KAMARAJ COLLEGE

SELF FINANCING COURSES
(Reaccredited with "A+" Grade by NAAC)
(Affiliated to Manonmaniam Sundaranar University, Tirunelveli.)
THOOTHUKUDI – 628003.

STUDY MATERIAL FOR BCA

PROGRAMMING IN C

SEMESTER – I

ACADEMIC YEAR 2022-2023

PREPARED BY

COMPUTER SCIENCE DEPARTMENT

Table of Content

UNIT-I

C Language Basic Syntax Rules

The rule specify how the character sequence will be grouped together, to form **tokens**. A smallest individual unit in C program is known as **C Token**. Tokens are either keywords, identifiers, constants, variables or any symbol which has some meaning in C language. A C program can also be called as a collection of various tokens.

In the following program,

```
#include
int main()
{
   printf("Hello,World");
   return 0;
}
```

if we take any one statement:

```
printf("Hello,World");
```

Then the tokens in this statement are→ printf, (, "Hello, World",) and ;.

So, C tokens are basically the building blocks of a C program.

Semicolon ;

Semicolon ; is used to mark the end of a statement and beginning of another statement. Absence of semicolon at the end of any statement, will mislead the compiler to think that this statement is not yet finished and it will add the next consecutive statement after it, which may lead to compilation(syntax) error.

```
#include
int main()
{
   printf("Hello,World")
   return 0;
}
```

In the above program, we have omitted the semicolon from the printf("...") statement, hence the compiler will think that starting from printf up till the semicolon after return 0 statement, is a single statement and this will lead to compilation error.

Comments

Comments are plain simple text in a C program that are not compiled by the compiler. We write comments for better understanding of the program. Though writing comments is not compulsory, but it is recommended to make your program more descriptive. It makes the code more readable.

There are two ways in which we can write comments.

1. Using // This is used to write a single line comment.
2. Using /* */: The statements enclosed within /* and */, are used to write multi-line comments.

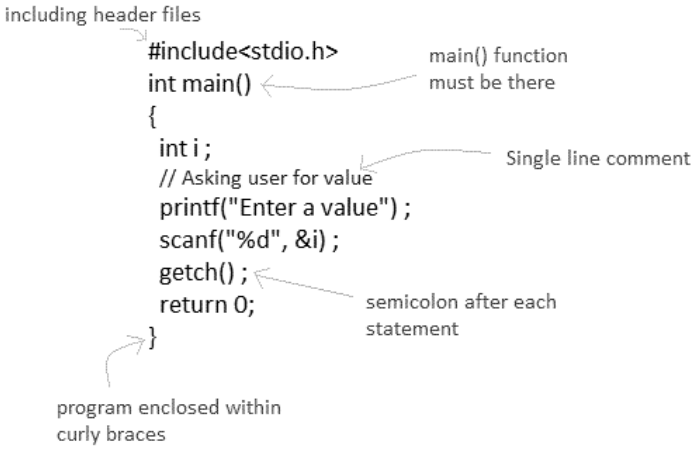
Example of comments:

```
// This is a comment

/* This is a comment */

/* This is a long
and valid comment */

// this is not
  a valid comment
```

Some basic syntax rule for C program

C is a case sensitive language so all C instructions must be written in lower case letter.

- All C statement must end with a semicolon.
- Whitespace is used in C to describe blanks and tabs.
- Whitespace is required between keywords and identifiers. We will learn about keywords and identifiers in the next tutorial.



What are Keywords in C?

Keywords are preserved words that have special meaning in C language. These meaning cannot be changed. Thus, keywords cannot be used as variable names because that would try to change the existing meaning of the keyword, which is not allowed. There are total 32 keywords in C language.

auto	Double	int	struct
break	Else	long	switch
case	Enum	register	typedef
const	Extern	return	union
char	Float	short	unsigned
continue	For	signed	volatile
default	Goto	sizeof	void
do	If	static	while

What are Identifiers?

In C language identifiers are the names given to variables, constants, functions and user-define data. These identifiers are defined against a set of rules.

Rules for an Identifier

1. An Identifier can only have alphanumeric characters (a-z, A-Z, 0-9) and underscore (_).
2. The first character of an identifier can only contain alphabet (a-z, A-Z) or underscore (_).
3. Identifiers are also case sensitive in C. For example, **name** and **Name** are two different identifiers in C.
4. Keywords are not allowed to be used as Identifiers.
5. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

When we declare a variable or any function in C language program, to use it we must provide a name to it, which identified it throughout the program, for example:

int myvariable = "Studytonight";

Here myvariable is the name or identifier for the variable which stores the value "Studytonight" in it.

Character set

In C language characters are grouped into the following categories,

1. Letters (all alphabets a to z & A to Z).
2. Digits (all digits 0 to 9).
3. Special characters, (such as colon :, semicolon ;, period ., underscore _, ampersand & etc).
4. White spaces.

Operators in C Language

C language supports a rich set of built-in operators. An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation. Operators are used in programs to manipulate data and variables.

C operators can be classified into following types:

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators
- Conditional operators
- Special operators

Arithmetic operators

C supports all the basic arithmetic operators. The following table shows all the basic arithmetic operators.

Operator	Description
+	adds two operands
-	subtract second operands from first
*	multiply two operand
/	divide numerator by denominator

%	remainder of division
++	Increment operator - increases integer value by one
--	Decrement operator - decreases integer value by one

Relational operators

The following table shows all relation operators supported by C.

Operator	Description
==	Check if two operand are equal
!=	Check if two operand are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

Logical operators

C language supports following 3 logical operators. Suppose a = 1 and b = 0,

Operator	Description	Example
&&	Logical AND	(a && b) is false
||	Logical OR	(a || b) is true

!	Logical NOT	(!a) is false

Bitwise operators

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to float or double (These are datatypes, we will learn about them in the next tutorial).

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift

Now let's see truth table for bitwise &, | and ^

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The bitwise **shift** operator, shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value have to be shifted. Both operands have the same precedence.

Example:

a = 0001000

b = 2

a << b = 0100000

a >> b = 0000010

Assignment Operators

Assignment operators supported by C language are as follows.

