SCSA1202 PROGRAMMING WITH C AND C++

UNIT1 BASICS OF C PROGRAMMING 9Hrs

Features of C - Structure of C program-Data Types-'C' Tokens-Input/output statements-Control Statement, Functions: – Types of Functions –Recursion.

The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by the following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

➢ C as a mother language

C language is considered as the mother language of all the modern programming languages because most of the compilers, JVMs, Kernels, etc. are written in C language, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

It provides the core concepts like the array, strings, functions, file handling, etc. that are being used in many languages like C++, Java, C#, etc.

➢ C as a system programming language

A system programming language is used to create system software. C language is a system programming language because it **can be used to do low-level programming (for example driver and kernel)**. It is generally used to create hardware devices, OS, drivers, kernels, etc. For example, Linux kernel is written in C.

It can't be used for internet programming like Java, .Net, PHP, etc.

➢ C as a procedural language

A procedure is known as a function, method, routine, subroutine, etc. A procedural language **specifies a series of steps for the program to solve the problem**.

A procedural language breaks the program into functions, data structures, etc.

C is a procedural language. In C, variables and function prototypes must be declared before being used.

- C as a structured programming language

A structured programming language is a subset of the procedural language. **Structure means to break a program into parts or blocks** so that it may be easy to understand.

In the C language, we break the program into parts using functions. It makes the program easier to understand and modify.

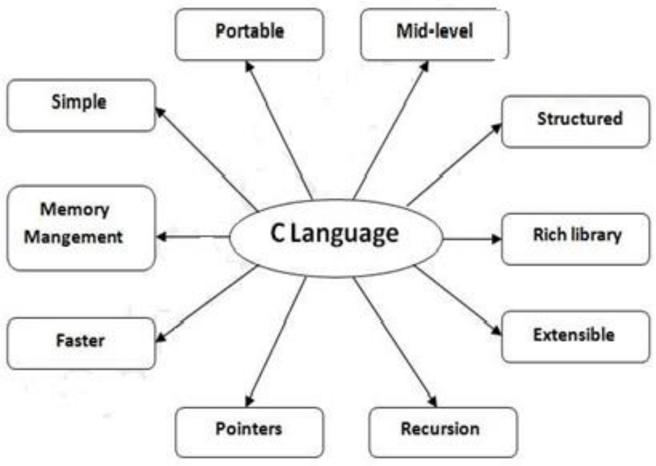
- C as a mid-level programming language

C is considered as a middle-level language because it **supports the feature of both low-level and high-level languages**. C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).

A **Low-level language** is specific to one machine, i.e., machine dependent. It is machine dependent, fast to run. But it is not easy to understand.

A **High-Level language** is not specific to one machine, i.e., machine independent. It is easy to understand.

- **FEATURES OF C LANGUAGE**



C is the widely used language. It provides many **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library

6. Memory Management
7. Fast Speed
8. Pointers
9. Recursion
10. Extensible

1) Simple

C is a simple language in the sense that it provides a **structured approach** (to break the problem into parts), **the rich set of library functions**, **data types**, etc.

2) Machine Independent or Portable

Unlike assembly language, c programs **can be executed on different machines** with some machine specific changes. Therefore, C is a machine independent language.

3) Mid-level programming language

Although, C is **intended to do low-level programming**. It is used to develop system applications such as kernel, driver, etc. It **also supports the features of a high-level language**. That is why it is known as mid-level language.

4) Structured programming language

C is a structured programming language in the sense that **we can break the program into parts using functions**. So, it is easy to understand and modify. Functions also provide code reusability.

5) Rich Library

C **provides a lot of inbuilt functions** that make the development fast.

6) Memory Management

It supports the feature of **dynamic memory allocation**. In C language, we can free the allocated memory at any time by calling the **free()** function.

7) Speed

The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

8) Pointer

C provides the feature of pointers. We can directly interact with the memory by using the pointers. We **can use pointers for memory, structures, functions, array**, etc.

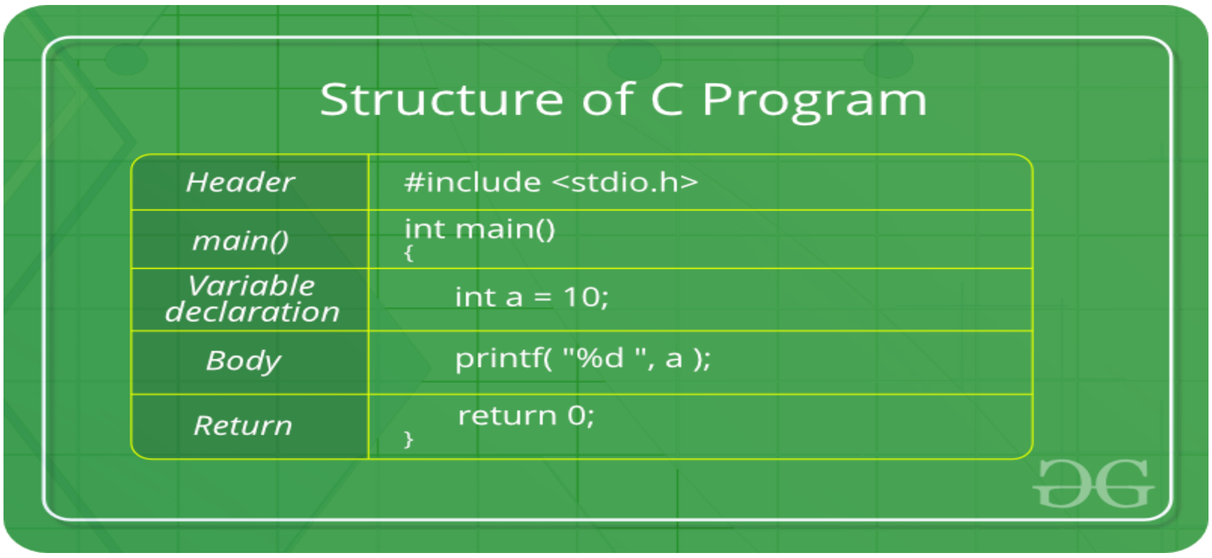
9) Recursion

In C, we **can call the function within the function**. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

10) Extensible

C language is extensible because it **can easily adopt new features**

STRUCTURE OF A C PROGRAM



Structure of C Program

Header	`#include <stdio.h>`

main()	`int main()`
	`{`

Variable declaration	`int a = 10;`
Body	`printf( "%d ", a );`

Return	`return 0;`
	`}`

1. **Header Files Inclusion**: The first and foremost component is the inclusion of the Header files in a C program.
   A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files.
   Some of C Header files:
   - stddef.h – Defines several useful types and macros.
   - stdint.h – Defines exact width integer types.
   - stdio.h – Defines core input and output functions
   - stdlib.h – Defines numeric conversion functions, pseudo-random network generator, memory allocation
   - string.h – Defines string handling functions
   - math.h – Defines common mathematical functions

   Syntax to include a header file in C:

   ```
   #include
   ```

2. **Main Method Declaration:** The next part of a C program is to declare the main() function. The syntax to declare the main function is:
   Syntax to Declare main method:

   ```
   int main()
   ```

```
{}
```

3. **Variable Declaration:** It refers to the variables that are to be used in the function, the variables are to be declared before any operation in the function.

   Example:

```
int main()

{

int a;
.
.
```

4. **Body:** Body of a function in C program, refers to the operations that are performed in the functions. It can be anything like manipulations, searching, sorting, printing, etc.

   Example:

```
int main()

{

int a;


printf("%d", a);
.
.
```

5. **Return Statement:** The return statement refers to the returning of the values from a function. This return statement and return value depend upon the return type of the function. For example, if the return type is void, then there will be no return statement. In any other case, there will be a return statement and the return value will be of the type of the specified return type.

   Example:

```
int main()

{

int a;


printf("%d", a);


return 0;
}
```

Writing first program:

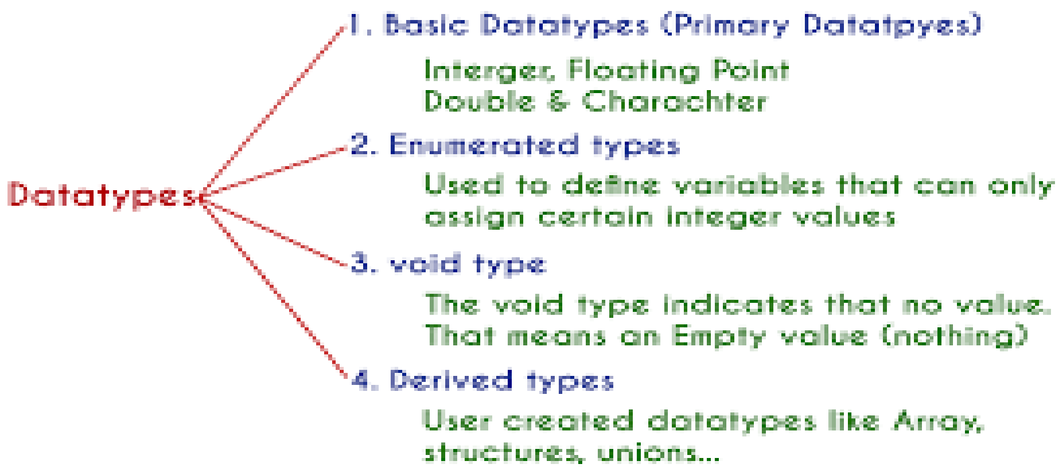
Following is first program in C

```
#include <stdio.h>
int main()
 {
   int number1, number2, sum;
```

```
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);
    // calculating sum
    sum = number1 + number2;
       printf("%d + %d = %d", number1, number2, sum);
    return 0;
}
```

DATA TYPES IN C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.


Datatypes
1. Basic Datatypes (Primary Datatpyes)
Interger, Floating Point
Double & Charachter
2. Enumerated types
Used to define variables that can only
assign certain integer values
3. void type
The void type indicates that no value.
That means an Empty value (nothing)
4. Derived types
User created datatypes like Array,
structures, unions...

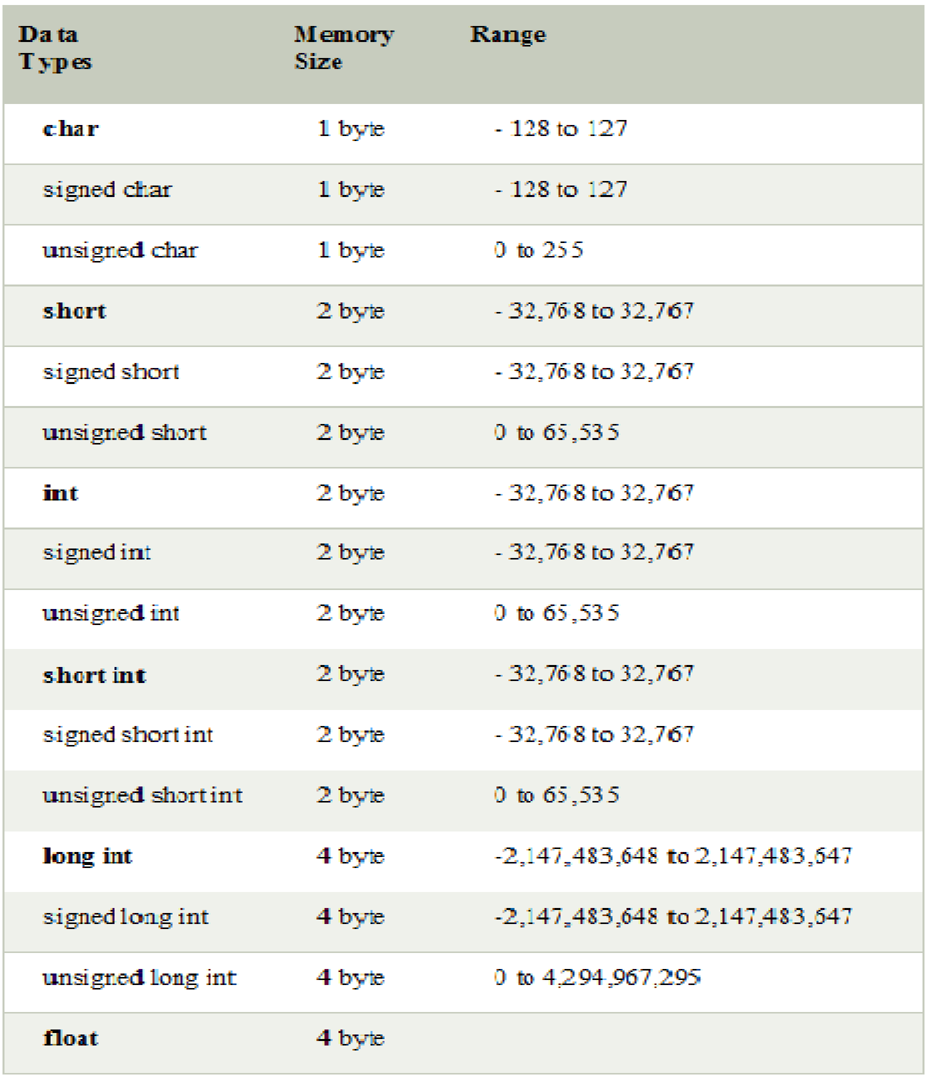
There are the following data types in C language.

Basic Data Types

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

- **char:** The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
- **int:** As the name suggests, an int variable is used to store an integer.
- **float:** It is used to store decimal numbers (numbers with floating point value) with single precision.
- **double:** It is used to store decimal numbers (numbers with floating point value) with double precision.

The memory size of the basic data types may change according to 32 or 64-bit operating system.



Data Types	Memory Size	Range
char	1 byte	- 128 to 127
signed char	1 byte	- 128 to 127
unsigned char	1 byte	0 to 255
short	2 byte	- 32,768 to 32,767
signed short	2 byte	- 32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	2 byte	- 32,768 to 32,767
signed int	2 byte	- 32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	- 32,768 to 32,767
signed short int	2 byte	- 32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	

TOKENS IN C

we cannot create a sentence without using words; similarly, we cannot create a program in C without using tokens in C. Therefore, we can say that tokens in C is the building block or the basic component for creating a program in C language.

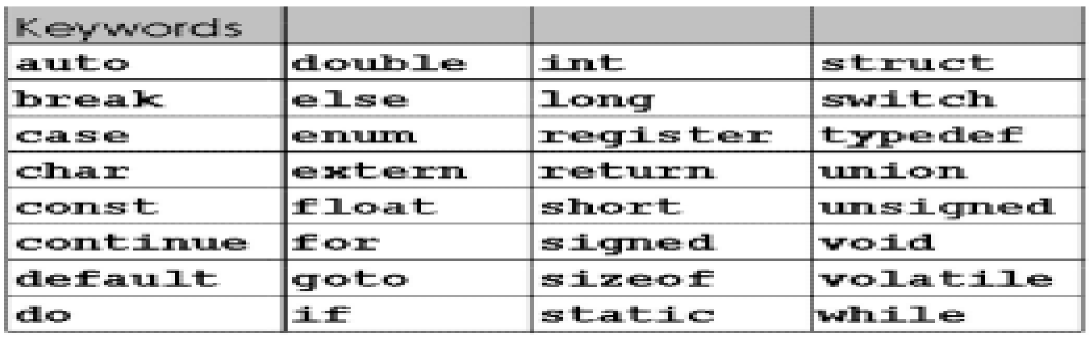
- Keywords in C
- Identifiers in C

- Strings in C
- Operators in C
- Constant in C
- Special Characters in C

- Keywords in C

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:



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

- **Identifiers in C**

Identifiers in C are used for naming variables, functions, arrays, structures, etc. Identifiers in C are the user-defined words. It can be composed of uppercase letters, lowercase letters, underscore, or digits, but the starting letter should be either an underscore or an alphabet. Identifiers cannot be used as keywords. Rules for constructing identifiers in C are given below:

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

- **Strings in C**

Strings in C are always represented as an array of characters having null character '\0' at the end of the string. This null character denotes the end of the string. Strings in C are enclosed within double quotes, while characters are enclosed within single characters. The size of a string is a number of characters that the string contains.

Now, we describe the strings in different ways:

char a[10] = "javatpoint"; // The compiler allocates the 10 bytes to the 'a' array.

char a[] = "javatpoint"; // The compiler allocates the memory at the run time.

char a[10] = {'j','a','v','a','t','p','o','i','n','t','\0'}; // String is represented in the form of characters.

- **Operators in C**

Operators in C is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands. Depending on the number of operands, operators are classified as follows:

- **Unary Operator**

A unary operator is an operator applied to the single operand. For example: increment operator (++), decrement operator (--), sizeof, (type)*.

- **Binary Operator**

The binary operator is an operator applied between two operands. The following is the list of the binary operators:

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators
- Conditional Operators
- Assignment Operator
- Misc Operator

- **Constants in C**

A constant is a value assigned to the variable which will remain the same throughout the program, i.e., the constant value cannot be changed.

There are two ways of declaring constant:

- Using const keyword

- Using #define pre-processor

- **Types of constants in C**

Constant	**Example**
Integer constant	**10, 11, 34, etc.**
Floating-point constant	**45.6, 67.8, 11.2, etc.**
Octal constant	**011, 088, 022, etc.**
Hexadecimal constant	**0x1a, 0x4b, 0x6b, etc.**
Character constant	**'a', 'b', 'c', etc.**
String constant	**"java", "c++", ".net", etc.**

- **Special characters in C**

Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.

- **Square brackets [ ]:** The opening and closing brackets represent the single and multidimensional subscripts.
- **Simple brackets ( ):** It is used in function declaration and function calling. For example, printf() is a pre-defined function.
- **Curly braces { }:** It is used in the opening and closing of the code. It is used in the opening and closing of the loops.
- **Comma (,):** It is used for separating for more than one statement and for example, separating function parameters in a function call, separating the variable when printing the value of more than one variable using a single printf statement.
- **Hash/pre-processor (#):** It is used for pre-processor directive. It basically denotes that we are using the header file.
- **Asterisk (*):** This symbol is used to represent pointers and also used as an operator for multiplication.
- **Tilde (~):** It is used as a destructor to free memory.
- **Period (.):** It is used to access a member of a structure or a union.

- **Comments in C**

Comments in C language are used to provide information about lines of code. It is widely used for documenting code. There are 2 types of comments in the C language.

1. Single Line Comments
2. Multi-Line Comments

Single Line Comments

Single line comments are represented by double slash \\. Let's see an example of a single line comment in C.

```
#include<stdio.h>
int main(){
   //printing information
   printf("Hello C");
return 0;
}
```

Mult Line Comments

Multi-Line comments are represented by slash asterisk * ... *\. It can occupy many lines of code, but it can't be nested. Syntax:

```
/*
code
to be commented
*/
```

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

- **Operators**

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

  C operators are classified into number of categories. They include

  1. Arithmetic operators
  2. Relational operators
  3. Logical operators
  4. Assignment operators
  5. Increment and decrement operators(unary operator)
  6. Conditional operator
  7. Bitwise operator
  8. Special operators

1.Arithmetic operators

C provides all basic arithmetic operators. They are listed below

Operator	meaning
+	addition/ unary plus
-	substraction/unary minus
/	division
%	modulo division

a.integer arithmetic

when both the operands in a single arithmetic expression such as a+b are integers, the expression is called as integer expression.

Example int a=10, b=20;

C= a+b;

Here in the above example we are using only integer values.

b.real arithmetic

an arithmetic which is involving real values(decimal values) is called real arithmetic.

c.mixed mode arithmetic

when one operand is integer and another is real i.e both the types of variables (integer and real) values are used for operation is called as mixed more arithmetic.

E.g.

```
Float a=10.5;
Int  b=30;
C=a+b;
```

This type of expression that c=a+b is called an mixed arithmetic.

2.Relational operator

we often compare two quantities, and depending on their relation, take certain decision

eg. a>b here we are checking the condition whether a > b using the relational operator >.

Operator	meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
= =	is equal to

! =	is not equal to

when arithmetic expressions are used on either side of the relational operator, the arithmetic operator will be executed first and then the results will be compared. Relational operators are used in *decision making* statements.

3.Logical operators

The logical operators are used when we want to check more than one relational expressions.

Operator	meaning
&&	logical AND
| |	logical OR
!	logical NOT

an example for logical operators

```
if(a>b && a>c)
{
printf("a is greater");
}
else
{
printf("a is not greater");
}
```

here in the above example we are checking whether a is greater in a single conditional statement itself by using an &&(AND). It will print a is greater if and only if the condition is satisfied that a>b and a>c.

truth table

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

eg.

```
if(age>55 &&salary<1000)
if(number<0 | | number>1000)
```

4.Assignment operator

Data values are given to the variable by assign statement. The assignment statement uses assignment operator (=)

the syntax is

```
Variable = data
```

eg. A=10;

eg. Sum=a+10;

Statement with simple Assignment operator	statement with shorthand operator
a=a+1	**a+=1**
a=a-1	**a -=1**
a=a*(n+1)	**a *=n+1**

the use of short hand operator has three advantages

- what appears on the left-hand side need not be repeated and therefore it becomes easier to write.
- The statement is more concise and easier to read.
- The statement is more efficient.

5. Increment and decrement operator

C has very useful operators not generally present in other languages. These are *increment* and *decrement* operators:

++ and –

the operator ++ adds 1 to the operand while – substracts 1 . both of them are *unary operators*.

++m is equivalent to m=m+1; (or m+=1)

- -m is equivalent to m=m- 1; (or m- =1)

we will be using this increment and decrement operators in **for** and **while** loop.

