OOPs in C++

The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except this function.

Class : It is a user defined data types, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Object : When a class is defined no memory is allocated but when it is instantiated (i.e., object is created) memory is allocated.

Encapsulation : In OOP, Encapsulation is defined as binding together the data and the functions that manipulates them

Abstraction : Abstraction means displaying only essential information and hiding the details.

- Abstraction using classes
- Abstraction using Header files (math.h → pow())

Polymorphism : In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

- Operator overloading
- Function overloading
  - ↳ int sum(10, 20, 30)
    int sum(10, 20)

Inheritance : The capability of a class to derive properties and characteristics from another class is called inheritance.

- SubClass
- SuperClass
- Reusability

Dynamic Binding : In dynamic binding, the code to be executed in response to function call is decided at run time.

Constructors : A constructor is a member function of a class which initializes objects of a class. In C++ constructor is automatically called when the object creates.

It has same name as class itself.

Constructor don't have a return type.

1. Default Constructor (No parameter passed)
2. Parametrized Constructor
3. Copy Constructor

Destructor in C++ : Derived class destructor will be invoked first, then the base class destructor will be invoked.

Access Modifier : Public – can be accessed by any class.

Private :- can be accessed only by a function in a class (inaccesible outside the class).

Protected :- It is also inaccessible outside the class but can be accessed by subclass of that class.

Note : If we do not specify any access modifier inside the class then by default the access modifier for the member will be private.

Friend class : A friend class can access private and protected members of other class in which it is declared as friend.

Ex-: friend class B ;

- Inheritance

```
Class subclass : accessmode baseclass
{ ____
}
```

1. Single inheritance

2. Multiple inheritance

3. Multilevel

4. Heerarchical inheritance

5. Hybrid

Combination of one or more type.

- Polymorphism

→ Compile time Poly → Operator overloading, Function overloading.

→ Run time Poly

↳ function overriding occurs when a derived class has a definition of one or more members of base class.

Advantages of Data Abstraction

- Avoid code duplication and inc. reusability.
- can change internal implementation of class independently.

Structure Vs class : Most important difference is security.
A structure is not secure and cannot hide its member function and variable while class is secure and can hide its programming & designing details.

Local Classes in C++ : A class declared inside a function becomes local to that function and is called local class.
All the methods of local class must be defined inside the class only.

Virtual function and Runtime Polymorphism :
A virtual function is a member function which is declared within a base class and redefined (overriden) by derived class.
Functions are declared with Virtual Keyword in base class.

Exception Handling in C++ :
try : represent a block of code that can throw an exception.
catch : represent a block of code that get executed when error is thrown
throw : Used to throw an exception.

There is a special ~~catch~~ block → catch(...)
It catches all types of error.

- Inline Function

→ inline is a request not command.
It is function that is expanded in line when it is called. When the inline function is called, whole code get inserted or substituted at the point of inline function call.

```
inline return-type fun( )
{
}
```

- Function Overloading is a feature in C++ where two or more functions can have same name but different parameters.

```
void print(int i)
{       cout << "Here is int " << i << endl;
}

void print (float i)
{       cout << " Here is float" << i << endl;
}

int main
{       print(10);
        print(10.12);
}
```

Differences b/w C and C++

C	C++
1. C supports procedural prog.	• C++ is known as hybrid language, because it support both procedural and object oriented programming.
2. As C does not support the OOPs concept so it has no support for polymorphism, encapsulation and inheritance.	• C++ has support for polymorphism, encapsulation and inheritance as it is an OOPs language.
3. C is a subset of C++	• C++ is superset of C
4. C contains 32 keywords	• C++ contain 52 keywords (public, private, protected, try, catch, throw...)
5. C is a function driven language	• C++ is an object driven language.
6. Function and operator overloading is not support in C.	• C++ supports function & operator overloading.
7. C does not support exception handling	• C++ ~~does~~ supports exception handing using try and catch

- Structure is a Collection of dissimilar elements

- Static Members in C++

  - **Static variable in a Function** : When a variable is declared as static, space for it gets allocated for the lifetime of the program. (defaut intialized to 0) Even if the function is called multiple times, the space for it is allocated once.

  - **Static variable in a class** :
    - → Declared inside the class body.
    - → Also known as class member variable.
    - → They must be defined outside the class.
    - → Static variable doesn't belong to any object, but to the whole class.
    - → There will be only one copy of static member variable for the whole class.

Ex:

```
class Account
{
    private :
        int balance;
        static ~~int~~ float roi ;
    public :
        void setBalance(int b)
        {   balance = b ;  }
};
// intialised outside class
float Account :: roi = 3.5f ;
void main
{
    Account a1;
}
```

- Object can also be declared as static.

```
static Account a1;
```

- Static function in a Class

Static member functions are allowed to access only the static data members or other static member functions.

- Constructors :

→ Constructors is an special member function of the class. It is automatically invoked when an object is created.

→ It has no return type.

→ Constructor has same name as class itself.

→ If we do not specify, then C++ compiler generates a default constructor for us.

Constructor

Default	Parameterized	Copy
Class_name():	Class_name(parameters)	Class_name(const Class_name &obj)
update()	update(int x, int y)	update(const update &p2)
{	{	{ a=p2.a;
a=10;	a=x;	b=p2.b;
b=20;	b=y;	}
}	}	

Compiler generates two constructor by itself.

1. Default Constructor
2. Copy Constructor

But if any of the constructor is created by user, then default constructor will not be created by compiler.

Construction overloading can be done just like function overloading.

Default (Compiler's) Copy constructor can done only shallow copy.

(Points to same memory location)

Deep Copy is possible only with user defined constructors. In user defined copy constructor, we make sure that pointers of copied object points to new memory location.

Can we make Copy Constructor private? Yes

Why argument to copy constructor must be passed as a reference?

Because if we pass value, then it would made to call copy constructor which becomes non-terminating.

Deep Copy

- Destructor

→ Destructor is a member function which destructs or deletes an object.

→ Destructor don't take any argument and don't have any return type.

→ Only one destructor is possible

→ Destructor cannot be static.

→ Actually destructor doesn't destroy object, it is the last function that invoked before object destroy.

Destructor is used, so that before deletion of obj we can free space allocated for this resource. B/c if obj gets deleted then space allocated for obj will be free but resource doesn't.

- Operator Overloading

C++ have the ability to provide special meaning to the operator.

```
class Complex
{
        ...

        Complex operator + (Complex &c1)
        {       Complex res;
                res.a = c1.a;
                res.b = c2.b;
        }
}

int main()
{       c = c1 + c2
}
```

As '+' can't add complex no's directly. So we can define a function with name + but we need write operator keyword before it.
So, we can use all operator like this.

Friend Class

A friend class can access the private and protected members of other class in which it is declared as friend.
There can be friend class and friend function.

Ex:

```
class Box
{
    private:
        double width;
    public:
        friend void printWidth(Box box);
        void setWidth(double Wid);
}

void Box::setWidth(Box double Wid)
{
        width = Wid;    }

void printWidth(Box box)
{
    cout << box.width;  }

int main()
{
    Box box;
    box.setWidth(14);
    printWidth(box);
}
```

Inheritance

It is a process of inherting properties and behaviour of existing class into a new class.

class Base_class
{
};
	class der_class : ~~base~~ Visibility_Mode Base class.
	{
	};

Ex:	class Car
	{
	};
		class Sports_Car : public Car
		{
		};

Types of Inheritance :

a). Single Inheritance :

```
class B : public A
{

};
```

b). Multilevel Inheritance :

```
Class B : public A
{
};
class C : public B
{
};
```

c). Multiple Inheritance

```
class A1          class A2
{                 {
};                };

class B : public A1, public A2
{
};
```

d). Heirarchial Inheritance

```
class B1 : public A
{
};
class B2 : public A
{
};
```

→ Visibility Mode :

A - base class

B - Sub Class

If B is subclass and visibility Mode is public.

```
class A : public B
{
};
```

then public member of A will be public in B, and protected will protected.

If visibility mode is private then both protected and public member of A will be private member of B

- Is a Relationship is always implemented as a public inheritance.

- Constructor and Destructor in Inheritance

First child class constructor will run during creation of object of child class, but as soon as obj is created child class constructor run and it will call constructor of it's parent class and after the execution of parent class constructor it will resume it constructor execution.

child

```
B() : A()
{
}
```

parent constructor call

constructor exec,
obj
↓
Child const
↓
parent const
↓
complete parent
↓
complete child

While in case of destructor, first child destructor exec, then parent desc. executed.

this pointer

Every object in C++ has access to its own address through an important pointer called this pointer.

