Unit 3

Variables and Data Types

Introduction:

C is a programming language developed at AT & T"s Bell laborites of USA in 1972 AD. It was designed and written by Dennis Ritchie. It is popular because, it is simple reliable and easy to use. C has relatively good programming efficiency as compared to machine oriented language and relatively good machine efficiency as compared to problem oriented language or high level language.

The history of C programming language is summarized in the table.

Year	Language	Developed by	Remarks
1960	ALGOL	International committee	Too general, too abstract
1963	CPL	Cambridge University	Hard to learn, difficult to implement
1967	BCPL	Martin Richard (Cambridge University)	Could deal with only specific problems
1970	B	Ken Thompson (AT & T Bell Lab)	Could deal with only specific problems
1972	C	Dennis Ritchie (AT & T Bell Lab)	Lost generality of BCPL and B restored.

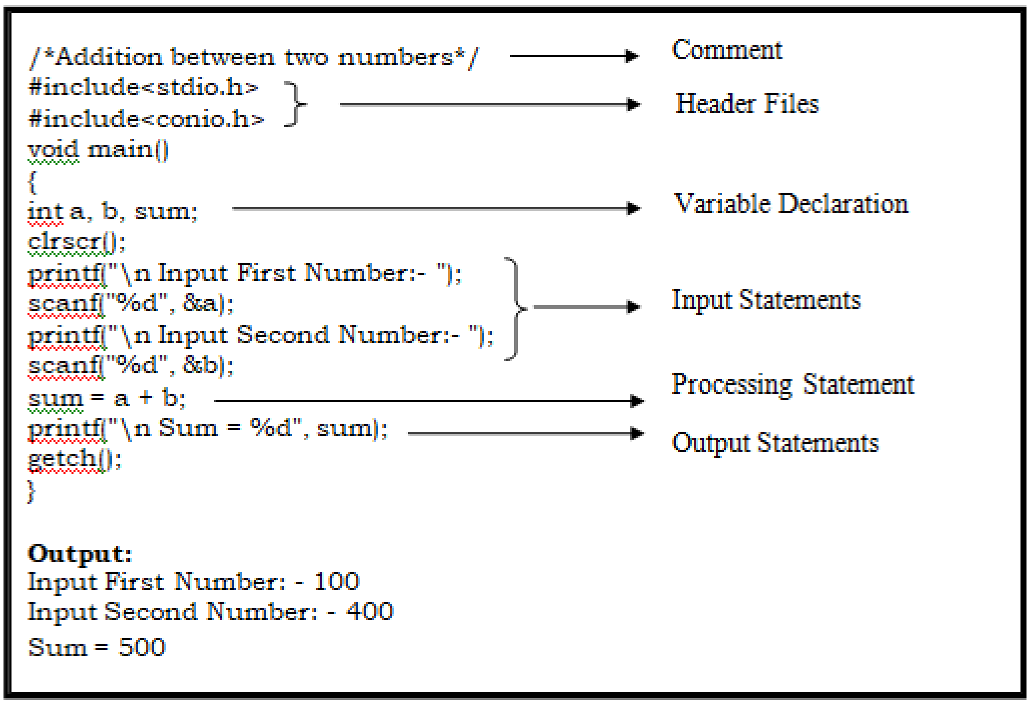
Features of C Program:

- It is a case sensitive programming language, so lowercases are used for programming.
- It is a middle level language that means it combines the features of high level language with the functionality of an assembly language.
- It is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.
- The programs are portable i.e. they can be run on any compiler with little or no modification
- It is more User friendly as compare to previous languages.
- It is a robust language with rich set of built-in functions and operators that can be used to write any complex program.
- C program is basically a collection of functions that are supported by C library. We can also create our own function and add it to C library.
- The compilation and execution time of C language is fast.
- C language is extensible because it can easily adopt new features.
- It supports the feature of dynamic memory allocation, pointer, recursion, etc.
- C language is the most widely used language in operating systems and embedded system development today.

Basic structure of C Program:

```
Header file
void main()
{
Variable declaration or initialization;
Input Statements;
Processing Statements;
Output Statements;
}
```

Example:



```
/*Addition between two numbers*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a, b, sum;
clrscr();
printf("\n Input First Number:- ");
scanf("%d", &a);
printf("\n Input Second Number:- ");
scanf("%d", &b);
sum = a + b;
printf("\n Sum = %d", sum);
getch();
}
```

Output:

```
Input First Number: - 100
Input Second Number: - 400
Sum = 500
```

Character Set

Every programming language has certain valid character sets. The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run. A character denotes any alphabet, digit or special symbols used to represent information. Compiler collects these sets to make sensible tokens otherwise it provide an error message. The characters in C are grouped into the following categories. Some character sets is listed below.

