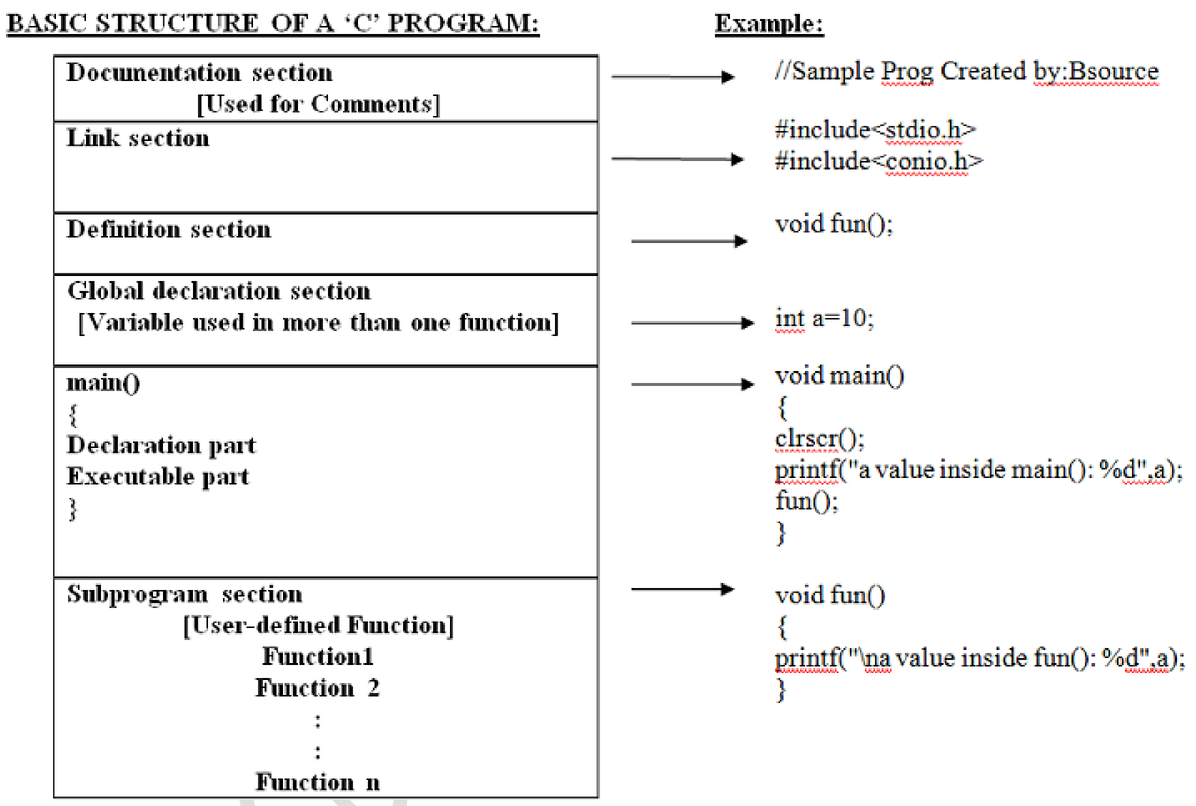
Basic structure of c program:-



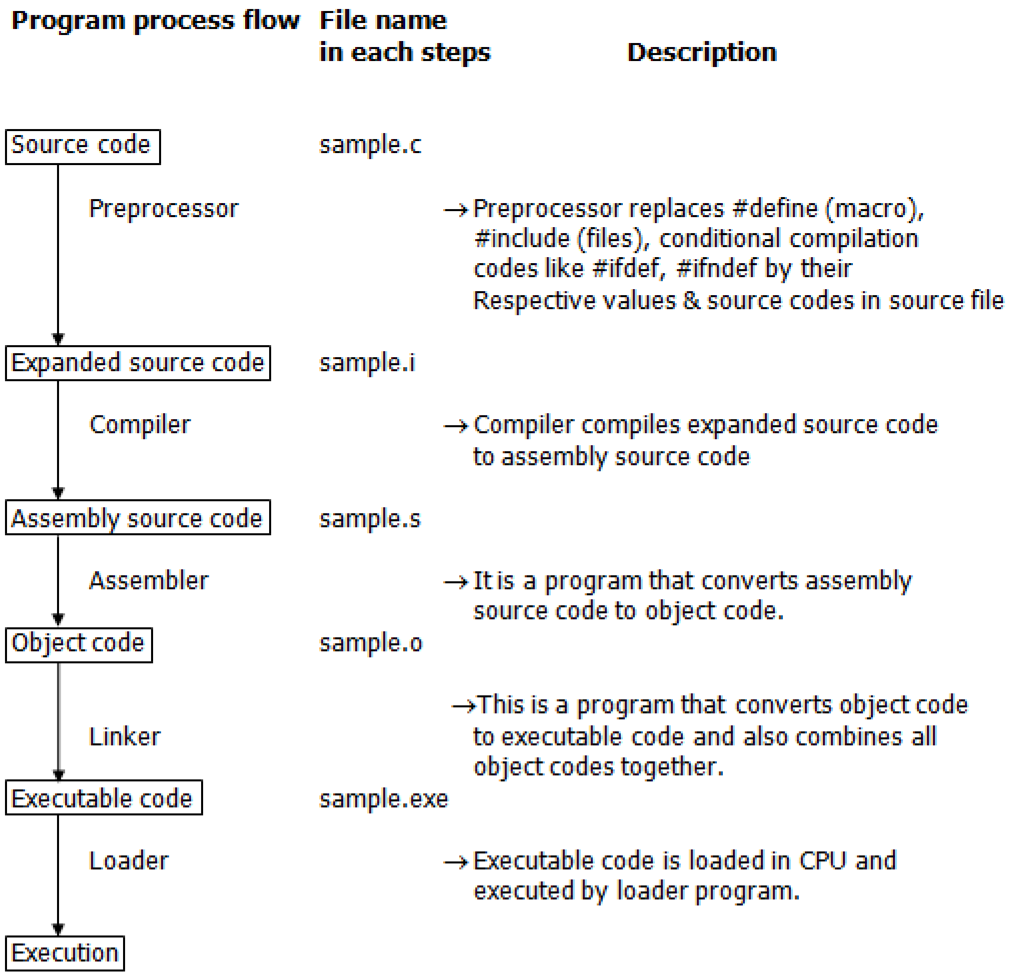
```
//Sample Prog Created by:Bsource

#include<stdio.h>
#include<conio.h>

void fun();

int a=10;

void main()
{
clrscr();
printf("a value inside main(): %d",a);
fun();
}

void fun()
{
printf("\na value inside fun(): %d",a);
}
```

Documentation	Consists of comments, some description of the program, programmer name and any other useful points that can be referenced later.
Link	Provides instruction to the compiler to link function from the library function
Definition	Consists of symbolic constants.
Global	Consists of function declaration and global variables.

declaration	

main( )
{
 	
}
	Every C program must have a main() function which is the starting point of the program execution.

Subprograms	User defined functions.



→Preprocessor replaces #define (macro), #include (files), conditional compilation codes like #ifdef, #ifndef by their Respective values & source codes in source file

→Compiler compiles expanded source code to assembly source code

→It is a program that converts assembly source code to object code.

→This is a program that converts object code to executable code and also combines all object codes together.

→Executable code is loaded in CPU and executed by loader program.

Sample Code of C "Hello World" Program

Example:

```
/* Author: Disha

Date: 5-07-2018

Description:

Writes the words "Hello, World!" on the screen */

#include<stdio.h>

Void main()

{

  Clrscr();

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

    getch();

}
```

or in a different way

```
/* Author: Disha

Date: 5-7-2018

Description:

Writes the words "Hello, World!" on the screen */

#include<stdio.h>

#include<conio.h>
```

```
int main()

{

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

    return 0;

}
```

Program Output:

```
Hello, World!
```

The above example has been used to print Hello, World! Text on the screen.

Let's look into various parts of the above C program.

/* Comments */	Comments are a way of explaining what makes a program. The compiler ignores comments and used by others to understand the code. or This is a comment block, which is ignored by the compiler. Comment can be used anywhere in the program to add info about program or code block, which will be helpful for developers to understand the existing code in the future easily.
#include<stdio.h>	stdio is standard for input / output, this allows us to use some commands which includes a file called stdio.h. or This is a preprocessor command. That notifies the compiler to include the header file stdio.h in the program before compiling the

	source-code.
int/void main()	int/void is a return value, which will be explained in a while.
main()	The main() is the main function where program execution begins. Every C program must contain only one main function. or This is the main function, which is the default entry point for every C program and the void in front of it indicates that it does not return a value.
Braces	Two curly brackets "{...}" are used to group all statements. or Curly braces which shows how much the main() function has its scope.
printf()	It is a function in C, which prints text on the screen. or This is another pre-defined function of C which is used to be displayed text string in the screen.
return 0	At the end of the main function returns value 0.

C PREPROCESSOR DIRECTIVES:

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with "#" symbol.

Variables:-

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory

The name of a variable can be composed of letters, digits, and the underscore character.

Syntax:-

type variable_list;
Example:-

```
int    i, j, k;
char   c, ch;
float  f, salary;
double d;
```

Defining Constants

There are two simple ways in C to define constants:

1.Using #define preprocessor

2. Using const keyword

1.The #define Preprocessor:-

Given below is the form to use #define preprocessor to define a constant:

#define identifier value

The following example explains it in detail:

```
#include <stdio.h>
#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'
int main()
{
 int area;
 area = LENGTH * WIDTH;
 printf("value of area : %d", area);
 printf("%c", NEWLINE);
 return 0;
}
```

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

```
value of area : 50
```

2.The const Keyword:-

You can use const prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

The following example explains it in detail:

```
#include <stdio.h>
int main()
{
 const int LENGTH = 10;
 const int WIDTH = 5;
 const char NEWLINE = '\n';
 int area;

 area = LENGTH * WIDTH;
 printf("value of area : %d", area);
 printf("%c", NEWLINE);
 return 0;
}
```

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

```
value of area : 50
```

Note that it is a good programming practice to define constants in CAPITALS

What is Data Type in C Programming ?