Friend function doesn't have a 'this' pointer, b/c friends are not members of a class. Only member function have this pointer.

```
Class Box
{
    private:
        int l, b, h;
    public:
        void set (int l, int b, int h)
        {
            this->l = l;
            this->b = b;
            this->h = h;
        }
};

int main ()
{
    Box b;
    b.set (5, 10, 4);
}
```

Method Over Riding
(achieved at run time)

It is the redefinition of base class function in its derived class, with same return type and same parameters.

while method Overloading is achieved at Compile time.

Ex:

```
Class Car
{
    private :
        int gearno ;
    public :
        void change_gear( int gear)
        {
            gear ++ ;
        }
}

Class SportsCar : public Car
{
        void change_gear( int gear)
        {
            if (gear > 5)
                gear ++ ;
        }
}

int main
{
        SportsCar sc ;
        sc . change_gear (4) ;
}
```

function of sports Car Class will be called.

While calling change_gear() , first it check if any fun with this name exist in ~~be set~~ calling class, otherwise it goes to baseClass.

Useful : like we have change_gear for all except one car which have unique method of gearChange.

Virtual Function

A virtual function is a member function which is declared with a 'virtual keyword' in the base class and redeclared (overridden) in a derived class. When you refer to a object of derived class using pointer to a base class, you can call a virtual function of that object and execute the derived class's version of the function.

- They are used to achieve Run time Polymorphism.
- Virtual Function cannot be static and also cannot be friend function of another class.

Compile-time (Early binding) Vs Run-time (Late Binding)

```
class base
{        public :
            virtual void print()
            {        cout<< " This is base print" << endl;
            }
            void  show()
            {     cout << " Base show fun" << endl;
            }
}
class  derived
{        public:
            void print()
            { cout << " derived Print" << endl; }
            void show()
            { cout << " derived show fun" << endl; }
}
```

```
int main ()
{
        base *bptr ;
        derived  der ;
        bptr  =  &der ;

        bptr -> print() ;          // Run time
        bptr -> show() ;           // Compile time
}
```

output: derived print // Late Binding
~~This~~ Base show fun // Early binding

As during compiler time bptr behaviour judged on the bases of which class it belong, so bptr represent base class.

If the function is not virtual then it will allow binding at compiler time and print fun of base class will get binded b/c bptr represent base class.

But at run time bptr points to the object of class derived, so it will ~~to~~ bind function of derived at run time.

Working of Virtual Function (VTable & VPtr)

If a class contains virtual function then compiler itself does two things:

1. A virtual pointer (VPTR) is created every time obj is created for that class which contains virtual function.
2. Irrespective of object is created or not, static array of pointer called VTABLE where each cell point to each virtual function is created, in base class and derived class.

Pure Virtual Function and abstract Class

Sometimes implementation of all function cannot be provide in the base class. Such a class is called abstract class.

A pure virtual function in C++ is a virtual function for which we don't have any implementation, we only declare it.

// Abstract Class

```
class Test
{
    public :
        virtual void fun() = 0;
}
```

Pure Virtual function

1. A class is abstract if it has at least one pure virtual function.

We cannot declare object of abstract Class.
Ex: Test t; will show error

2. We can have pointer or reference of abstract class.

3. We can access the other functions except virtual by object of its derived class.

4. If we dont override the pure virtual function in derived class then it becomes abstract.

5. An abstract class can have constructors.
(Read from GfG)

Template in C++

```
template <class X> int check (int a, x b)
                    {
                        if (a>b)
                            return a;
                        else return b;
                    }
```

It ~~does~~ just help in data type. So that we can write generic function that can be used for different data type.

Dynamic Constructor

When allocation of memory is done dynamically using dynamic memory allocator 'new' in constructor.

```
Class geeks
{       public:
            void fun() { p= new char[6]; }
}
int main()
{       geeks g = new geeks();
}
```

OBJECT ORIENTED PROGRAMMING

(with C++)

BOOKS TO REFER

- E. Balagurusamy,Object Oriented programming with C++.
- Robert Lafore,Object Oriented programming in C++.
- The Complete Reference (C++)

Procedure Oriented Programming

- Logically related statements are grouped into functions(or procedures).
- such function performs some logical task.
- A function can call other such functions.
- functions may share some global data.
- Main drawback is global data sharing, due to which adequate synchronization is to be established among all functions using it.

Object Oriented Programming

- The data and the function operating it is coupled to to form a CLASS.
- Main features of such programming are

1. **Encapsulation**

   (coupling data and functions together)

2. **Inheritance**

   (Inheriting some class and add more functionality to new class)

3. **Polymorphism**

   (more than one class in inheritance branch may have same named function)

Object oriented programming languages

1. ADA
2. C++
3. Java
4. C# (C sharp)

- Most of them also supports full syntax of procedure oriented programming .
- Each of them supports object oriented features to varied degree.

What is C++?

- An object oriented programming language.
- An extension of C (with added feature of classes).Thus it is a superset of C.
- Follow bottom up programming approach.

- **How C++ differs from ANSI C ?**

1. It has a different set of identifiers than C(like bool,export,using,false, true etc.)
2. In C++ main returns an “int” value always (cant has return type VOID).
3. It doesn’t put any limit on length of identifiers whereas some versions of C does.

3. C++ doesn't automatically convert an "int" value to an enumerated value.(which C does)

e.g enum colour {red,green,yellow};

colour c; c=red;

c=3;// **wrong in C++ ,correct in C**

(in C++ it should be c=colour(3);

- In C++ ,Function taking no arguments *should* be declared as (*not doing so wont result in any error)*

return_type Func_name(void).

Programming Features

1. Main returns an "Int"
2. Comments may be

✓ single line //

e.g //this is documentation

OR…..

✓ Multiple lines /*…….*/

e.g/* this is

documentation*/

3. Spaces and tabs can be inserted anywhere to increase readability

e.g a=12*b-(c/2) is same as 12 * b –(c / 2)

4. cout identifier is used with operator "<<"to output data to output devices.

   e.g cout<<name;

   << is also used for bit wise left shift operations, thus << is overloaded to perform both as shift operator and "put to " operator.

5. cin identifier is used with operator ">>" to take inputs from input devices.

   E.g cin>>name;

   >> is also overloaded operator to perform both as right shift and "get from"operator.

cin and cout operators can be cascaded as

cin>>a>>b; **OR** cout<<"name is"<<name;

Types of Applications with C++

- Object oriented libraries can be developed that can be used by several applications later.
- Since C++ is easily expandable and maintainable , any application that always requires new additions can be well developed with C++.
- C ++ standard library has two halves

1. Standard C library.
2. Class library(that supports object oriented programming)

NameSpace

- This defines a scope for the identifiers that are used in a program.
- For using the identifiers defined in the namespace scope we must include the **using directive**, like

  Using namespace std;

- **std** is the namespace where C++ standards class libraries are defined.
- This will bring all the identifiers defined in **std** to the current global scope.

Structure of C++ program

- Any C++ program includes

1. **Header files** (are referred with the help of #include directive)

Syntax is

#include<iostream.h>(this file will be searched in space assigned to include files)

OR

#include "iostream.h"(this file will be searched in current working directory)

2. **Class declaration**
3. **Member function definitions**
3. **Main function**

These sections can be **coded** in different files and then **linked** at the time of compilation.

Program is organised into 3 files

- The class declaration are placed in a header file.
- The definition of member functions go into another file
- Finally main program that uses the class placed in third file which includes other two files.

The approach is based on client-server model.

- Class definition including member function constitute **server** that provides sevices to the main program known as **client**.

Tokens

Smallest Individual units in a program. C++ has following tokens :

- Keywords
- Identifiers
- Constant
- Operators

KEYWORDS in C++

- Are reserved identifiers in C++.
- They cannot be used as names(identifiers) for the program variables , functions etc.
- Examples include

Double,new,else,if,break,continue,long,

int,float,for ,while,class,inline,struct,void etc.

IDENTIFIERS in C++

- **Identifiers** are user defined names for variables,functions,labels etc.
- Rules to name a variable in C++ are

1. It can include alphabets ,digits , underscore(_).
2. It can not start with a number.
3. C++ is Case sensitive thus name, and NAME are different identifiers.
4. Keywords cannot be used as variable names.

