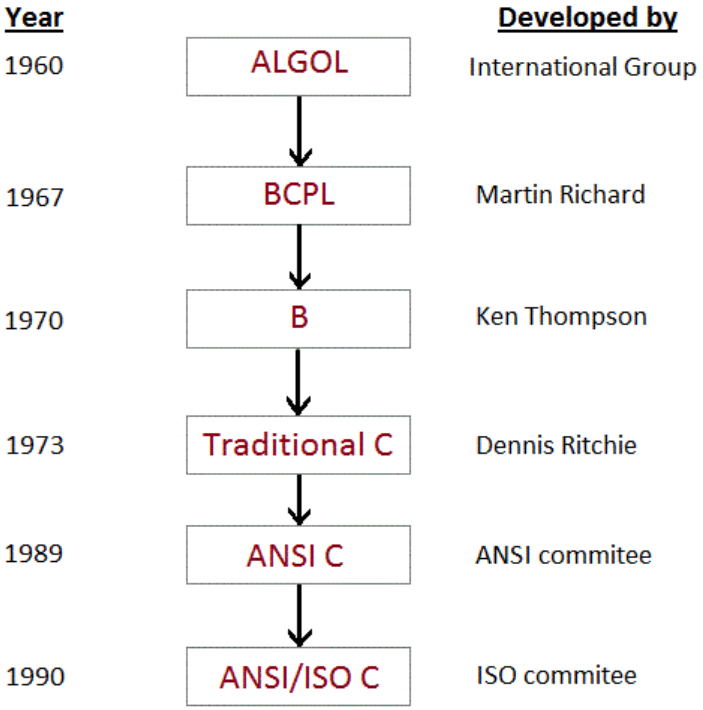
Chapter -1: Introduction to C

1.1 History of C

C is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented on the DEC PDP-11 computer in 1972.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a DEC PDP-7. BCPL and B are "type less" languages whereas C provides a variety of data types.

In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as the K&R standard.



The UNIX operating system, the C compiler, and essentially all UNIX application programs have been written in C. C has now become a widely used professional language for various reasons −

Easy to learn
Structured language
It produces efficient programs
It can handle low-level activities
It can be compiled on a variety of computer platforms

1.2 C Standards

C has been standardized by the American National Standards Institute (ANSI) since 1989 and subsequently by the International Organization for Standardization (ISO). C standards are as follows: C89, C99 and C11

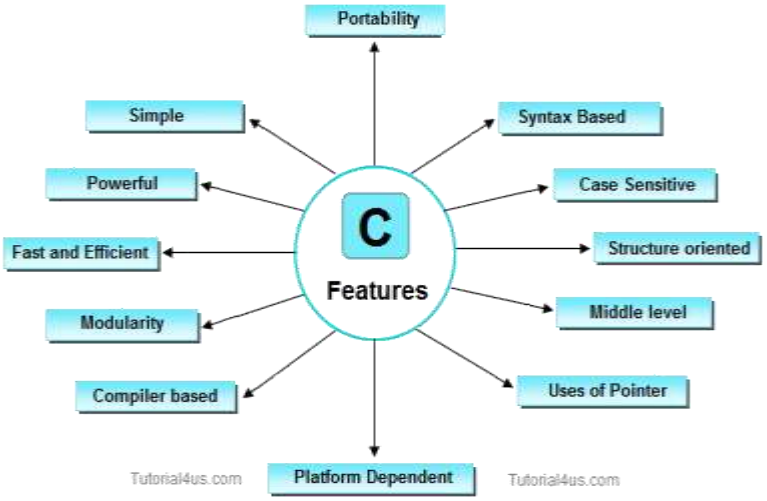
1.3 Facts about C

C was invented to write an operating system called UNIX.
C is a successor of B language which was introduced around the early 1970s.
The language was formalized in 1988 by the American National Standard Institute (ANSI).
Today C is the most widely used and popular System Programming Language.
Most of the state-of-the-art software have been implemented using C.
Today's most popular Linux OS and RDBMS MySQL have been written in C.

1.4 Why use C?

C was initially used for system development work, particularly the programs that make-up the operating system. It was adopted as a system development language because it produces code that runs nearly as fast as the code written in assembly language.
Some examples of the use of C are as follows.
Operating Systems
Language Compilers
Assemblers
Text Editors
Print Spoolers
Network Drivers
Modern Programs
Databases
Language Interpreters
Utilities

