UNIT 1:

Introduction to Programming

Basic Model of Computation

There are six basic computational models such as Turing, von Neumann, dataflow, applicative, object-based, predicate logic-based, etc. These models are known as basic models because they can be declared using a basic set of abstractions.

A hierarchy of subclasses can be defined for each of the basic models to any extent required. For instance, if the process abstraction is introduced into the von Neumann model, new subclasses can be defined according to whether all processes might have access to a global data space (Shared-memory subclass) or the processes could have their own local data spaces, and access remote data spaces by sending messages (message-passing subclass).

Computational models, languages, and architecture classes

Computational Model	Language Class	Architecture Class
Turing	'Type 0' languages	
von Neumann	Imperative	Von Neumann
Dataflow	Single Assignment (dataflow)	Dataflow
Applicative	Functional	Reduction
Object-based	Object-Oriented	Object-Oriented
Predicate logic based	Logic Programming	So far unnamed

For more details click on this link- https://www.tutorialspoint.com/what-are-the-basic-computational-models#:~:text=There%20are%20six%20basic%20computational,a%20basic%20set%20of%20abstractions.

Algorithms

An algorithm is a sequence of instructions that are carried out in a predetermined sequence in order to solve a problem or complete a work. A function is a block of code that can be called and executed from other parts of the program.

Features of the algorithm

It defines several important features of the algorithm, including:

- **Inputs**: Algorithms must receive inputs that can be represented as values or data.
- **Output**: The algorithm should produce some output. It can be a consequence of a problem or a solution designed to solve it.
- **Clarity**: Algorithms must be precisely defined, using unambiguous instructions that a computer or other system can follow unambiguously.
- **Finiteness**: The algorithm requires a limited steps. It means that it should be exited after executing a certain number of commands.
- **Validity**: The algorithm must be valid. In other words, it should be able to produce a solution to the problem that the algorithm is designed to solve in a reasonable amount of time.
- **Effectiveness:** An algorithm must be effective, meaning that it must be able to produce a solution to the problem it is designed to solve in a reasonable amount of time.
- **Generality:** An algorithm must be general, meaning that it can be applied to a wide range of problems rather than being specific to a single problem.

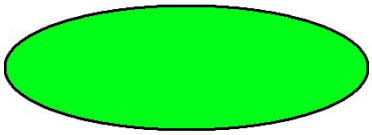
For more details click on this link- https://www.javatpoint.com/algorithm-in-c-language#:~:text=An%20algorithm%20is%20a%20sequence,other%20parts%20of%20the%20program.

Flow-charts

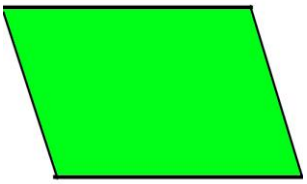
Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing.
The process of drawing a flowchart for an algorithm is known as “flowcharting”.

Basic Symbols used in Flowchart Designs

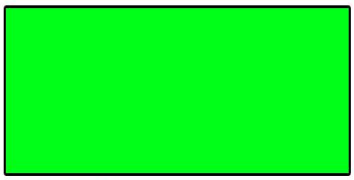
1. **Terminal:** The oval symbol indicates Start, Stop and Halt in a program’s logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



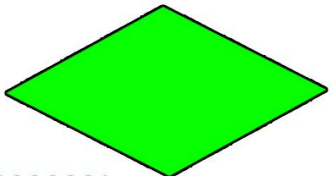
- **Input/Output:** A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.



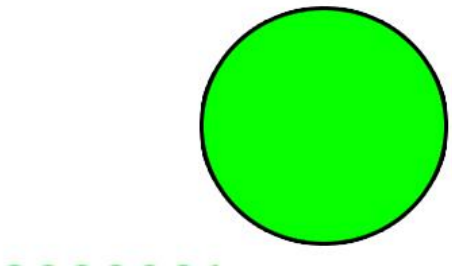
- **Processing:** A box represents arithmetic instructions. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action or process symbol.



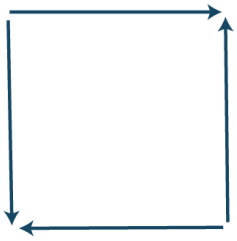
- **Decision** Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



- **Connectors:** Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. It is represented by a circle.



- **Flow lines:** Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.



Rules For Creating Flowchart :

A flowchart is a graphical representation of an algorithm.it should follow some rules while creating a flowchart

Rule 1: Flowchart opening statement must be ‘start’ keyword.

Rule 2: Flowchart ending statement must be ‘end’ keyword.

Rule 3: All symbols in the flowchart must be connected with an arrow line.

Rule 4: The decision symbol in the flowchart is associated with the arrow line.

