



For B.E. B.C.A. M.C.A. M.C.M. B.S.C. Polytechnic



Programming

by Choudhari Sir



CCIT
Keeping Pace with Technology

An ISO 9001 : 2008 Certified Company
website: www.ccitindia.com
Rajapeth: 0721-2563615 GadgeNagar: 0721-2552289

Index

Topic	Page
C++ Basic : History, Features, OOPS vs POP, C vs C++, Applications , C++ Program Format, Cout, Cin Objects DataTypes, Operators, Identifiers , Literals, Type Casting, Variables types/Scope	2
Control Structures If statement ,Nested if , Ladder, Ternary Operator, while , for , do-while, switch , Nested Loops	12
Arrays Single Dimensional , Multi Dimensional , Char Arrays, String Functions	49
Structures Defining Structures, Accessing Members, Structure Initialization, Anyonamoyus Structures, Type Def	
Functions Functions basic, types of functions, Local & Global Variables, Function Overloading, Default Arguments, Reference Variables, Reference Arguments, Funciton returing References.	57
OOPS Oops basic, classes, objects,data members , methods, member function overloading, access specifiers,	61
Constructors Default constructor, Constructor with Args, Constructor Overloading, Initialization blocks, private constructors.	81
Friend Functions / Classes Inline functions, Friend Functions, Friend Classes, Passing Objects as Arguments, Functions Returning Objects.	
Static Members Static data members, static member functions	87

Operator Overloading Basics,Unary Operator Overloading, Binary Operator Overloading, Type Casting, Conversion Function	91
Inheritance Basics, Method Overriding, , types of inheritance, abstract classes, Aggeration, Virtual Base Classes	108
Pointers Basics,new ,delete operators, Object Pointers	
Polymorphism Basics, compiletime vs runtime polymorphism, Virtual Functions, Pure Virtual Functions , Virtual Destructors	124
Input-Output / Streams Console I/O, File I/O ,File Modes, Block I/O, File Pointers, File Errors , Overoading >> and << Operators , Command Line Arguments	246
Exceptions Basics, try,catch, throw, throws, keywords, Exception classes, User defined Exceptions	
Templates Basics, Function Templates, Class Templates	260
Addational Topics String class, NameSpaces, MultiFile Programs, RTTI	
STL (standard template library) Basics, Containers, Iterators, Algorithms	
Preprocessor Basics, #define, #include, #ifdef.....	

C++ is a powerful, compiled, general-purpose, case-sensitive programming language that supports procedural, object-oriented, and generic programming.

Features

Object oriented Programming language

This main advantage of C++ is, it is object oriented programming language. It follows concept of oops like polymorphism, inheritance, encapsulation, abstraction.

Simple

Every C++ program can be written in simple English language so that it is very easy to understand and developed by programmer.

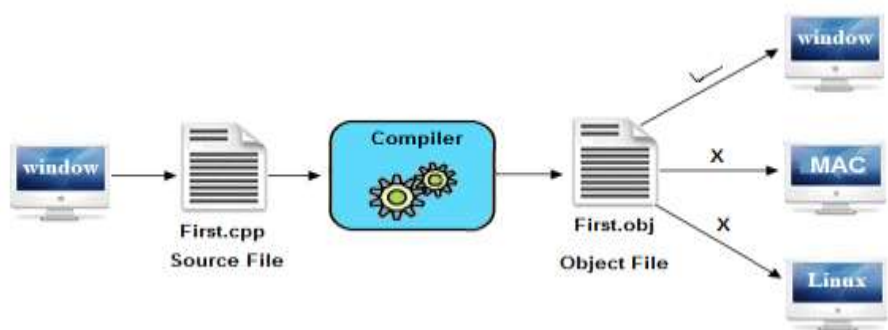
Case sensitive

C++ is a case sensitive programming language. In C++ programming 'break and BREAK' both are different.

If any language treats lower case latter separately and upper case latter separately than they can be called as case sensitive programming language [Example c, c++, java, .net are sensitive programming languages.] other wise it is called as **case insensitive** programming language [Example HTML, SQL is case insensitive programming languages].

Platform dependent

A language is said to be platform dependent whenever the program is executed in the same operating system where it was developed and compiled but not run and execute on other operating systems. C++ is a platform dependent language.



Note: .obj file of C++ program is platform dependent.

Syntax based language

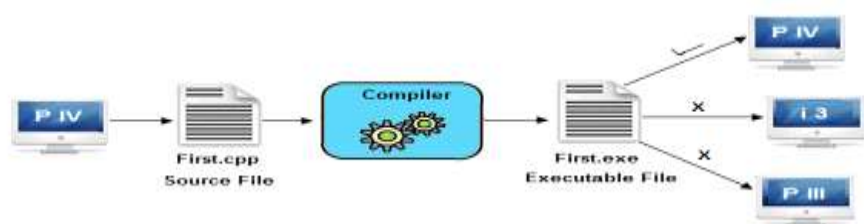
C++ is a strongly tight syntax based programming language. If any language follows rules and regulations very strictly known as strongly tight syntax based language. Example C, C++, Java, .net etc. If any language does not follow rules and regulations very strictly known as loosely tight syntax based language. Example HTML.

Powerful

C++ is a very powerful programming language, it has a wide variety of data types, functions, control statements, decision making statements, etc.

Portability

It is the concept of carrying the instruction from one system to another system. In C++ Language **.cpp** file contain source code, we can edit also this code. **.exe** file contain application, only we can execute this file. When we write and compile any C++ program on window operating system that program easily run on other window based system.



When we can copy .exe file to any other computer which contain window operating system then it works properly, because the native code of application an operating system is same.

Compiler based

C++ is a compiler based programming language that means without compilation no C++ program can be executed. First we need compiler to compile our program and then execute.

Efficient use of pointers

Pointers is a variable which hold the address of another variable, pointer directly direct access to memory address of any variable due to this performance of application is improve. In C++ language also concept of pointer are available.

History

In 1970 two programmers, Brian Kernighan and Dennis Ritchie, created a new language called C. C was designed with one goal in mind: writing operating systems.

C had one major problem, however. It was a procedure-oriented language. This meant that in designing a typical C program, the programmer would start by describing the data and then write procedures to manipulate that data. Programmers eventually discovered that it made a program clearer and easier to understand if they were able to take a bunch of data and group it together with the operations that worked on that data. Such a grouping is called an *object* or *class*.

Designing programs by designing classes is known as *object-oriented design (OOD)*.



Bjarne Stroustrup

In 1980 Bjarne Stroustrup started working on a new language, called "C with Classes." This language improved on C by adding a number of new features, the most important of which was classes. This language was improved, augmented, and finally became C++. C++ owes its success to the fact that it allows the programmer to organize and process information more effectively than most other languages.

OOPs Vs Procedure-oriented programming

Procedure Oriented Programming	Object Oriented Programming
In POP, program is divided into small parts called functions .	In OOP, program is divided into parts called objects .
In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world .
POP follows Top Down approach .	OOP follows Bottom Up approach .
POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
POP does not have any proper way for hiding data so it is less secure .	OOP provides Data Hiding so provides more security .
languages: C, FORTRAN, Pascal.	languages: C++, JAVA, VB.NET, C#.NET.

C Vs C++

C	C++
C was developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979 with C++'s predecessor "C with Classes".
When compared to C++, C is a subset of C++.	C++ is a superset of C. C++ can run most of C code while C cannot run C++ code.
C supports procedural programming paradigm for code development.	C++ supports both procedural and object oriented programming paradigms; ie. an hybrid language.
C does not support object oriented programming; therefore it has no support for polymorphism, encapsulation, and inheritance.	Being an object oriented programming language C++ supports polymorphism, encapsulation, and inheritance.
In C (because it is a procedural programming language), data and functions are separate and free entities.	In C++ (when it is used as object oriented programming language), data and functions are encapsulated together in form of an object. For creating objects class provides a blueprint of structure of the object.
In C, data are free entities and can be manipulated by outside code. This is because C does not support information hiding.	In C++, Encapsulation hides the data to ensure that data structures and operators are used as intended.
C is a middle level language.	C++ is a high level language.
C programs are divided into modules and procedures.	C++ programs are divided into classes and functions.

C, being a procedural programming, it is a function driven language.	While, C++, being an object oriented programming, it is an object driven language.
C does not support function and operator overloading.	C++ supports both function and operator overloading.
C does not allow functions to be defined inside structures.	In C++, functions can be used inside a structure.
C uses functions for input/output. For example scanf and printf.	C++ uses objects for input output. For example cin and cout.
C does not support reference variables.	C++ supports reference variables.
C has no support for virtual and friend functions.	C++ supports virtual and friend functions.
C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation.	C++ provides new operator for memory allocation and free operator for memory de-allocation.
C does not provide direct support for error handling (also called exception handling)	C++ provides support for exception handling. Exceptions are used for "hard" errors that make the code incorrect.
C programs use top-down approach.	C++ programs use bottom-up approach.
In C, all the variables must be declared at the beginning of a scope.	C++ allows declaring variables anywhere within the scope. This allows us to declare a variable when we use it for the first time.

Application of OOP

The promising areas for application of OOP include:

- Real-time systems.
- Simulation and modeling.
- Object-oriented databases.
- Hypertext, hypermedia and experttext.
- AI and expert system.
- Neural network and parallel programming.
- Decision support and office automation systems.
- CIM/CAM/CAD systems.

Applications of C++

Mainly C++ Language is used for **Develop Desktop application** and **system software**. Some application of C++ language are given below.

- For Develop Graphical application like computer and mobile games.
- To evaluate any kind of mathematical equation use C++ language.
- C++ Language are also used for design OS. Like window xp.
- Few parts of apple OS X are written in C++ programming language.
- Internet browser Firefox are written in C++ programming language
- Some of the Google app. such as Google file system, Google Chromium.
- C++ are used for design database like MySQL.

Benefits/Advantages of OOP

- **Reusability:** In OOP's programs functions and modules that are written by a user can be reused by other users without any modification.
- **Inheritance:** Through this we can eliminate redundant code and extend the use of existing classes.
- **Data Hiding:** The programmer can hide the data and functions in a class from other classes. It helps the programmer to build the secure programs.
- **Reduced complexity of a problem:** The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.
- **Easy to Maintain and Upgrade:** OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
- **Message Passing:** The technique of message communication between objects makes the interface with external systems easier.
- **Modifiability:** it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods;

• Keywords

Keyword is a predefined or reserved word in C++ with a fixed meaning and used to perform an internal operation.

32 Keywords which is also available in C language.

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

Another 30 reserved words that are new to C++

asm	dynamic_cast	namespace	reinterpret_cast
bool	explicit	new	static_cast
catch	false	operator	template
class	friend	private	this
const_cast	inline	public	throw
delete	mutable	protected	true
try	typeid	typename	using
using	virtual	wchar_t	

TOKENS

- Tokens are the basic building blocks in a language which are constructed together to write a program.
- Each and every smallest individual unit in a program is known as a token.

C tokens are of six types. They are,

1. Keywords (eg: int, while),
2. Identifiers (eg: main, total),
3. Constants (eg: 10, 20),
4. Strings (eg: "total", "hello"),
5. Special symbols (eg: (), {}),
6. Operators (eg: +, /, -, *)

TOKENS EXAMPLE PROGRAM:

```
int main()
{
int x, y, total;
x = 10, y = 20;
total = x + y;
cout<<"Total = "<< total<<endl;
}
```

where,

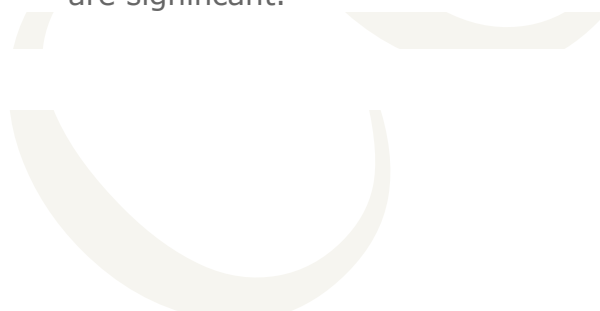
- main – identifier
- {, }, (,) – delimiter
- int – keyword
- +, =, << – operators
- x, y, total – identifier
- main, {, }, (,), int, x, y, total, +, =, << – tokens

IDENTIFIERS

- Each program elements in a program are given a name called identifiers.
- Names given to identify Variables, functions and arrays are examples for identifiers.
- eg. `int x;`
- `x` is a name given to integer variable

RULES FOR CONSTRUCTING IDENTIFIER NAME IN C:

1. First character should be an alphabet or underscore.
2. Succeeding characters might be digits or letter or underscore.
3. Punctuation and special characters aren't allowed except underscore.
4. Identifiers should not be keywords.
5. It should be up to 31 characters long as only first 31 characters are significant.



Constants/Literals

Constants/literal refers to the data items that do not change their value during the program execution. Several types of C constants that are allowed in C are:

1. Integer Constants

Integer constants are whole numbers without any fractional part. It must have at least one digit and may contain either + or – sign. A number with no sign is assumed to be positive.

There are three types of integer constants:

1.1. Decimal Integer Constants

Integer constants consisting of a set of digits, 0 through 9, preceded by an optional – or + sign.

Example of valid decimal integer constants
341, -341, 0, 8972

1.2. Octal Integer Constants

Integer constants consisting of sequence of digits from the set 0 through 7 starting with 0 is said to be octal integer constants.

Example of valid octal integer constants
010, 0424, 0, 0540

1.3. Hexadecimal Integer Constants

Hexadecimal integer constants are integer constants having sequence of digits preceded by 0x or 0X. They may also include alphabets from A to F representing numbers 10 to 15.

Example of valid hexadecimal integer constants
0xD, 0X8d, 0X, 0xbD

It should be noted that, octal and hexadecimal integer constants are rarely used in programming.

2. Real Constants

The numbers having fractional parts are called real or floating point constants. These may be represented in one of the two forms called *fractional form* or the *exponent form* and may also have either + or – sign preceding it.

Example of valid real constants in fractional form or decimal notation

0.05, -0.905, 562.05, 0.015 , 252E85, 0.15E-10, -3e+8

3. Character Constants

A character constant contains one single character enclosed within single quotes.

Examples of valid character constants

'a' , 'Z' , '5'

It should be noted that character constants have numerical values known as ASCII values, for example, the value of 'A' is 65 which is its ASCII value.

4. String Literals

String literals or constants are enclosed in double quotes "".

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"
```

```
"hello, \
```

```
dear"
```

```
"hello, " "d" "ear"
```

General format of a C++ program.

```
Preprocessor commands
Global declaration.
void main()
{
    Statements
    -----
}
```

Preprocessor Commands

These are the instructions which we want to give to our compiler (Its preprocessor). They indicate how the program is to be compiled. A preprocessor commands Starts with the symbol #.

eg. # include
define
ifdef

Global declaration

In this section we can define variables, functions, Structures, classes etc. Variables define in this section are accessible anywhere within the program.

Function main()

This is the entry point of our program. Execution of the program begins with function main.

Console I/O Objects

C++ provides us two special objects:-

- 1) cout &
 - 2) cin
- to perform console I/O operation.

- 1) **cout:-** This object is attached to our standard output device so instead of sending data directly to the output device we can send it through this object to perform such operation it provides us an insertion operator "<<".
- 2) **cin:-** This object is attached to our standard input device so received by this object. We can extract this data from the object cin and store it into our variables by using a extraction operator ">>".

Note:-Both the above objects cout & cin are defined in the header file

- * **Old version** =>iostream.h
- * **New version** =>iostream

Old version // Program to print addition of two no.'s

```
# include<iostream.h>
void main()
{ int a,b,c;
  cout <<"Enter two no.'s";
  cin >>a>>b;
  c=a+b;
  cout<<"Sum is "<<c;
}
```

New version

```
# include<iostream>
using namespace std;
void main()
{ int a,b,c;
  cout <<"Enter two no.'s";
  cin >>a>>b;
  c=a+b;
  cout<<"Sum is "<<c;
}
```

*** wap to read three integers find their mean.**

```
#include<iostream.h>
void main()
{
  int a,b,c;
  float m;
  cout << "Enter 3 no.'s";
  cin >> a >> b >> c;
  m= (a+b+c)/3.0;
  cout << "Mean is " << m;
}
```

*** wap to read radius and find area and circum. of circle.**

```
# include <iostream.h>
void main()
{
  float r , A , C;
  cout << "Enter radius of circle";
  cin >> r;
  A= 3.14 * r * r;
  cout << "Area is " << A;
  C=2 * 3.14 * r;
  cout << "Circumference is " << C;
}
```

*** wap to read marks for five different subjects and print total marks and percentage .**

```
# include <iostream.h>
void main()
{
    int a , b , c , d , e , s;
    float p;
    cout << "Enter marks for five subjects";
    cin >> a >> b >> c >> d >> e;
    s= a + b + c + d + e;
    p=(s*100)/500.0;
    cout << "Sum is " << s << "Percentage is " << p;
}
```

*** wap to read two no.'s a & b exchange their value.**

```
#include <iostream.h>
void main()
{
    int a , b , c;
    cout << "Enter two no.'s";
    cin >> a >> b;
    c=a;
    a=b;
    b=c;
    cout << "Exchange value are " << a << b;
}
```

Comments in C++

Generally **Comments** are used to provide the description about the Logic written in program. Comments are not display on output screen. When we are used the comments, then that specific part will be ignored by compiler.

Single line comments

Single line comments can be provided by using `//.....`

Multiple line comments

Multiple line comments can be provided by using `/*.....*/`

Note: When we are working with the multiple line comments then nested comments are not possible.

Basic Operators

Arithmetic operator:	<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code> , <code>++</code> , <code>--</code>
Assignmant :	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code>
Relational :	<code><</code> , <code>></code> , <code><=</code> , <code>>=</code> , <code>==</code> , <code>!=</code>
Logical :	<code>&&</code> , <code> </code> , <code>!</code>

Variable Types.

The sizes of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

<i>Type</i>	<i>Size</i>	<i>Range</i>
unsigned short int	2 bytes	0 to 65,535
short int	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int	4 bytes	-2,147,483,648 to 2,147,483,647
int (16 bit)	2 bytes	-32,768 to 32,767
int (32 bit)	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int (16 bit)	2 bytes	0 to 65,535
unsigned int (32 bit)	4 bytes	0 to 4,294,967,295
char	1 byte	256 character values
float	4 bytes	1.2e-38 to 3.4e38
double	8 bytes	2.2e-308 to 1.8e308

*** wap to read a number and find last digit of that no.**

```
# include <iostream.h>
void main()
{ int num , a;
  cout << "Enter a no.";
  cin >> a;
  num= a%10;
  cout << "Last digit of no. is " << num;
}
```

*** wap to read 2 no.'s a & b & find sum of their last digit.**

```
# include <iostream.h>
void main()
{ int a , b , c , d , sum;
  cout << "Enter two no.'s";
  cin >> a >> b;
  c= a%10;
  d= b%10;
  sum = c+d;
  cout << "Sum of Last digit is " << sum;
}
```


Operator Precedence

Category	Operator	Associativity
Postfix	() [] -> . ++ --	Left to right
Unary	+ - ! ~ ++ -- (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

If Statement

It is used to conditionally execute a block of code. If condition is true then statements within if block are executed. If condition is false then statements within else block are executed.

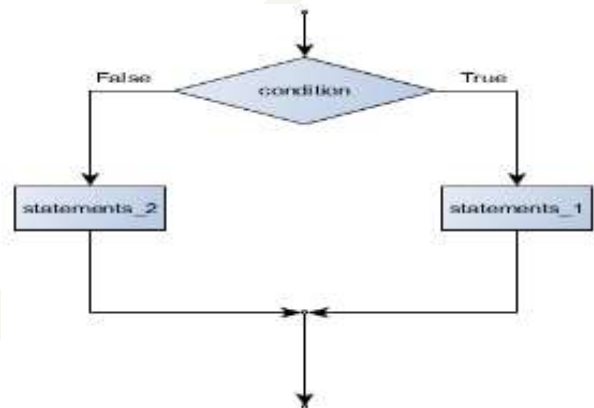
Syntax :=

```
if (condition)
{
    statements
}
else
{
    statements
}
next statements
```

Note:

else block is optional.

Condition can be specified by using a Boolean expression. A Boolean expression always returns a value *True* or *False* (i.e. 1 for *True* & 0 for *False*). **Boolean expression** can be created by using relational & logical operators.



* wap to read a no. and check if it is an even no. of odd no.

```
# include <iostream.h>
void main()
{ int a;
  cout << "Enter a no.";
  cin >> a ;
  if ( a%2 == 0)
    cout << "no. is even ";
  else
    cout << "no. is odd ";
}
```

*** wap to read three angles of a triangle and check if triangle can be found or not.**

```
# include <iostream.h>
void main()
{
    int a , b , c;
    cout << "Enter three angles";
    cin >> a >> b >> c;
    if ( a + b + c==180)
        cout << "Triangle is formed  ";
    else
        cout << "Triangle is not formed  ";
}
```

*** wap to read three angles of a triangle and check if triangle is an equilateral or not.**

```
# include <iostream.h>
void main()
{
    int a , b , c;
    cout << "Enter three angles";
    cin >> a >> b >> c;
    if ( a == 60 && b == 60 && c == 60)
        cout << "Triangle is equilateral  ";
    else
        cout << "Triangle is not equilateral  ";
}
```

* wap to read marks for five different subjects & print total marks and also print its percentage if student is pass.

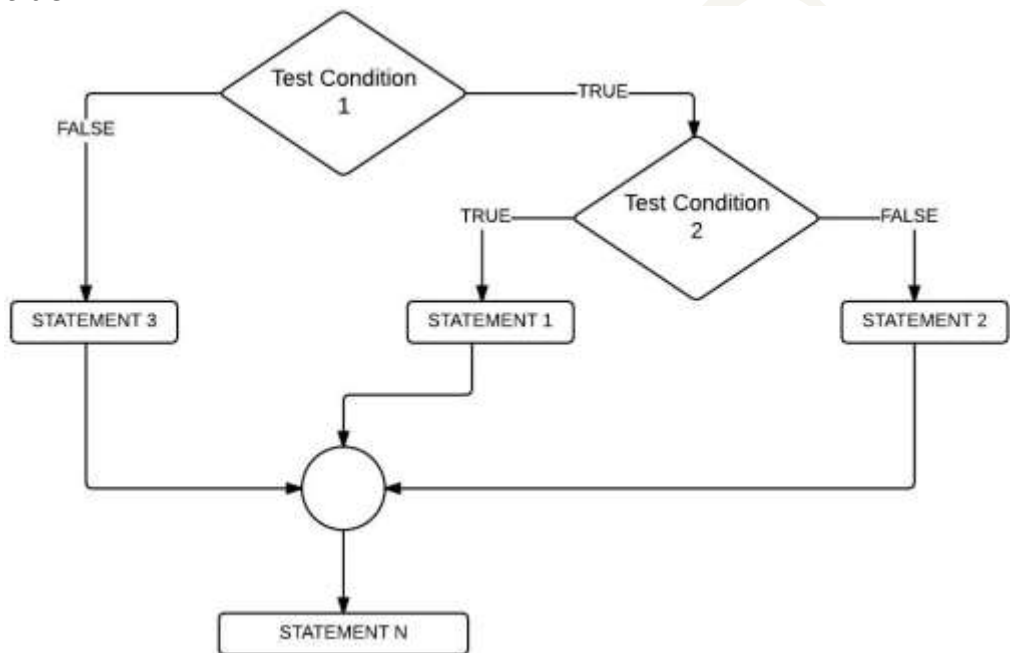
```
# include <iostream.h>
void main()
{
    int a , b , c , d , e , total;
    float p;
    cout << "Enter marks for 5 different subject";
    cin >> a >> b >> c >> d >> e;
    total = a + b + c + d + e;
    cout << "Total is " << total;

    if (a >= 35 && b >= 35 && c >= 35 && d >= 35 && e >= 35)
    {
        cout << "Student is pass ";
        p= (total * 100.0)/500.0;
        cout << "Percentage is " << p;
    }
    else
        cout << "Student is fail ";
}
```

Nested if

If a 'if' statement is use within a if statement then such a control structure is called as "nested if".

It is use to check a condition only if same other condition is true.



*** wap to read marks for five different subjects & print total marks and percentage also check for first class if student is pass.**

```
# include <iostream.h>
void main()
{
int a , b , c , d , e , total;
float p;
cout << "Enter marks for 5 different subject";
cin >> a >> b >> c >> d >> e;
total = a + b + c + d + e;
cout << "Total is " << total;
p= (total * 100.0)/500.0;
cout << "Percentage is  " << p;
if (a>=35 && b>=35 && c>=35 && d>=35 && e>=35)
{
    cout << "Student is pass  ";
    if (p >= 60)
    {
        cout<< "Student is pass in first class";
    }
}
else
    cout << "Student is fail  ";
}
```

* wap to read three angles of a triangle & check if triangle can be formed or not. If triangle can be formed then check if it is equilateral, isosceles or right angle triangle.