1.5 Features of C Language



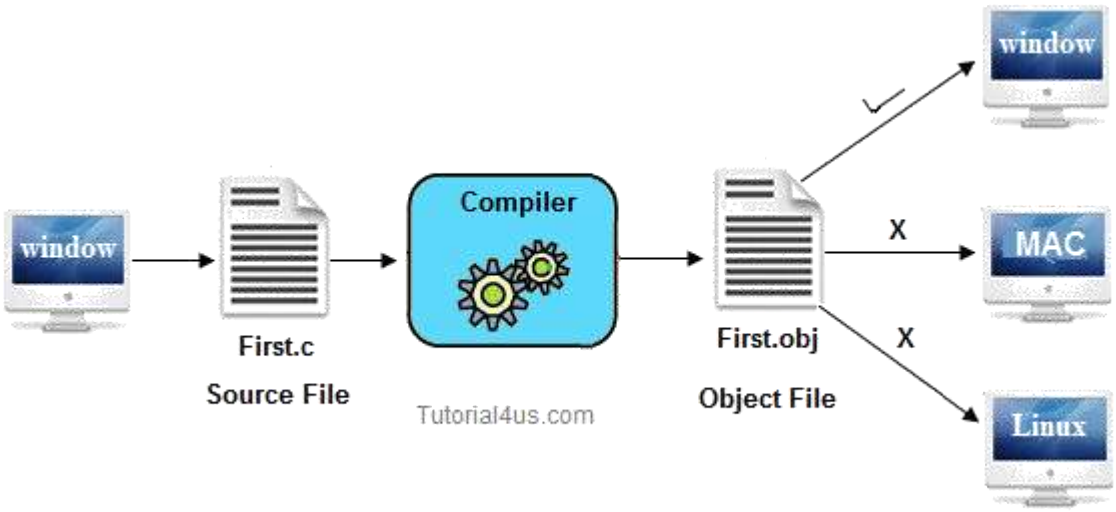
- **Simple**

Every c program can be written in simple English language so that it is very easy to understand and developed by programmer.

- **Platform dependent**

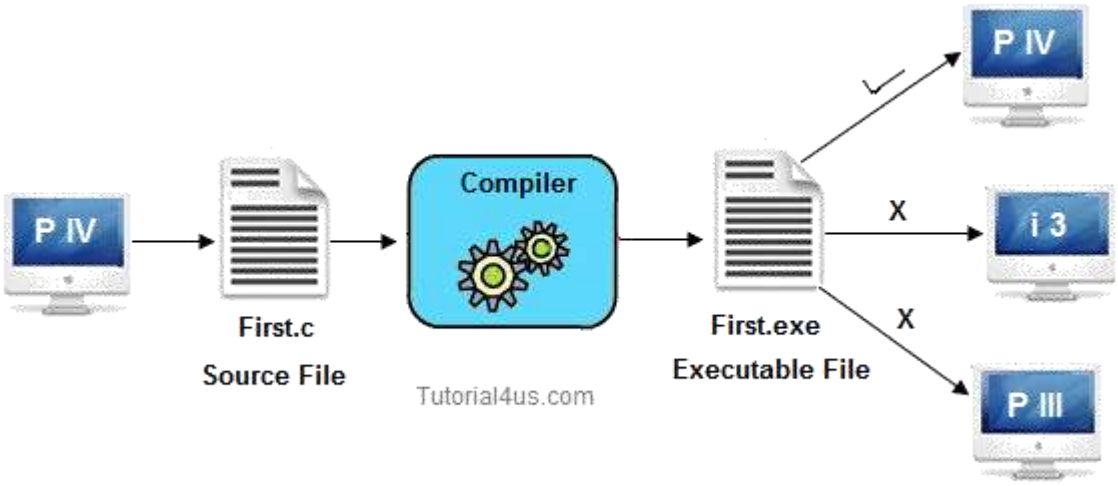
A language is said to be platform dependent whenever the program is execute in the same operating system where that was developed and compiled but not run and execute on other operating system. C is platform dependent programming language.

Note: .obj file of C program is platform dependent.



- **Portability**

It is the concept of carrying the instruction from one system to another system. In C Language .c file contain source code, we can also edit this code. .exe file contain application, only we can execute this file. When we write and compile any C program on window operating system that program easily run on other window based system.



When we can copy .exe file to any other computer which contain window operating system then it works properly, because the native code of application an operating system is same. But this exe file is not execute on other operating system.

- **Powerful**

C is a very powerful programming language. It has a wide verity of data types, functions, control statements, decision making statements, etc.

- **Structure oriented**

C is a Structure oriented programming language. Structure oriented programming language aimed on clarity of program, reduce the complexity of code, using this approach code is divided into sub-program/subroutines. These programming have rich control structure.

- **Modularity**

It is concept of designing an application in subprogram that is procedure oriented approach. In C programming, we can break our code into subprograms.
Example: Calculator program can be divided into sub programs.

```
void sum()
{
 .....
 .....
}
void sub()
{
 .....
}
```

- **Case sensitive**

It is a case sensitive programming language. In C programming 'break and BREAK' both are different.
If any language treats lower case latter separately and upper case latter separately than they can be called as case sensitive programming language [Example c, c++, java, .net are sensitive programming languages.] otherwise it is called as case insensitive programming language [Example HTML, SQL is case insensitive programming languages].