6.Conditional operator

The c language has a unusual operator, useful for making two way decisions. This operator is a combination of ?

and : and takes three operands. This operator is popularly known as conditional operator. The general syntax of this operator is .

```
Conditional expression ? expression1:
```

Eg.

```
Large=(num1>num2)?num1:num2
```

In the above example the largest number between num1 and num2 will be assigned to large. That is if num1 is larger then num2 num1 will be assigned to large . this is similar to if statement but the syntax differs.

7.Bitwise operators

C has distinction of supporting special operators known as **bitwise operator** for manipulating of data at bit level. These are used for testing the bits, or shifting them right or left.

Operator	meaning
&	bitwise AND
|	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right
~	one's complement

8.Special operators

C supports some special operators such as **comma**(,) operator, **sizeof** operator**, pointer** operator.

Comma operator can be used to link related expressions

Eg.

```
int a,b, c=a+b;
```

sizeof operator is used to find the size of the variable or identifier.

Eg.

```
M= sizeof (a);
```

Here in the above example the size of **a** will be assigned to **M.**

- **INPUT AND OUTPUT STATEMENTS**
    - **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.
    - All these built-in functions are present in C header files, we will also specify the name of header files in which a particular function is defined while discussing about it.
- 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.

- **printf() statement**

  printf( ) is predefined ,standard C function for printing output.

  The general form of printf() is,

  printf("<format string>",<list of variables>);

  <format string> could be,

%f	for	printing	real values.
%d	for	printing	integer values.
%c	for	printing	character values.

  Eg: printf("%f",si);

  printf("%d",i);

- **scanf() statement**

  scanf( ) is a predefined,standard C function used to input data from keyboard.

The general form of scanf() is

scanf("<format string>",<address of variables>);

Eg: scanf("%d",&n);

scanf("%f",&interest);

& is pointer operator which specifies the address of that variable.

```
#include<stdio.h>
int main(){
int number;
printf("enter a number:");
scanf("%d",&number);
printf("cube of number is:%d ",number*number*number);
return 0;  }
```

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. You can use this method in a loop in case you want to read more than one character. 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");
   /*
      Take a character as input and
      store it in variable c
   */
   c = getchar();
   /*
      display the character stored
      in variable c
   */
   putchar(c);
}
```

gets() & puts() functions

The gets() function reads a line from **stdin**(standard input) into the buffer pointed to by str pointer, until either a terminating newline or EOF (end of file) occurs.
The puts()function writes the string str and a trailing newline to **stdout**.

str → This is the pointer to an array of chars where the C string is stored. (Ignore if you are not able to understand this now.)

```
#include<stdio.h>


void main()
{
   /* character array of length 100 */
   char str[100];
   printf("Enter a string");
   gets( str );
   puts( str );
   getch();
}
```

- **CONTROL STATEMENTS**

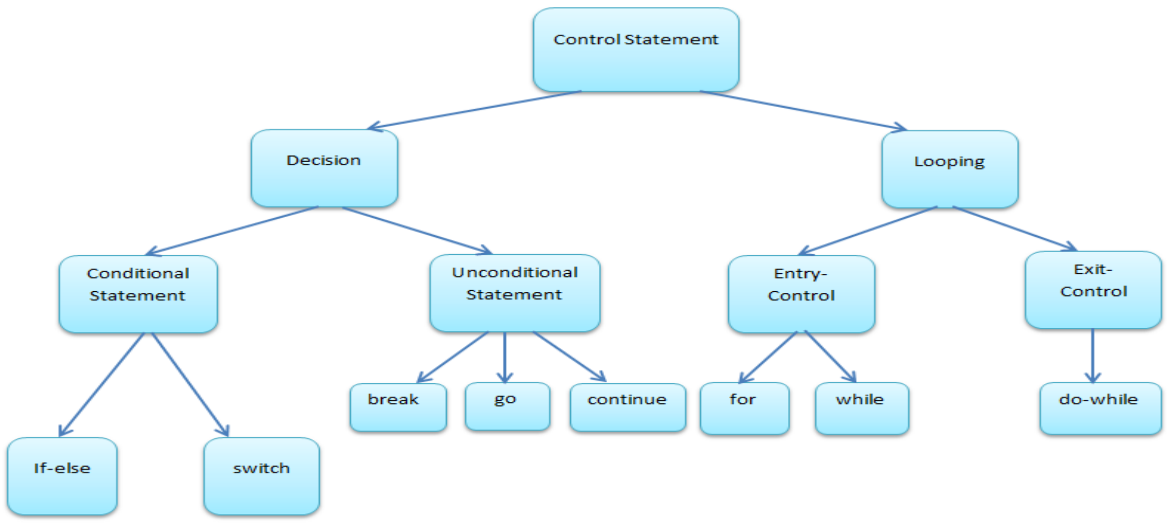
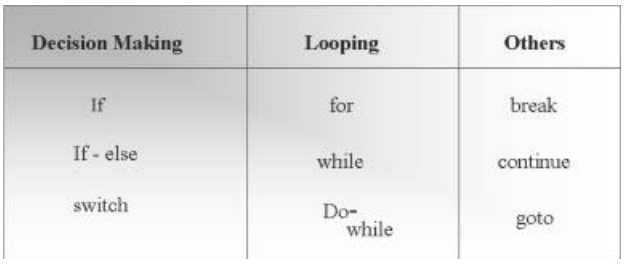


Fig: Control Statements in C

Control Structures
These are the statements which alter the normal (sequential)execution flow of a program.
There are three types of Control Statements in C



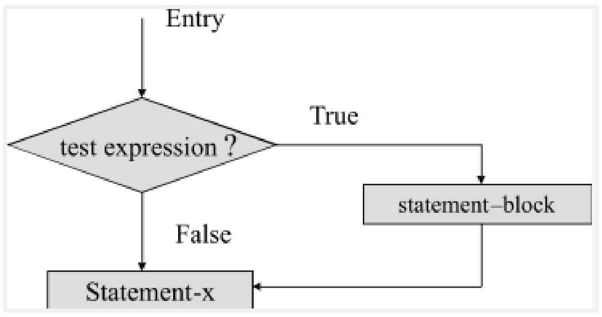
Decision Making	Looping	Others
If	for	break
If - else	while	continue
switch	Do-while	goto

- ***if* statement**

If the test expression is evaluated to true, statements inside the body of if are executed.

If the test expression is evaluated to false, statements inside the body of if are not executed.

```
if(test expression)
{
statement–block;
}
statement-x;
```



```
#include <stdio.h>
void main () {
intvarNumValue = 1;
if( varNumValue< 10 ) { // checks the condition
printf("if statement instructions");
}}
```

- If-else statement

If the test expression is evaluated to true,

- statements inside the body of if are executed.
- statements inside the body of else are skipped from execution.

If the test expression is evaluated to false,

- statements inside the body of else are executed
- statements inside the body of if are skipped from execution.

```
if(test expression)
{
true-block statements;
}
else
{
False-block statements;
}
statement-x;
```

- *If-elseif-else statement*

The if...else statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.

The if...else ladder allows you to check between multiple test expressions and execute different statements.

In this syntax is similar to:

```
if( expression to be evaluated ) {
// sets of instruction which needs to be executed for if-block
} else if{
// sets of instruction which needs to be executed for else-if block
} else {
// sets of instruction which needs to be executed for else block
}
```

Example Program

```
#include<stdio.h>
int main() {
int number1, number2;
printf("Enter two integers: ");
scanf("%d %d", &number1, &number2);
//checks if the two integers are equal.
if(number1 == number2) {
printf("Result: %d = %d",number1,number2);
    }
//checks if number1 is greater than number2.
elseif (number1 > number2) {
printf("Result: %d > %d", number1, number2);
  }
//checks if both test expressions are false
else {
printf("Result: %d < %d",number1, number2);
    }
return0;
}
```

- **switch...case**

The switch statement allows us to execute one code block among many alternatives.

You can do the same thing with the if...else..if ladder. However, the syntax of the switchstatement is much easier to read and write.

Syntax of switch...case

```
switch (expression)

{

case constant1:

// statements

break;


case constant2:

// statements

break;

   .

   .

default:

// default statements

}
```

Example Program

```
Program to create a simple calculator
#include<stdio.h>

int main(){
charoperator;
double n1, n2;

printf("Enter an operator (+, -, *, /): ");
scanf("%c",&operator);
printf("Enter two operands: ");
scanf("%lf %lf",&n1,&n2);

switch(operator)
{
case'+':
printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
break;

case'-':
printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
break;
case'*':
printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
break;

case'/':
printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
break;

// operator doesn't match any case constant +, -, *, /
default:
printf("Error! operator is not correct");
}

return0;
}
```

- C programming has three types of loops:

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

for Loop

The syntax of the for loop is:

```
for(initializationStatement;testExpression;updateStatement)
{
// statements inside the body of loop
```

How for loop works?

- The initialization statement is executed only once.
- Then, the test expression is evaluated. If the test expression is evaluated to false, the forloop is terminated.
- However, if the test expression is evaluated to true, statements inside the body of forloop are executed, and the update expression is updated.
- Again the test expression is evaluated.

This process goes on until the test expression is false. When the test expression is false, the loop terminates.

```
// Print numbers from 1 to 10
#include<stdio.h>
int main(){
inti;
for(i=1;i<11;++i)
{
printf("%d ",i);
}
return0;
}
```

while loop

The syntax of the while loop is:

```
while(testExpression)
{
// statements inside the body of the loop
}
```

How while loop works?

- The while loop evaluates the test expression inside the parenthesis ().
- If the test expression is true, statements inside the body of while loop are executed. Then, the test expression is evaluated again.
- The process goes on until the test expression is evaluated to false.
- If the test expression is false, the loop terminates (ends).

```
// Print numbers from 1 to 5

#include<stdio.h>
int main()
{
inti = 1;
while (i<= 5)
 {
printf("%d\n", i)
++i;
}
return0;
}
```

- do...while loop

The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.

The syntax of the do...while loop is:

```
do

{

// statements inside the body of the loop

}

while(testExpression);
```

Example Program:

```
#include<stdio.h>
int main (){
/* local variable definition */
int a =10;
/* do loop execution */
do{
printf("value of a: %d\n", a);
a = a +1;
}while( a <20);
return0;
}
```

How do...while loop works?

- The body of do...while loop is executed once. Only then, the test expression is evaluated.
- If the test expression is true, the body of the loop is executed again and the test expression is evaluated.
- This process goes on until the test expression becomes false.
- If the test expression is false, the loop ends.

- **C break**

The break statement ends the loop immediately when it is encountered. Its syntax is:

```
break;
```

- **C continue**

The continue statement skips the current iteration of the loop and continues with the next iteration. Its syntax is:

```
continue;
```

- go to Statement

The goto statement allows us to transfer control of the program to the specified label.

Syntax of goto Statement

```
goto label;

........
```

....

label:

statement;

FUNCTIONS IN C

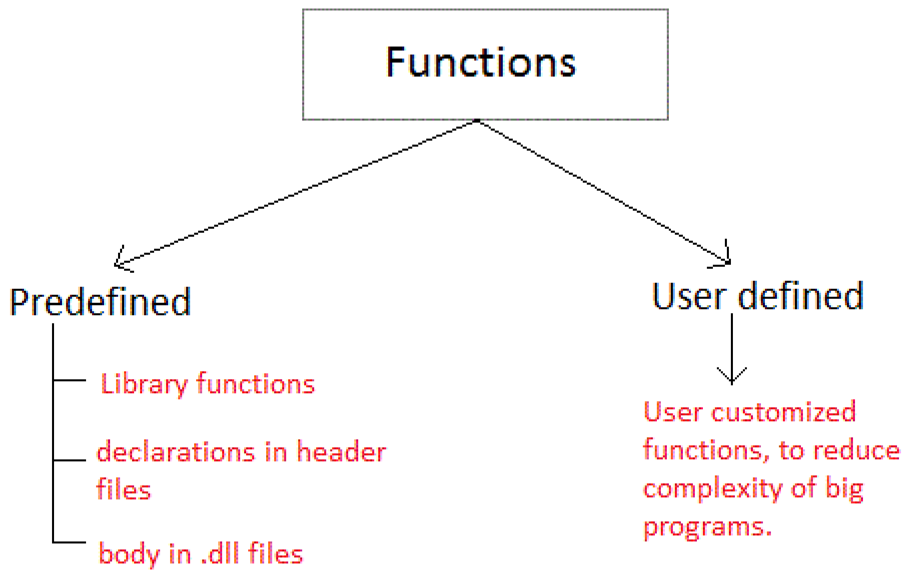
A **function** is a block of code that performs a particular task.

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,

```
returntypefunctionName(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**.

```
#include<stdio.h>

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

intmain()
{
inti, 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);

return0;
}

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

Function definition Syntax

```
returntypefunctionName(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**

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

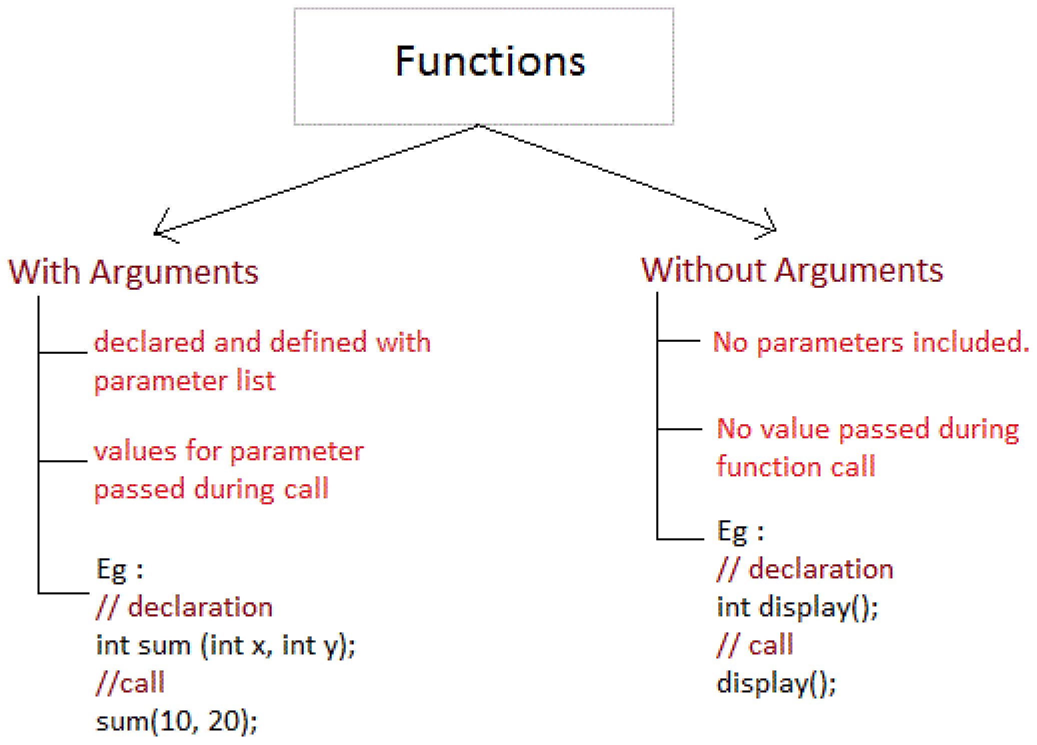
When a function is called, control of the program gets transferred to the function.

functionName(argument1, argument2,...);

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

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.



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

```
#include<stdio.h>

Void greatNum();        // function declaration

Int main()
{
greatNum();        // function call
return0;
}

Void greatNum()        // function definition
{
inti, 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

```
#include<stdio.h>

intgreatNum();        // function declaration

int main()
{
int result;
result = greatNum();        // function call
printf("The greater number is: %d", result);
return0;
}

intgreatNum()        // function definition
{
inti, 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
returngreaterNum;
}
```

Function with arguments and no return value

```
#include<stdio.h>

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

intmain()
{
inti, j;
printf("Enter 2 numbers that you want to compare...");
scanf("%d%d", &i, &j);
greatNum(i, j);        // function call
return0;
}

voidgreatNum(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

```
#include<stdio.h>

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

intmain()
{
inti, 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);
return0;
}

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

RECURSION

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

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

```
void recursion() {
   recursion(); /* function calls itself */
}

int main() {
   recursion();
}
```

Fibonacci Series

```
#include <stdio.h>

int fibonacci(int i) {

   if(i == 0) {
      return 0;
   }

   if(i == 1) {
      return 1;
   }
   return fibonacci(i-1) + fibonacci(i-2);
}

int  main() {

   int i;

   for (i = 0; i < 10; i++) {
      printf("%d\t\n", fibonacci(i));
   }

   return 0;
}
```

SCSA1202 PROGRAMMING WITH C AND C++

UNIT 2 - ARRAYS, STRINGS AND STRUCTURES 9Hrs

Arrays: Single and Multidimensional Arrays-– Array as Function Arguments, Strings: String Handling Functions, Structure: Nested Structures – Array of Structures – Structure as Function Argument–Function that Returns Structure, Union.

Array : - An array is defined as the collection of similar type of data items stored at contiguous memory locations. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

So far we have used only single variable name for storing one data item. If we need to store multiple copies of the same data then it is very difficult for the user. To overcome the difficulty a new data structure is used called arrays.

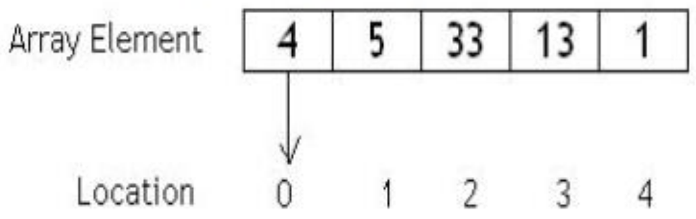
- An array is a linear and homogeneous data structure
- An array permits homogeneous data. It means that similar types of elements are stored contiguously in the memory under one variable name.
- An array can be declared of any standard or custom data type.

Example of an Array:

Suppose we have to store the roll numbers of the 100 students the we have to declare 100 variables named as roll1, roll2, roll3, ……. roll100 which is very difficult job. Concept of C programming arrays is introduced in C which gives the capability to store the 100 roll numbers in the contiguous memory which has 100 blocks and which can be accessed by single variable name.

1. C Programming Arrays is the Collection of Elements
2. C Programming Arrays is collection of the Elements of the same data type.
3. All Elements are stored in the Contiguous memory
4. All elements in the array are accessed using the subscript variable (index).

Pictorial representation of C Programming Arrays



The above array is declared as int a [5];   a[0] = 4;   a[1] = 5;   a[2] = 33; a[3] = 13; a[4] = 1;

In the above figure 4, 5, 33, 13, 1 are actual data items. 0, 1, 2, 3, 4 are index variables.

Index or Subscript Variable:

1. Individual data items can be accessed by the name of the array and an integer enclosed in square bracket called subscript variable / index
2. Subscript Variables helps us to identify the item number to be accessed in the contiguous memory.

   What is Contiguous Memory?

   a. When Big Block of memory is reserved or allocated then that memory block is called as Contiguous Memory Block.
   b. Alternate meaning of Contiguous Memory is continuous memory.
   c. Suppose inside memory we have reserved 1000-1200 memory addresses for special purposes then we can say that these 200 blocks are going to reserve contiguous memory.

Characteristics of an array:

1. The declaration int a [5] is nothing but creation of five variables of integer types in memory instead of declaring five variables for five values.
2. All the elements of an array share the same name and they are distinguished from one another with the help of the element number.
3. The element number in an array plays a major role for calling each element.
4. Any particular element of an array can be modified separately without disturbing the other elements.
5. Any element of an array a[ ] can be assigned or equated to another ordinary variable or array variable of its type.
6. Array elements are stored in contiguous memory locations.

Array Declaration:

Array has to be declared before using it in C Program. Array is nothing but the collection of elements of similar data types.
Syntax: array name [size1][size2].....[sizen];

Where Data type → Data Type of Each Element of the array and Data Type specifies the type of the array. We can compute the size required for storing the single cell of array,
Array name → Valid identifier is any valid variable or name given to the array. Using this identifier name array can be accessed.
Size → Dimensions of the Array, It is maximum size that array can have.

What does Array Declaration tell to Compiler?

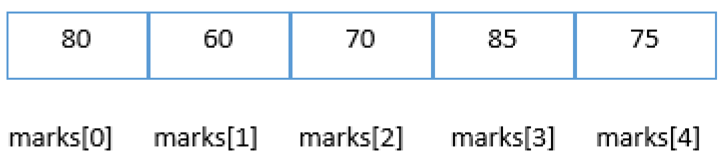
1. Type of the Array
2. Name of the Array
3. Number of Dimension
4. Number of Elements in Each Dimension

Initialization of Array

Initialization means assigning initial variable to an array. The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.

