➢ C Language

Introduction

C is a procedural programming language. It was initially developed by Dennis Ritchie in the year 1972. It was mainly developed as a system programming language to write an operating system. The main features of C language include low-level access to memory, a simple set of keywords, and clean style, these features make C language suitable for system programmings like an operating system or compiler development.
Many later languages have borrowed syntax/features directly or indirectly from C language. Like syntax of Java, PHP, JavaScript, and many other languages are mainly based on C language. C++ is nearly a superset of C language (There are few programs that may compile in C, but not in C++).

History of C language

The base or father of programming languages is 'ALGOL.' It was first introduced in 1960. 'ALGOL' was used on a large basis in European countries. 'ALGOL' introduced the concept of structured programming to the developer community. In 1967, a new computer programming language was announced called as 'BCPL' which stands for Basic Combined Programming Language. BCPL was designed and developed by Martin Richards, especially for writing system software. This was the era of programming languages. Just after three years, in 1970 a new programming language called 'B' was introduced by Ken Thompson that contained multiple features of 'BCPL.' This programming language was created using UNIX operating system at AT&T and Bell Laboratories. Both the 'BCPL' and 'B' were system programming languages. In 1972, a great computer scientist Dennis Ritchie created a new programming language called 'C' at the Bell

Laboratories. It was created from 'ALGOL', 'BCPL' and 'B' programming languages. 'C' programming language contains all the features of these languages and many more additional concepts that make it unique from other languages.

Advantages of C

- C is the building block for many other programming languages.
- Programs written in C are highly portable.
- Several standard functions are there (like in-built) that can be used to develop programs.
- C programs are collections of C library functions, and it's also easy to add functions to the C library.
- The modular structure makes code debugging, maintenance, and testing easier.

Disadvantages of C

- C does not provide Object Oriented Programming (OOP) concepts.
- There are no concepts of Namespace in C.
- C does not provide binding or wrapping up of data in a single unit.
- C does not provide Constructor and Destructor.

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).
- The UNIX OS was totally written in C.
- 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.

C Keywords

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example:

```
int money;
```

Here, `int` is a keyword that indicates `money` is a variable of type `int` (integer).

As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

C keywords:

auto	double	int	Struct
break	Else	long	Switch
case	Enum	register	Typedef
char	extern	return	Union
continue	For	signed	Void
do	If	static	While
default	Goto	sizeof	Volatile

`const`	`float`	`short`	`Unsigned`

C Keywords

- ## C Identifiers

Identifier refers to name given to entities such as variables, functions, structures etc.

Identifiers must be unique. They are created to give a unique name to an entity to identify it during the execution of the program. For example:

```c
int money;
double accountBalance;
```

Here, `money` and `accountBalance` are identifiers.

- ## Rules for naming identifiers

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a letter or an underscore.
3. You cannot use keywords as identifiers.

There is no rule on how long an identifier can be. However, you may run into problems in some compilers if the identifier is longer than 31 characters.

➢ C Data Types

In this tutorial, you will learn about basic data types such as int, float, char etc. in C programming.

In C programming, data types are declarations for variables. This determines the type and size of data associated with variables. For example,

```
int myVar;
```

Here, `myVar` is a variable of `int` (integer) type. The size of `int` is 4 bytes.

Basic types

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier
`int`	at least 2, usually 4	`%d`
`char`	1	`%c`
`float`	4	`%f`
`double`	8	`%lf`
`short int`	2 usually	`%hd`
`unsigned int`	at least 2, usually 4	`%u`
`long int`	at least 4, usually 8	`%li`
`long long int`	at least 8	`%lli`
`unsigned long int`	at least 4	`%lu`
`unsigned long long int`	at least 8	`%llu`
`signed char`	1	`%c`
`unsigned char`	1	`%c`
`long double`	at least 10, usually 12 or 16	`%Lf`

- **int**

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, `0`, `-5`, `10`
We can use `int` for declaring an integer variable.

```
int id;
```

Here, `id` is a variable of type integer.

You can declare multiple variables at once in C programming. For example,

```
int id, age;
```

The size of `int` is usually 4 bytes (32 bits). And, it can take 2^{32} distinct states from `-2147483648` to `2147483647`.

- ### float and double

`float` and `double` are used to hold real numbers.

```
float salary;
double price;
```

In C, floating-point numbers can also be represented in exponential. For example,

`float normalizationFactor = 22.442e2;` The size of `float` (single precision float data type) is 4 bytes. And the size of `double` (double precision float data type) is 8 bytes.

char

Keyword `char` is used for declaring character type variables. For example,

```
char test = 'h';
```

The size of the character variable is 1 byte.

void

`void` is an incomplete type. It means "nothing" or "no type". You can think of void as **absent**.
For example, if a function is not returning anything, its return type should be `void`.
Note that, you cannot create variables of `void` type.