- **Middle level language**

C programming language can supports two level programming instructions with the combination of low level and high level language that's why it is called middle level programming language.

- **Compiler based**

C is a compiler based programming language that means without compilation no C program can be executed. First we need compiler to compile our program and then execute.

- **Syntax based language**

C is a strongly tight syntax based (strongly typed) programming language. If any language follow rules and regulation very strictly known as strongly tight syntax based language. Examples: C, C++, Java, .net etc. If any language not follow rules and regulation very strictly known as loosely tight syntax based language. Example: HTML.

- **Efficient use of pointers**

Pointer is a variable which hold the address of another variable. It directly has access to memory address of any variable due to which, the performance of application is improved. In C language also concept of pointer are available.

1.6 Program structure

C program basically consists of the following parts.
- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World".

```
#include <stdio.h>
int main() {
  /* my first program in C */
  printf("Hello, World\n");
  return 0;
}
```

Let us take a look at the various parts of the above program.

The first line of the program #include <stdio.h> is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.

The next line int main() is the main function where the program execution begins.

The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.

The next line printf(...) is another function available in C which causes the message "Hello, World!" to be displayed on the screen.

The next line return 0; terminates the main() function and returns the value 0.

Compilation and Execution of C Program

Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps:

1) Open a text editor and add the above-mentioned code.
2) Save the file as hello.c
3) Open a command prompt and go to the directory where you have saved the file.
4) Type gcc hello.c and press enter to compile your code.

If there are no errors in your code, a.exe (in Windows) or a.out (in Ubuntu) is created and the prompt is returned back to allow the user to enter next command.

Note: GCC : GNU Compiler Collections
GNU : GNU Not Unix

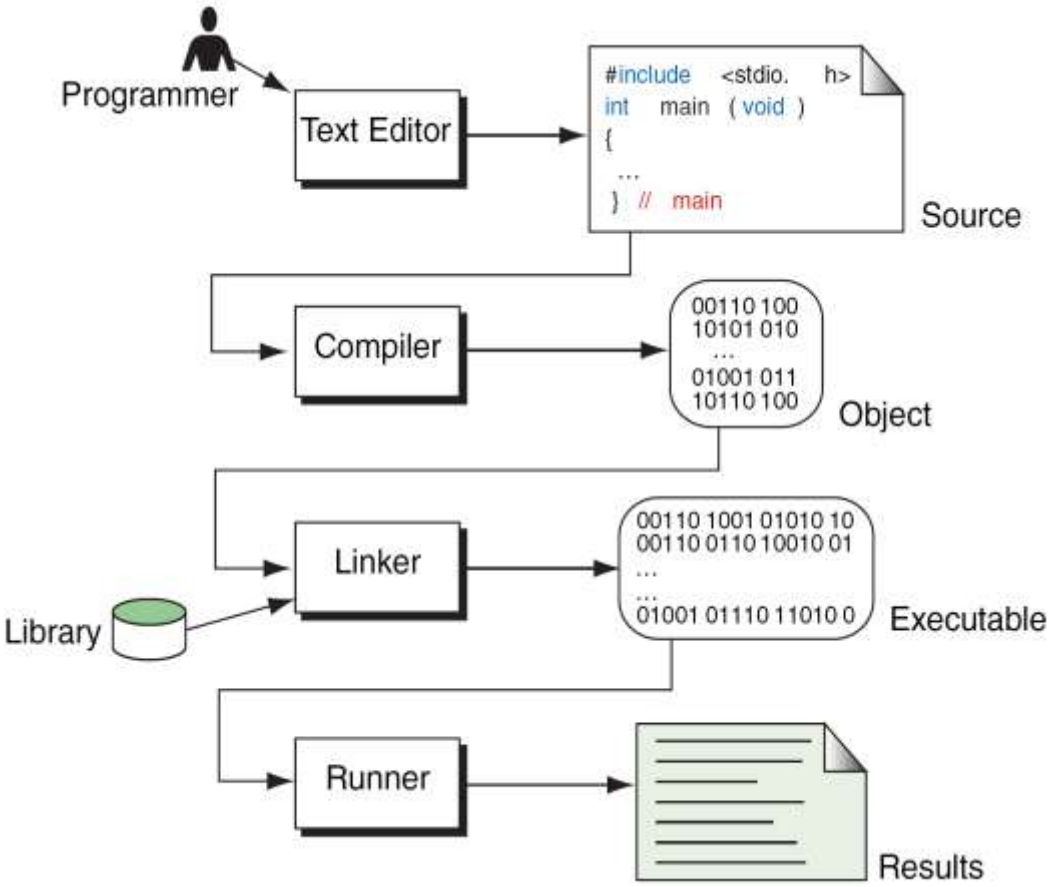
In Ubuntu, type ./a.out and press enter. This results in output.
In Windows, type a.exe and press enter. This results in output.
Output: "Hello World"

To name the executable, i.e., a.exe or a.out, we need to use -o option during compilation.
Example: gcc hello.c -o h

To ignore the warnings, we need to use -w option.
Example: gcc -w hello.c

1.7 Program Development Life Cycle (PDLC)

Phases involved in PDLC are Editing, Pre-processing, Compilation, Linking, Loading and execution by CPU.