```
int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
```

where marks is the name of the array.



Initialization of Array

Example Program for Array Intialization :

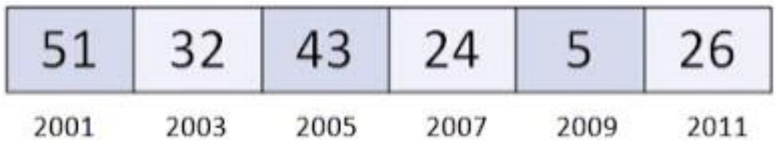
```
#include<stdio.h>
int main()
{
int i=0;
int marks[5];//declaration of array
marks[0]=80;//initialization of array
marks[1]=60;
marks[2]=70;
marks[3]=85;
marks[4]=75;
//traversal of array
for(i=0;i<5;i++){
printf("%d \n",marks[i]);
}//end of for loop
return 0;
}
```

Output

```
80
60
70
85
75
```

Accessing Array :

1. We all know that array elements are randomly accessed using the subscript variable.
2. Array can be accessed using array-name and subscript variable written inside pair of square brackets [ ]. Consider the below example of an array



In this example we will be accessing array like this arr[3] = Forth Element of Array arr[5] = Sixth Element of Array whereas elements are assigned to an array using below way arr[0] = 51; arr[1] = 32; arr[2] = 43; arr[3] = 24; arr[4] = 5; arr[5] =26;

TYPES OF ARRAY:

1. One dimensional array
2. Two dimensional array or multi dimensional array.

ONE DIMENSIONAL ARRAY:

In this we deal with only one dimension when storing data in the memory.

Example:

```
// Program to find the average of n (n < 10) numbers using arrays
#include <stdio.h>
int main()
{
int marks[10], i, n, sum = 0, average;
printf("Enter n: ");
scanf("%d", &n);
for(i=0; i<n; ++i)
{
printf("Enter number%d: ",i+1);
scanf("%d", &marks[i]);
sum += marks[i];
}
average = sum/n;
printf("Average = %d", average);
return 0;
}
```

Output:

```
Enter n: 5
Enter number1: 45
```

Operations with One Dimensional Array :

1. **Deletion –** Involves deleting specified elements form an array.
2. **Insertion –** Used to insert an element at a specified position in an array.
3. **Searching** – An array element can be searched. The process of seeking specific elements in an array is called searching.
4. **Merging** – The elements of two arrays are merged into a single one.
5. **Sorting** – Arranging elements in a specific order either in ascending or in descending order

Example Programs :

1) C Program for deletion of an element from the specified location from an Array

```
#include <stdio.h>
int main()
{
 int arr[30], num, i, loc;
printf("\nEnter no of elements:");
scanf("%d", &num); //Read elements in an array
 printf("\nEnter %d elements :", num);
for (i = 0; i < num; i++)
{
 scanf("%d", &arr[i]); } //Read the location
printf("\nLocation of the element to be deleted :");
scanf("%d", &loc); /* loop for the deletion */
while (loc < num)
{
arr[loc - 1] = arr[loc]; loc++; } num--; // No of elements reduced by 1
//Print Array
 for (i = 0; i < num; i++)
printf("\n %d", arr[i]);
 return (0);
}
```

Output :

```
Enter no of elements: 5
Enter 5 elements: 3 4 1 7 8
Location of the element to be deleted: 3
3 4 7 8
```

2) **C Program to delete duplicate elements from an array**

```
int main()
{
int arr[20], i, j, k, size;
printf("\nEnter array size: ");
scanf("%d", &size);
printf("\nAccept Numbers: ");
for (i = 0; i < size; i++)
scanf("%d", &arr[i]);
 printf("\nArray with Unique list: ");
for (i = 0; i < size; i++)
{
 for (j = i + 1; j < size;)
{
if (arr[j] == arr[i])
{
 for (k = j; k < size; k++)
{
arr[k] = arr[k + 1];
}
size--;
}
 else j++;
} }
 for (i = 0; i < size; i++)
{
 printf("%d ", arr[i]);
}
return (0); }
```

Output:

```
Enter array size: 5
Accept Numbers: 1 3 4 5 3
Array with Unique list: 1 3 4 5
```

3) C Program to find smallest element in an array

```
#include<stdio.h>
int main()
{
int a[30], i, num, smallest;
 printf("\nEnter no of elements :");
scanf("%d", &num); //Read n elements in an array
for (i = 0; i < num; i++)
scanf("%d", &a[i]); //Consider first element as smallest
smallest = a[0];
 for (i = 0; i < num; i++)
{
if (a[i] < smallest) { smallest = a[i];
} }
 // Print out the Result
printf("\nSmallest Element : %d", smallest);
return (0);
}
```

Output:

```
Enter no of elements : 5 11 44 22 55 99
Smallest Element : 11
```

4. C Program to find largest element in an array

```
#include int main()
{
int a[30], i, num, largest;
printf("\nEnter no of elements :");
scanf("%d", &num); //Read n elements in an array
 for (i = 0; i < num; i++)
scanf("%d", &a[i]);
 //Consider first element as largest
 largest = a[0];
for (i = 0; i < num; i++)
{
if (a[i] > largest)
    { largest = a[i]; }
 }
// Print out the Result
 printf("\nLargest Element : %d", largest);
return (0);
}
```

Output:

```
 Enter no of elements : 5 11 55 33 77 22
 Largest Element : 77
```

5. C Program to reverse an array elements in an array

```
#include<stdio.h>
 int main()
 { int arr[30], i, j, num, temp;
 printf("\nEnter no of elements : ");
scanf("%d", &num);
//Read elements in an array
for (i = 0; i < num; i++)
{ scanf("%d", &arr[i]);
}
 j = i - 1;  // j will Point to last Element
 i = 0; // i will be pointing to first element
while (i < j)
{
temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
 i++; // increment i
j--;  // decrement j
}
//Print out the Result of Insertion
printf("\nResult after reversal : ");
 for (i = 0; i < num; i++)
{
 printf("%d \t", arr[i]);
 }
return (0);
}
```

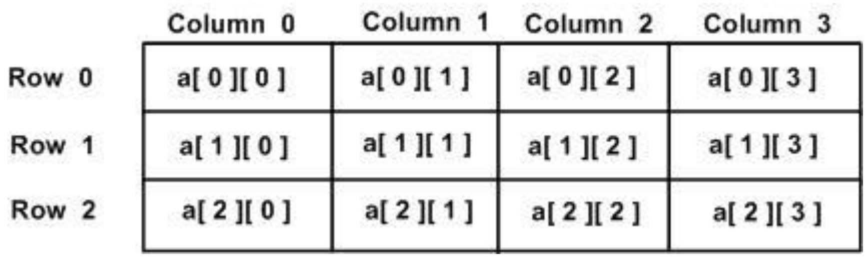
Output:

```
Enter no of elements: 5 11 22 33 44 55
Result after reversal : 55 44 33 22 11
```

TWO DIMENSIONAL ARRAY:
The two-dimensional array can be defined as an array of arrays.
The 2D array is organized as matrices which can be represented as the collection of rows and columns.

DECLARING A 2D ARRAY:

arr[rows][column];



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

INITIALIZING TWO-DIMENSIONAL ARRAYS:

Two dimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

int a[2][[2] = { {1,2}, {3,4} }; Where a[2][2] means there are two rows and two columns, in the nested brackets, the first set denotes values in the first row and next set in the second row.

as in the matrix $\begin{pmatrix} 0,0 & 0,1 \\ 0,\ 1 & 1,\ 1 \end{pmatrix}$ positions as in row and column.

values will appear as $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ according to the positions.

Example:

```
#include<stdio.h>
int main()
{
int i=0,j=0;
int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
//traversing 2D array
 for(i=0;i<4;i++) // since row size is 4
{
 for(j=0;j<3;j++) //since column size is 3
{
   printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
 }//end of j
}//end of i
return 0;
}
```

Output:

```
arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6
```

ARRAYS AS FUNCTIONS:

- A single array element or an entire array can be passed to a function.
- This can be done for both one-dimensional array or a multi-dimensional array.

PASSING ONE-DIMENSIONAL ARRAY IN FUNCTION

Single element of an array can be passed in similar manner as passing variable to a function.

Example:

```
#include <stdio.h>
int main()
{
int ageArray[] = { 2, 3, 4 };
display(ageArray[2]); //Passing array element ageArray[2] only.
return 0;
}
void display(int age)
{
printf("%d", age);
}
```

STRINGS

- The string can be defined as the one-dimensional array of characters terminated by a null ('\0').
- The character array or the string is used to manipulate text such as word or sentences.
- Each character in the array occupies one byte of memory, and the last character must always be 0.
- The termination character ('\0') is important in a string since it is the only way to identify where the string ends.
- When we define a string as char s[10], the character s[10] is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring string by char array

char ch[5]= {‘c’,’s’,’e’};

As we know, array index starts from 0, so it will be represented as in the figure given below.

0 1 2 3 (index)

c	s	e	\0

While declaring string, size is not mandatory. So we can write the above code as given below:

Char ch[]= {‘c’,’s’,’e’};

We can also define the string by the string literal in C language. For example:

char ch[]= “cse”;

In such case, '\0' will be appended at the end of the string by the compiler.

Example:

```
#include<stdio.h>
#include <string.h>
int main(){
  char ch[11]= {‘c’,’s’,’e’};
   char ch2[11]="cse";
   printf("Char Array Value is: %s\n", ch);
   printf("String Literal Value is: %s\n", ch2);
 return 0;
}
```

Output:

```
Char Array Value is: cse
String Literal Value is:cse
```

TRAVERSING STRING

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text. Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end the string and terminate the loop.

Hence, there are two ways to traverse a string.

- By using the length of string
- By using the null character.

USING THE LENGTH OF STRING

Let's see an example of counting the number of vowels in a string.

```
#include<stdio.h>
void main ()
{
   char s[11] = "javatpoint";
   int i = 0;
   int count = 0;
   while(i<11)
   {
      if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
      {
         count ++;
      }
      i++;
   }
   printf("The number of vowels is %d",count);
}
```

Output:

```
The number of vowels is 4
```

USING THE NULL CHARACTER

Let's see the same example of counting the number of vowels by using the null character.

```
#include<stdio.h>
void main ()
{
   char s[11] = "javatpoint";
   int i = 0;
   int count = 0;
   while(s[i] != NULL)
   {
      if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
      {
         count ++;
      }
      i++;
   }
   printf("The number of vowels is %d",count);
}
```

Output:

```
The number of vowels is 4
```

C GETS() AND PUTS() FUNCTIONS

The gets() and puts() are declared in the header file stdio.h. Both the functions are involved in the input/output operations of the strings.

C GETS() FUNCTION

The gets() function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null character is added to the array to make it a string. The gets() allows the user to enter the space-separated strings. It returns the string entered by the user.

DECLARATION

```
char[] gets(char[]);
```

C PUTS() FUNCTION

The puts() function is very much similar to printf() function. The puts() function is used to print the string on the console which is previously read by using gets() or scanf() function. The puts() function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by puts() will always be equal to the number of characters present in the string plus 1.

DECLARATION

```
int puts(char[])
```

Example:

```
#include<stdio.h>
#include <string.h>
int main(){
char name[50];
printf("Enter your name: ");
gets(name); //reads string from user
printf("Your name is: ");
puts(name);  //displays string
return 0;
}
```

Output:

```
Enter your name: abc
Your name is: abc
```

STRING FUNCTIONS

There are many important string functions defined in "string.h" library.

S.No	Function	Description
1.	strlen(string_name)	returns the length of string name.
2.	strcpy(destination, source)	copies the contents of source string to destination string.
3.	strcat(first_string, second_string)	concats or joins first string with second string. The result of the string is stored in first string.
4.	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5.	strrev(string)	returns reverse string.
6.	strlwr(string)	returns string characters in lowercase.
7.	strupr(string)	returns string characters in uppercase.

STRING LENGTH: STRLEN() FUNCTION

The strlen() function returns the length of the given string. It doesn't count null character '\0'

Example:

```
#include<stdio.h>
#include <string.h>
int main(){
char ch[20]= {'c', 's', 'e', '\0'};    };
   printf("Length of string is: %d",strlen(ch));
 return 0;
}
```

Output:

```
Length of string is: 3
```

COPY STRING: STRCPY()

The strcpy(destination, source) function copies the source string in destination.

Example:

```
#include<stdio.h>
#include <string.h>
int main(){
 char ch[5]={'c', 's', 'e', '\0'};
   char ch2[20];
   strcpy(ch2,ch);
   printf("Value of second string is: %s",ch2);
 return 0;
}
```

Output:

```
Value of second string is: cse
```

CONCATENATION: STRCAT()

The strcat(first_string, second_string) function concatenates two strings and result is returned to first_string.

Example:

```
#include<stdio.h>
#include <string.h>
int main(){
  char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
   char ch2[10]={'c', '\0'};
   strcat(ch,ch2);
   printf("Value of first string is: %s",ch);
 return 0;
}
```

Output:

```
Value of first string is: helloc
```

COMPARE STRING: STRCMP()

The strcmp(first_string, second_string) function compares two string and returns 0 if both strings are equal.

Here, we are using *gets()* function which reads string from the console.

Example:

```
#include<stdio.h>
#include <string.h>
int main(){
  char str1[20],str2[20];
  printf("Enter 1st string: ");
  gets(str1);//reads string from console
  printf("Enter 2nd string: ");
  gets(str2);
  if(strcmp(str1,str2)==0)
      printf("Strings are equal");
  else
      printf("Strings are not equal");
```

```
 return 0;
}
```

Output:

```
Enter 1st string: hello
Enter 2nd string: hello
Strings are equal
```

REVERSE STRING: STRREV()

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

Example:

```
#include<stdio.h>

#include <string.h>
int main(){
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nReverse String is: %s",strrev(str));
 return 0;
}
```

Output:

```
Enter string: cse
String is: cse
Reverse String is: esc
```

UPPERCASE: STRUPR()

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

Example:

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

Output:

```
Enter string: cse
String is: cse
Upper String is: CSE
```

LOWERCASE: STRLWR()

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

Example:

```
#include<stdio.h>
#include <string.h>
int main(){
  char str[20];
  printf("Enter string: ");
  gets(str);//reads string from console
  printf("String is: %s",str);
  printf("\nLower String is: %s",strlwr(str));
 return 0;
}
```

Output:

```
Enter string: CSE
String is: CSE
Lower String is: cse
```

STRUCTURE

- Structure in c is a user-defined data type that enables us to store the collection of different data types.
- Each element of a structure is called a member.
- The **struct** keyword is used to define the structure.

Let's see the **SYNTAX** to define the structure in c.

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeberN;
};
```

Let's see the **example** to define a structure for an entity employee in c.

```
struct employee
{   int id;
    char name[20];
    float salary;
};
```

Here, **struct** is the keyword; **employee** is the name of the structure; **id**, **name**, and **salary** are the members or fields of the structure.

DECLARING STRUCTURE VARIABLE

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

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

struct employee e1, e2;

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

2nd way:

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

```
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;
```

Comparing both approach to see which is good:

- If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.
- If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

ACCESSING MEMBERS OF THE STRUCTURE

There are two ways to access structure members:

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

Let's see the code to access the *id* member of e*1* variable by. (member) operator then access is done like this e1.id

Example:

```
#include<stdio.h>
#include <string.h>
struct employee
{   int id;
    char name[50];
}e1;  //declaring e1 variable for structure
int main( )
{
   //store first employee information
   e1.id=101;
   strcpy(e1.name, "abc");//copying string into char array
   //printing first employee information
   printf( "employee 1 id : %d\n", e1.id);
   printf( "employee 1 name : %s\n", e1.name);
return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : abc
```

NESTED STRUCTURE IN C

- C provides us the feature of nesting one structure within another structure by using which, complex data types are created.
- For example, we may need to store the address of an entity employee in a structure.
- The attribute address may also have the subparts as street number, city, state, and pin code.
- Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee.

Consider the following program as an exapmle.

```
#include<stdio.h>
struct address
{
    char city[20];
    int pin;
    char phone[14];
};
struct employee
{
    char name[20];
    struct address add;
};
void main ()
{
    struct employee emp;
    printf("Enter employee information?\n");
    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
    printf("Printing the employee information....\n");
    printf("name: %s\nCity: %s\nPincode: %d\nPhone: %s",emp.name,emp.add.city,emp.add.
pin,emp.add.phone;}
```

Output:

```
Enter employee information?

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun

City: Delhi

Pincode: 110001

Phone: 1234567890
```

The structure can be nested in the following ways.

1. By separate structure
2. By Embedded structure

SEPARATE STRUCTURE

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

```
struct Date
{
  int dd;
  int mm;
  int yyyy;
};
struct Employee
{
  int id;
  char name[20];
  struct Date doj;
}emp1;
```

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

EMBEDDED STRUCTURE

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

```
struct Employee
{
  int id;
  char name[20];
  struct Date
   {
     int dd;
     int mm;
     int yyyy;
   }doj;
}emp1;
```

ACCESSING NESTED STRUCTURE

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

1. e1.doj.dd
2. e1.doj.mm
3. e1.doj.yyyy

Nested Structure example

```
#include <stdio.h>
#include <string.h>
struct Employee
{
  int id;
  char name[20];
  struct Date
```

```
    {
      int dd;
      int mm;
      int yyyy;
    }doj;
}e1;
int main( )
{
   //storing employee information
   e1.id=101;
   strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
   e1.doj.dd=10;
   e1.doj.mm=11;
   e1.doj.yyyy=2014;
  //printing first employee information
   printf( "employee id : %d\n", e1.id);
   printf( "employee name : %s\n", e1.name);
   printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n", e1.doj.dd,e1.doj.mm,e1.d
oj.yyyy);
   return 0;
}
```

Output:

```
employee id : 101
employee name : Sonoo Jaiswal
employee date of joining (dd/mm/yyyy) : 10/11/2014
```

ARRAY OF STRUCTURES

WHY USE AN ARRAY OF STRUCTURES?

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

Example:

```
#include<stdio.h>
struct student
{
    char name[20];
    int id;
    float marks;
};
void main()
{
    struct student s1,s2,s3;
    printf("Enter the name, id, and marks of student 1 ");
    scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
    printf("Enter the name, id, and marks of student 2 ");
    scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
```

```
   printf("Enter the name, id, and marks of student 3 ");
   scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
   printf("Printing the details....\n");
   printf("%s %d %f\n",s1.name,s1.id,s1.marks);
   printf("%s %d %f\n",s2.name,s2.id,s2.marks);
   printf("%s %d %f\n",s3.name,s3.id,s3.marks);
}
```

Output:

```
Enter the name, id, and marks of student 1 James 90 90
Enter the name, id, and marks of student 2 Adoms 90 90
Enter the name, id, and marks of student 3 Nick 90 90
Printing the details....
James 90 90.000000
Adoms 90 90.000000
Nick 90 90.000000
```

- In the above program, we have stored data of 3 students in the structure.
- However, the complexity of the program will be increased if there are 20 students.
- In that case, we will have to declare 20 different structure variables and store them one by one.
- This will always be tough since we will have to declare a variable every time we add a student.
- Remembering the name of all the variables is also a very tricky task.
- However, c enables us to declare an array of structures by using which, we can avoid declaring the different structure variables; instead we can make a collection containing all the structures that store the information of different entities.

ARRAY OF STRUCTURES IN C

An array of structres in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

Let's see an example of an array of structures that stores information of 5 students and prints it.

```
#include<stdio.h>
#include <string.h>
struct student{
int rollno;
char name[10];
};
int main(){
int i;
struct student st[5];
printf("Enter Records of 5 students");
for(i=0;i<5;i++){
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter Name:");
scanf("%s",&st[i].name);
}
printf("\nStudent Information List:");
for(i=0;i<5;i++){
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
  return 0;
}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz

Student Information List:
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
Rollno:5, Name:Sarfraz
```

STRUCTURE IN FUNCTIONS

A structure can be passed to any function from main function or from any sub function.

Structure definition will be available within the function only.

It won't be available to other functions unless it is passed to those functions by value or by address (reference).

Else, we have to declare structure variable as global variable. That means, structure variable should be declared outside the main function. So, this structure will be visible to all the functions in a C program.

PASSING STRUCTURE TO FUNCTION IN C:

It can be done in below 3 ways.

1. Passing structure to a function by value
2. Passing structure to a function by address(reference)
3. No need to pass a structure – Declare structure variable as global

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION IN C BY VALUE:

In this program, the whole structure is passed to another function by value. It means the whole structure is passed to another function with all members and their values. So, this structure can be accessed from called function. This concept is very useful while writing very big programs in C.

Example:

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

struct student
{
        int id;
        char name[20];
        float percentage;
};
void func(struct student record);

int main()
{
        struct student record;
        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;
        func(record);
        return 0;
}

void func(struct student record)
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
}
```

Ouput:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

EXAMPLE PROGRAM – PASSING STRUCTURE TO FUNCTION IN C BY ADDRESS:

- In this program, the whole structure is passed to another function by address.
- It means only the address of the structure is passed to another function.
- The whole structure is not passed to another function with all members and their values.
- So, this structure can be accessed from called function by its address.

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

struct student
{
          int id;
          char name[20];
          float percentage;
};

void func(struct student *record);

int main()
{
         struct student record;

         record.id=1;
         strcpy(record.name, "Raju");
         record.percentage = 86.5;

         func(&record);
         return 0;
}

void func(struct student *record)
{
         printf(" Id is: %d \n", record->id);
         printf(" Name is: %s \n", record->name);
         printf(" Percentage is: %f \n", record->percentage);
}
```

Output:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

EXAMPLE PROGRAM TO DECLARE A STRUCTURE VARIABLE AS GLOBAL IN C:

- Structure variables also can be declared as global variables as we declare other variables in C.
- So, when a structure variable is declared as global, then it is visible to all the functions in a program.
- In this scenario, we don’t need to pass the structure to any function separately.
  #include <stdio.h>

```
#include <string.h>
struct student
{
           int id;
           char name[20];
```

```
        float percentage;
};
struct student record; // Global declaration of structure

void structure_demo();

int main()
{
        record.id=1;
        strcpy(record.name, "Raju");
        record.percentage = 86.5;
        structure_demo();
        return 0;
}

void structure_demo()
{
        printf(" Id is: %d \n", record.id);
        printf(" Name is: %s \n", record.name);
        printf(" Percentage is: %f \n", record.percentage);
}
```

Output:

```
Id is: 1
Name is: Raju
Percentage is: 86.500000
```

FUNCTION RETURNING STRUCTURE

Structure is user-defined data type, like built-in data types structure can be return from function.