Alphabets	UPPERCASE i.e. A, B, C, … Z lowercase i.e. a, b, c, …. Z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	!, @, #, $, %, ^, & , * , (,), + , = | etc
White space characters	Tab, blank, new line, carriage return, form feed

Comments

Comments are arbitrary text written by a programmer, which are placed between the character sets /* and */. Programmers are free to write comments in any programming language. These are used in program to enhance its readability and understanding. Comment lines are not executable statements and therefore any text between /* and */ is ignored by the compiler. In general, a comment can be inserted anywhere in the program by a programmer. Generally such comments are provided by a programmer for documentation purpose. Comments do not affect the execution speed and the size of a program, a program. They help the programmers and other users in understanding the various functions and operations of a program and serve as an aid to "debugging and testing". So writing comments in a program is a good technique.

In C program, a programmer can introduce comments in two ways: *single line* comment and *multiple line* comments. Single line comment is indicated by // (double slashes) where as multiple line comments is enclosed within /* and */

Example

i) // Single line comment

ii) ./* This program calculates sum of two different numbers*/

While using multiple line comments, it should start with /* and end with */. Anything in between them is ignored by the compiler. If we miss out the closing */, then the compiler searches for closing */ further down in the program, if not found, compiler treats all statements down to /* as a comment and produce error message.

Escape Sequence

Escape sequences are non-printing characters which are started with backslash (\) which causes an escape from the normal interruption of a string, so that the next character is recognized as one having special meaning. These escape sequences are used in output functions. It is also known as backslash character constant. C program supports various types of escape sequence and the list is given in following table.

Escape Sequences	Meanings	Purpose
\a	Audible alert (Bell)	It alerts user by sounding the speaker inside the computer.
\b	Back space	It moves the cursor one position to the left of its current position
\f	Form feed	It advances the computer stationery attached to the printer to the top of the next page.
\n	New line	It causes a new line to begin.
\r	Carriage return	It takes the cursor to the beginning of the line in which it is currently placed.
\t	Horizontal tab	It moves the cursor to the next tab stop.
\v	Vertical tab	
\‟	Single quote	It prints single quote symbol („) on the monitor
\”	Double quote	It prints double quote symbol (“) on the monitor
\?	Question mark	It prints question mark symbol (?) on the monitor
\\	Backslash	It prints backslash symbol (\) on the monitor
\0	Null	It specifies end of the string.

Tokens

In C program the smallest individual units are known as C tokens. C has following six (6) types of tokens.

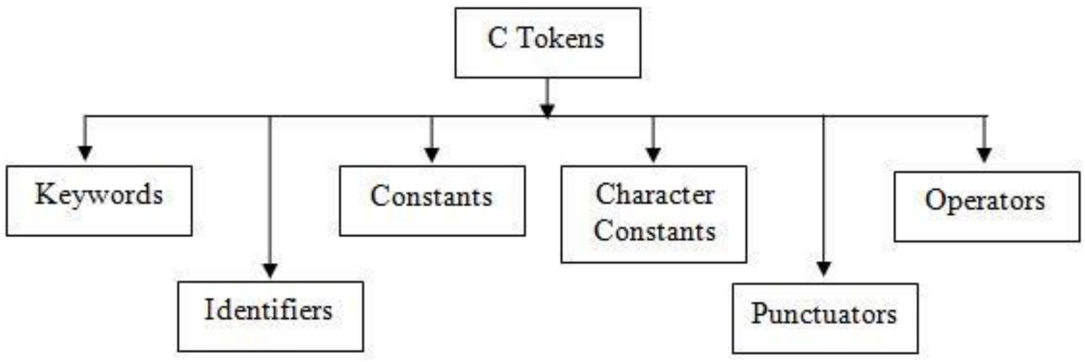


Fig: C -Tokens

Keywords:

Every programming language has different types of keywords which performs some specific task in the program. Keywords are predefined or reserved words that have strict meanings i.e. all keywords have fixed meanings and these meanings cannot be changed. The meanings of the keywords are already defined in the compiler. So the compiler interprets those keywords used in program, if any mistake occurs it reports an error message. Keywords serve basic building blocks for program statements. Keywords cannot be used for naming other identifiers. There are 32 keywords used in C program. C program is case sensitive program. So the keyword should in small case. They are:

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

Identifiers:

Identifiers are the valid set of characters which are used for naming variables used in the program. It can be used to represent the name for normal variables, arrays, structures, unions, etc. The identifiers are user defined names and consist of sequence of letters and digits but first letters should be a character. Every program has its own sets of rules for naming identifiers. Both uppercase and lowercase letters can be used as identifiers, although lowercase letters are commonly used. Keywords can"t be used as identifiers because they have some special meanings and functions in the program. Example: length, area, a, b, sum, etc.

Rules for naming identifiers:

- It is combination of alphabets, digits or underscore.
- Keywords can"t used as identifier
- It is not case sensitive i.e. lower case or upper case can be used as identifier. Both lower case and upper case has distinct meanings such as *num* and *NUM* are two different identifiers.
- Any characters, digits (except some special symbols such as #, . etc) can be used, but the first letter should be a character.
- The length of identifiers depends upon the compiler. In earlier compilers it was 8 characters, ANSI C allows 31 characters but in modern compilers it may be of any length.
- An identifier must be unique in a program.
- Blank spaces are not allowed.