1. Pre-Processing

This is the very first stage through which a source code passes. In this stage, the following tasks are done:

a. Macro substitution
b. Comments are stripped off
c. Expansion of the included files

```
$ gedit first.c
#include <stdio.h>
#define STRING "Hello World"
int main(void)
{
/* Using a macro to print 'Hello World'*/
printf(STRING);
return 0;
}
```

To understand Pre-processing better, compile the above ‘first.c’ program using flag -E, which will print the preprocessed output to stdout.

```
$ gcc  -E first.c
```

Even better, you can use flag ‘-save-temps’ as shown below. ‘-save-temps’ flag instructs compiler

to store the temporary intermediate files used by the gcc compiler in the current directory.
$ gcc -save-temps first.c -o First

So when we compile the program first.c with -save-temps flag, we get the following intermediate files in the current directory (along with the 'First' executable)

In Ubuntu, type ls
$ ls
first.i
first.s
first.o

In Windows, type dir /p
$ dir /p
first.i
first.s
first.o

The Pre-processed output is stored in the temporary file that has the extension .i (i.e 'first.i' in this example)
Now, lets open first.i file and view the content.

$ gedit first.i
.....
846 "/usr/include/stdio.h" 3 4
886 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__));
extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__)) ;
extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__));
.....
916 "/usr/include/stdio.h" 3 4
2 "first.c" 2
int main(void)
{
printf("Hello World");
return 0;
}
.....

In the above output, you can see that the source file is now filled with lots of information, but still at the end of it we can see the lines of code written by us. Let's analyze on these lines of code first.
1. The first observation is that the argument to printf() now contains directly the string "Hello World" rather than the macro. In fact the macro definition and usage has completely disappeared. This proves the first task that all the macros are expanded in the Pre-processing stage.
2. The second observation is that the comment that we wrote in our original code is not there. This proves that all the comments are stripped off.

3. The third observation is that beside the line '#include' is missing and instead of that we see whole lot of code in its place. So it is safe to conclude that stdio.h has been expanded and literally

included in our source file. Hence we understand how the compiler is able to see the declaration of printf() function. On searching first.i file, we can see that the function printf is declared as:

extern int printf (__const char *__restrict __format, ...);

2. Compiling

After the compiler is done with the pre-processor stage, the next step is to take first.i as input, compile it and produce an intermediate compiled output. The output file for this stage is 'first.s'. The output present in first.s is assembly level instructions. Open the first.s file in an editor and view the content.

```
$ gedit first.s
......
.file " first.c"
.section .rodata
.LC0:
.string "Hello World"
.text
.globl main
.type main, @function
......
```

A quick look at the file concludes that this assembly level output is in some form of instructions which the assembler can understand and convert it into machine level language.

3. Assembly

At this stage the first.s file is taken as an input and an intermediate file first.o is produced. This file is also known as the object file. This file is produced by the assembler that understands and converts '.s' file with assembly instructions into a '.o' object file which contains machine level instructions. At this stage only the existing code is converted into machine language. The function calls like printf() are not resolved. Since the output of this stage is a machine level file (first.o). So we cannot view the content of it. If you still try to open the first.o and view it, you'll see something that is totally not readable.

```
$ gedit first.o
……
^?ELF^B^A^A^@^@^@^@^@^@^@^@^@^A^@>^@^A^@^@^@^@^@^@^@^@^@^@^
@^@^@^@^@^@^@^@^@0^

^@UH<89>å¸^@^@^@^@H<89>Ç¸Hello World^@^@GCC: (Ubuntu 4.4.3-4ubuntu5)
4.4.3^@^
…….
```

The only thing we can explain by looking at the first.o file is about the string ELF. ELF stands for executable and linkable format.

This is a relatively new format for machine level object files and executable that are produced by gcc. Prior to this, a format known as a.out was used. ELF is said to be more sophisticated format than a.out

Note: If you compile your code without specifying the name of the output file, the output file produced has name 'a.out' but the format now have changed to ELF. It is just that the default executable file name remains the same.