```
# include <iostream.h>
void main()
{
int a , b , c;
cout << "Enter three angles";
cin >> a >> b >> c ;
if (a + b + c == 180)
{
    cout << " Triangle is formed ";
    if ( a==b && b==c)
        cout << "Triangle is equilateral";
    if (a==b || b==c || a==c)
        cout << "Triangle is Isosceles";
    if (a==90 || b==90 || a==90)
        cout << "Triangle is Right angle triangle";
}
else
    cout << "Triangle is not formed";
}
```

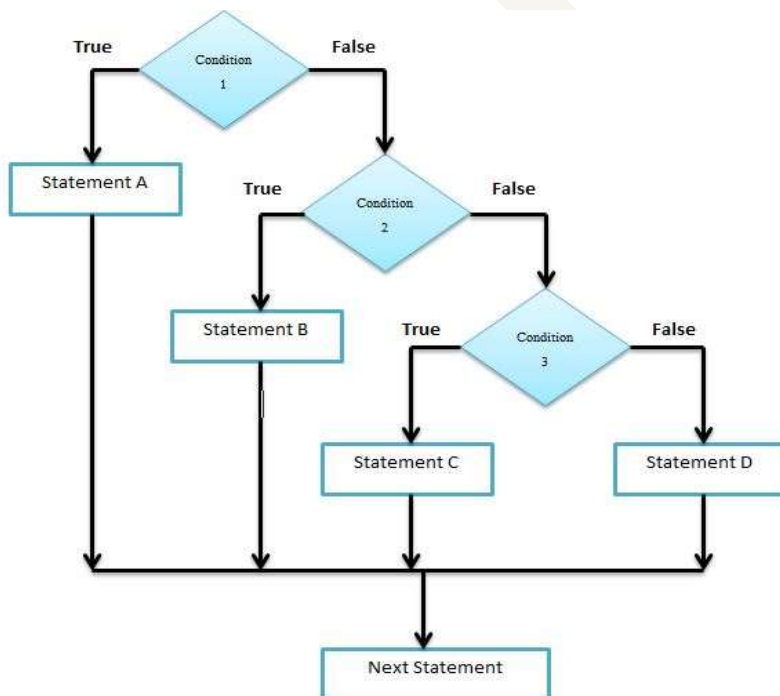
LADDER STATEMENT

If a 'if' statement is used within a else statement the such compound structure called is LADDER structure. If is use to check a condition only if some other condition is false.

Syntax: - **if (condition)**

 else if (condition)

 else if (condition)



***wap to read percentage obtain by a student & print division.**

```
# include <iostream.h>
void main()
{ float p;
cout << "Enter percentage";
cin >> p;
if (p >= 75)
    cout << " Distinction ";
else if (p >= 60)
    cout << " First class ";
else if (p >= 50)
    cout << " Second class ";
else if (p >= 40)
    cout << " Third class ";
else
    cout << "fail";
}
```

*** wap to read three different no.'s a, b, c and find greatest of them.**

```
# include <iostream.h>
void main()
{ int a , b , c;
cout << "Enter three no.'s";
cin >> a >> b >> c;
if (a > b && a > c)
    cout << " a is greatest ";
else if (b > a && b > c)
    cout << " b is greatest ";
else if (c > a && c > b)
    cout << " c is greatest ";
}
```

Conditional (Ternary) Operator

The conditional operator (?:) is C++'s only ternary operator; that is, it is the only operator to take three terms.

The conditional operator takes three expressions and returns a value:

(expression1) ? (expression2) : (expression3)
--

This line is read as "If expression1 is true, return the value of expression2; otherwise, return the value of expression3." Typically, this value would be assigned to a variable.

Eg `z = (x > y) ? x : y;`

```
#include <iostream>
using namespace std;
int main ()
{
    int x, y = 10;

    x = (y < 10) ? 30 : 40;

    cout << "value of x: " << x << endl;

    return 0;
}
```

Loops

Loops are used to repeatedly execute a block of code C++ provides us three different looping structures.

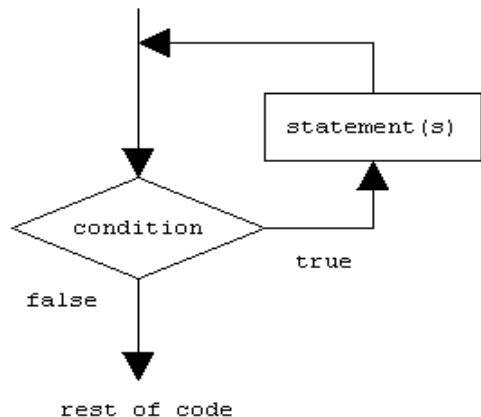
- 1> while loop
- 2> for loop
- 3> do-while loop

while loop

Statement within this loop is repeatedly executed while condition is true.

Syntax: -
 While (condition)
 {
 statements

 }
 Next statements



* **wap to print all no.'s from 1 to 10.**

```

# include <iostream.h>
void main()
{
    int i=1;
    while (i <=10 )
    {
        cout << i;
        i++;
    }
}
  
```

*** wap to read a no and print all no from 1 to that no.**

```
# include <iostream.h>
void main()
{
    int i=1,a;
    cout << "Enter a no."
    cin >> a;
    while (i <=a )
    {
        cout << i;
        i++;
    }
}
```

*** wap to read a no and find sum of all no from 1 to that no.**

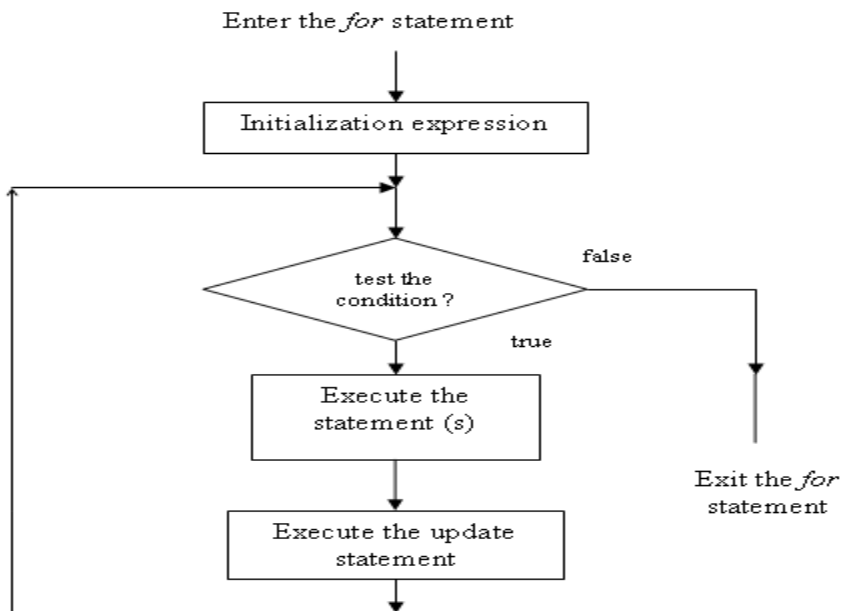
```
# include <iostream.h>
void main()
{
    int i=1,a,s=0;
    cout << "Enter a no."
    cin >> a;
    while (i <=a )
    {
        S=s+i;
        i++;
    }
    Cout<<"Sum is "<<s;
}
```

For loop

Statement within this loop are repeatedly executed while condition is true. Specialty of this loop is that initialization, condition, evaluation is specified at the starting of the loop.

Syntax :-

```
for(initialization;condition;evaluation)
{
    statements
    -----
}
Next statements
-----
```



*** wap to read a no and print all even no from 1 to that no.**

```
# include <iostream.h>
void main()
{ int i, a;
  cout << "Enter a no."
  cin >> a;
  for (i=1 ; i <= a ; i++)
    { if (i%2 == 0)
      cout << i;
    }
}
```

*** wap to read a no and print sum of all no from 1 to that no.**

```
# include <iostream.h>
void main()
{ int i , a , sum=0;
  cout << "Enter a no."
  cin >> a;
  for (i=1 ; i <= a ; i++)
    sum = sum + i
  cout << "Sum is "<< sum;
}
```

*** wap to read a no and print factorial of that no.**

```
# include <iostream.h>
void main()
{ int i , a , f=1;
  cout << "Enter a no."
  cin >> a;
  for (i=1 ; i <= a ; i++)
    f = f * i;
  cout << "Factorial is "<< f;
}
```

Nested loops

If a loop is used within a loop then such a control structure is called as Nested loop.

```
for(-----)
{
    for(-----)
    {
        -----
    }
}
```

Note: any loop can be used within any loop

* **wap to print following output.**

```
" 7 6 5 4 3 2 1
  7 6 5 4 3 2 1
  7 6 5 4 3 2 1
  7 6 5 4 3 2 1
  7 6 5 4 3 2 1 "
```

```
# include <iostream.h>
void main()
{
    int I , n ;
    for(n=1 ; n<=5 ; n++)
    {
        cout << endl;
        for (I=7 ; I>=1 ; I--)
            cout << I ;
    }
}
```

Comma Operator

In general, comma operator is used in the following situations:

- calling a function
- entering or repeating an iteration loop
- testing a condition

Note: if comma operator is used between 2 conditions then it acts like logical and operator

```
void main()
{
for(int i=1,j=9;i<=9,j>=1;i++,j--)
    cout<<i<<" "<<j<<endl;
}
```

Output:

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


do-while loop

Statements within this loop are repeatedly executed while condition is true. Specialty of this loop is that it is executed at least once even if condition is false.

Syntax :

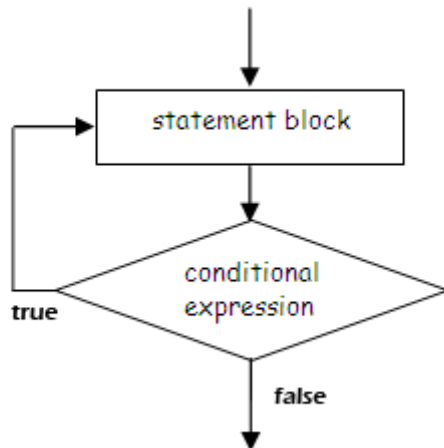
do

```
{
  Statements..
  -----
}
```

```
while(condition);
Next statement;
```

*** wap to read a no while the no. entered is not equal zero and find sum of all the no entered.**

```
# include <iostream.h>
void main()
{
  int N , s=0;
  do
  {
    cout << "Enter a no."
    cin >> N;
    s= s + N ;
  }
  while (N != 0);
  cout << "Sum is " << s;
}
```



* wap to read a no while the no. entered is not equal zero and find sum of all the even no entered.

```
# include <iostream.h>
void main()
{
    int N , s=0;
    do
    {
        cout << "Enter a no."
        cin >> N;
        if(N%2 == 0)
            s= s + N ;
    }
    while (N != 0);
    cout << "Sum is " << s;
}
```

Jump statements

C++ provides us different jump statements which are used to control the flow of execution of our program.

Break statement

It is used to throw program control out of a loop or a switch statement.

Eg.

```
# include <iostream.h>
void main()
{ int i=0;
  while(i<=10)
  {
    cout<<i;
    if (i==5)
      break;
    i++;
  }
}
```

o/p=> 12345

Continue Statement

This statement is used to throw program control at the starting of the loop.

```
Eg.# include <iostream.h>
void main()
{int i=0;
 while(i<=10)
 {
  i++;
  if (i%2==0)
    continue;
  cout << i;
 }
}
```

o/p=> 13579

Goto statement

This statement is used to transfer program control any where within the function.

Syntax: ***goto label;***

It transfer program control at specified label statement label can be defined by using .

syntax : identifier.

Eg.

```
# include <iostream.h>
void main()
{
    int i=1;
    ABC:
    Cout << "Ram";
    i++;
    if (i<=180)
        goto ABC;
    else
        goto XYZ;
        cout<< "Laxman"
    XYZ:
    Cout << "Seeta";
}
```

Symbolic Constants

A symbolic constant is an identifier with an associated value which cannot be altered by the program during normal execution

There are two ways to declare a symbolic constant in C++.

1] #define :

The #define directive is a **preprocessor** directive; the preprocessor replaces those macros by their body *before* the compiler even sees it. Think of it as an automatic search and replace of your source code.

For eg: #define PI 3.14

2] const :

A const variable declaration declares an actual variable in the language, which you can use... well, like a real variable: take its address, pass it around, use cast it, convert it, etc.

For eg: const float PI=3.14;

const Vs #define

- const means the data can't be changed while define has no such meaning.
- const is used by compiler and define is performed by pre-processor.
- const has scope while define can be considered global.
- const variable can be seen from debugger while macro can be seen.
- const guarantees the type which macro can't

Enumerated Constants

An enumeration is a set of named integer constants. An enumerated type is declared using the **enum** keyword.

The syntax for enumerated constants is to write the keyword **enum**, followed by the type name, an open brace, each of the legal values separated by a comma, and finally a closing brace and a semicolon. Here's an example:

```
enum COLOR { RED, BLUE, GREEN, WHITE, BLACK };
```

This statement performs two tasks:

1. It makes **COLOR** the name of an enumeration, that is, a new type.
2. It makes **RED** a symbolic constant with the value 0, **BLUE** a symbolic constant with the value 1, **GREEN** a symbolic constant with the value 2, and so forth.

Every enumerated constant has an integer value. If you don't specify otherwise, the first constant will have the value 0, and the rest will count up from there. Any one of the constants can be initialized with a particular value, however, and those that are not initialized will count upward from the ones before them. Thus, if you write

```
enum Color { RED=100, BLUE, GREEN=500, WHITE, BLACK=700 };
```

then **RED** will have the value 100; **BLUE**, the value 101; **GREEN**, the value 500; **WHITE**, the value 501; and **BLACK**, the value 700.

These advantages include a

- lower maintenance requirement,
- improved program readability
- better debugging capability.

Variables Vs Constants

Basis	Variables	Constants
Definition	A variable is a named location, whose value changes during a program run	A constant is a named location, whose value never changes during a program run.
Syntax	data_type variable_name	const data_type variable_name= value
Example	int a;	const int a=100;
Static Initialization	Variables can be initialize after its declaration also.	constants must be initialized at the time of decelARATION only.
Dynamic Initialization	Variables can be initialize dynamically	Dynamic Initialization of constant is not possible.

Prefix and Postfix

Both the increment operator (++) and the decrement operator(--) come in two varieties: prefix and postfix. The prefix variety is written before the variable name (++myAge); the postfix variety is written after (myAge++).

In a simple statement, it doesn't much matter which you use, but in a complex statement, when you are incrementing (or decrementing) a variable and then assigning the result to another variable, it matters very much. The prefix operator is evaluated before the assignment, the postfix is evaluated after.

This can be confusing at first, but if `x` is an integer whose value is 5 and you write

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

you have told the compiler to increment `x` (making it 6) and then fetch that value and assign it to `a`. Thus, `a` is now 6 and `x` is now 6.

If, after doing this, you write

```
int b = x++;
```

you have now told the compiler to fetch the value in `x` (6) and assign it to `b`, and then go back and increment `x`. Thus, `b` is now 6, but `x` is now 7. Listing 4.3 shows the use and implications of both types.

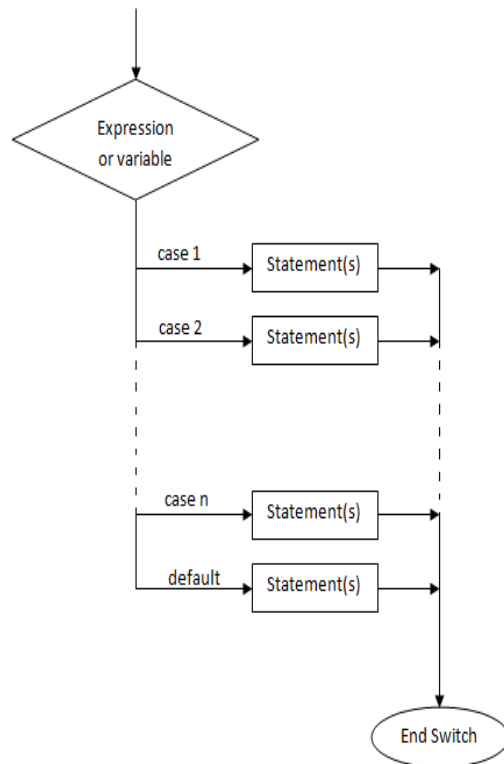
Switch statement

This statement is used to check variable for different values the variable different cases are executed.

Syntax :=>

Switch(variable)

```
{
case value :
-----
-----
break;
case value :
-----
-----
break;
case value :
-----
-----
break;
.
.
default :
-----
-
}
```



- If no matching case is found then code in the default block is executed.
- Default block is optional.
- We can use a break statement to throw program control out of a switch; otherwise, next cases will be executed without checking their values.

*** wap to read a single digit no. and print that no in words.**

```
# include <iostream.h>
void main()
{
    int n;
    cout << "Enter a no.";
    cin >> n;
    switch(n)
    {
        case 0: cout << "Zero";
                break;
        case 1: cout << "One";
                break;
        case 2: cout << "Two";
                break;
        case 3: cout << "Three";
                break;
        case 4: cout << "Four";
                break;
        case 5: cout << "Five";
                break;
        case 6: cout << "Six";
                break;
        case 7: cout << "Seven";
                break;
        case 8: cout << "Eight";
                break;
        case 9: cout << "Nine";
                break;
        default : cout << "Not a Single digit no.";
    }
}
```

* wap to read a single digit no. and print all no from that no. to 9 in words.

```
# include <iostream.h>
void main()
{ int n;
  cout << "Enter a no.";
  cin >> n;
  switch(n)
  {case 0: cout << "Zero";
   case 1: cout << "One";
   case 2: cout << "Two";
   case 3: cout << "Three";
   case 4: cout << "Four";
   case 5: cout << "Five";
   case 6: cout << "Six";
   case 7: cout << "Seven";
   case 8: cout << "Eight";
   case 9: cout << "Nine"; break;
   default : cout << "Not a Single digit no.";
  }
}
```

If statement vs Switch

- switch is usually more compact than lots of nested if else and therefore, more readable
- If you omit the break between two switch cases, you can fall through to the next case .No such problems with if statement
- Switch case values can only be constants.i.e. we can compare only with constant values. In if statement we can compare any type of values.
- switch only accepts only int ,char data types and float or double.
- Switch uses only equality operator for comparison.

Bitwise Operators

Bitwise operators are use to perform operations at bit level.

Operator	Symbol	Form	Operation
left shift	<<	$x \ll y$	all bits in x shifted left y bits
right shift	>>	$x \gg y$	all bits in x shifted right y bits
bitwise NOT	~	$\sim x$	all bits in x flipped
bitwise AND	&	$x \& y$	each bit in x AND each bit in y
bitwise OR		$x y$	each bit in x OR each bit in y
bitwise XOR	^	$x \wedge y$	each bit in x XOR each bit in y

```
#include <iostream.h> // program to calculate 2n
main()
{
    unsigned int a = 1;
    cout<<"Enter power ";
    cin>>n;
    a=a<<(n-1);
    cout<<"Result is "<<a<<endl;
}
```

```
#include <iostream>
using namespace std;

main()
{
    unsigned int a = 60;    // 60 = 0011 1100
    unsigned int b = 13;    // 13 = 0000 1101
    int c = 0;

    c = a & b;              // 12 = 0000 1100
    cout << "Line 1 - Value of c is : " << c << endl ;

    c = a | b;              // 61 = 0011 1101
    cout << "Line 2 - Value of c is: " << c << endl ;

    c = a ^ b;              // 49 = 0011 0001
    cout << "Line 3 - Value of c is: " << c << endl ;

    c = ~a;                 // -61 = 1100 0011
    cout << "Line 4 - Value of c is: " << c << endl ;

    c = a << 2;              // 240 = 1111 0000
    cout << "Line 5 - Value of c is: " << c << endl ;

    c = a >> 2;              // 15 = 0000 1111
    cout << "Line 6 - Value of c is: " << c << endl ;

    return 0;
}
```

Arrays

An array is a group of elements of same type sharing same name and stored in a consecutive memory location. Arrays are define if we want to store multiple values of same type.

Syntax :-

```
Datatype arrayname[size];
```

```
Eg. int a[100];
```

Note: Elements of an array can be accessed by using

```
syntax: arrayname[index]
```

Index no begins with zero .

*** wap to read an array to 10 integer and find sum of all the elements.**

```
# include <iostream.h>
void main()
{
    int a[10] , s=0 , i;
    cout << "Enter 10 no.'s";
    for( i = 0; i<=9 ; i++)
        cin >> a[i];
    for( i = 0; i<=9 ; i++)
        cout << "Sum is " << s;
}
```

*** wap to read temperature for 7 days of a week and find avg. temp. of that week.**

```
# include <iostream.h>
void main()
{ float a[7] , s=0 , avg ;
  int I;
  cout << "Enter temp. for seven days ";
  for( I = 0; I <=6 ; I++)
    cin >> a[I];
  for( I = 0; I<=6 ; I ++ )
    s= s + a[I];
  avg = s/7.0;
  cout << "average is " << avg;
}
```

*** wap to read an array of 10 integer and count all even no. and odd no. in that array.**

```
# include <iostream.h>
void main()
{
int a[10] , I , c=0 , d=0;
cout << "Enter 10 no.'s ";
for( I = 0; I <=9 ; I++)
  cin >> a[I];
for( I = 0; I<=9 ; I ++ )
  if(a[I]%2 == 0)
    c++;
  else
    d++;
cout << "Total even no. " << c << "\n Total odd no." << d;
}
```

*** wap to read an array of 10 integer and find greatest from that array.**

```
# include <iostream.h>
void main()
{
    int a[10] , g;
    cout << "Enter 10 no.'s ";
    for( I = 0; I <=9 ; I++)
        cin >> a[I];

    g= a[0];
    for( I = 0; I<=9 ; I ++ )
    {
        if (a[I] > g)
            g=a[I];
    }
    cout << "Largest no. is " << g;
}
```


Multidimensional Array

Multidimensional arrays are used to store matrix type of data and can be defined by using .

Syntax :-

Datatype arrayname [size1][size2][size3]...;

Eg. :-

Int [3][5]

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24

*** wap to read a matrix of size 3 * 5 and find sum of all the elements.**

```
# include <iostream.h>
void main()
{
    int a[3][5] , s=0 , I , j ;
    cout << "Enter a matrix of size 3 * 5 ";
    for( I = 0; I <=2 ; I++)
    {
        for( j = 0; j <=4 ; j++)
            cin >> a[I][j];
    }
    for( I = 0; I <=2 ; I++)
    {
        for( j = 0; j <=4 ; j++)
            s = s + a [I][j];
    }
    cout << "Sum is " << s;
}
```

*** wap to read a matrix of size 4 * 5 and count all even no. and odd no. in that matrix.**

```
# include <iostream.h>
void main()
{
    int a[4][5] , c=0 , d=0 , I , j ;
    cout << "Enter a matrix of size 4 * 5 ";
    for( I = 0; I <=3 ; I++)
    {
        for( j = 0; j <=4 ; j++)
            cin >> a[I][j];
    }
    for( I = 0; I <=3 ; I++)
    {
        for( j = 0; j <=4 ; j++)
            if (a[I][j]%2 == 0)
                c++;
            else
                d++;
    }
    cout << "Even no. is " << c;
    cout << "Odd no. is " << d;
}
```

Character Array

Character array are used to store strings. Whenever a string is stored into a character array its end is marked with a special character "*Null character*"("\0")

Eg :-

```
void main()
{
    char a[20];
    cout << "Enter your name";
    cin >> a;    OR    cin.getline(a,20);
    cout << "Hello" << a;
}
```

*** WAP to read a string and find length of that string**

```
# include <iostream.h>
void main()
{
    char a[20];
    int I=0;
    cout << "Enter your name";
    cin >> a;

    while (a[I] != '\0')
        I++;

    cout << "Length is " << I ;
}
```

*** WAP to read a string and count total no. of a's in that string.**

```
# include <iostream.h>
void main()
{
char a[20];
int I,c=0;
cout << "Enter your name";
cin >> a;
for (I=0;a[I]!='\0';I++)
    if (a[I]== 'a' || a[I]== 'A')
        c++;
cout << "Total no. of a's " << c ;
}
```

*** WAP to read a string and count total no. of Vowels in that string.**

```
# include <iostream.h>
void main()
{
char a[20];
int I,c=0;
cout << "Enter your name";
cin >> a;

for (I=0;a[I]!='\0';I++)
    if(a[I]=='a' || a[I]=='A' || a[I]=='e' || a[I]=='E' ||
        a[I]=='i' || a[I]=='I' || a[I]=='o' || a[I]=='O' ||
        a[I]=='u' || a[I]=='U')
        c++;

cout << "Total no. of vowels are " << c ;
}
```

String functions

C++ provide us different library functions to perform operations on strings. These functions are declared in the header file.

Old version :- string.h

New version :- cstring

1> int strlen (char *str) => returns length of the string.