Example:

```
#include<stdio.h>

    struct Employee

    {

        int Id;

        char Name[25];

        int Age;

        long Salary;

    };

    Employee Input();           //Statement   1

    void main()

    {

        struct Employee Emp;
        Emp = Input();
        printf("\n\nEmployee Id : %d",Emp.Id);
        printf("\nEmployee Name : %s",Emp.Name);
        printf("\nEmployee Age : %d",Emp.Age);
        printf("\nEmployee Salary : %ld",Emp.Salary);
    }
    Employee Input()
    {
        struct Employee E;
            printf("\nEnter Employee Id : ");
            scanf("%d",&E.Id);
            printf("\nEnter Employee Name : ");
            scanf("%s",&E.Name);
            printf("\nEnter Employee Age : ");
            scanf("%d",&E.Age);
            printf("\nEnter Employee Salary : ");
            scanf("%ld",&E.Salary);
        return E;          //Statement   2
    }
```

Output :

```
        Enter Employee Id : 10
        Enter Employee Name : Ajay
        Enter Employee Age : 25
        Enter Employee Salary : 15000


        Employee Id : 10
        Employee Name : Ajay
        Employee Age : 25
        Employee Salary : 15000
```

In the above example, statement 1 is declaring Input() with return type Employee. As we know structure is user-defined data type and structure name acts as our new user-defined data type, therefore we use structure name as function return type. Input() have local variable E of Employee type. After getting values from user statement 2 returns E to the calling function and display the values.

UNION:

- C Union is also like structure, i.e. collection of different data types which are grouped together. Each element in a union is called member.

- Union and structure in C are same in concepts, except allocating memory for their members.
- Structure allocates storage space for all its members separately.
- Whereas, **Union allocates one common storage space for all its members**
- We can access only one member of union at a time.
- We can't access all member values at the same time in union.
- But, structure can access all member values at the same time.
- This is because, Union allocates one common storage space for all its members.
- Where as Structure allocates storage space for all its members separately.
- Many union variables can be created in a program and memory will be allocated for each union variable separately.

Below table will help you how to form a C union, declare a union, initializing and accessing the members of the union.

Using normal variable	Using pointer variable

Syntax:
union tag_name
{
data type var_name1;
data type var_name2;
data type var_name3;
};
	Syntax:
	union tag_name
	{
	data type var_name1;
	data type var_name2;
	data type var_name3;
	};

Example:
union student
{
int mark;
char name[10];
float average;
};
	Example:
	union student
	{
	int mark;
	char name[10];
	float average;
	};

Declaring union using normal variable: union student report;	**Declaring union using pointer variable:** union student *report, rep;

Initializing union using normal variable:
union student report = {100, “Mani”, 99.5};
	Initializing union using pointer variable:
	union student rep = {100, “Mani”, 99.5};
	report = &rep;

Accessing union members using normal variable:
report.mark;
report.name;
report.average;
	Accessing union members using pointer variable:
	report -> mark;
	report -> name;
	report -> average;

Example:

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

union student
{
            char name[20];
            char subject[20];
            float percentage;
};

int main()
{
   union student record1;
   union student record2;

   // assigning values to record1 union variable
      strcpy(record1.name, "Raju");
      strcpy(record1.subject, "Maths");
      record1.percentage = 86.50;

      printf("Union record1 values example\n");
      printf(" Name       : %s \n", record1.name);
      printf(" Subject    : %s \n", record1.subject);
      printf(" Percentage : %f \n\n", record1.percentage);

   // assigning values to record2 union variable
      printf("Union record2 values example\n");
      strcpy(record2.name, "Mani");
      printf(" Name       : %s \n", record2.name);

      strcpy(record2.subject, "Physics");
      printf(" Subject    : %s \n", record2.subject);

      record2.percentage = 99.50;
```

```
        printf(" Percentage : %f \n", record2.percentage);
        return 0;
}
```

Output:

```
Union record1 values example
Name :
Subject :
Percentage : 86.500000;
Union record2 values example
Name : Mani
Subject : Physics
Percentage : 99.500000
```

SCSA1202 PROGRAMMING WITH C AND C++

UNIT3 –POINTERS & FILE PROCESSING 9Hrs

Pointers: Introduction, Arrays Using Pointers – Structures Using Pointers – Functions Using Pointer, Dynamic Memory Allocation, Storage Classes, File Handling in 'C'.

INTRODUCTION TO POINTERS

A pointer is a variable whose value is the address of another variable. Like any variables, we must declare a pointer variable at the beginning of the program. We can create pointer to any variable type as given in te below examples.

The general SYNTAX of a pointer variable declaration is as follows:-

datatype *pointervariable;

Examples:

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

Pointer Operators

Operator	Operator Name	Purpose
*	Value at address Operator	Gives Value stored at Particular address
&	Address of Operator	Gives Address of Variable

Pointer Address Operator

Pointer address operator is denoted by '&' symbol

When we use ampersand symbol as a prefix to a variable name '&', it gives the address of that variable.

Take an example

&n - It gives an address of variable n

Working Of Address Operator

Examples:

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

```
void main()
{
```

```
int n = 10;
printf("\nValue of n is : %d",n);
printf("\nAddress of n is : %u",&n);
}
```

Output :

```
Value of n is : 10 Address of n is : 1002
```

Explanation:

Consider the above example, where we have used to print the address of the variable using ampersand operator.

In order to print the variable we simply use name of variable while to print the address of the variable we use ampersand along with %u

```
printf("\nValue of &n is : %u",&n);
#include<stdio.h>
void main()
{
int n=20;
printf("The value of n is: %d",n);
printf("The address of n is: %u",&n);
printf("The value of n is: %d",*(&n));
}
```

OUTPUT:

```
The value of n is:20
The address of n is:1002
The value of n is:20
```

Explanation:

In the above program, first printf displays the value of n. The second printf displays the address of the variable n i.e) 1002, which is obtained by using &n(address of variable n). The last printf can be explained as follows,

*(&n) = *(Address of variable n)

=*(1002)

=Value at address 1002

Therefore   *(&n)=20

Understanding Address Operator

Initialization of Pointer can be done using following 4 Steps :

- Declare a Pointer Variable and Note down the Data Type.
- Declare another Variable with Same Data Type as that of Pointer Variable.
- Initialize Ordinary Variable and assign some value to it.
- Now initialize pointer by assigning the address of ordinary variable to pointer variable.

Below example will clearly explain the initialization of Pointer Variable.

```
#include<stdio.h>
int main()
{
int a;    // Step 1
 int *ptr; // Step 2
a = 10; // Step 3
ptr = &a; // Step 4
return(0);
}
```

Explanation of Above Program :

Pointer should not be used before initialization.

“ptr” is pointer variable used to store the address of the variable.

Stores address of the variable ‘a’ .

Now “ptr” will contain the address of the variable “a” .

Note : Pointers are always initialized before using it in the program

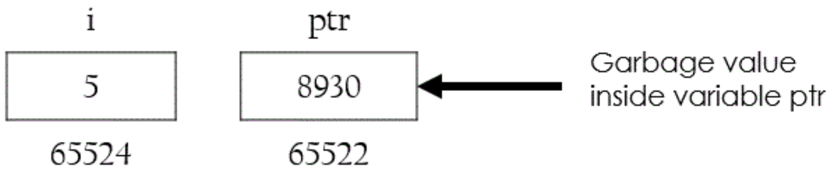
Consider the following program –

```
#include<stdio.h>
void main()
{
int i = 5;
int *ptr;
ptr = &i;
printf("\nAddress of i  : %u",&i);
printf("\nValue of ptr is : %u",ptr);
}
```

```
OUTPUT:
Address of i     : 65524
Value of ptr is : 65524
```

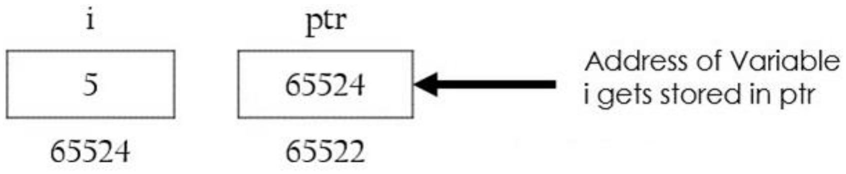
After declaration memory map will be like this

```
int i = 5;
int *ptr;
```



After assigning the address of variable to pointer, i.e after the execution of this statement

```
ptr = &i;
```



```
/* Program to display the contents of the variable and their address using pointer variable*/
#include<stdio.h>
main()
{
int i = 3, *j;
j = &i;
printf("\nAddress of i = %u", &i);
printf("\nAddress of i = %u", j);
printf("\nAddress of j = %u", &j);
printf("\nValue of j = %u", j);
printf("\nValue of i = %d", i);
printf("\nValue of i = %d", *(&i));
printf("\nValue of i = %d", *j);
}
```

```
Output :
Address of i = 65524
Address of i = 65524
Address of j = 65522
```

```
Value of j = 65524
Value of  i = 3
Value of  i = 3
Value of i = 3
```

Variable	Actual Value
Value of i	3
Value of j	65524
Address of i	65524
Address of j	65522

```
#include< stdio.h >
main()
{
int num, *intptr;
float x, *floptr;
char ch, *cptr;
num=123;
 x=12.34;
ch=’a’;
intptr=&num;
cptr=&ch;
floptr=&x;
printf(“Num %d stored at address %u\n”,*intptr,intptr);
printf(“Value %f stored at address %u\n”,*floptr,floptr);
printf(“Character %c stored at address %u\n”,*cptr,cptr);
}
```

Output :

```
Num 123 stored at address 1000
Value 12.34 stored at address 2000
Character a stored at address 3000
```

Pointer Expressions

Like any other variables pointer variables can be used in an expression. In general, expressions involving pointer conform to the same rules as other expressions. The pointer expression is a linear combination of pointer variables, variables and operators. Pointer expression gives either numerical output or address output.

Example:

```
y = *p1 * *p2;
sum = sum + *p1;
z = 5 - *p2/*p1;
*p2 = *p2 + 10;
```

```
/*Pointer expression and pointer arithmetic*/
#include< stdio.h >
void main()
{
int *ptr1,*ptr2;
int a,b,x,y;
a=30;
b=6;
ptr1=&a;
ptr2=&b;
x=*ptr1+ *ptr2 –b;
y=b - *ptr1/ *ptr2 +a;
printf("\nAddress of a %u",ptr1);
printf("\nAddress of b %u",ptr2);
printf("\na=%d, b=%d",a,b);
printf("\nx=%d,y=%d",x,y);
}
```

OUTPUT:

```
Address of a 65522
Address of b 65524
a=30 b=6
x=30 y=31
```

Explanation of Program:

In the above example program, ptr1, ptr2 are the pointer variables which are used to store the address of the two variables a and b respectively using the statements ptr1=&a, ptr2=&b. In the pointer expressions which are given below, the value of x and y are calculated as follows,

```
x=*ptr1+ *ptr2 - b;
```

=30 + 6 – 6 x=30

y=b - *ptr1/ *ptr2 +a;

=6 - 30/6 + 30

=6 – 5 + 30 y=31

Important Points about Pointers in C:

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

Pointer Assignment

We can use a pointer on the right-hand side of an assignment statement to assign its value to another pointer. For example,

```
#include< stdio.h >
void main()
{
int *p1,*p2;
int x=99;
p1=&x;
p2=p1; /*pointer assignment*/
printf("\nValues at p1 and p2: %d %d",*p1,*p2); /*print the value of x twice*/
printf("\nAddresses pointed to by p1 and p2: %u %u",p1,p2); /*print the address of x twice*/
}
```

OUTPUT:

```
Values at p1 and p2: 99 99
```

Addresses pointed to by p1 and p2: 5000 5000

Explanation of Program:

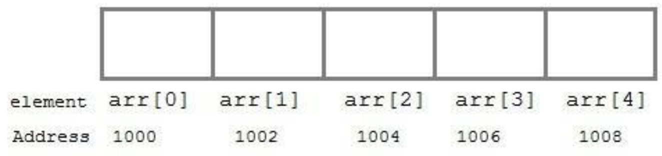
After the assignment sequence, p1=&x; p2=p1; Both p1and p2 point to x. Thus both p1 and p2 refer to the same value.

ARRAYS USING POINTER

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address gives location of the first element which 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 byte, the five element will be stored as follows



Here variable arr will give the base address, which is a constant pointer pointing to the element, arr[0]. Therefore arr is containing the address of arr[0] i.e 1000.

arr is equal to &arr[0] // by default

We can 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 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

Array name by itself is an address or pointer. It points to the address of the first element (0th element of an array). The elements of the array together with their address can be displayed by using array name itself. Array elements are always stored in contiguous memory locations.

A simple example to print the address of array elements

```
#include <stdio.h>
int main( )
{
  int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;
  for ( int i = 0 ; i < 7 ; i++ ) /* loop to print value and address of each element of array*/
  {
    printf("val[%d]: value is %d and address is %d\n", i, val[i], &val[i]);
  }
  return 0;
}
```

Output:

```
val[0]: value is 11 and address is 1423453232
val[1]: value is 22 and address is 1423453234
val[2]: value is 33 and address is 1423453236
val[3]: value is 44 and address is 1423453238
val[4]: value is 55 and address is 1423453240
val[5]: value is 66 and address is 1423453242
val[6]: value is 77 and address is 1423453244
```

There is a difference of 2 bytes between each element because that's the size of an integer. This means all the elements are stored in consecutive contiguous memory locations in the memory.

In the above example &val[i] is used to get the address of ith element of the array. We can also use a pointer variable instead of using the ampersand (&) to get the address.

Example – Array and Pointer

```
#include <stdio.h>
int main( )
{
  int *p; /*Pointer variable*/
  int val[7] = { 11, 22, 33, 44, 55, 66, 77 }; /*Array declaration*/
  p = var; /* Array name represents the address of the first element p = &val[0]; */
  for ( int i = 0 ; i<7 ; i++ )
  {
    printf("val[%d]: value is %d and address is %p\n", i, *p, p);
    p++; /* Incrementing the pointer so that it points to next element on every increment */
  }
  return 0;
}
```

Output:

```
val[0]: value is 11 and address is 0x7fff51472c30
```

```
val[1]: value is 22 and address is 0x7fff51472c32
val[2]: value is 33 and address is 0x7fff51472c34
val[3]: value is 44 and address is 0x7fff51472c36
val[4]: value is 55 and address is 0x7fff51472c38
val[5]: value is 66 and address is 0x7fff51472c40
val[6]: value is 77 and address is 0x7fff51472c42
```

- While using pointers with array, the data type of the pointer must match with the data type of the array.
- You can also use array name to initialize the pointer like p = var; because the array name alone is equivalent to the base address of the array (val==&val[0];)
- In the loop the increment operation (p++) is performed on the pointer variable to get the next location (next element's location), this arithmetic is same for all types of arrays (for all data types double, char, int etc.) even though the bytes consumed by each data type is different.

if p = &val[0] which means *p ==val[0]

(p+1) == &val[2] & *(p+1) == val[2]

(p+2) == &val[3] & *(p+2) == val[3]

(p+n) == &val[n+1) & *(p+n) == val[n+1]

Using this logic we can rewrite our code in a better way like this:

```
#include <stdio.h>
int main( )
{
  int *p;
  int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;
  p = val;
  for ( int i = 0 ; i<7 ; i++ )
  {
    printf("val[%d]: value is %d and address is %p\n", i, *(p+i), (p+i));
  }
  return 0;
}
```

We don't need the p++ statement in this program.

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

```
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++;
}
```

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 pointer and print all the values.

```
/*Program to print the addresses of array elements */
#include <stdio.h>
void main()
{
char c[4]; int i;
for(i=0;i<4;++i)
{
printf("Address of c[%d]=%x\n",i,&c[i]);
}
}
```

OUTPUT:

```
Address of c[0]=28ff44
Address of c[1]=28ff45
Address of c[2]=28ff46
Address of c[3]=28ff47
```

Notice, that there is equal difference (difference of 1 byte) between any two consecutive elements of array.

Consider the following:

```
int my_array[] = {1,23,17,4,-5,100};
```

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to my_array, i.e. using my_array[0] through my_array[5]. But, we could alternatively access them via a pointer as follows:

```
int *ptr;
ptr = &my_array[0];   /* pointer points to the first integer in our array */
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
#include <stdio.h>
int main(void)
{
int my_array[] = {1,23,17,4,-5,100};
int *ptr;
int i;
ptr = &my_array[0] ; /* point pointing to the first element of the array */ printf("\n\n");
for (i = 0; i < 6; i++)
{
printf("my_array[%d] = %d ",i,my_array[i]); /*<-- A */
printf("ptr + %d = %d\n",i, *(ptr + i));           /*<-- B */
}
return 0;
}
```

Compile and run the above program and carefully note lines A and B and that the program prints out the same values in either case. Also observe how we dereferenced our pointer in line B, i.e. we first added i to it and then dereferenced the new pointer. Change line B to read:

printf("ptr + %d = %d\n",i, *ptr++); and run it again.

then change it to: printf("ptr + %d = %d\n",i, *(++ptr)); and try once more. Each time try and predict the outcome and carefully look at the actual outcome.

In C, the standard states that wherever we might use &var_name[0] we can replace that with var_name, thus in our code where we wrote:

```
ptr = &my_array[0];
```

we can write:

ptr = my_array; to achieve the same result.

/* Example program to print the array elements using pointer */

Access Array Elements Using Pointers

```
#include <stdio.h>
int main()
{
    int data[5];
    printf("Enter elements: ");
    for (int i = 0; i < 5; ++i)
        scanf("%d", data + i);
    printf("You entered: \n");
    for (int i = 0; i < 5; ++i)
        printf("%d\n", *(data + i));
    return 0;
}
```

Output

```
Enter elements:
10
20
30
40
50
You entered:
10
20
30
40
50
```

```
/* Program to find sum of array elements using pointer */
#include<stdio.h>
#include<conio.h>
void main()
{
int numArray[10];
int i, sum = 0;
int *ptr;
printf("\nEnter 10 elements : ");
for (i = 0; i < 10; i++)
scanf("%d", &numArray[i]);
ptr = numArray;
for (i = 0; i < 10; i++)
{
sum = sum + *ptr; ptr++;
}
printf("The sum of array elements :
%d", sum);
}
```

OUTPUT

```
Enter 10 elements : 11 12 13 14 15 16 17 18 19 20
The sum of array elements is 155
```

Explanation of Program:

Accept the 10 elements from the user in the array.

```
for (i = 0; i < 10; i++)
scanf("%d", &numArray[i]);
```

We are storing the address of the array into the pointer.

```
ptr = numArray;          /* a=&a[0] */
```

Now in the for loop we are fetching the value from the location pointer by pointer variable. Using De-referencing pointer we are able to get the value at address.

