#include <stdio.h> includes the standard input output library functions. The printf() function is defined in stdio.h .

int main() The main() function is the entry point of every program in c language.

printf() The printf() function is used to print data on the console.

return 0 The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

How to compile and run the c program

By menu

Now click on the compile menu then compile sub menu to compile the c program.

Then click on the run menu then run sub menu to run the c program.

By shortcut

Or, press ctrl+f9 keys compile and run the program directly.

You will see the following output on user screen.

c program output

You can view the user screen any time by pressing the alt+f5 keys.

Now press Esc to return to the turbo c++ console.

What is a compilation?

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

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

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

```
Output
Enter two integer numbers: 56
11
Enter two float numbers:
78.98
56.45
Enter two double data type numbers:
47789.7149
1234.7987
The subtraction of two integer numbers: 45
The subtraction of two float numbers: 22.530003
The subtraction of two double numbers: 46554.916200
```

Multiplication Operator

The multiplication operator is represented as an asterisk (*) symbol, and it is used to return the product of n1 and n2 numbers. The data type of the given number can be different types such as int, float, and double in the C programing language.

Syntax

```
C = A * B;
```

For example, there are two operands 15 and 6, and we want to get their product. So, we can use the '*' Operator between the given numbers that return int data 90.

```
/* program to multiply two numbers using astrisk (*) operator.
*/
#include <stdio.h>
#include <conio.h>
int main ()
{
// declare integer variables
int num1, num2, res;
```

```
// declare float data type number
float f1, f2, res1;
// declare double variables
double d1, d2, res2;
printf (" Enter two integer numbers: ");
scanf ("%d %d", &num1, &num2);
res = num1 * num2; // use * operator

printf (" Enter two float numbers: \n ");
scanf ("%f %f", &f1, &f2);
res1 = f1 * f2; // use * operator

printf (" Enter two double data type numbers: \n ");
scanf ("%lf %lf", &d1, &d2);
res2 = d1 * d2; // use * operator
printf (" The multiplication of two integer numbers: %d \n",
res);
printf (" The multiplication of two float numbers: %f \n ",
res1);
printf (" The multiplication of two double numbers: %lf",
res2);
return 0;
}
Output
Enter two integer numbers: 15
12
Enter two float numbers:
2.5
3.5
Enter two double data type numbers:
234.324
798.124
The multiplication of two integer numbers: 180
The multiplication of two float numbers: 8.750000
```

```
      int a = 5;
      int b = 10;

      // Use Not Equal To (!=) Operator
printf (" a != b : %d", (a != b));
      if (a != b)
printf ("\n %d is equal to %d", a, b);
    else
printf (" \n %d is not equal to %d", a, b);



      int x = 5;
      int y = 5;

      // Use Not Equal To (!=) Operator
printf (" \n x != y : %d", (x != y));
    if (a != b)
printf (" \n %d is equal to %d", x, y);
    else
printf ("\n %d is not equal to %d", x, y);
    return 0;
}
```

Output

```
a != b : 1
 5 is equal to 10
x != y : 0
 5 is equal to 5
```

Less than Operator (<)

It is used to check whether the value of the left operand is less than the right operand, and if the statement is true, the operator is known as the Less than Operator.

Modulus and Assign Operator (%=):

An operator used between the left operand and the right operand divides the first number (n1) by the second number (n2) and returns the remainder in the left operand.

Syntax

```
A %= B;
```

Or

```
A = A % B;
```

Let's create a program to use the divide and assign operator (%=) in C.

```
#include <stdio.h>
#include <conio.h>
int main ()
{
     // initialize variables
     int n1, n2, c;

printf (" Enter the value of n1: ");
scanf ("%d", &n1);

printf (" \n Enter the value of n2: ");
scanf ("%d", &n2);
     n1 %= n2; // Use modulus and Equal operator (a = a % b)
printf (" \n The modulus value of n1: %d", n1);

    return 0;
}
```

Output

```
Enter the value of n1: 23
Enter the value of n2: 5
The modulus value of n2: 3
```

Example 2: Let's create another program to use the pre-increment operator in mathematical expression.