CONSTANTS in C++

- **Constants** refer to fixed values that do not change during the execution of a program.
- Like C , C++ supports various literal constants. They include integers , characters, floating point numbers and strings.

1. 123
2. 12.34
3. “Abc”
4. ‘h’

OPERATORS in C++

- Relational operators(*represent relationships between values)*

Operator	Action
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equals

- Logical operators (*used to connect relationships*)

Operator	Action
&&	AND
||	OR
!	NOT

- Bit wise operators(*performs operations on bits of char and int data types)*

Operator	Action
&	**AND**
|	**OR**
^	**Exclusive OR**
~	**Complement**
>>	**Shift Right**
<<	**Shift Left**

- Mathematical Operators (perform mathematical operations)

- Addition(+)
- Subtraction(-)
- Multiplication(*)
- Division(/)
- Modulus(%)
- Unary minus(-)
- Unary Plus(+)

- Assignment Operator(=), *for assigning value to a variable.*
- scope resolution operator (::) *to define scope of a variable or function*.
- Conditional Operator(?)is a ***ternary*** operator can replace if-then-else structure

Expr? Val1:Val2

Expr is evaluated ,if true value of expression will be *Val1* else *Val2.*

- Address of operator(&) returns the memory ***address of*** the operand

```
int *p,a,b;
p=&a;
```

- Pointer operator(*) returns ***value at address*** specified by the operand.

```
b=*p;
```

- Sizeof operator returns length (in bytes) of the operand

```
int Len , a;
Len=sizeof(a);
```

(Len will be assigned 2(size of integer data type))

DATA TYPES in C++

DATA TYPES

Built-in

Integer

Character

Float

Void

Derived

Array

Pointer

Reference

User-defined

Structure

Union

Class

Enumeration

Void Data Types

- **Two uses** of void are:

1 To specify the return type of function.

2 To indicate an empty list of arguments.

Eg: **void func1(void);**

- Another use of void is in declaration of generic pointers. Eg: **void *gp;**
- A generic pointer can be assigned a pointer value of any basic data type. Eg:

```
int *ip;
gp = ip;
```

Enumerated Data Type

- User Defined data type.
- Provides a way of attaching names to no.
- *Enum* keyword assigns the values 0,1,2 & so on. e.g
  enum colour {red, green, yellow};
- We can explicitly assign the values. Eg :
  enum colour {red, green=4, yellow=7};
- By using these tag names, we can declare new variables. Eg: **colour background;**
- C++ doesn't automatically convert an "int" value to an enumerated value.(which C does)

  colour c; c=red;

  c=3; // **wrong in C++ ,correct in C**

  c=colour(3); // in C++ it should be

POINTER & REFERENCE DATA TYPES

1. **Pointer:** A pointer is a variable that holds the memory address of other variable. It is of different data types, e.g- char pointer can store address of only char variables, int pointer can store address of int variables and so on.
2. **Reference:** A reference in the simplest sense is an alias or alternate name for a previously defined variable.

Declaration of variables in C++

- C++ permits declaration of variables anywhere,before or at its first use.
- E.g

```
int main()
{
…….
For(int I=1;I<5;I++)
{…….}
double sum,number;
}
```

Dynamic initialization of variables

- In C++ variables can be initialized at run time(*without using constant expressions*)
- *E.g*

```
int main()
{.....
int sum=32 ,number=4;
.........
float average=sum/number;
}
```

Operators in C++

- :: scope resolution operator
- delete memory release operator
- endl line feed operator
- new memory allocation operator
- setw field width operator
- ::* Pointer to member declarator
- ->* Pointer to member operator
- .* Pointer to member operator

Manipulators

- The endl manipulator , causes a linefeed to be inserted. It has the same effect as that of “\n” .

Ex: cout << « Infosys" << endl;
cout << "Training";

- The setw manipulator sets the minimum field width on output. The syntax is: setw(x)

```
int  main( )
 {
   int  x1=12345, x2= 23456,  x3=7892;
   cout<<setw(8) <<”Infosys”<<setw(20)<<”Values”<< endl
   << setw(8) << “E1234567” << setw(20)<< x1 << end
   << setw(8) << “S1234567” << setw(20)<< x2 << end
   << setw(8) << “A1234567” << setw(20)<< x3 << end;
   return 0 ;
   }
```

Reference Variables

- A reference variable provides an alias (alternative name) for a previously defined variables. Syntax :

  Data-type & reference-name = variable name ;

- **Ex : float total =100;**

  float & sum = total;

- Here **&** is not an address operator but ***float &*** means reference to float.

```
int n[10] ;
int & x = n[10] ;
char & a ='\n' ;
```

(i)

```
int x;
int *p = &x ;
int & m = *p;
```

(ii)

```
int & n = 50;
```

```
//Program to illustrate References
#include<iostream.h>
int main(void)
{
    int var;
    int & refvar=var;
    //here a reference variable to var is declared
    var=10;            //var is given the value 10
    cout<<var<<endl;
    refvar=100;    //reference variable of var is changed
    cout<<var;   //but var also gets changed
    return 0;
}
```

Scope Resolution Operator

- The scope resolution operator (::) in C++ is used to define the already declared member functions.
- Syntax : *:: variable-name*

```
#include <iostream>
using namespace std;
int n = 12;  // A global variable
int main()
{ int n = 13;  // A local variable
  cout << ::n << endl;   // Print the global variable: 12
  cout << n << endl;  // Print the local variable: 13
 }
```

Memory Management Operators

- Two operators new & delete are used to allocate the memory dynamically. Syntax:
- ***pointer-variable = new data-type;***
- int *p = new int ;

  float *q = new float;

  *p = 25 ;

  *q = 7.5 ;
- Can also initialize as

  pointer-variable = new data-type(value);
- For arrays :

  pointer-variable = new data-type[size];

- The fields must not be empty.

```
array_ptr = new int [4][6][6] ; // legal
array_ptr = new int [4][ ][8] ; //illegal
array_ptr = new int [ ][3][9] ; // illegal
```

- The delete operator is as:

delete pointer-variable ;

Ex: delete p;

delete q;

For array:

delete [size] pointer-variable;

Ex : [ ] a;

Types of expressions

- **Constant Expression**(all values are constant)

for e.g 13+2 and 6+7*67

- **Integral Expression** (integral results)

For e.g m*2 and a+b*3-c(a,b,c are integers)

- **Float Expression** (float results)

For e.g a+b/2 and 12.34+a

- **Pointer Expression**(producing address)

For e.g &m and *ptr

- **Relational Expression**(yielding true/false)

  For e.g a>b , (c==4)

- **Logical expression** (having &&, | | etc)

  For e.g a>2 **&&** b>=5

- **Bit wise expressions (having bitwise operators)**

  For e.g a<<3(shift 3 bit position to left)

  b>>2(shift 2 bit positions to right)

Special assignment expressions

1. **Chained Assignment**

   z=(y=12) or z=y=12

   // same z=12 and y=12

2. **Embedded assignment**

   x=(y=22)+12;

   // y is assigned 22 first then x is assigned 34.

3. **Compound Assignment(*combination of assignment operator with binary arithmetic operator)***

for e.g x+=12; // **same as x=x+12;**

Also called as *short hand* operator.

Implicit Conversions

- **While evaluating an expression having mixed data types *"smaller" type is converted to a "wider" type***

ie if one operand is float and other is int for a binary operator then int is being widened to float type.

Short/char→ Int→ unsigned→long int→unsigned long int→float→double→long double.

Type Casting

- Explicitly changing data type(***either a smaller into wider or vice versa)***
- type-name (expression)

  int a=23; float b=12.32; int(b);
- If type-name is not an identifier e.g char * then
- **(char*)var** is used **p = (int *)q;**

  OR
- **typedef char* char_pt;**

  then char_pt(var) is used.

  p = char_pt(q);

Control Structures in C++

- If-else
- Switch

} **Selection**

- do-while
- while
- For
- Break

} **Iteration/Loop**

- Continue
- go to
- Return

} **Jump**

these structures are used to control the flow of execution in a C++ program.

A C++ program

```
// FIRST PROGRAM
#include<iostream.h>
 int main()
{
Float n1,n2,sum,average;
cout<<"enter two numbers";
cin>>n1>>n2;
sum=n1+n2;
average=sum/2;
cout<<"sum is"<<sum<<"and average is "<<<average;
return o;

}
```

Questions

- 1. Which is a valid identifier in C++?

  a. abs.no

  b. _front!

  c. int_no12

  d. 1_no123

  e. int$no_is

ANS C

2. Which is true about C and C++?

a. C is a superset of C++
b. C is a subset of C++
c. Both have different spaces
d. Both are same
e. None.