1. A Data Type is a **Type of Data**.
2. Data Type is a Data Storage Format that can contain a **Specific Type or Range of Values**.
3. When computer programs store data in variables, each variable must be assigned a **specific data type**.

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

size Required to Store Variable of Different Data Types

Format Specifier	Associated Datatype	Description
%c	Char	Used to format single character
%d or %i	Int	Used to format a signed integer
%u	Int	Used to format an unsigned integer in decimal form
%o	Int	Used to format an unsigned integer in octal form
%x or %X	Int	Used to format an unsigned integer in hex form
%h	Int	Used to format an short integer
%e or %E	float or double	Used to format a float or double in exponential form
%f	float or double	Used to format a float or double in decimal format
%s	char[]	Used to format a string/character sequence

Example of Print Format Specfiers

Now that we know how to print any data type, let's take a look at a simple example.

```
#include <stdio.h>

 int main()

{

    int i = 30;

    char c = 'z';

    float f = 5.67;

    char s[] = "This is a string";

    printf("i=%d, c=%c, f=%f, s=%s", i,c,f,s);

    return 0;

}
```

Output:

```
i=30, c=z, f=5.670000, s=This is a string
```

Storage Classes:-

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program:

- auto

- register
- static
- extern

1.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 within the same storage class. ‘auto’ can only be used within functions, i.e., local variables.

2.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.

3.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 class data member, 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
C Programming
26
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

4.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)
{
C Programming
27
 printf("count is %d\n", count);
}
```

Here, extern is being used to declare count in the second file, whereas 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: 5

Operator:-

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

We will, in this chapter, look into the way each operator works.

Arithmetic Operators

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

Show Examples

Operator	Description	Example
+	Adds two operands.	A + B = 30
−	Subtracts second operand from the first.	A − B =

		-10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

Relational Operators

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

Show Examples

Operator	**Description**	**Example**
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes	(A != B) is true.

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

Logical Operators

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

Show Examples

Operator	**Description**	**Example**
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.

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

Bitwise Operators

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

P	Q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows −

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then −

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -60, i.e,. 1100 0100 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits	A >> 2 = 15 i.e., 0000

	specified by the right operand.	1111

Assignment Operators

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

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A

/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is

		same as C = C | 2

Misc Operators ↦ sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

Show Examples

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

Operators Precedence in C

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

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

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

Show Examples

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

Example

Try the following example to understand all the bitwise operators available in C:

```
#include <stdio.h>
main()
{
C Programming
36
 unsigned int a = 60; /* 60 = 0011 1100 */
 unsigned int b = 13; /* 13 = 0000 1101 */
 int c = 0;
 c = a & b; /* 12 = 0000 1100 */
 printf("Line 1 - Value of c is %d\n", c );
 c = a | b; /* 61 = 0011 1101 */
 printf("Line 2 - Value of c is %d\n", c );
 c = a ^ b; /* 49 = 0011 0001 */
 printf("Line 3 - Value of c is %d\n", c );
 c = ~a; /*-61 = 1100 0011 */
 printf("Line 4 - Value of c is %d\n", c );
 c = a << 2; /* 240 = 1111 0000 */
 printf("Line 5 - Value of c is %d\n", c );
 c = a >> 2; /* 15 = 0000 1111 */
 printf("Line 6 - Value of c is %d\n", c );
}
```

When you compile and execute the above program, it produces the following result:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 1
```

Example

Try following example to understand all the miscellaneous operators available in C:

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

```
{
 int a = 4;
 short b;
 double c;
 int* ptr;
C Programming
41
 /* example of sizeof operator */
 printf("Line 1 - Size of variable a = %d\n", sizeof(a) );
 printf("Line 2 - Size of variable b = %d\n", sizeof(b) );
 printf("Line 3 - Size of variable c= %d\n", sizeof(c) );
 /* example of & and * operators */
 ptr = &a; /* 'ptr' now contains the address of 'a'*/
 printf("value of a is %d\n", a);
 printf("*ptr is %d.\n", *ptr);
 /* example of ternary operator */
 a = 10;
 b = (a == 1) ? 20: 30;
 printf( "Value of b is %d\n", b );
 b = (a == 10) ? 20: 30;
 printf( "Value of b is %d\n", b );
}
```

When you compile and execute the above program, it produces the following result:

```
value of a is 4
*ptr is 4.
Value of b is 30
Value of b is 20
```

Decision making in C

Decision making is about deciding the order of execution of statements based on certain conditions or repeat a group of statements until certain specified conditions are met. C language handles decision-making by supporting the following statements,

- if statement
- switch statement
- conditional operator statement (? : operator)
- goto statement

Decision making with if statement

The if statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple if statement
2. if....else statement
3. Nested if....else statement
4. Using else if statement

Simple if statement

The general form of a simple if statement is,

```
if(expression)
{
    statement inside;
}
    statement outside;
```

If the *expression* returns true, then the **statement-inside** will be executed, otherwise **statement-inside** is skipped and only the **statement-outside** is executed.

Example:

```
#include <stdio.h>

void main( )
{
    int x, y;
    x = 15;
    y = 13;
    if (x > y )
    {
        printf("x is greater than y");
    }
}
```

```
x is greater than y
```

---

if...else statement

The general form of a simple if...else statement is,

```
if(expression)
{
    statement block1;
}
else
{
    statement block2;
}
```

If the *expression* is true, the **statement-block1** is executed, else **statement-block1** is skipped and **statement-block2** is executed.

Example:

```
#include <stdio.h>

void main( )
{
    int x, y;
    x = 15;
    y = 18;
    if (x > y )
    {
        printf("x is greater than y");
    }
    else
    {
        printf("y is greater than x");
    }
}

y is greater than x
```

Nested if....else statement

The general form of a nested if...else statement is,

```
if( expression )
{
   if( expression1 )
   {
      statement block1;
   }
   else
   {
      statement block2;
   }
}
else
{
   statement block3;
}
```

if *expression* is false then **statement-block3** will be executed, otherwise the execution continues and enters inside the first if to perform the check for the next if block, where if *expression 1* is true the **statement-block1** is executed otherwise **statement-block2** is executed.

Example:

```
#include <stdio.h>

void main( )
{
   int a, b, c;
   printf("Enter 3 numbers...");
   scanf("%d%d%d",&a, &b, &c);
```

```
    if(a > b)
    {
        if(a > c)
        {
            printf("a is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
    else
    {
        if(b > c)
        {
            printf("b is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
}
```

else if ladder

The general form of else-if ladder is,

```
if(expression1)
```

```
{
   statement block1;
}
else if(expression2)
{
   statement block2;
}
else if(expression3 )
{
   statement block3;
}
else
   default statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as a **true** condition is found, the statement associated with it is executed.

Example :

```
#include <stdio.h>

void main( )
{
   int a;
   printf("Enter a number...");
   scanf("%d", &a);
   if(a%5 == 0 && a%8 == 0)
   {
      printf("Divisible by both 5 and 8");
   }
   else if(a%8 == 0)
```

```
    {
        printf("Divisible by 8");
    }
    else if(a%5 == 0)
    {
        printf("Divisible by 5");
    }
    else
    {
        printf("Divisible by none");
    }
}
```

Points to Remember

1. In if statement, a single statement can be included without enclosing it into curly braces { ... }
2. int a = 5;
3. if(a > 4)

   printf("success");

   No curly braces are required in the above case, but if we have more than one statement inside ifcondition, then we must enclose them inside curly braces.

4. == must be used for comparison in the expression of if condition, if you use = the expression will always return **true**, because it performs assignment not comparison.
5. Other than **0(zero)**, all other values are considered as **true**.
6. if(27)

```
printf("hello");
```

In above example, **hello** will be printed.

Switch statement in C

When you want to solve multiple option type problems, for example: Menu like program, where one value is associated with each option and you need to choose only one at a time, then, switchstatement is used.

Switch statement is a control statement that allows us to choose only one choice among the many given choices. The expression in switch evaluates to return an integral value, which is then compared to the values present in different cases. It executes that block of code which matches the case value. If there is no match, then **default** block is executed(if present). The general form of switch statement is,

```
switch(expression)
{
   case value-1:
          block-1;
          break;
   case value-2:
          block-2;
          break;
   case value-3:
          block-3;
          break;
   case value-4:
          block-4;
             break;
   default:
             default-block;
          break;
```

}

Rules for using switch statement

1. The expression (after switch keyword) must yield an **integer** value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
2. The case **label** values must be unique.
3. The case label must end with a colon(:)
4. The next line, after the **case** statement, can be any valid C statement.

Points to Remember

1. We don't use those expressions to evaluate switch case, which may return floating point values or strings or characters.
2. break statements are used to **exit** the switch block. It isn't necessary to use break after each block, but if you do not use it, then all the consecutive blocks of code will get executed after the matching block.