- ### short and long

If you need to use a large number, you can use a type specifier `long`. Here's how:

```
long a;
long long b;
long double c;
```

Here variables `a` and `b` can store integer values. And, `c` can store a floating-point number.

If you are sure, only a small integer (`[-32,767, +32,767]` range) will be used, you can use `short`.

```
short d;
```

You can always check the size of a variable using the `sizeof()` operator.

```
#include <stdio.h>
int main() {
  short a;
  long b;
  long long c;
  long double d;

  printf("size of short = %d bytes\n", sizeof(a));
  printf("size of long = %d bytes\n", sizeof(b));
      printf("size of long long = %d bytes\n",
sizeof(c));
      printf("size of long double= %d bytes\n",
sizeof(d));
      return 0;
    }
```

signed and unsigned

In C, `signed` and `unsigned` are type modifiers. You can alter the data storage of a data type by using them. For example,

```
unsigned int x;
int y;
```

Here, the variable `x` can hold only zero and positive values because we have used the `unsigned` modifier.

Considering the size of `int` is 4 bytes, variable `y` can hold values from -2^{31} to $2^{31}-1$, whereas variable `x` can hold values from `0` to $2^{32}-1$.

Other data types defined in C programming are:

- bool Type
- Enumerated type
- Complex types

Derived Data Types

Data types that are derived from fundamental data types are derived types. For example: arrays, pointers, function types, structures, etc.

➢ C Programming Operators

An operator is a symbol that operates on a value or a variable. For example: `+` is an operator to perform addition.C has a wide range of operators to perform various operations.

• C Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus

Operator	Meaning of Operator
*	multiplication
/	division
%	remainder after division (modulo division)

Example 1: Arithmetic Operators

```c
// Working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9,b = 4, c;

    c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
        printf("a-b = %d \n",c);
        c = a*b;
        printf("a*b = %d \n",c);
        c = a/b;
        printf("a/b = %d \n",c);
        c = a%b;
        printf("Remainder when a divided by b = %d \n",c);

        return 0;
    }
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

The operators `+`, `-` and `*` computes addition, subtraction, and multiplication respectively as you might have expected.
In normal calculation, `9/4 = 2.25`. However, the output is `2` in the program.

It is because both the variables `a` and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer `2` instead of `2.25`.
The modulo operator `%` computes the remainder. When `a=9` is divided by `b=4`, the remainder is `1`. The `%` operator can only be used with integers.
Suppose `a = 5.0`, `b = 2.0`, `c = 5` and `d = 2`. Then in C programming,

```
// Either one of the operands is a floating-point number

a/b = 2.5

a/d = 2.5

c/b = 2.5



// Both operands are integers

c/d = 2
```

- ## C Increment and Decrement Operators

C programming has two operators increment `++` and decrement `--` to change the value of an operand (constant or variable) by 1.
Increment `++` increases the value by 1 whereas decrement `--` decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 2: Increment and Decrement Operators

```c
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;

    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
        printf("++c = %f \n", ++c);
        printf("--d = %f \n", --d);

        return 0;
    }
```

Output

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Here, the operators `++` and `--` are used as prefixes. These two operators can also be used as postfixes like `a++` and `a--`. Visit this page to learn more about how increment and decrement operators work when used as postfix.

- ## C Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is `=`

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b

Operator	Example	Same as
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3: Assignment Operators

```c
// Working of assignment operators
#include <stdio.h>
int main()
{
    int a = 5, c;

    c = a;      // c is 5
    printf("c = %d\n", c);
    c += a;     // c is 10
        printf("c = %d\n", c);
        c -= a;     // c is 5
        printf("c = %d\n", c);
        c *= a;     // c is 25
        printf("c = %d\n", c);
        c /= a;     // c is 5
        printf("c = %d\n", c);
        c %= a;     // c = 0
        printf("c = %d\n", c);

        return 0;
    }
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

- ## C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	`5 == 3` is evaluated to 0
>	Greater than	`5 > 3` is evaluated to 1
<	Less than	`5 < 3` is evaluated to 0
!=	Not equal to	`5 != 3` is evaluated to 1
>=	Greater than or equal to	`5 >= 3` is evaluated to 1
<=	Less than or equal to	`5 <= 3` is evaluated to 0