Program2.c

```
#include <stdio.h>
#include <conio.h>
int main ()
{
     // declare integer variables
     int a, b, c, d, x;

     // initialization of the variables
     a = 5;
     b = 7;
     c = 12;
     d = 15;

     // use pre-increment operator in the mathematical expression
     x = ++a + ++b + ++c + ++d;
printf( " The value of x is: %d ", x);

     // print the updated value of a, b, c, and d
printf (" \n The updated value of a = %d, b = %d, c = %d and d = %d ", a, b, c, d);
    return 0;
}
```

Output

```
The value of x is: 43
The updated value of a = 6, b = 8, c = 13 and d = 16
```

Post-increment Operator

Post-increment is an increment operator, represented as the double plus (a++) symbol followed by an operator 'a'. It increments the value of the operand by 1 after using it in the mathematical expression. In other words, the variable's original value is used in the expression first, and then the post-increment operator updates the operand value by 1.

Syntax

```
x = a++;
```

In the above syntax, the operand 'a' value is assigned to the variable x, and then the post increment operator increases or updates the value of 'a' by 1.

Example 1: Let's create a simple program to use the post-increment operator in C programming language.

Program1.c

```
#include <stdio.h>
#include <conio.h>
int main ()
{
// declaration of the variables
int a = 7;
int b = 0;

// print the value of the increment operator
printf (" Before using the post-increment operator " );
printf (" \n The value of a is %d ", a);
printf (" \n The value of b is %d ", b);
// use post increment operator
b = a++;
printf (" \n\n After using the post-increment operator ");
printf (" \n The value of a is %d ", a);
```

```
printf (" \n The updated value of a = %d, b = %d, c = %d and d
= %d ", a, b, c, d);
    return 0;

}
```

Output

```
The value of x is: 39
 The updated value of a = 6, b = 8, c = 13 and d = 16
```

Program to use the Pre-increment and Post-increment Operator

Let's create a simple program to use the pre-increment and post-increment operator in the C programming language.

```
#include <stdio.h>
#include <conio.h>
int main ()
{
     int x, y, z, exp;

printf (" Enter the value of x: ");
scanf (" %d", &x);
printf (" \n Enter the value of y: ");
scanf (" %d", &y);
printf (" \n Enter the value of z: ");
scanf (" %d", &z);
printf (" \n Before using the increment operator: ");
printf (" \n The original value of x: %d", x);
printf (" \n The original value of x: %d", y);
printf (" \n The original value of x: %d", z);

     // use pre-increment and post-increment operator
     exp = x++ + ++x + ++y + y++ + ++z;
printf (" \n\n After using the increment operator: ");
printf (" \n The result of the expression is: %d", exp);
printf (" \n The updated value of x: %d", x);
```

Unary Operator in C

In this section, we will discuss the unary operator in the C programming language. Operators are the special symbols used to perform mathematical and logical operations to the given numbers or operands and returns results based on passed operator between the operands.

A unary operator is an operator used to operate on a single operand to return a new value. In other words, it is an operator that updates the value of an operand or expression's value by using the appropriate unary operators. In Unary Operator, operators have equal priority from right to left side associativity.

Unary Operator in C

Types of the Unary Operator

Following are the types of the unary operators in the C programming language.

Unary Minus (-)
Unary Plus (+)
Increment (++)
Decrement (--)
Logical Negation (!)
Address Operator (&)
Sizeof() operator

Unary Minus (-)

The Unary Minus operator is represented using the symbol (-). The unary operator is used to change the sign of any positive value to a negative value. It means it changes the positive number to the negative, and a negative number becomes the positive number using the unary minus operator.

The value of b is 19.

Unary Sizeof() Operator

The sizeof is a keyword used to find the size of different data types or operands like int, float, char, double, etc.