```
int i = 1;
switch(i)
{
   case 1:
      printf("A");        // No break
   case 2:
      printf("B");        // No break
   case 3:
      printf("C");
      break;
    }
```

Output:-

```
A B C
```

The output was supposed to be only **A** because only the first case matches, but as there is no break statement after that block, the next blocks are executed too, until it a break statement in encountered or the execution reaches the end of the switch block.

3. **default** case is executed when none of the mentioned case matches the switch expression. The default case can be placed anywhere in the switch case. Even if we don't include the default case, switch statement works.
4. Nesting of switch statements are allowed, which means you can have switch statements inside another switch block. However, nested switch statements should be avoided as it makes the program more complex and less readable.

Example of switch statement

```
#include<stdio.h>
void main( )
{
   int a, b, c, choice;
   while(choice != 3)
   {
      /* Printing the available options */
```

```
        printf("\n 1. Press 1 for addition");
        printf("\n 2. Press 2 for subtraction");
        printf("\n Enter your choice");
        /* Taking users input */
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
                printf("Enter 2 numbers");
                scanf("%d%d", &a, &b);
                c = a + b;
                printf("%d", c);
                break;
            case 2:
                printf("Enter 2 numbers");
                scanf("%d%d", &a, &b);
                c = a - b;
                printf("%d", c);
                break;
            default:
                printf("you have passed a wrong key");
                printf("\n press any key to continue");
        }
    }
}
```

Difference between switch and if

- if statements can evaluate float conditions. switch statements cannot evaluate floatconditions.
- if statement can evaluate relational operators. switch statement cannot evaluate relational operators i.e they are not allowed in switch statement.

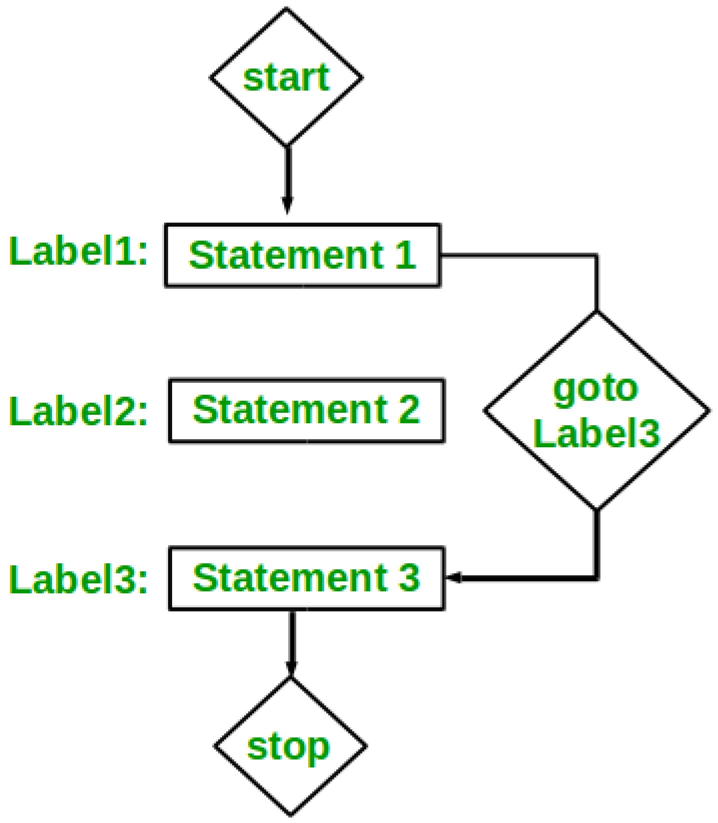
---

goto statement in C:-

The goto statement is a jump statement which is sometimes also referred to as unconditional jump statement. The goto statement can be used to jump from anywhere to anywhere within a function.
Syntax:

```
Syntax1      |  Syntax2
----------------------------
goto label;  |   label:
.            |   .
.            |   .
.            |   .
label:       |   goto label;
```

In the above syntax the first line tells the compiler to go to or jump to the statement marked as label. Here label is a user defined identifier which indicates the target statement. The statement immediately followed after ‘label:’ is the destination statement. The ‘label:’ can also appear before the ‘goto label;’ statement in above syntax.



Below are some examples on how to use goto statement:

Examples:

- **Syntax1 Example**: In this case we will see a situation similar to as shown in Syntax1 above. Suppose we need to write a program where we need to check if a number is even or not and print accordingly using goto statement. Below program explains how to do this:

```
// C/C++ program to check if a number is
// even or not using goto statement
#include <iostream>
using namespace std;

// function to check even or not
void checkEvenOrNot(int num)
{
   if (num % 2 == 0)
      goto even; // jump to even
   else
      goto odd; // jump to odd
even:
    cout << num << " is evenn";
    return; // return if even
odd:
    cout << num << " is oddn";
}

// Driver program to test above function
int main()
{
    int num = 26;
    checkEvenOrNot(num);
    return 0;
}
```

Output:-

```
26 is even
```

Syntax2:-

In this case we will see a situation similar to as shown in Syntax1 above. Suppose we need to write a program which prints numbers from 1 to 10 using goto statement. Below program explains how to do this

```
// C/C++ program to print numbers
// from 1 to 10 using goto statement
#include <iostream>
using namespace std;

// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    cout << n << " ";
    n++;
    if (n <= 10)
        goto label;
}
```

```
// Driver program to test above function
int main()
{
   printNumbers();
   return 0;
}
```

Output:-

```
1 2 3 4 5 6 7 8 9 10
```

Disadvantages of using goto statement:

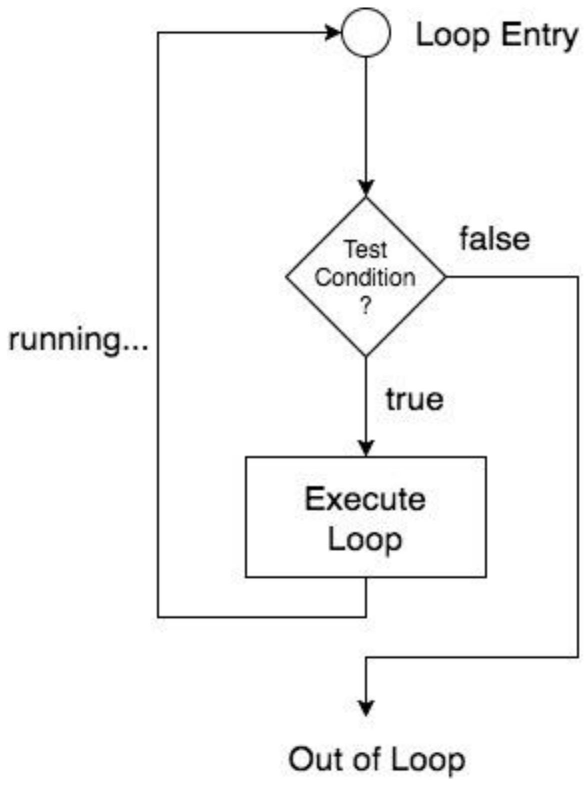
1. The use of goto statement is highly discouraged as it makes the program logic very complex.
2. use of goto makes really hard to modify the program.
3. Use of goto can be simply avoided using break and continue statements.

Loops in C

loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

How it Works

The below diagram depicts a loop execution,



As per the above diagram, if the Test Condition is true, then the loop is executed, and if it is false then the execution breaks out of the loop. After the loop is successfully executed the execution again starts from the Loop entry and again checks for the Test condition, and this keeps on repeating.

The sequence of statements to be executed is kept inside the curly braces { } known as the **Loop body**. After every execution of the loop body, **condition** is verified, and if it is found to be **true** the loop body is executed again. When the condition check returns **false**, the loop body is not executed, and execution breaks out of the loop.

Types of Loop

There are 3 types of Loop in C language, namely:

1. while loop
2. for loop
3. do while loop

1.while loop

while loop can be addressed as an **entry control** loop. It is completed in 3 steps.

- Variable initialization.(e.g int x = 0;)
- condition(e.g while(x <= 10))
- Variable increment or decrement (x++ or x-- or x = x + 2)

Syntax :

```
variable initialization;
while(condition)
{
   statements;
   variable increment or decrement;
}
```

Example: Program to print first 10 natural numbers

```
#include<stdio.h>

void main( )
{
   int x;
   x = 1;
   while(x <= 10)
   {
      printf("%d\t", x);
      /* below statement means, do x = x+1, increment x by 1*/
      x++;
```

```
    }
}
```

```
1 2 3 4 5 6 7 8 9 10
```

2.for loop

for loop is used to execute a set of statements repeatedly until a particular condition is satisfied. We can say it is an **open ended loop.**. General format is,

```
for(initialization; condition; increment/decrement)
{
    statement-block;
}
```

In for loop we have exactly two semicolons, one after initialization and second after the condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. But it can have only one **condition**.

The for loop is executed as follows:

1. It first evaluates the initialization code.
2. Then it checks the condition expression.
3. If it is **true**, it executes the for-loop body.
4. Then it evaluate the increment/decrement condition and again follows from step 2.
5. When the condition expression becomes **false**, it exits the loop.

Example: Program to print first 10 natural numbers

```
#include<stdio.h>
```

```
void main( )
{
    int x;
    for(x = 1; x <= 10; x++)
    {
        printf("%d\t", x);
    }
}
```

```
1 2 3 4 5 6 7 8 9 10
```

Nested for loop

We can also have nested for loops, i.e one for loop inside another for loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)
{
    for(initialization; condition; increment/decrement)
    {
        statement ;
    }
}
```

Example: Program to print half Pyramid of numbers

```
#include<stdio.h>
```

```
void main( )
{
    int i, j;
    /* first for loop */
    for(i = 1; i < 5; i++)
    {
        printf("\n");
        /* second for loop inside the first */
        for(j = i; j > 0; j--)
        {
            printf("%d", j);
        }
    }
}
```

```
1
21
321
4321
54321
```

3.do while loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of do-while loop. do statement evaluates the body of the loop first and at the end, the condition is checked using while statement. It means that the body of the loop will be executed at least once, even though the starting condition inside while is initialized to be **false**. General syntax is,

```
do
{
    .....
    .....
}
while(condition)
```

Example: Program to print first 10 multiples of 5.

```
#include<stdio.h>

void main()
{
    int a, i;
    a = 5;
    i = 1;
    do
    {
        printf("%d\t", a*i);
        i++;
    }
    while(i <= 10);
}
```

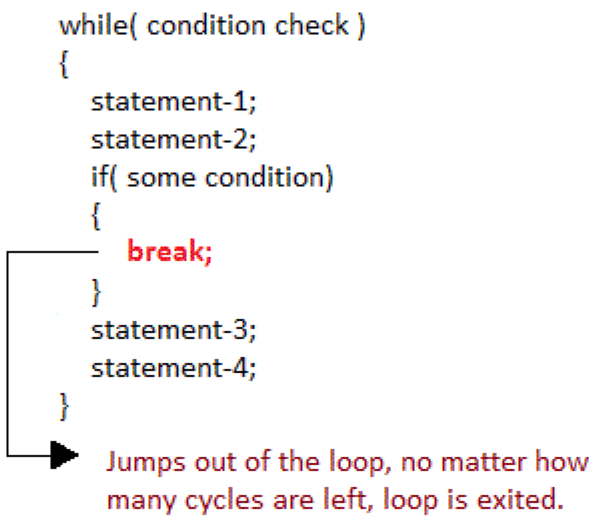
```
5 10 15 20 25 30 35 40 45 50
```

Jumping Out of Loops

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes **true**. This is known as jumping out of loop.