```
#include < string.h >
#include<iostream.h>
void main()
{
    char a[20];
    cout << "Enter a string";
    cin >> a;
}
```

2>strupr (char *str)=> convert the string into uppercase.

```
#include<string.h>
#include<iostream.h>
void main()
{
    char a[20];
    cout << "Enter a string";
    cin >> a;
    strupr(a);
    cout << a;
}
```

3> **strlwr (char *str)=>** convert the string into lowercase.

```
#include<string.h>
#include<iostream.h>
void main()
{ char a[20];
  cout << "Enter a string";
  cin >> a;
  strlow(a);
  cout << a;
}
```

4> **strrev (char *str)=>** will reverse the contents of the string.

```
#include<string.h>
#include<iostream.h>
void main()
{ char a[20];
  cout << "Enter a string";
  cin >> a;
  strrev(a);
  cout << a;
}
```

5> **strcpy (char *dest, char *source)=>**
will copy source string at specified destination.

```
#include<string.h>
void main()
{ char a[20] , b[20];
  cout << "Enter a string";
  cin >> a;
  strcpy(b,a);
  cout << a;
}
```

6> **Strcat (char *s1, char *s2)=>**
It joints two strings.

Eg.:-

```
#include<string.h>
#include<iostream.h>
void main()
{
    char a[20];
    strcpy(a, "Seeta");
    strcpy(a, "Ram");
    cout << a;
}
```

7> **int Strcmp (char *s1, char *s2)=>**
will compare two strings and return difference
between their ASCII values.

Eg.:-

```
n = strcmp("Ram", "Ram");      =>    0
n = strcmp("Ram", "amit");     =>    +ve
n=  strcmp("ram", "Ram");      =>    -ve
```

Structures

structure is user defined data type that allows to combine data items of different kinds.

Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows –

```
struct [structure tag] {  
    member definition;  
    member definition;  
    ...  
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure –

```
struct Books {  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
} book;
```


Accessing Structure Members

To access any member of a structure, we use the member access operator (.)

<i>Syntax :</i> VariableName.MemberName
--

```
#include <iostream.h>
#include<string.h>
struct Books {
    char title[50];
    int  book_id;
};

int main( )
{
    struct Books Book1;      /* Declare Book1 of type Book */

    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    Book1.book_id = 6495407;

    /* print Book1 info */
    cout<< "Book 1 title : %s\n"<<Book1.title;
    cout<< "Book 1 book_id : %d\n"<< Book1.book_id;

    return 0;
}
```

Structure Initialization

While defining a structure variable we can directly store values into it.

Syntax: `struct tagname varaiblename={values....};`

For eg:

```
#include<iostream.h>
struct stud
{ int rollno;
  char name[20];
  char sex;
};
main()
{ struct stud a={4117,"Amit Jain",'m'};
  cout<<" Rollno is "<<a.rollno<<endl;
  cout<<" Name is "<<a.name<<endl;
  cout<<" Sex is "<<a.sex<<endl;
}
```

Anyonamoyus Structures

If a structure variable is defined while defining a structure then there is no need to name the structure. Such a structure is called as anyonymous structure.

For eg:

```
struct
{ int rollno;
  char name[20];
}a;

main()
{ cout<<"Enter the rollno,name=>";
  cin>>a.rollno>>a.name;
  cout<<"Rollno ="<< a.rollno<<" Name ="<< a.name;
}
```

Structure Vs Classes

- 1) structure :- In structure have a by default public.
In class have a by default private.
- 2) There is no data hiding features comes with structures. Classes do, private, protected and public.
- 3) A structure can't be abstract, a class can.
- 4) Structure are value type, They are stored as a stack on memory. where as class are reference type. They are stored as heap on memory.

Typedef

typedef is a keyword used to assign alternative names to existing types. Its mostly used with user defined data types, when names of data types get slightly complicated. Following is the general syntax for using typedef,

```
typedef existing_name alias_name
```

Lets take an example and see how typedef actually works.

```
typedef unsigned long ulong;
```

The above statement define a term **ulong** for an unsigned long type. Now this **ulong** identifier can be used to define unsigned long type variables.

```
ulong i, j ;
```

Functions

A function is a block of code designed to perform certain task.

Functions are of two types :-

1> **Functions returning values =>**

Such functions when called always return us some value

Eg.:-

```
Z = 100 + fact (5);
P = sqrt (10);
```

Such functions return us some value so they must always be used in an expression.

1> **Functions not returning values =>**

Such functions are called just to perform some task.

Eg.:-

```
clrscr();
textcolor(RED);
```

Such functions don't return any value so they can't be used in an expression.

General format to define a function

```
returntype functionName(datatype arg1,datatype arg2 ,...)
{
    statements
    -----
    -----
    return value;
}
```

returntype

It indicates the datatype of the value the function is going to return. It can be any datatype such as int, long, float, double, char, a pointer, a structure of all object etc.

If function is not returning any value then returntype must be void.

Note => If no returntype is specified then by default returntype is **int**.

***Design a function star which will display 50 stars**

```
# include<iostream.h>
void star()
{int I;
for(I=0;I<=50;I++)
    cout << "*" ;
}
void main()
{
    cout << endl << " CCIT " << endl;
    star();
    cout << endl << " Amravati " << endl;
    star();
}
```

Function with argument

While calling a function we can pass some data (actual parameters) this data is pass on to the function as arguments (formal parameters) and according to the argument values the function call perform different task.

```
Eg:- # include<iostream.h>
      void star(int N)
      { int I;
        for(I=0;I<=N;I++)
          cout << "*" ;
      }

void main()
{
  cout << endl << " CCIT " << endl;
  star(20);
  cout << endl << " Amravati " << endl;
  star(50);
}
```

***Design a function display which will display all numbers form 1 to n**

```
# include<iostream.h>
void display(int N)
{ int I;
  for(I=0;I<=N;I++)
    cout << I ;
}

void main()
{int a;
  cout << " Enter the no." ;
  cin << a;
  display(a);
}
```

Function returning values

If we want to use our function in an expression then such function must return a value by using return statement. Return statement returns the value at the point from where the function is called. +

```
# include<iostream.h>
int fact(int N)
{
    int I, f=1;
    for(I=1; I<=N; I++)
        f=f*I;
    return f;
}
void main()
{
    int z;
    z=100 + fact (5);
    cout << z;
}
```

*** Design a function intrest which will return simple intrest when three values p, r, & n are passed as arguments.**

```
# include<iostream.h>
float intrest(int p, int r, int n)
{
    float I;
    I = p * r * n /100.0;
    return I;
}
void main()
{
    float s;
    s= intrest(5000, 10.25, 3);
    cout << " intrest is " << s;
}
```

- * **Design a function volume which will return volume of a sphere when radius is pass as argument.**

```
# include<iostream.h>
float volume (int r)
{float v;
v = (4 / 3.0) * 3.14 * r * r * r;
return v;
}
void main()
{int a;
float b;
cout << "Enter radius";
cin >> a;
b= volume (a);
cout << "Volume of sphere is " << b;
}
```

- * **Design a function max. which will return greatest of two numbers.**

```
# include<iostream.h>
int max(int a , int b)
{
if (a>b)
return a;
else
return b;
}
void main()
{
int z;
z = max ( 4117 , 2636)
cout << " Greatest no. is " << z;
}
```


* **wap to read four no. a , b , c , & d and find greatest of them by using our previous max function.**

```
# include<iostream.h>

int max(int a , int b)
{if (a>b)
    return a;
else
    return b;
}

void main()
{
    int a , b , c , d , z ;
    cout << "Enter 4 no.s"
    cin >> a >> b >> c >> d;
    z = max ( max(a,b),max(c,d));
    cout << " Greatest no. is  " << z;
}
```

Local Variables

Variables defined within a function body is called as Local Variables . They are accessible only within the function .When the function returns, the local variables are no longer available.

Global Variables

Variables defined outside of any function have global scope and thus are available from any function in the program, including `main()` . They are present for the life time of the program.

If a function has a variable with the same name as a global variable, the name refers to the local variable--not the global--when used within the function.

Function Overloading

*Defining multiple functions in a program having same name is called as **function overloading**.* Only precaution to be taken is that function must have different no of or type of argument. Compiler will decide which to call depending on no. of or type of arguments passed while calling the function.

Eg.

```
# include<iostream.h>
int volume(int L, int B, int H)      // function to
{                                     // calculate
    int v=L*B*H;                     // volume of
    return v;                         // box
}

int volume(int N)                    // function to
{                                     // calculate
    int v=N*N*N;                     // volume of
    return v;                         // cube
}

void main()
{
    int a=volume(5,7,8);
    cout<<"Volume of box is "<<a;

    int b=volume(5);
    cout<<"Volume of cube is "<<b;

}
```

Default Arguments

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead.

Eg.

```
# include<iostream.h>
//this function can be called by passing 3 or 2 args
float intrest(int p, int r, int n=1)
{
    float I;
    I = p * r * n /100.0;
    return I;
}

void main()
{
    float s;
    s= intrest(5000, 10.25, 3);
    cout << " intrest is " << s;
    s= intrest(5000, 10.25);
    cout << " intrest is " << s;
    // if 3rd argument is not passed then it will
    // take the default value n=1
}
```

Function Prototype

If a function is called before its definition then at least its prototype must be defined by using syntax

```
returntype functionName( datatype of args);
```

The Function prototype serves the following purposes –

- 1) It tells the return type of the data that the function will return.
- 2) It tells the number of arguments passed to the function.
- 3) It tells the data types of the each of the passed arguments.
- 4) Also it tells the order in which the arguments are passed to the function.

NOTE: If function contains default args then they must be declared in function prototype

Eg.

```
# include<iostream.h>

int fact(int); // prototype of function fact

void main()
{ int z;
  z=100 + fact (5);
  cout << z;
}

int fact(int N)
{int I, f=1;
 for(I=1;I<=N;I++)
   f=f*I;
 return f;
}
```

Reference

A reference is an alias; when you create a reference, you initialize it with the name of another object, the target. From that moment on, the reference acts as an alternative name for the target, and anything you do to the reference is really done to the target.

You create a reference by writing the type of the target object, followed by the reference operator (&), followed by the name of the reference

<code>datatype &referenceName=variable;</code>

Eg.

```
void main( )
{
    int a=5;
    int &b=a; //define reference for a
    int &c=b; //define another reference for a

    cout<<a<<b<<c; // o/p will be 5 5 5

    a++;
    b++;
    c++;

    cout<<a<<b<<c; // o/p will be 8 8 8
                  // as a b & c are name of
                  // same variable.
}
```

Reference Arguments

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value.

Advantages of passing by reference:

- It allows a function to change the value of the argument, which is sometimes useful.
- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.
- References can be used to return multiple values from a function.
- References must be initialized, so there's no worry about null values.

Disadvantages of passing by reference:

- Because a non-const reference cannot be made to an rvalue (e.g. a literal or an expression), reference arguments must be normal variables.
- It can be hard to tell whether a parameter passed by non-const reference is meant to be input, output, or both.
- It's impossible to tell from the function call whether the argument may change. An argument passed by value and passed by reference looks the same.

When to use pass by reference:

- When passing structs or classes (use const if read-only).
- When you need the function to modify an argument.

When not to use pass by reference:

- When passing fundamental types (use pass by value).

So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
void swap(int& x, int& y);
int main ()
{ // local variable declaration:
  int a = 100;
  int b = 200;
  cout << "Before swap, value of a :" << a << endl;
  cout << "Before swap, value of b :" << b << endl;
  /* calling a function to swap the values.*/
  swap(a, b);
  cout << "After swap, value of a :" << a << endl;
  cout << "After swap, value of b :" << b << endl;
  return 0;
}
// function definition to swap the values.
void swap(int& x, int& y)
{ int temp;
  temp = x; /* save the value at address x */
  x = y;    /* put y into x */
  y = temp; /* put x into y */
  return;
}
```


Returning values by reference

When a function returns a reference, it returns an implicit pointer to its return value. This way, a function can be used on the left side of an assignment statement. For example, consider this simple program:

```
double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};

double& setValues( int i )
{   return vals[i];    // return a reference to the ith element
}

int main ()
{   cout << "Value before change" << endl;
    for ( int i = 0; i < 5; i++ )
    {   cout << "vals[" << i << "] = ";
        cout << vals[i] << endl;
    }
    setValues(1) = 20.23; // change 2nd element
    setValues(3) = 70.8;  // change 4th element
    cout << "Value after change" << endl;
    for ( int i = 0; i < 5; i++ )
    {   cout << "vals[" << i << "] = ";
        cout << vals[i] << endl;
    }
}
```

Mathematical functions

Math library functions allow the programmer to perform a number of common mathematical calculations. These functions are defined in header file `math.h` (old) / `cmath` (new)

<code>cos(x)</code>	returns cosine of x , x in radians
<code>sin(x)</code>	returns sine of x , x in radians
<code>tan(x)</code>	returns tangent of x , x in radians
<code>exp(x)</code>	exponential function, e to power x
<code>log(x)</code>	natural log of x (base e), $x > 0$
<code>sqrt(x)</code>	square root of x , $x \geq 0$
<code>fabs(x)</code>	absolute value of x
<code>floor(x)</code>	largest integer not greater than x
<code>ceil(x)</code>	smallest integer not less than x
<code>abs(x)</code>	Returns absolute value

Storage classes

In C++ language, each variable has a **storage class** which decides scope, visibility and lifetime of that variable. The following storage classes are most oftenly used in C++ programming,

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

Automatic variables

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

```
void main()  
{  
    int detail; or auto int detail;    //Both are same  
}
```

External or Global variable

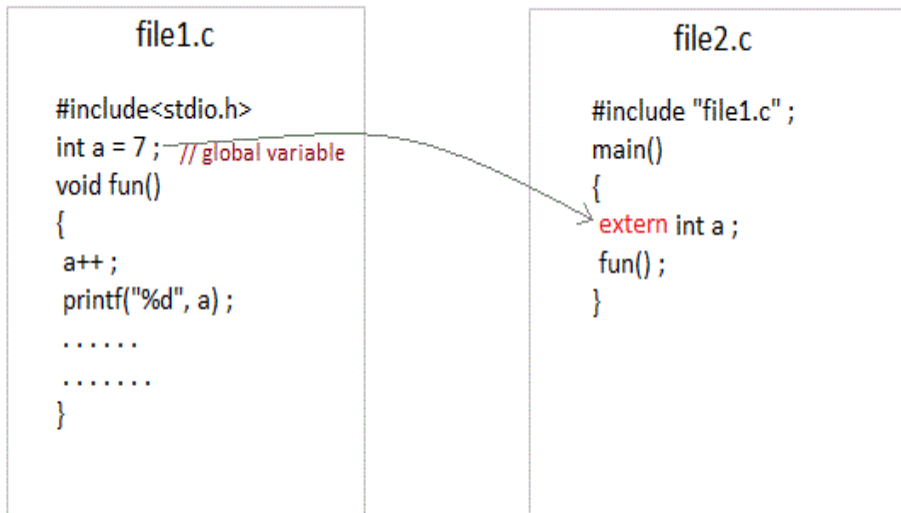
A variable that is declared outside any function is a **Global variable**. **Global** variables remain available throughout the entire program. One important thing to remember about global variable is that their values can be changed by any function in the program.

```
int number;  
void main()  
{  
    number=10;  
}  
fun1()  
{  
    number=20;  
}  
fun2()  
{  
    number=30;  
}
```

Here the global variable **number** is available to all three functions.

extern keyword

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



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

Register variable

Register variable inform the compiler to store the variable in register instead of memory. **Register** variable has faster access than normal variable. Frequently used variables are kept in register. Only few variables can be placed inside register.

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

For eg: register int number;

Static variables

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

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

```
void test();    //Function declaration

main()
{
    test();
    test();
    test();
}

void test()
{
    static int a = 0;    //Static variable
    a = a+1;
    printf("%d\t",a);
}
```

output : 1 2 3

OOPS

C++ is an object oriented language. It provides us a programming environment where we can create objects and perform operations on them. The key features of OOP (object-oriented programming languages) are

Abstraction

Abstraction is the concept of taking some object from the real world, and converting it to programming terms. **Abstraction** in **Object Oriented Programming** helps to hide the irrelevant details of an object. **Abstraction** is separating the functions and properties that logically can be separated to a separate entity which the main type depends on.

Encapsulation

Encapsulation in C++ is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class, therefore it is also known as data hiding.

Inheritance

Inheritance is the ability, offered by many OOP languages, to derive a new class (the derived or inherited class) from another class (the base class). The derived class automatically inherits the properties and methods of the base class.

For example, you could define a generic Shape class with properties such as Color and Position and then use it as a base for more specific classes (for example, Rectangle, Circle, and so on) that inherit all those generic properties.

Message

A **message** is a request to an object to invoke one of its methods. A message therefore contains

- the **name** of the method and
- the **arguments** of the method.

This interacting between objects is based on *messages* which are sent from one object to another asking the recipient to apply a method on itself. we could create new objects and invoke methods on them. For example,

```
Account a(4117,5000) ;
/* Define a new Account object with account no 4117 & balance 5000 */
a.deposit(1500); /* deposit 1500 into account */
```

Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Object

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. **Object is an instance of a class.** Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

An object has two characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.

Objects Vs Classes

Object	Class
Object is an instance of a class.	Class is a blueprint or template from which objects are created.
Object is a real world entity such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a group of similar objects .
Object is a physical entity.	Class is a logical entity.
Object is created through new keyword mainly e.g. Student s1;	Class is declared using class keyword e.g. class Student{}
Object is created many times as per requirement.	Class is declared once .
Object allocates memory when it is created .	Class doesn't allocated memory when it is created .
There are many ways to create objects .	There is only one way to define class in C++ using class keyword.

Classes

A class is a generic definition of an object. It is a blue print of an object.

Definition:-

A class is a user defined datatype where we can bind its data and its related functions together.

```
Syntax:-      class name
                {
                Data Members
                -----
                Member Function
                -----
                };
```

A class is defined in terms of its data members and member function.

Data members:-

They indicate informations about objects or current state of objects.

```
Syntax:-      Datatype member name;
```

Member Function:-

It indicates operation which we can perform on an object.

```
Syntax:-
    returntype function name(args.)
    {
    -----
    .....
    .....
    return value;
    }
```

*** Design a class rectangle containing data members length & breadth and member function area & perimeter**

```
class rectangle
{
    int length,breadth;

    void area()
    {
        area=length*breadth;
        cout<<"area is"<<area;
    }

    void perimeter()
    {
        p=2*(length*breadth);
        cout<<"perimeter is"<<p;
    }
}
```

*** .Design a class sphere containing data members radius and member function volume.**

```
class sphere
{
    int r;

    void volume()
    {
        float v;
        v=(4*3.14*r*r*r)/3;
        cout<<"volume is"<<v;
    }
}
```

OBJECTS

A class provides the blueprints for objects, so basically an object is created from a class i.e. An object is an instance of class.

In C++ Objects can be created in different ways

1] While defining a class we can specify list of objects after its ending brace.