Constants and Variables:

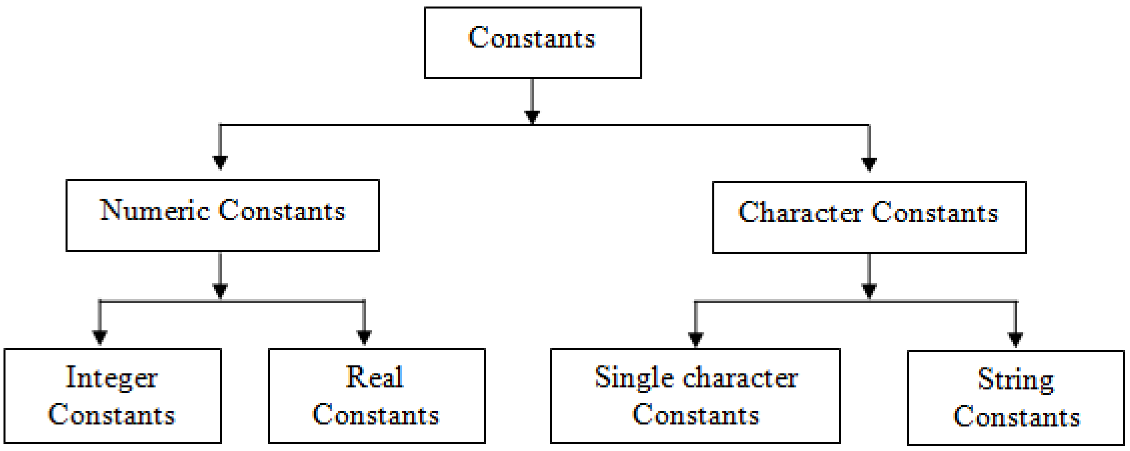


Fig: Basic types of C constants

Constants are the fixed values that do not change during the execution of program. C supports different kinds of constants which are described in above diagram.

Integer Constant
An integer constant refers to a sequence of digits without decimal values. It may be negative or positive numbers. There are three types of integers, and they are: *decimal*, *octal* and *hexadecimal.*

I. *A **decimal integer*** consists of a set of **10** separate digits: **0** through **9**, also known as base **10** number system.
   Example: 1000, -521312, 5 etc.

   Embedded spaces, commas, & no-digit characters are not allowed between digits.
   For example: Rs5000 25,435 36 780 etc are illegal numbers.

II. An ***Octal integer*** consists of a set of **8** separate digits: **0** through **7,** also known as base **8 n**umber system.. Such numbers are written using **0** before an octal digit.
   Example: 0570, 0412, 03124 etc.

III. A ***Hexadecimal integer*** consists of **16** separate digits: **0** through **15**, also known as bas **16** number system. It is denoted by 0x or 0X followed by sequence of digits. It may also include alphabets **A** through **F**. the letters **A** through **F** represents **10** through **15** respectively.
   Example: 0XB2 0X9456 0X00 etc.

But we rarely use octal and hexadecimal numbers in programming.

Rules for constructing integer constants:
- It must have at least one digit
- It must not have a decimal point or fractional part.
- It can be either negative or positive.
- It no sign is provided to an integer, and then it is assumed to be positive.
- No commas or blanks are allowed within an integer constants

Real constants
The fixed numeric constant which may or may not have fractional part or exponential part is called real constants (or floating point constants). Such numeric constants are used to represent quantities such as distance, temperature, price, salary, heights etc. It may be either positive or negative value.

Example: 10.078 -0.123 +280.0 etc

It is possible to omit digits before decimal point or digits after the decimal point and it will also valid real numbers such as.
Example: 10.078 -.123 +280 etc

A real number may be expressed in exponential (or scientific) notation.
For Example: 312.789 may be written as 3.1279e2 in exponential. Where e2 means multiply by 10^2. In exponential form of representation, the real constant is represented in two parts. The general form of a scientific or exponential number is

mantissa **e** exponent

The part appearing before „e" is called *mantissa* and the part following „e" is called exponent.
Example: +1.2e-10 -0.23e5 1.78e+4 etc

Rules for constructing real or float:
- It must have at least one digit
- It must have a decimal point or fractional part.

- It can be either negative or positive.
- It no sign is provided to an integer, and then it is assumed to be positive.
- No commas or blanks are allowed within an integer constants

Single Character Constants
A character represented within a pair of single quotes denotes a single character constant.
Examples: „a‟, „x‟, „1‟ „ „ etc

Rules for constructing character:
- It is a single alphabet, a single digit or single special symbol enclosed within a single inverted comma.
- The maximum length of a character can be 1 character

Strings Constants
A string constant is the sequence of characters enclosed within double quotes. The characters may be letters, numbers, special characters and blank space

Example: "C Programming" "Computer Science" "521312" "A" etc

Note: a character constant e.g. „A‟ is not equivalent to the single character string constant "A". Similarly, a single character string constant does not have an equivalent integer value while a character constant has an integer value.