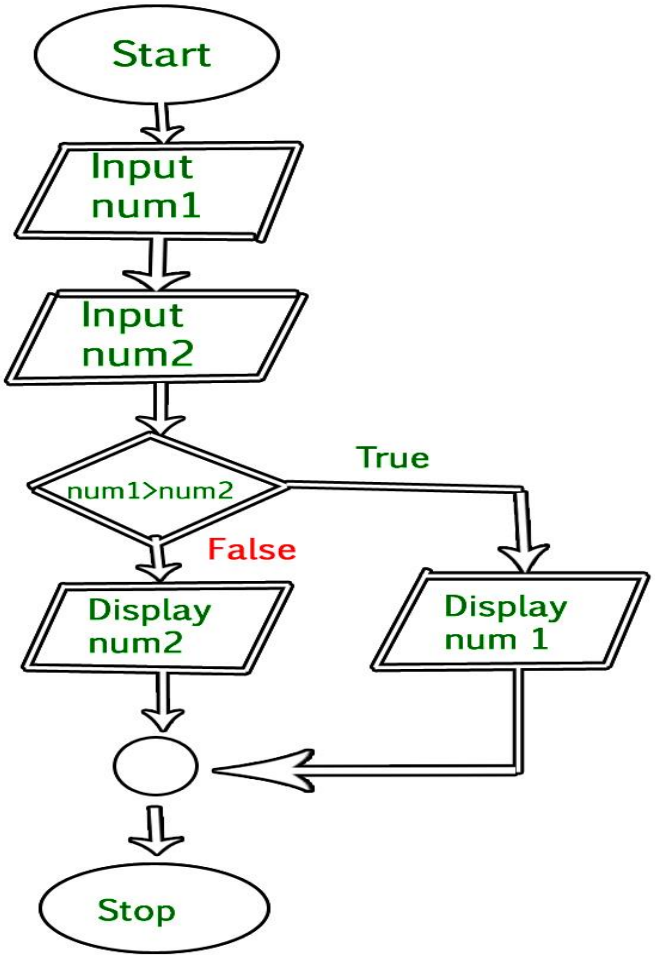
Advantages of Flowchart:

- Flowcharts are a better way of communicating the logic of the system.
- Flowcharts act as a guide for blueprint during program designed.
- Flowcharts help in debugging process.
- With the help of flowcharts programs can be easily analyzed.
- It provides better documentation.
- Flowcharts serve as a good proper documentation.
- Easy to trace errors in the software.
- Easy to understand.
- The flowchart can be reused for inconvenience in the future.
- It helps to provide correct logic.

Disadvantages of Flowchart:

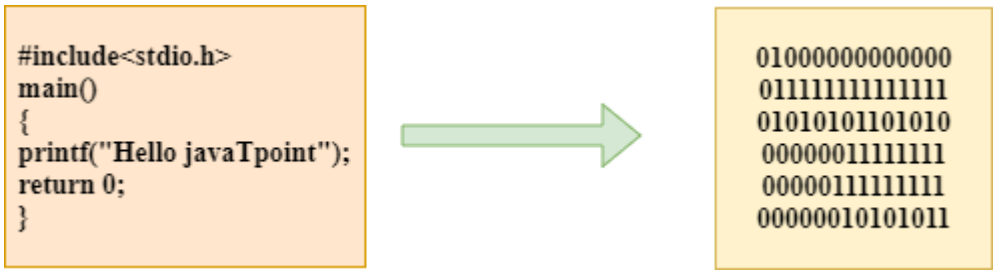
- It is difficult to draw flowcharts for large and complex programs.
- There is no standard to determine the amount of detail.
- Difficult to reproduce the flowcharts.
- It is very difficult to modify the Flowchart.
- Making a flowchart is costly.
- Some developer thinks that it is waste of time.
- It makes software processes low.
- If changes are done in software, then the flowchart must be redrawn

Example : Draw a flowchart to input two numbers from the user and display the largest of two numbers



Compilation

The compilation is a process of converting the source code into object code. It is done with the help of the compiler. The compiler checks the source code for the syntactical or structural errors, and if the source code is error-free, then it generates the object code.

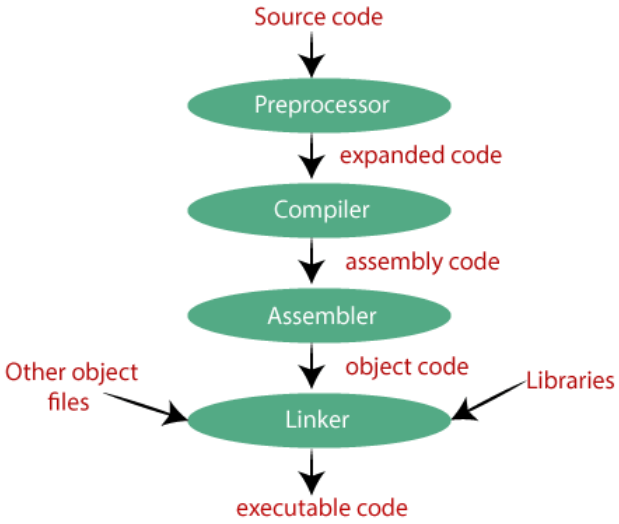


The c compilation process converts the source code taken as input into the object code or machine code. The compilation process can be divided into four steps, i.e., Pre-processing, Compiling, Assembling, and Linking.

The preprocessor takes the source code as an input, and it removes all the comments from the source code. The preprocessor takes the preprocessor directive and interprets it. For example, if **<stdio.h>,** the directive is available in the program, then the preprocessor interprets the directive and replace this directive with the content of the **'stdio.h'** file.