4. Linking

This is the final stage at which all the linking of function calls with their definitions are done. As discussed earlier, till this stage gcc doesn't know about the definition of functions like printf(). Until the compiler knows exactly where all of these functions are implemented, it simply uses a place-holder for the function call. It is at this stage, the definition of printf() is resolved and the actual address of the function printf() is plugged in. The linker also does some extra work: It combines some extra code to our program that is required when the program starts and when the program ends.

5. Loader

Loader transfers the executable code from Hard Disk to RAM(Random Access Memory) when command is issued to execute the code.

1.8 Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens

```
printf("Hello, World! \n");
```

The individual tokens are −

```
printf
(
"Hello, World! \n"
)
;
```

Semicolons

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Given below are two different statements −

```
printf("Hello, World! \n");
return 0;
```

Comments

Comments are helping texts in C program and they are ignored by the compiler.

```
// single line comment in C
/* multiline comment C */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

An identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9). C does not allow punctuation characters such as @, $, and % within identifiers. C is a case-sensitive programming language. Thus, Manpower and manpower are two different identifiers in C.

Here are some examples of acceptable identifiers −

mohd zara abc move_name a_123
myname50 _temp j a23b9 retVal

Keywords

The following list shows some of the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

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

DataTypes

The amount of storage to be reserved for the specified variable.

Significance of data types are given below:

1) Memory allocation
2) Range of values allowed
3) Operations that are bound to this type
4) Type of data to be stored

Data types are categorized into **primary and non-primary (secondary) types**.

Primary ---> int, float, double, char, void

Secondary--> Derived and User-defined types

Derived-->Arrays

User-defined--> struct, union, enum, typedef

Data types can be extended using qualifiers.

---> Size Qualifiers

---> Sign Qualifiers

Size Qualifiers:

long
short

Examples:

```
long int i;
short int s;
long long int d;
```

Note: Some of the qualifiers cannot be applied to some of the types.

Sizeof operator:
Used to get the size of the type on that particular machine. Refer the below code which is compiled using gcc on windows.

```
#include<stdio.h>
int main()
{
        printf("%d\n",sizeof(float));           //4
        printf("%d\n",sizeof(int));             //4
        printf("%d\n",sizeof(double));          //8
        printf("%d\n",sizeof(char));            //1
        printf("%d\n",sizeof(short int));       //2
        printf("%d\n",sizeof(long int));        //4
        printf("%d\n",sizeof(long long int)); //8
        printf("%d\n",sizeof(long double));  //12
        printf("%d",sizeof(long long double));//12
}
```

C standards follow the below Rules.

sizeof(short int)<=sizeof(int)<=sizeof(long int)<=sizeof(long long int)
sizeof(float)<=sizeof(double)<=sizeof(long double)

Sign Qualifiers:

```
unsigned        //No bit is used for representing the sign
signed          // Most significant bit is used for representing the sign.
```

Examples:

```
unsigned int i;
signed int s;
signed char d;
```

Refer the below codes which demonstrate sign qualifiers.

Code 1:

```
#include<stdio.h>
int main()
{
        signed int i=23;
        printf("the number is %u\n",i);         // 23 is interpreted as unsigned integer.
        unsigned int j=-23;
        printf("the number is %u\n",j);         // -23 is interpreted as unsigned integer.
        printf("Both using %d and %d",i,j);   // interpret i and j as integer
        signed float f;         // Error: sign qualifiers cannot be used with float
        return 0;
}
```

Code 2:

```
#include<stdio.h>
int main()
{
        signed char i='a';// only for completeness signed and unsigned available with char type
        printf("the number is %u",i);
```

```
        unsigned char j='b';
        printf("the number is %u\n",j);
        printf("Both using %d and %d",i,j);
        return 0;
}
```

Variable

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types, there will be the following basic variable types

Type	Description
char	Typically a single byte. This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. It specifies a data type and contains a list of one or more variables of that type as follows:
type variable_list;

Here, type must be a valid C data type including char, int, float, double, or any user-defined object and variable_list may consists of one or more identifier names separated by commas. Some valid declarations are shown here:

```
#1. int    i, j, k;
#2. char   c, ch;
#3. float  f, salary;
#4. double d;
```

The line #1 declares and defines the variables i, j, and k, which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:
type variable_name = value;
Some examples are given below:

```
int d = 3, f = 5;          // definition and initializing d and f.
char x = 'x';              // the variable x has the value 'x'.
```

For definition without an initializer, the initial value of all variables are undefined.

Lvalues and Rvalues in C

There are two kinds of expressions in C: lvalue and rvalue

lvalue − Expressions that refer to a memory location are called Location values or "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment.

rvalue − The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment.

Variables are lvalues and so they may appear on the left-hand side of an assignment. Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side. Take a look at the following valid and invalid statements −