Operator	Description	Example
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	multiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

Conditional operator

The conditional operators in C language are known by two more names

1. **Ternary Operator**
2. **?: Operator**

It is actually the if condition that we use in C language decision making, but using conditional operator, we turn the if condition statement into a short and simple operator.

The syntax of a conditional operator is:

expression 1? expression 2: expression 3

Explanation:

- The question mark **"?"** in the syntax represents the **if** part.
- The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)
- If (expression 1) returns true then the expression on the left side of **" : "** i.e. (expression 2) is executed.
- If (expression 1) returns false then the expression on the right side of **" : "** i.e. (expression 3) is executed.

Special operator

Operator	Description	Example
sizeof	Returns the size of a variable	**sizeof(x)** return size of the variable **x**
&	Returns the address of a variable	**&x ;** return address of the variable **x**
*	Pointer to a variable	***x ;** will be pointer to a variable **x**

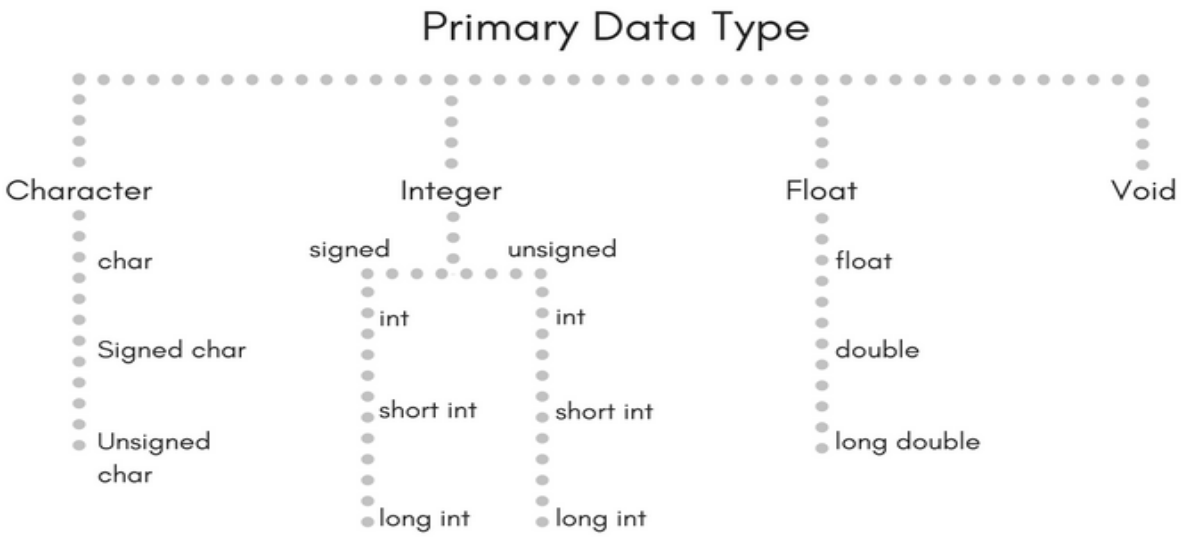
Data types in C Language

Data types specify how we enter data into our programs and what type of data we enter. These data types have different storage capacities.

C language supports 2 different type of data types:

1. **Primary data types**: These are fundamental data types in C namely integer(int), floating point(float), character(char) and void.
2. **Derived data types**: Derived data types are nothing but primary datatypes but a little twisted or grouped together like **array**, **structure**, **union** and **pointer**.

Data type determines the type of data a variable will hold. If a variable x is declared as int. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is.



Integer type

Integers are used to store whole numbers.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range
int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type

Floating types are used to store real numbers.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

Void Type

Void type means no value. This is usually used to specify the type of functions which returns nothing. We will get acquainted to this datatype as we start learning more advanced topics in C language, like functions, pointers etc.

Variables in C Language

The naming of an address is known as **variable**. Variable is the name of memory location. Unlike constant, variables are changeable, we can change value of a variable during execution of a program. A programmer can choose a meaningful variable name. Example: total, height, age, etc.

Datatype of Variable

A **variable** in C language must be given a type, which defines what type of data the variable will hold. It can be:

- char: Can hold/store a character in it.
- int: Used to hold an integer.
- float: Used to hold a float value.
- double: Used to hold a double value.
- void

Rules to name a Variable

1. Variable name must not start with a digit.
2. Variable name can consist of alphabets, digits and special symbols like underscore _.
3. Blank or spaces are not allowed in variable name.
4. Keywords are not allowed as variable name.
5. Upper- and lower-case names are treated as different, as C is case-sensitive, so it is suggested to keep the variable names in lower case.

Declaring, Defining and Initializing a variable

Declaration of variables must be done before they are used in the program. Declaration does the following things.

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. Until the variable is defined the compiler doesn't have to worry about allocating memory space to the variable.
4. Declaration is more like informing the compiler that there exists a variable with following datatype which is used in the program.

int a;

float b, c;

Initializing a variable means to provide it with a value. A variable can be initialized and defined in a single statement, like:

int a = 10;

Let's write a program in which we will use some variables.

```
#include <stdio.h>

int main ()

{

   /* variable definition: */

   int a, b;

   /* actual initialization */

   a = 7;
```

```
    b = 14;
    /* using addition operator */
    c = a + b;
    /* display the result */
    printf("Sum is : %d \n", c);
        return 0;
}
```

Sum is : 21

Difference between Variable and Identifier?

An Identifier is a name given to any variable, function, structure, pointer or any other entity in a programming language. While a variable, as we have just learned in this tutorial is a named memory location to store data which is used in the program.

Identifier	Variable
Identifier is the name given to a variable, function etc.	While, variable is used to name a memory location which stores data.
An identifier can be a variable, but not all identifiers are variables.	All variable names are identifiers.