ANS B

3. Type Casting in C++ is ….

1. To convert smaller data type into a wider one.
2. To convert wider data type into smaller one
3. same as explicit type conversion .
4. Never required in C++, as automatic conversion take place in it.

ANS 1,2 and 3

4. Which will run well with C++?

1. Printf(“ my name is XYZ”);
2. Cout>>” Enter your name”;
3. Cin>>name>>Number;
4. Do { statements } while condition;

ANS 1, 3 and 4

5. Which syntax is not correct in C++?

a. For(a=1,b=3;a*b<100;a++);
b. Switch (char array)
   {case “abc”: ……;break;}
c. for(;;){ statements…..}
d. For(;a<10;a++);
e. For (int a=1;a<13;);
f. All of the above

ANS b

Assignment -I

- Differences between switch and if-then else structures.
- Meaning and usage of keywords…

  static,const,extern
- Differences between break and continue statements.
- How to use Exit() function? Where it is defined? What can be its arguments?

Functions in C++

Function Prototype

- A function declaration.
- All functions must be declared(prototyped) (*anywhere)* before its use.
- Return_type function name (argument-list);
- Argument list has structure as

  (data type1,datatype2,data type3)
- Names for arguments are not required while declaring it *(they don't have to be matched with names used for definition or call)*

- All library functions used in a program has prototypes and definitions in header files included.
- Compiler uses prototype to check number and types of arguments used while calling that function.

Function Definition

- The general form of a function definition in C++ is as follows:

```
return-type function-name( parameter-list )
{
  local-definitions;
  function-implementation;
}
```

- If the function returns a value then the type of that value must be specified in *return-type*. For the moment this could be int, float or char. If the function does not return a value then the function-type must be void.

- Prototype is not required if function is defined before its use.
- The *function-name* follows the same rules of composition as identifiers.
- The *parameter-list* lists the formal parameters of the function together with their types.
- The *local-definitions* are definitions of variables that are used in the function-implementation. These variables have no meaning outside the function.
- The *function-implementation* consists of C++ executable statements that implement the effect of the function.

Passing arguments in functions

1. Call by value (*copy of actual parameter is acted upon)*
2. Call by address(*address of actual parameter is acted upon)*

```
Void swap1(int a,int b)
{
int temp=a;
A=b;
b=temp;
}
```

- Void swap2(int* a, int* b)

```
{
  int temp;
  temp=*a;
  *a=*b;
  *b=temp;
}
```

- Void swap3(int &a, int b)

```
{
int temp;
temp=a;
a=b;
b=temp;
}
```

```
int main()
{
int m=2;n=1;
swap1(m,n);
cout<<m<<n;
swap2(&m,&n);
cout<<m<<n;
swap3(m,n);
cout<<m<<n;
}
```

What is an inline function?

- Functions defined with “inline” directive
- at each call to such function , its function body is inserted at that point in compiled code.
- If function body is small and called at several places then it should be made inline as there is no overhead in CALL and RETURN statements .
- Functions defined inside a class are by default INLINE.

using inline function….

```
#include<iostream.h>
inline float prod(float a,float b)
 { return(a*b); }
inline double div(double a,double b)
{ Return(a/b);}
 int main()
{
float a=12.654,b=3.45;
cout<<prod(a,b)<<endl;
cout<<div(a,b)<<endl;
return 0;
}
```

- Inline keyword sends only a request*(not a command)* to the compiler, the compiler may ignore the request and treat the function as normal one if function definition is too long or complicated.
- **Situations where inline expansion may not work**:

1 For functions returning values , if a loop , a switch, or a goto exist.

2 For functions not returning values, if a return statement exists.

3 If functions contains static variables.

4 If inline functions are recursive.

Default arguments

- The argument whose default value is mentioned at function declaration can be skipped from actual parameter list at the time of function call.

e.g ret-type function-name(type1,type2 n=6)

- If some value is passed to default argument then default value is replaced by that value.
- Are useful where one or more parameter remain same for each call.

```
#include<iostream.h>
int main(void)
{  char name[12];
   int no;
  cout <<"enter name and no";
  cin >>name>>no;
  void display (int , char name[12]="default");
  display(no,name);
  display(no);       }
  void display (int n,char name[12])
  { cout<<"name is“<<name;
     cout<<endl<<"no is"<<n<<endl;
}
```

- Output of the program will be

enter name and no aarti 23

name is aarti

no is23

name is default

no is23

Function Overloading

What is Function overloading?

- A number of functions having same name but performing different tasks.
- Every function has differs from others in return type or argument list.
- At function call ,**function definition with corresponding arguments is invoked.**
- Argument lists may differ in either type of arguments or number of arguments.

Example:

- // Declaration

```
int add(int a, int b);   // 1
int add(int a, int b, int c);  //2
double add( double x, double y); //3
double add (int p , double q); //4
double add (double p, int q );  //5
```

- //Function calls

```
cout<< add(5,10);  // uses 1
cout<< add(15,10.0); // uses 4
cout<< add(12.5,7.5); // uses 3
cout<< add(5,10,15); // uses 2
cout<< add(9.75,5); // uses 5
```

- If compiler is unable to find an exact match then it tries to use promotions to the actual arguments and again match new argument list with function definitions.

  char --→int

  float--→double

- If after promotions more than one match is found then it gives error for ambiguity.

  long square (long n);

  double square (double x);

Function call : square(10); // so Error

```
#include<iostream.h>
#include<conio.h>
double volume (float l, float w, float h)
{
return (l*w*h);
}
/*double volume (float l, float w, double h)
{
return (l*w*h);
} */( WILL  GIVE AN ERROR)
```

```
double volume (float r, float h)
{
double vol=3.14*r*r*h;
return vol;
}
int main(void)
{
clrscr();
cout<<endl<<"volume of a cube is
   "<<volume(2.6,12,21.25);
cout<<endl<<"volume of a cylinder
   "<<volume(2,2.8);
return 0;
}
```

OUTPUT

- volume of a cube is 662.999976
- volume of a cylinder 35.167999

Assignment II

A) Write a c++ program to overload average function to compute average of three and four numbers respectively.

B) Write a C++ program to overload a function for computing area of a rectangle and a circle .

C) Write any C++ program to use overloaded constructor for a class with one constructor having default arguments.

OBJECT ORIENTED PROGRAMMING

(with C++)

Unit-3

Generic Function

Unit-III

Inheritance, Class hierarchy, derivation – public, private & protected; aggregation, composition vs classification hierarchies

polymorphism, categorization of polymorphic techniques, method polymorphism, polymorphism by parameter, operator overloading, parametric polymorphism,

generic function – template function, function name overloading, overriding inheritance methods, run time polymorphism.

Templates

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- Templates allows function and classes to operate with generic types.
- *There are 2 type of templates:*
  - *Function templates* – A function template is a *"generic"* function that can work with different data types. L*ike macros, It enables software reuse. But unlike macros, fn templates help eliminate many type of errors through scrutiny of full c++ type checking.*
  - *Class templates* - Allows type-specific versions of generic classes. Just like function templates, we can also define class templates.

Templates

- ***Template Parameter*** - A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameter which is used to pass values to a function, template parameter allows to pass values and also types to a function.
- ***Template Instantiation*** – *When the compiler generates a class, function or static data members from a template, it is referred to as template instantiation.*
- *A function generated from a function template is called a generated function.*

Function Templates

Function generated from function template is called template function.

Syntax:

```
template <class T> returnType FnName(Parameter List)
    //Template function
{
    :
}
```

Templates function with 2 arguments of same/different type depending on argument passed e.g.

```
template <class T, class U> T getMin(T a, U b)
{
    return (a>b?a:b);
}
```

Function Templates

Example:

```
#include <iostream.h>
template<class x> void swap(x &a, x& b)
{
	x temp;
	temp=a;
	a=b;
	b=temp;
}
```

```
main()
{
	int i=10,j=20;
	double x=10.1,y=21.3;
	char a='x', b='z';
	swap(i,j);
	swap(x,y);
	swap(a,b);
	cout<<i<<j;
	cout<<a<<b;
	cout<<x<<y;
}
```

Function Templates

- A function template can be overloaded as well as overrided.
- Example of function Template overloading:

```
#include<iostream.h>
template <class t> void Max(t a, t b)
{
      if(a>b)
          cout<<a;
      else
          cout<<b;
}
```

```
template <class t> void Max(t a, t b, t c)
{
        if(a>b && a>c)
            cout<<a;
        else if(b>a && b>c)
            cout<<b;
        else
            cout<<c;
}
main()
{
    Max(1,2);
    Max(3,2,1);
}
```

Function Templates

- If a template is invoked with a user defined type, and if that template uses functions or operators (e.g. ==,+,<= etc.) with objects of that class type, then those functions and operators must be overloaded for the user-defined type. Forgetting to overload such operators causes compilation error.