```
for (i = 0; i < 10; i++)
{
sum = sum + *ptr; ptr++;
}
```

Suppose we have 2000 as starting address of the array. Then in the first loop we are fetching the value at 2000.

i.e sum = sum + (value at 2000)

= 0 + 11

= 11

In the Second iteration we will have following calculation –

sum = sum + (value at 2002)

= 11 + 12

= 23

Pointer example-1

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

```
main()
{
int num [ ] = { 10,20,30,40,50 };
print ( &num, 5, num);}print ( int *j, int n, int b[5]){int i;
for(i=0;i<=4;i++)
{
printf ( " %u %d %d %u \n ", &j[i]  ,  *j , *(b+i)  ,  &b); j++;
}
}
```

In this example we have a single dimensional array num and a function print . We are passing, the address to the first element of the array, the number of elements and the array itself, to this function. When the function receives this arguments, it maps the first one to another pointer j and the array num is copied into another array b . (The type declarations are made here itself. Note that these declarations can also be given just below this line). j is now a pointer to the array b.

Inside the function we are printing out the address of the array element and the value of the array element in two ways. One using the pointer j and the other using the array b. If we compile and run this code we get the following output,

```
3221223408 10 10 3221223376
3221223416 20 20 3221223376
3221223424 30 30 3221223376
3221223432 40 40 3221223376
3221223440 50 50 3221223376
```

Note that as we increment j it points to the successive elements of the array. We can get both the address of the array elements and the value stored there using this. However the array name, which acts also as the pointer to its base address, is not able to give us the address of its elements. Or in other words, the array name is a constant pointer. Also note that while j is points to the elements of the array num, b is pointing to its copy.

Next we have an example that uses a two dimensional array. Here care should be taken to declare the number of columns correctly.

Pointer example-2

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

```
int arr [ ][3] = {{11,12,13}, {21,22,23},{31,32,33},{41,42,43},{51,52,53}};
int i, j ;
int *p , (*q) [3], *r ; p = (int *) arr ;
q = arr;
r = (int *) q ;
printf ( " %u %u %d %d %d  %d \n  ", p ,  q  ,  *p , *(r)  ,  *(r+1),  *(r+2)); p++ ;
q++ ;
r = (int *) q ;
printf ( " %u %u %d %d %d %d \n ", p , q , *p , *(r) , *(r+1), *(r+2));
}
```

Here we have a pointer p and a pointer array q. The first assignment statement is to make the pointer p points to the array arr. While assigning, we also declare the type of the variable arr. Note that variables on both side of this statement should have the same type. Next line is a similar statement, now with q and arr. Since q is a pointer array, the array can be directly assigned to it and there is no need for specifying the type of the variable. In the next line we make the pointer r to point to the pointer array q . Then we will print out the different values. Here is what we get from this,

```
3221223344  3221223344  11 11 12 13
3221223348  3221223356  12 21 22 23
```

Here we see that incrementing p make it just jump through each element of the array, where as incrementing q, will move it from one row to another row.

ARRAY OF POINTERS

Just like array of integers or characters, there can be array of pointers too. An array of pointers can be declared as :

```
<datatype> *<pointername> [number-of-elements];
```

For example : char *ptr[3];

The above line declares an array of three character pointers. Let’s take a working example :

```
#include<stdio.h>
int main(void)
{
char *p1 = "Himanshu";
char *p2 = "Arora";
char *p3 = "India";
char *arr[3];
arr[0] = p1;
arr[1] = p2;
arr[2] = p3;
printf("\n p1 = [%s] \n",p1);
printf("\n p2 = [%s] \n",p2);
printf("\n p3 = [%s] \n",p3);
printf("\n arr[0] = [%s] \n",arr[0]);
printf("\n arr[1] = [%s] \n",arr[1]);
printf("\n arr[2] = [%s] \n",arr[2]);
return 0;
}
```

In the above code, we took three pointers pointing to three strings. Then we declared an array that can contain three pointers. We assigned the pointers 'p1′, 'p2′ and 'p3′ to the 0,1 and 2 index of array.

Let's see the output :

```
p1 = [Himanshu]
p2 = [Arora]
p3 = [India]
arr[0] = [Himanshu]
arr[1] = [Arora]
arr[2] = [India]
```

So we see that array now holds the address of strings.

Let us consider the following example, which makes use of an array of 3 integers:

```
int main ()
{
int var[] = {10, 100, 200};
int i;
for (i = 0; i < 3; i++)
{
printf("Value of var[%d] = %d\n", i, var[i] );
}
return 0;
}
```

OUTPUT:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:

```
int *ptr[3];
```

This declares ptr as an array of 3 integer pointers. Thus, each element in ptr, now holds a pointer to an int value.

Following example makes use of three integers, which will be stored in an array of pointers as follows:

```
#include <stdio.h>
int main ()
{
int var[] = {10, 100, 200};
int i, *ptr[3];
ptr[0] = &var[0]; /* assign the address of 1st integer element */
ptr[1] = &var[1]; /* assign the address of 2nd integer element */
ptr[2] = &var[2]; /* assign the address of 3rd integer element */
for ( i = 0; i < 3; i++)
{
printf("Value of var[%d] = %d\n", i, *ptr[i] );
}
return 0;
}
```

OUTPUT:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

STRUCTURE USING POINTERS

Syntax:

```
struct name {
   member1;
   member2;
```

```
    .
    .
};
int main()
{
    struct name *ptr, Harry;
}
```

Here, ptr is a pointer to struct.

Example: Access members using Pointer

To access members of a structure using pointers, we use the -> operator.

```
#include <stdio.h>
struct person
{
   int age;
   float weight;
};
int main()
{
    struct person *personPtr, person1;
    personPtr = &person1;
    printf("Enter age: ");
    scanf("%d", &personPtr->age);
    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);
    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);
    return 0;
}
```

In this example, the address of person1 is stored in the personPtr pointer using personPtr = &person1;.

Now, you can access the members of person1 using the personPtr pointer.

By the way,

personPtr->age is equivalent to (*personPtr).age personPtr->weight is equivalent to (*personPtr).weight

Accessing Structure Members with Pointer

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

```
struct Book
{
char name[10];
int price;
}
int main()
{
struct Book b;
struct Book * ptr = &b;
ptr->name = "Dan Brown";    //Accessing Structure Members
ptr->price = 500;
}
```

Example program for C structure using pointer:

In this program, “record1″ is normal structure variable and “ptr” is pointer structure variable. As you know, Dot(.) operator is used to access the data using normal structure variable and arrow(->) is used to access data using pointer variable.

```
#include <stdio.h>
#include <string.h>
struct student
{
int id;
char name[30];
float percentage;
};
int main()
{
int i;
struct student record1 = {1, "Raju", 90.5};
struct student *ptr;
ptr = &record1;
printf("Records of STUDENT1: \n");
printf("Id is: %d \n", ptr->id);
printf("Name is: %s \n", ptr->name);
printf("Percentage is: %f \n\n", ptr->percentage);
return 0;
}
```

POINTER AND FUNCTION

Function Pointer

```
#include <stdio.h>
void subtractAndPrint(int x, int y);
void subtractAndPrint(int x, int y)
{
int z = x - y;
printf("Simon says, the answer is: %d\n", z);
}
int main()
{
void (*sapPtr)(int, int) = subtractAndPrint;
(*sapPtr)(10, 2);
sapPtr(10, 2);
}
```

The pointer can be used as an argument in functions. The arguments or parameters to the function are passed in two ways.

- Call by value
- Call by reference

Call by Value:

This method copies the value of actual parameter into the formal parameter of the function. The changes of the formal parameters cannot affect the actual parameters, because formal arguments are photocopy of the actual argument.

The changes made in formal argument are local to the block of the called functions. Once control return back to the calling function the changes made disappear.

Example:

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

```
#include<conio.h>
void cube(int);
int cube1(int);
void main()
{
int a; clrscr();
printf("Enter one values");
scanf("%d",&a);
printf("Value of cube function is=%d", cube(a));
printf("Value of cube1 function is =%d", cube1(a ));
getch();
}
void cube(int x)
{
x=x*x*x;
return x;
}
int cube1(int x)
{
x=x*x*x;
return x;
}
```

Output:

```
Enter one values 3
Value of cube function is 27
Value of cube1 function is 27
```

Call by reference

- Call by reference is another way of passing parameter to the function.
- Here the address of argument are copied into the parameter inside the function, the address is used to access arguments used in the call.
- Hence changes made in the arguments are permanent.
- Here pointer are passed to function, just like any other arguments. Example:-

```
#include<stdio.h>
#include<conio.h>
void swap(int,int);
void main()
{
```

```
int a=5,b=10;
clrscr();
printf("Before swapping a=%d b=%d",a,b);
swap(&a,&b);
printf("After swapping a=%d b=%d",a,b);
getch();
}
void swap(int *x,int *y)
{
int *t;
t=*x;
*x=*y;
*y=t;
}
```

Output:

```
Before swapping a=5 b=10
After swapping a=10 b=5
```

Function Returning Pointer

A function can return a single value by its name or return multiple values through pointer parameters. Since pointers are a data type in c, we can also force a function to return a pointer to the calling function.

Program:

```
int *larger(int*,int*); void main()
{
int a=10;
int b=20;
int *p;
p =larger(&a,&b);
printf("%d",*p);
}
int *larger(int *x, int *y)
{
if(*x>*y)
return(x);
else
return (y);
}
```

Output: 20

DYNAMIC MEMORY ALLOCATION IN C

The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

Static Memory Allocation	**Dynamic Memory Allocation**
Memory is allocated at compile time.	Memory is allocated at run time.
Memory can't be increased while executing program.	Memory can be increased while executing program.
Used in array.	Used in linked list.

Various methods used for dynamic memory allocation.

malloc()	Allocates single block of requested memory.
calloc()	Allocates multiple block of requested memory.
realloc()	Reallocates the memory occupied by malloc() or calloc() functions.
free()	Frees the dynamically allocated memory.

malloc() function:

- The malloc() function allocates single block of requested memory.
- It doesn't initialize memory at execution time, so it has garbage value initially.
- It returns NULL if memory is not sufficient.
- The syntax of malloc() function is ptr=(cast-type*)malloc(byte-size)
- **Example:** ptr = (int*) malloc(100 * sizeof(int));
- Since the size of int is 2 bytes, this statement will allocate 200 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

Example Program of malloc() function

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int n,i,*ptr,sum=0;
```

```
	printf("Enter number of elements: ");
	scanf("%d",&n);
	ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
	if(ptr==NULL)
	{
		printf("Sorry! unable to allocate memory");
		exit(0);
	}
	printf("Enter elements of array: ");
	for(i=0;i<n;++i)
	{
		scanf("%d",ptr+i);
		sum+=*(ptr+i);
	}
	printf("Sum=%d",sum);
	free(ptr);
	return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements of array:
10
10
10
Sum=30
```

calloc() function:

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.
- The syntax of calloc() function is ptr=(cast-type*)calloc(number, byte-size)
- **Example:** ptr = (float*) calloc(25, sizeof(float));
- This statement allocates contiguous space in memory for 25 elements each with the size of the float.

Example Program of calloc() function

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
        int n,i,*ptr,sum=0;
        printf("Enter number of elements: ");
        scanf("%d",&n);
        ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
        if(ptr==NULL)
        {
                printf("Sorry! unable to allocate memory");
                exit(0);
        }
        printf("Enter elements of array: ");
        for(i=0;i<n;++i)
        {
                scanf("%d",ptr+i);
                sum+=*(ptr+i);
        }
        printf("Sum=%d",sum);
        free(ptr);
        return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements of array:
10
10
10
Sum=30
```

realloc() function:

- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

- The syntax of realloc() function is ptr=realloc(ptr, new-size)

Example Program of realloc() function

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int *ptr, i , n1, n2;
   printf("Enter size: ");
   scanf("%d", &n1);
   ptr = (int*) malloc(n1 * sizeof(int));
   printf("Addresses of previously allocated memory: ");
   for(i = 0; i < n1; ++i)
        printf("%u\n",ptr + i);
   printf("\nEnter the new size: ");
   scanf("%d", &n2);
   // rellocating the memory
   ptr = realloc(ptr, n2 * sizeof(int));
   printf("Addresses of newly allocated memory: ");
   for(i = 0; i < n2; ++i)
        printf("%u\n", ptr + i);
   free(ptr);
   return 0;
}
```

Output

```
Enter size: 2
Addresses of previously allocated memory:
26855472
26855474
Enter the new size: 4
Addresses of newly allocated memory:
26855472
26855474
26855476
26855478
```

free() function:

- The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.
- The syntax of free() function is free(ptr)

STORAGE CLASSES IN C

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

- Automatic
- External
- Static
- Register

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

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.
- The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

Example 1

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

```
{
int a; //auto
char b;
float c;
printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
return 0;
}
```

Output:

```
garbage garbage garbage
```

Example 2

```
#include <stdio.h>
int main()
{
int a = 10,i;
printf("%d ",++a);
{
int a = 20;
for (i=0;i<3;i++)
{
printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
}
}
printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
}
```

Output:

```
11 20 20 20 11
```

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.

- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

Example 1

```
#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
}
```

Output:

```
0 0 0.000000 (null)
```

Example 2

```
#include<stdio.h>
void sum()
{
static int a = 10;
static int b = 24;
printf("%d %d \n",a,b);
a++;
b++;
}
void main()
{
int i;
for(i = 0; i< 3; i++)
{
sum(); // The static variables holds their value between multiple function calls.
}
}
```

Output:

```
10 24
11 25
12 26
```

Register

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We cannot dereference the register variables, i.e., we cannot use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.
- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler's choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables cannot be stored into the register since we cannot use more than one storage specifier for the same variable.

Example 1

```
#include <stdio.h>
int main()
{
register int a; // variable a is allocated memory in the CPU register. The initial default value o
f a is 0.
printf("%d",a);
}
```

Output:

```
0
```

Example 2

```
#include <stdio.h>
int main()
{
register int a = 0;
printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.
}
```

Output:

```
main.c:5:5: error: address of register variable ?a? requested
printf("%u",&a);
```

External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally, i.e., we cannot initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

```
#include <stdio.h>
int main()
{
extern int a;
printf("%d",a);
}
```

Output

```
main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status
```

Example 2

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

```
extern int a; // variable a is defined globally, the memory will not be allocated to a
printf("%d",a);
}
```

Output

```
0
```

Example 3

```
#include <stdio.h>
int a;
int main()
{
extern int a = 0; // this will show a compiler error since we can not use extern and initializer a
t same time
printf("%d",a);
}
```

Output

```
compile time error
main.c: In function ?main?:
main.c:5:16: error: ?a? has both ?extern? and initializer
extern int a = 0;
```

Example 4

```
#include <stdio.h>
int main()
{
extern int a; // Compiler will search here for a variable a defined and initialized somewhere in
 the pogram or not.
printf("%d",a);
}
int a = 20;
```

Output

```
20
```

Example 5

```
extern int a;
int a = 10;
#include <stdio.h>
```

```
int main()
{
printf("%d",a);
}
int a = 20; // compiler will show an error at this line
```

Output

```
Compile time error
```

Additional Examples of Storage Classes

Auto:

eg:1

```
#include<stdio.h>
void main()
{
auto num = 20 ;
{
auto num = 60;
printf("Num : %d",num);
}
printf("Num : %d",num);
}
```

Output :

```
Num : 60
Num : 20
```

Two variables are declared in different blocks, so they are treated as different variables.

eg:2

```
#include<stdio.h>
void increment(void);
void main()
{
increment();
increment();
increment();
increment();
}
```

```
void increment(void)
{
auto int i = 0;
printf ( “%d”, i) ;
i++;
}
```

Output:

```
0 0 0 0
```

Extern

eg:1

```
#include<stdio.h>
int num = 75 ;
void display();
void main()
{
extern int num ;
printf("Num : %d",num);
display();
}
void display()
{
extern int num ;
printf("Num : %d",num);
}
```

Output :

```
Num : 75
Num : 75
```

Declaration within the function indicates that the function uses external variable. Functions belonging to same source code do not require declaration (no need to write extern). If variable is defined outside the source code, then declaration using extern keyword is required.

eg:2

```
#include<stdio.h>
int x = 10 ;
void main( )
```

```
{
extern int y;
printf("The value of x is %d \n",x);
printf("The value of y is %d",y);
}
int y=50;
```

Output:

```
The value of x is 10
The value of y is 50
```

Static

eg:1

```
#include<stdio.h>
void Check();
int main()
{
Check();
Check();
Check();
}
void Check()
{
static int c=0;
printf("%d\t",c);
c+=5;
}
```

Output

```
0       5       10
```

During first function call, it will display 0. Then, during second function call, variable c will not be initialized to 0 again, as it is static variable. So, 5 is displayed in second function call and 10 in third call. If variable c had been automatic variable, the output would have been:

```
0       0       0
```

eg:2

```
#include<stdio.h>
void increment(void);
```

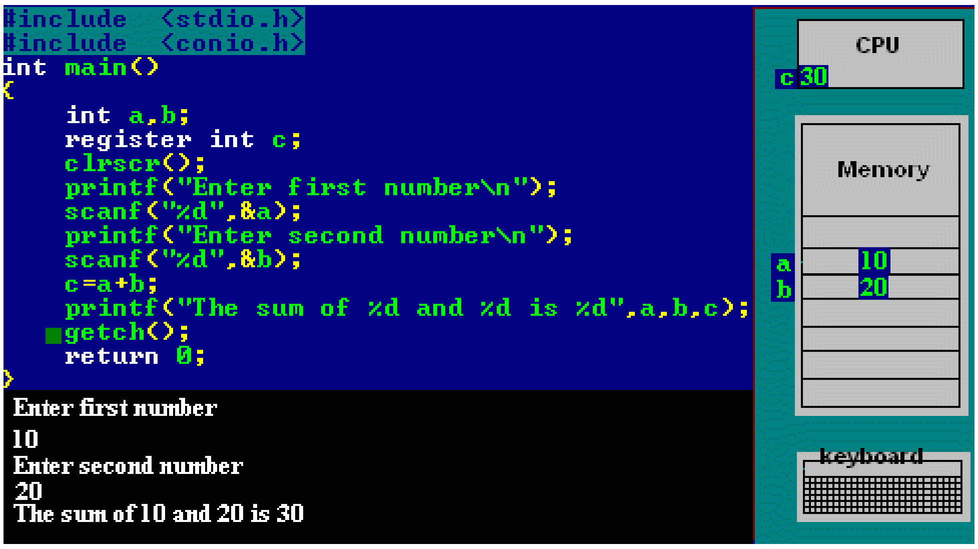
```
int main()
{
increment();
increment();
increment();
increment();
return 0;
}
void increment(void)
{
static int i = 0;
printf ( “%d”, i);
i++;
}
```

Output: 0 1 2 3

Register

eg:1

```
#include<stdio.h>
void main()
{
int num1,num2;
register int sum;
printf("\nEnter the Number 1 : ");
scanf("%d",&num1);
printf("\nEnter the Number 2 : ");
scanf("%d",&num2);
sum = num1 + num2;
printf("\nSum of Numbers : %d",sum);
}
```


```
#include  <stdio.h>
#include  <conio.h>
int main()
{
    int a,b;
    register int c;
    clrscr();
    printf("Enter first number\n");
    scanf("%d",&a);
    printf("Enter second number\n");
    scanf("%d",&b);
    c=a+b;
    printf("The sum of %d and %d is %d",a,b,c);
    getch();
    return 0;
}
```

```
Enter first number
10
Enter second number
20
The sum of 10 and 20 is 30
```

eg:2

```
#include <stdio.h>
int main()
{
register int i;
int arr[5];// declaring array
arr[0] = 10;// Initializing array
arr[1] = 20;
arr[2] = 30;
arr[3] = 40;
arr[4] = 50;
for (i=0;i<5;i++)
{
// Accessing each variable
printf("value of arr[%d] is %d \n", i, arr[i]);
}
return 0;
}
```

Output:

```
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
```

value of arr[3] is 40

value of arr[4] is 50

FILE HANDLING IN C:

A file is a place on the disk where a group of related data is stored. C supports a number of functions to perform basic file operations, which include

- Naming a file
- Creating a new file or Opening an existing file
- Reading data from a file
- Writing data to a file and
- Closing a file

Functions for file handling

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

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

Defining a file

If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system. They include

- File Name
- Data Structure
- Purpose

File name is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with extension

Examples: Input.data, PROG.C, Student.c, Text.out

Data structure of a file is defined as FILE in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. FILE is a defined data type. When we open a file, we must specify what we want to do with the file. Example, we may write data to the file or read the already existing data.

Example:

FILE *fp;

Opening a file:

The general format is

FILE *fp;

fp=fopen(“filename”, “mode”);

The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program. The second statement also specifies the purpose of opening the file.

We can use one of the following modes in the fopen() function.

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

The fopen function works in the following way.

- Firstly, it searches the file to be opened.
- Then, it loads the file from the disk and places it into the buffer. The buffer is used to provide efficiency for the read operations.

- It sets up a character pointer which points to the first character of the file.

Consider the following statements:

FILE *p1, *p2;

p1=fopen(“data”, “r”);

p2=fopen(“results”, “w”);

the file data is opened for reading and results is opened for writing. In case the results file already exists, its contents are deleted and the files are opened as a new file. If data file does not exist error will occur

Closing a file:

The input output library supports the function to close a file; it is in the following format.

fclose(file_pointer);

A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer.

Observe the following program.

….

FILE *p1 *p2;

p1=fopen (“Input”,”w”);

p2=fopen (“Output”,”r”);

….

… fclose(p1); fclose(p2)

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.

Input/Output Operations on Files

Following functions are used in input/output operations on files.

- getc and putc
- getw and putw
- fprintf and fscanf

getc() and putc() functions:

The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time. The putc function writes the character contained in character variable c to the file associated with the pointer fp1.

putc(c,fp1);

similarly getc function is used to read a character from a file that has been open in read mode.

c=getc(fp2).

The program shown below displays use of a file operations. The data enter through the keyboard and the program writes it. Character by character, to the file input. The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *f1;
char c;
clrscr();
printf("Data input output");
f1=fopen("Input.txt","w"); /*Open the file Input*/
while((c=getchar())!=EOF) /*get a character from key board*/
putc(c,f1); /*write a character to input*/ fclose(f1); /*close the file input*/ printf("\nData output\n");
f1=fopen("INPUT.txt","r"); /*Reopen the file input*/
while((c=getc(f1))!=EOF)
printf("%c",c); fclose(f1); getch();
}
```

Output

```
Data input output
LALITHA THENMOZHI
^Z
Data output LALITHA THENMOZHI
```

The getw and putw functions:

These are integer-oriented functions. They are similar to get c and putc functions and are used to read and write integer values. These functions would be usefull when we deal with only integer data. The general forms of getw and putw are:

```
putw(integer,fp);
getw(fp);
```

```
/*Example program for using getw and putw functions*/
#include<stdio.h>
```

```
#include<conio.h>
void main()
{
FILE *f1; int c,i; clrscr();
f1=fopen("Input1.txt","w");
for(i=0;i<5;i++)
{ scanf("%d",&c); putw(c,f1);
}
fclose(f1);
f1=fopen("Input1.txt","r");
while((c=getw(f1))!=EOF)
printf("%d\n",c);
fclose(f1);
getch();
}
```

Output

```
23
23
45
56
56

23
23
45
56
56
```

The fprintf & fscanf functions:

The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files. The first argument of theses functions is a file pointer which specifies the file to be used. The general form of fprintf is

fprintf(fp,”control string”, list);

Where fp id a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

fprintf(f1,%s%d%f”,name,age,7.5);

Here name is an array variable of type char and age is an int variable The general format of fscanf is

fscanf(fp,”controlstring”,list);

This statement would cause the reading of items in the control string.

Example:

fscanf(f2,”5s%d”,item,&quantity”);

Like scanf, fscanf also returns the number of items that are successfully read.