Example 4: Relational Operators

```c
// Working of relational operators
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;

    printf("%d == %d is %d \n", a, b, a == b);
    printf("%d == %d is %d \n", a, c, a == c);
    printf("%d > %d is %d \n", a, b, a > b);
        printf("%d > %d is %d \n", a, c, a > c);
        printf("%d < %d is %d \n", a, b, a < b);
        printf("%d < %d is %d \n", a, c, a < c);
        printf("%d != %d is %d \n", a, b, a != b);
        printf("%d != %d is %d \n", a, c, a != c);
        printf("%d >= %d is %d \n", a, b, a >= b);
        printf("%d >= %d is %d \n", a, c, a >= c);
        printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);

    return 0;
}
```

Output

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c==5) || (d>5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `!(c==5)` equals to 0.

Example 5: Logical Operators

```c
// Working of logical operators

#include <stdio.h>
```

```c
int main()
{
    int a = 5, b = 5, c = 10, result;

    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) is %d \n", result);
        result = (a == b) && (c < b);
        printf("(a == b) && (c < b) is %d \n", result);
        result = (a == b) || (c < b);
        printf("(a == b) || (c < b) is %d \n", result);
        result = (a != b) || (c < b);
        printf("(a != b) || (c < b) is %d \n", result);
        result = !(a != b);
        printf("!(a == b) is %d \n", result);
        result = !(a == b);
        printf("!(a == b) is %d \n", result);

        return 0;
    }
```

Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

• C Bitwise Operators

- During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR

Operators	Meaning of operators
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

C "Hello, World!" Program

In this example, you will learn to print "Hello, World!" on the screen in C programming.

Program to Display "Hello, World!"

```c
#include <stdio.h>
int main() {
   // printf() displays the string inside quotation
   printf("Hello, World!");
   return 0;
}
```

Output

```
Hello, World!
```

➢C - Decision control statements in C programming language

Using decision control statements we can control the flow of program in such a way so that it executes certain statements based on the outcome of a condition (i.e. true or false). In C Programming language we have following decision control statements.

1. **if statement**
2. 2. **if-else & else-if statement**
3. 3. **switch-case statements**

1.If statement: The statements inside **if** body executes only when the condition defined by **if statement** is true. If the condition is false then compiler skips the statement enclosed in if's body. We can have any number of if statements in a C program.

- **syntax of if statement**:

The statements inside the body of "if" only execute if the given condition returns true. If the condition returns false then the statements inside "if" are skipped.

```
if (condition)
{
     //Block of C statements here
     //These statements will only execute if the condition is true
}
```

2.If-else statement: In this decision control statement, we have two block of statements. If condition results true then **if** block gets executed else statements inside **else** block executes. else cannot exist without if statement. In this tutorial, I have covered else-if statements as well.

If condition returns true then the statements inside the body of "if" are executed and the statements inside body of "else" are skipped.
If condition returns false then the statements inside the body of "if" are skipped and the statements in "else" are executed.

```
if(condition) {
   // Statements inside body of if
}
else {
   //Statements inside body of else
}
```

1. Switch-case statement: This is very useful when we need to evaluate multiple conditions. The switch block defines an expression (or condition) and case has a block of statements, based on the result of expression, corresponding case block gets executed. A switch can have any number of cases, however there should be only one default handler.

Syntax :

```
switch (variable or an integer expression)
{
     case constant:
     //C Statements
     ;
     case constant:
     //C Statements
     ;
     default:
     //C Statements
     ;
}
```

➢ Iteration Statements

Iteration statements repeat some processes during which some inner `Statements` are executed repeatedly until the `Condition` that limits the execution cycle becomes false or they are executed for all values of the same data type.

For Loop

Firstly, the Initialization is set up. Secondly, the `Condition` is evaluated and if its value is true, the `Statement` is executed. Finally, the `Iteration` is made.

During the next cycle of the loop, the `Condition` is evaluated again and if it is true, `Statement` is executed and `Iteration` is made. This way the process repeats until the `Condition` becomes false. Then the loop is terminated and the process continues with the other part of the program.If the `Condition` is false at the beginning, the process jumps over the `Statement` out of the loop.

```
for (Initialization;Condition;Iteration)
     Statement
```

- Example of For loop

```
#include <stdio.h>
int main()
{
   int i;
   for (i=1; i<=3; i++)
   {
       printf("%d\n", i);
   }
   return 0;
}
```

- Output:

```
1
2
3
```

- ## Do-While Loop

  Firstly, the `Statement` is executed. Secondly, the value of the `Condition` is evaluated. If its value is true, the `Statement` is executed again and then the `Condition` is evaluated again and the loop either continues (if it is true again) or stops and jumps to the next or higher level subprocesses (if it is false).