1) break statement

When break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

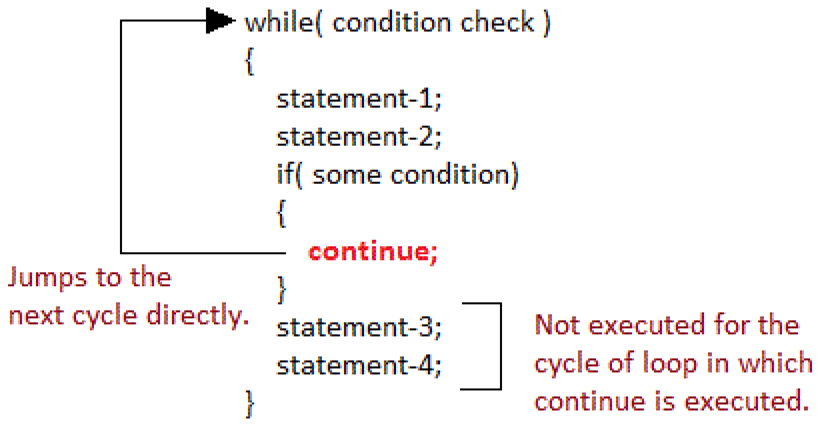


```
while( condition check )
{
   statement-1;
   statement-2;
   if( some condition)
   {
      break;
   }
   statement-3;
   statement-4;
}
```

Jumps out of the loop, no matter how many cycles are left, loop is exited.

2) continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering continue, cursor leave the current cycle of loop, and starts with the next cycle.



```
while( condition check )
{
    statement-1;
    statement-2;
    if( some condition)
    {
        continue;
    }
    statement-3;
    statement-4;
}
```

Jumps to the next cycle directly.
Not executed for the cycle of loop in which continue is executed.

Arrays in C:-

In C language, arrays are reffered to as structured data types. An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.

Here the words,

- **finite** *means* data range must be defined.
- **ordered** *means* data must be stored in continuous memory addresses.
- **homogenous** *means* data must be of similar data type.

Example where arrays are used,

- to store list of Employee or Student names,
- to store marks of students,
- or to store list of numbers or characters etc.

Since arrays provide an easy way to represent data, it is classified amongst the data structures in C. Other data structures in c are **structure**, **lists**, **queues**, **trees** etc. Array can be used to represent not only simple list of data but also table of data in two or three dimensions.

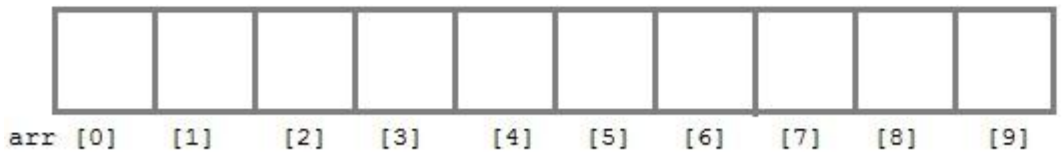
Declaring an Array

Like any other variable, arrays must be declared before they are used. General form of array declaration is,

data-type variable-name[size];

/* Example of array declaration */

int arr[10];



Here int is the data type, arr is the name of the array and 10 is the size of array. It means array arr can only contain 10 elements of int type.

Index of an array starts from 0 to **size-1** i.e first element of arr array will be stored at arr[0]address and the last element will occupy arr[9].

Initialization of an Array

After an array is declared it must be initialized. Otherwise, it will contain **garbage** value(any random value). An array can be initialized at either **compile time** or at **runtime**.

Compile time Array initialization

Compile time initialization of array elements is same as ordinary variable initialization. The general form of initialization of array is,

data-type array-name[size] = { list of values };

```
/* Here are a few examples */
int marks[4]={ 67, 87, 56, 77 };    // integer array initialization


float area[5]={ 23.4, 6.8, 5.5 };   // float array initialization


int marks[4]={ 67, 87, 56, 77, 59 };    // Compile time error
```

One important thing to remember is that when you will give more initializer(array elements) than the declared array size than the **compiler** will give an error.

```
#include<stdio.h>

void main()
{
   int i;
   int arr[] = {2, 3, 4};      // Compile time array initialization
   for(i = 0 ; i < 3 ; i++)
   {
      printf("%d\t",arr[i]);
   }
}
```

```
2 3 4
```

Runtime Array initialization

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large arrays, or to initialize arrays with user specified values. Example,

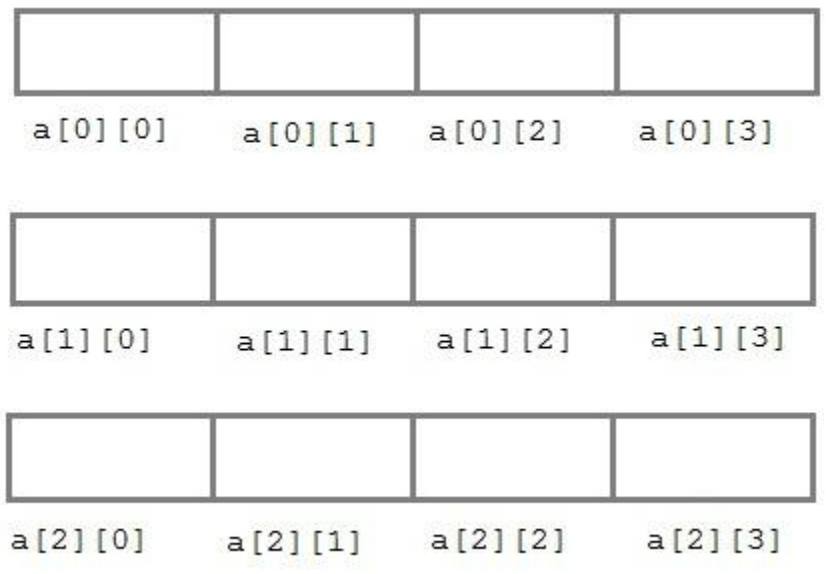
```
#include<stdio.h>


void main()
{
    int arr[4];
    int i, j;
    printf("Enter array element");
    for(i = 0; i < 4; i++)
    {
        scanf("%d", &arr[i]);    //Run time array initialization
    }
    for(j = 0; j < 4; j++)
    {
        printf("%d\n", arr[j]);
    }
}
```

Two dimensional Arrays

C language supports multidimensional arrays also. The simplest form of a multidimensional array is the two-dimensional array. Both the row's and column's index begins from 0.

Two-dimensional arrays are declared as follows, data-type array-name[row-size][column-size]

/* Example */
int a[3][4];



An array can also be declared and initialized together. For example,

```
int arr[][3] = {
   {0,0,0},
   {1,1,1}
};
```

Note: We have not assigned any row value to our array in the above example. It means we can initialize any number of rows. But, we must always specify number of columns, else it will give a compile time error. Here, a 2*3 multi-dimensional matrix is created.

Runtime initialization of a two dimensional Array

```
#include<stdio.h>
```

```
void main()
{
   int arr[3][4];
   int i, j, k;
   printf("Enter array element");
   for(i = 0; i < 3;i++)
   {
      for(j = 0; j < 4; j++)
      {
         scanf("%d", &arr[i][j]);
      }
   }
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 4; j++)
      {
         printf("%d", arr[i][j]);
      }
   }
}
```

String and Character Array:-

String is a sequence of characters that is treated as a single data item and terminated by null character '\0'. Remember that C language does not support strings as a data type. A **string** is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

For example: The string "hello world" contains 12 characters including '\0' character which is automatically added by the compiler at the end of the string.

Declaring and Initializing a string variables

There are different ways to initialize a character array variable.

char name[13] = "StudyTonight";   // valid character array initialization

char name[10] = {'L','e','s','s','o','n','s','\0'};   // valid initialization

Remember that when you initialize a character array by listing all of its characters separately then you must supply the '\0' character explicitly.

Some examples of illegal initialization of character array are,

char ch[3] = "hell";   // Illegal

char str[4];

str = "hell";   // Illegal

String Input and Output

Input function scanf() can be used with **%s** format specifier to read a string input from the terminal. But there is one problem with scanf() function, it terminates its input on the first white space it encounters. Therefore if you try to read an input string "Hello World" using scanf() function, it will only read **Hello** and terminate after encountering white spaces.

However, C supports a format specification known as the **edit set conversion code %[..]** that can be used to read a line containing a variety of characters, including white spaces.