The following are the phases through which our program passes before being transformed into an executable form:

- **Preprocessor**
- **Compiler**
- **Assembler**
- **Linker**



Preprocessor

The source code is the code which is written in a text editor and the source code file is given an extension ".c". This source code is first passed to the preprocessor, and then the preprocessor expands this code. After expanding the code, the expanded code is passed to the compiler.

Compiler

The code which is expanded by the preprocessor is passed to the compiler. The compiler converts this code into assembly code. Or we can say that the C compiler converts the pre-processed code into assembly code.

Assembler

The assembly code is converted into object code by using an assembler. The name of the object file generated by the assembler is the same as the source file. The extension of the object file in DOS is '.obj,' and in UNIX, the extension is 'o'. If the name of the source file is **'hello.c',** then the name of the object file would be 'hello.obj'.

Linker

Mainly, all the programs written in C use library functions. These library functions are pre-compiled, and the object code of these library files is stored with '.lib' (or '.a') extension. The main working of the linker is to combine the object code of library files with the object code of our program. Sometimes the situation arises when our program refers to the functions defined in other files; then linker plays a very important role in this. It links the

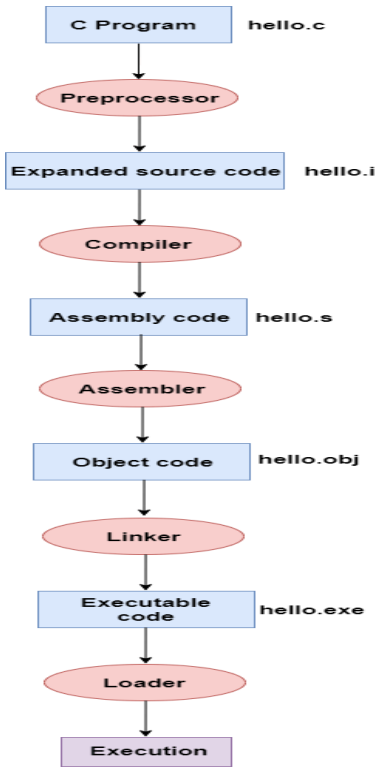
object code of these files to our program. Therefore, we conclude that the job of the linker is to link the object code of our program with the object code of the library files and other files. The output of the linker is the executable file. The name of the executable file is the same as the source file but differs only in their extensions. In DOS, the extension of the executable file is '.exe', and in UNIX, the executable file can be named as 'a.out'. For example, if we are using printf() function in a program, then the linker adds its associated code in an output file.

Let's understand through an example.

hello.c

```
1. #include <stdio.h>
2. int main()
3. {
4.     printf("Hello javaTpoint");
5.     return 0;
6. }
```

Now, we will create a flow diagram of the above program:



In the above flow diagram, the following steps are taken to execute a program:

- Firstly, the input file, i.e., **hello.c,** is passed to the preprocessor, and the preprocessor converts the source code into expanded source code. The extension of the expanded source code would be **hello.i.**
- The expanded source code is passed to the compiler, and the compiler converts this expanded source code into assembly code. The extension of the assembly code would be **hello.s.**
- This assembly code is then sent to the assembler, which converts the assembly code into object code.
- After the creation of an object code, the linker creates the executable file. The loader will then load the executable file for the execution.

Linking and Loading

Linking and Loading are utility programs that play an important role in the execution of a program. Linking intakes the object codes generated by the assembler and combines them to generate the executable module. On the other hand, the loading loads this executable module to the main memory for execution.

What is Loading?

To bring the program from secondary memory to main memory is called Loading. It is performed by a loader. It is a special program that takes the input of executable files from the linker, loads it to the main memory, and prepares this code for execution by a computer. There are two types of loading in the operating system:

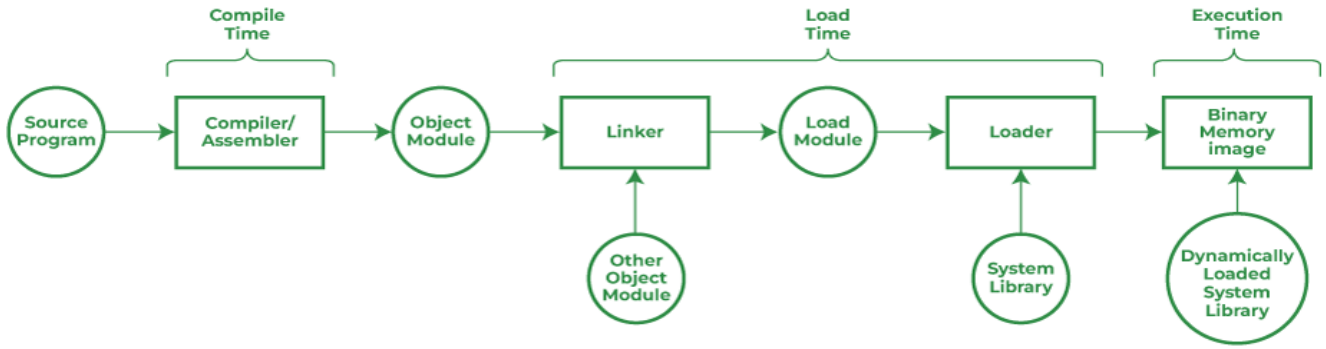
- **Static Loading:**
    - Loading the entire program into the main memory before the start of the program execution is called static loading.
    - If static loading is used then accordingly static linking is applied.
- **Dynamic Loading:**
    - Loading the program into the main memory on demand is called dynamic loading.
    - If dynamic loading is used then accordingly dynamic linking is applied.