Variable Declaration:

A variable is a data name that may be used to store a data value i.e. a variable name is name give to locations of memory. Variables may take different values at different times during program execution. All variables should be declared, to tell the compiler what the variable names are and what type of data they hold.

Rules for constructing variable names:
- It may be combination of 1 to 31 alphabets.
- The first character in the variable name should be an alphabet or underscore.
- Commas or blanks spaces are not allowed within a variable name.
- Special symbol other than an underscore can‟t be used.

Variable declaration and initialization

Each and every variable used in a program either to input data, processing or output values must be declared before its use. In C, variables are declared locally to the block or globally. They must be declared at the beginning of the file. Variable declaration does two things:
- It tells the compiler what the variable name is? and
- It specifies what type of data the variable would hold.

Syntax: **data_type variable_names;**

Example:
```
int x;
int a,b,sum;
float percentage;
char c;
```

In above examples, the compiler reserves memory locations of different sizes, depending upon data types of the variables (**2 bytes** for each **int** data type, **1 byte** for each **char** data type and **4 bytes** for each **float** data type) on RAM as variable names **x**, **a**, **b**, **sum**, **percentage**, **c**, which can store integers, float and character respectively. While declaration only, the variables holds garbage value, which are not required by the users.

We can provide an initial value to the variables while declaring those variables. The process of variable declaration and assigning value at the same time is known as ***variable initialization.***

Syntax: `data_type variable_name = initial value;`

Example:

```
    int x = 10;
```

Here, the value **10** is assigned to variable **x**.

Data Types:

A data type defines a set of values that a variable can store along with a set of operations that can be performed on that variable. Each and every variable should be declared before using those variables in the program with its data type. There are two types of data types used in C.

a) Fundamental or Basic or primary data type
b) Derived or secondary data type

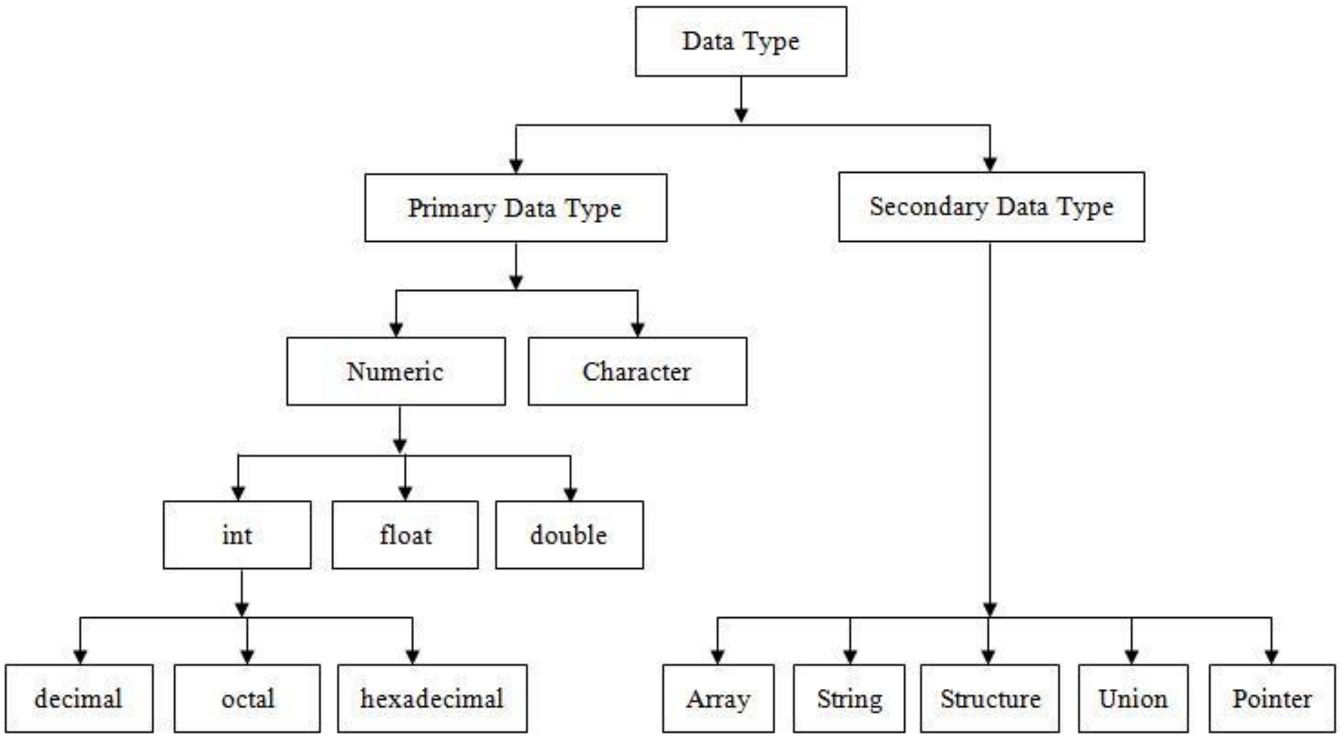


Fig. Classification of Data Types

Fundamental Data type

Fundamental data types are the built in data types in C. There are four basic data type: *int, float, char,* and *double.* C also provides various kinds of data types which are extended from the basic data types by adding different qualifiers.