Example:
// a variable
int studytonight;
// or, a function
int studytonight() {
..
}
	Example:
	// int variable
	int a;
	// float variable
	float a;

C Input and Output

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

scanf() and printf() functions

The standard input-output header file, named stdio.h contains the definition of the functions printf() and scanf(), which are used to display output on screen and to take input from user respectively.

```
#include<stdio.h>
void main()
{
    int i;
   printf("Please enter a value...");
   scanf("%d", &i);
   printf( "\nYou entered: %d", i);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered on screen.

You must be wondering what is the purpose of %d inside the scanf() or printf()functions. It is known as **format string** and this informs the scanf() function, what type of input to expect and in printf() it is used to give a heads up to the compiler, what type of output to expect.

Format String	Meaning
%d	Scan or print an integer as signed decimal number
%f	Scan or print a floating-point number
%c	To scan or print a character
%s	To scan or print a character string. The scanning ends at whitespace.

We can also **limit the number of digits or characters** that can be input or output, by adding a number with the format string specifier, like "%1d" or "%3s", the first one means a single numeric digit and the second one means 3 characters, hence if you try to input 42, while scanf() has "%1d", it will take only 4 as input. Same is the case for output.

getchar() & putchar() functions

The getchar() function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. The putchar() function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time. In case you want to display more than one characters, use putchar() method in a loop.

```
#include <stdio.h>
void main( )
{
   int c;
   printf("Enter a character");
   c = getchar();
   putchar(c);
}
```

When you will compile the above code, it will ask you to enter a value. When you will enter the value, it will display the value you have entered.

gets() & puts() functions

The gets() function reads a line from stdin(standard input) . The puts()function writes the to stdout.

```
#include<stdio.h>

void main() {

   char str[100];

   printf("Enter a string");

   gets( str );

   puts(str );

   getch();

}
```

When you will compile the above code, it will ask you to enter a string. When you will enter the string, it will display the value you have entered.

Difference between scanf() and gets()

The main difference between these two functions is that scanf() stops reading characters when it encounters a space, but gets() reads space as character too.

If you enter name as **Study Tonight** using scanf() it will only read and store **Study** and will leave the part after space. But gets() function will read it completely.

UNIT-II

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:

```
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");
   }
}
```

Switch statement in C

When you want to solve multiple option type problems, switch statement 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;
          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.
5. break statements are used to **exit** the switch block.
6. We don't use those expressions to evaluate switch case, which may return floating point values or strings or characters.

Example **of switch** statement

```
#include<stdio.h>
void main( )
{
   int a, b, c, choice;
   while(choice != 3)
   {
      printf("\n 1. Press 1 for addition");
      printf("\n 2. Press 2 for subtraction");
      printf("\n Enter your choice");
      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

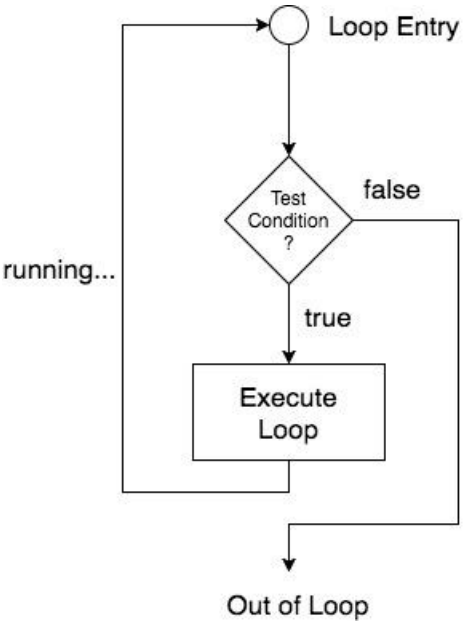
- if statements can evaluate float conditions. switch statements cannot evaluate float conditions.
- if statement can evaluate relational operators. switch statement cannot evaluate relational operators i.e. they are not allowed in switch statement.

How to use Loops in C

In any programming language including 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:

- while loop
- for loop
- do while loop

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

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.**

Syntax:

```
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

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);
}
```

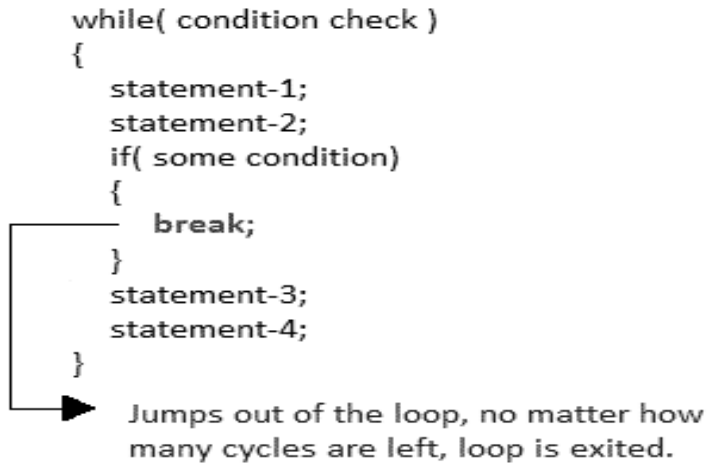
5 10 15 20 25 30 35 40 45 50

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.

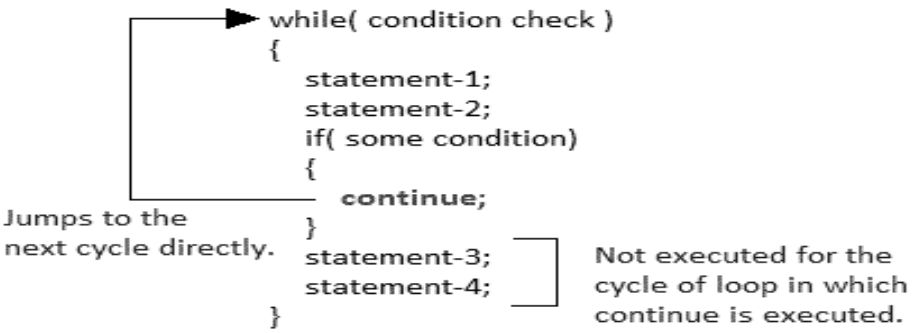
1) break statement

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



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.



