UNIT-III

Polymorphism

- **Definition -**

Polymorphism -
Polymorphism is basically an ability to create variable, function or object that has more than one form.

* Operator Overloading -

→ • It is defined as ability to define a new meaning for an existing (built-in) "operator"

- Various types of operators are:
  +, -, *, /, <, >, &&, ||, =, <<, >>

- We can overload operators by defining a function with keyword operator. Then write operator as a function name

- Only existing operators can be overloaded. and overloaded operators must have at least one operand.

★ Program to concatenate two strings using operator overloading on '+' operator

→
```
#include <iostream>
#include <cstring>
using namespace std;
class string1
{
public:
    char S[15];
    string1()
    {
        strcpy(S, "10");
    }
    string1(char T[15])
    {
        strcpy(S,T);
    }
    string1 operator +(string1 k)
    {
        strcat(S, k.S);
        strcat(S, "10");
        return S;
    }
};
int main()
{
    string1 s1("Hello"), s2("Friends");
    string1 s;
    s = s1 + s2;
    cout << s.S << endl;
    return 0;
}
```

★ Type Casting -

→ • It is a technique in which data of one type gets converted to another data type.

• Eg, conversion of class type to basic data type

```
#include <iostream>
using namespace std;
class test
{
    public:
    operator int();
};
test :: operator int()
{
    int sum;
    sum = 5;
    return sum;
}
int main()
{
    test obj;
    int x;
    x = 10;
    x = int(obj)    // class assigns value to variable
return 0;
}
```

* Function Overloading -

→ • It is a concept in which one can use many functions having same function name, but can pass different no. of parameters

• Eg. program to take two integer / two float nos. and output smallest no.

```
#include <iostream>
using namespace std;
class Test
{
    public:
        int smallest (int, int)          // Function
        float smallest (float, float)    // Overloading Defn
};
int Test :: smallest (int a, int b)
{
    if (a < b)
        return a;
    else
        return b;
}
float Test :: smallest (float x, float y)
{
    if (x < y)
        return x;
    else
        return y;
}
```

- **Definition –**

Runtime Polymorphism –

Runtime Polymorphism is also known as dynamic polymorphism or late binding.
In runtime polymorphism, function call is resolved at run time

★ **Comparison: Compile Time v/s Run Time Polymorphism**

	Compile Time Poly.	Run Time Poly.
1)	Call to functions having same name is made at compile time	1) Call to functions having same name is made at run time
2)	In this, function overloading mechanism is used	2) In this, function overriding mechanism is used
3)	Function overloading & operator overloading techniques are used	3) Virtual functions & pointers are used
4)	It provides fast execution	4) It provides slow execution
5)	It is less flexible	5) It is more flexible

★ Pure Virtual Function –

→ • A pure virtual function is a virtual function which is to be implemented by derived class.

• Class that contains pure virtual function is called abstract class.

• Eg,

```
#include <iostream>
using namespace std;
class A
{
    public:
        virtual void display() = 0;
};

class B : public A
{
    public:
        void display()
        {
            cout << " In derived class B" << endl;
        }
};
int main()
{
    A *p;
    B b;
    p = &b;
    p -> display();
    return 0;
}
```

★ Virtual Destructor –

→ • Virtual destructor is basically a base class destructor preceded by keyword virtual

• Base class is declared as virtual in order to deallocate the memory of desired class explicitly

UNIT - IV

Files And Streams

★ Stream -

→ • Stream is basically, a channel on which data flows from sender to receiver.
• Data can be sent out from program on output stream or received into program on input stream

Stream	Meaning	Device
cin	Standard input	Connected to Keyboard
cout	Standard output	Screen
cerr	Standard error	Screen
clog	Buffer of error	Screen

★ Stream Errors -

Member Function	Purpose
bool ios:: good()	Returns true if there is no error
bool ios:: bad()	Returns true if no r/w operation is performed or invalid r/w operation performed
bool ios:: eof()	Returns true if input operation is reached at end of file
bool ios:: fail()	Returns true if input operation failed to read / output operation failed to generate desired characters