Example

Program to define the function template for calculating the square of given numbers with different data types.

```
template <class T> T square(T number)
{ return number * number; }
int main()
{
// Get an integer and compute its square
int iValue;
cout << "Enter an integer value: ";
cin >> iValue;
// compiler creates int square(int) at call to square with an int argument
cout << "The square is " << square(iValue);  //square<int>(iValue);
// Get a double and compute its square
cout << "\nEnter a double value: ";
double dValue;
cin >> dValue;
// compiler creates double square(double)on call to square with double arg
cout << "The square is " << square(dValue);  // //square<double>(dValue);
return 0;
}
```

Class Templates

- Class created from a class template is called template class
- Creating a class from class template is called Instantiation
- Syntax:

```
template <class t> class className
{
      :
}
```

For objects:

```
className <Type> objectName(Arguments);
```

e.g.

```
 className <int> obj1(10);
className <float> obj2;
```

Class Templates

- Non-Type Parameters & Default Type

Non-Type Parameter

```
template <class T, int Size>
    class Array
{
        T a[size];
        :
};
```

For Object:

```
        Array<int,10> obj1;
                //Instantiation
        Array<float, 5> obj2;
```

Default type

```
template <class T= int> class Array
{
        T a[5];
        :
};
```

For Object:

```
        Array<> obj;  //Instantiation
```


Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Class Templates

- Member function templates - Syntax:

```
template <class T> retType className<T>::fnName(Arguments)
{
    :
}
```

e.g.

```
template<class T>
T Point<T> :: operator *(Point & P)
{
    :
}
```


Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Example

Program to find the bigger of two entered numbers using class template.

```
template <class T> class pair
{
        T a; T b;
        public:
        pair()
        {
        cin>>a>>b;
        }
        T get_max();
};
Template<class T>
T pair<T>::get_max()
{
        T ret;
        ret=a>b?a:b;
        return ret;
}
```

```
void main()
{
        cout<<"\nEnter 2 Integer numbers:";
        pair<int> obj1;
        cout<<"Greatest Integer is”
        <<obj1.get_max();
        Cout<<“Enter 2 Float Numbers:\n";
        pair<float> obj2;
        cout<<"Greatest Float is”
        <<obj2.get_max();
}
```

Templates & Inheritance

A class template can be derived from a template/non-template class

Example

```
template <class T> class Base
{
    protected:
            T var;
    public:
            Base(T val): var(val)      { }
            T Get()  { return var;      }
};
template <class T> class Derived : public Base<T>
{
    public:
            Derived(T valu): Base<T>(valu)  { }
            T fun()  {  return var;      }
};
```


Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Function Templates & Static Variable

Each instantiation of function template has its own copy of local static variables.

Example

```
template <class T> void fun(const T &x)
{
    static int i=10;
    cout<< ++i;
}
main()
{
    fun<int> (1);   //prints 11
    fun<int> (2);   //prints 12
    fun<double> (1.1);     //prints 11
}
```

Output:

11
12
11

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Class Templates & Static Variable

Each instantiation of class template has its own copy of member static variables.

Example

```
template <class T> class Test
{
    private:
            T val;
    public:
    static int count;

    Test()
    {
            count++;
    }
};
Template<class T> int Test<T>::count=0;
```

```
main()
{
    Test<int> a;      //Value of count for
                          Test<int> is 1 now
    Test<int> b;      //Value of count for
                          Test<int> is 2 now
    Test<double> c;    //Value of count for
                          Test<double> is 1 now
    cout<<Test<int>::count;  //prints 2

    cout<<Test<double>::count;//prints 1
}
```

End of Unit-3

OBJECT ORIENTED PROGRAMMING

(with C++)

Unit-4

Exception Handling

Unit-IV

Standard C++ classes, using multiple inheritance, persistant objects, streams and files, namespaces, **exception handling**, generic classes, standard template library: Library organization and containers, standard containers, algorithm and Function objects, iterators and allocators, strings, streams, manipulators, user defined manipulators, vectors, valarray, slice, generalized numeric algorithm.

Exception

- 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.
- Note that an exceptional circumstance is not necessarily an error.

Exception Handling

- Exception handling is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program.
- Exception handling mechanism consists of following parts:
  - Find the problem (Hit the exception)
  - Inform about its occurrence (Throw the exception)
  - Receive error information (Catch the exception)
  - Take proper action (Handle the exception)

Exception Handling

C++ consists of 3 keywords for handling the exception:

- **try:** Try block consists of the code that may generate exception. Exception are thrown from inside the try block.
- **throw:** Throw keyword is used to throw an exception encountered inside try block. After the exception is thrown, the control is transferred to catch block.
- **catch:** Catch block catches the exception thrown by throw statement from try block. Then, exception are handled inside catch block.

Exception Handling

Syntax:

```
try
{
        statements;
        … … …
        throw exception;
}
catch (type argument)
{
        statements;
        … … …
}
```

Example:

```
int main ()
{
try
{
        throw 20;
}
catch (int e)
{
        cout << "An exception
        occurred with value  " << e;
}
        return 0;
}
```


Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Exception Handling

Example:

```
int main()
{
   int x = -1;
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }
    cout << "After catch (Will be executed) \n";
   return 0;
}
```

Output:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```


Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Multiple Catch Exception

Multiple catch exception statements are used when a user wants to handle different exceptions differently. For this, a user must include catch statements with different declaration.

Syntax:

```
try {
        body of try block }
 catch (type1 argument1)
 {   statements;   … … … }
 catch (type2 argument2)
 {   statements;   … … … }
 catch (typeN argumentN)
 {   statements;   … … … }
```

Catch All Exceptions

Sometimes, it may not be possible to design a separate catch block for each kind of exception. In such cases, we can use a single catch statement with ellipsis (...) that catches all kinds of exceptions which are not caught by other handlers.

Syntax:

```
catch (...)
{
        statements;
        ... ... ...
}
```


Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Example

Program to divide two numbers using try catch block.

```
int main()
{
    int a,b;
    cout << "Enter 2 numbers: ";
    cin >> a >> b;
    try
    {
    if (b != 0)
    {
    float div = (float)a/b;
    if (div < 0) throw 'e';
//'e' is not implicitly converted to int
            cout << "a/b = " << div;
    }
    else
            throw b;
}
catch (int e)
{
cout << "Exception: Division by zero";
}
catch (char st)
{
cout << "Exception: Division is less than 1";
}
catch(...)
{
cout << "Exception: Unknown";
}
return 0;
}
```


Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Nested try-catch blocks

```
int main()
{
    try {
        try  {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; //Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

Output:
Handle Partially Handle remaining

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Throwing exception from a function

Example:

```
void xTest(int test)
{
    cout<<"Inside xTest: "<<test;
    if(test)
            throw test;
}
main()
{
    cout<<"Main Start";
    try{
            cout<<"Inside try block";
            xTest(0);
            xTest(1);
            xTest(2);
    }
    catch (int i)
    {
            cout<<"Caught exception : ";
            cout<<i;
    }
    cout<<"Main End";
}
```

Output:

```
Main Start
Inside try block
Inside xTest: 0
Inside xTest: 1
Caught Exception : 1
Main End
```


Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Exception Specification

To specify the set of exceptions that might be thrown as part of the function declaration e.g.