UNIT-III

Arrays in C

In C language, arrays are referred to as structured data types. An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations. (collection of elements of same data type)

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,

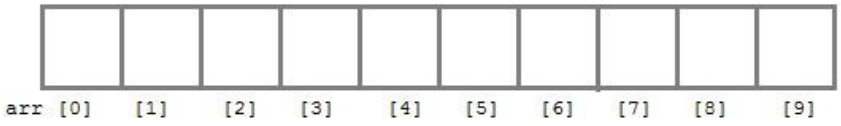
- to store list of Employee or Student names,
- to store marks of students,
- or to store list of numbers or characters etc.
-

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++)
   {
```

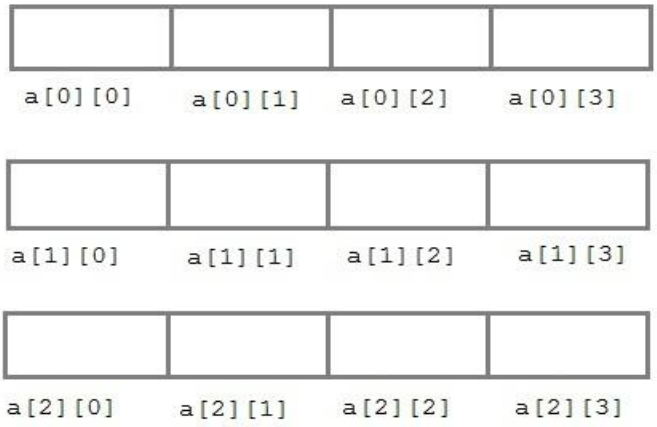
```
        printf("%d\n", arr[j]);
    }
}
```

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]);
       }
    }
}
```

Multi-dimensional Array

In C, we can define multidimensional arrays in simple words as array of arrays. Data in multidimensional arrays are stored in tabular form (in row major order).
General form of declaring N-dimensional arrays:

```
data_type  array_name[size1][size2]....[sizeN];
```

data_type: Type of data to be stored in the array.

array_name: Name of the array

size1, size2,... ,sizeN: Sizes of the dimensions

Examples:
Two-dimensional array:
int two_d[10][20];

Three-dimensional array:
int three_d[10][20][30];

Size of multidimensional arrays

Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
For example:
The array **int x[10][20]** can store total (10*20) = 200 elements.
Similarly array **int x[5][10][20]** can store total (5*10*20) = 1000 elements.

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] = "Study Tonight";       // 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

1) strcat() function

```
strcat("hello", "world");
```

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

2)strlen() function

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

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

12

3) 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

4) 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

5) 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

Other String Functions

1) strncpy

- strncpy( ) function copies portion of contents of one string into another string.
- Example:
  strncpy (str1, str2, n) – It copies first n characters of str2 into str1.

  If destination string length is less than source string, entire source string value won't be copied into destination string.

EXAMPLE PROGRAM FOR strncpy( ) FUNCTION IN C:

In this program, only 5 characters from source string "fresh2refresh" is copied into target string using strncpy( ) function.

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

int main( )
{
  char source[ ] = "fresh2refresh" ;
  char target[20]= "" ;
  printf ( "\nsource string = %s", source ) ;
  printf ( "\ntarget string = %s", target ) ;
  strncpy ( target, source, 5 ) ;
  printf ( "\ntarget string after strcpy( ) = %s", target ) ;
  return 0;
}
```

OUTPUT:

```
source string = fresh2refresh
target string =
target string after strncpy( ) = fresh
```

2) strncat() function

- strncat( ) function concatenates (appends) portion of one string at the end of another string.
  Example:
  strncat (str1, str2, n); – First n characters of str2 is concatenated at the end of str1.
- As you know, each string in C is ended up with null character ('\0').
  In strncat( ) operation, null character of destination string is overwritten by source string's first character and null character is added at the end of new destination string which is created after strncat( ) operation.

EXAMPLE PROGRAM FOR strncat( ) FUNCTION IN C:

In this program, first 5 characters of the string "fresh2refresh" is concatenated at the end of the string "C tutorial" using strncat( ) function and result is displayed as "C tutorial fresh".

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

int main( )
{
  char source[ ] = "fresh2refresh" ;
  char target[ ]= "C tutorial" ;

  printf ( "\nSource string = %s", source ) ;
  printf ( "\nTarget string = %s", target ) ;

  strncat ( target, source, 5 ) ;

  printf ( "\nTarget string after strncat( ) = %s", target ) ;
}
```

OUTPUT:

```
Source string = fresh2refresh
Target string = C tutorial
Target string after strcat( ) = C tutorialfresh
```

3)strncmp() function

This function compares only the first n (specified number of) characters of strings and returns following value based on the comparison.

Example:
strncmp (str1, str2, n);

- 0, if both the strings str1 and str2 are equal
- negative number , if str1 is less than str2
- positive number, if str1 is greater than str2

EXAMPLE PROGRAM FOR strncmp( ) FUNCTION IN C:

```
/* C strncmp Function example  */
#include <stdio.h>
#include<string.h>
int main()
{
char str1[50] = "abcdef";
char str2[50] = "abcd";
```

```
char str3[] =  "ghi";
int i, j, k;

i = strncmp(str1, str2, 4);
printf("\n The Comparison of str1 and str2 Strings = %d", i);

j = strncmp(str1, str2, 6);
printf("\n The Comparison of str1 and str2 Strings = %d", j);

k = strncmp(str1, str3, 3);
printf("\n The Comparison of str1 and str3 = %d", k);
}
```

OUTPUT:

```
The Comparison of str1 and str2 Strings = 0
The Comparison of str1 and str2 Strings = 1
The Comparison of str1 and str3 = -1
```

3)strstr() function

It can be used to locate a substring in a string.

Example:
strstr (str1, str2);
searches str2 is contained in str1.

- if yes, returns the position of the first occurrence of str2.
- if no, returns null pointer.

EXAMPLE PROGRAM FOR strstr( ) FUNCTION IN C:

```
#include <stdio.h>
#include<string.h>
int main()
{
char str1[30] = "Learning C is awesome";
char str2 [15] = "C";
char *st;
st = strstr(str1, str2);
printf("%s", st);
return 0;
}
```

OUTPUT:

```
C is awesome
```

UNIT-IV

Need For User Defined 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 easier to understand.

Function definition

A function definition ,also known as function implementation shall include the following elements

1. function name
2. function type
3. list of parameters

Syntax

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.