#include<stdio.h>

#include<string.h>

```
void main()
{
   char str[20];
   printf("Enter a string");
   scanf("%[^\n]", &str);  //scanning the whole string, including the white spaces
   printf("%s", str);
}
```

Another method to read character string with white spaces from terminal is by using the gets()function.

```
char text[20];
gets(text);
printf("%s", text);
```

String Handling Functions

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in **string.h** library. Hence, you must include **string.h** header file in your programs to use these functions.

The following are the most commonly used string handling functions.

Method	**Description**
strcat()	It is used to concatenate(combine) two strings
strlen()	It is used to show length of a string
strrev()	It is used to show reverse of a string

strcpy()	Copies one string into another
strcmp()	It is used to compare two string

---

strcat() function

strcat("hello", "world");

strcat() function will add the string **"world"** to **"hello"** i.e it will ouput helloworld.

---

strlen() function

strlen() function will return the length of the string passed to it.

```
int j;
j = strlen("studytonight");
printf("%d",j);
```

```
12
```

---

strcmp() function

strcmp() function will return the ASCII difference between first unmatching character of two strings.

```
int j;
j = strcmp("study", "tonight");
printf("%d",j);
```

```
-1
```

strcpy() function

It copies the second string argument to the first string argument.

```
#include<stdio.h>
#include<string.h>

int main()
{
   char s1[50];
   char s2[50];

   strcpy(s1, "StudyTonight");     //copies "studytonight" to string s1
   strcpy(s2, s1);     //copies string s1 to string s2

   printf("%s\n", s2);

   return(0);
}
```

```
StudyTonight
```

strrev() function

It is used to reverse the given string expression.

```
#include<stdio.h>

int main()
{
    char s1[50];

    printf("Enter your string: ");
    gets(s1);
    printf("\nYour reverse string is: %s",strrev(s1));
    return(0);
}
```

```
Enter your string: studytonight
Your reverse string is: thginotyduts
```

Functions in C:-

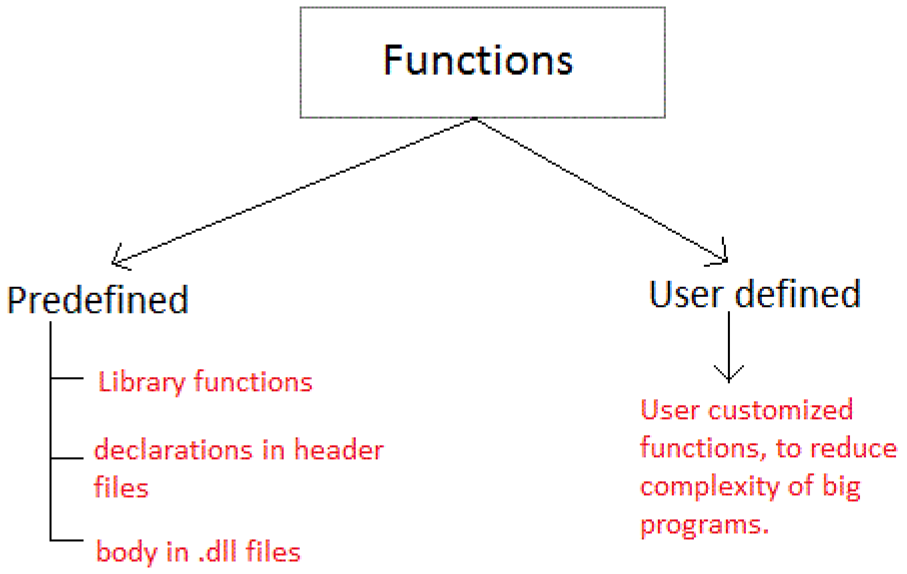
A **function** is a block of code that performs a particular task.

There are many situations where we might need to write same line of code for more than once in a program. This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer. So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required.

These functions defined by the user are also know as **User-defined Functions**

C functions can be classified into two categories,

1. **Library functions**
2. **User-defined functions**


Functions
Predefined
Library functions
declarations in header files
body in .dll files
User defined
User customized functions, to reduce complexity of big programs.

Library functions are those functions which are already defined in C library, example printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

A **User-defined functions** on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

Benefits of Using Functions

1. It provides modularity to your program's structure.
2. It makes your code reusable. You just have to call the function by its name to use it, wherever required.
3. In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
4. It makes the program more readable and easy to understand.

Function Declaration

General syntax for function declaration is,

returntype functionName(type1 parameter1, type2 parameter2,...);

Like any variable or an array, a function must also be declared before its used. Function declaration informs the compiler about the function name, parameters is accept, and its return type. The actual body of the function can be defined separately. It's also called as **Function Prototyping**. Function declaration consists of 4 parts.

- returntype
- function name
- parameter list
- terminating semicolon

returntype

When a function is declared to perform some sort of calculation or any operation and is expected to provide with some result at the end, in such cases, a return statement is added at the end of function body. Return type specifies the type of value(int, float, char, double) that function is expected to return to the program which called the function.

Note: In case your function doesn't return any value, the return type would be void.

---

functionName

Function name is an identifier and it specifies the name of the function. The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

---

parameter list

The parameter list declares the type and number of arguments that the function expects when it is called. Also, the parameters in the parameter list receives the argument values when the function is called. They are often referred as **formal parameters**.

Example

Let's write a simple program with a main() function, and a user defined function to multiply two numbers, which will be called from the main() function.

```
#include<stdio.h>

int multiply(int a, int b);     // function declaration

int main()
{
    int i, j, result;
    printf("Please enter 2 numbers you want to multiply...");
    scanf("%d%d", &i, &j);

    result = multiply(i, j);        // function call
    printf("The result of muliplication is: %d", result);

    return 0;
}

int multiply(int a, int b)
{
    return (a*b);       // function defintion, this can be done in one line
```

}

Function definition Syntax

Just like in the example above, the general syntax of function definition is,

```
returntype functionName(type1 parameter1, type2 parameter2,...)
{
   // function body goes here
}
```

The first line *returntype* **functionName(type1 parameter1, type2 parameter2,...)** is known as **function header** and the statement(s) within curly braces is called **function body**.

Note: While defining a function, there is no semicolon(;) after the parenthesis in the function header, unlike while declaring the function or calling the function.

functionbody

The function body contains the declarations and the statements(algorithm) necessary for performing the required task. The body is enclosed within curly braces { ... } and consists of three parts.

- **local** variable declaration(if required).
- **function statements** to perform the task inside the function.
- a **return** statement to return the result evaluated by the function(if return type is void, then no return statement is required).

Calling a function

When a function is called, control of the program gets transferred to the function.

functionName(argument1, argument2,...);

In the example above, the statement multiply(i, j); inside the main() function is function call.

Types of User-defined Functions in C Programming

1. No arguments passed and no return Value
2. No arguments passed but a return value
3. Argument passed but no return value
4. Argument passed and a return value

1.Function with no arguments and no return value

Such functions can either be used to display information or they are completely dependent on user inputs.

Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>

void greatNum();      // function declaration

int main()
{
   greatNum();       // function call
   return 0;
}

void greatNum()       // function definition
{
   int i, j;
   printf("Enter 2 numbers that you want to compare...");
```

```
    scanf("%d%d", &i, &j);
    if(i > j) {
        printf("The greater number is: %d", i);
    }
    else {
        printf("The greater number is: %d", j);
    }
}
```

2.Function with no arguments and a return value

We have modified the above example to make the function greatNum() return the number which is greater amongst the 2 input numbers.

```
#include<stdio.h>

int greatNum();       // function declaration

int main()
{
    int result;
    result = greatNum();        // function call
    printf("The greater number is: %d", result);
    return 0;
}

int greatNum()        // function definition
{
    int i, j, greaterNum;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    if(i > j) {
        greaterNum = i;
    }
    else {
        greaterNum = j;
    }
    // returning the result
    return greaterNum;
}
```

3.Function with arguments and no return value

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function greatNum() take two intvalues as arguments, but it will not be returning anything.

```
#include<stdio.h>

void greatNum(int a, int b);       // function declaration

int main()
{
    int i, j;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    greatNum(i, j);        // function call
    return 0;
}

void greatNum(int x, int y)       // function definition
{
   if(x > y) {
      printf("The greater number is: %d", x);
   }
   else {
      printf("The greater number is: %d", y);
   }
}
```

4.Function with arguments and a return value

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```
#include<stdio.h>

int greatNum(int a, int b);      // function declaration

int main()
{
   int i, j, result;
   printf("Enter 2 numbers that you want to compare...");
   scanf("%d%d", &i, &j);
   result = greatNum(i, j); // function call
   printf("The greater number is: %d", result);
   return 0;
}
```

```
int greatNum(int x, int y)       // function definition
{
   if(x > y) {
      return x;
   }
   else {
      return y;
   }
}
```

Nesting of Functions

C language also allows nesting of functions i.e to use/call one function inside another function's body. We must be careful while using nested functions, because it may lead to infinite nesting.

```
function1()
{
   // function1 body here

   function2();

   // function1 body here
}
```

If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate.

Not able to understand? Lets consider that inside the main() function, function1() is called and its execution starts, then inside function1(), we have a call for function2(), so the control of program will go to the function2(). But as function2() also has a call to function1() in its body, it will call function1(), which will again call function2(), and this will go on for infinite times, until you forcefully exit from program execution.

What is Recursion?