Syntax

```
sizeof(data_variable);
```

```
#include <stdio.h>
#include <conio.h>
int main ()
{
// declaration of different types of data variables
int x;
float y;
char ch;
double z;

// use sizeof() operator and pass the different data type variable to get the result
printf (" The size of the int (x) variable is: %d", sizeof(x));
printf (" \n The size of the float (y) variable is: %d", sizeof(y));
printf (" \n The size of the char (ch) variable is: %d", sizeof(ch));
printf (" \n The size of the double (z) variable is: %d", sizeof(z));
return 0;
}
```

Logical operators

An operator can be defined as a symbol that is used for performing different operations. In a programming language, there are various types of operators such as arithmetic operators, relational operators, logical operators, assignment operator, increment/decrement operators, conditional operators, bitwise operators, and shift operators.

In this article, we are discussing the logical operators and their types, along with an example of each. Here, we will discuss all types of logical operators irrespective of a particular programming language. Some programming languages support limited operators, so some of the logical operators that we are discussing may or may not be supported by the programming language you are using.

Logical operators are generally used for combining two or more relational statements. They return Boolean values. The logical operators are used primarily in the expression evaluation to make a decision. These operators allow the evaluation and manipulation of specific bits within the integer.

The types of Logical operators with their description are tabulated as follows -

Operators	Description
&& (Logical AND)	This operator returns true if all relational statements combined with && are true, else it returns false.
|| (Logical OR)	This operator returns true if at least one of the relational statements combined with || is true, else it returns false.
! (logical NOT)	It returns the inverse of the statement's result.

```
to be commented
*/
Let's see an example of a multi-Line comment in C.

#include<stdio.h>
int main(){
    /*printing information
      Multi-Line Comment*/
printf("Hello C");
return 0;
}
Output:

Hello C
```

C Format Specifier

The Format specifier is a string used in the formatted input and output functions. The format string determines the format of the input and output. The format string always starts with a '%' character.

The commonly used format specifiers in printf() function are:

Format specifier	Description
%d or %i	It is used to print the signed integer value where signed integer means that the variable can hold both positive and negative values.
%u	It is used to print the unsigned integer value where the unsigned integer means that the variable can hold only positive value.
%o	It is used to print the octal unsigned integer where octal integer value always starts with a 0 value.
%x	It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always starts with a 0x value. In this, alphabetical characters are printed in small letters such as a, b, c, etc.
%X	It is used to print the hexadecimal unsigned integer, but %X prints the alphabetical characters in uppercase such as A, B, C, etc.
%f	It is used for printing the decimal floating-point values. By default, it prints the 6 values after '.'.

Specifying Precision

We can specify the precision by using '.' (Dot) operator which is followed by integer and format specifier.

```
1. int main()
2. {
3.  float x=12.2;
4.   printf("%.2f", x);
5.   return 0;
6. }
```

Output

C Boolean

In C, Boolean is a data type that contains two types of values, i.e., 0 and 1. Basically, the bool type value represents two types of behavior, either true or false. Here, '0' represents false value, while '1' represents true value.

In C Boolean, '0' is stored as 0, and another integer is stored as 1. We do not require to use any header file to use the Boolean data type in C++, but in C, we have to use the header file, i.e., stdbool.h. If we do not use the header file, then the program will not compile.

Syntax

```
bool variable_name;
```

In the above syntax, bool is the data type of the variable, and variable_name is the name of the variable.

Let's understand through an example.

Play Video


```
#include <stdio.h>
#include<stdbool.h>
int main()
{
bool x=false; // variable initialization.
if(x==true) // conditional statements
{
printf("The value of x is true");
}
else
printf("The value of x is FALSE");
return 0;
}
```

```
printf("%s", binary);
printf("\nThe ones complement of the binary number is :");

   // Finding onescomplement in C
for(int i=0;i<n;i++)
   {
       if(binary[i]=='0')
onescomplement[i]='1';
       else if(binary[i]=='1')
onescomplement[i]='0';
   }
onescomplement[n]='\0';
printf("%s",onescomplement);


printf("\nThe twos complement of a binary number is :");

// Finding twoscomplement in C
for(int i=n-1; i>=0; i--)
    {
       if(onescomplement[i] == '1' && carry == 1)
       {
twoscomplement[i] = '0';
       }
       else if(onescomplement[i] == '0' && carry == 1)
       {
twoscomplement[i] = '1';
           carry = 0;
       }
       else
       {
twoscomplement[i] = onescomplement[i];
       }
    }
```

```
number is not equal to 10, 50 or 100

Output
enter a number:50
number is equal to 50
number is equal to 100
number is not equal to 10, 50 or 100
```