```
returnType functionName(Arg_list) throw(type_list)
{
    :
}
```

This specifies functionName may throw only exceptions of type_list and exceptions derived from these types but no others.

To throw character as well as integer from xTest() use:

```
void xTest(int test) throw (int, char)
```

To throw no type exception from xTest() use:

```
Void xTest(int test) throw()
```

Re- Throwing an exception

Example:

```
void f(int a)
{
try
{
if(a==0)
        throw a;
else
        cout<<"Good";
}
catch(int)
{
cout<<"Rethrowing";
throw;
}
}
```

```
main()
{
        try{
        f(0);
        }
        catch(int)
        {
        cout<<"Caught Exception";
        }
}
```

Output:

Rethrowing
Caught Exception

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Exception & Classes

When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

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

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

Output:
Constructor of Test
Destructor of Test
Caught 10

Exception & Classes

- If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class. If we put base class first then the derived class catch block will never be reached. For example, following C++ code prints *"Caught Base Exception"*

```
class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   try
   {
      throw d;
   }
catch(Base b)
{
        cout<<"Caught Base Exception";
}
catch(Derived d)
{
        //This catch block is NEVER executed
        cout<<"Caught Derived Exception";
 }
   return 0;
}
```

Exception & Classes

- In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modified program and it prints *"Caught Derived Exception"*

```
class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   try {
       throw d;
   }
 catch(Derived d)
{
        cout<<"Caught Derived Exception";
   }
catch(Base b)
{
        cout<<"Caught Base Exception";
}
   return 0;
}
```

To be continued…

OBJECT ORIENTED PROGRAMMING

(with C++)

Unit-4

Standard Library, Streams & File

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Unit-IV

Standard C++ classes, using multiple inheritance, persistant objects, **streams and files**, namespaces, exception handling, generic classes, standard template library: Library organization and containers, standard containers, algorithm and Function objects, iterators and allocators, strings, **streams, manipulators, user defined manipulators**, vectors, valarray, slice, generalized numeric algorithm.

Standard Library

A standard library is something that every implementer must supply so that every programmer can rely on it.

The C++ standard library:

- Provides support for language features such as memory management and run time type information
- Supplies information about implementation-defined aspects of the language such as the largest float value
- Supplies functions that can not be implemented optimally in the language itself for every system such as sqrt()
- Supplies facilities that a programmer can rely on for portability such as lists, sort functions & I/o streams
- Provides common foundation for other libraries

Standard Library

The facilities of the standard library are defined in the std namespace and presented as a set of headers.

```
using namespace std;
```

Standard library is a collection of classes and functions which are written in the core language. Header files in the c++ standard library do not end with '.h' extension e.g.

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

Stream I/O Library Header Files

- A stream is a sequence of bytes.
- A stream acts as an interface between program & i/p, o/p device.
- A stream is nothing but a flow of data.
  - ***iostream*** -- contains basic information required for all stream I/O operations
  - ***iomanip*** -- contains information for performing formatted I/O with parameterized stream manipulators
  - ***fstream*** -- contains information for performing file I/O operations
  - ***strstream*** -- contains information for performing in-memory I/O operations (i.e., into or from strings in memory)

Stream Classes for I/O & File Operations

- ios is the base class.
- istream and ostream inherit from ios
- ifstream inherits from istream (and ios)
- ofstream inherits from ostream (and ios)
- iostream inherits from istream and ostream (& ios)
- fstream inherits from ifstream, iostream, and ofstream

C++ Stream I/O -- Stream Manipulators

- Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream.
- Stream manipulators are defined in <iomanip>
- These manipulators provide capabilities for
  - setting field widths,
  - setting precision,
  - setting and unsetting format flags,
  - flushing streams,
  - inserting a "newline" and flushing output stream,
  - skipping whitespace in input stream

C++ Stream I/O -- Stream Manipulators

setprecision (int precision)

- To set floating point precision i.e. number of significant digits to be printed e.g.
  cout << setprecision (2) << 1.2345;
  Output: 1.23 // two significant digits

setw (int width)

- To set the field width (can be used on input or output, but only applies to next insertion or extraction) e.g.
  cout << setw (10) <<12345;
  Output: _ _ _ _ _ 1 2 3 4 5 //5 blanks+5 digits

C++ Stream I/O -- Stream Manipulators

setfill (char fillChar)

- To set the fill character.

  e.g.

  cout << setfill('*') << setw(10)<<12345;

  Output: * * * * * 1 2 3 4 5 //5 stars + 5 digits

endl

- To insert new line and flush stream

  cout<<endl;

C++ Stream I/O - Stream Manipulators

- Various ios *format flags* specify the kinds of formatting to be performed during stream I/O.
- There are member functions (and/or stream manipulators) which control flag settings.
- There are various flags for trailing zeros and decimal points, justification, number base, floating point representation, and so on.

C++ Stream I/O -- Stream Manipulators

setiosflags(long f)

- Sets the format flag e.g.

```
cout<<setiosflags(ios::scientific)<<1.0000;
```

Output: 1.0000e+000

resetiosflags(long f)

- Clears the flag specified by setiosflag e.g.

```
cout<<resetiosflags(ios::scientific)<<1.0000;
```

Output: 1.0000

I/O Stream Member Functions

.setf ( )

- Allows the setting of an I/O stream format flag.

- e.g.

```
// To show the + sign in front of positive numbers
cout.setf (ios::showpos) ;

// To output the number in hexadecimal
cout.setf (ios::hex, ios::basefield) ;
```

I/O Stream Member Functions

.precision ( ) ;

- Select output precision, i.e., number of significant digits to be printed.
- e.g.

```
cout.precision (2) ;    // two significant digits
```

.width ( ) ;

- Specify field width. (Can be used on input or output, but only applies to next insertion or extraction).
- e.g.

```
cout.width (4) ;      // field is four positions wide
```

I/O Stream Member Functions

.eof ( ) ;

- Tests for end-of-file condition.
- e.g.

```
cin.eof ( ) ;    // true if eof encountered
```

.fail ( ) ;

- Tests if a stream operation has failed.
- e.g.

```
cin.fail ( ) ;   // true if a format error occurred
! cin;           // same as above; true if format error
```

I/O Stream Member Functions

.clear ( ) ;

- Normally used to restore a stream's state to "good" so that I/O may proceed or resume on that stream.

- e.g.

```
cin.clear ( ) ;    // allow I/O to resume on a "bad"
                   // stream, in this case "cin",
                   // on which error had previously
                   // occurred
```

Using Manipulators & Member Functions

```
#include <iostream>          // No ".h" (standard header)
#include <iomanip>           // No ".h" (standard header)
using namespace std;         // To avoid "std::"
int main ( )
{
   int a, b, c = 8, d = 4 ;
   float k ;
   char name[30] ;
   cout  <<  "Enter  your name"  <<  endl ;
   cin.getline (name, 30) ;
   cout << "Enter two integers and a float " << endl ;
   cin  >>  a  >>  b  >>  k ;
```

Using Manipulators & Member Functions

```
// Now, let's output the values that were read in
  cout << "\nThank you, " << name <<  ", you entered"
    <<  endl  <<  a  <<   ", "  <<  b  <<  ", and " ;
   cout.width (4) ;
   cout.precision (2) ;
   cout << k << endl ;
//  Control the field and precision another way
   cout <<"\nThank you, " << name << ", you entered"
     << endl  <<  a << ", " <<  b << ", and " << setw (c)
     << setprecision (d) << k << endl ;
}
```

Example Program Output

Enter your name

Dr. Gaurav

Enter two integers and a float

12 24 67.85

Thank you, Dr. Gaurav, you entered

12, 24, and 68

Thank you, Dr. Gaurav, you entered

12, 24, and 67.85

Creating own manipulator

Syntax:

```
ostream & manipulator_name(ostream &out)
{
    :
    return out;
}
```

How to use it:

```
ostream &unit(ostream &out)
{
    out<<inches;
    return out;
}
```

Example:

```
cout<<36<<unit;
```

➔ 36 inches

Own manipulator with multiple operations

Example:

```
ostream &currency(ostream & output)
{
    output<<"Rs";
    return output;
}
ostream &show(ostream &output)
{
    output.setf(ios::showpos);
    output.setf(ios::showpoint);
    output.fill('*');
    output.precision(2);
    output.setiosflags(ios::fixed)<<setw(10);
    return output;
}
main()
{
cout<<currency<<show<<503.2;
}
```

Output:

Rs***+503.20

Files & Streams

- A file is a collection of related data stored in a particular area on the disk.
- C++ views each file as a sequence of bytes.
- Each file ends with an ***end-of-file*** marker.
- When a file is ***opened***, an object is created and a stream is associated with the object.
- To perform file processing in C++, the header files **<iostream.h>** and **<fstream.h>** must be included.
- <fstream.h> includes <ifstream> and <ofstream>

How to open a file?

A file can be opened in 2 ways:

- ***Using Constructor:*** Used when only one file is in the stream. *Syntax:*
  FileStreamClass objectName("fileName", open_Mode)

e.g. ofstream outFile("Try.dat",ios::out);

OR

- ***Using member function – open():*** Used to manage multiple files using one stream. *Syntax:*

  FileStreamClass objectName;
  objectName.open ("fileName", open_Mode)

e.g.

ofstream outFile;
outFile.open("clients.dat", ios:out)

File Open Modes

- **ios:: app** - (append) write all output to the end of file
- **ios:: ate** - data can be written anywhere in the file
- **ios:: binary** - read/write data in binary format
- **ios:: in** - (input) open a file for input i.e. for read only
- **ios:: out** - (output) open a file for output i.e. for write only
- **ios:: trunc** -(truncate) discard the file's content if it exists
- **ios:: nocreate** - if the file does **NOT** exists, the open operation fails
- **ios:noreplace** - if the file exists, the open operation fails

How to close a file in C++?

The file is closed implicitly when a destructor for the corresponding object is called

OR

by using member function *close:*