Recursion is a special way of nesting functions, where a function calls itself inside it. We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.

```
function1()
{
   // function1 body
   function1();
   // function1 body
}
```

Example: Factorial of a number using Recursion

```
#include<stdio.h>

int factorial(int x);       //declaring the function

void main()
{
   int a, b;

   printf("Enter a number...");
   scanf("%d", &a);
```

```
    b = factorial(a);       //calling the function named factorial
    printf("%d", b);
}

int factorial(int x) //defining the function
{
    int r = 1;
    if(x == 1)
        return 1;
    else
        r = x*factorial(x-1);       //recursion, since the function calls itself

    return r;
}
```

Similarly, there are many more applications of recursion in C language. Go to the programs section, to find out more programs using recursion.

Types of Function calls in C

Functions are called by their names, we all know that, then what is this tutorial for? Well if the function does not have any arguments, then to call a function you can directly use its name. But for functions with arguments, we can call a function in two different ways, based on how we specify the arguments, and these two ways are:

1. Call by Value
2. Call by Reference

1.Call by Value

Calling a function by value means, we pass the values of the arguments which are stored or copied into the formal parameters of the function. Hence, the original values are unchanged only the parameters inside the function changes.

```
#include<stdio.h>


void calc(int x);


int main()
{
   int x = 10;
   calc(x);
   // this will print the value of 'x'
   printf("\nvalue of x in main is %d", x);
   return 0;
}

void calc(int x)
{
   // changing the value of 'x'
   x = x + 10 ;
   printf("value of x in calc function is %d ", x);
}
```

```
value of x in calc function is 20
value of x in main is 10
```

In this case, the actual variable x is not changed. This is because we are passing the argument by value, hence a copy of x is passed to the function, which is updated during function execution, and that copied value in the function is destroyed when the function ends(goes out of scope). So the variable x inside the main() function is never changed and hence, still holds a value of 10.

But we can change this program to let the function modify the original x variable, by making the function calc() return a value, and storing that value in x.

```
#include<stdio.h>


int calc(int x);


int main()
{
   int x = 10;
   x = calc(x);
   printf("value of x is %d", x);
   return 0;
}


int calc(int x)
{
   x = x + 10 ;
   return x;
}
```

```
value of x is 20
```

2.Call by Reference

In call by reference we pass the address(reference) of a variable as argument to any function. When we pass the address of any variable as argument, then the function will have access to our variable, as it now knows where it is stored and hence can easily update its value.

In this case the formal parameter can be taken as a **reference** or a **pointer**(don't worry about pointers, we will soon learn about them), in both the cases they will change the values of the original variable.

```
#include<stdio.h>


void calc(int *p);      // functin taking pointer as argument


int main()
{
   int x = 10;
   calc(&x);      // passing address of 'x' as argument
   printf("value of x is %d", x);
   return(0);
}

void calc(int *p)      //receiving the address in a reference pointer variable
{
   /*
      changing the value directly that is
      stored at the address passed
   */
   *p = *p + 10;
}
```

```
value of x is 20
```

NOTE: If you do not have any prior knowledge of pointers, do study Pointers first. Or just go over this topic and come back again to revise this, once you have learned about pointers.

- **<u>How to pass Array to a Function</u>**

Whenever we need to pass a list of elements as argument to any function in C language, it is prefered to do so using an array. But how can we pass an array as argument to a function? Let's see how its done.

Declaring Function with array as a parameter

There are two possible ways to do so, one by using call by value and other by using call by reference.

We can either have an array as a parameter.

```
int sum (int arr[]);
```

Or, we can have a pointer in the parameter list, to hold the base address of our array.

```
int sum (int* ptr);
```

We will study the second way in details later when we will study pointers.

- **<u>Returning an Array from a function</u>**

We don't return an array from functions, rather we return a pointer holding the base address of the array to be returned. But we must, make sure that the array exists after the function ends i.e. the array is not local to the function.

```
int* sum (int x[])

{

   // statements

   return x ;
```

}

We will discuss about this when we will study pointers with arrays.

Passing arrays as parameter to function

Now let's see a few examples where we will pass a single array element as argument to a function, a one dimensional array to a function and a multidimensional array to a function.

1.Passing a single array element to a function

Let's write a very simple program, where we will declare and define an array of integers in our main() function and pass one of the array element to a function, which will just print the value of the element.

```
#include<stdio.h>

void giveMeArray(int a);

int main()
{
   int myArray[] = { 2, 3, 4 };
   giveMeArray(myArray[2]);       //Passing array element myArray[2] only.
   return 0;
}
```

```
void giveMeArray(int a)

{

    printf("%d", a);

}
```

Output:-

```
4
```

2.Passing a complete One-dimensional array to a function

To understand how this is done, let's write a function to find out average of all the elements of the array and print it.

We will only send in the name of the array as argument, which is nothing but the address of the starting element of the array, or we can say the starting memory address.

```
#include<stdio.h>


float findAverage(int marks[]);


int main()

{

    float avg;

    int marks[] = {99, 90, 96, 93, 95};
    avg = findAverage(marks);       // name of the array is passed as argument.

    printf("Average marks = %.1f", avg);

    return 0;

}


float findAverage(int marks[])

{

    int i, sum = 0;

    float avg;

    for (i = 0; i <= 4; i++) {

        sum += age[i];

    }

    avg = (sum / 5);

    return avg;

}
```

Output:-

```
94.6
```

3.Passing a Multi-dimensional array to a function

Here again, we will only pass the name of the array as argument.

```
#include<stdio.h>
```

```
void displayArray(int arr[3][3]);


int main()

{

    int arr[3][3], i, j;

    printf("Please enter 9 numbers for the array: \n");

    for (i = 0; i < 3; ++i)

    {

        for (j = 0; j < 3; ++j)

        {

            scanf("%d", &arr[i][j]);

        }

    }

    // passing the array as argument

    displayArray(arr);

    return 0;

}


void displayArray(int arr[3][3])

{

    int i, j;
    printf("The complete array is: \n");
    for (i = 0; i < 3; ++i)
    {
        // getting cursor to new line
        printf("\n");
        for (j = 0; j < 3; ++j)
        {
            // \t is used to provide tab space
            printf("%d\t", arr[i][j]);
        }
    }
}
```

```
Please enter 9 numbers for the array:
1
2
3
4
5
6
7
8
```

9

The complete array is:

1 2 3

4 5 6

7 8 9

Introduction to Structure:-

Structure is a user-defined datatype in C language which allows us to combine data of different types together.

For example:

If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc,

---

Defining a structure

struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived datatypes.

Syntax:

```
struct [structure_tag]
{
   //member variable 1
   //member variable 2
   //member variable 3
   ...
}[structure_variables];
```

As you can see in the syntax above, we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.

Note: The closing curly brace in the structure type declaration must be followed by a semicolon(;).

Example of Structure

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    // F for female and M for male
    char gender;
};
```

Here struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called **structure elements or members**.

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. **Student** is the name of the structure and is called as the **structure tag**.

Declaring Structure Variables

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways:

1) Declaring Structure variables separately

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   //F for female and M for male
   char gender;
};

struct Student S1, S2;      //declaring variables of struct Student
```

---

2) Declaring Structure variables with structure definition

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   //F for female and M for male
   char gender;
}S1, S2;
```

Here S1 and S2 are variables of structure Student. However this approach is not much recommended.

Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot . operator also called **period** or **member access** operator.

For example:

```
#include<stdio.h>
#include<string.h>

struct Student
{
   char name[25];
   int age;
   char branch[10];
   //F for female and M for male
   char gender;
};

int main()
{
   struct Student s1;

   /*
      s1 is a variable of Student type and
      age is a member of Student
   */
   s1.age = 18;
   /*
      using string function to add name
   */
   strcpy(s1.name, "Viraaj");
```

```
    /*
        displaying the stored values
    */
    printf("Name of Student 1: %s\n", s1.name);
    printf("Age of Student 1: %d\n", s1.age);

    return 0;
}
```

```
Name of Student 1: Viraaj
Age of Student 1: 18
```

We can also use scanf() to give values to structure members through terminal.

```
scanf(" %s ", s1.name);
scanf(" %d ", &s1.age);
```

Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
struct Patient
{
    float height;
    int weight;
    int age;
};

struct Patient p1 = { 180.75 , 73, 23 };    //initialization
or,
struct Patient p1;
p1.height = 180.75;     //initialization of each member separately
p1.weight = 73;
p1.age = 23;
```

- **Array of Structure**

We can also declare an array of **structure** variables. in which each element of the array will represent a **structure** variable. **Example :** struct employee emp[5];

The below program defines an array emp of size 5. Each element of the array emp is of type Employee.

```
#include<stdio.h>

struct Employee
{
   char ename[10];
   int sal;
};

struct Employee emp[5];
int i, j;
void ask()
{
   for(i = 0; i < 3; i++)
   {
      printf("\nEnter %dst Employee record:\n", i+1);
      printf("\nEmployee name:\t");
      scanf("%s", emp[i].ename);
      printf("\nEnter Salary:\t");
```

```
        scanf("%d", &emp[i].sal);
    }
    printf("\nDisplaying Employee record:\n");
    for(i = 0; i < 3; i++)
    {
        printf("\nEmployee name is %s", emp[i].ename);
        printf("\nSlary is %d", emp[i].sal);
    }
}
void main()
{
    ask();
}
```

➢ <u>Nested Structures</u>

Nesting of structures, is also permitted in C language. Nested structures means, that one structure has another stucture as member variable.

Example:

```
struct Student
{
    char[30] name;
    int age;
    /* here Address is a structure */
    struct Address
    {
        char[50] locality;
        char[50] city;
        int pincode;
    }addr;
};
```

➢ Structure as Function Arguments

We can pass a structure as a function argument just like we pass any other variable or an array as a function argument.