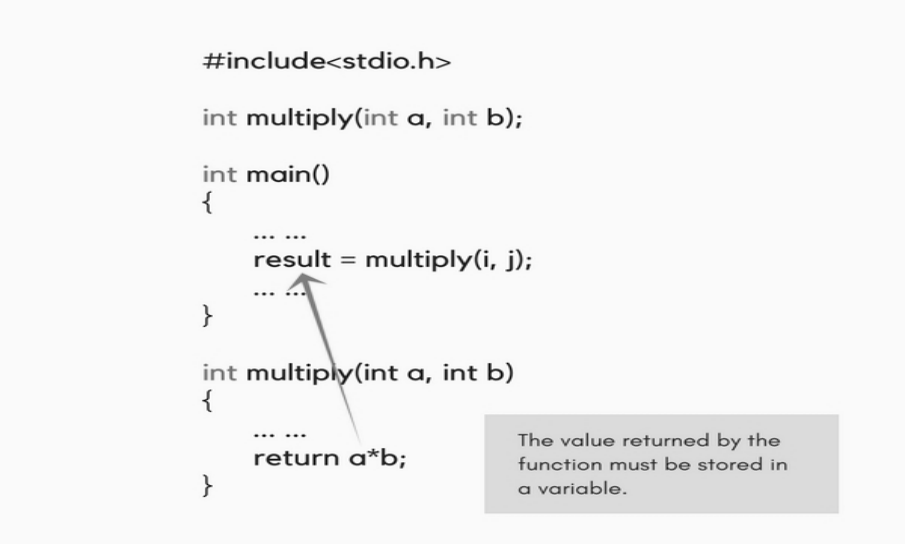
function body

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).

Returning a value from function

A function may or may not return a result. But if it does, we must use the return statement to output the result. return statement also ends the function execution, hence it must be the last statement of any function. If you write any statement after the return statement, it won't be executed.



The datatype of the value returned using the return statement should be same as the return type mentioned at function declaration and definition. If any of it mismatches, you will get compilation error.

Calling a function

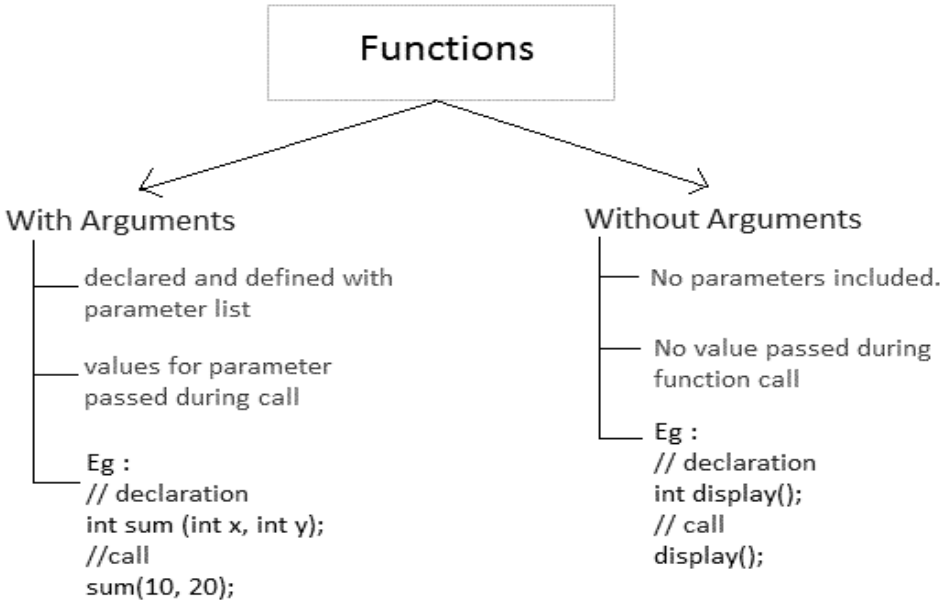
When a function is called, control of the program gets transferred to the function.

functionName(argument1, argument2,...);

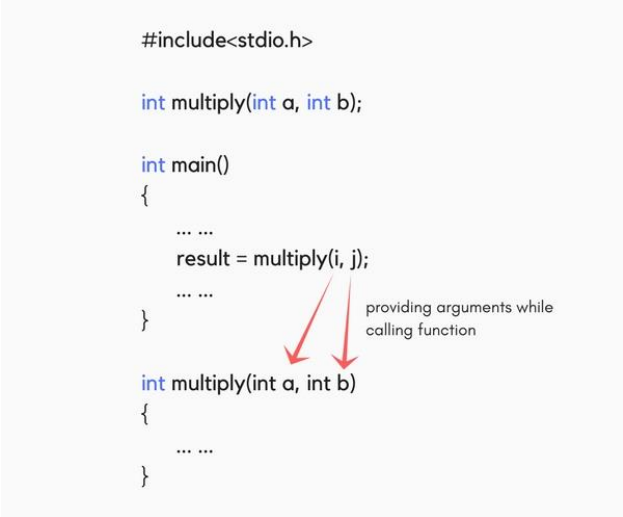
In the example above, the statement multiply(i, j); inside the main() function is function call.

Passing Arguments to a function

Arguments are the values specified during the function call, for which the formal parameters are declared while defining the function.



It is possible to have a function with parameters but no return type. It is not necessary, that if a function accepts parameter(s), it must return a result too.



While declaring the function, we have declared two parameters a and b of type int. Therefore, while calling that function, we need to pass two arguments, else we will get compilation error. And the two arguments passed should be received in the function definition, which means that the function header in the function definition should have the two parameters to hold the argument values. These received arguments are also known as **formal parameters**. The name of the variables while declaring, calling and defining a function can be different.

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 accepted, 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.

- return type
- 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**.

Time for an 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 multiplication is: %d", result);
   return 0;
}
int multiply(int a, int b)
{
   return (a*b);       // function definition, this can be done in one line
}
```

Type of User-defined Functions in C

There can be 4 different types of user-defined functions, they are:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

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);
}
}
```

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;
}
```

Function with arguments and no return value

This time, we have modified the above example to make the function greatNum() take two int values 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);
    }
}
```

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;
}

}
```

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;
}
```

How to pass Array to a Function in C

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

1. We can either have an array as a parameter.
   int sum (int arr[]);

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

   int sum (int* ptr);.