```
/*Program to handle mixed data types*/
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
int number,i; char item[10]; clrscr();
fp=fopen("input2.txt","w");
fscanf(stdin,"%s%d",item,&number);
fprintf(fp,"%s%d",item,number);
fclose (fp);
fp=fopen("input2.txt","r");
fprintf(stdout,"%s%d",item,number);
fclose(fp);
getch();
}
```

Output

```
Pencil 10
Pencil 10
```

Random access to files:

Sometimes it is required to access only a particular part of the and not the complete file. This can be accomplished by using the following function:

fseek function:

The general format of fseek function is a s follows:

fseek(file pointer,offset, position);

This function is used to move the file position to a desired location within the file. Fileptr is a pointer to the file concerned. Offset is a number or variable of type long, and position in an integer number. Offset specifies the number of positions (bytes) to be moved from the location specified bt the position. The position can take the 3 values.

Value	Meaning
0	Beginning of the file
1	Current position
2	End of the file.

The offset may be positive or negative. Positive means move forward, negative means move backward.

Example

Statement	Meaning
fseek(fp,0L,0);	Go to the beginning (similar to rewind)
fseek(fp,0L,1);	Stay at the current position. (Rarely used)
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end(mth character from the end).

When the operation is successful, fseek returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and fseek returns -1 (minus one). It is good practice to check whether an error has occurred or not, before proceeding further.

Ftell funtion

ftell takes a file pointer and returns a number of type long,that corresponds to the

current position. This function is useful in saving the current position of a file. n=ftell(fp);
n would give the relative offset(in bytes)of the current position. This means that n bytes have already been read(or written).

Example of fseek and ftell

```
#include<stdio.h>
#include<conio.h>
#include<stdio.h>
void main()
{
FILE *fp;
long n;
char c;
clrscr();
fp=fopen("g1.txt","w");
while((c=getchar())!=EOF)
putc(c,fp);
printf("No. of characters entered=%ld\n",ftell(fp));
fclose(fp);
fp=fopen("g1.txt","r");
n=0L;
while(feof(fp)==0)
{
fseek(fp,n,0);
printf("Position of %c is %ld\n",getc(fp),ftell(fp));
n=n+5L;
}
getch();
}
```

```
Output: ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
No. Of characters entered =26
Position of A is 0
Position of F is 5
Position of K is 5
Position of P is 15
```

Position of U is 20

Position of Z is 25

Position of   is 30

Explanation

During the first reading, the file pointer crosses the end-of-file mark when the parameter n of fseek(fp,n,0) becomes 30. Therefore, after printing the content of position 30, the loop is terminated. (There is nothing in the position 30)

For reading the file from the end, we use the statement

fseek(fp,-1L,2)

to position the file pointer to the last character. Since every read causes the position to move forward by one position, we have to move it back by two positions to read the next character. This achieved by the function.

fseek(fp,-2L,1);

in the while statement. This statement also tests whether the file pointer has crossed the file boundary or not. The loop is terminated as soon as it crosses it.

Rewind Funtion

rewind takes a file pointer and resets the position to the start of the file.

rewind(fp)

n=ftell(fp) would assign 0 to n because the file position has been set to the start of the file by rewind. Remember, the first byte in the file is numbered as 0, second as 1, and so on. This function helps us in reading a file more than once, without having to close and open the file. Remember that whenever a file is opened for reading or writing, a rewind is done implicitly.

Binary files

Binary files are very similar to arrays of structures. Binary files have two features that distinguish them from text files:

- We can instantly use any structure in the file.
- We can change the contents of a structure anywhere in the file.

After opened the binary file, we can read and write a structure or seek a specific position in the file. A file position indicator points to record 0 when the file is opened. A read operation reads the structure where the file position indicator is pointing to. After reading the structure the pointer is moved to point at the next structure. A write operation will write to the currently pointed-to structure. After the write operation the file position indicator is moved to point at the next structure. The fseek function will move the file position indicator to the record that is requested. The file position indicator can not only point at the beginning of a structure, but can also point to the file.

Fread () and fwrite() funtions

The fread and fwrite function takes four parameters:

- A memory address
- Number of bytes to read per block
- Number of blocks to read
- A file variable

Example of ‘write’:

```
#include<stdio.h>
#include<conio.h>
struct rec
{
int x;
};
void main()
{
int i; FILE *fp;
struct rec my1; clrscr(); fp=fopen("test.txt","w"); for (i=1; i <= 10;i++)
{
my1.x= i;
fwrite(&my1, sizeof(struct rec), 1,fp);
}
fclose(fp);
fp=fopen("test.txt","r");
for (i=1; i <= 10;i++)
{
fread(&my1, sizeof(struct rec), 1,fp);
printf(“%d\n”,my1.x);
}
fclose(fp);
getch()
}
```

Output

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

In this example we declare a structure rec with the members x, y and z of the type integer. In the main function we open (fopen) a file for writing (w). Then we check if the file is open, if not, an error message is displayed and we exit the program. In the "for loop" we fill the structure member x with a number. Then we write the record to the file. We do this ten times, thus creating ten records. After writing the ten records, we will close the file.

SCSA1202 PROGRAMMING WITH C AND C++

UNIT3 –OBJECT ORIENTED PROGRAMMING CONCEPTS 9Hrs

Introduction-Procedure vs. object oriented programming-Concepts: Classes and Objects-Operator and Function Overloading-Inheritance-Polymorphism and Virtual Functions.

Procedure-oriented Programming

- The high level languages, such as BASIC, COBOL, C, FORTRAN are commonly known as Procedure Oriented Programming (POP).
- In Procedure Oriented Programming approach, the problem is viewed in sequence of things to be done such as reading, calculating and printing.
- To carry out these tasks the function concept is used i.e., a number of functions are written to accomplish these tasks.

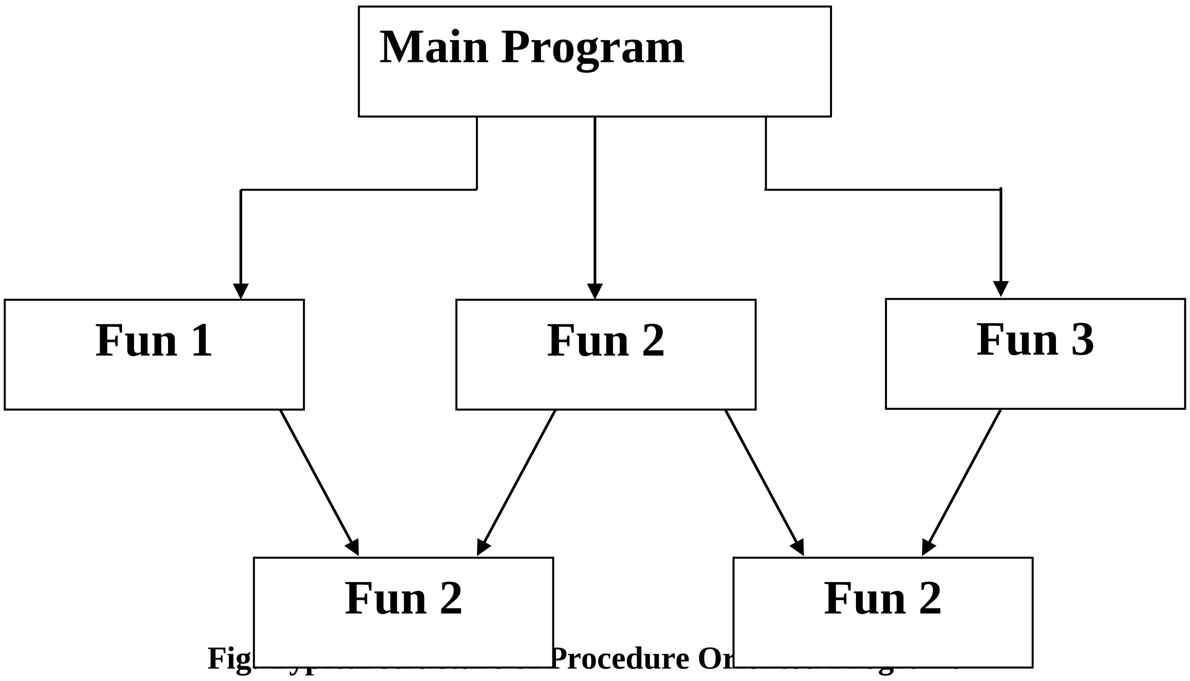


Fig Procedure Or

- The function concept basically consists of number of statements and these statements are organized or grouped into functions.

- While developing these functions the programmer must care about the data that is being used in various functions.
- A multi-function program, the data must be declared as global, so that data can be accessed by all the functions within the program & each function can also have its own data called local data.

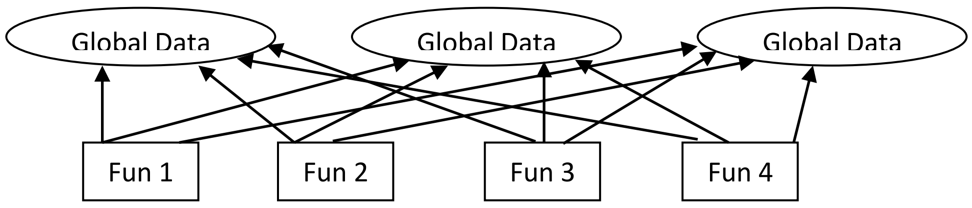


Fig. Relationship of data and functions in procedural programming

- The global data can be accessed anywhere in the program.
- In large program it is very difficult to identify what data is accessed by which function.
- In this case we must revised about the external data and as well as the functions that access the global data. At this situation there is so many chances for an error.

✓ **Characteristics of Procedure-oriented Programming (C):**

- Emphasis is on doing things (algorithms)
- Larger programs are divided into smaller programs known functions.
- Most of the functions share global data.
- Data move openly around the system from function to functions
- Functions transforms data from one form to another
- Employs TOP-DOWN approach in program design.

Obect-oriented programming

- This programming approach is developed to reduce the some of the drawbacks encountered in the Procedure Oriented Programming approach.

- The OOProgramming approach treats data as critical element and does not allow the data to flow freely around the program.
- It bundles the data more closely to the functions that operate on it; it also protects data from the accidental modification from outside the function.
- The object oriented programming approach divides the program into number of entities called objects and builds the data and functions that operates on data around the objects.
- The data of an object can only access by the functions associated with that object.
- However the functions of one object can access the functions of other objects.

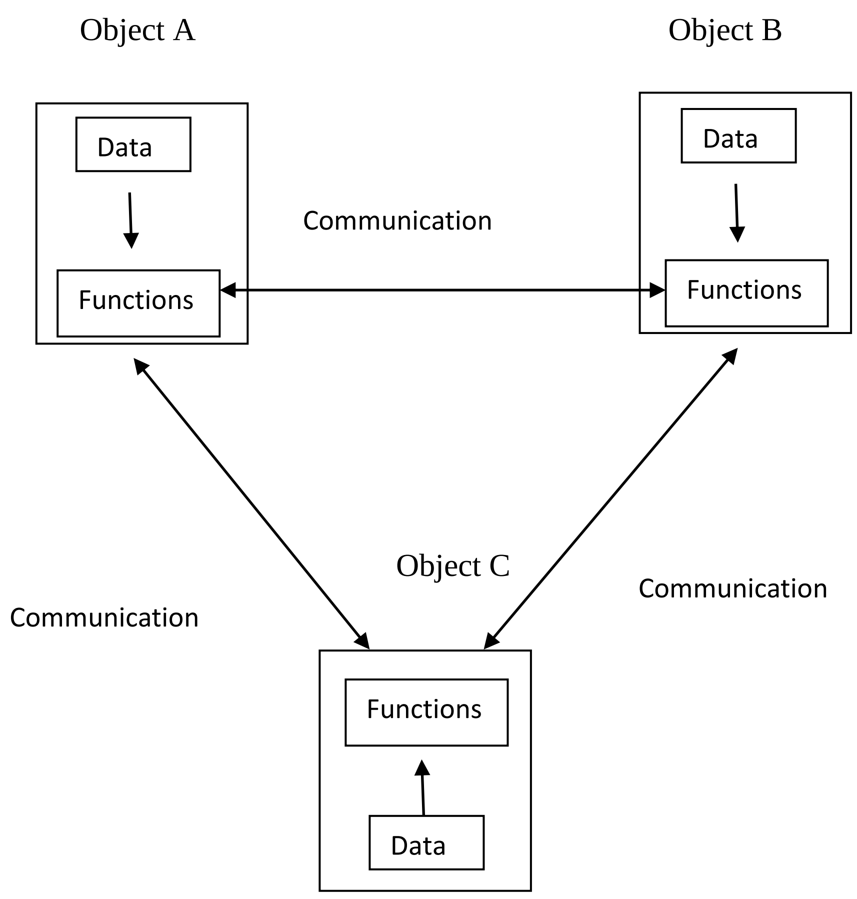


Fig. Organisation of data and functions in OOP

✓ **Characteristics of Object-oriented Programming (C++)**

- Emphasis is on data rather than procedure.
- Programs are divided into what known as OBJECTS
- Data structures are designed such that they characterize the objects
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions
- New data and functions can be easily added whenever necessary
- Follows BOTTOM-UP approach in program design.

❖ *Difference between C & C++*

C is an procedure oriented programming language, whereas C++ is an object oriented programming language .The need for an object oriented environment is -procedure oriented programming language such as C fails to show the desired results for larger programs inters of bug free, complexity & reusable programs. C++ eliminates the pitfalls faced by C language.

There are many differences which makes C++ as an apt alternative to C. There are many concepts in C++ which C language doesn't support. They are data hiding, data encapsulation, inheritance, polymorphism, classes, objects, operator overloading & message passing.These concepts can eliminate the complexity of C for larger programs.

The main difference between C & C++ is where we use the class (user defined data type), Through class we derive the other concepts. Class is similar to structure in C. In structure only mixed data type can be declared, whereas C++ allows us to declare & define functions in addition to the data's. This makes the main function more simple & grouping all functions into a single class.

Operator overloading concept in C++ makes us to create a new language of or own. Through this we can perform arithmetic operations on variables of class i.e. objects which makes the program easier to understand. In C we can’t perform arithmetic operation on structure variable as we do in C++.

In c for larger programs the presence of redundant codes is unavoidable. This can be eliminated in C++ by following inheritance.

In C there may be accidental invading of codes of one program by another program. This can be eliminated in C++ which supports data hiding which helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

In C, if we want to add additional features to the program without modifying the function is very difficult. This can be eliminated in C++ which supports inheritance where we can add additional features to an existing class without modifying it. This concept provides the idea of reusability of the programs.

S.No	**Procedure oriented Programming (C)**	**Object Oriented Programming (C++)**
1.	Programs are divided into smaller sub-programs known as functions	Programs are divides into objects & classes
2.	Here global data is shared by most of the functions	Objects are easily communicated with each other through functions.
3.	It is a Top-Down Approach	It is a Bottom-Up Approach
4.	Data cannot be secured and available to all the functions.	Data can be secured and can be available in the class in which it is

		declared.
5.	Here, the reusability is not possible, hence redundant code cannot be avoided.	Here, we can reuse the existing one using the Inheritance concept

❖ Basics of C++ Programming

C++ was developed by BJARNE STROUSSTRUP at AT&T BELL Laboratories in Murry Hill, USA in early 1980's.

STROUSSTRUP combines the features of 'C' language and 'SIMULA67' to create more powerful language that support OOPS concepts, and that language was named as "C with CLASSES". In late 1983, the name got changed to C++.

The idea of C++ comes from 'C' language increment operator (++) means more additions.

C++ is the superset of 'C' language, most of the 'C' language features can also applied to C++, but the object oriented features (Classes, Inheritance, Polymorphism, Overloading) makes the C++ truly as Object Oriented Programming language.

OOPs (Object Oriented Programming System) concepts

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use Function overloading and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Classes/Objects

C++ is an object-oriented programming language.

Everything in C++ is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an **object**. The car has **attributes**, such as weight and color, and **methods**, such as drive and brake.

Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as "class members".

A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the class keyword:

Example

Create a class called "MyClass":

```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};
```

Example explained

- The class keyword is used to create a class called MyClass.
- The public keyword is an **access specifier**, which specifies that members (attributes and methods) of the class are accessible from outside the class. You will learn more about access specifiers later.
- Inside the class, there is an integer variable myNum and a string variable myString. When variables are declared within a class, they are called **attributes**.
- At last, end the class definition with a semicolon ;.

Create an Object

In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name.

To access the class attributes (myNum and myString), use the dot syntax (.) on the object:

Example

Create an object called "myObj" and access the attributes:

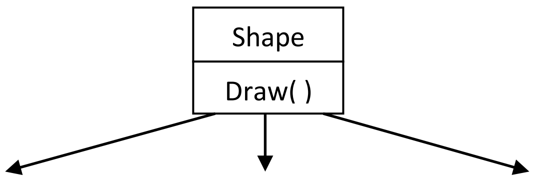
```
class MyClass {       // The class
  public:             // Access specifier
    int myNum;        // Attribute (int variable)
    string myString;  // Attribute (string variable)
};

int main() {
  MyClass myObj;  // Create an object of MyClass

  // Access attributes and set values
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;
  return 0;
}
```

- Polymorphism is a key to the power of OOPs.
- It is a Greek word, means the ability to take more than one form(*many forms)*.
- It is the concept that supports the capability of data to be processed in more than one form i.e., It allows a single name to be used for more than one related purpose, which are technically different.
- For **example**, an operation may exhibit different behavior in different instances.
- The behavior depends upon the types of data used in the operation.

The following are the different ways of achieving polymorphism in a C++ program:

- Function overloading
- Operator overloading
- Dynamic binding

Example:



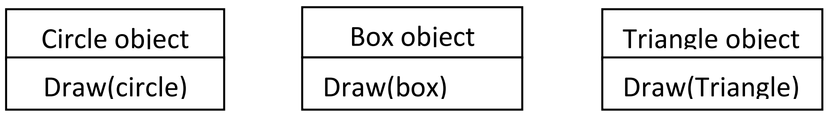


Fig. Polymorphism

Fig. illustrates that a single function name can be used to handle different number and different types of arguments. This is similar to a particular word having several different meanings depending on the context. Using a single function name to perform different types of tasks is known as *function overloading*.

- Let us consider the operation of addition.
    - For two numbers, the operation (+) will generate a sum.
    - If the operands are strings then the operation (+) would produce a third string by concatenation. This is known as *operator overloading*.

➢ **Dynamic binding**

- Binding refers to the linking of a procedure call to the code to be executed in response to the call.
- Dynamic binding (also known as *late binding*) means that the code associated with a given procedure is not known until the time of the call at run-time.
- It is associated with *polymorphism* and *inheritance.*

FUNCTION OVERLOADING:-

We can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in oop. Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

Example:-

Declarations:-

1. int add (int a, int b);

2. int add (int a, int b, int c);
3. double add (double x, double y);
4. double add (int p, double q);
5. double add (double p, int q);

Function calls:-

```
cout << add (0.75, 5);     // uses 5
cout << add (5, 10);       // 1
cout << add (15, 10.0); // 4
cout << add (12.5, 7.5); // 3
cout << add (5, 10.15); //2
```

```
#include<iostream.h>
class funoverloading
  {
        int a, b, c;
        public:
            void add ( )
               {
                    cin>>a>>b;
                    c = a + b;
                    cout << c;
                }
             int add (int a, int b);
             {
                   c = a + b;
                   return c;
             }
             void add (int a)
             {
                  cin>>b;
                  c = a + b;
                  cout<<c;
              }
  }
```

```
}
void main ( )
{
    funoverloading F;
    int x;
    F. add ( );
    x = F. add (10, 5);
    cout<<x;
    F. add (5);
}
```

Output:-

```
    5      3
    c =   8
    c = 15
    5
    c = 10
```

<u>OPERATOR OVERLOADING:-</u>

C ++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

The process of overloading involves the following steps:

- Create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op()in the public part of the class
- It may be either a member function or a friend function
- Define the operator function to implement the required operations.

Syntax:-

returntype classname:: operator op (argument list)

```
        {
                Function body
        }.
```

Example:-

```
        void space:: operator-( )
        {
                x=-x;
        }
```

OVERLOADING UNARY OPERATORS:-

Let us consider the unary minus operator. A minus operator when used as a unary takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an int or float variable. The unary minus when applied to an object should change the sign of each of its data items.

```
#include<iostream.h>
{
        int x;
        int y;
        int z;
        public:
                void getdata(int a, int b, int c);
                void display(void);
                void operator-(); //overload unary minus
};
void space::getdata(int a, int b, int c)
{
        x=a;
        y=b;
        z=c;
}
void  space::display(void)
{
        cout<<x<<” ”;
        cout<<y<<” “;
        cout<<z<<” “;
}
void space::operator-()
{
        x=-x;
        y=-y;
        z=-z;
}
int main()
{
        space S;
        S.getdata(10,-20,30);
        cout<<”S=”;
        S.display();
        - S;
        cout<<”S=”;
        S.display();
        return 0;
}
```

Output:-

```
        S= 10 -20 30
        S=  -10 20 -30
```

Note:

The function operator-() takes no argument. Then, what does this operator function do? It changes the sign of data members of the object S. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

OVERLOADING BINARY OPERATORS:-

The same mechanism which is used in overloading unary operator can be used to overload a binary operator.