```
int g = 20; // valid statement
10 = 20; // invalid statement; would generate compile-time error
```

Constants

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. Constants are of different types.

1) Numeric Constants:
   integer
   real

2) Non-numeric Constants
   char
   string

3) Symbolic constants - To be discussed Later

4) Enum constants - To be discussed Later

Note: Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal. An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of integer literals:

```
85         /* decimal */
0213       /* octal */
0x4b       /* hexadecimal */
30         /* int */
30u        /* unsigned int */
30l        /* long */
30ul       /* unsigned long */
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.
While representing in decimal form, you must include the decimal point, the exponent, or both. While representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Character Constants

Character literals are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of char type. A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

String Literals

String literals or constants are enclosed in double quotes " and ". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.
You can break a long line into multiple lines using string literals and separating them using white spaces. Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"

"hello, \
dear"

"hello, " "d" "ear"
```

1.9 Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. Operator is a symbol used for calculations or evaluations

Classification of operators **based on the number of operands.**

1) Unary
2) Binary
3) Ternary

1) Unary
   +, -, ++, --, sizeof ,&
2) Binary
   +, -, *, /, %
3) Ternary
   ?:

Classification of operators **based on the operation on operands**

1) Arithmetic Operators includes increment and decrement operators
   +, -, *, /, %, ++, --
2) Relational
   ==, !=, >=, >, <=, <
3) Logical
   &&, ||, !
4) Assignment
   =
5) Short-hand
   +=, -=, *=, /=, %=
6) Bitwise
   &, |, ^, >>, <<
7) misc operators
   sizeof , &, *

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 10 and variable B holds 20.

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Increment and Decrement Operators

There are two types of Increment (++) and Decrement (--) Operators.

1. **Pre - increment and Pre-decrement** operators: First increment the value and then use it in the expression.

2. **Post - increment and Post - decrement** operators: First use the value in the expression and then increment.

When ++ and -- are used in stand-alone expression, pre and post increment and decrement doesn't make difference. Refer the below code which demonstrate stand-alone expressions

```
#include<stdio.h>
int main(){
{
   int i=5;
```

```
    int j=5;
    printf("Beginning i value is %d\n",i);
    printf("Beginning j value is %d\n",i);
    ++i;i++;
    printf("Later i value is %d\n",i);
    j--;
    --j;
    printf("Later j value is %d\n",i);
}
```

Refer the below code which demonstrates the usage of ++ and -- in an expression.

```
#include<stdio.h>
int main()
{
    int a=34; int b;
    printf("Beginning a value is %d\n",a);
    b=a++; // Use the value of a in the expression and then increment
    // b=++a;                  // First increment and then use the value in the expression
    printf("b is %d and a is %d\n",b,a);
    // same works for pre and post decrement operators
}
```

Note: The behaviour of increment and decrement operator in an expression is undefined. It depends on compiler and machine architecture.

Evaluation of expressions in printf() is from Right to Left and Printing is done from Left to Right.

```
#include<stdio.h>
int main()
{
    int i=10;
    printf("%d %d %d %d\n",i,i--,--i,i);
    //printf("%d %d %d %d\n",i--,i++,i--,i);
    //printf("%d %d %d \n",++i,i,++i);
    //printf("%d %d %d \n",--i,i--,i--);
    //printf("%d %d %d %d\n",++i,i,i++,++i);
    return 0;
}
```

Relational Operators

The following table shows all the relational operators supported by C. Assume variable A holds 10 and variable B holds 20.

==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0.

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61, i.e,. 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment Operators

The following table lists the assignment operators supported by the C language.

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Misc. Operators

There are a few other important operators including sizeof and ? : supported by the C Language.

sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

Example:

x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

1.10 Input and Output

There are **two types of input and output functions**

1. Formatted I/O functions
2. Unformatted functions

Formatted I/O functions

For interactive input and output, scanf()[formatted input] and printf()[formatted output] functions are used.

scanf():

A predefined function in "stdio.h" header file.In the C programming language, scanf is a function that reads formatted data from stdin (i.e, the standard input stream, which is usually the keyboard) and then writes the results into the arguments given.

The scanf function has the following prototype/signature:

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

where

int (integer) is the return type

format is a string that contains the type specifier(s). Refer format specifiers table.

"..." (ellipsis) indicates that the function accepts a variable number of arguments; each argument must be a memory address where the converted result is written to. A simple type

specifier consists of a percent (%) symbol and an alpha character that indicates the type.
The function works by reading input from the standard input stream and then scans the contents of "format" for any format specifiers, trying to match the two. On success, the function writes the result into the arguments passed.

Example: If the function call is scanf("%c%d", &var1, &var2); and the user types "a1", the function will write "a" into "var1" and "1" into "var2". If the function call, however, is scanf("%x", &var); the same input will be read as the hexadecimal number "a1," which is 161 in decimal.

The function returns the following value:
>0 — The number of items converted and assigned successfully.
0 — No item was assigned.
<0 — Read error encountered or end-of-file (EOF) reached before any assignment was made.

printf():
The C library function printf() sends formatted output to stdout. Is also 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.
On success, it returns the number of characters successfully written on the output. On failure, a negative number is returned.

The scanf function has the following prototype/signature:
int printf(const char *format, ...)

Variations in scanf() is explained through code 1 and code 2.
Code 1:

