Finally, we get:

```
if (FALSE && TRUE)
```

Operator	Description	Associativity	Rank
( )	Function call	Left to right	1
[ ]	Array element reference		
+	Unary plus	Right to left	2
-	Unary minus		
++	Increment		
--	Decrement		
!	Logical negation		
~	Ones complement		
*	Pointer reference		
&	Address		
sizeof	Size of an object		
(type)	Type cast (conversion)		
*	Multiplication	Left to right	3
/	Division		
%	Modulus		
+	Addition	Left to right	4
-	Subtraction		
<<	Left Shift	Left to right	5
>>	Right Shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		
==	Equality	Left to Right	7
!=	Inequality		
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression	Right to left	13
=	Assignment operators	Right to left	14
*= /= **%=**			
+= **-=** **&=**			
^= |=			
<<= >>=			
,	Comma operator	Left to right	15

Table 12.2. Summary of C Operators

Because one of the conditions is FALSE, the complex condition is FALSE. In the case of &&, it is guaranteed that the second operand will not be evaluated if the first is zero and in the case of ||, the second operand will not be evaluated if the first is non-zero.

Rules of Precedence and Associativity

- Precedence rules decide the order in which different operators are applied
- Associativity rule decides the order in which multiple occurrences of the same level operator are applied

References

1. Programming in ANSI C: Balagurusamy
2. https://www.rcet.org.in/uploads/academics/rohini_66913029385.pdf
3. https://www.geeksforgeeks.org/basic-input-and-output-in-c/