  Since the `Condition` is at the end of the loop, even if it is false at the beginning of the subprocess, the `Statement` is executed at least once.

Example of do while loop

```
#include <stdio.h>
int main()
{
        int j=0;
        do
        {
                printf("Value of variable j is: %d\n", j);
                j++;
        }while (j<=3);
        return 0;
}
```

Output:

```
Value of variable j is: 0
Value of variable j is: 1
Value of variable j is: 2
Value of variable j is: 3
```

While Loop

The processing depends on the value of the `Condition`. If its value is true, the `Statements` is executed and then the `Condition` is evaluated again and the processing either continues (if it is true again) or stops and jumps to the statement following the cycle (if it is false).Since the `Condition` is at the beginning of the loop, if it is false before entrance to the loop, the `Statements` is not executed at all and the loop is jumped over.

Example of while loop

```
#include <stdio.h>
int main()
{
   int count=1;
   while (count <= 4)
   {
     printf("%d ", count);
     count++;
   }
   return 0;
}
```

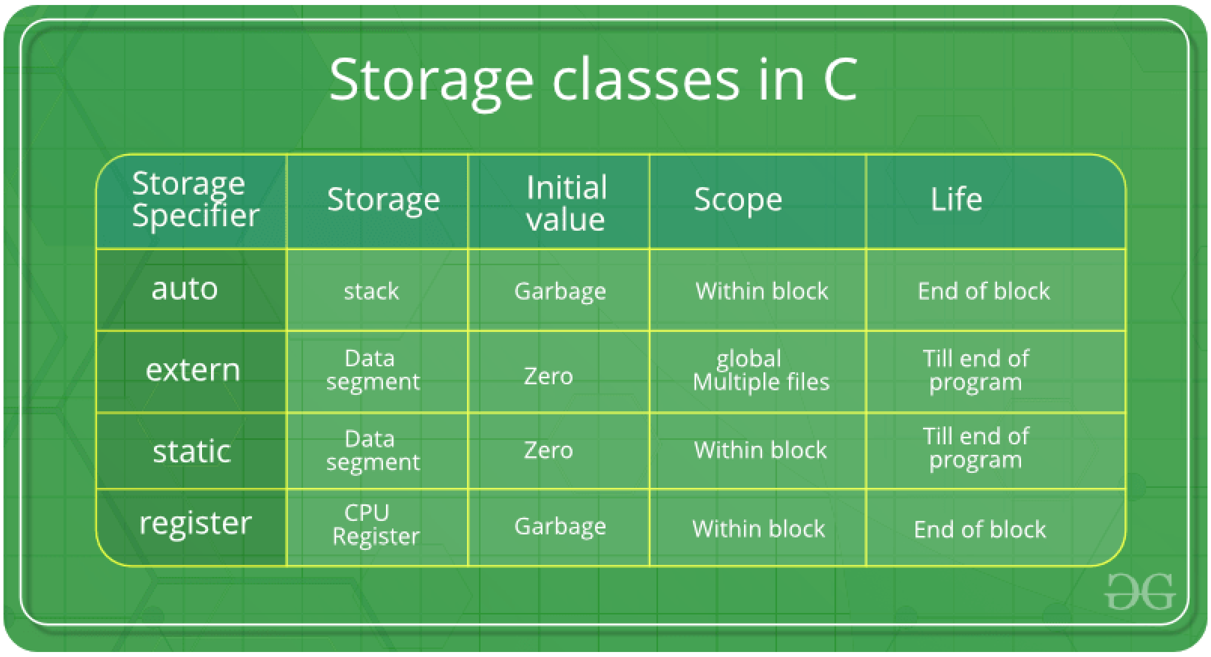
Output:

```
1 2 3 4
```

➢ Storage Classes in C

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program.

C language uses 4 storage classes, namely:



Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

- The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
   int mount;
   auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

- **The register Storage Class**

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
   register int  miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

- ## The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>

/* function declaration */
void func(void);

static int count = 5; /* global variable */

main() {

   while(count--) {
      func();
   }

   return 0;
}

/* function definition */
```

```
void func( void ) {

   static int i = 5; /* local static variable */
   i++;

   printf("i is %d and count is %d\n", i, count);
}
```

When the above code is compiled and executed, it produces the following result −

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

- ## The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>

int count ;
extern void write_extern();

main() {
   count = 5;
   write_extern();
}
```

Second File: support.c

```
#include <stdio.h>

extern int count;

void write_extern(void) {
   printf("count is %d\n", count);
}
```