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 += marks[i];
   }
   avg = (sum / 5);
   return avg; }
```

94.6

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

Passing Strings to Function

Strings can be passed to a function in a similar way as arrays.

Example:

```
#include <stdio.h>

void displayString(char str[]);

int main()

{    char str[50];

printf("Enter string: ");

gets(str);

displayString(str);     // Passing string to a function.

return 0;

}void displayString(char str[])

{    printf("String Output: ");

puts(str);  }
```

Types of Function calls in C

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

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.

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);      // function 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
{
       *p = *p + 10;
}
```

value of x is 20

The Scope, Visibility and Lifetime of Variables

In C language, each variable has a storage class which decides the following things:

- **scope** i.e where the value of the variable would be available inside a program.
- **default initial value** i.e if we do not explicitly initialize that variable, what will be its default initial value.
- **lifetime** of that variable i.e for how long will that variable exist.

The following storage classes are most oftenly used in C programming,

1. **Automatic variables**
2. **External variables**
3. **Static variables**
4. **Register variables**

1) Automatic variables: auto

Scope: Variable defined with auto storage class are local to the function block inside which they are defined.

Default Initial Value: Any random value i.e garbage value.

Lifetime: Till the end of the function/method block where the variable is defined.

A variable declared inside a function without any storage class specification, is by default an **automatic variable**. They are created when a function is called and are destroyed **automatically** when the function's execution is completed. Automatic variables can also be called **local variables** because they are local to a function. By default, they are assigned **garbage value** by the compiler.

```
#include<stdio.h>

void main(){

   int detail;
   // or
   auto int details;    //Both are same
}
```

2) External or Global variable

Scope: Global i.e everywhere in the program. These variables are not bound by any function, they are available everywhere.

Default initial value: 0(zero).

Lifetime: Till the program doesn't finish its execution, you can access global variables.

A variable that is declared outside any function is a **Global Variable**. Global variables remain available throughout the program execution. By default, initial value of the Global variable is 0(zero).

One important thing to remember about global variable is that their values can be changed by any function in the program.

```
#include<stdio.h>
int number;     // global variable
void main()
{
   number = 10;
   printf("I am in main function. My value is %d\n", number);
   fun1();     //function calling, discussed in next topic
   fun2();     //function calling, discussed in next topic
}
/* This is function 1 */
fun1()
{
   number = 20;
   printf("I am in function fun1. My value is %d", number);
}
/* This is function 1 */
fun2()
{
   printf("\nI am in function fun2. My value is %d", number);
}
```

I am in function main. My value is 10

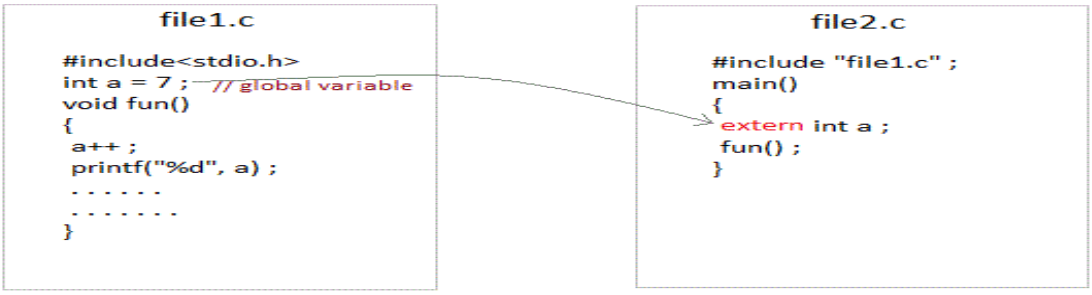
I am in function fun1. My value is 20

I am in function fun2. My value is 20

Here the global variable number is available to all three functions and thus, if one function changes the value of the variable, it gets changed in every function.

Note: Declaring the storage class as global or external for all the variables in a program can waste a lot of memory space because these variables have a lifetime till the end of the program. Thus, variables, which are not needed till the end of the program, will still occupy the memory and thus, memory will be wasted.

extern keyword : The extern keyword is used with a variable to inform the compiler that this variable is declared somewhere else. The extern declaration does not allocate storage for variables**.**



global variable from one file can be used in other using extern keyword.

Problem when extern is not used

```
int main()
{
   a = 10;     //Error: cannot find definition of variable 'a'
   printf("%d", a);
}
```

Example using extern in same file

```
int main()
{
   extern int x;   //informs the compiler that it is defined somewhere else
   x = 10;
   printf("%d", x);
}
int x;      //Global variable x
```

3) Static variables

Scope: Local to the block in which the variable is defined

Default initial value: 0(Zero).

Lifetime: Till the whole program doesn't finish its execution.

A static variable tells the compiler to persist/save the variable until the end of program. Instead of creating and destroying a variable every time when it comes into and goes out of scope, static variable is initialized only once and remains into existence till the end of the program. A static variable can either be internal or external depending upon the place of declaration. Scope of **internal static** variable remains inside the function in which it is defined. **External static** variables remain restricted to scope of file in which they are declared.

They are assigned **0 (zero)** as default value by the compiler.

```
#include<stdio.h>
void test();    //Function declaration (discussed in next topic)
int main()
{
   test();
   test();
   test();
}
void test()
{
   static int a = 0;       //a static variable
   a = a + 1;
   printf("%d\t",a);
}
```

1 2 3

4) Register variable

Scope: Local to the function in which it is declared.

Default initial value: Any random value i.e garbage value

Lifetime: Till the end of function/method block, in which the variable is defined.

Register variables inform the compiler to store the variable in CPU register instead of memory. Register variables have faster accessibility than a normal variable. Generally, the frequently used variables are kept in registers. But only a few variables can be placed inside registers. One application of register storage class can be in using loops, where the variable gets used a number of times in the program, in a very short span of time.

NOTE: We can never get the address of such variables.