Example:

```
#include<stdio.h>

struct Student
{
    char name[10];
    int roll;
};

void show(struct Student st);

void main()
{
    struct Student std;
    printf("\nEnter Student record:\n");
    printf("\nStudent name:\t");
    scanf("%s", std.name);
    printf("\nEnter Student rollno.:\t");
    scanf("%d", &std.roll);
    show(std);
}

void show(struct Student st)
{
    printf("\nstudent name is %s", st.name);
    printf("\nroll is %d", st.roll);
}
```

➢ **Pointer to a Structure**

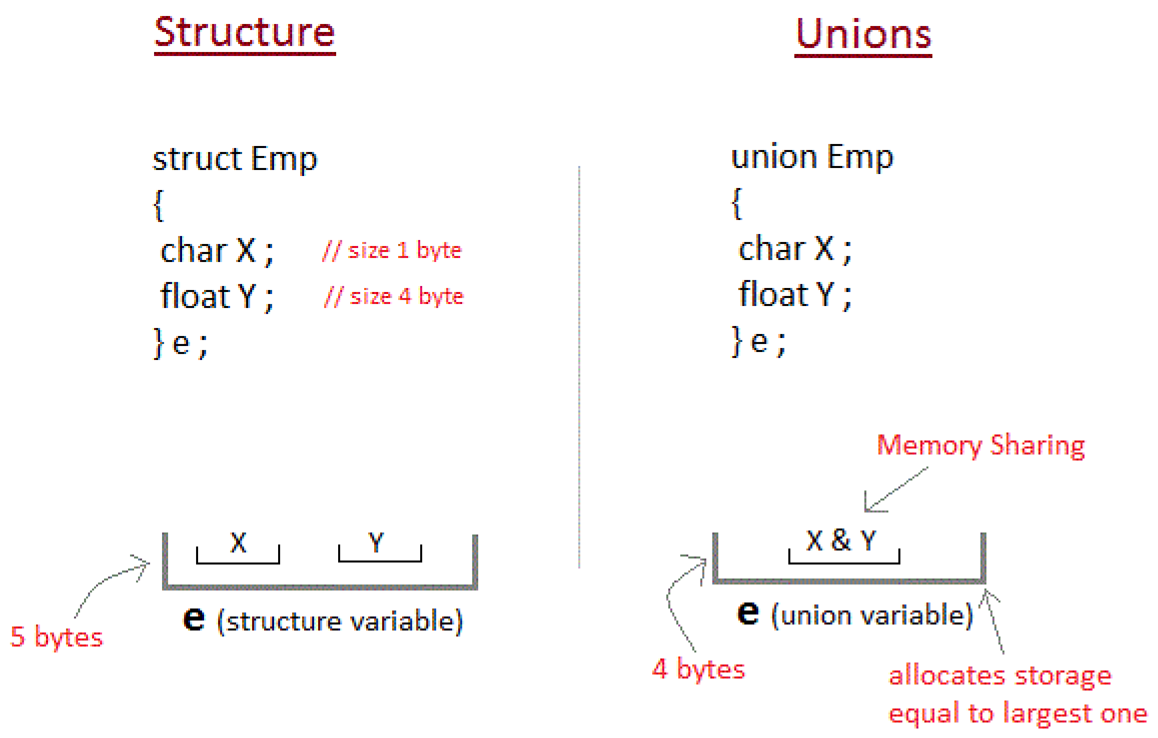
To access members of structure using the structure variable, we used the dot . operator. But when we have a pointer of structure type, we use arrow -> to access structure members.

```
#include <stdio.h>

struct my_structure {
    char name[20];
    int number;
    int rank;
};

int main()
{
    struct my_structure variable = {"StudyTonight", 35, 1};

    struct my_structure *ptr;
    ptr = &variable;

    printf("NAME: %s\n", ptr->name);
    printf("NUMBER: %d\n", ptr->number);
    printf("RANK: %d", ptr->rank);

    return 0;
}
```

```
NAME: StudyTonight
NUMBER: 35
RANK: 1
```

Unions in C Language:-

Unions are conceptually similar to **structures**. The syntax to declare/define a union is also similar to that of a structure. The only differences is in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.



This implies that although a **union** may contain many members of different types, **it cannot handle all the members at the same time**. A **union** is declared using the union keyword.

```
union item
{
   int m;
   float x;
   char c;
}It1;
```

This declares a variable It1 of type union item. This union contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for all the union variables, irrespective of their size. The compiler allocates the storage that is large enough to hold the largest variable type in the union.

In the union declared above the member x requires **4 bytes** which is largest amongst the members for a 16-bit machine. Other members of union will share the same memory address.

Accessing a Union Member

Syntax for accessing any union member is similar to accessing structure members,

```
union test
{
    int a;
    float b;
    char c;
}t;

t.a;    //to access members of union t
t.b;
t.c;
```

Time for an Example

```
#include <stdio.h>

union item
{
    int a;
    float b;
    char ch;
};

int main( )
```

```
{
   union item it;
   it.a = 12;
   it.b = 20.2;
   it.ch = 'z';

   printf("%d\n", it.a);
   printf("%f\n", it.b);
   printf("%c\n", it.ch);

   return 0;
}
```

```
-26426
20.1999
z
```

As you can see here, the values of a and b get corrupted and only variable c prints the expected result. This is because in union, the memory is shared among different data types. Hence, the only member whose value is currently stored will have the memory.

In the above example, value of the variable c was stored at last, hence the value of other variables is lost.

Introduction to Pointers:-

A Pointer in C language is a variable which holds the address of another variable of same data type.

Pointers are used to access memory and manipulate the address.

Address of a memory location" means?

Address in C

Whenever a variable is defined in C language, a memory location is assigned for it, in which it's value will be stored. We can easily check this memory address, using the & symbol.

If var is the name of the variable, then &var will give it's address.

Let's write a small program to see memory address of any variable that we define in our program.

```
#include<stdio.h>


void main()
{
   int var = 7;
   printf("Value of the variable var is: %d\n", var);
   printf("Memory address of the variable var is: %x\n", &var);
}
```

```
Value of the variable var is: 7
Memory address of the variable var is: bcc7a00
```

You must have also seen in the function scanf(), we mention &var to take user input for any variable var.

```
scanf("%d", &var);
```

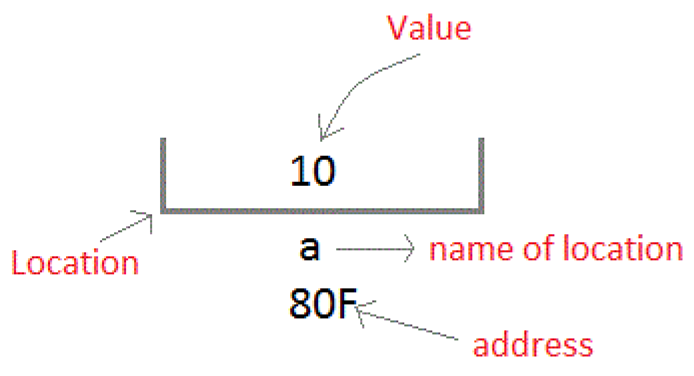
This is used to store the user inputted value to the address of the variable var.

Concept of Pointers

Whenever a **variable** is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value. This location has its own address number, which we just saw above.

Let us assume that system has allocated memory location 80F for a variable a.

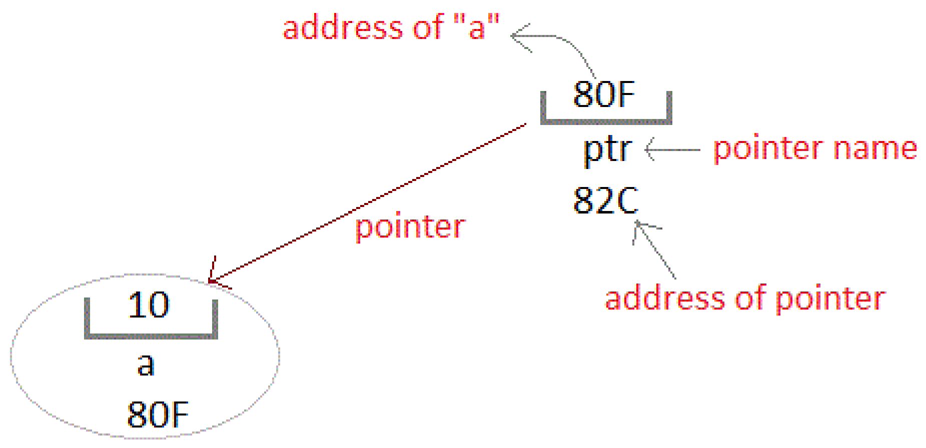
int a = 10;



We can access the value 10 either by using the variable name a or by using its address 80F.

The question is how we can access a variable using it's address? Since the memory addresses are also just numbers, they can also be assigned to some other variable. The variables which are used to hold memory addresses are called **Pointer variables**.

A **pointer** variable is therefore nothing but a variable which holds an address of some other variable. And the value of a **pointer variable** gets stored in another memory location.



Benefits of using pointers

Below we have listed a few benefits of using pointers:

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
3. It reduces length of the program and its execution time as well.
4. It allows C language to support Dynamic Memory management.

In the next tutorial we will learn syntax of pointers, how to declare and define a pointer, and using a pointer. See you in the next tutorial.

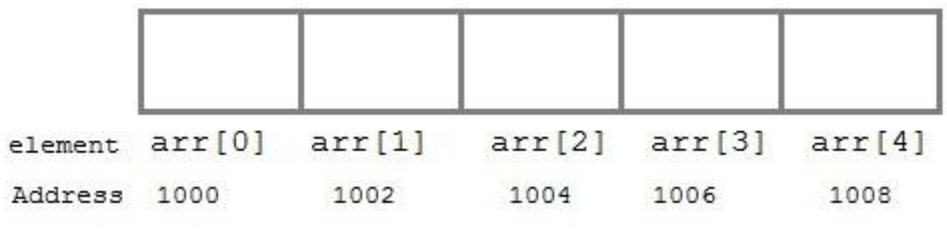
Pointer and Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array arr,

int arr[5] = { 1, 2, 3, 4, 5 };

Assuming that the base address of arr is 1000 and each integer requires two bytes, the five elements will be stored as follows:



Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0]. Hence arr contains the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] by default

We can also declare a pointer of type int to point to the array arr.

```
int *p;
p = arr;
// or,
p = &arr[0];    //both the statements are equivalent.
```

Now we can access every element of the array arr using p++ to move from one element to another.