Here, *extern* is being used to declare *count* in the second file, where as it has its definition in the first file, main.c. Now, compile these two files as follows −

```
$gcc main.c support.c
```

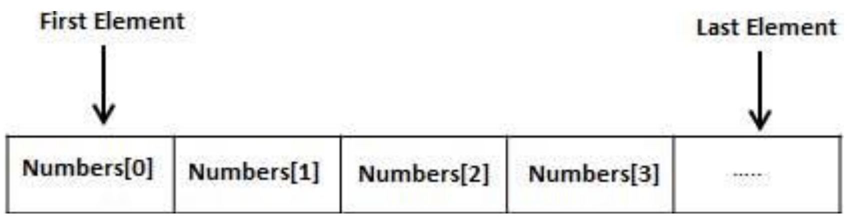
It will produce the executable program **a.out**. When this program is executed, it produces the following result −

```
count is 5
```

➢C – Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows −

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement −

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows −

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0,
50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

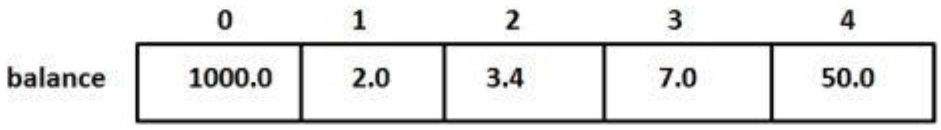
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write −

```
double balance[] = {1000.0, 2.0, 3.4, 7.0,
50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array −

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above −



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example −

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example

Shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

Live Demo

```
#include <stdio.h>

int main () {

   int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ ) {
      n[ i ] = i + 100; /* set element at location i to i + 100
*/
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ ) {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Types of array

1.single dimensional array

2.multi dimensional array

1.single dimensional:

A dimensional is used representing the elements of the array for example

int a[5]

The [] is used for dimensional or the sub-script of the array that is generally used for declaring the elements of the array For

Accessing the Element from the array we can use the Subscript of the Array like this

a[3]=100

This will set the value of 4th element of array

So there is only the single bracket then it called the Single Dimensional Array

This is also called as the Single Dimensional Array.

2.multi dimensional:

The Multidimensional Array are used for Representing the Total Number of Tables of Matrix A Three dimensional Array is used when we wants to make the two or more tables of the Matrix Elements for Declaring the Array Elements we can use the way like this

int a[3][3][3]

In this first 3 represents the total number of Tables and the second 3 represents the total number of rows in the each table and the third 3 represents the total number of Columns in the Tables;

So this makes the 3 Tables having the three rows and the three columns

The Main and very important thing about the array that the elements are stored always in the Contiguous in the memory of the Computer

C Program to Search an element in Array

```
#include<stdio.h>

int main() {
  int a[30], ele, num, i;

  printf("\nEnter no of elements :");
  scanf("%d", &num);

  printf("\nEnter the values :");
  for (i = 0; i < num; i++) {
    scanf("%d", &a[i]);
  }

  //Read the element to be searched
  printf("\nEnter the elements to be searched :");
  scanf("%d", &ele);

  //Search starts from the zeroth location
  i = 0;
  while (i < num && ele != a[i]) {
    i++;
  }

  //If i < num then Match found
  if (i < num) {
    printf("Number found at the location = %d", i + 1);
  } else {
    printf("Number not found");
  }

  return (0);
}
```

Output :

```
Enter no of elements : 5
11 22 33 44 55
Enter the elements to be searched : 44
```

4 Number found at the location = 4

Pattern program in C

```
#include <stdio.h>
int main()
{
  int row, c, n;

  printf("Enter the number of rows in pyramid of
stars to print\n");
  scanf("%d", &n);

  for (row = 1; row <= n; row++)  // Loop to print
rows
  {
    for (c = 1; c <= n-row; c++)  // Loop to print
spaces in a row
      printf(" ");

    for (c = 1; c <= 2*row - 1; c++) // Loop to print
stars in a row
      printf("*");

    printf("\n");
  }

  return 0;
}
```

output:

```
   *
   ***
  *****
 *******
*********
```

Program to Check Vowel or consonant

```c
#include <stdio.h>
int main() {
    char c;
    int lowercase, uppercase;
    printf("Enter an alphabet: ");
    scanf("%c", &c);

    // evaluates to 1 if variable c is lowercase
    lowercase = (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');

    // evaluates to 1 if variable c is uppercase
    uppercase = (c == 'A' || c == 'E' || c == 'I' || c == 'O' || c == 'U');

    // evaluates to 1 if c is either lowercase or uppercase
    if (lowercase || uppercase)
        printf("%c is a vowel.", c);
    else
        printf("%c is a consonant.", c);
    return 0;
}