```
class ClassName  
{  
.....  
.....  
}object list;
```

2] After defining class objects can be created whenever required using syntax:-

```
ClassName objectlist;
```

Eg; rectangle a,b,c;

3] Objects can also be dynamically created by using operator new.

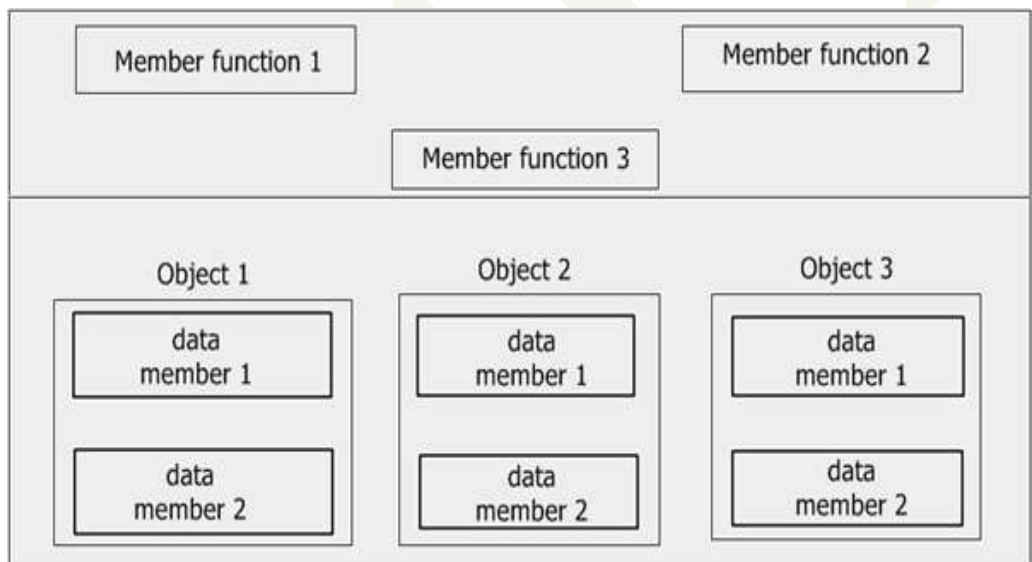
```
Pointer =new className(args...);
```

Memory Allocation for Objects

Whenever an object is created space is reserved for its data members. Data of different objects may be different but code of functions is common for all objects.

For ex:

Suppose a class contains 2 data members and 3 member functions and 3 objects are created of that class then memory will be allocated as



Accessing members:

Accessing a member depends solely on the access control of that data member. If its public, then the member can be easily accessed using the direct member access (.) operator with the object of that class.

Syntax:- **objectname . membername;**

Access specifiers:

- It indicate accessibility of member of class.
- It can be private , public or protected.

Private:-

Private keyword, means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private. Generally datamembers are kept private,because we want to hhide internal details of object.

Public:-

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public. Generally members function are kept public.

Protected:-

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class.

Setter function

These are used to set value of private data members.
Generally the name start with 'set'

Eg:

```
#include<iostream.h>
class rectangle
{
    private:
    int length,breadth;
    public:
    void area()
    {
        int a=length*breadth;
        cout <<"area is "<<a;
    }
    void perimeter()
    {
        int p=2*(length+breadth);
        cout<<"perimeter is"<<p;
    }
    void setdimension(int m,int n)
    {
        length=m;
        breadth=n;
    }
}
void main()
{
    rectangle a,b;
    a.setdimension(4,7);
    b.setdimension(3,3);
    a.area();
    b.perimeter();
}
```

*** design a class box containing data member length breadth height and member function set diamention and volume.**

```
#include<iostream.h>
class box
{
private:
    int  length,breadth,height;

public:
void volume()
    {
        int v=length*breadth*height;
        cout<<"volume is"<<v;
    }

    void setdiamention(int m,int n,int p)
    {
        length=m;
        breadth=n;
        height=p;
    }
};

void main()
{
    box a,b;
    a.setdiamention(4,5,7);
    b.setdiamention(3,2,5);
    a.volume();
    b.volume();
}
```


*** Design a class worker containing wages and working days and member function setdata and payment**

```
#include<iostream.h>
class worker
{
private:
    int wages,working_days;

public:

    void payment()
    {
        int p=wages*working_days;
        cout<<"payment is"<<p;
    }

    void setdata(int m,int n)
    {
        wages=m;
        working_days=n;
    }
};

void main()
{
    worker a,b;
    a.setdata(230,27);
    b.setdata(210,20);
    a.payment();
    b.payment();
}
```

***Design a class account containing data members accno,balance and member function setdata and interest .**

```
#include<iostream.h>
class account()
{
    private:
    int accno,balance;
    public:
    void setdata(int m,int n)
    {
        accno=m;
        balance=n;
    }
    void interest(float r,int n)
    {
        float i;
        i=balance*n*r;
        cout<<"interest is"<<i;
    }
};
void main()
{
    account a,b;
    a.setdata(4114,5000)
    b.setdata(3012,7000)
    a.interest(10.25,3);
    b.interest(11.75,2);
}
```

Member function returning values:

If we want to use our member function in an expression then such member function must return a value by using return statement .

```
#include<iostream.h>
class worker
{
    int wages,working_days;

public:

    int payment()
    {
        int p=wages*working_days;
        cout<<"payment is"<<p;
        return p;
    }

    void setdata(int m,int n)
    {
        wages=m;
        working_days=n;
    }
};

void main()
{
    worker a,b;
    a.setdata(230,27);
    b.setdata(210,20);
    int z=a.payment()+ b.payment();
    cout<<"total payment is"<<z;
    cout<<"payment of a is a. is"<<a.payment();
    cout<<"payment of b is"<<b.payment();
}
```

Member function with default arguments

While declaring member function we can assign some default values to its argument so that same function can be called in different ways

Eg:=

```
#include<iostream.h>
class account()
{
int accno,balance;
public:
    void setdata(int m,int n)
    {
        accno=m;
        balance=n;
    }
    void interest(float r,int n=1)
    {
        float i;
        i=balance*r*n/100;
        cout<<"interest is"<<i;
    }
};
void main()
{
    account a,b;
    a.setdata(4114,5000)
    b.setdata(3012,7000)
    a.interest(10.25,3);
    b.interest(11.75,2);
}
```

Member function overloading:-

Defining multiple member function in class having same name is called as member function overloading. only precaution is taken that no of argument must be different.

```
#include<iostream.h>
class box
{
    int length,breadth,height;
public:
    void volume()
    {
        int v=length*breadth*height;
        cout<<"volume is"<<v;
    }
    void setdiamention(int m,int n,int p)
    {
        length=m;
        breadth=n;
        height=p;
    }
    void setdiamention(int x)
    {
        length=x;
        breadth=x;
        height=x;
    }
};
void main()
{
    box a,b,c;
    a.setdiamention(4,5,7);
    b.setdiamention(3,2,5);
    c.setdaimention(5);
    a.volume();
    b.volume();
    c.volume();
}
```

*** Design a class account containing data members accno and balance and member function open ,deposit,withdraw,showbalance**

```
#include<iostream.h>
class account
{
    int accno,balance;
public:
    void open(int x,int y)
    {
        accno=x;
        balance=y;
    }
    void deposit(int d)
    {
        balance=balance+d;
        cout<<"balance is"<< balance;
    }
    void withdraw(int n)
    {
        balance=balance-n;
        cout<<"balance is"<<b;
    }
    void showbalance()
    {
        cout<<"balance is"<<balance;
    }
};

void main()
{
    account a,b;
    a.open(4117,5000);
    b.open(3010,7000);
    a.deposit(3000);
    b.withdraw(2000);
    a.deposit(5000);
    a.showbalance();
    b.showbalance();
}
```

Private Member functions

You should make a function private when you don't need other objects or classes to access the function, when you'll be invoking it from within the class.

```
#include<iostream.h>
class account
{
    int accno,balance;
public:
    void open(int x,int y)
    {
        accno=x;
        balance=y;
    }
    void deposit(int d)
    {
        balance=balance+d;
    }
    void withdraw(int n)
    {if(canwithdraw(n))
        balance=balance-n;
        else
            cout<<"Cannot withdraw";
    }
    void showbalance()
    {
        cout<<"balance is"<<balance;
    }
private:
    int canwithdraw(int amt)
    {return (amt<=balance)?1: 0;
    }
};
void main()
{
    account a;
    a.open(4117,5000);
    b.withdraw(2000);
    a.showbalance();
}
```

Constructors

Constructors Are special member functions of a class which gets automatically invoked when ever an object of its class is created.

Characteristics

- They have the same name of the class name.
- They doesn't have any return type not even void.
- They are used for initialization of the objects.
- Multiple constructors can be defined in a class
- Constructors can have default arguments.
- C++ automatically adds an empty constructor without arguments if constructor is not defined
- Constructors are implicitly called when objects are created But they can also be explicitly called if required.

Ex.

```

class circle
{
    int r;
public:
    void setradius (int n)
    {
        r = n;
    }
    void area ()
    {
        double a = 3.14 * r * r;
        cout<<"area is "<< a;
    }
    circle ()           //constructor
    {
        r = 1;
    }
};

void main ( )
{
    Circle a, b;
    a.setradius (7);
    a.area ();
    b.area ();
}

```

Constructors Vs Member Functions

- Constructor name must be same as class name but functions cannot have same name as class name.
- Constructor does not have return type whereas functions must have.
- Member function can be virtual, but, there is no concept of virtual constructor in C++.
- Constructors are invoked at the time of object creation automatically and can be called explicitly but functions are called explicitly using class objects.

Parameterised constructor

while creating an object we can pass some data. This data is passed on to constructor as an argument where it is used for initialization of object.

Note: If class contains a constructor with arguments & it doesn't have any default constructor then we have to compulsorily call this constructor by passing data.

Eg.

```
class circle
{
    int r;
public:
    void area( )
    {
        float a=3.14*r*;
        cout<<"area is"<<a;
    }

    circle(int n)           //constructor
    {                       //with argument
        r =n;
    }
};

main( )
{
    circle a(5), a(7);
    a.circle( );
    b.area( );
}
```

Constructor overloading

Defining multiple constructors in class is called as constructor overloading. Only precaution to be taken is that type of argument must be different. An appropriate constructor will be called depending on no of argument or type of argument that are passed while creating an object.

Eg:-

```
class circle
{
    int r;
    public:
    void area( )
    {
        float a=3.14*r*;
        cout<<"area is"<<a;
    }

    circle(int n)          // overloaded constructors
    {
        r=n;
    }
    circle( )
    {
        r=1;
    }
};
main( )
{
    circle a(5),b(7),c;
    a.area a( );
    b.area( );
    c.area( );
}
```

Copy Constructor

It is used to initialize an object by using data of another object of same type. The copy constructor is automatically called when an object is assigned to it. The reference of the object which is assigned comes as argument to copy constructor. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (const classname &obj)
{
    // body of constructor
}
```

Here, **obj** is a reference to an object that is being used to initialize another object.

For Eg:

```
class circle
{
    int r;
public:
    void area( )
    {
        float a=3.14*r*;
        cout<<"area is"<<a;
    }

    circle(int n)
    {
        r=n;
    }
    circle(circle &z)           // Copy constructor
    {
        r=z.r;
    }
};
main( )
{
    circle a(5);

    circle b=a;           // or circle b(a);

    a.area a( );
    b.area( );
}
```

Constructor with default argument

while declaring constructor we can assign some default value to its argument. so that same constructor will be used to initialize different ways of object.

Eg.

```
class circle
{
    int r;
public:
    void area( )
    {
        float a=3.14*r*;
        cout<<"area is"<<a;
    }
    circle(int n=1)    // constructor with
    {                  // default argument
        r=n;
    }
};

void main( )
{
    circle a(5),b(7),c;
    a.area( );
    b.area( );
    c.area( );
}
```

*** Design a class box whose object can be created in 3 different ways by passing 3,1 or without argument.**

```
#include<iostream.h>
class box
{
    int length,breadth,height;
public:
    void volume( )
    {
        int v;
        v=length*breadth*height;
        cout<<"volume is"<<v;
    }
    box(int l,int b,int h)
    {
        length=l;
        breadth=b;
        height=h;
    }
    box(int m)
    {
        length=m;
        breadth=m;
        height=m;
    }
    box( )
    {
        length=1;
        breadth=1;
        height=1;
    }
};
main( )
{
    box a(5,4,7),b(5),c;
    a.volume( );
    b.volume( );
    c.volume( );
}
```

Destructors

A **destructor** is a special member function of a class that is executed whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

```
~ClassName()  
{  
    Statements..  
    .....  
}
```

Characteristics of Destructors

- Do not accept arguments.
- Cannot specify any return type (including **void**).
- Cannot return a value using the **return** statement.
- Cannot be declared as **const**, **volatile**, or **static**.
- Can be declared as **virtual**. Using virtual destructors, you can destroy objects without knowing their type .


```
#include<iostream.h>
class Numbers
{
    int *data;
    int count;
public:
    //constructor
    Numbers(int cnt)
    {
        count=cnt;
        data=new int[count];
        cout<<"memory reserved"<<endl;
    }
    //destructor
    ~Numbers()
    {
        delete data;
        cout<<"memory released"<<endl;
    }
    void readdata()
    {
        cout<<"Enter "<<count<<" nos "<<endl;
        for(int i=0;i<count;i++)
            cin>>data[i];
    }
    void sum()
    {
        int s=0;
        for(int i=0;i<count;i++)
            s=s+data[i];
        cout<<"Sum is "<<s<<endl;
    }
};
```

```

void main()
{
Numbers a(10);
a.readdata();
a.sum();
}

```

Constructors Vs Destructors

Constructors	Destructors
Constructor is used to initialize the instance of a class .	Destructor destroys the objects when they are no longer needed.
Constructor is Called when new instance of a class is created.	Destructor is called when instance of a class is deleted or released.
Constructor allocates the memory .	Destructor releases the memory .
Constructors can have arguments.	Destructor can not have any arguments .
Overloading of constructor is possible.	Overloading of Destructor is not possible.
Constructor has the same name as class name.	Destructor also has the same name as class name but with (~) tiled operator .
ClassName(Arguments) { //Body of Constructor }	~ ClassName() { }

Member functions

Member functions of class can be defined in the class as well as outside the class.

syntax:-

```
returntype function name : : function name (args...)
{
    statements;
    -----
    return value;
}
```

Note: If we want to define them outside the class then at least their prototype must be declared in the class.

```
class circle
{
    int r;
    public:
        float area( );
        circle(int);
};
float circle :: area( )
{
    float a=3.14*r*;
    return a;
}
circle :: circle(int n)
{
    r=n;
}
main( )
{
    circle a (5);
    cout<<"area is"<<a.area( );
}
```

***Design a class rect containing data members length and breadth and member function area and perimeter.**

```
class rect
{
    int length,breadth;
public:
    int area( ) ;
    int perimeter( );
    rect(int,int);
};

int rect :: area( )
{
    int a=length*breadth;
    return a;
}

int rect :: perimeter( )
{
    int p=2*(length*breadth);
    return p;
}

main( )
{
    rect a(5,7);
    cout<<"area is"<<a.area( );
    cout<<"perimeter is"<<a.perimeter( );
}
```

Note:

1. If a member function of class is define outside the class then it increases readability of class. Such programs are easy to manage, maintain and Debug.
2. If a function is define in the class then compiler will be default treat that function as inline function.

Inline functions

If a function is declared as inline then while compilation compiler will replace compiler call with code of function instead of its address. Due to this execution speed of program increases as overhead require to call a function are removed.

A function can be declared as inline by using a keyword inline before its function prototype or by defining a function in the class. Declaring a function as inline is required to compiler and not a command.

Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

- If a function contains a loop. (for, while, do-while)
- If a function contains static variables.
- If a function is recursive.
- If a function return type is other than void, and the return statement doesn't exist in function body.
- If a function contains switch or goto statement.

Advantages:

- Function call overhead doesn't occur.
- It also saves the overhead of push/pop variables on the stack when function is called.
- It also saves overhead of a return call from a function.
- Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Disadvantages:

- The added variables from the inlined function consumes additional registers
- If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.
- Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.
- Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled.
- Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

Macro Vs Inline Functions

- Inline follows strict parameter type checking, macros do not.
- Macros are always expanded by preprocessor, whereas compiler may or may not replace the inline definitions.

For ex:

```
class circle
{
    int r;
public:
    void setradius(int n)
    {
        r=n;
    }
    inline void area( );
    void circumference( );
};
void circle::area( )
{
    float a=3.14*r*r;
    cout<<"area is"<<a;
}
void circle::circumference( )
{
    float c=2*3.14*r;
    cout<<"circumference is"<<c;
}
main()
{
    circle a;
    a.setradius(5); //will be treated as inline
    a.area();        //will be treated as inline
    a.circumference(); //will be called like
}                    //regular function
```

Friend functions

These are function which can access private or protected members of an object. Normally, a function that is defined outside of a class cannot access such information. A friend function is declared by the class that is granting access, explicitly stating what function from a class is allowed access. A similar concept is that of friend class.

Advantages

1. They provide a degree of freedom in the interface design options.
2. We can able to access the other class members in our class if,we use friend keyword.
3. We can access the members without inheriting the class.

Disadvantages

1. The major disadvantage of friend functions is that they require an extra line of code when you want dynamic binding.
2. A derived class can't inherit friend functions.
3. Since the friend function can access to the data members and member functions which may be either private or public of any class remaining outside of the class so it can break the security.

A friend function is called by passing object as argument. Such a function has to be declared as friend within class.

Syntax:- ***friend returntype functionname(datatype of arguments);***

This declaration can be placed in any section private or public

*** Design a friend function payment which will print payment of worker passing as argument .**

```
class worker
{
    int wages,wdays;
public:
    worker(int a,int b)
    {
        wages=a;
        wdays=b;
    }
    friend void payment(worker);
};

void payment(worker z)
{
    int p=z.wages*z.wdays
    cout<<"payment is"<<p;
}

main( )
{
    worker a(210,25), b(130,25);
    payment(a);
    payment(b)
}
```

*** Design a friend function convert which will convert value of feet object in inches**

Eg.

```
#include<iostream.h>
class feet
{
private:
    int feet,inches;
public:
    feet(int a,int b)
    {
        feet=a;
        inches=b;
    }
    friend void convert(feet);
};
int convert(feet z)
{
    int c=12*z.feet+z.inches;
    return c;
}
void main( )
{
    feet a(2,7),b(3,1)
    cout<<"value of a is "<<convert(a)<<" inches";
    cout<<"value of b is "<<convert(b) <<" inches";
}
```

Friend Class

A class can also be declared to be the friend of some other class. When we create a friend class then all the member functions of the friend class also become the friend of the other class. This requires the condition that the friend becoming class must be first declared or defined (forward declaration).

```
#include <iostream>
using namespace std;

class Square;

class Rectangle
{
    int width, height;

public:
    Rectangle(int w = 1, int h = 1)
    {
        Width=w;
        Height=h;
    }
    void display()
    {
        cout << "Rectangle: " << width * height << endl;
    };
    void morph(Square &);
};
```

```
class Square
{
    int side;

public:
    Square(int s = 1)
    {
        side=s;
    }
    void display()
    {
        cout << "Square: " << side * side << endl;
    };
    friend class Rectangle;
};

void Rectangle::morph(Square &s) {
    width = s.side;
    height = s.side;
}

int main () {
    Rectangle rec(5,10);
    Square sq(5);
    cout << "Before:" << endl;
    rec.display();
    sq.display();

    rec.morph(sq);
    cout << "\nAfter:" << endl;
    rec.display();
    sq.display();
    return 0;
}
```

Object As Function Arguments

The objects of a class can be passed as arguments to member functions as well as nonmember functions either **by value** or **by reference** or **by address**.

By Value:

When an object is passed by value, a copy of the actual object is created inside the function. This copy is destroyed when the function terminates. Moreover, any changes made to the copy of the object inside the function are not reflected in the actual object.

By Reference:

In pass by reference, only a reference to that object (not the entire object) is passed to the function. Thus, the changes made to the object within the function are reflected in the actual object.

By Address:

In pass by Address, a address of that object (not the entire object) is passed to the function. Thus, the changes made to the object within the function by using that address are reflected in the actual object.

Example of pass by value

```
#include<iostream.h>
class feet
{
private:
    int feets,inches;
public:
    feet(int a,int b)
    {
        feets=a;
        inches=b;
    }
    void display()
    {
        cout<<feets<<" Feets "<<inches<<" Inches"<<endl;
    }
    friend void increment(feet);
};

void increment(feet z)
{
    z.inches++;
    if(z.inches==12)
    {
        z.feets++;
        z.inches=0;
    }
}

void main( )
{
    feet a(2,7);
    a.display(); //will display 2 Feets 7 Inches
    increment(a);
    a.display(); //will display 2 Feets 7 Inches
}                // i.e. no changes as pass by value.
```

Example of pass by reference

```
#include<iostream.h>
class feet
{
private:
    int feets,inches;
public:
    feet(int a,int b)
    {
        feets=a;
        inches=b;
    }
    void display()
    {
        cout<<feets<<" Feets "<<inches<<" Inches"<<endl;
    }
    friend void increment(feet&);
};

void increment(feet &z)
{
    z.inches++;
    if(z.inches==12)
    {
        z.feets++;
        z.inches=0;
    }
}

void main( )
{
    feet a(2,7);
    a.display(); //will display 2 Feets 7 Inches
    increment(a);
    a.display(); //will display 2 Feets 8 Inches
}                // i.e. changes as pass by reference.
```

Example of pass by address

```
#include<iostream.h>
class feet
{
private:
    int feets,inches;
public:
    feet(int a,int b)
    {
        feets=a;
        inches=b;
    }
    void display()
    {
        cout<<feets<<" Feets "<<inches<<" Inches"<<endl;
    }
    friend void increment(feet*);
};

void increment(feet *z)
{
    z->inches++;
    if(z->inches==12)
    {
        z->feets++;
        z->inches=0;
    }
}

void main( )
{
    feet a(2,7);
    a.display(); //will display 2 Feets 7 Inches
    increment(a);
    a.display(); //will display 2 Feets 8 Inches
}                // i.e. changes as pass by address.
```


Arrays of object

If we require multiple object of same type then an array can be define by using syntax:-

className arrayname[size];

```
#include<iostream.h>
class person
{
    char name[20],city[20];
public:
    void getdata( )
    {
        cout<<"enter the name and city"<<;
        cin>>name>>city;
    }
    void putdata( )
    {
        cout<<"name is"<<name;
        cout<<":city is"<<city;
    }
};

void main( )
{
    person a[5];

    for(int i=0;i<=a;i++)
        a[i].getdat( );

    for(i=0;i<=4;i++)
        a[i].putdata( );
}
```

Arrays of object Initilization

Array of objects can be Initilized by using syntax

```
classname arrayname[size]={classname(arg1,..),.....};
```

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

class student
{
int rollno;
char name[20];
public :
    student(int rn,char nm[])
    {
        rollno=rn;
        strcpy(name,nm);
    }
    void showinfo()
    {
        cout<<"RollNo "<<rollno<<endl;
        cout<<"Name "<<name<<endl;
    }
};

void main()
{
student a[2]={    student(4117,"Amit Jain"),
                student(3012,"Gopal Pandey")};
for(int i=0;i<2;i++)
    a[i].showinfo();
}
```

Array as class Member

Like primitive data types we can use arrays as class data members.

```
#include<iostream.h>
class student
{
int rollno;
int marks[5];
public :
    void readinfo()
    {
        cout<<"Enter RollNo and Marks for 5 subjects :";
        cin>>rollno;
        for(int i=0;i<=4;i++)
            cin>>marks[i];
    }
    void showinfo()
    {
        cout<<"RollNo "<<rollno<<endl;
        int t=0;
        for(int i=0;i<5;i++)
            t=t+marks[i];
        cout<<"Total is "<<t<<endl;
    }
};

void main()
{
    student a;
    a.readinfo();
    a.showinfo();
}
```

Static data members

If a data member of a class is declared as static then only one copy of that data members is created for entire class and all objects of that class can store that data. So if multiple objects want to share some values then such data must be declared as static.

Space for static data member has to be specially defined after the definition of the class by using syntax:

<code>datatype classname :: staticMembeName=value;</code>

Features of Static Data Members

- Only one copy of that member is created for the entire class and is shared by all the objects of that class, so it is also called class data member.
- It can be accessed not only by the object name but also with the class name.
- It is initialized to zero if no default value is assigned to it.
- It is visible only within the class, but its lifetime is the entire program.

*** Design a class account containing data member accno,balance and irate.(irate must be shared by all objects)
Default value of irate=10.25**

```
#include<iostream.h>

class account
{
    int accno,balance;
    static float irate;
public:
    account(int an,intb)
    {
        accno=an;
        balance=b;
    }
    void interest( )
    {
        float i=accno*irate*balance/100;
        cout<<"interest is"<<i;
    }
};
float interest : : irate=10.25;
main( )
{
    account a(4117,5000),b(3012,7000)
    a.interest(3);
    b.interest(2);
}
```

Static member functions

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

Class name :: function name(arguments....);
--

Note: In static member function we can access only other static members.

Features of static functions

- In static member function we can access only other static members.
- static member functions do not use this pointer.
- A static member function cannot be virtual.
- Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.
- A static member function can not be declared *const*, *volatile*, or *const volatile*.