★ Disk Files I/O with Streams –

▲ Open File Operation –

→ • Flags used in modes of file open operation:

ios::in	Open for input operation
ios::out	Open for output operation
ios::binary	Open for binary operation
ios::ate	If flag set, then initial position is set at end of file, else at beginning
ios::app	Output operations are appended to file
ios::trunc	Contents of pre existing file get destroyed and replaced by new one

★ C++ Program to read contents of text file

→
```
#include <iostream>
#include <fstream>
#include <stdlib.h>
using namespace std;
int main()
{
    ifstream in;
        char Data[80];
        in.open("text.dat");
    if(!in)
    {
```

```
        cerr << " Could not open file " << endl;
        exit(1);
    }
    cout << " Contents are : " << endl;
    while (in)
    {
        in.getline (Data, 80);
        cout << " In " << Data;
    }
    in.close();
    return 0;
}
```

★ Overloading Extraction & Insertion operators –

→ • C++ code for overloading insertion and extraction operators is:

```
// insertion function
ofstream &operator << (ostream &st, Point obj)
{
    st << obj.x << ", " << obj.y << "\n";
    return st;
}

// extractor function
istream & operator >> (istream &st, Point &obj)
{
    cout << " Enter x " ;
    st >> obj.x ;
    return st;
}
```

★ Command Line Arguments –

→ • Command line arguments are the arguments that are passed to main function. They are represented as
int main (int argc, char *argv [])

• Here the argc represents total number of arguments and argv is array of characters that store command line arguments.

- C++ program -

```
#include <iostream>
using namespace std;
int main (int argc, char *argv[])
{
    cout<<"\n Total no. of arguments = "<< argc;
    for (int i=0; i<argc; i++)
        cout<<" Argument "<<i<< argv[i];
    return 0;
}
```

UNIT-V

Exception Handling & Templates

* Simple Exception Handling -

→ • When any runtime errors occur in our program then exceptions are raised by handling control to special functions called Handlers.
This provides built-in error handling mechanisms which is known as exception handling

• C++ exception handling uses 3 keywords - try, catch and throw

• 'Try' represents block of statements in which there are chances of occurring exceptional condition

• When exception is detected, it is thrown using 'throw' statement

• The exception thrown is handled appropriately in the 'catch' block.

* Multiple Catching -

→ • When an exception occurs, then control is transferred to catch block, and at that time try block is terminated

• There can be multiple exceptions in multiple catch statements with one try block.

- Following is structure of program when multipl catch blocks are allowed:

```
void function
{
  ...
  try
  {....
  }
  catch(datatype1 arg)
  {....
  }
  catch (datatypeN arg)
  {....
  }
}
```

★ User- Defined Exception -

→ • User Defined Exception is a kind of exception defined by the user

- As most of exception that we need to handle are of class type
- Object of class type is passed to exception handler and with help of object, error is processed

- Example,

Program to compute square root of number. If it is negative, user defined func. mysqrt() should raise a exception:

```
#include <iostream>
# include < math.h>
void mysqrt ( double val)
{
  try
  {
  if (val < 0.0)
        throw " Negative";
   else
        cout <<" Sqrt of "<< val << "is" << sqrt (val) << endl;
  }
   catch (char *str)
   {
      cout << could not handle" << str <<" no." << endl;
   }
}
int main()
{
   cout <<" Enter no.: ";
   double num;
   cin >> num;
   mysqrt (num);
   return 0;
}
```

* Re-throwing an Exception –

→ • When exception is thrown inside try block, it is propagated to catch block.

• But sometimes, handler may decide to rethrow it to propagate next catch statement

• In such a situation, exception can be rethrown by simply writing throw without any argument

• Example,

```
void function()
{
    try
    {
        throw "myworld";
    }
    catch (char *)
    {
        cout << " Inside function()";
        throw;                    // Exception Rethrown
    }
}
int main()
{
    try
    {
        function();
    }
    catch (char *)          // Rethrown exception catch here
```

- Definition -

Generic Programming -
The generic programming is a technique that allows to write code for any data type elements. Template is used as a tool for generic programming

★ Function Template -

→ • For performing identical operations for each data type conveniently, function templates are used

• One can write single function template definition.