Syntax :

register int number;

Note: Even though we have declared the storage class of our variable number as register, we cannot surely say that the value of the variable would be stored in a register. This is because the number of registers in a CPU are limited. Also, CPU registers are meant to do a lot of important work. Thus, sometimes they may not be free. In such scenario, the variable works as if its storage class is auto.

Which storage class should be used and when

To improve the speed of execution of the program, following points should be kept in mind while using storage classes:

- static - when we want the value of the variable to remain same every time.
- register - variables that are used in our program very oftenly.
- external - variables that are being used by almost all the functions in the program.
- If we do not have the purpose of any of the above-mentioned storage classes, then we should use the automatic storage class.

C Structures

Structure is a user-defined data type in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

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, how can I use arrays for this problem, I will require something which can hold data of different types together.

In structure, data is stored in form of **records**.

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

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 chartype 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, "Viraj");
    /*
       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: Viraj

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];

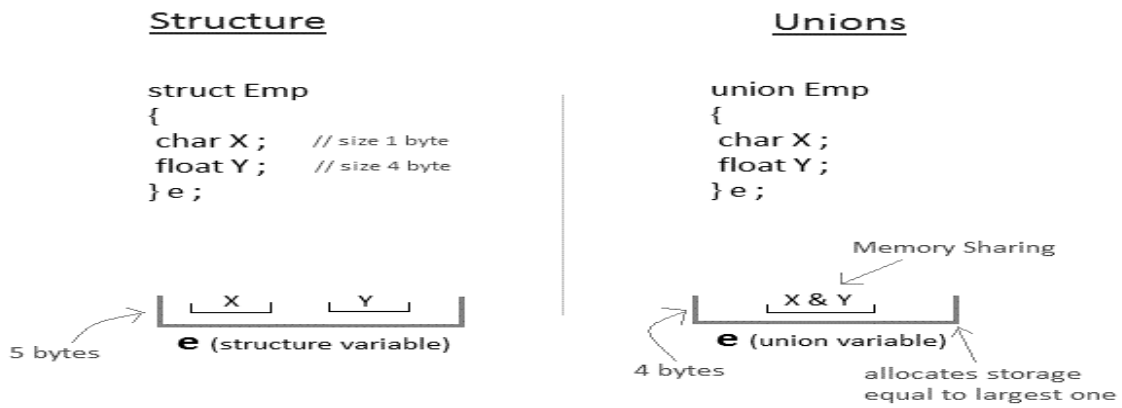
The below program defines an array emp of size 5. Each element of the array emp is of type Employee.

```
#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();
}
```

C Unions

Unions are conceptually similar to **structures**. The syntax to declare/define a union is also similar to that of a structure. The only differences are in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** use 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.

Example

```
#include <stdio.h>
union item
{
   int a;
   float b;
   char ch;                                    -26426

};                                              20.1999
int main( )                                     z

{
   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;
}
```

UNIT-V

INTRODUCTION TO C 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.

Before we start understanding what pointers are and what they can do, let's start by understanding what does "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 its 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 its 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);

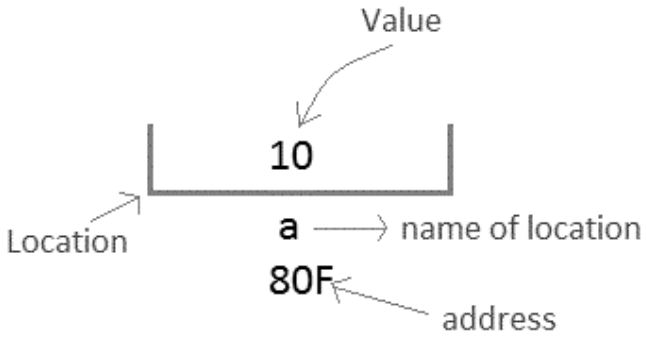
This is used to store the user inputted value to the address of the variable var.

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.

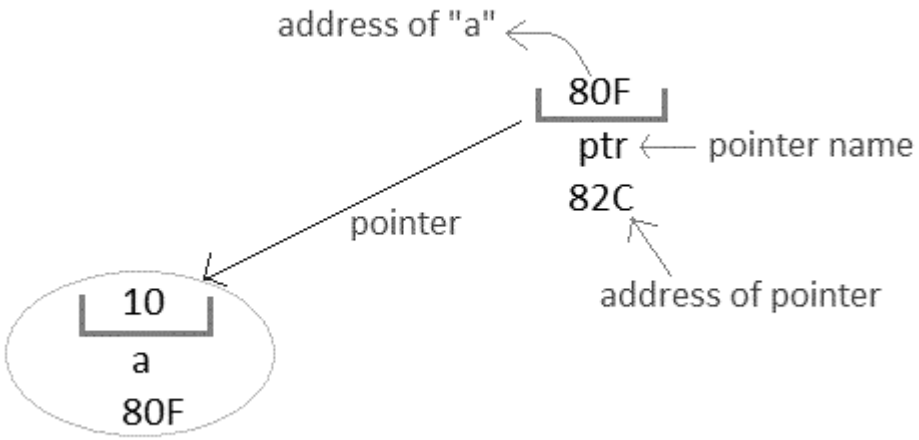
int a = 10;



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 its 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.

Declaring, Initializing and using a pointer variable in C

In this tutorial, we will learn how to declare, initialize and use a pointer. We will also learn what NULL pointer are and where to use them. Let's start!

Declaration of C Pointer variable

General syntax of pointer declaration is,

datatype *pointer_name;

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing. void type pointer works with all data types, but is not often used.