```

#include<iostream.h>

class account
{
    int accno,balance;
    static float irate;
public:
    account(int an,intb)
    {
        accno=an;
        balance=b;
    }
    void interest(int n)
    {
        float i=accno*irate*balance/100;
        cout<<"interest is"<<i;
    }
    static void changeirate(float r) //can be called
    {      irate=r;                //without
          }                          // creating objects
};

float account : : irate=10.25;

main()
{
    account a(4117,5000),b(3012,7000);
    a.interest(3);
    b.interest(2);
    account ::changeirate (11.75);
    a.interest(3);
    b.interest(2);
}

```

Non-static Vs Static Data Members

Non Static Data Members / Object Members	Static Data Members/ Class Members
<p>These variable should not be preceded by any static keyword Example:</p> <pre>class A { int a; }</pre>	<p>These variables are preceded by static keyword. <i>Example</i></p> <pre>class A { static int b; }</pre>
Memory is allocated for these variable whenever an object is created	Memory is allocated for these variable at the time of loading of the class.
Memory is allocated multiple time whenever a new object is created.	Memory is allocated for these variable only once in the program.
Non-static variable also known as instance variable while because memory is allocated whenever instance is created.	Memory is allocated at the time of loading of class so that these are also known as class variable.
Non-static variable are specific to an object	Static variable are common for every object that means there memory location can be sharable by every object reference or same class.
<p>Non-static variable can access with object reference. <i>Syntax</i></p>	<p>Static variable can access with class reference. <i>Syntax</i></p>
obj_name.variable_name	class_name::variable_name

Operator Overloading

Redefining operator for user defined data type is called Operator Overloading.

For built-in datatype we don't need to specify how operator should behave but for user define datatype we have to specify how the operator should behave for datatype. This can be done by overloading operator function.

Operators are of two types

- 1> Unary operators: ++,--,!
- 2> Binary operators:- +,-,*,/,%,<=,>= etc..

The following operators cannot be defined by a user:

- . member selection
- .* member selection with pointer-to-member
- ?: conditional
- :: scope resolution
- sizeof object size information
- typeid object type information

Operators that cannot be overloaded by using friend functions

- = assignment operator
- [] subscript operator
- () function call operator
- -> indirect member access operator
- ->* indirect pointer to member access operator

Overloading unary operators

Unary operators can be overloaded by defining special operator function with syntax:-

```
returntype operator op( )
{
    Statements
    .....
    Return value;
}
```

This operator function is automatically called when object is used with defined operator in an expression.

```
#include<iostream.h>
class circle
{
    int r;
public:
    circle(int n)
    {
        r=n;
    }
    void area( )
    {
        float a=3.14*r*;
        cout<<"area is"<<a;
    }
    void operator ++( )
    {
        r++;
    }
};
main()
{
    circle a(5);
    a.area( );           // area with R=5
    ++a;
    a.area( );           // area with R=6
}
```

*** Design a class date and overload ++ and - - operator which will increment or decreament year by one**

```
#include<iostream.h>
class date
{
    int day,month,year;
public:
    date(int a,int m,int y)
    {
        day=d;
        month=m;
        year=y;
    }
    void display( )
    {
        cout<<day<<.. <<month<<.. <<year;
    }
    void operator ++( )
    {
        y++;
    }
    void operator --( )
    {
        y--;
    }
};

main( )
{
    date a(25,10,2007),b(10,5,2007);
    a.display( );
    ++a;
    a.display( );
    b.display( );
    --b;
    b.display( );
}
```

*** Design a class point and overload not operator which will change sign of coordinates.**

```
#include<iostream.h>
class point( )
{
    int x,y;
public:
    point (int x,int y)
    {
        x=x;
        y=y;
    }
    void display( )
    {
        cout<<"x coordinate is"<<x;
        cout<<"y coordinate is"<<y;
    }
    point operator !( )
    {
        point z(-x,-y)
        return z;
    }
};
main( )
{
    point a(5,-7),b;
    a.display( );
    b=! a;
    a.display( );
    b.display( );
}
```

*** Design a class feet and overload ++ operator which will increament its value by 1 inch.**

```

class feet
{
    int feet,inches;
public:
    feet(int m,int n)
    {
        feet=m;
        inch=n;
    }
    void display( )
    {
        cout<<"feet is"<<feet<<"inches are"<<inch;
    }
    void operator ++()
    {
        inch ++;
        if(inch==12)
        {
            feet++;
            inch=0;
        }
    }
    void operator --( )
    {
        inch --;
        if(inch===-1)
        {
            feet--;
            Inch=11;
        }
    }
};

main( )
{
    feet a(4,10);
    a.display( );
    ++a;
    a.display( );
    --a;
    a.display( );
}

```

Overloading Binary Operators

Binary operator can be overloaded by defining special operator function.

Syntax: **returntype operator op(datatype arg)**
 {
 Return value;
 }

This operator function is call whenever an object is used in an expression with defined operator. This operator function is called for the first operand & the second operand comes as argument to the function.

```
class Date
{
    int date ,month , year;
public:
    date(int d=0,int m=0,int y=0)
    {
        date=d;
        month=m;
        year=y;
    }
    void display( )
    {
        cout<<date<<"-"<<month<<"-"<<year<<endl;
    }
    Date operator +(int n)
    {
        date z(date ,month ,year+n);
        return z;
    }
};

main( )
{
    date a(25,10,2007),b;
    a.display( );
    b=a+4;
    b.display( );
}
```

*** Design a class feet and overload '+' and '-' operator which will add n feet or subtract n feet.**

```
#include<iostream.h>
class feet
{
    int feet,inches;
public:
    feet(int m,int n)
    {
        feet=m;
        inches=n;
    }
    void display( )
    {
        cout<<"feets are"<<feet;
        cout<<"inches are"<<inches;
    }
    feet operator +(int n)
    {
        feet z;
        z=(feet+n,inches);
        return z;
    }
};

main( )
{
    feet a(3,7),b(4,9),c,d;
    c=a+5;
    c.display( );
    d=b-2;
    d.display( );
}
```

***Design a class MyString and overload + operator to Concat 2 MyString objects**

```
#define SIZE 100
#include<iostream.h>
class MyString
{
    char Data[SIZE];
public:
    MyString(char *str)
    {
        strcpy(Data,str);
    }
    MyString()
    {
        strcpy(Data,"");
    }
    void display( )
    {
        cout<<Data;
    }
    MyString operator +(MyString c)
    {
        MyString z;
        strcpy(z.Data,Data);
        strcat(z.Data,c.Data);
        return z;
    }
};
main( )
{
    MyString a("Seeta"),b("Ram"),c;
    a.display( );
    b.display( );
    c=a+b;
    c.display( );
}
```

O/P
Seeta
Ram

SeetaRam

Example to overload = Operator

```

class Distance
{
    int feet;           // 0 to infinite
    int inches;         // 0 to 12
public:
    Distance(){
        feet = 0;
        inches = 0;
    }
    Distance(int f, int i){
        feet = f;
        inches = i;
    }
    void operator=(const Distance &D )
    {
        feet = D.feet;
        inches = D.inches;
    }
    void displayDistance()
    {
        cout << "F: " << feet << " I:" << inches << endl;
    }
};

int main()
{
    Distance D1(11, 10), D2(5, 11);
    cout << "First Distance : ";
    D1.displayDistance();
    cout << "Second Distance :";
    D2.displayDistance();

    // use assignment operator
    D1 = D2;
    cout << "First Distance :";
    D1.displayDistance();

    return 0;
}

```

Pitfalls/Limitations of Operator Overloading

- You cannot add new operators in the language.
- You cannot change arguments/operands count of any operator.
- You cannot change precedence and associativity of any operator.
- Certain operators cannot be overloaded as friend function `()`, `->`, `=`, `[]`.
- Certain operators cannot be overloaded ? `:`, `::`, `.*`, `typeid`, `typecasting` operators. Overloading these operators will make c++ programs inconsistent and result in abnormal behaviour.
- Many operators are built in language features and does not make any sense of overloading.
- Operator overloading is made for extending meaning of operator so that it can be used for objects of user defined class; not for changing their meanings.

Advantages of Operator Overloading

- It will act differently depending on the operands provided. Its called extesibility.
- It is not limited to work only with primitive Data Type.
- By using this overloading we can easily acces the objects to perform any operations.
- It makes code much more readable.

Rules of Operator Overloading in C++

1. Only existing operators can be overloaded. New operators cannot be overloaded.
2. The overloaded operator must have at least one operand that is of user defined type.
3. We cannot change the basic meaning of an operator. That is to say, We cannot redefine the plus(+) operator to subtract one value from the other.
4. There are some operators that cannot be overloaded like size of operator(sizeof), membership operator(.), pointer to member operator(*), scope resolution operator(::), conditional operators(?:) etc
5. We cannot use "friend" functions to overload certain operators. However, member function can be used to overload them. Friend Functions can not be used with assignment operator(=), function call operator(()), subscripting operator([]), class member access operator(->) etc.
6. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
7. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
8. Binary arithmetic operators such as +, -, *, and / must explicitly return a value. They must not attempt to change their own arguments.

Operator overloading by using friend function

A friend operator function can be overloaded by using syntax:-

```
return type operator op(datatype arg1,datatype arg2)
{
    -----
    return value;
}
```

The first operand will come as first argument second operand will come as second second argument.

Eg.

```
class complex
{
    float r,i;
public:
    complex(float m=0,float n=0)
    {
        r=m;
        i=n;
    }
    void display( )
    {
        cout<<"r="<<r<<"i="<<i<<endl;
    }
    friend complex operator+(complex,complex);
};

complex operator + (complex m, complex n)
{
    complex z;
    z.r=m.r+n.r;
    z.i=m.i+n.i;
    return z;
}

main()
{
    complex a(0.1725,0.0721),b(0.2113,0.0111),c;
    c=a+b;
    c.display( );
}
```

Functions Returning Objects

The function could return

1. An object
2. A reference to an object

For eg: In this Examples we are returning object of type complex.

```
#include<iostream.h>
class complex
{
    float real , imaginary;
public:
    complex(float r, float i)
    {
        real=r;
        imaginary=i;
    }
    void display( )
    {
        cout<<"real part is"<<real;
        cout<<"imaginary part is"<<imaginary;
    }
    complex operator +(complex c)
    {
        complex z;
        z.real=real + c.real ;
        z.imaginary=imaginary + c.imaginary;
        return z;
    }
};

main( )
{
    complex a(0.1275,0.0712),b(0.2113,0.0111),c;
    a.display( );
    b.display( );
    c=a+b;
    c.display( );
}
```

For eg: In this Examples we are returning reference to cout object

```
#include<iostream.h>
#include<string.h>
class student
{
    int rollno;
    char name[20];
public:
    student(int rn,char n[])
    {
        rollno=rn;
        strcpy(name,n);
    }
friend ostream& operator<<(ostream &out,student &s);
};

ostream& operator << ( ostream &out, student &s)
{
    cout<<"roll no is "<<rollno<<endl;
    cout<<"name is "<<"name<<endl;
}

main()
{
    student a(4117,"Amit Jain");
    cout<<a;    // we can print data of object like built in types
}
```

Example to overload += operator

```
#include<iostream.h>
class complex
{
    float real , imaginary;
public:
    complex(float r, float i)
    {
        real=r;
        imaginary=i;
    }
    void display( )
    {
        cout<<"real part is"<<real;
        cout<<"imaginary part is"<<imaginary;
    }
    Complex& operator +=(complex c)
    {
        this->read+=c.real;
        this->imaginary+=c.imaginary;
        return *this;
    }
};

main( )
{
    complex a(0.1275,0.0712),b(0.2113,0.0111),c;
    a.display( );
    b.display( );
    c=a+b;
    c.display( );
}
```

Overloading subscript operator[]

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class person
{
char fname[10],mname[10],lname[10];

public:
    person(char *fn,char *mn,char *ln)
    {
        strcpy(fname,fn);
        strcpy(mname,mn);
        strcpy(lname,ln);
    }
    char * operator [](char *type)
    {
        if(strcmp(type,"firstname")==0)
            return fname;
        if(strcmp(type,"middlename")==0)
            return fname;
        if(strcmp(type,"lastname")==0)
            return fname;
        return NULL;
    }
};

void main()
{
    clrscr();
    person a("amit","mohan","joshi");

    cout<<a["firstname"];
}
```


Overload – operator to compare 2 strings

```
#include <iostream>
using namespace std;
#include <string.h>
class String
{   char str[100];
public:
    String()
        { strcpy(str, ""); }
    String( char s[] )
        { strcpy(str, s); }
    void display()
        { cout << str; }
    void getstr(){ cin.get(str, 100); }
    bool operator == (String ss)
        {
            return ( strcmp(str, ss.str)==0 ) ? true : false;
        }
};

int main(){
    String s1 = "yes";
    String s2 = "no";
    String s3;

    cout << "\nEnter 'yes' or 'no': ";
    s3.getstr();

    if(s3==s1)
        cout << "You typed yes\n";
    else if(s3==s2)
        cout << "You typed no\n";
    else
        cout << "You didn't follow instructions\n";
    return 0;
}
```

Type casting

Converting one type of value into another is called as Type Casting. It can be achieved implicitly or explicitly.

Implicit type casting

For built in datatype c++ automatically performs type casting.

Eg:-

```
float a=3.2;  
int b;  
b=a;           // here float value is implicitly converted to int  
cout<<b;
```

output=3

Explicit type casting

For built in datatype explicit type casting can be achieved by using cast operator.

Syntax: *dataType(exp)*

Will convert the exp. iIn specified type

eg:-

```
int a=3,b=2;  
float c;  
c=float(a)/b; //here we are explicitly converting int to float  
cout<<c;
```

output=1.5

Type casting for user defined types

For user defined datatype there are different possibilities.

1) Type casting from built into user defined

To perform such type of operation our object must contain a constructor with single argument. This constructor is automatically called when value is assign to an object. The value is passed on as argument to the constructor.

Eg:-

```
#include<iostream.h>
class circle
{
    int r;
public:
    void setradius(int n)
    {r=n;
    }
    void area( )
    {
        float a=3.14*r*r;
        cout<<"area is "<<a;
    }
    circle(int n=1)
    {
        r=n;
    }
};

main( )
{
    int b=5;
    circle a;
    a=b;           // here we are assigning int value to
    a.area( );     // object of type circle
}
```

2) Type casting from user defined to built in

To perform such type of operation our object must contain a special operator function returning a value.

Syntax:-

```
operator datatype( )
{
    .....
    return value;
}
```

This operator function is automatically called when object is assigned to a variable.

Eg:-

```
#include<iostream.h>
class circle
{    int r;
public:
    operator int ( )
    {    return r;
    }
    void main( )
    {    float a=3.14*r*;
        cout<<"area is"<<a;
    }
    circle int (n=1)
    {    r=n;
    }
};
main( )
{
    circle a(5);
    int b;
    a=b;                                // here operator function will be called
    cout<<"radius is"<<b;              // & will return radius .
}
```

*** Design a class feet which will return its value in inches when it is assign to a variable of int type**

```
#include<iostream.h>
class feet
{
    int feet,inches;
public:
    feet(int m,int n)
    {
        feet=m;
        inches=n;
    }
    operator int( )
    {
        int a=feet*12+inches;
        return a;
    }
};
main( )
{
    feet a(2,7);
    int b;
    b=a;
    cout<<"b"<<inches;
}
```

3) Type casting from user defined to user defined

To perform such type of operation then the object which is assigned must contain special operator function and object to which it is assigned must contain a single argument.

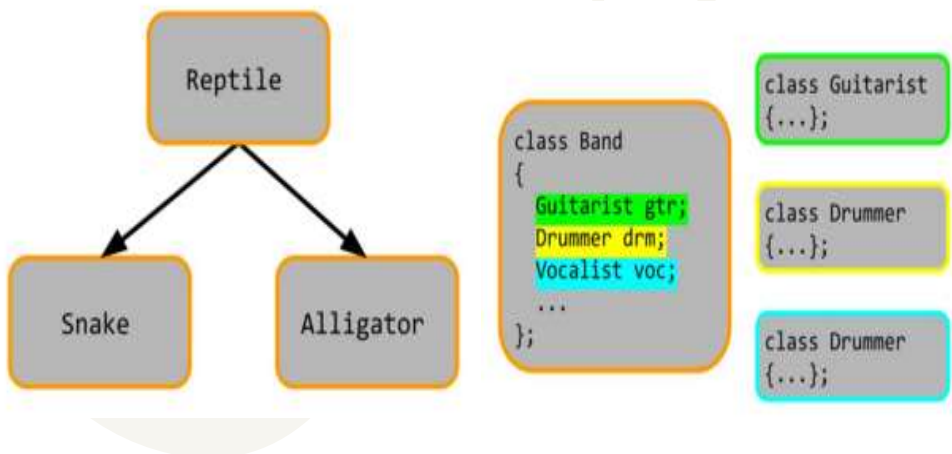
```
#include<iostream.h>
class circle
{   int r;
public: circle(int n=1)
    {   r=n;
    }
    void area( )
    {   float a=3.14*r*r;
        cout<<"area is"<<a;
    }
};
class sphere
{   int r;
public: sphere(int n=1)
    {r=n;
    }
    void volume( )
    {   float v=4/3.0*3.14*r*;
        cout<<"volume is"<<v;
    }
    operator int( )
    {   return r;
    }
};
main( )
{   circle a;
    sphere b(5);
    a=b;
    a.area( );
    b.volume( );
}
```

Composition/ContainerShip

Composition is the process of making one class a data member of another class. For example, you could have a class called Band. The data members of Band could consist of objects from the Guitarist, Drummer, and Vocalist classes. These objects are data members of the Band class, but not descendants or parent classes. They are related by composition, not by inheritance.

“Is A” vs. “Has A” Relationships

Inherited classes are commonly described as having an “is a” relationship with each other while composition involves a “has a” relationship between classes. We can say that Snake *is a* Reptile and that Alligator *is a* Reptile. But we cannot say that Guitarist *is a* Band. Instead, we say that Band *has a* Guitarist.



If we create objects of your existing class inside the new class. This is called *composition* because the new class is composed of objects of existing classes. Accessing the member functions of the embedded object (referred to as a *subobject*) simply requires another member selection. It's probably more common to make the embedded objects **private**, so they become part of the underlying implementation .

```
class Person
{char  Name[20],gender;
public: void setData(char n[],char g)
        {strcpy(name,n);
         gender=g;
        }
        void showData()
        {Cout<<"Name is "<<<name<<endl;
         Cout<<"Gender is "<<<gender<<endl;
        }
};

class Couple
{
    Person m,f;
public: void setData(char mname[],char fname[])
        { m.setData(mname,'m');
          f.setData(fname,'f');
        }
        void showData()
        { m.showData();
          f.showData();
        }
};

main()
    {Couple a;
      a.setData("Amit","Amruta");
      a.showData();
    }
```


Hierarchy

The two types of hierarchies in OOA are:

- **“IS-A” hierarchy** : It defines the hierarchical relationship in inheritance, whereby from a super-class, a number of subclasses may be derived which may again have subclasses and so on. For example, if we derive a class Rose from a class Flower, we can say that a rose “is-a” flower.
- **“Has-A / PART-OF ” hierarchy** : It defines the hierarchical relationship in aggregation by which a class may be composed of other classes. For example, a flower is composed of sepals, petals, stamens, and carpel. It can be said that a petal is a “part-of” flower.

Modularity

Modularity is the process of decomposing a problem (program) into a set of modules so as to reduce the overall complexity of the problem.

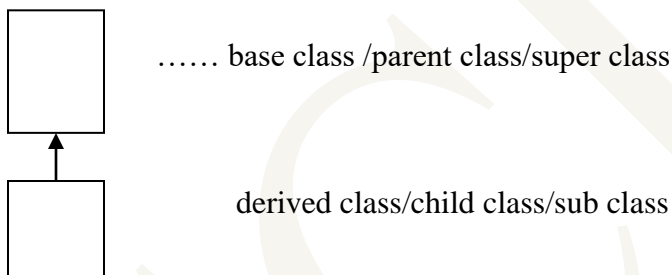
“Modularity is the property of a system that has been decomposed into a set of loosely coupled modules.”

Modularity is intrinsically linked with encapsulation. Modularity can be visualized as a way of mapping encapsulated abstractions into real, physical modules having high cohesion within the modules and their inter-module interaction or coupling is low.

Inheritance

It is a technique of deriving new classes from existing one. The derived class inherits all the features of base class from which it is derived.

When you inherit, you are saying, "This new class is like that old class." You state this in code by giving the name of the class, as usual, but before the opening brace of the class body, you put a colon and the name of the *base class* (or classes, for multiple inheritance). When you do this, you automatically get all the data members and member functions in the base class.



syntax:-

```

class Name : visibilityMode BaseClassName
{
    Data members
    .....
    Member functions
    .....
};
  
```

Visibility mode:- It indicates visibility of base class member in derived class. It can be private, public or protected.

```
class account
{
    protected: int accno,balance;
public:
    void open(int an,int b)
    {
        accno=an;
        balance=b;
    }
    void deposit(int amt)
    {
        balance=balance+amt;
    }
    void withdraw(int qmt)
    {
        balance=balance-amt;
    }
    void showbalance( )
    {
        cout<<"balance is"<<balance
    }
};

class saccount:public account
{
    public:
    void add_interest(float r,float n)
    {
        float i=balance*r*n/100;
        balance=balance+i;
    }
};

main( )
{
    account a;
    a.open(4117,5000);
    a.deposit(7000);
    a.showbalance();
    saccount b;
    b.open(3012,10000);
    b.withdraw(5000);
    b.deposit(1000);
    b.add_interest(10.25,3);
    b.showbalance( );
}
```

*** Design a class 'rect' containing data members length,breadth and member function setdimension and area.they derived a new class 'arect' containing additional member function perimeter**

```
#include<iostream.h>
class rect
{protected: int length,breadth;
public:
    void setdimension(int l,int b)
    {
        length=l;
        breadth=b;
    }
    void area( )
    {
        int a=length*breadth;
        cout<<"area is"<<a;
    }
};
class arect: public rect
{
    public: void perimeter( )
    {
        int p=2*(length+breadth);
        cout<<"perimeter is"<<p;
    }
};
main( )
{
    rect a;
    a.setdiamension(4,7);
    a.area( );
    arect b;
    b.setdiamension(3,2);
    b.area( );
    b.perimeter( );
}
```

Function Overriding

Redefining a base class function derived class is called function overriding. The overridden function must have same name , same no. & types of arguments similar to a base class function.

This overridden function will be called for derived class object instead of the inherited class function.