What is Linking?

Establishing the linking between all the modules or all the functions of the program in order to continue the program execution is called linking. Linking is a process of collecting and maintaining pieces of code and data into a single file. Linker also links a particular module into the system library. It takes object modules from the assembler as input and forms an executable file as output for the loader. Linking is performed at both compile time when the source code is translated into machine code, and load time, when the program is loaded into memory by the loader. Linking is performed at the last step in compiling a program.

- **Static Linking:**
    - A statically linked program takes constant load time every time it is loaded into the memory for execution.
    - Static linking is performed by programs called linkers as the last step in compiling a program. Linkers are also called link editors.
    - In static linking, if any of the external programs has changed then they have to be recompiled and re-linked again else the changes won't reflect in the existing executable file.
- **Dynamic Linking:**
    - Dynamic linking is performed at run time by the operating system.
    - In dynamic linking, this is not the case and individual shared modules can be updated and recompiled. This is one of the greatest advantages dynamic linking offers.
    - In dynamic linking load time might be reduced if the shared library code is already present in memory.

For more information please refer to the Static and Dynamic Linking in Operating Systems article.



Differences between Linking and Loading

Linking	Loading
The process of collecting and maintaining pieces of code and data into a single file is known as Linking in the operating system.	Loading is the process of loading the program from secondary memory to the main memory for execution.
Linking is used to join all the modules.	Loading is used to allocate the address to all executable files and this task is done by the loader.
Linking is performed with the help of Linker. In an operating system, Linker is a program that helps to link object modules of a program into a single object file. It is also called a link editor.	A loader is a program that places programs into memory and prepares them for execution.
Linkers are an important part of the software development process because they enable separate compilation. Apart from that organizing a large application as one monolithic source file, we can decompose it into smaller, more manageable modules that can be modified and compiled separately.	The loader is responsible for the allocation, linking, relocation, and loading of the operating system.

Testing and Debugging

Testing: Testing is the process of verifying and validating that a software or application is bug-free, meets the technical requirements as guided by its design and development, and meets the user requirements effectively and efficiently by handling all the exceptional and boundary cases.
Debugging: Debugging is the process of fixing a bug in the software. It can be defined as identifying, analyzing, and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging. Below is the difference between testing and debugging:

Testing	Debugging
Testing is the process to find bugs and errors.	Debugging is the process of correcting the bugs found during testing.
It is the process to identify the failure of implemented code.	It is the process to give absolution to code failure.
Testing is the display of errors.	Debugging is a deductive process.
Testing is done by the tester.	Debugging is done by either programmer or the developer.
There is no need of design knowledge in the testing process.	Debugging can’t be done without proper design knowledge.
Testing can be done by insiders as well as outsiders.	Debugging is done only by insiders. An outsider can’t do debugging.
Testing can be manual or automated.	Debugging is always manual. Debugging can’t be automated.

Testing	Debugging
It is based on different testing levels i.e. unit testing, integration testing, system testing, etc.	Debugging is based on different types of bugs.
Testing is a stage of the software development life cycle (SDLC).	Debugging is not an aspect of the software development life cycle, it occurs as a consequence of testing.
Testing is composed of the validation and verification of software.	While debugging process seeks to match symptoms with cause, by that it leads to error correction.
Testing is initiated after the code is written.	Debugging commences with the execution of a test case.
Testing process based on various levels of testing-system testing, integration testing, unit testing, etc.	Debugging process based on various types of bugs is present in a system.

Documentation

The complete program is divided into different sections, which are as follows −

- **Documentation Section** − Here, we can give commands about program like author name, creation or modified date. The information written in between/* */ or // is called as comment line. These lines are not considered by the compiler while executing.
- **Link section** − In this section, header files that are required to execute the program are included.
- **Definition section** − Here, variables are defined and initialised.
- **Global declaration section** − In this section, global variables are defined which can be used throughout the program.

- **Function prototype declaration section** − This section gives information like return type, parameters, names used inside the function.
- **Main function** − The C Program will start compiling from this section. Generally, it has two major sections called as declaration and executable section.
- **User defined section** − User can define his own functions and performs particular task as per the user's requirement.