a) integer data type (int)

- It can hold numeric values without fractional parts.
- The keyword *int* is used to define integer data type.
- The size depends on operating system. If the operating system is of 16 bits, then it occupies 2 bytes of memory space where as if the operating system is of 32 then it occupies 0 to 4294967297 s 4 bytes of memory space.
- It can be used as *int* (or *signed int), unsigned* (or *unsigned int), short (*or *signed short* or *singed short int), unsigned short (*or *unsigned short int) long (*or *signed long int* or *signed long)* and *unsigned long (*or *unsigned long int).*
- The format specifier %d is used to indicate integer data type.
- E.g. int x;

- The memory space used and range for integer data type is shown in following table.

Data type	Memory requirement	Range
Int	2 bytes in 16 bit OS 4 bytes in 32 bit OS	-32768 to 32767 -2147483648 to 2147483647
Unsigned	2 bytes in 16 bit OS 4 bytes in 32 bit OS	0 to 65536 0 to 4294967297
Short	2 bytes	-32768 to 32767
unsigned short	2 bytes	0 to 65536
Long	4 bytes	-2147483648 to 2147483647
unsigned long	4 bytes	0 to 4294967297

b) float data type (float)

- It can hold numeric values with or without fractional parts.
- The keyword *float* is used to define float data type.
- It occupies 4 bytes of memory space.
- The format specifier %f is used to indicate float data type and 6 digits are displayed after fractional point by default.
- E.g. float y;
- The memory space used and range for integer data type is shown in following table.

Data type	Memory requirement	Range
Float	4 bytes	-3.4e38 to -3.4e-38 up to 0 and 3.4e38 to -3.4e-38
Double	8 bytes	-1.7e308 to -1.7e308
long double	10 bytes	-1.7e4932 to -1.7e4932 up to 0 and -1.7e4932 to -1.7e4932
unsigned short	2 bytes	0 to 65536
Long	4 bytes	-2147483648 to 2147483647
unsigned long	4 bytes	0 to 4294967297

c) double data type (double)

- It can hold large range of data such as scientific value.
- The keyword *double* is used to define double data type.
- It occupies 8 bytes of memory space.
- The format specifier %lf or %e is used to indicate double data type.
- E.g. double z;

d) character data type (char)

- It can hold single character and are placed between single quotes („ „).
- The keyword *char* is used to define character data type.
- It occupies 1 bytes of memory space.
- It can be used as *char*, *signed char* and *unsigned char*.
- The format specifier %c and %s (for string, collection of characters) is used to indicate character data type.
- E.g. char x;
- The memory requirements and the range of character data type is given in following table

Data type	Memory requirement	Range
Char	1 byte	0 to 255 or -128 to 127
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127

Format Specifier:

Format specifier is a single character followed by % symbol, which tells compiler to print the value of data in specified format. Some format specifiers are listed in following table.

Data type		Format specifier
integer	short signed short unsigned long signed long signed unsigned hexadecimal unsigned octal	%d or %i %u %ld %lu %x %o
Real	float double	%f %lf
character	signed character unsigned character	%c %c
String		%s

```
/*Demonstration of format specifiers*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a=10;
float b=10.25,sum;
clrscr();
sum=a+b;
printf("\n A = %d, B = %f",a,b);
printf("\n Sum = %f",sum);
getch();
}
```

Output:

```
A = 10, B = 10.250000
Sum = 20.250000
```

Here, in the first **printf ()** function in the program, it first places the cursor at the beginning of the next line, **A =** is dumped in the screen and the moment **%d** is met, the variable **A** is picked and its value (i.e. 10) is printed. Then after again, **B =** is also dumped in the monitor, and the moment **%f** is reached, it picks next variable **b** from the variable list and prints its value (i.e. 10.250000) in specified format. Similar process is also applied in the second **printf ()** function of the program.

We can provide following optional specifier in the format specifications.

Specifier	Description
Dd	d (digit) specifies the field width
.	It separates the field width form precision
-	(Minus sign) it is used for left justification of the output in the specified field width

Type conversion

When writing an expression, we use different types of data and when evaluating expressions with such mixed data types. There are different ways of data conversion.

a) automatic (implicit) type conversion
b) type cast (explicit) type conversion

Automatic type conversion

There is a rule for automatic type conversion which is illustrated below:

long double   double   float   long   int   short   char

Explicit type conversion

Sometimes we are required to force the compiler to explicitly convert the value of an expression to a particular data type. This explicit conversion is also called “cast”. Following syntax is used for explicit type conversion.

(data_type) expression;

For example:

```
int x, y;
float result;
result = (float)(x+y);
```

Here, the variable *result* is of data type *float* and the variables *x* and *y* are of *int* data type and the data type of *x* and *y* is converted into *float* for calculation.