*** Design a class 'box' containing data member length, breath, height and member function setdimension, volume and surface area then derive a new class openbox from class box.**

Eg:-

```
#include<iostream.h>
class box
{
    protected: int length,breadth,height;
    public: void setdimension(int l,intb,inth)
    {
        length=l;
        breadth=b;
        height=h;
    }
    void volume( )
    {
        int v=length*breadth*height;
        cout<<"volume is"<<v;
    }
    void surfacearea( )
    {
        int a=2*l*h+2*h*b+2*l*b;
        cout<<"surface area is"<<a;
    }
};
```

```

class open box:public box
{
    public:void surfacearea( )
    {
        int s=2*l*h+2*b*h+l*b;
        cout<<"surface area of open box is "<<s;
    }
};

main( )
{
    box a;
    a.setdimension(4,5,7);
    a.volume( );
    a.surfacearea( );

    openbox b;
    b.setdimension(3,4,5);
    b.volume( );
    b.s.area( );
}

```

Advantages of Inheritance

- Development model closer to real life object model with hierarchical relationships
- Reusability -- facility to use public methods of base class without rewriting the same
- Extensibility -- extending the base class logic as per business logic of the derived class
- Data hiding -- base class can decide to keep some data private so that it cannot be altered by the derived class

*** Design a class ‘account’ containing data member accno and balance and member function open,deposit,withdraw and showbalance then derive new class current account from class account having restriction it will charge 1 rs. For each transaction**

```
#include<iostream.h>
class account
{
    protected: int accno,balance;
    public: void open(int an,int b)
    {
        accno=an;
        balance=b;
    }
    void deposit(int amt)
    {
        balance=balance+amt;
    }
    void withdraw(int amt)
    {
        balance=balance-amt;
    }
    void showbalance( )
    {
        cout<<"balance is"<<balance;
    }
};
class caccount:public account
{
    public:
    void deposit(int amt)
    {
        balance=(balance+amt)-1;
    }
    void withdraw(int amt)
    {
        balance=(balance-amt)-1;
        cout<<"balance is"<<balance;
    }
};
```

```

main( )
{
    account a;
    a.open(4117,5000);
    a.deposit(3000);
    a.withdraw(4000);
    a.showbalance( );
    caccount b;
    b.open(3012,7000);
    b.deposit(5000);
    b.withdraw(7000);
    b.withdraw(3000);
    b.showbalance(1);
}

```

Calling base class function in derived class

This can be done by using a special syntax:-

BaseClassName: :functionName(arguments);

Eg:-

```

class caccount:public account
{
    public: void deposit(int amt)
    {
        account::deposit(amt) ;
        balance=balance-1;
    }
    void withdraw(int amt)
    {
        account::withdraw(amt) ;
        balance=balance-1;
    }
};

```


*** Design a class set containing data members x and y and member function setdata,sum,mean then derive additional d,m,z**

Eg:-

```
#include<iostream.h>
class set
{
    protected:int x,y;
    public:
    void setdata(int m,int n)
    {
        x=m;
        y=n;
    }
    int sum( )
    {
        int sum=x+y;
        return sum;
    }
    float mean( )
    {
        float m=(x+y)/2.0
        return m;
    }
};

class xset:public set
{
    int c;
    public:
    void setdata(int m,int n,intp)
    {
        x=m;
        y=n;
        z=p;
    }
}
```

```
int sum()
{
    int s = set::sum()+z;
    return s;
}
float mean()
{
    float m;
    m=sum()/3.0;
    return m;
}
};

main( )
{
    set a;
    a.setdata(4,7);
    cout<<"sum of set a "<<a.sum( );
    cout<<"mean of set a "<<a.mean( );

    xset b;
    b.setdata(3,7,2);
    cout<<"sum of set b"<<b.sum( );
    cout<<"mean is"<<b.mean( );
}
```

Overloading Vs Overriding

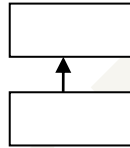
Method Overloading	Method Overriding
In Method Overloading, Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.	In Method Overriding, sub class have the same method with same name and exactly the same number and type of parameters and same return type as a base class.
Method Overloading means more than one method shares the same name in the class but having different signature.	Method Overriding means method of base class is re-defined in the derived class having same signature.
Method Overloading is to “add” or “extend” more to method’s behavior.	Method Overriding is to “Change” existing behavior of method.
It is a compile time polymorphism .	It is a run time polymorphism .
It may or may not need inheritance in Method Overloading.	It always requires inheritance in Method Overriding.
In Method Overloading, methods have same name different signatures but in the same class.	In Method Overriding, methods have same name and same signature but in the different class.
Method Overloading does not require more than one class for overloading.	Method Overriding requires at least two classes for overriding.

Types of inheritance

We can derive our classes in different ways

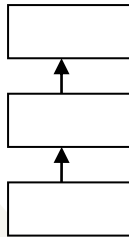
1) Single level inheritance :-

If a class is derived from a single class then such type of inheritance is called as single level inheritance.



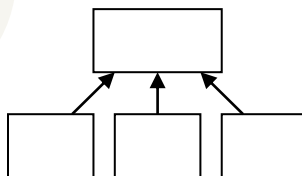
2) Multi level inheritance:-

If a class is derived from a derived class, which is then such type of inheritance is called as multilevel inheritance.



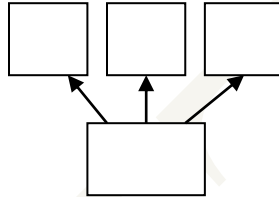
3) Hierarchical inheritance:-

If multiple classes are derived from a single class then such type of inheritance is called as hierarchical inheritance.



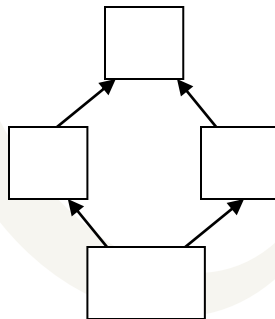
4) Multiple inheritance:-

If a single class is derived from multiple classes then such type of inheritance is called as multiple inheritance.



5) Hybrid Inheritance :-

If mixed type of inheritance is used then it is called as hybrid.



*** Design a class 'person' containing data member name and city and member function getdata and putdata then derive a new class employee from class employee containing additional data member job and salary.**

```
#include<iostream.h>
```

```
class person
{protected:char name[20];
    char city[20];
public:
    void getdata( )
    {cout<<"enter the name and city";
    cin>>name>>city;
    }
    void putdata( )
    {cout<<"name is"<<name;
    cout<<"city is"<<city;
    }
};
class employee:public person
{    char job[20];
    int salary;
public:
    void getdata( )
    {person::getdata( );
    cout<<"enter job and salary";
    cin>>job>>salary;
    }
    void putdata( )
    {person::putdata( );
    person::putdata( )
    cout<<"job is"<<job;
    cout<<"salary is"<<salary;
    }
};
```

Note: In this ex it is not necessary to declare name and city as protected because we are not using them directly in class employee.

*** Design a class person and derive employee from it then again derive class manager from employee.**

// from previous example

Ex of Multilevel Inheritance

```
class manager:public employee
{
    char branch[20];
public:
    void getdata( )
    {
        employee::getdata( );
        cout<<"enter the branch";
        cin>>branch;
    }
    void putdata( )
    {
        employee::putdata( );
        cout<<"branch is"<<branch;
    }
};

void main()
{
    manager m;
    m.getdata();
    m.putdata()
}
```

//when an object of manager will be created space will be reserved for name, city , job, salary & branch

*** Derived two classes student and professor from class person.**

```
#include<iostream.h>           //Ex of Hierarchical Inheritance
class person
{
protected:
    char name[20];
public:
    void getdata( )
    {
        cout<<"enter the name of person";
        cin>>name;
    }
    void putdata( )
    {
        cout<<"name is"<<name;
    }
};

class student:public person
{
    int rollno;
    char branch[30];
public:
    void getdata( )
    {
        person::getdata( )
        cout<<"enter rollno &branch of student;
        cin>>rollno>>branch;
    }
    void putdata( )
    {
        person::putdata( )
        cout<<rollno<<branch;
    }
};
```



```
class professor:public person
{
    char dept[40];
public:
    void getdata( )
    {
        person::getdata( )
        cout<<"enter the dept";
        cin>>dept;
    }
    void putdata( )
    {
        person::putdata()
        cout<<"dept is"<<dept;
    }
};
```

```
void main()
{
    student a,b;
    professor c;

    a.getdata();
    b.getdata();
    c.getdata();

    a.putdata();
    b.putdata();
    c.putdata();

}
```

We have 3 classes i.e. Person, Student, Professor. We can create as many Objects as required. We are creating 2 students & a Professor Object.

*** Derive a class student from class Person and test.**

```
#include <iostream.h>           // Ex. Of multiple inheritance
class person
{
protected:
    char name[20];
    char city[20];
public:
    void getdata()
    {
        cout<<"enter the name and city";
        cin>>name>>city;
    }
    void putdata()
    {
        cout<<"name is"<<name;
        cout<<"city is"<<city;
    }
};

class test
{
protected: int marks
public:
    void getdata()
    {
        cout<<"enter the marks";
        cin>>marks;
    }
    void putdata( )
    {
        cout<<"marks are"<<marks;
    }
};
```

```
class student:public person,public test
{
    int rollno;
public:
    void getdata()
    {
        person: :getdata();
        test: :getdata();
        cout<<"enter rollno";
        cin>>rollno;
    }
    void putdata()
    {
        person: :putdata();
        test: :putdata();
        cout<<"roll no is"<<rollno;
    }
};

void main()
{
    student a,b;

    a.getdata();
    b.getdata();

    a.putdata();
    b.putdata();
}
```

Constructor and inheritance

Whenever an object of derived class is created both constructor are executed. First base class constructor is executed and then derived class is executed (if present).

```
class base
{public:
    base()
    {
        cout<<"bcc";
    }
};
class derived:public base
{
public:
    derived()
    {
        cout<<"dcc";
    }
};
derived b;           // o/p will be bccdcc
```

Note: Actually when derived class object is created derived class constructor is called first and then base class constructor is called.

Calling base class constructor from derived class.

This can be done by using special syntax while defining derived class constructor.

```
Derived(arguments.....) : BaseClassName(arg values...)
{
    .....
}
```

```
#include<iostream.h>
class person
{
    char name[20];
public:
    person(char n[])
    {
        strcpy(name,n);
    }
    void showinfo()
    {
        cout<<"name is"<<name;
    }
};
class student:public person
{
    int roll no;
public:
    student(int rn,charm[]):person(m)
    {
roll no=rn;
    }
    void showinfo()
    {
        cout<<"roll no is"<<roll no<<endl;
        person: :showinfo();
    }
};
main()
{
    student b(4117,"gopal pandey");
    b.showinfo();
}
```

Constructor and Destructor invoking sequence with inheritance

During inheritance, base class may also contain constructor and destructor.

In this case if you create an instance for the derived class then base class constructor will also be invoked and when derived instance is destroyed then base destructor will also be invoked and the order of execution of constructors will be in the same order as their derivation and order of execution of destructors will be in reverse order of their derivation.

```
#include<iostream.h>
class Base
{
public:
Base() //Constructor
{
    cout<<"Base Class Constructor called"<<endl;
}
~Base() //Destructor
{
    cout<<"Base Class Destructor called"<<endl;
}
};
```

```
class Derived:public Base
{
public:
Derived() //Constructor
    {
        cout<<"Derived Class Constructor called"<<endl;
    }
~Derived() //Destructor
    {
        cout<<"Derived Class Destructor called"<<endl;
    }
};

void main()
{
Derived d;
}
```

Output:

Base Class Constructor called
Derived Class Constructor called
Derived Class Destructor called
Base Class Destructor called

Visibility mode

It indicates member in derived class. It can be private public or protected.

private Inheritance

If a base class is privately inherited in derived class then all the public and protected members of base class becomes private in derived class.

```
class base
{
private : a
protected:b
public:   c
};
```

```
class derived:private base
{
private:  l
protected:m
public:   n
};
```

//in this class we can access base class members b & c. But they are inherited as private
Even public members of base class are not accessable with object

```
class sderived:private derived
{
private:  x
protected:y
public:   z
};
```

//in this class we can access base class members m & n. But they are inherited as private.
Even the public & protected members of class base are not accessible.

Protected Inheritance

If a base class is inherited in a derived class as protected then all the public and protected members of base class are inherited as protected in derived class.

```
class base
{
private : a
protected:b
public:    c
};
```

```
class derived:protected base
{
private:  l
protected:m
public:   n
};
```

//in this class we can access base class members b & c.
But they are inherited as protected.
Even the public member c of base class cannot be accessed with derive class

```
class sderived:protected derived
{
private:  x
protected:y
public:   z
};
```

//in this class we can access base class members b, c, m & n.
But they are inherited as protected.
Even the public & protected members of

Protected Inheritance

If a base class is publicly inherited in derived class then all public & protected member of base class remains as it is in derived class i.e. public & protected.

```
class base
{
private : a
protected:b
public:   c
};
```

```
class derived:public base
{
private:  l
protected:m
public:   n
};
```

//in this class we can access base class members b & c. But they are inherited as protected & public respectively. The public member c of base class can also be accessed with derive class object

```
class sderived:public derived
{
private:  x
protected:y
public:   z
};
```

//in this class we can access base class members b, c, m & n. But they are inherited as protected & public respectively. The public member c & n of base classes can also be accessed with derive class object.

Virtual Base Class

- An ambiguity can arise when several paths exist to a class from the same base class.
- This means that a child class could have duplicate sets of members inherited from a single base class.
- C++ solves this issue by introducing a virtual base class.
- When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.
- When two or more classes are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited.
- Such a base class is known as virtual base class. This can be achieved by preceding the base class' name with the word virtual.

Consider following example:

```
class A
{
    public:
        int i;
};
```

```
class B : virtual public A
{
    public:
        int j;
};
class C: virtual public A
{
    public:
        int k;
};
class D: public B, public C
{
    public:
        int sum;
};
int main()
{
    D ob;
    ob.i = 10; //unambiguous since only one copy of i is
inherited.
    ob.j = 20;
    ob.k = 30;
    ob.sum = ob.i + ob.j + ob.k;
    cout << "Value of i is : "<< ob.i<<"\n";
    cout << "Value of j is : "<< ob.j<<"\n"; cout << "Value of k is
:"<< ob.k<<"\n";
    cout << "Sum is : "<< ob.sum <<"\n";

    return 0;
}.
```

Pointers

A pointer is a variable in which we can store addresses of another variable.

1>Defining pointer

Datatype *pointername

Eg int *p;
 char *q;
 Student *s;

Note:- Size of Different types of pointer is same.

2>Storing Addresses :

Address of a variable or an object
can be find out by using operator '&'

Syntax:-

Ptr=&variablename;
Ptr=&objectname

3>Accessing value by using pointer

If a pointer is pointing to a variable then value of that variable can be accessed by using

syntax:- ****ptrname;***

Ex:-

```
int a=5;
int *p;
p=&a;
cout<<a;
cout<<*p;
```

4>Object Pointers

Object pointer is a pointer in which we can store address of an object then member of that object can be accessed by using
syntax:-

Ptrname->membername

Pointers Arithmetic

With pointer variable we can use different arithmetic operators such as ++,--,+,-..etc. But they behave different with pointers. Result of expression depends on data type of pointer (size of data type)

Ex:-1>

```
int a=5,*p;
p=&a;      // store address of a into p i.e. p is pointing to a
```

```
a++;  // will increment value of a by 1
```

```
p++;  // will increment value of p by 2 as p is a pointer to int
```

```
(*p)++;  // will increment value of a by 1 as pointer p is pointing to a
```

Example of object pointer

```
#include<iostream.h>
class circle
{
    int r;
public:
    void setradius(int n)
    {
        r=n;
    }
    void area()
    {
        float a=3.14*r*;
        cout<<"area is"<<a;
    }
};
main()
{
    circle a;
    circle*p;
    p=&a;
    p->setradius(7);
    p->area();
}
```

Pointer to pointer

A pointer to pointer is a pointer in which we can store address of another pointer

It can be defined by using syntax : **Datatype **ptrname;**

```
main()
{
int a=5;
int *p;
int **q;
p=&a;
q=&p;
cout<<a<<endl;
cout<<*p<<endl;
cout<<**q<<endl;
}
```

Array of pointer

If we want to store addresses of multiple memory location then we can use array of pointer .

It can be defined by using the syntax :

Datatype *ptrname[size];

```
main()
{
    int s=0,i;
int a=5,b=27,c=311,d=36,e=52;
int *p[5];
p[0]=&a;
p[1]=&b;
p[2]=&c;
p[3]=&d;
p[4]=&e;
for(i=0;i<=4;i++)
    s=s+*p[i];
printf("sum is %d",s);
}
```

Passing arrays as arguments

Whenever a function is called by passing an array actually base address of that array is passed as arguments to the function

```
int sum(*p,int n)
{
    int s=0,i;
    for(i=0;i<=n;i++)
        { s=s+*p;
          p++;
        }
    return s;
}
main()
{
    int a[10]={7,11,9,13,12,8,16,32,34,31};
    int z;
    z=sum(a,10);
    cout<<"sum is "<<z<<endl;
}
```

Deign a function mean which will return mean of values from an integer array.

<pre>float mean(int *p,int n) { float m; int s=0,i; for(i=0;i<=n;i++) { s=s+*p; p++; } m=(float)s/n; return m; } void main() { int a[10]={7,11,9,18,13,14,16,19,32,52}; float z=mean(a,10); cout<<"mean is "<<z; }</pre>	OR	<pre>float mean(int b[],int n) { float m; int s=0,i; for(i=0;i<n;i++) s=s+b[i]; m=(float)s/n; return m; }</pre>
---	----	--

Design a function length which will return length of a string

```
int length(char *p)
{
    int i=0;
    while(*p!='\0')
    {
        i++;
        p++;
    }
    return i;
}

main()
{
    char a[20];
    cout<<"enter a string";
    cin>>a;
    int i=length(a);
    cout<<"length is "<<i;
}
```

Design a function join which will concat 2 strings

```
void join(char *p,char *q)
{
    while(*p!='\0')
        p++;

    while(*q!='\0')
    {
        *p=*q;
        p++;
        q++;
    }
    *p='\0';
}

main()
{
    char a[20]="seeta";
    char b[20]="ram"
    join(a,b);
    cout<<"result is "<<a;           //result will be seetaram
}
```

Function Pointers

A function pointer is a pointer in which we can store address of another function. Function pointers are generally used to pass function as arguments to functions. A function pointer can be defined by using syntax.

Return type(*ptr name)(datatype of arguments...);

Function addresses

Address of a function can be find out just by using the function name

Syntax: **Function pointer=function name**

Calling functions by using function pointer

If a pointer is pointing to a function then such function can be calling by using the syntax:

(* pointername)(argument value...);

```
void star(int n)
{ int i;
  for(i=1;i<=n;i++)
  printf("*");
}

void main()
{ void (*p)(int);           // define a function pointer
  p=star;                   // store address of function in it
  (*p)(17);                 // call function by using fnction ptr.
}
```

Dynamic Memory Management

Memory in your C++ program is divided into two parts:

- **The stack:** All variables declared inside the function will take up memory from the stack.
- **The heap:** This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use **delete** operator, which de-allocates memory previously allocated by new operator.

C++ provides two new operator '**new**' and '**delete**' to perform dynamic memory management.

1> new

This operator reserves a block of memory at runtime. It reserves block of memory required for specified datatype and returns starting address.

Syntax:- *pointer=new datatype;*

- [illegible]

2> delete

This operator is used to release memory which was dynamically reserve by operator new.

Syntax:- ***delete pointer;***

It releases memory block pointed by specified pointer.

Eg `delete p;` //will release the memory block pointed by the pointer.

Program to Dynamically read n nos.

```
void main()
{
int n;

//Ask how many nos
cout<<"Enter How many nos you want to store ";
cin>>n;

//dynmacally reserve n int blocks
int *p=new int[n];

//read nos
cout<<"Enter "<<n<<" values ";
for(int i=0;i<n;i++)
    cin>>*p[i];

//calculate sum
Int s=0;
for(i=0;i<n;i++)
    s=s+*p[i];

//release reserved memory
delete p;

//display sum
cout<<"Sum is "<<s<<endl;
}
```

Pointers to Objects

- An object pointer is a pointer in which we can store address of an object.
- If a pointer contains address of an object in it then it means that the pointer is pointing to that object.
- If a pointer is pointing to an Object then by using such pointer we can access member of object by using member indirection operator (->)

Syntax : **PointerName -> MemberName**

```
#include<string.h> // Eg. Of dynamic memory management
class person
{
    char name[20];
public:
    person(char n[])
    {
        strcpy(name,n);
    }
    void showinfo()
    {
        cout<<"name is"<<name;
    }
};

main()
{
    person *p;
    p=new person("Amit"); // here object is dynamically created
    p->showinfo();
    delete p;
}
```

```
class person // Eg. Of dynamic memory management
{
    char *name;
public:
    person(char n[])
    {
        int z=strlen(n);
        name=new char[z+1]; // here space to store name is dynamically
        strcpy(name,n); // created
    }
    void showinfo()
    {
        cout<<"name is"<<name;
    }
};

main()
{
    person *p;
    p=new person("chandon");
    p->showinfo();
    delete p;
}
```

Destructors

These are special member function of class which are automatically invoked whenever an object of its type is deleted.

1>they have same name as class name preceded with symbol '~' (tilde')

2>they doesn't take any argument neither does they return any value.

3>they are generally used to release resources used by object.

```
class person
{
    char *name;
public:
    person(char n[])
    {
        int z=strlen(n);
        name=new char[z+1];
        strcpy(name,n);
    }
    void showinfo()
    {
        cout<<"name is"<<name;
    }
    ~person()
    {
        delete name;
    }
};
```

If destructor is not defined it will delete the object but not the memory space reserved by the object for name

```
main()
{
    person *p;
    p=new person("chandon");
    p->showinfo();
    delete p;           // when object is deleted destructor will be
                        // automatically invoked
}
```


this pointer

- Every object has a special pointer "this" which points to the object itself.
- This pointer is accessible to all members of the class but not to any static members of the class.
- Can be used to find the address of the object in which the function is a member.
- Friend functions do not have a **this** pointer, because friends are not members of a class.

```
class student
{
    int rollno;
    char name[20];
public:
    student(int rollno, char name[])
    {
        this->rollno=rollno;
        strcpy(this->name, name);
    }
    void showInfo()
    {
        cout<<"Roll No is "<<rollno<<endl;
        cout<<"Name is "<<name<<endl;
    }
    student* getAddress() // function returning Address of object
    {
        return this;
    }
    student getObject() // function returning object
    {
        return *this;
    }
};

void main()
{
    student a(4117, "Amit");
    student *p=a.getAddress();
    student b=a.getObject();
}
```

Polymorphism

Polymorphism is the ability of different classes to expose similar (or identical) interfaces to the outside. This similarity simplifies your job as a programmer because you don't have to remember hundreds of different names and syntax formats. We create generic procedures that act on all the different Objects and therefore noticeably reduce the amount of code you have to write.

It means that a single statement can have different meaning at different time. We have already seen different types of polymorphism such as function overloading, function overriding, operator overloading etc(compile time polymorphism).

Polymorphism can also be achieved through dynamic linking / late binding. To achieve polymorphism through late binding c++ provides us a feature **"a base class pointer can point to object of its derived class"** *but through that pointer we can access only those member which are declared in base class .*

Static Linking(Early Binding);

In early binding which function to call is decided at compile time depending on type of pointer and not on type of object (with respect to object pointers).

Note: C++ by default performs early binding.