Nested switch case statement

We can use as many switch statement as we want inside a switch statement. Such type of statements is called nested switch case statements. Consider the following example.

```
1. #include <stdio.h>
2. int main () {
3.
4.    int i = 10;
5.    int j = 20;
6.
7.    switch(i) {
8.
9.       case 10:
10.             printf("the value of i evaluated in outer
   switch: %d\n",i);
11.       case 20:
12.         switch(j) {
13.             case 20:
14.                     printf("The value of j evaluated in
   nested switch: %d\n",j);
15.         }
16.    }
17.
18.    printf("Exact value of i is : %d\n", i );
19.    printf("Exact value of j is : %d\n", j );
20.
```

9

10

Program to print table for the given number using while loop in C

```
1. #include<stdio.h>
2. int main(){
3. int i=1,number=0,b=9;
4. printf("Enter a number: ");
5. scanf("%d",&number);
6. while(i<=10){
7. printf("%d \n",(number*i));
8. i++;
9. }
10.  return 0;
11.  }
```

Output

```
Enter a number: 50
50
100
150
200
250
300
350
400
450
500

Enter a number: 100
100
200
300
400
500
```

For loop

Let's see the infinite 'for' loop. The following is the definition for the infinite for loop:

```
1. for(; ;)
2. {
3.     // body of the for loop.
4. }
```

As we know that all the parts of the 'for' loop are optional, and in the above for loop, we have not mentioned any condition; so, this loop will execute infinite times.

Let's understand through an example.

```
1. #include <stdio.h>
2. int main()
3. {
4.    for(;;)
5.    {
6.      printf("Hello java");
7.    }
8. return 0;
9. }
```

In the above code, we run the 'for' loop infinite times, so "Hello java" will be displayed infinitely.

```
    return 0;
}
```

Output

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

As you can see, 2 2 is not printed on the console because inner loop is continued at i==2 and j==2.

C goto statement

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statment can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complecated.

Syntax:

```
1. label:
2. //some part of the code;
3. goto label;
```

goto example

Let's see a simple example to use goto statement in C language.

```
1. #include <stdio.h>
2. int main()
3. {
4.   int num,i=1;
5.   printf("Enter the number whose table you want to
     print?");
6.   scanf("%d",&num);
7.   table:
8.   printf("%d x %d = %d\n",num,i,num*i);
9.   i++;
10.    if(i<=10)
11.     goto table;
12.  }
```

Output:

```
Enter the number whose table you want to print?10
10 x 1 = 10
10 x 2 = 20
10 x 3 = 30
```

```
0 0 1
0 0 2
0 1 0
0 1 1
0 1 2
0 2 0
0 2 1
0 2 2
0 3 0
came out of the loop
```

Consider the following example for the call by reference.

```
1. #include<stdio.h>
2. void change(int *num) {
3.       printf("Before adding value inside function num=%d
   \n",*num);
4.     (*num) += 100;
5.       printf("After adding value inside function num=%d
   \n", *num);
6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.       change(&x);//passing reference in function
11.       printf("After function call x=%d \n", x);
12.  return 0;
13.  }
```

Output

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```


Call by reference Example: Swapping the values of the two variables

```
1. #include <stdio.h>
2. void swap(int *, int *); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.      printf("Before swapping the values in main a = %d, b
   = %d\n",a,b); // printing the value of a and b in main
8.     swap(&a,&b);
```

```
9.      printf("After swapping values in main a = %d, b =
   %d\n",a,b); // The values of actual parameters do change
   in call by reference, a = 10, b = 20
10.  }
11.  void swap (int *a, int *b)
12.  {
13.      int temp;
14.      temp = *a;
15.      *a=*b;
16.      *b=temp;
17.      printf("After swapping values in function a = %d, b
   = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
18.  }
```

Output

```
Before swapping the values in main a = 10, b = 20
After swapping values in function a = 20, b = 10
After swapping values in main a = 20, b = 10
```