```

Output

```
Enter an alphabet: G
G is a consonant.
```

Program to Check Palindrome

```c
#include <stdio.h>
int main() {
    int n, reversedN = 0, remainder, originalN;
    printf("Enter an integer: ");
    scanf("%d", &n);
    originalN = n;

    // reversed integer is stored in reversedN
    while (n != 0) {
            remainder = n % 10;
            reversedN = reversedN * 10 + remainder;
            n /= 10;
        }

        // palindrome if orignalN and reversedN are equal
        if (originalN == reversedN)
            printf("%d is a palindrome.", originalN);
        else
            printf("%d is not a palindrome.", originalN);

        return 0;
    }

```

Output

```
Enter an integer: 1001
1001 is a palindrome.
```

➢ C Functions

In this tutorial, you will be introduced to functions (both user-defined and standard library functions) in C programming. Also, you will learn why functions are used in programming.

A function is a block of code that performs a specific task.Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

- create a circle function
- create a color function

Dividing a complex problem into smaller chunks makes our program easy to understand and reuse.

Types of function

There are two types of function in C programming:

- Standard library functions
- User-defined functions

- **Standard library functions**

The standard library functions are built-in functions in C programming.

These functions are defined in header files. For example,

- The `printf()` is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the `stdio.h` header file. Hence, to use the `printf()` function, we need to include the `stdio.h` header file using `#include <stdio.h>`.
- The `sqrt()` function calculates the square root of a number. The function is defined in the `math.h` header file.
  Visit standard library functions in C programming to learn more.

- **User-defined function**

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

How user-defined function works?

```
#include <stdio.h>

void functionName()

{

    ... .. ...

    ... .. ...
}



int main()

{

    ... .. ...

    ... .. ...



    functionName();



    ... .. ...

    ... .. ...

}
```

The execution of a C program begins from the `main()` function.
When the compiler encounters `functionName();`, control of the program jumps to

```
 void functionName()
```

And, the compiler starts executing the codes inside `functionName()`.

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a function −

Sr.No.	Call Type & Description
1	Call by value This method copies the actual value of an argument into the forma parameter of the function. In this case, changes made to th parameter inside the function have no effect on the argument.
2	Call by reference This method copies the address of an argument into the forma parameter. Inside the function, the address is used to access th actual argument used in the call. This means that changes mad to the parameter affect the argument.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

Live Demo

```
#include <stdio.h>

int main () {

  /* local variable declaration */
  int a, b;
  int c;

  /* actual initialization */
  a = 10;
  b = 20;
  c = a + b;

  printf ("value of a = %d, b = %d and c = %d\n", a,
b, c);

  return 0;
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.A global variable can be accessed by any function. The following program show how global variables are used in a program.

Live Demo

```
#include <stdio.h>

/* global variable declaration */
int g;

int main () {

  /* local variable declaration */
  int a, b;

  /* actual initialization */
  a = 10;
  b = 20;
  g = a + b;

  printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

  return 0;
}
```

Program to find cube using function

```
/**
 * C program to find cube of any number using function
 */
#include <stdio.h>

/* Function declaration */
double cube(double num);

int main()
{
    int num;
    double c;

    /* Input number to find cube from user */
    printf("Enter any number: ");
    scanf("%d", &num);
    c = cube(num);

    printf("Cube of %d is %.2f", num, c);

    return 0;
}

/**
 * Function to find cube of any number
 */
double cube(double num)
{
    return (num * num * num);
}
```

Output:

```
Enter any number: 5
Cube of 5 is 125.00
```

Program to find all prime numbers in given range using function

```
/**
 * C program to list all prime number between an interval using function.
 */

#include <stdio.h>


/* Function declarations */
int isPrime(int num);
void printPrimes(int lowerLimit, int upperLimit);



int main()
{
    int lowerLimit, upperLimit;

    printf("Enter the lower and upper limit to list primes: ");
    scanf("%d%d", &lowerLimit, &upperLimit);
```