Dynamic Linking (Late binding)

In late binding which function to call is decided at run time depending on type of object and not on the type of pointer (with respect to object pointers).

Virtual functions


In object-oriented programming, when a derived class inherits from a base class, an object of the derived class may be referred to via a pointer or reference of the base class type instead of the derived class type. If there are base class methods overridden by the derived class, the method actually called by such a reference or pointer can be bound either 'early' (by the compiler), according to the declared type of the pointer or reference, or 'late' (i.e. by the runtime system of the language), according to the actual type of the object referred to.

Virtual functions are resolved 'late'. If the function in question is 'virtual' in the base class, the most-derived class's implementation of the function is called according to the actual type of the object referred to, regardless of the declared type of the pointer or reference. If it is not 'virtual', the method is resolved 'early' and the function called is selected according to the declared type of the pointer or reference.

Virtual functions allow a program to call methods that don't necessarily even exist at the moment the code is compiled.

In C++, *virtual methods* are declared by prepending the **virtual** keyword to the function's declaration in the base class.

This modifier is inherited by all implementations of that method in derived classes, meaning that they can continue to over-ride each other and be late-bound.



A virtual function is a member function that is declared as virtual within a base class and redefined by a derived class. To create virtual function, precede the function's declaration in the base class with the keyword `virtual`. When a class containing virtual function is inherited, the derived class redefines the virtual function to suit its own needs.

If a base class function is declared as virtual then c++ performs late binding for that function. So which function to be call will be decided at run time instead of compile time depending on type of object.

```
#include<iostream.h>
class shape()
{public:
    virtual void area()
    {
        cout<<"dummy";
    }
};
```

```
class circle:public shape
{    int r;
public:
    circle(int n)
    {    r=n;
    }
    void area()
    {float a=3.14*r*r;
    cout<<"area is"<<a;
    }
};
```

```
class rect:public shape
{int l,b;
public:
    rect(int m,int n)
    { l=m;b=n;
    }
    void ares()
    { float a=l*b;
    cout<<"area is"<<a;
    }
};
```

```
main()
{
    shape *p,*q;    // we have defined pointer of type shape
    p=new circle(5);
    p->area();       // will call function area of object circle
    q=new rect(4,7);
    q->area();       // will call function area of object rect
}
```

Note: if base class function is not declared as virtual then o/p of both statements will be “dummy”

Virtual Table:

A virtual table is a mechanism to perform dynamic polymorphism i.e., run time binding. Virtual table is used to resolve the function calls at runtime. Every class that uses virtual functions is provided with its own virtual functions.

Every entry in the virtual table is a pointer that points to the derived function that is accessible by that class. A hidden pointer is added by a compiler to the base class which in turn calls `*_vptr` which is automatically set when an instance of the class is created and it points to the virtual table for that class..

Pure virtual function

If a base class virtual function doesn't have a function body then such a virtual function declared as pure virtual function by using syntax:-

Virtual returntype functionname(datatype of args.....)=0;

```
class shape
{
public:
    virtual void area()=0;
};
```

If a class contains pure virtual function then such class becomes an abstract class. We cannot create object of an abstract class. But if required we can define pointer of its type

Abstract classes are define to achieve polymorphism through late binding. Multiple classes can be derived from an abstract class. If a class is derived from an abstract class then it has to compulsorily override all the pure virtual function declared in base class.

Ex of array of pointers (from previous example)

```
#include<iostream.h>
//all classes are defined i.e. shape, circle, rect
main()
{
    shape *p[5];
    p[0]=new circle(7);
    p[1]=new rect(4,3);
    p[2]=new rect(3,7);
    p[3]=new circle(5);
    p[4]=new rect(10,10);
    for(int i=0;i<=4;i++)
        p[i]→area();
}
```

Rules for Virtual functions

- Virtual functions cannot be static and must be member function of class.
- A Constructor cannot be virtual.
- A Distrator can be virtual.
- If base class contains a virtual function and it is not redefined in derived class then base class function is invoked.
- Operator functions can also be virtual.
- A virtual function can be friend of another class.

Characteristics of Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Type of polymorphism

- Compile time polymorphism
- Run time polymorphism

Compile time Vs Run time Polymorphism

Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the compiler .	In Run time Polymorphism, call is not resolved by the compiler.
It is also known as Static binding, Early binding and overloading as well.	It is also known as Dynamic binding, Late binding and overriding as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	Overriding is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by function overloading and operator overloading .	It is achieved by virtual functions and pointers .
It provides fast execution because known early at compile time.	It provides slow execution as compare to early binding because it is known at runtime.
Compile time polymorphism is less flexible as all things execute at compile time.	Run time polymorphism is more flexible as all things execute at run time.

Virtual destructors

Destructors in the Base class can be Virtual. Whenever Upcasting is done, Destructors of the Base class must be made virtual for proper destruction of the object when the program exits.

NOTE : Constructors are never Virtual, only Destructors can be Virtual.

- If the destructor in the base class is not made virtual, then an object that might have been declared of type base class and instance of child class would simply call the base class destructor without calling the derived class destructor.
- Hence, by making the destructor in the base class virtual, we ensure that the derived class destructor gets called before the base class destructor.



Upcasting without Virtual Destructor

Lets first see what happens when we do not have a virtual Base class destructor.

```
class Base
{
public:
~Base() {cout << "Base Destructor\t"; }
};

class Derived:public Base
{
public:
~Derived() { cout<< "Derived Destructor"; }
};

int main()
{
Base* b = new Derived;    //Upcasting
delete b;
}
```

Output : Base Destructor

In the above example, delete b will only call the Base class destructor, which is undesirable because, then the object of Derived class remains undestructed, because its destructor is never called. Which results in memory leak.

Upcasting with Virtual Destructor

Now lets see. what happens when we have Virtual destructor in the base class.

```
class Base
{
public:
    virtual ~Base()
        {cout << "Base Destructor\t"; }
};

class Derived:public Base
{
public:
    ~Derived()
        { cout<< "Derived Destructor"; }
};

int main()
{
    Base* b = new Derived;    //Upcasting
    delete b;
}
```

Output :
Derived Destructor
Base Destructor

When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behaviour.

Pure Virtual Destructors

- Pure Virtual Destructors are legal in C++. Also, pure virtual Destructors must be defined, which is against the pure virtual behaviour.
- The only difference between Virtual and Pure Virtual Destructor is, that pure virtual destructor will make its Base class Abstract, hence you cannot create object of that class.
- There is no requirement of implementing pure virtual destructors in the derived classes.

```
class Base
{
public:
    virtual ~Base() = 0;    //Pure Virtual Destructor
};

Base::~~Base() { cout << "Base Destructor"; } //Definition of Pure
Virtual Destructor

class Derived:public Base
{
public:
    ~Derived() { cout<< "Derived Destructor"; }
};
```

Call to constructor and destructor explicitly

Constructor is a special member function that is automatically called by compiler when object is created and destructor is also special member function that is also implicitly called by compiler when object goes out of scope. They are also called when dynamically allocated object is allocated and destroyed, new operator allocates storage and calls constructor, delete operator calls destructor and free the memory allocated by new.

Is it possible to call constructor and destructor explicitly?

Yes, it is possible to call special member functions explicitly by programmer. Following program calls constructor and destructor explicitly.

```
class Test
{
public:
    Test() { cout << "Constructor is executed\n"; }
    ~Test() { cout << "Destructor is executed\n"; }
};

int main()
{
    Test(); // Explicit call to constructor
    Test t; // local object
    t.~Test(); // Explicit call to destructor
}
```

When the constructor is called explicitly the compiler creates a nameless temporary object and it is immediately destroyed. That's why 2nd line in the output is call to destructor.

const keyword

In C++ const keyword can be used in different ways.

Const class Object

When an object is declared or created with const, its data members can never be changed, during object's lifetime.

```
const class_name object;
```

for eg: `const Student a(4117,"Amit Jain");`

Const Function Arguments

We can make the return type or arguments of a function as const. Then we cannot change any of them.

```
void f(const int i)
{
    i++;    // Error
}
```

Const member functions

A function becomes const when const keyword is used in function's declaration. The idea of const functions is not allow them to modify the object on which they are called. It is recommended practice to make as many functions const as possible so that accidental changes to objects are avoided.

Following is a simple example of const function.

```
class Test
{
    int value;
public:
    Test(int v = 0)
    {
        value = v;
    }

    // We get compiler error if we add a line like "value = 100;" in this function.
    int getValue() const
    {
        return value;
    }
};

int main() {
    Test t(20);
    cout<<t.getValue();
    return 0;
}
```

Const class Data members

These are data variables in class which are made const. They are not initialized during declaration. Their initialization occur in the constructor.

```
class Test
{
    const int i;
public:
    Test (int x)
    {
        i=x;
    }
};

int main()
{
    Test t(10);
    Test s(20);
}
```

In this program, **i** is a const data member, in every object its independent copy is present, hence it is initialized with each object using constructor. Once initialized, it cannot be changed.

Pointers with Const

Pointers can be made **const** too. When we use const with pointers, we can do it in two ways, either we can apply const to what the pointer is pointing to, or we can make the pointer itself a const.

1]Pointer to Const

This means that the pointer is pointing to a const variable.

```
const int* u;
```

Here, u is a pointer that points to a const int. We can also write it like,

```
int const* v;
```

still it has the same meaning. In this case also, v is a pointer to an int which is const.

2] Const pointer

To make the pointer const, we have to put the const keyword to the right of the *.

```
int x = 1;
```

```
int* const w = &x;
```

Here, w is a pointer, which is const, that points to an int. Now we can't change the pointer but can change the value that it points to.

NOTE : We can also have a const pointer pointing to a const variable.

```
const int* const x;
```

Scope resolution operator ::

In C++, scope resolution operator is ::. It is used for following purposes.

1) To access a global variable when there is a local variable with same name:

```
// C++ program to show that we can access a global variable
// using scope resolution operator :: when there is a local
// variable with same name
```

```
int x; // Global x
int main()
{ int x = 10; // Local x
  cout << "Value of global x is " << ::x;
  cout << "\nValue of local x is " << x;
  return 0;
}
```

2) To define a function outside a class.

```
// C++ program to show that scope resolution operator :: is used
// to define a function outside a class
```

```
class A
{public:
  void fun();// Only declaration
};

void A::fun()           // Definition outside class using ::
{  cout << "fun() called";
}

int main()
{  A a;
  a.fun();
  return 0;
}
```

3) To access a class's static variables.

```
// C++ program to show that :: can be used to access static
// members when there is a local variable with same name
```

```
class Test
{
    static int x;
public:
    static int y;

    // Local parameter 'a' hides class member
    // 'a', but we can access it using ::
    void func(int x)
    {
        // We can access class's static variable
        // even if there is a local variable
        cout << "Value of static x is " << Test::x;

        cout << "\nValue of local x is " << x;
    }
};

// In C++, static members must be explicitly defined
// like this
int Test::x = 1;
int Test::y = 2;

int main()
{
    Test obj;
    int x = 3 ;
    obj.func(x);
    cout << "\nTest::y = " << Test::y;
    return 0;
}
```

4)In case of multiple Inheritance

If same variable name exists in two ancestor classes, we can use scope resolution operator to distinguish.

// Use of scope resolution operator in multiple inheritance.

```
class A
{
protected:
    int x;
public:
    A() { x = 10; }
};

class B
{
protected:
    int x;
public:
    B() { x = 20; }
};

class C: public A, public B
{
public:
    void fun()
    {
        cout << "A's x is " << A::x;
        cout << "\nB's x is " << B::x;
    }
};

int main()
{
    C c;
    c.fun();
    return 0;
}
```

Console I/O

C++ provides us different classes for handling input and output on Stream. A Stream is a flow of data from a device to program or from program to a device. The device may be a file, i/o devices (keyboard/printer etc...) or a remote system etc.

class ios

This class provides us basic features to perform I/O operations on stream. These classes are define in the header file iostream.h.

class istream

This class is derived from class ios. It provides basic features required to read data into an input stream .

It contains functions

- a) **int get()** - Will read a character from the input stream

eg. char ch;
 cout<<"enter a character";
 ch=cin.get();
 cout.put(ch);

- b) **getline(char * buffer,int buffresize)** -It will read a line with maximum given size and stored it at the specified buffer.

Eg. char a[10];
 cout<<"enter string";
 cin.getline(a,10);

- c) **read(char * buffer,int size)** – It will read a block of data & store it into buffer.

It also contains overloaded extraction operator ">>" which is used to extract data from an output stream.

Note: The object cin is of this class.

class ostream

This class is derived from class ios. It provides basic features required to write data into an output stream.

It contains functions

- a) **put(int ch)** – to write a char into output stream

eg. cout.put('Z');
 cout.put(65)

- b) **write(char * buffer, int size)** -to write block of data into output stream.

eg. char a[10]="amravati";
 cout.write(a,5); output=amrav

It also contains overloaded insertion operator "<<" which is used to insert data into an output stream.

Note: c++ provides us a readymade object cout of this class which is directly attached to our standard output device

Formatting functions of class ostream

- c) **width(int n)** – it sets the width of data to be displayed

eg. cout<<"Rs";
 cout.width(10);
 cout<<4117; o/p=> Rs 4117

- d) **fill(char ch)** – it fills the blank spaces left by width

eg. cout<<"Rs";
 cout.fill('\$');
 cout.width(10);
 cout<<4117; o/p=> Rs\$\$\$\$\$4117

- c) **precision(int n)** - it sets the precision of floating pt. value.

eg. float a=3.14087
 cout<<a; o/p=> 3.140870
 cout.presion(2);
 cout<<a; o/p=> 3.14

c) **setf(int flag)** – it sets the formatting flag according to which data will be displayed

<code>ios::hex</code>	Output numbers hexadecimal format.
<code>ios::oct</code>	Output numbers in base 8, octal format.
<code>ios::dec</code>	Output numbers in base 10, decimal format
<code>ios::showpoint</code>	Show a decimal point for all floating point numbers whether or not it's needed.
<code>ios::showpos</code>	Put a plus sign before all positive numbers.
<code>ios::scientific</code>	Convert all floating point numbers to scientific notation
<code>ios::fixed</code>	Convert all floating point numbers to fixed point on output
<code>ios::showbase</code>	Print out a base indicator at the beginning of each number For example: hexadecimal numbers are preceded with "0x"
<code>ios::skipws</code>	Skip leading white-space characters on input.
<code>ios::right</code>	Output is right justified.
<code>ios::left</code>	Output is left justified.
<code>ios::internal</code>	Numeric output is padded by inserting a fill character between the sign or base character and the number itself
<code>ios::uppercase</code>	When converting hexadecimal numbers show the digits A-F as uppercase.

Eg.

```
int a=15;
cout.setf(ios::hex);
cout<<a;                                o/p=>      f
cout.setf(ios::oct);
cout<<a;                                o/p=>      17
cout.setf(ios::dec);
cout<<a;                                o/p=>      15
```

I/O manipulator

An I/O manipulator is a special function that can be used in an I/O statement to change the formatting. You can think of a manipulator as a magic bullet that when sent through an input or output file changes the state of the file. A manipulator doesn't cause any output; it just changes the state.

The I/O manipulators are defined in the include file `<iomanip.h>`.

Hex	Output numbers in hexadecimal format.
Oct	Output numbers in octal format.
Dec	Output numbers in decimal format
setw(int width)	Set the width of the output
setprecision(int precision)	Set the precision of floating point output
setfill (char ch)	Set the fill character
endl	Output end-of-line
setbase(int base)	Set conversion base to 8, 10, or 16
setiosflags(long flags)	Set selected conversion flags.
resetiosflags(long flags)	Reset selected flags.

eg.

```
int number = 15;
```

```
cout << "Number is " << hex << setw(10) << number;
```

o/p => number is _____f

user defined manipulators

C++ provides many predefined manipulators but you can also create your own manipulators. The syntax for creating user defined manipulators is defined as follows.

```
ostream&manipulator_name(ostream&ostrObj)
{
    set of statements;
    return ostrObj;
}
```

For example

```
#include < iostream.h>
#include < iomanip.h>

ostream&curr(ostream&ostrObj)
{
    cout << fixed << setprecision(2);
    cout << "Rs.";
    return ostrObj;
}

void main()
{
    float amt = 10.5478;
    cout << curr << amt;
}
```

OverLoading << operator by using friend function

A << operator can be overloaded for class ostream by using syntax

```
ostream& operator <<(ostream &out, ClassType obj)
{
    .....
    return out;
}
```

Eg.

```
#include<iostream.h>
#include<string.h>
class student
{
    int rollno;
    char name[20];
public:
    student(int rn,char n[])
    {
        rollno=rn;
        strcpy(name,n);
    }
friend ostream& operator<<(ostream &out,student &s);
};

ostream& operator << ( ostream &out, student &s)
{out<<"roll no is "<<s.rollno<<endl;
out<<"name is "<<"s.name<<endl;
return out;
}

main()
{
    student a(4117,"Amit Jain");
    cout<<a;    // we can print data of object like built in types
}
```

OverLoading >> operator by using friend function

A >> operator can be overloaded for class ostream by using syntax

```
istream& operator >>(istream &in, ClassType obj)
{
    .....
    return in;
}
```

```
#include<iostream.h>
#include<string.h>
class student
{
    int rollno;
    char name[20];
public:
    void showinfo()
    {cout<<"RollNo is "<<rollno<<endl;
      cout<<"Name is "<<name<<endl;
    }
    friend istream& operator>>(istream &in,student &s);
};

istream& operator >> ( istream &in, student &s)
{in>>s.rollno;
 in>>s.name;
 return in;
}

main()
{ student a;
  cout<<"Enter RollNo and Name";
  cin>>a;          // we can read data similarly as built in type
  a.showinfo();
}
```

File(Disk) I/O

C++ provides us many classes for handling input output operations on files. These classes are defined in header file `fstream.h`

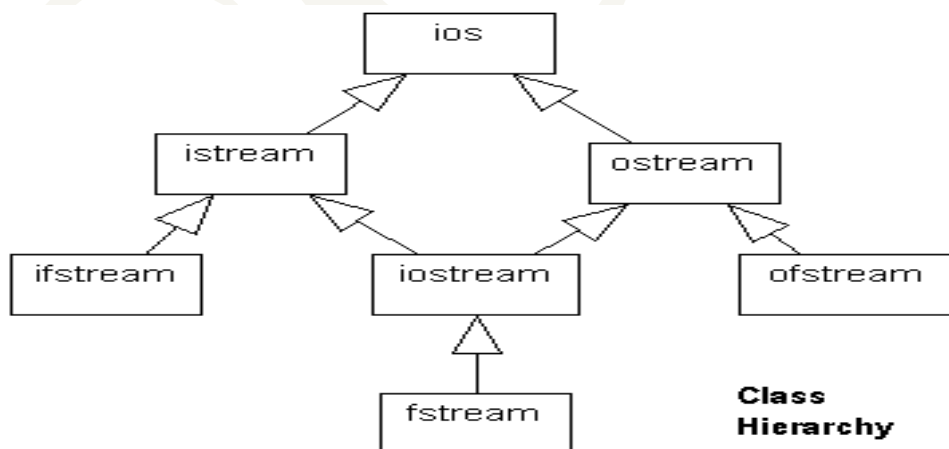
1) class `fstreambase`:- This class is derived from class `ios`. It contains the basic features needed for handling input and output operations on file.

2) class `ifstream`:- This class derived from class `fstreambase` and `istream`. It contains the basic features needed for read data from the file. It contains function `get`, `getline` and read an extraction operator. It also contains a function `open` for opening file with default input mode.

3) class `ofstream`:- This class is derived from class `fstreambase` and `ostream`. It contains basic features needed for write data from the file. It contains function `put` and `write` and insertion operator. It also contains function `open` for opening file in default output mode.

4) class `fstream`:-

This class is derived from class `ifstream` and `ofstream`. It contains the function `open` for opening file. the `open` function contains two arguments (name of the file and mode)



Function **eof()**:- This function is read to check end of file

int eof()

This function returns true(1) at eof.

*** Write a programme to display a file**

```
#include<iostream.h>
#include<fstream.h>
void main( )
{
    ifstream a;
    char ch;
    a.open("star.cpp");
    while(!a.eof( ))
    {
        ch=a.get( );
        cout.put(ch)
    }
    a.close();
}
```

*** W.a.p to read a file copy the contains of the file into another file**

```
#include<iostream.h>
#include<fstream.h>
main( )
{
    ifstream a;
    ofstream b;
    a.open("star.cpp");
    b.open("xx.cpp");
    while(!a.eof( ))
    {
        ch=a.get( );
        b.put(ch);
    }
    a.close( );
    b.close( );
    cout<<"File copied...";
}
```

File Modes

Files can be opened in different modes . These modes are defined in class ios.

ios::in	Open for input i.e. for reading
ios::out	Open file for output i.e. for writing
ios::app	Append data to the end of the output file
ios::ate	Append & Edit
ios::binary	Binary file to perform block i/o operations
ios::trunc	Discard contents of existing file when opening for write
ios::nocreate	Fail if the file does not exist
ios::noreplace	Do not overwrite existing file

Note: multiple modes can be combined by using binary OR (|) operator

Eg. **W.a.p to read a file copy the contents of the file into another file**

```
#include <iostream>
#include <fstream>
main()
{
    fstream a,b;
    a.open("pc.cpp",ios::in);
    b.open("cc",ios::out);
    while(!a.eof( ))
    {
        char ch=a.get( );
        b.put(ch);
    }
    a.close( );
    b.close( );
    getch( );
}
```

BLOCK I/O

Our stream classes provides us functions to perform block I/O operations. i.e. we can read a block of data from a file or write a block of data into file.

- a) **read(char * buffer,int size)** – It will read a block of data from input stream & store it into buffer.
- b) **write(char * buffer,int size)** -to write block of data into output stream.

*WAP to store 10 int values from an array into file nos.dat

```
#include<fstream.h>
void main()
{ int a[10]={545,7785,3222,666,32,577,10,3,3,533};
  fstream f;
  f.open("nos.dat",ios::out|ios::binary); //will open file in
                                          binary mode for writing
  f.write((char*)&a[0],20); // will store 20 bytes into file
  f.close();
}
```

*WAP to read 10 int values from file nos.dat & print them

```
#include<fstream.h>
void main()
{ int a[10];
  fstream f;
  f.open("nos.dat",ios::in|ios::binary); //will open file in
                                          binary mode for reading
  f.read((char*)&a[0],20); // will read 20 bytes from file
  f.close();
  for(int i=0;i<=9;i++)
      cout<<a[i]<<endl;
}
```

***WAP to Store & read a Object of type student into file stud.dat in binary format**

```
#include<iostream.h>           // save this file in stud.cpp
class student
{
    int rollno;
    char name[20];
public:
    void setInfo(int rn,char n[])
    {rollno=rn;
    strcpy(name,n);
    }
    void showInfo()
    { cout<<"Roll No is "<<rollno<<endl;
      cout<<"Name is "<<name<<endl;
    }
};

void main() // program to store object in file
{ student a;
a.setInfo(4117,"Amit Jain");
fstream f;
f.open("stud.dat",ios::out|ios::binary);
f.write((char*)a,sizeof(a));
f.close();
}

void main()// program to read object from file
{ student a;
fstream f;
f.open("stud.dat",ios::in|ios::binary);
f.read((char*)a,sizeof(a));
f.close();
a.showInfo();
}
```


FILE Pointers

Our stream objects contains 2 pointer i.e. a getPointer & a putPointer.