General form of a 'C' program

The general form of a C program is as follows −

```
/* documentation section */
preprocessor directives
global declaration
main ( ){
  local declaration
  executable statements
}
returntype function name (argument list){
  local declaration
  executable statements
}
```

Example

Following is the C program using function with arguments and without return value to perform addition −

```
#include<stdio.h>
void main(){
  //Function declaration - (function has void because we are not returning any values for function)//
  void sum(int,int);
  //Declaring actual parameters//
  int a,b;
  //Reading User I/p//
  printf("Enter a,b :");
  scanf("%d,%d",&a,&b);
```

```
  //Function calling//
  sum(a,b);
}
void sum(int a, int b){//Declaring formal parameters
  //Declaring variables//
  int add;
  //Addition operation//
  add= a+b;
  //Printing O/p//
  printf("Addition of a and b is %d",add);
}
```

Output

You will see the following output −

Enter a,b :5,6

Addition of a and b is 11

Character set

A character denotes any alphabet, digit or special symbol used to represent information. Valid alphabets, numbers and special symbols allowed in C are

Alphabets	A, B, ….., Y, Z a, b, ……, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ‘ ! @ # % ^ & * ( ) _ - + = | \ { } [ ] : ; " ' < > , . ? /

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords.

Variables

Variables are the data items whose values may vary during the execution of the program. A specific location or address in the memory is allocated for each variable and value of that variable is stored in that location.

Rules for declaring variable name

- Variable name may be a combination of alphabet, digits, or underscores and its length should not exceed eight characters.
- First character must be an alphabet.
- No commas or blank spaces are allowed in variable name.
- Among the special symbols, only underscore can be used in variable name.
- Example: emp_age and item_ 4

Variable Declaration and values Assignment

All the variables must be declared before their use. Declaration does two things:

- It tells the compiler what the variable name is.
- It specifies the type of data, the variable will hold.

A variable declaration has the form:

```
type_specifier list_of_variables;
```

here type_specifier is one of the valid data types. List_ of_variables is a comma separated list of identifiers representing the program variables.

Examples:

```
int
a,b,c;
char
ch;
```

To assign values to the variable, assignment operator (=) is used. Assignment is of the form

```
variable_name = value;
```

Example:

```
int  A,B;
A=10;
B=50;
```

It is also possible to assign a value to the variable at the time of declaration.

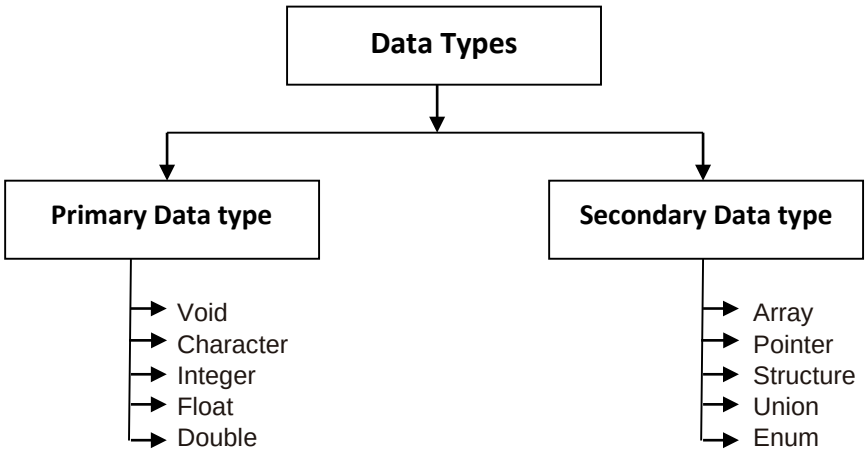
```
Data type variable_name=value;
```

Example

```
int
A=10;
char
ch=
'@';
```

The Data Types

The power of a programming language depends, among other things, on the range of different types of data it can handle. Data values passed in a program may be of different types. The C data types can be broadly divided into two categories.



Primary Data Types

There are five primary data types in C language.

- **char** Char data type is used to store a single character belonging to the defined character set of 'C' language.
- **int** int data type is used to store signed integers, for example, positive or negative integers.
- **float** float data type is used to store real numbers with single. precision (precision of six digits after decimal points).
- **double** double data type stores real numbers with double precision, that is, twice the storage space required by float.

- **void** void data type is used to specify empty set containing no values.

Character

Data Type	Storage Space	Format	Range of Values
Char	1 byte	%c	ASCII character set (-128 to 127)
unsigned char	1 byte	%c	ASCII character set (0 to 255)

Integer

Data Type	Storage Space	Format	Range of Values
Int	2 bytes	%d ,%i	**-32768 to +32767**
unsigned int	2 bytes	%u	**0 to 65535**
long int	4 bytes	%ld	**-2147483648 to +2147483648**
long unsigned int	4 bytes	%lu	**0 to 4,294,967,295**

Real Numbers

Data Type	Storage Space	Format	Range of Values
Float	4 bytes	%f	**-3.4*1038 to +3.4*1038**
Double	8 bytes	%lf	**$-1.7*10^{308}$ to $+1.7*10^{308}$**
long double	10 bytes	%Lf	**$-1.7*10^{4932}$ to $+1.7*10^{4932}$**

Identifiers

Identifiers are user defined word used to name of entities like variables, arrays, functions, structures etc. Rules for naming identifiers are:

1) name should only consists of alphabets (both upper and lower case), digits and underscore (_) sign.
2) first characters should be alphabet or underscore
3) name should not be a keyword
4) since C is a case sensitive, the upper case and lower case considered differently, for example code, Code, CODE etc. are different identifiers.
5) Identifiers are generally given in some meaningful name such as value, net_salary, age, data etc. An identifier name may be long, some implementation recognizes only first eight characters, most recognize 31 characters. ANSI standard compiler recognize 31 characters. Some invalid identifiers are 5cb, int, res#, avg no etc.