```c
    // Call function to print all primes between the given range.
    printPrimes(lowerLimit, upperLimit);

    return 0;
}



/**
 * Print all prime numbers between lower limit and upper limit.
 */
void printPrimes(int lowerLimit, int upperLimit)
{
    printf("All prime number between %d to %d are: ", lowerLimit, upperLimit);

    while(lowerLimit <= upperLimit)
    {
        // Print if current number is prime.
        if(isPrime(lowerLimit))
        {
            printf("%d, ", lowerLimit);
        }

        lowerLimit++;
    }
}



/**
 * Check whether a number is prime or not.
 * Returns 1 if the number is prime otherwise 0.
 */
int isPrime(int num)
{
    int i;

    for(i=2; i<=num/2; i++)
    {
        /*
         * If the number is divisible by any number
         * other than 1 and self then it is not prime
         */
        if(num % i == 0)
        {
            return 0;
        }
    }

    return 1;
}
```
Output
```
Enter the lower and upper limit to list primes: 10 50
All prime number between 10 to 50 are: 11, 13, 17, 19, 23, 29, 31, 37,
41, 43, 47,
```

➢C – Pointers

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let’s start learning them in simple and easy steps.

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is −

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
```

```
float   *fp;     /* pointer to a float */
char    *ch      /* pointer to a character
```

Advantages :

- Pointers reduce the length and complexity of a program.
- They increase execution speed.
- A **pointer** enables us to access a variable that is defined outside the function.
- Pointers are more efficient in handling the data tables.

KEY POINTS TO REMEMBER ABOUT POINTERS IN C:

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for 16 bit compiler).

Example using pointer

```
#include <stdio.h>

int main () {

   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration
*/

   ip = &var;  /* store address of var in pointer
variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
  printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

C - Pointer arithmetic

A pointer in c is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and –

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

Live Demo

```
#include <stdio.h>

const int MAX = 3;
```

```
int main () {

   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = var;

   for ( i = 0; i < MAX; i++) {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* move to the next location */
      ptr++;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

Live Demo

```
#include <stdio.h>

const int MAX = 3;

int main () {

   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = &var[MAX-1];

   for ( i = MAX; i > 0; i--) {
      printf("Address of var[%d] = %x\n", i-1, ptr );
      printf("Value of var[%d] = %d\n", i-1, *ptr );

      /* move to the previous location */
      ptr--;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Address of var[2] = bfedbcd8
Value of var[2] = 200
Address of var[1] = bfedbcd4
Value of var[1] = 100
Address of var[0] = bfedbcd0
Value of var[0] = 10
```

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

```
new_address= current_address + (number * size_of(data type))
```

32-bit

For 32-bit int variable, it will add 2 * number.

64-bit

For 64-bit int variable, it will add 4 * number.

Let's see the example of adding value to pointer variable on 64-bit architecture.

```
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
```

```c
printf("Address of p variable is %u \n",p);
p=p+3;   //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}
```

Output

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., 4*3=12 increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., 2*3=6. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

```
new_address= current_address - (number * size_of(data type))
```

32-bit:For 32-bit int variable, it will subtract 2 * number.

64-bit:for 64-bit int variable, it will subtract 4 * number.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

```c
#include<stdio.h>
int main(){
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-3; //subtracting 3 from pointer variable
printf("After subtracting 3: Address of p variable is %u \n",p);

return 0;
    }
```

Output

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is
3214864288
```

Printing each character address of a string using pointer variable

Example

```c
/*
 * Program  : printing address of each char in the st
ring
 * using pointer variable
 * Language : C
 */

#include<stdio.h>

int main()
{
    char str[6] = "Hello";
    char *ptr;
    int i;

    //string name itself a base address of the string
    ptr = str; //ptr references str

    for(i = 0; ptr[i] != '\0'; i++)
        printf("&str[%d] = %p\n",i,ptr+i);


    return 0;
}
```

Output:

```
&str[0] = 0x7ffc86566b26

&str[1] = 0x7ffc86566b27

&str[2] = 0x7ffc86566b28

&str[3] = 0x7ffc86566b29

&str[4] = 0x7ffc86566b2a
```

C - Structures

A structure is a user defined data type in C/C++. A structure creates a data type that can be used to group items of possibly different types into a single type.

We use **struct** keyword to create a **structure in C**. The struct keyword is a short form of **structured data type.** Here struct_name can be anything of your choice. Members data type can be same or different. Once we have declared the structure we can use the struct name as a data type like int, float etc.

```
struct struct_name {
   DataType member1_name;
   DataType member2_name;
   DataType member3_name;
   …
};
```

declare variable of a structure?

```
struct  struct_name  var_name;
```

or

```
struct struct_name {
   DataType member1_name;
   DataType member2_name;
   DataType member3_name;
   …
} var_name;
```

Structure members are accessed using dot (.) operator.