```
#include<stdio.h>
int main()
{

        int d;
        printf("enter a number\n");
        scanf("%d",&d);                 // ampersand(&) is used because %d represents single location. We need to get the address of the location to store some element in that


// If the user enters 23pesu , then 23 is copied to address of ele. scanf extracts the value corresponding to the format specifier if available in the beginning of user input.
// If the user enters 23 pesu , then 23 is copied to address of ele. scanf terminates when white space characters(\t,\n and space) are encountered.
// If user enters pesu23, nothing is copied to address of ele. So junk is displayed.
// If user enters multiple spaces and then 23, leading spaces are ignored and 23 is copied to ele.
        int ele;
        printf("enter a number again\n");
        scanf("%d",&ele);
        printf("entered is %d",ele);
```

```
	char str[200];		// variable str of char array type
	printf("enter the string\n");
	scanf("%s",str);	//  & not used. Because str contains the base address of the array. since %s, it represents successive locations
	printf("U enterd %s",str);


	// disadvantage of scanf: If the entered string is I like pes, it reads everything till white space character. So, only I is stored . scanf skips white space characters(tab,space or new line)
	// this can be avoided by using perplex character [next example]

	//If the input is lot of spaces and then characters, then scanf ignores the whitespaces and reads everything from non-whitespace till whitespace character
	return 0;
}
```

Code 2:

```
#include<stdio.h>
int main()
{
	char str[200];
	printf("enter the string\n");
	scanf("%[^\n]",str);	// till \n read everything .. perplex character
	printf("U enterd %s",str);

	/*
	char str[200];
	printf("enter the string\n");
	scanf("%s\n",str);	// When taking the input from the user, till u press some non-white space character, cursor will be on terminal only.
	printf("U enterd %s and the length of it is %d\n",str,strlen(str));
	*/

	/*
	char str[200];
	printf("enter the string\n");
	scanf("%[^a-z]",str);	// read till a-z is encountered. input 123%tvs
	printf("U enterd %s",str);
	*/

	/*
	char str[200];
	printf("enter the string\n");
	scanf("%[a-z,0-9]",str);	// input can have only a-z and 0-9. If something other than this is encounterd, skip the rest. Give different inputs and observe the output
	printf("U enterd %s",str);
	*/
```

```
        // to include space in the above input
        /*
        char str[200];
        printf("enter the string\n");
        scanf("%[a-z,0-9 ]",str);        // Give different inputs and observe the output
        printf("U enterd %s",str);
        */


        /*
        char d[20];
        printf("enter the string\n");
        scanf("%*s%s",d);
        printf("U entered %s",d);
        */

        // To include some of the symbols in the input
        /*
        char str[200];
        printf("enter the string\n");
        scanf("%[a-z,0-9*!@%/]",str);        // Give different inputs and observe the output
        printf("U enterd %s",str);
        */
        return 0;
}
```

Unformatted I/O functions

1. gets() and puts()
2. getchar() and putchar()

1. gets() and puts() functions

gets(str): Reads a line from stdin and stores it into the string pointed to by str. It stops when either the newline character is read or when the end-of-file is reached, whichever comes first. Returns str on success, and NULL on error or when end of file occurs.

puts(str): Writes a string to stdout up to but not including the null character. A newline character is appended to the output. If successful, non-negative value is returned. On error, the function returns EOF.

Note: These two functions are dangerous to use. Because we are not specifying the number of bytes str can hold. So it is safe to use fgets() with stdin as last argument and fputs() with stdout as last argument. Refer to notes on Files for fgets() and fputs().

Code:

```
#include<stdio.h>
int main()
{
        int x = 17;
        printf("%05d", x);  /* "00017" */
        char str[200];
```

```
        printf("enter the string\n");
        gets(str);        //spaces are also taken as a part of string
        //appends null character to the input string entered by the user
        // till \n read everything
        printf("U entered:");
        puts(str);        //prints until it finds null character
        printf("%d",strlen(str));

        /*
        char r[]={'x','y','a','d'};
        printf("%lu\n",sizeof(r));
        puts(r);          // behaviour is undefined because r may or may not have null character
        */