```
ofstream outFile;
outFile.close();
```

Sequential file (Using Constructor)

```
#include <iostream.h>
#include <fstream.h>
int main()
{
   // ofstream constructor opens file
    ofstream outFile( “File”);
    char name[30]=“Gaurav”;
    int year=2018;
    outFile<<name<<“\n”;
    outFile<<year<<“\n”;
    outFile.close();
```

```
ifstream inFile(“File”);
inFile>>name;
inFile>>year;
cout<<name;
cout<<year;
}
```

Sequential file (Using open())

```
#include <iostream.h>
#include <fstream.h>
int main()
{
   char str[100];
    ofstream outFile;
    outFile.open("Country");
    outFile<<"India\n";
    outFile<<"Nepal\n";
    outFile.close();
    outFile.open("States");
    outFile<<"Delhi";
    outFile<<"Kathmandu";
    outFile.close();
    ifstream inFile;
    inFile.open("Country");
while(inFile)  //(inFile.eof()!=0)
{
    inFile.getline(str,100);
    cout<<str;
}
inFile.close();
inFile.open("States");
while(inFile)  //eof()=0 on
    reaching end of file condition
{
    inFile.getline(str,100);
    cout<<str;
}
inFile.close();
}
```

Reading and printing a sequential file

```
// Reading and printing a sequential file
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
void outputLine( int, const char *, double );
int main()
{
   // ifstream constructor opens the file
   ifstream inFile( "clients.dat", ios::in );
   if ( !inFile ) {
      cerr << "File could not be opened\n";
      exit( 1 );
   }
```

```
   int account;
   char name[ 30 ];
   double balance;
   cout << setiosflags( ios::left ) << setw( 10 ) << "Account“ << setw( 13 ) <<
"Name" << "Balance\n";

   while ( inFile >> account >> name >> balance )
      outputLine( account, name, balance );

   return 0;  // ifstream destructor closes the file
}

void outputLine( int acct, const char *name, double bal )
{
   cout << setiosflags( ios::left ) << setw( 10 ) << acct << setw( 13 ) << name <<
setw( 7 ) << setprecision( 2 ) << resetiosflags( ios::left )
 << setiosflags( ios::fixed | ios::showpoint )
       << bal << '\n';
}
```

File position pointer

- <istream> and <ostream> classes provide member functions for repositioning the *file pointer* (the byte number of the next byte in the file to be read or to be written.)
- These member functions are:

  seekg (seek get) for istream class – Moves get (i/p) pointer to a specified location

  Syntax: **seekg(offset, refPosition);**

Where ***offset*** is no. of bytes, file pointer is to be moved from refPosition.

***refPosition*:**

- ios::beg →start of file
- ios::curr→Current position of pointer
- ios::end → End of file

seekp (seek put) for ostream class – Moves put pointer to a specified location.

seekp(offset, refPosition);

Examples of moving a file pointer

- **inFile.seekg(o)** - repositions the file get pointer to the beginning of the file
- **inFile.seekg(n, ios::beg)** - repositions the file get pointer to the n-th byte of the file
- **inFile.seekg(m, ios::end)** -repositions the file get pointer to the m-th byte from the end of file
- **inFile.seekg(o, ios::end)** - repositions the file get pointer to the end of the file
- The same operations can be performed with <ostream> member function - ***seekp.***

Member functions tellg() and tellp()

- Member functions **tellg** and **tellp** are provided to return the current locations of the get and put pointers, respectively.

  long location = inFile.tellg();

- To move the pointer relative to the current location use:

  ios::cur

- inFile.seekg(n, ios::cur) - moves the file get pointer n bytes forward from current position of pointer.

Sequential file-I/O operations (put() & get())

```
int main()
{
    char str[100];
    cin>>str;
    int len=strlen(str);
    fstream file;
    file.open("Try",ios::in|ios::out);
    for(int i=0;i<len;i++)
            file.put(str[i]);
    file.seekg(0);  //Go to start
    char ch;
    while(file)
    {
        file.get(ch);
        cout<<ch;
     }
}
```

Sequential file - write() & read()

Used to handle data in binary form. Syntax:

- inFile.read((char*)&var,sizeof(var));
- outFile.write((char*)&var,sizeof(var));

```
main()
{
float arr[4]={1.2,2.3,3.4,4.5,5.6};
ofstream outfile;
outfile.open("Try");
outfile.write((char*)&arr,sizeof(arr));
outfile.close();
arr={0,0,0,0};
ifstream infile;
infile.open("Try");
infile.read((char*)&arr,sizeof(arr));
for(i=0;i<4;i++)
cout<<arr[i];
infile.close();
}
```

Sequential file – Reading & Writing Class Object

Data members are written to disk file not the member function

```
class ABC
{
char name[50];
int roll;
public:
   void GetData()   {
   cin>>name>>roll;  }
   void Display()    {
   cout<<name<<roll;
}};

main()
{
   ABC obj[5];
   fstream file;
```

```
file.open("Try.DAT", ios::in|ios::out);
cout<<"Enter Data: ";
for(int i=0;i<5;i++)       {
obj[i].GetData();
file.write((char*)&obj[i],sizeof(obj[i]));
}
file.seekg(0);
cout<<"Output : ";
for(int i=0;i<5;i++)       {
file.read((char*)&obj[i],sizeof(obj[i]));
obj[i].Display();
}
file.close();
}
```

Updating a sequential file

- Data that is formatted and written to a sequential file **cannot be modified easily** without the risk of destroying other data in the file.
- If we want to modify a record of data, the new data may be longer than the old one and it could overwrite parts of the record following it.

Problems with sequential files

- Sequential files are inappropriate for so-called “instant access” applications in which a particular record of information must be located immediately.
- These applications include banking systems, point-of-sale systems, airline reservation systems, (or any data-base system.)

Random access files

- Instant access is possible with random access files.
- Individual records of a **random access file** can be accessed directly (and quickly) without searching many other records.

<ostream> function *write*

- The <ostream> member function *write* outputs a fixed number of bytes beginning at a specific location in memory to the specific stream.
- When the stream is associated with a file, the data is written beginning at the location in the file specified by the "put" file pointer.
- The *write* function expects a first argument of type *const char **, the address of the *object is converted* to a *const char **. The second argument of *write* is an integer of type size_t specifying the number of bytes to written. *Thus* the *sizeof(object)*.

<istream> function *read*

The <istream> function inputs a specified (by sizeof(object)) number of bytes from the current position of the specified stream into an object.

Operations on random access file

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
classs ABC
{
    char name[20];
    int roll;
    public:
            void GetData();
            void Display();
};
main()
{
ABC obj;
fstream inoutfile;
inoutfile.open("Try.DAT", ios::ate|
ios::in|ios::out|ios::binary);
inoutfile.seekg(0,ios::beg);
while(inoutfile.read((char*)&obj,
            sizeof(obj)))      {
    obj.Display();
}
inoutfile.clear();

//Adding Data
obj.GetData();
inoutfile.write((char*)&obj,sizeof(obj));

//Display the appended file
inoutfile.seekg(0);
while(inoutfile.read((char*)&obj,sizeof(
    obj)) {
    obj.Display();
}
```

Operations on random access file

```
//Find no. of objects in the file
int last=inoutfile.tellg(); //current
    position of get pointer which
    will be last position
int n=last/sizeof(obj); //no. of
    objects

//Modify details
cout<<"Enter object no. to be
    updated : ";
int objNo;
int location=(objNo-1) *
    sizeof(obj);
if(inoutfile.eof())
    inoutfile.clear();  //To turnoff
    eof flag
inoutfile.seekp(location);
cout<<"Enter value for object : ";
obj.GetData();

inoutfile.write((char*)&obj,sizeof(obj));
//Display updated file
inoutfile.seekg(0);
while(inoutfile.read((char*)&obj,sizeof(
    obj))  {
    obj.Display();
}
inoutfile.close();
}
```

To be continued…

OBJECT ORIENTED PROGRAMMING

(with C++)

Unit-4

Standard Template Library

Dr. Gaurav Gupta
Associate Professor (IT Deptt.)