The working mechanism of type conversion is illustrated in following program.

```
/*Demonstration without of explicit conversion*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a=10, b=4;
float result;
clrscr();
result=a/b;
printf("\n Result  = %f”, result);
getch();
}
```

Output:

```
Result = 2.000000
```

Here the output of the program is 2.000000 but not 2.5 this is because, 10 and 4 both are integers and hence **10/4** yields an integer 2. This value 2 when stored in *result* is converted to 2.000000. To overcome from such situations, either **a** or **b** as **float.** The process is described in following program.

```
/*Demonstration of explicit conversion*/
#include<stdio.h>
#include<conio.h>
```

```
void main()
{
int a=10, b=4;
float result;
clrscr();
result=(float)a/b;
printf("\n Result  = %f", result);
getch();
}
```

Output:

```
Result = 2.500000
```

Here expression **(float)** causes the variable **a** to be converted from type **int** to type **float** before being used in the division operation. The value of **a** doesn"t get permanently changed as the result of typecasting.

Simple input / output Function:

C program has some set of input/output functions which are used to take inputs from the standard input devices (keyboard) and display the messages as well as results to the standard output device (monitor). An standard input-output library functions `printf( )` and `scanf( )` is used for output and input respectively. Both functions are the part of standard library function so a standard header file `<stdio.h>` is included before using the functions.

scanf( ) function
It is used to input (or read) data from standard input device such as keyboard.
Syntax:

i) *scanf("control strings ",arguments);*

Here,
Control string may be any valid format specifiers (conversions specifications) such as %d, %c, %f, %s, etc which is recommended. You can use numbers of arguments as required by the program and those arguments should be of any valid identifiers, which is address location of memory.

Example:

i) *scanf("%d",&a);* // Single input
ii) *scanf("%d%f%d",&a,&b,&c);* // Multiple inputs

In above first example, the data inputted by the user is stored at the address of ***a***, as an integer value. Similarly in second example, there are three inputs: first inputted number is stored at location ***a*** as *integer*, second data at location ***b*** as *float* and third data at location ***c*** as an *integer*.

Let, the input given by the user is:

```
100    110.28    200
```

Then those values 100, 110.280000 and 200 are stored at the location of *a*, *b* and *c* respectively.

printf ( ) function

It is used to display message or result in the monitor.

Syntax:

```
i)   printf("Message");
ii)  printf("control strings", arguments);
```

Example:

```
i)   printf("Hello Class 12 Students");
ii)  printf("Sum = %d", sum);
```

Conversion specifier

Conversion specifier control the conversion of the value of arguments in *printf()* to be converted to display formats as according to their type before the control string is displayed along with their values. They provide the type of data that is being displayed. They are also known as ***conversion specification.***

For example:

```
printf (“Sum = %d”, sum);
```

It tells that the value of *sum* that we are going to display is of the type integer. So it places the value of *sum* at the position where *%d* is placed.

The conversion specifier starts with *%* and ends with conversion character. Format of conversion specification looks like:

%[flags] [width] [.precision] conversion_character

- *Flag* causes output justification: left justification, right justification.
- *Width* specifies total characters to print, padding with blanks or zeros.
- *.precision* specifies the floating point data i.e. it specifies the number of digits to be printed after point.
- *Conversion_character* specifies the character to be used with specified data type in the argument.

Now let us examine what happens when we declare data as follows:

```
int a = 12345;
float b = 12.34567;
```

The output of *printf()* with different conversion specifier is described below.

- `printf( “%d”, a)` displays output as: initial position of the screen

1	2	3	4	5

- `printf( “%10d”, a)` displays output as:

					1	2	3	4	5

Total width will be 10, 5 spaces are used by the numbers and it leaves 5 blank spaces before the number i.e. the numbers are displayed as right justification.

- `printf( “%-10d”, a)` displays output as:

1	2	3	4	5					

Total width will be 10, 5 spaces are used by the numbers and it leaves 5 blank spaces after the number i.e. the numbers are displayed as left justification.

- `printf( “%f”, b)` displays output as:

1	2	.	3	4	5	6	7	0

The numbers are displayed from the beginning of the line with 6 digits in the decimal places.

- `printf( “%10f”, b)` displays output as:

	1	2	.	3	4	5	6	7	0

The numbers are displayed as right justification. Here precision is not specified, so 6 digits are displayed after point.

- `printf( "%10.2f", b)` displays output as:

					1	2	.	3	4

Here total width is 10 but the numbers needs only 5 spaces: 2 spaces after point, 2 spaces before point and 1 space for the decimal point, 5 spaces are left as blank before the numbers and the numbers are right justified.

- `printf( "%-10.2f", b)` displays output as:

1	2	.	3	4					

Here total width is 10 but the numbers needs only 5 spaces: 2 spaces after point, 2 spaces before point and 1 space for the decimal point, 5 spaces are left as blank after the numbers and the numbers are left justified.

- `printf( "%e", b)` displays output as:

1	.	2	3	4	5	6	7	E	+	0	1

There is only 1 digit before the decimal point.