Recursion in C

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

```
1. #include <stdio.h>
2. int fact (int);
3. int main()
4. {
5.     int n,f;
6.     printf("Enter the number whose factorial you want to
   calculate?");
7.     scanf("%d",&n);
8.     f = fact(n);
9.     printf("factorial = %d",f);
10.  }
11.  int fact(int n)
12.  {
```

Storage Classes in C

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- Automatic
- External
- Static
- Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.

```
12.                     temp = a[i];
13.                     a[i] = a[j];
14.                     a[j] = temp;
15.                 }
16.             }
17.         }
18.         printf("Printing Sorted Element List ...\n");
19.         for(i = 0; i<10; i++)
20.         {
21.             printf("%d\n",a[i]);
22.         }
23.   }
```

```
17.             for (j=0;j<3;j++)
18.             {
19.                 printf("%d\t",arr[i][j]);
20.             }
21.         }
22.   }
```

```
Output
Enter a[0][0]: 56
Enter a[0][1]: 10
Enter a[0][2]: 30
Enter a[1][0]: 34
Enter a[1][1]: 21
Enter a[1][2]: 34
Enter a[2][0]: 45
Enter a[2][1]: 56
Enter a[2][2]: 78

 printing the elements ....

56      10      30
34      21      34
45      56      78
```

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %x \n",p); // p contains
   the address of the number therefore printing p gives the
   address of number.
7. printf("Value of p variable is %d \n",*p); // As we know
   that * is used to dereference a pointer therefore if we
   print *p, we will get the value stored at the address
   contained by p.
8. return 0;
9. }
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```


Pointer to array

```
1. int arr[10];
2. int  *p[10]=&arr;  // Variable  p  of  type  pointer  is
   pointing to the address of an integer array arr.
```

Pointer to a function

```
1. void show (int);
2. void(*p)(int) = &display; // Pointer p is pointing to the
   address of a function
```

Advantage of pointer

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2) We can return multiple values from a function using the pointer.

```
1. #include<stdio.h>
2. int main(){
3. int a=10,b=20,*p1=&a,*p2=&b;
4.
5. printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
6. *p1=*p1+*p2;
7. *p2=*p1-*p2;
8. *p1=*p1-*p2;
9. printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);
10.
11.  return 0;
12.  }
```

Output

```
Before swap: *p1=10 *p2=20
After swap: *p1=20 *p2=10
```

```
10.  printf("Value of *p variable is %d \n",*p);
11.  printf("Address of p2 variable is %x \n",p2);
12.  printf("Value of **p2 variable is %d \n",*p);
13.  return 0;
14.  }
```

```
Output
Address of number variable is fff4
Address of p variable is fff4
Value of *p variable is 50
Address of p2 variable is fff2
Value of **p variable is 50
```

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+1;
8. printf("After increment: Address of p variable is %u
   \n",p); // in our case, p will get incremented by 4
   bytes.
9. return 0;
10.  }
```

Output

```
Address of p variable is 3214864300
After increment: Address of p variable is 3214864304
```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

```
1. new_address= current_address - i * size_of(data type)
```

32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

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

```
6. printf("Address of p variable is %u \n",p);
7. p=p-1;
8. printf("After decrement: Address of p variable is %u
   \n",p); // P will now point to the immidiate previous
   location.
9. }
```

Output

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

C Pointer Addition

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

```
1. 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.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+3;   //adding 3 to pointer variable
8. printf("After adding 3: Address of p variable is %u
   \n",p);
9. return 0;
10.  }
```

```
9. return 0;
10.  }
```

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

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.
However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

void pointer in C

Till now, we have studied that the address assigned to a pointer should be of the same type as specified in the pointer declaration. For example, if we declare the int pointer, then this int pointer cannot point to the float variable or some other type of variable, i.e., it can point to only int type variable. To overcome this problem, we use a pointer to void. A pointer to void means a generic pointer that can point to any data type. We can assign the address of any data type to the void pointer, and a void pointer can be assigned to any type of the pointer without performing any explicit typecasting.

Syntax of void pointer