Unit-IV

Standard C++ classes, using multiple inheritance, persistant objects, streams and files, namespaces, exception handling, generic classes, **standard template library: Library organization and containers, standard containers, algorithm and Function objects, iterators and allocators,** strings, streams, manipulators, user defined manipulators, **vectors, valarray, slice, generalized numeric algorithm.**

Standard Template Library(STL)

STL is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

- STL provides a collection of classes to make C++ programming more efficient
  - Algorithms, Containers, and Iterators
  - Function objects, Allocators, and Adapters

Components of STL

- **Containers** - Generic class templates for storing collection of data. Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, etc.
- **Algorithms** - Generic function templates for operating on containers. Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
- **Iterators** - Generalized 'smart' pointers that facilitate use of containers. They provide an interface that is needed for STL algorithms to operate on STL containers.

Containers

Three types of containers

- *Sequence containers:*
  - linear data structures such as vectors and linked lists
    - vector (dynamic table)
    - dequeue (double ende queue, using dynamic table)
    - list (double linked one)
    - (arrays and strings not STL containers, but operable by STL algorithms)
- *Associative containers:*
  - non-linear containers such as hash tables
    - set (using binary tree)
    - multiset
    - map, multimap
- *Container adapters:*
  - constrained sequence containers such as stacks and queues

Iterators

- Iterators are pointers to elements of first-class containers
  - Type ***const_iterator*** defines an iterator to a container element that *cannot* be modified
  - Type ***iterator*** defines an iterator to a container element that *can* be modified
- Used as arguments to algorithms
- Containers provide several iterators
  - Begin, End, Reverse Begin, Reverse End
- 5 categories – each has specific operations
  - Input, Output, Forward, Bidirectional, Random

Algorithms

- Operate on containers using iterator interface
  - generic, but not as fast as containers' methods
  - slow on some combinations of algorithm/container
  - may operate on various containers at the same time
- 4 categories
  - Non-modifying
    For each, Find, Count, Equal, Search, etc.
  - Mutating
    Copy, Generate, Reverse, Remove, Swap, etc.
  - Sorting Related
    Sort, Stable sort, Binary search, Merge, etc.
  - Numeric
    Accumulate, Inner product, Partial sum, Adjacent difference

Function Object as Algorithm Argument

- Used by some algorithms
  - Applied to each element in input by For each
  - Used to create new elements by Generate
- Predicates - Function Objects that return bool
  - Unary and Binary predicates used to test equality
  - Find If uses unary, Sort uses binary
- function objects are
  - easily optimizable by compiler
  - may have class member variables,internal state" passed by the constructor argument
  - we may have many function objects of the same class

Vector container

Provides an alternative to the built in array. A vector is self grown. Use It instead of the built in array!

- Vector – similar to an array
  - Vectors allow access to its elements by using an index in the range from 0 to n-1 where n is the size of the vector
- Vector vs array
  - Vector has operations that allow the collection to grow and contract dynamically at the rear of the sequence

Syntax:

vector<of type>

Example :

vector<int> - vector of integers.

vector<string> - vector of strings.

vector<Shape> - vector of Shape objects where Shape is a user defined class.

Vector container

Example :

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{ // create a vector to store
int vector<int> vec;
int i; // display the original size of vec
cout << "vector size = " << vec.size() <<
endl; // push 5 values into the vector
for(i = 0; i < 5; i++) {
vec.push_back(i); } /
/ display extended size of vec
cout << "extended vector size = " <<
vec.size() << endl; // access 5 values
                    from the vector
for(i = 0; i < 5; i++) {
cout << "value of vec [" << i << "] = " <<
vec[i] << endl;
} // use iterator to access the values
vector<int>::iterator v = vec.begin();
while( v != vec.end())
{
cout << "value of v = " << *v << endl;
v++;
}
return 0; }
```

Strings

- In C, char * was used to represent a string.
- The C++ standard library provides a common implementation of a string class abstraction named string.
- To use the string type simply include its header file.

```
#include <string>
string str = “some text”;
```

or

```
string str(“some text”);
```

other ways:

```
string s1(7,‘a’);
string s2 = s1;
```

String Operations

String Length: The length of string is returned by its size() operation.

```
#include <string>
string str = “something”;
cout << “The size of “<< str << “is “ << str.size() << “characters.” << endl;
```

String concatenation: Concatenating one string to another is done by the ‘+’ operator.

```
string str1 = “Here ”;
string str2 = “comes the sun”;
string concat_str = str1 + str2;
```

Valarray slice

A *valarray slice* is defined by a *starting index*, a *size*, and a *stride*.

- The *starting index* (start) is the index of the first element in the selection.
- The *size* (size) is the number of elements in the selection.
- The *stride* (stride) is the span that separates the elements selected.
  Therefore, a slice with a stride higher than 1 does not select contiguous elements in the valarray;
- For example, slice(3,4,5) selects the elements 3, 8, 13 and 18.

Valarray slice

Example:

```
#include <iostream> // std::cout
#include <cstddef> // std::size_t
#include <valarray> // std::valarray, std::slice
using namespace std;
int main () {
    valarray<int> foo (12);
    for (int i=0; i<12; ++i)
        foo[i]=i;
    valarray<int> bar = foo[slice(2,3,4)];
    cout << "slice(2,3,4):";
    for (size_t n=0; n<bar.size(); n++)
        cout << ' ' << bar[n]<<'\n';
    return 0;
}
```

Output:

slice(2,3,4): 2 6 10

Generalized Numeric Algorithm

Some of the Numeric algorithms in STL are:

- ***iota Method:*** assigns all the successive elements in range [first, last] as an incremented value of the element itself. Syntax is:
  - iota(iterator first, iterator last, int value).
- ***accumulate Method:*** performs operation **op** on all the element in the range [first, last] and stores the result into the container result. Syntax is:
  - accumulate(iterator first, iterator last, object_type result, binaryoperator op)
- ***partial_sum Method***: assigns every element in the range starting from iterator **result** of the operation **op** on the successive range in [first, last]. Syntax is:
  - partial_sum(iterator first, iterator last, iterator result, binary_operation op)

iota method

```
#include<iostream>
#include<numeric>
#include<vector>
using namespace std;
int main()
{
vector<int> v(10); /* now vector v is : 0,0,0,0,0,0,0,0,0,0 */
iota(v.begin(), v.end(), 10 ); /* now the vector v is
   10,11,12,13,14,15,16,17,18,19 */
}
```

Accumulate method

```
#include<iostream>
#include<numeric>
#include<vector>
using namespace std;
int myoperator(int a, int b )
{
return a*b;
}
int main() {
    vector<int> v;
    for(int i = 0 ; i < 10; i++) {
            v.push_back(i);
    }
/* now vector v is :
    0,1,2,3,4,5,6,7,8,9 */
    int result;
```

```
accumulate(v.begin(), v.end(), result) ;
/* as no operator is specified,
accumulate add all the elements
between v.begin() and v.end() and store
the sum in result */
/* now result = 45 */

accumulate(v.begin(), v.end(), result,
myoperator) ;
/* applies myoperator on all the
elements in the range v.begin() and
v.end() and store them in result */
/* now result = 9! */
}
```

partial_sum method

```
#include<iostream>
#include<numeric>
#include<vector>
using namespace std;
int myoperator(int a, int b) {
    return a*b;
}
int main() {
    int a[] = {1,2,3,4,5};
    vector<int> v (a,a+5);
    vector<int> v2; /* vector v is 1,2,3,4,5 */
    v2.resize(v.size());
    partial_sum(v.begin(), v.end(), v2.begin());
    /* now v2 is : 1,3,6,10,15 */
    /* sum of the successive range in v.begin() and v.end() */
    partial_sum(v.begin(), v.end(), v2.begin(), myoperator);
    /* now v2 is : 1,2,6,24,120 */
}
```

End of Unit-IV…

Syllabus Finished…