- `printf( "%12,3e", b)` displays output as:

			1	.	2	3	4	E	+	0	1

Here, the total width is 12 spaces with 3 digits after the point in exponential format. The numbers are displayed as right justification.

Operators:

C program consists of different kinds of character set and symbols which has some special meanings. Operators are the symbols used in between the operands in-order to perform some specific task such as mathematical or logical operations. There are different kinds of operators. Operators are classified in two broad categories:

- On the basis of number of operands used
- On the basis of behavior of operators

On the basis of number of operands:
There are two types of operators: monadic (unary) and dyadic (binary) operators. If an operator requires an operand, then such operator is called as monadic operator. E.g. increment (++), decrement (--), minus (-), addition (+), etc. whereas dyadic operators are those operators which takes two or more operands. E.g. relational operators (>, <, = =, etc), arithmetic operators (+, - , *, /, %), etc. Let us consider following expression:

a – b	Here, - (minus) symbol takes two operands: *a* and *b*. so it is dyadic operator.
- a	Here, the same symbol – (minus) takes only single operand *a*, so it is monadic operator

On the basis of behavior of operators:
There are various types of operators available in C and some of frequently used operators are:

a) Arithmetic operator

It is used to perform some basic arithmetic operations between numbers of operands. They are:

Let us consider, the value of a = 10 and b = 3, then

Operator	Meaning	Expression	Result
+	Addition	a + b	10 + 3 = 13
-	Subtraction	a - b	10 - 3 = 7
*	Multiplication	a * b	10 * 3 = 30
/	Division (Real)	a / b	3 (Quotient)
%	Modulus Division	a % b	1 (Remainder)

b) Relational operator

It is used to create a relationship between operands i.e. they are used to compare values of the given operands. It may provide either TRUE value (1) or FALSE value (0), depending upon the values of the operands. The relational operators are described in following table.

Let us consider, a = 10 and b = 3, then

Operator	Meaning	Expression	Remarks
>	Greater than	a > b	TRUE
<	Less than	a < b	FALSE
> =	Greater than or equals to	a > = b	TRUE
< =	Less than or equals to	a < = b	FALSE
= =	Equals to	a = = b	FALSE
! =	Not equals to	a != b	TRUE

c) Logical operator

It is used to create a logical relationship between operands. They are:

Operator	Meaning	Expression	Remarks
&&	Logical AND	a && b	Results TRUE value if and only if all the given inputs are true, otherwise results FALSE value
||	Logical OR	a || b	Results TRUE value if any one of the given inputs are true, results FALSE value, if all the inputs are false.
!	Logical NOT	a ! b	Results complement value of the given input.

d) Increment / Decrement operator

The increment and decrement operators are unary operators, which takes only one operand. The increment operator (++) adds the value 1 to the current value of the operand and the decrement operator (- -) subtracts the value 1 from the current value of the operand. These operators are extensively used in loop control structures in C. Further, the increment and decrement operator is divided into categories: postfix and prefix increment / decrement operator.

- Postfix increment (operand ++)
- Prefix increment (++operand)
- Postfix decrement (operand - -)
- Prefix increment (- -operand)

Postfix increment:

In postfix increment the operator is placed after the operand. Here in postfix increment, first the value is assigned to the variable on the left and then increments the operand. *Syntax:* operand++;

Eg: x++; which is similar to x = x + 1;
Let us consider following expressions.
x = 10; // the value 10 is assigned to variable x
y = x++;
Here, the value of y will be 10, then only the value of is increased by 1 and it becomes 11.

Prefix increment:

In prefix increment the operator is placed before the operand. Here in prefix increment, the operator first adds 1 to the operand and then the updated result is assigned to the variable on the left.
Syntax: ++operand;
Eg: ++x; which is also similar to x = x + 1;
Let us consider following expressions.
x = 20; // the value 20 is assigned to variable x
y = ++x;
Here, the value of x will be 21, then the new value of x (21) is assigned to variable y. so y will be 21

Postfix decrement:

In postfix decrement the operator is placed after the operand. Here in postfix decrement, first the value is assigned to the variable on the left and then decrements the operand. *Syntax:* operand- -;

Eg: x- -; which is similar to x = x - 1;
Let us consider following expressions.
x = 10; // the value 10 is assigned to variable x
y = x- -;
Here, the value of y will be 10, then only the value of x is decreased by 1 and it becomes 9.