```
# include<iostream.h>
class complex
{
        float x;
        float y;
        public:
                complex()
                {
                }
                complex(float real, float imag)
                {
                        x=real;
                        y=imag;
                }
                complex operator+(complex)
                void display(void);
};
complex complex::operator+(complex c)
{
        complex temp;
        temp.x=x+c.x;
        temp.y=y+c.x;
        return(temp);
}
void complex::display(void)
{
        cout<<x<<”j”<<y<<”\n”;
}
int main()
{
        complex C1,C2,C3;
        C1=complex(2.5,3.5)
        C2=complex(1.6,2.7)
        C3= C1 + C2;
        cout<<"C1 = ";
        C1.display();
        cout<<"C2 = ";
        C2.display();
        cout<<"C3 = ";
        C3.display();
        return 0;
}
```

Output:-

```
        C1=2.5 +j3.5
        C2=1.6 + j2.7
        C3= 4.1 +j6.2
```

INHERITANCE

The mechanism of deriving a new class from an old one is called inheritance(or derivation). The old class is referred to as the base class and the new one is called the derived class. The derived class with only one base class is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance. The following figure shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance.

DEFINITION:

The mechanism of deriving the new class from an old one is called inheritance. The old class is called as base class and the new class is called as derived class.

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **derived class** (child) - the class that inherits from another class

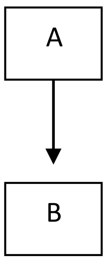
- **base class** (parent) - the class being inherited from

To inherit from a class, use the : symbol.

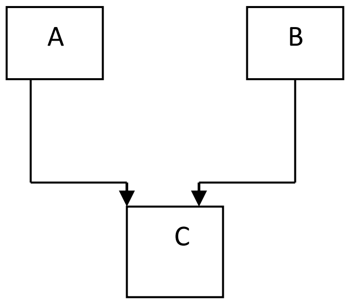
Types of Inheritance:

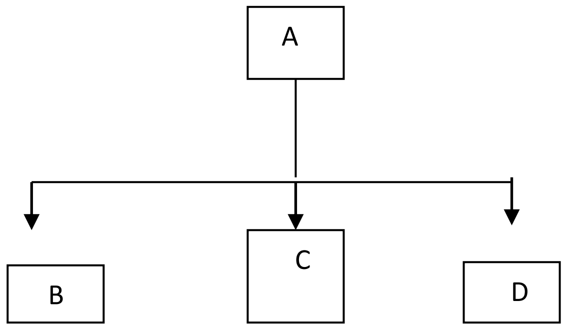
1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance(refer Assignment)
5. Hybrid inheritance

Single Inheritance



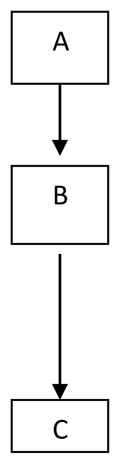
Multiple Inheritance



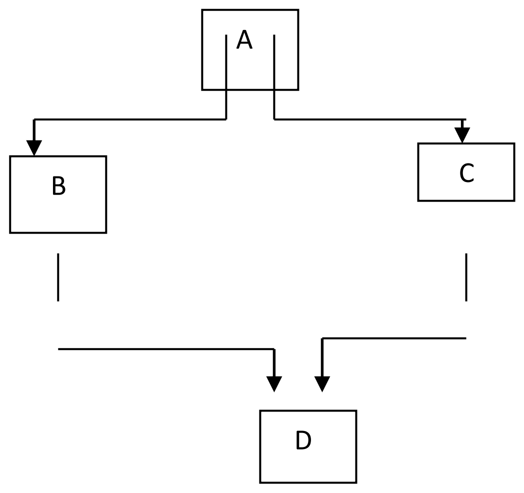


Hierarchical Inheritance

Multilevel Inheritance



Hybrid Inheritance



Defining derived classes:-

A derived class is defined by specifying its relationship with the base class in addition to its own details.

The general form:-

```
class derived-class-name: visibly-mode base-class-name
{
        ------------//
        ------------//members of derived class
        ------------
};
```

The colon indicates that the derived-class-name is derived from the base-class-name. The visibility mode is optional and, if present, may be either private or public. The default visibility-mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

Examples:-

```
class ABC:private XYZ//private derivation
{
	members of ABC
};
class ABC:public XYZ//public derivation
{
	members of ABC
};
class ABC:XYZ//private derivation by default
{
	members of ABC
};
```

When base class is privately inherited by a derived class, ’public members’ of the base class become ‘private members’ of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. Remember, a public member of a class be accessed by its own objects using the dot operator. The result is that no member of the base class is accessible to the objects of the derived class.

When the base class is publicly inherited, ‘public members’ of the base class become ‘public members’ of the derived class and therefore they are accessible to the objects of the derived class. In both the cases, the private members are not inherited and therefore, the private members of the base class will never become the members of its derived class.

1.SINGLE INHERITANCE:-

The new class can be derived from only one base class is called single inheritance.

Let us consider a simple example to illustrate inheritance. The following program shows the base class B and a derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.

```
#include<iostream.h>
class B
{
        int a;//private; not inheritable
        public:
                int b;//public; ready for inheritance
                void get_ab()
                {
                        a=5;
                        b=10;
                }
                int get_a(void)
                {
                        return a;
                }
                void show_a(void)
                {
                        cout<<”a=”<<a<<endl;
                }
};
class D:public B//public derivation
{
        int c;
        public:
                void mul(void)
                {
                        c=b*get_ab();
                }
                void display()
                {
                        cout<<"a="<<get_ab()<<endl;
                        cout<<"b="<<b<<endl;
                        cout<<"c="<<c<<endl;
                }
};
void main()
{
        D d;
        d.get_ab();
        d.mul();
        d.show_a();
        d.display();
        d.b=20;
        d.mul();
        d.display();
}
```

Output:-

```
        a=5
        a=5
        b=10
        c=50

        a=5
        b=20
        c=100
```

The class D is a public derivation of the class B. Therefore, D inherits all the public members of B and retains their visibility. Thus a public member of the base class B is also a public member of the derived class D. The private members of B cannot be inherited by D. The program illustrates that the objects of class D have access to all the public members of B. Let us have a look at the functions show_a() and mul().

```
        void show_a()
        {
                cout<<"a="<<a<<endl;
        }
        void mul()
        {
                c=b*get_a();//c=b*a
```

Although the data member a is private in B and cannot be inherited, objects of D are able to access it through an inherited member function of B.

Let us now consider the case of private derivation.

```
            #include<iostream.h>
    class B
    {
            int a;
            public:
                    int b;
                    void get_ab();
                    int get_a(void);
                    void show_a(void);
    };
    class D:private B
    {
            int c;
            public:
                    void mul(void);
                    void display();
    };
```

In private derivation, the public members of the base class become private members of derived class. Therefore, the objects of D can not have direct access to the public member functions of B.The statement such as d.get_ab(),d.get_a(),d.show_a() will not work.

Program:-

```
    #include<iostream.h>
    class B
    {
          int a;//private; not inheritable
          public:
                    int b;//public; ready for inheritance
                    void get_ab()
                    {
                              a=5;
                              b=10;
                    }
                    int get_a(void)
                    {
                              return a;
                    }
                    void show_a(void)
                    {
                              cout<<"a="<<a<<endl;
                    }
};
class D:private B//private derivation
{
          int c;
          public:
                    void mul(void)
                    {
                              c=b*get_ab();
                    }
                    void display()
                    {
                              cout<<"a="<<get_ab()<<endl;
                              cout<<"b="<<b<<endl;
                              cout<<"c="<<c<<endl;
                    }
};
void main()
```

```
{
        D d;
        // d.get_ab(); it won’t work
        d.mul();
        //d.show_a(); won’t work
        d.display();
        //d.b=20; won’t work
        d.mul();
        d.display();
}
```

Output:-

```
a=5
b=10
c=50

a=12
b=20
c=240
```

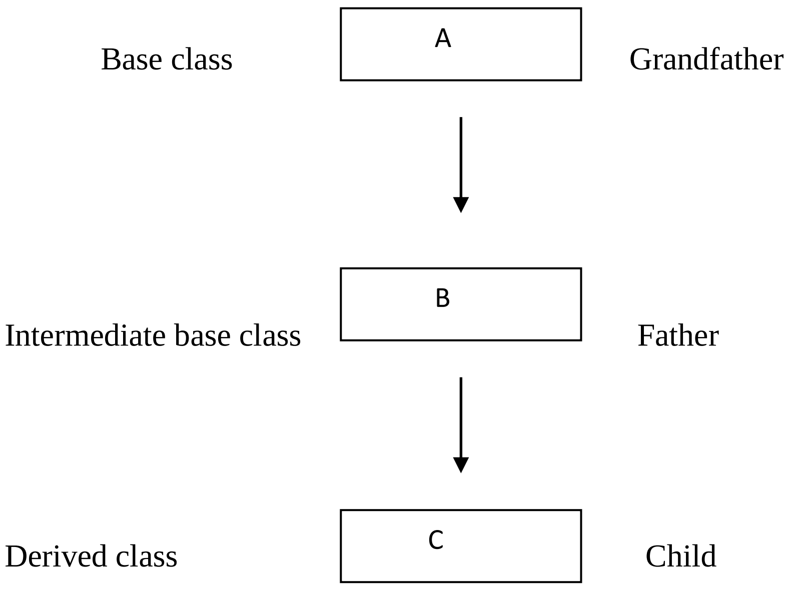
A private member of a base class cannot be inherited and therefore it is not available for derived class directly. A protected which serve a limited purpose in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

```
class alpha
{
        private:
                -------  //optional
                -------  // visible to member functions
                ------- // within its class
        protected:
                ----------  // visible to member functions
                ----------  // of its own and derived class
        public:
                ----------  // visible to all functions
                ---------- //in the program
   }
```

When a protected member is inherited in public mode, it becomes protected in the derived class too, and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A protected member, inherited in the private mode derivation, becomes private in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance.

2.Multilevel Inheritance:-

The class **A** serves as a base class for the derived class **B** which in turn serves as a base class for the derived class **C.** The chain **ABC** is known as inheritance path.



A derived class with multilevel inheritance is declared as follows.

class A(….) ;

class B:public A(…..);

class C:public B(…..);

The multilevel inheritance is defined as, the new class is derived from another derived class. Here the class A serves as a base class for the derived class B which in turn serves as a base class for the derived class C. The derived class is defined as,

```
Class A
{
        //body of base class A
};
Class B : public A
{
        //body of intermediate base class B
};
Class C : public B
{
        //body of derived class
};
```

Example Program

EXAMPLE :1

```
#include <iostream.h>
class student
{
        protected:
                int rollno:
        public:
                void getroll(int x)
                {
        rollno = x;
}
};
void test: public student
{
        protected:
                float sub1, sub2;
        public:
                void getmart(int y, int z)
                {
                sub1 = y; sub2= z;
                }
};
void result : public test
{
        float total;
        public:
        void display( )         {
total =  sub1 + sub2;
cout<<rollno<<sub1<<sub2;
cout<<total;
                }
};
void main( )
{
        result s;
        s.getroll(101);
        s.getmark(90, 100);
        s.display( );
```

EXAMPLE :2

```
#include<iostream.h>
class student
{
        protected:
                int roll_number;
        public:
                void get_number(int);
                {
                        roll_number=a;
                }
                void put_number(void)
                {
                        cout<<"Roll Number:"<<roll_number<<endl;
                }
};
class test: public student  //First level derivation
{
        protected:
                float sub1;
                float sub2;
        public:
                void get_marks(float,float)
                {
                        sub1=x;
                        sub2=y;
                }
                void put_marks(void)
                {
                        cout<<"Marks in sub1="<<sub1<<endl;
                        cout<<"Marks in sub2="<<sub2<<endl;
                }
};
class result : public test // second level derivation
{
        float total;
        public:
                void display(void)
                {
                        total=sub1+sub2;
                        put_number();
                        put_marks();
                        cout<<"Total="<<total<<"\n";
```

```
			}
);
void main()
{
	result student1;
	student1.get_number(111);
	student1.get_marks(75.0,59.5);
	student1.display();
}
```

Output:-

```
	Roll Number:111
	Marks in sub1:75
	Marks in sub2:59.5
	Total:134.5
```

<u>**3.MULTIPLE INHERITANCE:-**</u>

A class can inherit the attributes or properties of two or more classes that is a new class can be derived from more than one base class is called multiple inheritance.

Here the class A , class B and class C serves as a base class for the derived class D

The derived class definition syntax is:

```
class d: visibility base-1, visibility base 2,…
{
	body of class D;
};
```

<u>Example:</u>

```
#include<iostream.h>
class A
{
 protected:
	int rollno;
public:
	void getroll(int x)
```

```
            {
            rollno=x;
            }
};
class B
{
protected:
            int sub1,sub2;
public:
            void getmark(int y,int z)\
            {
            sub1=y;
            sub2=z;
            }
};
class C : public A, public B
{
            int total;
            public:
            void display()
            {
            total=sub1+sub2;
            cout<<”roll no:”<<rollno<<”sub1:”<<sub1<<”sub2:”<<sub2;
            cout<<”total”<<total;
            }
};
void main()
{
            C s;
            s. getroll(435);
            s.getmark(100,90);
            s.display();
}
```

Run:

```
Roll no : 435
Sub1: 100
Sub2: 90
```

<u>4.HIERARCHICAL INHERITANCE:-</u>

The hierarchical inheritance structure is given below .This type is helpful when we have class hierarchies. In this scheme the base class will include all the features that are common to the subclasses.

```
#include<iostream.h>
#include<string.h>
class A
{
protected:
int x, y;
public:
void get ( )
{
cout<<”Enter two values”<<endl;
cin>> x>>y;
}
};
class B : public A
{
private:
int m;
public:
void add( )
{
m= x + y;
cout<<”The Sum is “<<m;
}
};
```

```
class C : public A
{
private:
int n;
public:
void mul( )
{
n= x * y;
cout << “The Product is “<<n;
}
};
class D : public A
{
private:
float l;
public:
void division( )
{
l = x / y;
cout <<”The Quotient is “<< l;
}
};
void main( )
{
B obj1;
C obj2;
D obj3;
B .get( );
B .add( );
C .get( );
C .mul( );
D .get( );
D .division( );
}
```

Output of the Program

```
Enter two values
    12 6
The Sum is 18
The Product is 72
The Quotient is 2.0
```

<u>5.HYBRID INHERITANCE:-</u>

Hybrid inheritance = multiple inheritance + multi-level inheritance

The hybrid inheritance is defined as a combination of multilevel and multiple inheritances. This new class can be derived by either multilevel or multiple method or both.

Example program

In this program the derived class (result) object will be inheriting the properties of both test class and sports class.

```
#include <iostream.h>
class student
{
        protected:
                int rollno;
        public:
                void getroll (int x)
                {
                        rollno = x;
                }
};
class test: public student
{
protected:
        int sub1, sub2'
public:
        void getmark (int y, int z)
{
        sub1 = y;  sub2 = z;
}
};
class  sports
{
protected:
        int score;
public:
        void  getscore (int a )
        {
        score=a;
        }
};
class result: public test, public sports
{
        int total;
public:
        void display()
        {
        total= sub1+sub2+score;
        cout<<"rollno:"<<rollno<< "total:"<<"Score:"<<total;
        }
};
void main( )
{
        result S;
        s.getroll(101);
        s.getmark(90,98);
        s.getscore(2000);
        s. display( );
}
```

Run:

```
Rollno: 101
```

Total: 188

Score: 2000

VIRTUAL FUNCTION

When we use the same function name in base and derived classes, the function in the base classes is declared as virtual using keyword ‘virtual’ preceding its normal declaration.

The member function that can be changed at runtime is called virtual function.

```
Class classname
{
        public:
        . . . . .
        virtual returntype functionname (arguments)
        {
                . . . . .
                . . . . .}};
```

The virtual function should be defined in the public section of a class. When one function is made virtual and another one is normal, and compiler determines which function to call at runtime. In this we have to create a ‘basepointeer’.

If the basepointer paints to a base class object means it will execute the baseclass function.

If the basepointer points to a derived class object means it will execute the derived class function.

Example Program:

```
#include <iostream.h>
Class A
{
        public:
        void displayA( )
        {
                cout<<”Welcome”;
        }
        virtual void show( )
        {
                cout<<”Hello”;
        }
};
class B: public A
{
        public:
        void display( )
        {
                cout<<”Good Morning”;
        }
        void show( )
        {
                cout<<”Hai”;
        }
};
void main( )
{
        A S1;
        B S2;
        A *bptr;
        bptr = &S1;
        bptr→show( );           //Calls base class show
        bptr=&S2( );
        bptr→show( );           //Calls derived class show
        S2. displayA( );        //Calls base class displayA
        S2.display( );          //Calls derived class display
}
```

RUN:

```
Hello
Hai
Welcome
Good Morning
```

Explanation:

The base pointer first holds the address of the base class object means it will run the base class member function display and show will be executed. Next the base pointer points to the derived class object, in this before executing the derived class function if will check whether the corresponding base class function is ‘virtual’ or not, if it is not virtual then execute the base class function otherwise it will execute the derived class function.

Rules for virtual function:

1. Virtual function must be a members of some class.
2. They cannot be static members.
3. They are accessed by using base pointer.
4. A virtual function in a base class must be defined eventhough it may not be used.
5. A prototype (or declaration) of a base class virtual function and all the derived class version must be identical.
6. We cannot we a pointer to a derived class to access an object of the base class.

Pure virtual Function:

The virtual function doesn’t have any operation is called “do-nothing” function or pure virtual function.

This is represented as

virtual void display()=0;

SCSA1202 PROGRAMMING WITH C AND C++

UNIT5 –TEMPLATES AND EXCEPTION HANDLING 9Hrs

Function Templates and Class Templates – Name spaces – Standard Template Library - Casting – Exception Handling –case study.

TEMPLATES:

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

FUNCTION TEMPLATES

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

However, a better approach would be to use function templates because you can perform the same task writing less and maintainable code.

A function template starts with the keyword template followed by template parameter/s inside < > which is followed by function declaration.

Declaration of Function Template:

```
template <class T>

T someFunction(T arg)

{

   ... .. ...

}
```

In the above code, T is a template argument that accepts different data types (int, float), and class is a keyword.

You can also use keyword typename instead of class in the above example.

When, an argument of a data type is passed to someFunction( ), compiler generates a new version of someFunction() for the given data type.

Example 1: Function Template to find the largest number

```
#include <iostream.h>

// template function

template <class T>

T Large(T n1, T n2)

{

        return (n1 > n2) ? n1 : n2;

}

int main()

{

        int i1, i2;

        float f1, f2;

        char c1, c2;

        cout << "Enter two integers:\n";

        cin >> i1 >> i2;

        cout << Large(i1, i2) <<" is larger." << endl;

        cout << "\nEnter two floating-point numbers:\n";

        cin >> f1 >> f2;

        cout << Large(f1, f2) <<" is larger." << endl;

        cout << "\nEnter two characters:\n";

        cin >> c1 >> c2;

        cout << Large(c1, c2) << " has larger ASCII value.";

        return 0;

}
```

Output:

```
Enter two integers:
```

```
5
10
10 is larger.
Enter two floating-point numbers:
12.4
10.2
12.4 is larger.
Enter two characters:
z
Z
z has larger ASCII value.
```

In the above program, a function template Large() is defined that accepts two arguments n1 and n2 of data type T. T signifies that argument can be of any data type.

Large() function returns the largest among the two arguments using a simple conditional operation.

Inside the main() function, variables of three different data types: int, float and char are declared. The variables are then passed to the Large() function template as normal functions.

During run-time, when an integer is passed to the template function, compiler knows it has to generate a Large() function to accept the int arguments and does so.

Similarly, when floating-point data and char data are passed, it knows the argument data types and generates the Large() function accordingly.

This way, using only a single function template replaced three identical normal functions and made your code maintainable.

Example 2: Swap Data Using Function Templates

Program to swap data using function templates.

```
#include <iostream.h>
template <typename T>
void Swap(T &n1, T &n2)
{
      T temp;
      temp = n1;
      n1 = n2;
      n2 = temp;
}
int main()
{
      int i1 = 1, i2 = 2;
      float f1 = 1.1, f2 = 2.2;
      char c1 = 'a', c2 = 'b';
      cout << "Before passing data to function template.\n";
      cout << "i1 = " << i1 << "\ni2 = " << i2;
      cout << "\nf1 = " << f1 << "\nf2 = " << f2;
      cout << "\nc1 = " << c1 << "\nc2 = " << c2;
      Swap(i1, i2);
      Swap(f1, f2);
      Swap(c1, c2);
    cout << "\n\nAfter passing data to function template.\n";
      cout << "i1 = " << i1 << "\ni2 = " << i2;
      cout << "\nf1 = " << f1 << "\nf2 = " << f2;
      cout << "\nc1 = " << c1 << "\nc2 = " << c2;
      return 0;
}
```

Output:

```
Before passing data to function template.
i1 = 1
i2 = 2
f1 = 1.1
f2 = 2.2
c1 = a
```

```
c2 = b
```

After passing data to function template.

```
i1 = 2

i2 = 1

f1 = 2.2

f2 = 1.1

c1 = b

c2 = a
```

In this program, instead of calling a function by passing a value, a call by reference is issued.

The Swap() function template takes two arguments and swaps them by reference.

CLASS TEMPLATES

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Normally, you would need to create a different class for each data type OR create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.

However, class templates make it easy to reuse the same code for all data types.

Declaration class template :

```
class className

{


public:

   T var;

   T someOperation(T arg);

   ... .. ...