Operators

An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation on data stored in variables. The variables that are operated are termed as operand.

C operators can, be classified into a number of categories. They include:

- Arithmetic operators
- Relational operators
- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operators
- Special operators

Arithmetic operators

C provides all the basic arithmetic operators. There are five arithmetic operators in C.

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication

/	Division
%	Remainder after integer division

The division operator (/) requires the second operand as non zero, though the operands not be integers.

The operator (%) is known as modulus operator. It produces the remainder after the division of the two operands. The second operands must be non zero.

Example: if a = 25, b = 4

then a + b = 29

a - b =21

a * b = 100

a/b = 6 (decimal parts truncated)

a % b = 1

Relational Operators

Relational operator is used to compare two operands to see whether they are equal to each other, unequal, or one is greater or lesser than the other.

The operands can be variable, constants, or expression and the result is a numerical value. There are six relational operators.

Operator	Meaning
==	equality
!=	Not equal to
<	less than
>	greater than
< =	less than or equal to
> =	greater than or equal to

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

ae-1 relational operator ae-2

Example:

Expressions	Result
4.5 <= 10	1 (True)

4.5 < -10	0 (False)
-35 >= 0	0 (False)
10 < 7+5	1 (True)

Logical Operators

Logical operators are used to combine two or more relational expressions. C provides three different logical operators.

Operator	Meaning
&&	Logical and
I I I I	Logical or
!	Logical not

Logical AND (&&)

The result of Logical AND will be true only if both operands are true.

Result_exp1	Result_exp2	Final Result
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

e.g. if (10 > 15 && 5 < 10) ⟶ 0 (false)

if (5 < 10 && 10 > 6) ⟶ 1 (true)

Logical OR (||)

The result of Logical OR will be true if any one operand is true

Result_exp1	Result_exp2	Final Result
0	**0**	**0**
0	**1**	**1**
1	**0**	**1**

1	**1**	**1**

e.g. if (10 > 15 || 5 < 10) 1 (True)

if (5 > 10 || 9 < 6) 0 (false)

if (5 > 10 || 2 > 6) 0 (false)

Logical NOT (!)

Logical NOT (!) is used to reverse the value of the expression.

Expr_Result	Final Result
0	1
1	0

e.g. if (! (5 > 10)) → 1 (TRUE)

if (! (10 > 15 && 5 < 10)) → 1 (TRUE)

if (! (5 > 10 || 9 < 6)) → 1 (TRUE)

Assignment Operator

Assignment operators are used to assign the result of an expression to a variable. The most commonly used assignment operator is (=).

An expression with assignment operator is of the following form:

Identifier = expression;

Example:

```
#include <stdio.h>

void main()
    {
        int i ;
    i = 5;
```

```
        printf ("%d", i);
        i = i + 10;
    printf
    ("\n%d", i):
}
```

Output will be : 5
10

Expressions like i = i+10; , i = i-5; , i = i*2; , i = i/6, and i = i% 10 can be rewritten using shorthand assignment operators.

Increment and Decrement Operators

'C' has two very useful operators + + and -- called increment and decrement operators respectively. These are generally not found in other languages. These operators are called unary operators as they require only one operand. This operand should necessarily be variables not constant.

The increment operator (+ +) adds one to the operand while the decrement operator (--) subtracts one from the operand.

These operators may be used in two ways.

1) Prefix :

When the operator used before the operand, it is termed as prefix. e.g. ++A , --B in this case the value of operand follow First Change Then Use (F.C.T.U) concept

2) Postfix:

When the operator used before the operand, it is termed as prefix.
e.g. A++, B-- in this case the value of operand follow First Use Then Change (F.U.T.C) concept.

Example:

Postfix

```
int N=10, R;
R = N++;        // post increment
printf("R=%d \n N=%d ", R , N);
```

it will produce **output** : R=10 (Because before increment, value will assign first)
N=11

Prefix

```
int N=10, R;
R = ++N;      // pre increment
printf("R=%d \n N=%d ", R , N);
```

it will produce **output:** R=11 (Because value will increment first, then value will assign)

N=11

Conditional or Ternary Operator

A ternary operator is one which contains three operands. The only ternary operator available in C language is conditional operator pair " **? :** " . It is of the form

exp1 ? exp2 : exp3 ;

This operator works as follows. Exp1 is evaluated first. If the result is true then exp2 is executed otherwise exp3 is executed.