Here are a few examples:

```
int *ip     // pointer to integer variable
float *fp;      // pointer to float variable
double *dp;     // pointer to double variable
char *cp;       // pointer to char variable
```

Initialization of C Pointer variable

Pointer Initialization is the process of assigning address of a variable to a **pointer**variable. Pointer variable can only contain address of a variable of the same data type. In C language **address** operator & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
#include<stdio.h>

void main()
{
   int a = 10;
   int *ptr;       //pointer declaration
   ptr = &a;       //pointer initialization
}
```

Pointer variable always point to variables of same data type. Let's have an example to showcase this:

```
#include<stdio.h>

void main()
{
   float a;
   int *ptr;
   ptr = &a;       // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULL value is called a **NULL pointer**.

```
#include <stdio.h>

int main()
{
   int *ptr = NULL;
   return 0;
}
```

Using the pointer or Dereferencing of Pointer

Once a pointer has been assigned the address of a variable, to access the value of the variable, pointer is **dereferenced**, using the **indirection operator** or **dereferencing operator** *.

```
#include <stdio.h>

int main()
{
   int a, *p;  // declaring the variable and pointer
   a = 10;
   p = &a;     // initializing the pointer

   printf("%d", *p);    //this will print the value of 'a'
   printf("%d", *&a);   //this will also print the value of 'a'
   printf("%u", &a);    //this will print the address of 'a'
   printf("%u", p);     //this will also print the address of 'a'
   printf("%u", &p);    //this will print the address of 'p'
   return 0;
}
```

Points to remember while using pointers

1. While declaring/initializing the pointer variable, * indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand &.
3. The pointer variable stores the address of a variable. The declaration int *adoesn't mean that a is going to contain an integer value. It means that a is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, * is used. Here, the * can be read as **'value at'**.

Time for an Example!

Let's take a simple code example,

```
#include <stdio.h>

int main()
{
      int i = 10;     // normal integer variable storing value 10
      int *a;     // since '*' is used, hence its a pointer variable
```

```
    a = &i;
    printf("Address of variable i is %u\n", a);
    printf("Value at the address, which is stored by pointer variable a is %d\n", *a);
    return 0;
}
```

Address of variable i is 2686728 (The address may vary)

Value at an address, which is stored by pointer variable a is 10

Pointer variable always point to variables of same data type. Let's have an example to showcase this:

```
#include<stdio.h>

void main()
{
   float a;
   int *ptr;
   ptr = &a;       // ERROR, type mismatch
}
```

If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable. A pointer which is assigned a NULL value is called a **NULL pointer**.

```
#include <stdio.h>

int main()
{
   int *ptr = NULL;
   return 0;
}
```

Pointers as Function Argument in C

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 affect 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;
}

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

m = 10

n = 20

After Swapping:

m = 20

n = 10

File Input/output in C

A **file** represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure.

In C language, we use a structure **pointer of file type** to declare a file.

FILE *fp;

C provides a number of functions that helps to perform basic file operations. Following are the functions,

Function	**description**
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file

fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the beginning point

Opening a File or Creating a File

The fopen() function is used to create a new file or to open an existing file.

General Syntax:

*fp = FILE *fopen(const char *filename, const char *mode);

Here, *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened(or created) file.

filename is the name of the file to be opened and **mode** specifies the purpose of opening the file. Mode can be of following types,

mode	**description**
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode

r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

Closing a File

The fclose() function is used to close an already opened file.

General Syntax :

int fclose(FILE *fp);

Here fclose() function closes the file and returns **zero** on success, or **EOF** if there is an error in closing the file. This **EOF** is a constant defined in the header file **stdio.h**.

Input/output operation on File

In the above table we have discussed about various file I/O functions to perform reading and writing on file. **getc() and putc()** are the simplest functions which can be used to read and write individual characters to a file.

```
#include<stdio.h>

int main()
{
   FILE *fp;
```

```
    char ch;
    fp = fopen("one.txt", "w");
    printf("Enter data...");
    while( (ch = getchar()) != EOF) {
       putc(ch, fp);
    }
    fclose(fp);
    fp = fopen("one.txt", "r");
     while( (ch = getc(fp)! = EOF)
    printf("%c",ch);
     // closing the file pointer
    fclose(fp);
     return 0;
}
```

Reading and Writing to File using fprintf() and fscanf()

```
#include<stdio.h>
struct emp
{
   char name[10];
   int age;
};
void main()
{
   struct emp e;
   FILE *p,*q;
   p = fopen("one.txt", "a");
   q = fopen("one.txt", "r");
   printf("Enter Name and Age:");
   scanf("%s %d", e.name, &e.age);
   fprintf(p,"%s %d", e.name, e.age);
   fclose(p);
   do
   {
      fscanf(q,"%s %d", e.name, e.age);
      printf("%s %d", e.name, e.age);
   }
   while(!feof(q));
}
```

In this program, we have created two FILE pointers and both are refering to the same file but in different modes.

fprintf() function directly writes into the file, while fscanf() reads from the file, which can then be printed on the console using standard printf() function.

Difference between Append and Write Mode

Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exist already.

The only difference they have is, when you **open** a file in the **write** mode, the file is reset, resulting in deletion of any data already present in the file. While in **append** mode this will not happen. Append mode is used to append or add data to the existing data of file(if any). Hence, when you open a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.

fseek(), ftell() and rewind() functions

- fseek(): It is used to move the reading control to different positions using fseek function.
- ftell(): It tells the byte location of current position of cursor in file pointer.
- rewind(): It moves the control to beginning of the file.

Error Handling in C

C language does not provide any direct support for error handling. However, a few methods and variables defined in **error.h** header file can be used to point out error using the return statement in a function. In C language, a function returns -1 or NULL value in case of any error and a global variable **errno** is set with the error code. So, the return value can be used to check error while programming.

What is errno?

Whenever a function call is made in C language, a variable named errno is associated with it. It is a global variable, which can be used to identify which type of error was encountered while function execution, based on its value.

errno value	Error
1	Operation not permitted
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error

6	No such device or address
7	Argument list too long
8	Exec format error
9	Bad file number
10	No child processes
11	Try again
12	Out of memory
13	Permission denied

C language uses the following functions to represent error messages associated with **errno**:

- perror(): returns the string passed to it along with the textual representation of the current errno value.
- strerror() is defined in **string.h** library. This method returns a pointer to the string representation of the current errno value.

Time for an Example

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
 int main ()
{
   FILE *fp;
    /*
      If a file, which does not exist, is opened,
      we will get an error
   */
   fp = fopen("IWillReturnError.txt", "r");
    printf("Value of errno: %d\n ", errno);
   printf("The error message is : %s\n", strerror(errno));
   perror("Message from perror");
```

```
    return 0;
}
```

Value of errno: 2

The error message is: No such file or directory

Message from perror: No such file or directory