```
#include<stdio.h>

struct Point
{
   int x, y;
};

int main()
{
   struct Point p1 = {0, 1};

   // Accessing members of point p1
   p1.x = 20;
   printf ("x = %d, y = %d", p1.x, p1.y);

   return 0;
}
```

Example of Structure in C

```
#include <stdio.h>
/* Created a structure here. The name of the
structure is
 * StudentData.
 */
struct StudentData{
    char *stu_name;
    int stu_id;
```

```c
    int stu_age;
};
int main()
{
     /* student is the variable of structure
StudentData*/
     struct StudentData student;

     /*Assigning the values of each struct member
here*/
     student.stu_name = "Steve";
     student.stu_id = 1234;
     student.stu_age = 30;

     /* Displaying the values of struct members */
     printf("Student Name is: %s",
student.stu_name);
     printf("\nStudent Id is: %d", student.stu_id);
     printf("\nStudent Age is: %d",
student.stu_age);
     return 0;
}
```

Output:

```
Student Name is: Steve
Student Id is: 1234
Student Age is: 30
```

➢File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file. A list of file functions are given below:

next →← prev

File Handling in C

In programming, we may require some specific input data to be generated several numbers of times
the data on the console. The data to be displayed may be very large, and only a limited amount of d
since the memory is volatile, it is impossible to recover the programmatically generated data again a
may store it onto the local file system which is volatile and can be accessed every time. Here, comes

File handling in C enables us to create, update, read, and delete the files stored on the local file syst
operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file
- Writing to the file
- Deleting the file

Functions for file handling

There are many functions in the C library to open, read, write,
list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Modes in the fopen() function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode
wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

C Program to Find the Number of Lines in a Text File

```
/*
 * C Program to Find the Number of Lines in a Text File
 */
#include <stdio.h>

int main()
{
    FILE *fileptr;
    int count_lines = 0;
        char filechar[40], chr;

        printf("Enter file name: ");
        scanf("%s", filechar);
        fileptr = fopen(filechar, "r");
       //extract character from file and store in chr
        chr = getc(fileptr);
        while (chr != EOF)
        {
            //Count whenever new line is encountered
            if (chr == 'n')
            {
                count_lines = count_lines + 1;
            }
            //take next character from file.
            chr = getc(fileptr);
        }
        fclose(fileptr); //close file.
        printf("There are %d lines in %s  in a file\n", count_lines, filechar);
        return 0;
    }
```

```
$ cc pgm49.c
$ a.out
Enter file name: pgm2.c
There are 43 lines in pgm2.c  in a file
```

➢Preprocessor directives

Preprocessors are a way of making text processing with your C program before they are actually compiled. Before the actual compilation of every C program it is passed through a Preprocessor. The Preprocessor looks through the program trying to find out specific instructions called Preprocessor directives that it can understand. All Preprocessor directives begin with the # (hash) symbol. C++ compilers use the same C preprocessor.[1]

There are 4 main types of preprocessor directives:

- **Macros.**

- File Inclusion.
- Conditional Compilation.
- Other directives.

1.Macros

A **macro** is a fragment of code which has been given a name. ... Object-like **macros** resemble data objects when used, function-like **macros** resemble function calls. You may define any valid identifier as a **macro**, even if it is a **C** keyword.

C predefined macros example

File: simple.c

```c
#include<stdio.h>
 int main(){
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("STDC :%d\n", __STDC__ );
   return 0;
 }
```

Output:

```
File :simple.c

Date :Dec 6 2015
Time :12:28:46
Line :6
STDC :1
```

File Inclusive Directives

1. File inclusive Directories are **used to include user define header file** inside C Program.
2. File inclusive directory checks included header file inside same directory (if path is not mentioned).

3. File inclusive directives begins with **#include**
4. If Path is mentioned then it will include that **<u>header file into current scope</u>**.

Including Standard Header Files

```
#include<filename>
```

Conditional Compilation

The third useful facility provided by the preprocessor is **conditional compilation**; i.e. the selection of lines of source code to be compiled and those to be ignored. While conditional compilation can be used for many purposes, we will illustrate its use with debug statements. In our previous programming examples, we have discussed the usefulness of printf() statements inserted in the code for the purpose of displaying debug information during program testing. Once the program is debugged and accepted as ``working'', it is desirable to remove these debug statements to use the program.

The syntax is:

- #<if-part>
- <statements>
- [# <elseif-part>
- <statements>]
- [#<else-part>
- <statements>]
- #<endif-part>

4.other directive:Apart from the above directives there are two more direrctive which are most commonly used .These are:

1.#undef directive: the #undef directive is used to undefine an extra macro.

2.#undefLIMIT: Using this statement will undefine the existing macro LIMIT.