Bitwise operators

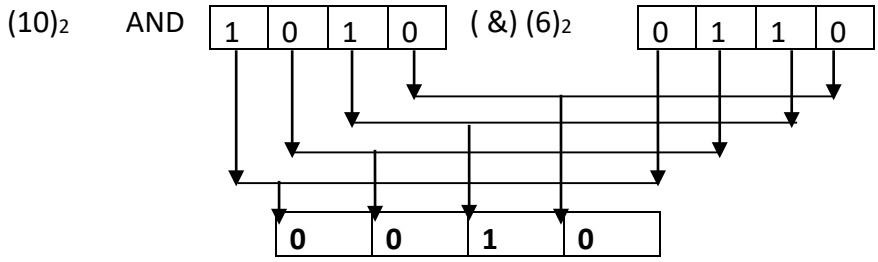
Bitwise operators are used to manipulate of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double data type.

Some Bitwise Operators	
Operator	Meaning
&	Bitwise Logical AND
|	Bitwise Logical OR
, ^	Bitwise Logical XOR
<<	Left shift
>>	Right shift
~	One's complement

Bitwise Logical AND (&)

e.g.

```
int  N=10;   N= N & 6
;
                Printf("N=%d",N);
```



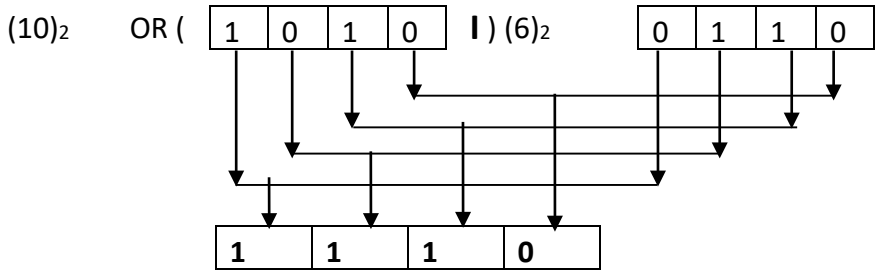
This will produce output:

N=2

Bitwise Logical OR (|)

e.g.

```
int  N=10;   N= N | 6
;
                Printf("N=%d",N);
```

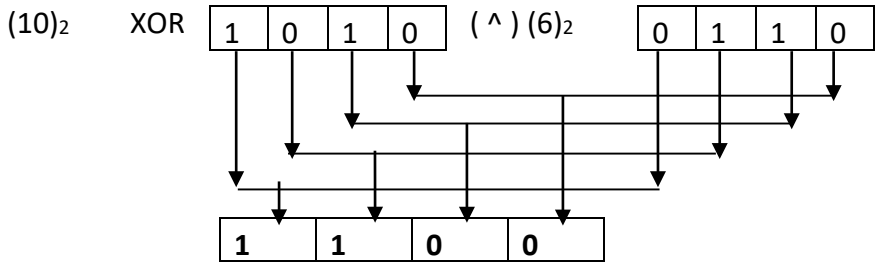


This will produce output: N=14

Bitwise XOR (^)

The Bitwise XOR operator produce result 1 if both bits are different otherwise it will produce 0. e.g.

```
int  N=10;   N= N ^ 6
;
                Printf("N=%d",N);
```



This will produce output:
N=12

Shift Operators

Shift operators use a shift distance (number of shifts) as the right-hand operand and the value which is to be shifted as the left-hand operand.

Left Shift (<<)

Left shift operator shifted bits left. When bits are shifted left, zero is filled in from right. Each left-shift corresponds to multiplication of the value by 2.

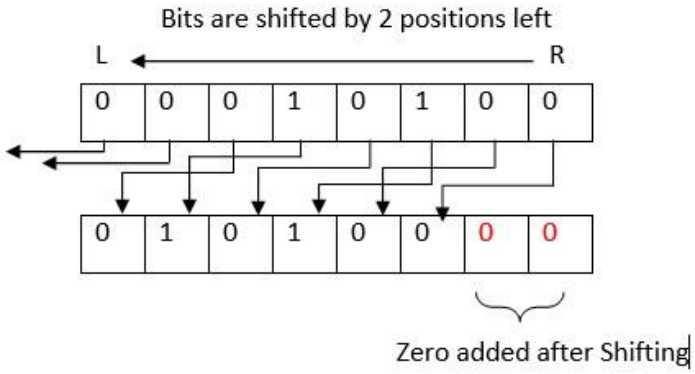
It will give result $V=V*2^n$, where **n** is the number of bits to be shifted and **v** is the value to be shifted.

. e.g.

int N=20;
N=N<<2;

it will produce Value
$N=N*2^2$
$N=20*2_2$
N=20 * 4
N=80

$(20)_{10}$ ➔ $(1\ 0\ 1\ 0\ 0)_2$



After Shifting the Value $(20)_{10}$ - $(0\ 0\ 0\ 1\ 0\ 1\ 0\ 0)_2$ will be $(80)_{10}$ - $(0\ 1\ 0\ 1\ 0\ 0\ 0\ 0)_2$

Right Shift (>>)

Right shift operator shifted bits right . when bits are shifted right, sign bit are filled in from left. Each right-shift corresponds to division of the value by 2.