Prefix decrement:

In prefix decrement the operator is placed before the operand. Here in prefix decrement, the operator first subtracts 1 from the operand and then the updated result is assigned to the variable on the left.
Syntax: - -operand;
Eg: - -x; which is also similar to x = x - 1;
Let us consider following expressions.
x = 20; // the value 20 is assigned to variable x
y = - -x;
Here, the value of x will be 19, and then the new value of x (19) is assigned to variable y. so y will be 19

e) Assignment operator (=)

It is used to assign value to an operand as well as is used to copy value of an operand to another operand. There may be one or more operands right (after) to the assignment operator but only one operand left (before) to the assignment operator.
Let us consider:

```
x = 100;    // value 100 is assigned to variable x
y = x;      // the value of x is copied or assigned to variable y
z = x + y;  // first the value of x and y are added, then only the added value is assigned to variable z.
```

Invalid assignment operators
x + y = z;
100 = x;

f) Compound operator

Compound operators are combination of arithmetic and an assignment operator. Generally it is used to write short, concise and more efficient codes in the program. Some compound operators and its meanings are:

Operator	Expression	Meanings
+ =	a + = 10	a = a + 10
- =	a - = 10	a = a – 10
* =	a * = 10	a = a * 10
/ =	a / = 10	a = a / 10
% =	a % = 10	a = a % 10

g) Conditional operator (? :)

Conditional operator is a ternary operator which requires three expressions as operands. It is denoted by **?** and **:** symbol.

Syntax: test_condition ? True_expression : False_expression

E.g. a > b ? a : b

In such expression, firstly the test condition is evaluated, if the condition is satisfied then TRUE expression is evaluated, otherwise FALSE expression is evaluated.

```
/*Demonstration of conditional operator*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a,b;
clrscr();
printf("\n Input First Number:- ");
scanf("%d", &a);
printf("\n Input Second Number:- ");
scanf("%d", &b);
if(a>b?printf("\n %d is greatest", a):printf("\n %d is greatest" ,b))
getch();
}
```

Output:

```
Input First Number:- 10
Input Second Number:- 50
50 is greatest
```

h) sizeof( ) operator

It is a unary operator. This operator gives the size of its operand in terms of bytes occupied in the memory. The operand can be a variable, constant or any data type: int, float, char, etc.

Example: result = sizeof(x);

Here sizeof( ) operator calculates the memory space occupied by variable x and is assigned to variable result.

Precedence and Associativity of operators

A C expression can consist numbers of operators. However the compiler evaluates a single operator at a time. If more operators are present in an expression then compiler should decide which one to evaluate first. The rule that specifies which operator to evaluate first is called *precedence* of the operator. The operator with higher precedence is evaluated first and lower precedence operators are evaluated next. When an expression contains two operators of equal priority, then the associativity of operators decides which operation would be performed first. Associativity can be of two types - *Left to Right* or *Right to Left*. The process of evaluating the operators of same priority is known as associativity. In the Left to Right associativity, the operator at the left most part is evaluated first and it moves towards right whereas in Right to Left associativity, the operator at the right most part is evaluated fist and it moves toward left. The priorities of commonly used operators are shown in following figure.

Priority	**Operators**	**Description**
1st	* / %	Multiplication, Division, Modular division
2nd	+ -	Addition, Subtraction
3rd	=	Assignment

Summary of Operators used in C Program:

Operators	**Description**	**Associativity**	**Rank**
( ) [ ]	Function Call Array element reference	Left to right	1
+ - ++ -- ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical NOT Ones complement Indirection Address of Size of an object Type cast c(conversion)	Right to left	2
* / %	Multiplication Division Modulus	Left to right	3
+ -	Addition Subtraction	Left to right	4
<< >>	Left shift Right shift	Left to right	5
< <= > >=	Less than Less than or equals to Greater than Greater than or equals to	Left to right	6
= = ! =	Equals to In equals to	Left to right	7
&	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	Left to right	13
= *=, /=. %= +=, -= ^=, |= <<=, >>=	Assignment operators	Right to left	14
,	Comma operator	Left to right	15

C-expressions and algebraic expressions

There are certain rules for writing expressions in C and some of them are described as:

- C allows only one variable or constant at left hand side of any expression that has assignment (=) sign.

  Example

valid statement	Invalid statement
x = a + b;	x + a = b;

- Any operators should be written between operands

  Example

valid statement	Invalid statement
x = a * b;	x = a . b;
	x = a b;

- There is no any operators or symbols for writing square root in c x = a * * 2;

  x = a ^ 2;

- We must always use parenthesis (i.e. „(" and „)"), braces („{„ , „}") and square bracket (i.e. [,]) are not allowed in between the expressions.

  Example

valid statement	Invalid statement
(3 + (5 + (1 + 4) * 5) + 10)	[3 + {5 + (1 + 4) * 5} + 10]

- While writing polynomial expressions we can directly convert the given polynomial expressions into C. but we must minimize the number of computations.

  $4x^4 + 3x^3 + 2x^2 + x + 12$ it can be written in C expression

  as: 4*x*x*x*x + 3*x*x*x + 2*x*x + x + 12

  In this expression, there are 9 multiplication and 4 addition operations. It can be minimized as:

  $4x^4 + 3x^3 + 2x^2 + x + 12$

  $= x (4x^3 + 3x^2 + 2x + 1) + 12$

  $= 12 + x [1 + x (4x^2 + 3x + 2)]$

  $= 12 + x [1 + x \{2 + x (4x + 3)\}]$

  It can be converted in C expression as:

  12 + x * (1 + x * (2 + x * (4 * x + 3)))

Here, the operations are minimized: only 4 multiplication operations and 4 addition operations. So the unwanted burden of compiler is minimized.