        /*
        char r[]={'x','y','\0','d'};
        printf("%lu\n",sizeof(r));
        puts(r);                  //There is \0. So xy is printed.
        */
        return 0;
}
```

2. getchar() and putchar()

getchar(): Used to get a character from stdin. This is equivalent to getc with stdin as its argument. Returns the character read as an unsigned char cast to an int or EOF on end of file or error.

putchar(): Writes a character specified by the argument char to stdout. Returns the character written as an unsigned char cast to an int or EOF on error.

Code:

```
#include<stdio.h>
int main()
{
printf("enter the character\n");
char ch=getchar();
printf("you entered ");
putchar(ch);
}
```

1.11 Format string

Used for the beautification of the output.The format string is of the form

% [flags] [field_width] [.precision] conversion_character

where components in brackets [] are optional. The minimum is therefore a % and a conversion character (e.g. %d)

A) Flags

-	The output is left justified in its field, not right justified (the default).
+	Signed numbers will always be printed with a leading sign (+ or -).
Space	Positive numbers are preceded by a space (negative numbers by a - sign).
0	For numeric conversions, pad with leading zeros to the field width.
#	An alternative output form. For **o**, the first digit will be '**0**'. For **x** or **X**, "**0x**" or "**0X**" will be prefixed to a non-zero result. For **e**, **E**, **f**, **F**, **g** and **G**, the output will always have a decimal point; for **g** and **G**, trailing zeros will not be removed.

B) Field Width

The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width, it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally ' ' (space), but is '**0**' if the zero padding flag (**0**) is present. If the field width is specified as *, the value is computed from the next argument.

C) Format Specifiers or Conversion Characters

In C programming we need lots of format specifier to work with various data types. Format specifiers define the type of data to be printed on standard output or the type of data to be read from the standard input.

Format specifier	**Description**	**Supported data types**
%c	Character	char
		unsigned char
%d	Signed Integer	short
		unsigned short
		int
		long
%e or %E	Scientific notation of float values	float
		double

%f	Floating point	float
%g or %G	Similar as %e or %E	float
		double
%hi	Signed Integer(Short)	short
%hu	Unsigned Integer(Short)	unsigned short
%i	Signed Integer	short
		unsigned short
		int
		long
%l or %ld or %li	Signed Integer	long
%lf	Floating point	double
%Lf	Floating point	long double
%lu	Unsigned integer	unsigned int
		unsigned long
%lli, %lld	Signed Integer	long long
%llu	Unsigned Integer	unsigned long long
%o	Octal representation of Integer.	short
		unsigned short
		int
		unsigned int
		long
%p	Address of pointer to void void *	void *
%s	String	char *
%u	Unsigned Integer	unsigned int

		unsigned long
%x or %X	Hexadecimal representation of Unsigned Integer	short
		unsigned short
		int
		unsigned int
		long
%n	Prints nothing	
%%	Prints % character	

```
#include<stdio.h>
int main()
{
        int i=34;
        float f=3.5;
        double d=78.6;
        char c='y';
        long double ld=67.6;
        printf("%d\n",i);
        printf("%f\n",f);
        printf("%lf\n",d);
        printf("%c\n",c);
        printf("%Lf\n",ld);
        return 0;
}
```

Note: If using windows, MINGW GCC, compile the below code with –D__USE_MINGW_ANSI_STDIO and run as usual.

Examples of Format String

```
x=34;
printf("%10dtest", x);  /* prints x on the right side of the 10-space field */
//        34test
printf("%-10d", x); /* prints X on the left side of the 10-space field */
//34        test
```

```
int width = 12;
int value = 3490;
printf("%*d\n", width, value);
```

// field width is specified as *, the value is computed from the next argument, which must be an int.

When it comes to floating point, you can also specify how many decimal places to print by making a field width of the form "x.y" where x is the field width (you can leave this off if you want it to be just wide enough) and y is the number of digits past the decimal point to print.

```
float f = 3.1415926535;
```

```
printf("%.2ftest", f);                  /* 3.14test */
printf("%7.3ftest", f);       //  3.141test   */
```

```
int x = 17;
printf("%05d", x);  //00017    // the remaining field width, fill with zeros.
```

```
int i=234567;
printf("%3d",i);// If field with is lesser than the number of letters in the data specified, data will never be lost. Field with is ignored
```

1.12 Escape sequences

Are special character constants which are represented by two key strokes and represents one character.

\n	newline	-- move cursor to next line
\t	tab space	
\r	carriage return	-- Move cursor to the beginning of that line
\a	alarm or bell	
\"	double quote	
\'	single quote	
\\	black slash	

```
#include<stdio.h>
int main()
{

        printf("Hello .. good morning\n");
        printf("Hello\t good morning\n");
        printf("good morning\rhello");
        printf("\n");
        printf("I like \"C\"\n");
        printf("Hello\a\a\a, Have a nice day");
        printf("\n");
        printf("This is \\unusual\\");

}
```