It will give result $\mathbf{V=V/2^n}$, where **n** is the number of bits to be shifted and **v** is the value to be shifted. e.g.

int N=20;
N=N>>2;

it will produce Value

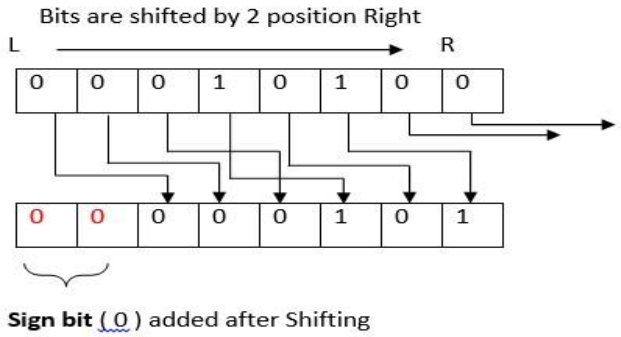
$N=N/2^2$

$N=20 / 2_2$

$N=20 / 4$

$N=5$

$(20)_{10}$ ⍰ $(1\ 0\ 1\ 0\ 0)_2$



Sign bit for Positive number is zero (0) , and for Negative number (1)

After Shifting the Value $(20)_{10}$ ⍰ $(0\ 0\ 0\ 1\ 0\ 1\ 0\ 0)_2$ will be $(5)_{10}$ ⍰ $(0\ 0\ 0\ 0\ 0\ 1\ 0\ 1)_2$

Special operators

'C' language supports some special operators such as comma operator, sizeof operator, pointer operators (& and *), and member selection operators (. and ->). Pointer operators will be discussed while introducing pointers while member selection operator will be discussed with structures and union. Let us discuss comma operator and sizeof operator.

comma operator

This operator is used to link the related expressions together.

Example:

```
intval, x, y;
value = (x= 10, y = 5, x+y);  it first assigns 10 to x then 5
to y finally sum x + y to value.
```

sizeof operator

The sizeof operator is a compile time operator and when used with an operand, it returns the number of bytes the operand occupies. The operand may be a variable, constant, or a data type qualifier.

Example:

```
int n;
n = sizeof
(int);  printf
("n=%d \n",
n); n = sizeof
(double);
printf
("n=%d", n);
```

Output: n = 2
n = 8

Expressions

An expression is a combination of variables, constants, operators and function call. It can be arithmetic, logical and relational.

for example:- int z= x+y // arithmatic expression

a>b //relational

a==b // logical

func(a, b) // function call

Expressions consisting entirely of constant values are called *constant expressions*. So, the expression 121 + 17 – 110 is a constant expression because each of the terms of the expression is a constant value. But if i were declared to be an integer variable, the expression 180 + 2 – j would not represent a constant expression.

Constants

Constants are the fixed values that remain unchanged during the execution of a program and are used in assignment statements. Constants are stored in variables.

To declare any constant, the syntax is

```
const data type var_name = value ;
```

In 'C' language, there are five types of constants.

1	character	character constant consist of a single character, single digit, or a single special symbol enclosed within a pair of single inverted commas. i.e. ‘A’,’%’
2	integer	An integer constant refers to a sequence of digits. There are. Three types of integers: decimal, octal and hexadecimal. In octal notation, write (0) immediately before the octal representation. For example: 0.76, -076. In hexadecimal notation, the constant is preceded by 0x. Example: 0x3E, -0x3E. No commas or blanks are allowed in integer constants.
3	real	A real constants consist of three parts : Sign (+ or 0) , Number portion (base), exponent portion i.e. +.72 , +72 , +7.6E+2 , 24.3e-5
4	string	A string constant is a sequence of one or more characters enclosed within a pair of double quotes (" "). If a single character is enclosed within a pair of double quotes, it will also be interpreted as a string constant. Examples: "Welcome To Microtek \ n” , “a” , “123”
5	logical	A logical constant can have either a true value or a false value. In 'C all the non zero values are treated as true value while 0 is treated as false.

Basic Input and Output in C

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.

scanf()

The scanf() method, in C, reads the value from the console as per the type specified.

Syntax: ***scanf(“%X”, &variableOfXType);***

*where **%X** is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and **&** is the address operator in C, which tells the compiler to change the real value of this variable, stored at this address in the memory.*

printf()

The printf() method, in C, prints the value passed as the parameter to it, on the console screen.

Syntax: ***printf("%X", variableOfXType);***

where ***%X*** *is the format specifier in C. It is a way to tell the compiler what type of data is in a variable and* ***&*** *is the address operator in C, which tells the compiler to change the real value of this variable, stored at this address in the memory.*

How to take input and output of basic types in C?

The basic type in C includes types like int, float, char, etc. Inorder to input or output the specific type, the **X** in the above syntax is changed with the specific format specifier of that type. The Syntax for input and output for these are:

- **Integer:**
  Input: scanf("%d", &intVariable);
  Output: printf("%d", intVariable);
- **Float:**
  Input: scanf("%f", &floatVariable);
  Output: printf("%f", floatVariable);
- **Character:**
  Input: scanf("%c", &charVariable);
  Output: printf("%c", charVariable);

Simple 'C' programs

For more C programs visit - https://www.programiz.com/c-programming/examples