```
1. void *pointer name;
```

Size of the void pointer in C

The size of the void pointer in C is the same as the size of the pointer of character type. According to C perception, the representation of a pointer to void is the same as the pointer

What is a Null Pointer?

A Null Pointer is a pointer that does not point to any memory location. It stores the base address of the segment. The null pointer basically stores the Null value while void is the type of the pointer.

A null pointer is a special reserved value which is defined in a stddef header file. Here, Null means that the pointer is referring to the 0^{th} memory location.

If we do not have any address which is to be assigned to the pointer, then it is known as a null pointer. When a NULL value is assigned to the pointer, then it is considered as a Null pointer.

Applications of Null Pointer

Following are the applications of a Null pointer:

- It is used to initialize a pointer variable when the pointer does not point to a valid memory address.
- It is used to perform error handling with pointers before dereferencing the pointers.
- It is passed as a function argument and to return from a function when we do not want to pass the actual memory address.

Let's look at the situations where we need to use the null pointer.

- When we do not assign any memory address to the pointer variable.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int *ptr;
5.     printf("Address: %d", ptr); // printing the value of
   ptr.
```

```
6.      printf("Value: %d", *ptr); // dereferencing the
   illegal pointer
7.    return 0;
8. }
```

In the above code, we declare the pointer variable *ptr, but it does not contain the address of any variable. The dereferencing of the uninitialized pointer variable will show the compile-time error as it does not point any variable. According to the stack memory concept, the local variables of a function are stored in the stack, and if the variable does not contain any value, then it shows the garbage value. The above program shows some unpredictable results and causes the program to crash. Therefore, we can say that keeping an uninitialized pointer in a program can cause serious harm to the computer.

How to avoid the above situation?

We can avoid the above situation by using the Null pointer. A null pointer is a pointer pointing to the 0^{th} memory location, which is a reserved memory and cannot be dereferenced.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int *ptr=NULL;
5.     if(ptr!=NULL)
6.     {
7.         printf("value of ptr is : %d",*ptr);
8.     }
9.     else
10.       {
11.           printf("Invalid pointer");
12.       }
13.    return 0;
14.  }
```

C String Uppercase: strupr()

The strupr(string) function returns string characters in uppercase. Let's see a simple example of strupr() function.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.   char str[20];
5.   printf("Enter string: ");
6.   gets(str);//reads string from console
7.   printf("String is: %s",str);
8.   printf("\nUpper String is: %s",strupr(str));
9.  return 0;
10.  }
```

Output:

```
Enter string: java
String is: java
Upper String is: JAVA
```

C String strstr()

The strstr() function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

Syntax:

```
1. char *strstr(const char *string, const char *match)
```

String strstr() parameters

string: It represents the full string from where substring will be searched.

match: It represents the substring to be searched in the full string.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.   char str[100]="this is java with c and java";
5.   char *sub;
```

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
1. struct employee
2. {    int id;
3.      char name[50];
4.      float salary;
5. };
```

Now write given code inside the main() function.

```
1. struct employee e1, e2;
```

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in C++ and Java.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

```
1. struct employee
2. {    int id;
3.      char name[50];
4.      float salary;
5. }e1,e2;
```


Which approach is good

If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

Accessing members of the structure

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Let's see the code to access the id member of p1 variable by. (member) operator.

```
1. p1.id
```

```
employee 1 id : 101
employee 1 name : Ram
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000
```

typedef in C

The typedef is a keyword used in C programming to provide some meaningful names to the already existing variable in the C program. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

Syntax of typedef

```
1. typedef <existing_name> <alias_name>
```

In the above syntax, 'existing_name' is the name of an already existing variable while 'alias name' is another name given to the existing variable.

For example, suppose we want to create a variable of type unsigned int, then it becomes a tedious task if we want to declare multiple variables of this type. To overcome the problem, we use a typedef keyword.

```
1. typedef unsigned int unit;
```

In the above statement, we have declared the unit variable of type unsigned int by using a typedef keyword.