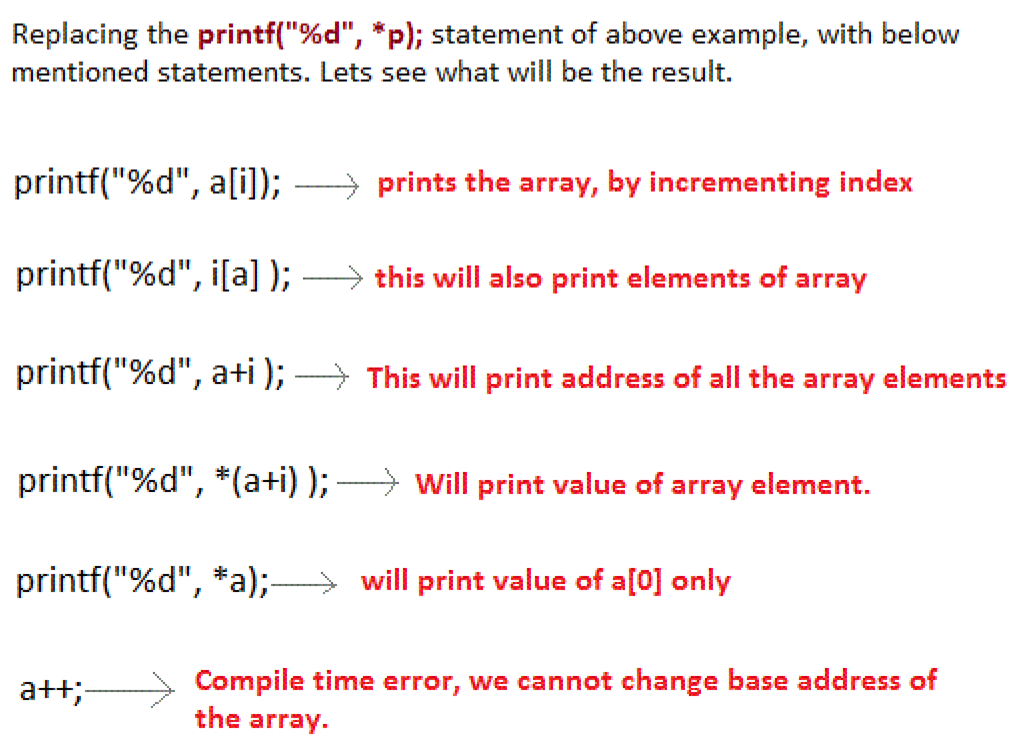
NOTE: You cannot decrement a pointer once incremented. p-- won't work.

Pointer to Array

As studied above, we can use a pointer to point to an array, and then we can use that pointer to access the array elements. Lets have an example,

```
#include <stdio.h>


int main()
{
   int i;
   int a[5] = {1, 2, 3, 4, 5};
   int *p = a;     // same as int*p = &a[0]
   for (i = 0; i < 5; i++)
   {
      printf("%d", *p);
      p++;
   }

   return 0;
}
```

In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as a pointer and print all the values.



Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); → **prints the array, by incrementing index**

printf("%d", i[a]); → **this will also print elements of array**

printf("%d", a+i); → **This will print address of all the array elements**

printf("%d", *(a+i)); → **Will print value of array element.**

printf("%d", *a); → **will print value of a[0] only**

a++; → **Compile time error, we cannot change base address of the array.**

The generalized form for using pointer with an array,

*(a+i)

is same as:

a[i]

Pointer to Multidimensional Array

A multidimensional array is of form, a[i][j]. Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In a[i][j], a will give the base address of this array, even a + 0 + 0 will also give the base address, that is the address of a[0][0]element.

Here is the generalized form for using pointer with multidimensional arrays.

```
*(*(a + i) + j)
```

which is same as,

```
a[i][j]
```

Pointer and Character strings

Pointer can also be used to create strings. Pointer variables of char type are treated as string.

```
char *str = "Hello";
```

The above code creates a string and stores its address in the pointer variable str. The pointer strnow points to the first character of the string "Hello". Another important thing to note here is that the string created using char pointer can be assigned a value at **runtime**.

```
char *str;
str = "hello";      //this is Legal
```

The content of the string can be printed using printf() and puts().

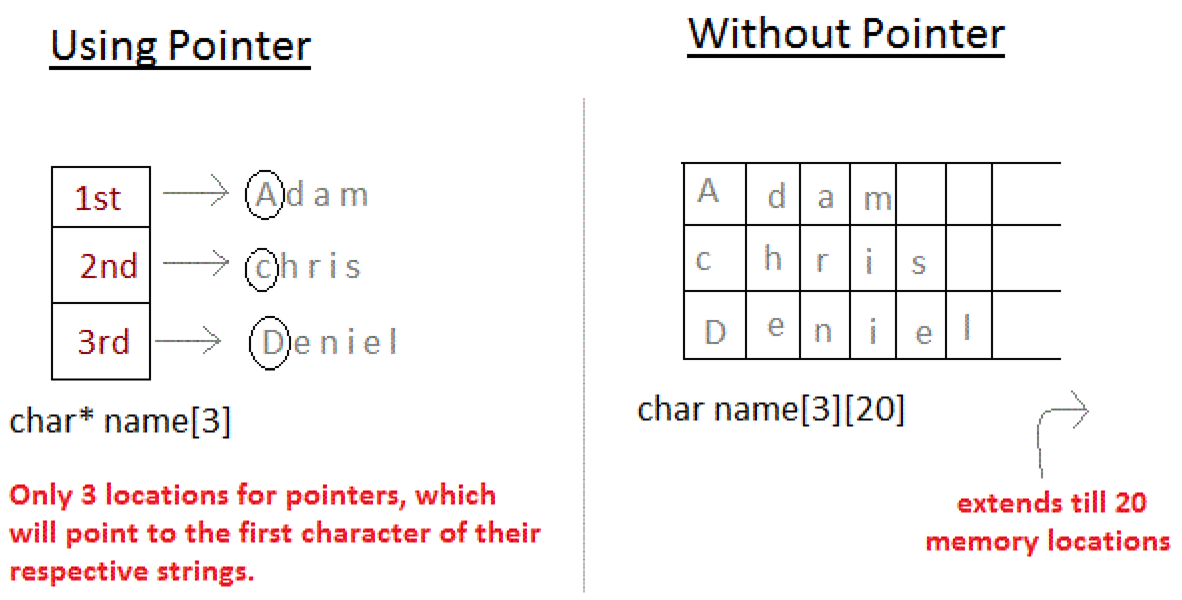
```
printf("%s", str);
puts(str);
```

Notice that str is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator *.

Array of Pointers

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

```
char *name[3] = {
    "Adam",
    "chris",
    "Deniel"
};
//Now lets see same array without using pointer
char name[3][20] = {
   "Adam",
   "chris",
   "Deniel"
};
```


Using Pointer
1st
2nd
3rd
Adam
chris
Deniel
char* name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
Without Pointer
A d a m
c h r i s
D e n i e l
char name[3][20]
extends till 20 memory locations

In the second approach memory wastage is more, hence it is prefered to use pointer in such cases.

When we say memory wastage, it doesn't means that the strings will start occupying less space, no, characters will take the same space, but when we define array of characters, a contiguos memory space is located equal to the maximum size of the array, which is a wastage, which can be avoided if we use pointers instead.

Pointers as Function Argument

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as **call by reference**. When a function is called by reference any change made to the reference variable will effect the original variable.

Example Time: Swapping two numbers using Pointer

```
#include <stdio.h>

void swap(int *a, int *b);

int main()
{
    int m = 10, n = 20;
    printf("m = %d\n", m);
    printf("n = %d\n\n", n);

    swap(&m, &n);    //passing address of m and n to the swap function
    printf("After Swapping:\n\n");
    printf("m = %d\n", m);
    printf("n = %d", n);
    return 0;
}

/*
    pointer 'a' and 'b' holds and
    points to the address of 'm' and 'n'
*/
```

```
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

```
m = 10
n = 20
After Swapping:
m = 20
n = 10
```

Functions returning Pointer variables

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

```
#include <stdio.h>

int* larger(int*, int*);

void main()
{
    int a = 15;
    int b = 92;
    int *p;
    p = larger(&a, &b);
    printf("%d is larger",*p);
}

int* larger(int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}
```

```
92 is larger
```

Safe ways to return a valid Pointer.

1. Either use **argument with functions**. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function as well.

2. Or, use static **local variables** inside the function and return them. As static variables have a lifetime until the main() function exits, therefore they will be available througout the program.

Pointer to functions

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

type (*pointer-name)(parameter);

Here is an example :

int (*sum)();   //legal declaration of pointer to function

int *sum();   //This is not a declaration of pointer to function

A function pointer can point to a specific function when it is assigned the name of that function.

int sum(int, int);

int (*s)(int, int);

s = sum;

Here s is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

s (10, 20);

Example of Pointer to Function

```
#include <stdio.h>

int sum(int x, int y)
{
   return x+y;
}

int main( )
{
   int (*fp)(int, int);
   fp = sum;
```

```
    int s = fp(10, 15);
    printf("Sum is %d", s);

    return 0;
}
```

```
25
```