GetPointer: It indicates the position form where data will be read . It is present in stream if stream is opened for reading. We have the functions

long tellg() - it returns current position of getpointer

seekg(int offset,int reference) - it moves the
getpointer by offset with respect to given
reference

reference can be	ios::beg	indicates beginning of file
	ios::cur	indicates current position
	ios::end	indicated end of file

PutPointer: It indicates the position where data will be stored in stream. It is present in stream if stream is opened for writing. We have the functions

long tellp() - it returns current position of putpointer

seekp(int offset,int reference) - it moves the
putpointer by offset with respect to given
reference

***WAP to read 2nd 5th & 2nd last int value from file nos.dat (this file contains int values in binary format)**

```
#include<fstream.h>
#include<iostream.h>

void main()
{
    int a,b,c;
    fstream f;

    f.open("nos.dat",ios::in|ios::binary); //will open file in
                                           binary mode for reading
    f.seekg(2,ios::beg); // will move to byte no 2 from beginning of file
    f.read((char*)&a,2); // will read 2 bytes & will store it into variable a

    f.seekg(4,ios::cur); // will move by 4 byte from current position
    f.read((char*)&b,2); // will read 2 bytes & will store it into variable b

    f.seekg(-4,ios::end); // will move by 4 bytes in backward direction from
                          end of file
    f.read((char*)&c,2); // will read 2 bytes & will store it into variable c

    f.close();

    cout<<"nos are "<<a<<" "<<b<<" "<<c<<endl;
}
```

File Errors

While performing file i/o operations different conditions may arise. Such as

- Attempting to open a non existing file in read mode.
- Trying to open a read-only marked file in write-mode.
- Trying to open file with invalid filename.
- Insufficient disk space while writing file.
- Attempting to read beyond an End Of file.
- Media errors while performing read/write.

Every Stream has a state information associated with it. We can retrieve that information by using functions.

function	discription	Example Assumne f is a fstream Object
eof()	It returns True(1) at end of file	<pre>while(!f.eof()) { }</pre>
fail()	It returns True (1) if file read/write operation has failed.	<pre>f.read(...); if(f.fail()) cout<<"cannot read";</pre>
bad()	It returns True(1) if invalid operation is performed	<pre>f.read(..); if(f.bad()) cout<<"bad command";</pre>
good()	It returns True (1) if operation is successfully performed	<pre>f.read(); if(f.good()) cout<<"data read successfully";</pre>
rdstate()	It returns the status state data member of class ios	<pre>long s=f.rdstate();</pre>
clear()	Clear error status	<pre>f.clear();</pre>

Command Line Arguments

The ANSI C++ definition for declaring the `main()` function is either:

```
int main()
```

or

```
int main(int argc, char **argv)
```

The second version allows arguments to be passed from the command line. The parameter `argc` is an argument counter and contains the number of parameters passed from the command line. The parameter `argv` is the argument vector which is an array of pointers to strings that represent the actual parameters passed. The following example allows any number of arguments to be passed from the command line and prints them out. `argv[0]` is the actual program. The program must be run from a command prompt.

```
#include <iostream.h>
void main(int argc, char **argv)
{
    int counter;
    cout<<"The arguments to the program are:"<<endl;
    for (counter=0; counter<argc; counter++)
        cout<<argv[counter];
}
```

If the program name was `argdisplay.cpp`, it could be called as follows from the command line.

```
argdisplay one two three four
```

The next example uses the file handling routines to copy a text file to a new file.
(eg, txtcpy one.txt two.txt).

```
#include <stdio.h>
void main(int argc, char **argv)
{
    fstream in, out;
    int key;
    if (argc < 3)
    {
        cout<<"Usage: txtcpy source destination\n";
    }
    if (!in.open(argv[1],ios::in))
    {
        cout<<"Unable to open the file to be copied";
    }
    if (!out.open(argv[2], ios::out) )
    {
        puts("Unable to open the output file");
    }
    while (!in.eof())
    {
        key = in.get();
        out.put(key);
    }
    in.close();
    out.close();
}
```

Exception Handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**.

- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Syntax *try*

```
{  
    statements.....  
    .....  
}
```

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.

Syntax *throw value/object ;*

- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.

```
Syntax:      catch(datatype1 err1)
              {
                error handling code....
              }
catch(datatype2 err2)
              {
                error handling code....
              }
catch(...)
              {
                error handling code....
              }
```

This catch block with 3 periods will be called if no matching catch block is found. It is called as default catch block.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

eg.

```
main()
{
    int r;
    float a,c;
    try{
        cout<<"Enter radius in range (1-100) ";
        cin>>r;
        if(r<0)
            throw 1001;
        if(r>100)
            throw 1002;
        a=3.14*r*r;
        c=2*3.14*r;
        cout<<"Area is "<<a<<endl;
        cout<<"Circumference is "<<c<<endl;
    }
    catch(int err)
    {
        if(err==1001)
            cout<<"Radius cannot be -ve";
        if(err==1002)
            cout<<"Radius must be in range 1-100";
    }
}
```


C++ Standard Exceptions

C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs.

Exception	Description
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by new .
std::bad_cast	This can be thrown by dynamic_cast .
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by typeid .
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occurred when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

User Defined Exceptions

You can define your own exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use `std::exception` class to implement your own exception in standard way:

```
#include <iostream>
#include <exception>
using namespace std;
struct MyException : public exception
{
    const char * what () const throw ()
    {
        return "C++ Exception";
    }
};

int main()
{ try
  {
      throw MyException();
  }
  catch(MyException& e)
  {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
  }
  catch(std::exception& e)
  {
      //Other errors
  }
}
```

Templates

A template is a generic construct designed for a generic data type. A template can be used to define a family of generic classes & functions which can be used for any data type instead of some particular type. By using this technique we design algorithms so that they will work for variety of data types.

Advantages

- Allow for generalization of type
- Decrease the amount of redundant code you need to type
- Help to build type-safe code
- Are evaluated at compile-time
- Can increase performance (as an alternative to polymorphism)
- Help to build very powerful libraries

Disadvantages

- Can get complicated quickly if one isn't careful
- Most compilers give cryptic error messages(i.e error messages are not clear)
- It can be difficult to use/debug highly templated code
- Have at least one syntactic quirk (the >> operator can interfere with templates)

Function Templates

A function template is a function designed for generic datatype. Compiler automatically generates different versions of same function at compile time for different datatypes according to the values passed while calling the function. So once a function template is designed it can be used for different datatypes.

Syntax: *template*<class typename>
 function declearation...


```
#include<iostream.h>
template<class T>
    T max(T a, T b)
    { if(a>b)
        return a;
      else
        return b;
    }
```

```
main()
{
cout<<"Greatest of int values"<<max(4117,3012);

cout<<"Greatest of float values"<<max(41.17,301.2);

cout<<"Greatest of char values"<<max('A','Z');
}
```

Note : In above example different version of function max will be created by compiler at compile time according to data passed.

Class Templates

Templates are a way of making your classes more abstract by letting you define the behavior of the class without actually knowing what datatype will be handled by the operations of the class. In essence, this is what is known as generic programming; this term is a useful way to think about templates because it helps remind the programmer that a templated class does not depend on the datatype (or types) it deals with. To a large degree, a templated class is more focused on the algorithmic thought rather than the specific nuances of a single datatype.

A class template is a class designed for generic datatype. Compiler automatically generates different versions of same class at compile time for different datatypes according to the datatype specified while creating object. So once a class template is designed it can be used for different datatypes.

Syntax: to Define class template

```
template<class typename>
    class declaration...
    .....
```

Syntax: to Create an object from class template

```
className <dataType> objectName;
```

eg.

```
template<class T>
class MyData
{
    T data;
public:
    void setData(T n)
    {
        data=n;
    }
    void showData()
    {
        cout<<"data is "<<data;
    }
};

main()
{
    MyData <int> a;           // will create a MyData Object
                             // with datamember of type int

    a.setData(27);
    a.showData();

    MyData <float> b;        // will create a MyData Object
                             // with datamember of type float

    b.setData(2.7);
    b.showData();
}
```

Class Templates with Default arguments

While defining a class template we can specify some default data type. Due to this we can create object by specifying data type as well as without specifying it.

```
template<class T=int>
class MyData
{
    T data;
public:
    void setData(T n)
    {
        data=n;
    }
    void showData()
    {
        cout<<"data is "<<data;
    }
};

main()
{
    MyData <int> a;           // will create a MyData Object
    a.setData(27);           // with datamember of type int
    a.showData();

    MyData <float> b;         // will create a MyData Object
    b.setData(2.7);          // with datamember of type float
    b.showData();

    MyData c;                // will create a MyData Object
    c.setData(27);           // with datamember of type int
    c.showData();
}
```

Class string

string is a library class defined in header file <string> . It is used to store string type of data.

Constructors

```
string()           //create an empty string object
string(char *str)  // create an string object with given string
```

Member functions

```
string & append(const char *str) //appends string at the end.
string & append(const string str) //appends string at the end.
string & append(const char *str, size_type index, size_type len)
                                //appends a substr of given string

string & insert(size_type index,const char *str)
                                //inserts string at the given pos.
string & insert(size_type index,const string str) //inserts string at given pos.

size_type length( )           // returns length of string
size_type max_size( )         // returns max nos of chars the string can
                                contain.

int compare(const string &str) //compares 2 string & returns difference
                                between their ASCII values

size_type find(string &str,size_type index=0 )
                                // it starts searching for pattern from given pos
string & replace(size_type index,size_type len, const char *str)
                                //replace string in given range with given string

bool empty()                  // returns true if string is empty

string substr(size_type index, size_type len)
                                // returns substring starting from given pos. of given size
```


This class also provides us overloaded operators such as
+ (concat) , += , == , < , <= , > , >= , !=

eg.

```
#include<iostream>
#include<string>
using namespace std;
void main()
{
string a("seeta"),b("ram"),c;
c=a+b;
cout<<"string is "<<c<<endl; // o/p will be seetarram

cout<<"length of string is "<<c.length()<<endl;

c=c.insert(5," and ");
cout<<"now string is "<<c<<endl; // o/p will be seeta and ram

c=c.replace(5,5," laxman ")
cout<<"now string is "<<c<<endl; // o/p will be seeta laxman ram

}
```

NameSpaces

A namespace is a collection of classes which are defined in a block to avoid name collision. In a C++ project we may use multiple libraries designed by different companies/programmers. They may contain class having same name so to avoid name collision the libraies must be defined in a namespace.

Eg.

```
namespace MyLib
{
class Student
    {.....
    .....
    }
.
}

namespace YourLib
{
class Student
    {.....
    .....
    }
.
}

main()
{
MyLib::Student a(4117,"Amit Jain",'m');
YourLib::Student b(3012,"Gopal Pandey","computer",3);
.....
}
```

Using Statement

This statement is use to access member of a namespace. Normally members of namespace are access by using scope resolution operator (::).

For eg. MyLib::Student a(4117,"Amit Jain");

But if we are frequently referencing to the members of a namespace , Then it will be tedious having to specify the namespace & scope resolution operator. The using statement is use to avoid this problem.

Syntax: **using namespace *name*;**

For eg	using namespace MyLib; Student a(4117,"Amit jain");
--------	--

Namespace std

Standard C++ provides a different library classes & functions. These library is defined in a namespace **std**. So if we want to directly use members of it we have to use scope resolution operator.

Eg.	<pre>#include<iostream> main() { std::cout<<"Welcome to CCIT"<<endl; }</pre>
-----	--

But if we use using statement then members of namespace can be directly used.

Eg.	<pre>#include<iostream> using namespace std; main() { cout<<"Welcome to CCIT"<<endl; }</pre>
-----	---

RTTI

Run Time Type Information is a technique to get information about data type at run time.

- RTTI stands for Run-time Type Identification.
- RTTI is useful in applications in which the type of objects is known only at run-time.
- Use of RTTI should be minimized in programs and wherever possible static type system should be used.
- RTTI allows programs that manipulate objects or references to base classes to retrieve the actual derived types to which they point to at run-time.
- Two operators are provided in C++ for RTTI.
- **dynamic_cast operator.** The `dynamic_cast` operator can be used to convert a pointer that refers to an object of class type to a pointer to a class in the same hierarchy. On failure to cast the `dynamic_cast` operator returns 0.
- **typeid operator.** The `typeid` operator allows the program to check what type an expression is. When a program manipulates an object through a pointer or a reference to a base class, the program needs to find out the actual type of the object manipulated.
- The operand for both `dynamic_cast` and `typeid` should be a class with one or more virtual functions.

C++ provides us different functions/operators to perform such type of operation. These functions are defined in header file **typeinfo**.

typeid(object)

This function returns a reference of an object of type **type_info** which contains information about the object .

for eg.

```
#include<iostream>
#include<typeinfo>
using namespace std;
main()
{
    Student a(4117,"Amit Jain");
    cout<<"Type of a is "<<typeid(a).name();
    // it will return name of class i.e. Student

    int b=3012;
    cout<<"Type of b is "<<typeid(b).name();
    // it will return datatype of b i.e. int

}
```

dynamic_cast<target-type>(exp)

This operator performs the run time cast that verifies the validity of a cast. If the cast is invalid then the cast fails.

Eg. Suppose class Circle & Rectangle are derived from class shape. Then a shape pointer can point to Circle Object as well as rect object

```
Shape *p;
.....
.... // we have created object of either Circle or Rect
.....
Circle *c=dynamic_cast<Circle *> p;
if ( c )
    cout<<"object is of type Circle"<<endl;

Rect *r=dynamic_cast<Rect *> p;
if( r )
    cout<<"object is of type Rect"<<endl;
```

const_cast<type>(exp)

This operator is used to explicitly override constantness in a cast. The target type must be same as source type except for the alteration of its **const** attribute.

Where

type specifies the target type of cast &

exp specifies the expression being cast into the new type.

Eg.

```
const int a=4;
int *p;
p=const_cast<int *>(&a);
*p=(*p) * (*p);
cout<<"Value of a is "<<a<<endl;
```

static_cast<type>(exp)

This operator is used to perform a nonpolymorphic cast. It may be used for any standard conversion.

where

type specifies the target type of cast &

exp specifies the expression being cast into the new type.

For eg.

```
for(int i=0;i<=10;i++)
    cout<<static_cast<double>(i)/3<<endl;
```

reinterpret_cast<type>(exp)

This operator converts one type into fundamentally different type.

Where

type specifies the target type of cast &

exp specifies the expression being cast into the new type.

For eg. We can change a pointer into an integer or int into pointer.

For eg.

```
int a=1000;
int *p;
```

```
p=reinterpret_cast<int*>(a);
```

Multi-file programs

Multi-file programs are those programs where code is distributed into multiple files communicating with each other. This is a more practical and realistic approach towards advanced programming where we want loosely coupled code module that can communicate with each other.

Advantages

- if you write code for a class in a separate document, you can use that class in multiple programs. It increases reusability of the code.
- if you want to change anything in a class, you will only have to change it in that particularly document and the change will be automatically reflected in all the documents referring to this document.
- In large organizations; several programmers are working on a project. In such scenarios, each programmer is responsible for designing designated modules; therefore separate documents for each programmer, are convenient to code and then subsequently integrate.

For eg

Suppose our program contains 2 class Student and Result then they can be defined in separate files such as

- 1] student.cpp
- 2] result.cpp.

And then classes can be used in our main program by including these classes by using preprocessor command #include

```
#include "student.cpp"  
#include "result.cpp"
```


STL

The standard template library (STL) is the C++ library providing generic programming for many standard data structures and algorithms. The STL provides three types of components — containers, iterators, and algorithms — that support a standard for generic programming.

Advantages:

1. Support of Generic programming
2. Deliver fast, efficient, and robust code
3. Easy to use

```
#include <iostream>
#include <vector>
using namespace std

int main ()
{
vector<int>v(100); //100 is vector's size
for (int i =0;i <100;++i)
v [i ] ==i;
for (vector<int>::iterator p =v.begin();
p !=v.end();++p)
cout <<*p <<'\\t';
cout <<endl;
}
```

```
#include <iostream>
#include <vector>
using namespace std;
The library vector contains the STL 's template for the
component vector<>.
_ vector<int>v(100); //100 is vector's size
```

The STL container vector is used in place of an ordinary int array. As with any other template, it is instantiated with an existing type. Here, we use the native type int. The template class has a number of constructors. The one used here generates an int vector of size 100.

```
_ for (int i =0;i <100;++i)
v [i ] ==i;
```

The first for statement is written in exactly the same manner as a C++ loop on ordinary data. In most instances, vectors can be used in place of native arrays without changing working code besides the declarations.

```
_ for (vector<int>::iterator p =v.begin();p !=v.end();++p)
        cout <<*p <<'\t';
```

The second for statement is written using the iterator p. An iterator behaves as a pointer. STL provides the member functions begin() and end() as initial and terminal position values for the container. Note that end() returns the iterator position (or address), one past the last element of the container. Thus, end() is a guard location, or a value signaling that you are finished traversing the container.

STL Components

The STL is divided into three different components categories, as shown in the following list:

1] Containers

2] Iterators

3] Algorithms

Containers

A container is an object that holds other objects, and there are several different types. For example, the vector defines a dynamic array, a list provides a linear list, deque creates a double-ended queue, etc.

Containers come in two major families: sequence and associative.

Sequence Containers

Sequence containers store elements in a linear sequence i.e. there is a first element. A last element and each element except the first and last has exactly one element immediately ahead of it and one element after it. An array and a linked list are examples of sequence. Since, elements in sequence have a definite order, operations such as inserting values at a particular location and erasing a particular range become possible.

The STL provides three of sequence containers.

- vector
- list
- deque

Associative containers

An associative container associates the value with a key and uses the key to find the value. For example, the value should be structured representing employee information such as name, address, phone no. , etc. and the key could be unique employee number. To fetch the employee information, the program would use the key to locate the employee structure.

The associative containers are design to provide direct access to store and retrieve elements through search keys. The STL provides four type of associative containers as follows.

- set
- multiset
- map
- multimap

Iterators

Iterators are object that are, more or less, pointers. They give you the ability to cycle through the contents of the containers in much the same way that you use a pointer to cycle through an array.

Specialize Iterator

There are two specialized forms of iterators.

- Iterator adapters – which change the behavior of iterators.
- Stream iterators – allow input and output stream to behave like iterator.

Category	Description
Input	Used to read an element from a container. An input iterator can move only in the forward direction and one element at a time. This type of iterator cannot be used to pass through a sequence twice.
Output	Used to write an element to a container. An output iterator can move only in the forward direction and one element at a time. This type of iterator cannot be used to pass through a sequence twice.
Forward	Combines the capabilities of input and output iterators retains their position in the container.
Bi-directional	Combines the capabilities of a bi-directional iterator with the ability to move in the backward direction.
Random access	Combines the capabilities of a bi-directional iterator with the ability to directly access any element of the container

Algorithm

Algorithms are functions that can be used generally across a variety of containers for processing their contains. It provides plenty of algorithms, which includes sorts, searches and numerical algorithms.

Algorithm acts on container. Although each container provides functions for its basic operation, standard algorithm supports ore extended or complex operation. Standard algorithms also permit us to work with two different types of container at the same time. Standard algorithms are template functions.

To have access to STL algorithm, you must include<algorithm> in your program. The STL defines large number of algorithms; some of them are summrized in the following table.

Algorithm	Purpose
adjacent_find	Searches for adjacent matching elements within a sequence and returns an iterator to he first match.
binary_search	Performs binary search on an ordered sequence.
copy	Copies a sequence first.
count	Returns the number of elements in the sequence.
equal	Determines if two ranges are the same.
equal_range	Returns the range in which an element can be inserted into a sequence without disrupting the ordering of the sequence.
fill and fill_n	Fills a range with a specified value.
find	Searches a range for a value and returns an iterator to the first occurrence of the element.
find_end	Searches a range of subsequence. It returns an iterator to the end of the subsequence within the range.
find_if	Searches a range for an element for which a user-defined unary predicate returns true.
for_each	Applies a function to a range of elements.
max	Returns the maximum of two values.
min	Returns the minimum of two values.

Preprocessor Commands

The preprocessors are the directives, which give instruction to the compiler(**its Preprocessor**) to preprocess the information before actual compilation starts.

The **Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation.

Some preprocessor command are

Directive	Description
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.

#define Preprocessor

The `#define` preprocessor directive creates symbolic constants. The symbolic constant is called a **macro** and the general form of the directive is:

```
#define macro-name replacement-text
```

When this line appears in a file, all subsequent occurrences of macro in that file will be replaced by replacement-text before the program is compiled. For example:

```
#define PI 3.14159
void main ()
{
    cout << "Value of PI :" << PI << endl;
}
```

Function-Like Macros:

You can use `#define` to define a macro which will take argument as follows:

```
#define SQR(a) a*a
void main ()
{
    cout << "The square of 5 is " << SQR(5) << endl;
}
```

#include Directive

include Directive is used to include a file into a source file before sending source file to compilation.

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

These directives tell the CPP to get `stdio.h` from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

Conditional Directives

You can use the `#if` directive to create a conditional directive. Conditional directives are useful for testing a symbol or symbols to check if they evaluate to true. If they do evaluate to true, the compiler evaluates all the code between the `#if` and the next directive.

```
#define DEBUG
int main ()
{
    #ifdef DEBUG
        cout <<"Trace: Inside main function" << endl;
    #endif

    #ifdef DEBUG
        cout <<"Trace: Coming out of main function" << endl;
    #endif
}
```


Predefined Macros

C++ provides a number of predefined macros mentioned below:

Macro	Description
__LINE__	This contain the current line number of the program when it is being compiled.
__FILE__	This contain the current file name of the program when it is being compiled.
__DATE__	This contains a string of the form month/day/year that is the date of the translation of the source file into object code.
__TIME__	This contains a string of the form hour:minute:second that is the time at which the program was compiled.

Let us see an example for all the above macros:

```
void main ()
{
    cout << "Value of __LINE__ : " << __LINE__ << endl;
    cout << "Value of __FILE__ : " << __FILE__ << endl;
    cout << "Value of __DATE__ : " << __DATE__ << endl;
    cout << "Value of __TIME__ : " << __TIME__ << endl;
}
```

Diploma in Data Science



- PHP + My-SQL
- Python + Django
- Data Analysis
- Artificial Intelligence
- Machine learning
- Apache Hadoop

- C
- C++
- Html 5 + CSS 3
- JavaScript
- AJAX
- JQuery
- Bootstrap
- XML + JSON

FREE

With 3 Major Projects

Diploma in Java



- Core Java
- Advance Java [J2EE]
- Android Programming
- Hibernate
- Spring
- My-SQL

- C
- C++
- Html 5 + CSS 3
- JavaScript
- AJAX
- JQuery
- Bootstrap
- XML + JSON

FREE

With 3 Major Projects

Diploma in .Net



- C#
- VB.net
- ASP.net
- IIS server
- SQL Server

- C
- C++
- Html 5 + CSS 3
- JavaScript
- AJAX
- JQuery
- Bootstrap
- XML + JSON

FREE

With 3 Major Projects

Activate Word
to continue

Diploma in Web Technology



- PHP + MySQL
 - React
 - Node.js
 - Amazon Web Services
 - Kotlin [Android]
- C
 - C++
 - Html 5 + CSS 3
 - JavaScript
 - AJAX
 - JQuery
 - Bootstrap
 - XML + JSON

FREE

With 3 Major Projects

Arturo M...