Now, we can create the variables of type unsigned int by writing the following statement:

```
1. unit a, b;
```

instead of writing:

```
1. unsigned int a, b;
```

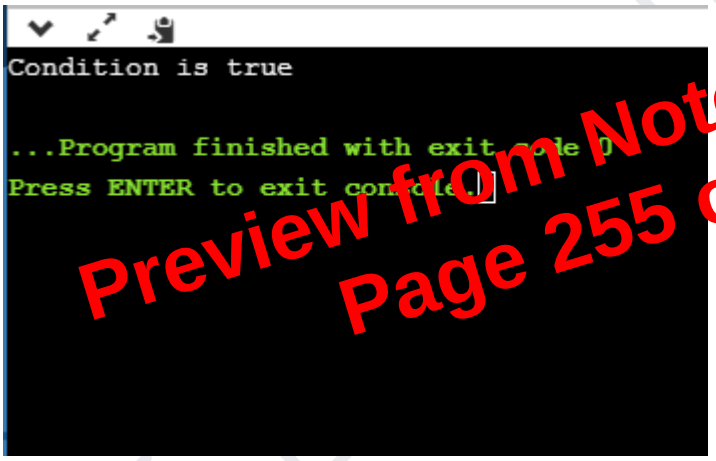
Till now, we have observed that the typedef keyword provides a nice shortcut by providing an alternative name for an already existing variable. This keyword is useful when we are dealing with the long data type especially, structure declarations.

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

Let's see a simple program of "&&" operator.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int x = 4;
5.     int y = 10;
6.     if ( (x <10) && (y>5))
7.     {
8.         printf("Condition is true");
9.     }
10.       else
11.       printf("Condition is false");
12.       return 0;
13.  }
```

Output



Factorial Program using recursion in C

Let's see the factorial program in c using recursion.

```
1. #include<stdio.h>
2.
3. long factorial(int n)
4. {
5.   if (n == 0)
6.     return 1;
7.   else
8.     return(n * factorial(n-1));
9. }
10.
11.  void main()
12.  {
13.    int number;
14.    long fact;
15.    printf("Enter a number: ");
16.    scanf("%d", &number);
17.
18.    fact = factorial(number);
19.    printf("Factorial of %d is %ld\n", number, fact);
20.    return 0;
21.  }
```

Output:

```
Enter a number: 6
Factorial of 5 is: 720
```

C Program to swap two numbers without third variable

We can swap two numbers without using third variable. There are two common ways to swap two numbers without using third variable:

1. By + and -
2. By * and /

Program 1: Using + and -

Let's see a simple c example to swap two numbers without using third variable.

```
1. #include<stdio.h>
2.  int main()
3. {
4. int a=10, b=20;
5. printf("Before swap a=%d b=%d",a,b);
6. a=a+b;//a=30 (10+20)
7. b=a-b;//b=10 (30-20)
8. a=a-b;//a=20 (30-10)
9. printf("\nAfter swap a=%d b=%d",a,b);
10.  return 0;
11. }
```

Output:

```
Before swap a=10 b=20
After swap a=20 b=10
```

My hobbies are: Watching news channels, Playing volleyball, Listening to music.

Possible Answer 2

"Good morning/afternoon/evening" sir/mam, it's my pleasure to introduce myself. I am Anshika Bansal. I belong to Meerut. I have done my B.Tech in CSE from Lovely Professional University.

While coming to my family members, there are 4 members including me. My father is a doctor, and any mother is a teacher. My younger sister will appear her 12th CBSE board exam this year.

Now coming to me, I am sweet smart, confident, and hardworking person. I am a cool hearted person, so usually see every difficulty with a positive side and keep myself always smiling which makes me stronger even more.

I can carry out any task assigned to me without hesitation.

My hobbies are dancing, Internet surfing, playing Chess, listening to music, watching the news channel. In my spare time, I like to read news on my phone and traveling to my hometown.

Thank you for giving this opportunity to introduce myself.

Possible Answer 3

"Good morning/afternoon/evening" sir/mam, it's my pleasure to introduce myself. I am Anshika Bansal. I belong to Meerut. I have done my B.Tech in CSE from Lovely Professional University.

I am carrying 5 years of experience at top Wall Street Companies. In my recent company, I led the development of an award-winning new trading platform. I can survive in a fast-paced environment.