};
```

In the above declaration, T is the template argument which is a placeholder for the data type used.

Inside the class body, a member variable var and a member function someOperation() are both of type T.

creation of class template object:

To create a class template object, you need to define the data type inside a < > when creation.

```
className<dataType> classObject;
```

For example:

```
className<int> classObject;

className<float> classObject;

className<string> classObject;
```

Example 3: Simple calculator using Class template

Program to add, subtract, multiply and divide two numbers using class template

```
#include <iostream.h>

template <class T>

class Calculator

{

private:

        T num1, num2;

public:

        Calculator(T n1, T n2)

        {

                num1 = n1;

                num2 = n2;

        }

        void displayResult()

        {

                cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;

                cout << "Addition is: " << add() << endl;

                cout << "Subtraction is: " << subtract() << endl;

                cout << "Product is: " << multiply() << endl;
                cout << "Division is: " << divide() << endl;
        }

        T add() { return num1 + num2; }
        T subtract() { return num1 - num2; }
        T multiply() { return num1 * num2; }
        T divide() { return num1 / num2; }
};
int main()
{
        Calculator<int> intCalc(2, 1);
        Calculator<float> floatCalc(2.4, 1.2);
        cout << "Int results:" << endl;
        intCalc.displayResult();
        cout << endl << "Float results:" << endl;
        floatCalc.displayResult();
        return 0;
}
```

Output

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2
Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
```

```
Division is: 2
```

In the above program, a class template Calculator is declared.

The class contains two private members of type T: num1 & num2, and a constructor to initalize the members.

It also contains public member functions to calculate the addition, subtraction, multiplication and division of the numbers which return the value of data type defined by the user.

Likewise, a function displayResult() to display the final output to the screen.

In the main() function, two different Calculator objects intCalc and floatCalc are created for data types: int and float respectively. The values are initialized using the constructor.

Notice we use <int> and <float> while creating the objects. These tell the compiler the data type used for the class creation.

This creates a class definition each for int and float, which are then used accordingly.

Then, displayResult() of both objects is called which performs the Calculator operations and displays the output.

NAMESPACES:

Namespace is a container for identifiers. It puts the names of its members in a distinct space so that they don't conflict with the names in other namespaces or global namespace.

Creating a Namespace

Creating a namespace is similar to creation of a class.

```
namespace MySpace
{
   // declarations
}

int main()
{
   // main function
}
```

This will create a new namespace called MySpace, inside which we can put our member declarations.

Rules to create Namespaces:

The namespace definition must be done at global scope, or nested inside another namespace.

Namespace definition doesn't terminates with a semicolon like in class definition.

You can use an alias name for your namespace name, for ease of use.

Example for Alias:

```
namespace Study
{
    void study();
    class Learn
    {
        // class defintion
    };
}

// St is now alias for Study
namespace St = Study
```

You cannot create instance of namespace.

There can be unnamed namespaces too. Unnamed namespace is unique for each translation unit. They act exactly like named namespaces.

Example for Unnamed namespace:

```
namespace
{
    class Head
    {
        // class defintion
    };
    // another class
    class Tail
    {
        // class defintion
    };
    int i,j,k;
}

int main()
{
    // main function
}
```

A namespace definition can be continued and extended over multiple files, they are not redefined or overriden.

For example, below is some header1.h header file, where we define a namespace:

```
namespace MySpace
{
    int x;
    void f();
}
```

We can then include the header1.h header file in some other header2.h header file and add more to an existing namespace:

```
#include "header1.h";
namespace MySpace
{
    int y;
    void g();
}
```

Using a Namespace in C++

There are three ways to use a namespace in program,

- Scope resolution operator (::)
- The using directive
- The using declaration

With Scope resolution operator (::)

Any name (identifier) declared in a namespace can be explicitly specified using the namespace's name and the scope resolution :: operator with the identifier.

```
namespace MySpace
{
    class A
    {
        static int i;
        public:
        void f();
    };
    // class name declaration
    class B;
    //gobal function declaration
    void func();
}
// Initializing static class variable
int MySpace::A::i=9;
class MySpace::B
{
    int x;
    public:
    int getdata()
    {
        cout << x;
    }
    // Constructor declaration
    B();
}

// Constructor definition
MySpace::B::B()
{
    x=0;
}
```

Using Directive:

using keyword allows you to import an entire namespace into your program with a global scope. It can be used to import a namespace into another namespace or any program.

Conside a header file Namespace1.h:

```
namespace X
{
    int x;
    class Check
    {
        int i;
    };
}
```

Including the above namespace header file in Namespace2.h file:

```
include "Namespace1.h";
namespace Y
{
    using namespace X;
    Check obj;
    int y;
}
```

We imported the namespace X into namespace Y, hence class Check will now be available in the namespace Y.

Hence we can write the following program in a separate file, let's say program1.cpp

```
#include "Namespace2.h";
void test()
{
    using Namespace Y;
    // creating object of class Check
    Check obj2;
}
```

Hence, the using directive makes it a lot easier to use namespace, wherever you want.

The using declaration

When we use using directive, we import all the names in the namespace and they are available throughout the program, that is they have a global scope.

But with using declaration, we import one specific name at a time which is available only inside the current scope.

NOTE: The name imported with using declaration can override the name imported with using directive

Consider a file Namespace.h:

```
namespace X
{
   void f()
   {
      cout << "f of X namespace\n";
   }
   void g()
   {
      cout << "g of X namespace\n";
   }
}

namespace Y
{
   void f()
   {
      cout << "f of Y namespace\n";
   }
   void g()
   {
      cout << "g of Y namespace\n";
   }
}
```

Now let's create a new program file with name program2.cpp with below code:

```
#include "Namespace.h";
```

```
void h()
{
    using namespace X;  // using directive
    using Y::f; // using declaration
    f();    // calls f() of Y namespace
    X::f(); // class f() of X namespace
}
```

Output:

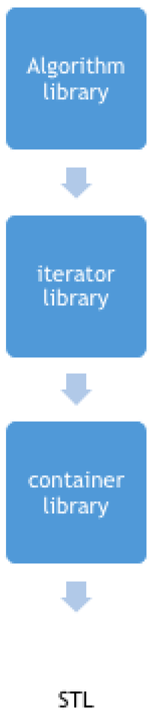
```
f of Y namespace
f of X namespace
```

In using declaration, we never mention the argument list of a function while importing it, hence if a namespace has overloaded function, it will lead to ambiguity.

STANDARD TEMPLATE LIBRARY:

STL is an acronym for standard template library. It is a set of C++ template classes that provide generic classes and function that can be used to implement data structures and algorithms .STL is mainly composed of :

1. Algorithms
2. Containers
3. Iterators



STL provides numerous containers and algorithms which are very useful in completive programming , for example you can very easily define a linked list in a single statement by using list container of container library in STL , saving your time and effort.

STL is a generic library , i.e a same container or algorithm can be operated on any data types , you don’t have to define the same algorithm for different type of elements.

For example , sort algorithm will sort the elements in the given range irrespective of their data type , we don’t have to implement different sort algorithm for different datatypes.

C++: Algorithms in STL

STL provide number of algorithms that can be used of any container, irrespective of their type. Algorithms library contains built in functions that performs complex algorithms on the data structures.

For example: one can reverse a range with reverse() function, sort a range with sort() function, search in a range with binary_search() and so on.

Algorithm library provides abstraction, i.e you don't necessarily need to know how the the algorithm works.

C++: Containers in STL

Container library in STL provide containers that are used to create data structures like arrays, linked list, trees etc.

These container are generic, they can hold elements of any data types, for example: vector can be used for creating dynamic arrays of char, integer, float and other types.

C++: Iterators in STL

Iterators in STL are used to point to the containers. Iterators actually acts as a bridge between containers and algorithms.

For example: sort() algorithm have two parameters, starting iterator and ending iterator, now sort() compare the elements pointed by each of these iterators and arrange them in sorted order, thus it does not matter what is the type of the container and same sort() can be used on different types of containers.

Use and Application of STL

STL being generic library provide containers and algorithms which can be used to store and manipulate different types of data thus it saves us from defining these data structures and algorithms from the scratch. Because of STL, now we do not have to define our sort function every time we make a new program or define same function twice for the different data types, instead we can just use the generic container and algorithms in STL.

This saves a lot of time, code and effort during programming, thus STL is heavily used in the competitive programming, plus it is reliable and fast.

Containers:

Containers Library in STL gives us the Containers, which in simplest words, can be described as the objects used to contain data or rather collection of object. Containers help us to implement and replicate simple and complex data structures very easily like arrays, list, trees, associative arrays and many more.

The containers are implemented as generic class templates, means that a container can be used to hold different kind of objects and they are dynamic in nature!

Following are some common containers :

vector : replicates arrays

queue : replicates queues

stack : replicates stack

priority_queue : replicates heaps

list : replicates linked list

set : replicates trees

map : associative arrays

Classification of Containers in STL:

Containers are classified into four categories :

- Sequence containers : Used to implement data structures that are sequential in nature like arrays(array) and linked list(list).

- Associative containers : Used to implement sorted data structures such as map, set etc.

- Unordered associative containers : Used to implement unsorted data structures.

- Containers adaptors : Used to provide different interface to the sequence containers.

Using Container Library in STL

Below is an example of implementing linked list, first by using structures and then by list containers.

```
#include <iostream..h>

struct node
{
   int data;
   struct node * next;
}

int main ()
{
   struct node *list1 = NULL;
}
```

The above program is only creating a list node, no insertion and deletion functions are defined, to do that, you will have to write more line of code.

Now lets see how using Container Library simplifies it. When we use list containers to implement linked list we just have to include the list header file and use list constructor to initialize the list.

```
#include <iostream.h>
#include <list.h>
int main ()
{
   list<int> list1;
}
```

And that's it! we have a list, and not just that, the containers library also give all the different methods which can be used to perform different operations on list such as insertion, deletion, traversal etc.

Thus you can see that it is incredibly easy to implement data structures by using Container library

PAIR Template in STL

Although Pair and Tuple are not actually the part of container library but we'll still discuss them as they are very commonly required in programming competitions and they make certain things very easy to implement.

SYNTAX of pair is:

pair<T1,T2> pair1, pair2 ;

The above code creates two pairs, namely pair1 and pair2, both having first object of type T1 and second object of type T2.

Now T1 will be referred as first and T2 will be referred as second member of pair1 and pair2.

Example of Pair

Pair Template: Some Commonly used Functions

Here are some function for pair template :

Operator = : assign values to a pair.

swap : swaps the contents of the pair.

make_pair() : create and returns a pair having objects defined by parameter list.

Operators(== , != , > , < , <= , >=) : lexicographically compares two pairs.

Program demonstrating PAIR Template

```
#include <iostream.h>

int main ()

{

   pair<int,int> pair1, pair3;    //creats pair of integers

   pair<int,string> pair2;    // creates pair of an integer an a string


   pair1 = make_pair(1, 2);     // insert 1 and 2 to the pair1
```

```
  pair2 = make_pair(1, "Studytonight") // insert 1 and "Studytonight" in pair2
  pair3 = make_pair(2, 4)
  cout<< pair1.first << endl;  // prints 1, 1 being 1st element of pair1
  cout<< pair2.second << endl; // prints Studytonight

  if(pair1 == pair3)
      cout<< "Pairs are equal" << endl;
  else
      cout<< "Pairs are not equal" << endl;

  return 0;
}
```

TUPLE in STL

Tuple and Pair are very similar in their structure. Just like in pair we can pair two heterogeneous object, in tuple we can pair three heterogeneous objects.

SYNTAX of a tuple is:

```
// creates tuple of three object of type T1, T2 and T3
tuple<T1, T2, T3> tuple1;
```

Tuple Template

Tuple Template: Some Commonly used Functions

Similar to pair, tuple template has its own member and non-member functions, few of which are listed below :

A Constructor to construct a new tuple

Operator = : to assign value to a tuple

swap : to swap value of two tuples

make_tuple() : creates and return a tuple having elements described by the parameter list.

Operators(== , != , > , < , <= , >=) : lexicographically compares two pairs.

Tuple_element : returns the type of tuple element

Tie : Tie values of a tuple to its refrences.

Program demonstrating Tuple template

```
#include <iostream.h>
```

```
int main ()
{
   tuple<int, int, int> tuple1;   //creates tuple of integers
   tuple<int, string, string> tuple2;    // creates pair of an integer an 2 string

   tuple1 = make_tuple(1,2,3);  // insert 1, 2 and 3 to the tuple1
   tuple2 = make_pair(1,"Studytonight", "Loves You");
   /* insert 1, "Studytonight" and "Loves You" in tuple2  */
   int id;
   string first_name, last_name;
   tie(id,first_name,last_name) = tuple2;
   /* ties id, first_name, last_name to
   first, second and third element of tuple2 */
   cout << id <<" "<< first_name <<" "<< last_name;
   /* prints 1 Studytonight Loves You  */
   return 0;
}
```

TYPE CASTING:

Converting an expression of a given type into another type is known as type-casting. We have already seen some ways to type cast:

Implicit conversion

Implicit conversions do not require any operator. They are automatically performed when a value is copied to a compatible type. For example**:**

```
short a=2000;
int b;
b=a;
```

Here, the value of a has been promoted from short to int and we have not had to specify any type-casting operator. This is known as a standard conversion. Standard conversions affect fundamental data types, and allow conversions such as the conversions between numerical types (short to int, int to float, double to int...), to or from bool, and some pointer conversions. Some of these conversions may imply a loss of precision, which the compiler can signal with a warning. This warning can be avoided with an explicit conversion.

Implicit conversions also include constructor or operator conversions, which affect classes that include specific constructors or operator functions to perform conversions

For example:

```
class A {};

class B { public: B (A a) {} };

A a;

B b=a;
```

Here, an implicit conversion happened between objects of class A and class B, because B has a constructor that takes an object of class A as parameter. Therefore implicit conversions from A to B are allowed.

Explicit conversion

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion.

We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;

int b;

b = (int) a;    // c-like cast notation

b = int (a);    // functional notation
```

The functionality of these explicit conversion operators is enough for most needs with fundamental data types.

However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors.

For example, the following code is syntactically correct:

```
// class type-casting

#include <iostream.h>


class CDummy {

   float i,j;

};

class CAddition {

          int x,y;

  public:

          CAddition (int a, int b) { x=a; y=b; }

          int result() { return x+y;}

};
```

```
int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d;
  cout << padd->result();
  return 0;
}
```

DYNAMIC_CAST

Dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

Therefore, dynamic_cast is always successful when we cast a class to one of its base classes:

```
class CBase { };
class CDerived: public CBase { };
CBase b; CBase* pb;
CDerived d; CDerived* pd;
pb = dynamic_cast<CBase*>(&d);     // ok: derived-to-base
pd = dynamic_cast<CDerived*>(&b);  // wrong: base-to-derived
```

STATIC_CAST:

Static_Cast can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived.

This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type.

Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of dynamic_cast is avoided.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

This would be valid, although b would point to an incomplete object of the class and could lead to runtime errors if dereferenced.

static_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly, like for example standard conversion between fundamental types:

```
double d=3.14159265;

int i = static_cast<int>(d);
```

Or any conversion between classes with explicit constructors or operator functions as described in "implicit conversions" above.

reinterpret_cast

reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.

It can also cast pointers to or from integer types. The format in which this integer value represents a pointer is platform-specific. The only guarantee is that a pointer cast to an integer type large enough to fully contain it, is granted to be able to be cast back to a valid pointer.

The conversions that can be performed by reinterpret_cast but not by static_cast are low-level operations, whose interpretation results in code which is generally system-specific, and thus non-portable. For example:

```
class A {};

class B {};

A * a = new A;

B * b = reinterpret_cast<B*>(a);
```

This is valid C++ code, although it does not make much sense, since now we have a pointer that points to an object of an incompatible class, and thus dereferencing it is unsafe.

const_cast:

This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter

EXCEPTION HANDLING:

- The exception refers to unexpected condition in a program.
- The unexpected conditions could be faults, causing an error which in turn causes the program to fail.
- The error handling mechanism of C++ is generally referred to as exception handling.
- **Types of Exceptions**

1. Synchronous Exception
2. Asynchronous Exception

- **Synchronous Exception**

  The exception which occurs during the program execution, due to some fault in the input data or technique that is not suitable to handle the current class of data, with in the program are known as synchronous exception.

  Example: out of range, divide by zero, overflow, underflow

- **Asynchronous Exception**

  The errors that are caused by events beyond the control of the program are called asynchronous exceptions.

  Example: disk failure

Exception handling Mechanism:

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action a\can be taken.

The exception handling mechanism consists of the following task or operations:

1 Find the problem (hit the exception)

2 Inform that an error has occurred (throw the exception)

3 Receive the error information (catch the exceptions)

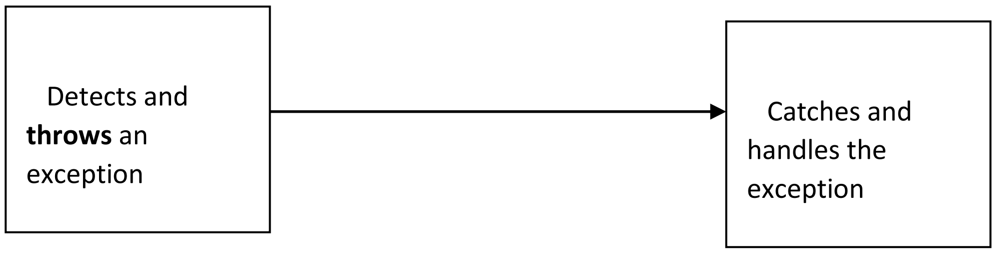
4 Take corrective actions(handle the exceptions)

Error handling basically consists of two segments;

- One to detect and to thro exceptions
- Other to catch the exceptions and to take appropriate actions.

Exception Handling Model:

The exception handling mechanism uses three blocks try, throw, catch. The relationship of those three exception handling model shown below


Detects and **throws** an exception
Catches and handles the exception

try block   **catch** block

Exception

- The keyword try is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements known as try bock.
- When an exception is detected, it is thrown using a **throw** statement in the try block.
- A catch block defined by the keyword **catch** catches the exception thrown by the throw statement in the try block, and handle it appropriately.
- **General format:**

```
…………
…………
try
{
        ……….                  // block of statements which detects and
        throw   exception      // throws an exception
        ……….
        ……….
}
catch ( type arg)
{
        ………                   // block of statements that handles the exception
        ………
        ………
}
……..
……..
```

- When the **try** block throws an exception, the program control leaves the try block and enters the catch statement of the catch block.
- Exceptions are objects or variables used to transmit information about a problem.
- If the type of object thrown matches the arg type in the catch statement, then catch block is executed for handling the exception.

- If they do not match, the program is aborted with the help of the **abort()** function which is invoked automatically.
- The catch block may have more than one catch statements, each corresponding to a particular type of exception.
- For example if the throw block is likely to throw two exception, the catch block will have two catch statements one for each type of exception. Each catch statement is called is exception handler.
- When no exception is detected and thrown, the control goes to the statement immediately after the catch block. That is catch block is skipped.
- Example: divide by zero

```
#include<iostream.h>
void main( )
{
    int x, y, z;
    cout<<"enter the values of x, y, z";
    cin>>x>>y>>z;
    try
    {
        if(x= =y)
            throw y;
        else
            int d= z / ( x- y);
            cout<<"the result of z/ (x-y )is:"<< d;
    }
    catch( int y)
    {
        cout<<"exception caught x-y is "<< x-y ;
    }
}
```

RUN 1:

```
    enter the values of x , y, z
        50  30 20
        the result of z/ (x-y ) is: 1
```

RUN 2:

```
        Enter the values of x, y, z
                30   30 20
        exception caught x-y is : 0
```

Using function that generate exceptions:

```
    #include<iostream.h>
    void divide ( int x, int y,int z)
    {
                cout<<"we are inside the function";
                if(x= =y)
                        throw (x-y);
                else
                        int d= z / ( x- y);
                        cout<<"the result of z/ (x-y )is:"<< d;
    }
    void main( )
    {
        try
        {
                cout<<" we are inside the try block";
                divide( 10, 20, 30);            // invoke divide ()
                divide( 10, 10, 20);            // invoke divide ()
        }
        catch( int i)                                   // catches the exception
        {
                cout<<"exception caught x-y is :  "<<i;
        }
    }
```

RUN:

```
we are inside the try block
we are inside the function
```

```
the result of z/ (x-y )is:  -3
we are inside the function       exception caught x-y is: 0
```

Multiple catches with object:

- To throw an exception with an object it needs to define an abstract class.
- Abstract class is nothing but a class with empty data members.
- Class name is called as nameless object which is used to throw the exceptions

Example:

```
#include<iostream.h>
class positive {  };
class negative {  };
class zero {  };

void greater ( int num)
{
 if( num >0)
        throw positive( );
else
        if (num < 0)
                throw negative( );
        else
                throw zero( );
}
void main( )
{
        int num;
        cout<<"enter any number";
        cin>>num;
        try
        {
                greater ( num);
        }
        catch( positive)
        {
                cout<<"+ve exception";
        }
        catch(negative)
        {
                cout<<"-ve exception";
        }
        catch (zero)
        {
                cout<<"0 exception";
        }
}
```