• Based on argument types provided in calls to function, compiler automatically instantiates seperate object code functions

• Syntax is -
template < class name_of_datatype >
name_of_datatype function_name (name_of_datatype id1, ...
name_of_datatype id2)

• Program -

```
#include <iostream>
using namespace std;
template < class T >
T min(T a, T b)
{
    if (a<b)
```

```
        return a;
    else
        return b;
}
```

★ Class Template -

→ • Using class template, we can write class whose members use template parameters as types

• Syntax is -

```
template < class Type >
class classname
{
    ....        // body
};
```

• Program -

```
#include <iostream>
using namespace std;
template <class T>
class Compare {
    T a, b;
public:
    Compare(T f, T s)
    {
        a = f;
        b = s;
    }
```

Standard Template Library

- **Definition –**

STL –

The standard template library (STL) is a collection of well structured generic C++ classes (templates) and functions.

STL consists 3 basic components:

1) Container 2) Algorithms 3) Iterators

→ # Container is collection of objects of different types.

There are 2 types of containers –

i) Sequence container

ii) Associative container

* Vectors (Sequence Container)

→ • Vector stores the element in contagious memory locations.

• Program illustrating vector functions –

```
int main()
{
    vector <char> v(10);    // creates a vector
    int i;
```

```
cout<<" Size = " << v.size() << endl;   // display size

cout << " Inserting elements ... " << endl;
for(i=0; i<5; i++)
    v.push_back(i+10+'A');

cout << " Deleting elements ... " << endl;
for (i=0; i<5; i++)
    v.pop_back();
```

- Explaination -

The function v.size() returns size of vector. Then we inserted same characters in vector by v.push_back() function.
Using v.pop_back() we have deleted last five elements and retained original vector

★ Deque –

- Deque is data structure in which element can be inserted / deleted from both, front and rear end.

- Program illustrating operations on Deque -

```
int main()
{
    int item;
    deque<int> dq;
    deque<int>:: iterator i;

    cout << " Enter element to insert: ";
```

```
cin >> item;
dq.push_back(item);

cout << "Enter element to insert from front";
cin >> item;
dq.push_front(item);

item = dq.front();
dq.pop_front();
cout << "Deleted element" << item;

cout << "Elements of DEQUE are:";
for(i = dq.begin(); i != dq.end(); i++)
    cout << "i << " ";
}
```

★ Map (Associative Container)

→ • Map is a associative container in which the elements are stored in form of key value and mapped value

• Program to implement Map using STL -

```
#include<iostream>
#include<map>
using namespace std;
int main()
{
    int count, key;
    map<int, char> m;
    char item;
    map<int, char>::iterator i;
```

```
cout << "Enter element : ";
cin >> key;
cout << " Enter value : ";
cin >> item;
m.insert ( pair< int, char > (key, item));

cout << " Enter key to be deleted : ";
cin >> key;
m.erase (key);

count = m.size();
cout << " Size is : " << count;

cout << " Enter key for searching ";
cin >> key;
if (m.count (key) != 0)
    cout << "Element " << m.find (key) -> second << " is present"
else
    cout << " Element absent " << endl;

cout << "elements of map are : ";
for ( i = m.begin(); i != m.end(); i++)
    cout << " [ " << (*i).first << ", " << (*i).second << "] ";
}
```

★ Stack - (Derived Container)

→ • Stack is a LIFO data structure. That means elements inserted at last get popped off first.

• Basic operations that can be implemented on stack are:
- pop :
- push

• Implementation of Stack using STL -

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
  stack <int> s;
   int item;
   char ans;
   int choice;
   do
   {
      cout << " 1. Push ";
      cout << " 2. Pop ";
      cout <<" 3. Display ";
      cout <<"enter choice : ";
      cin >> choice;
      switch (choice)
      {
         case 1 : cout << " Enter push element : ";
                  cin >> item;
```

```
            s.push(item)
            break;
        case 2 : if(!s.empty())
            {
              s.pop();
            }
            else {
              cout << "stack empty";
            }
            break;
        case 3 : if(!s.empty())
            {
              cout << "Top elements :" << s.top();
            }
            else {
              cout << "stack empty";
            }
            break;

    }
    }
    return 0;
}
```

★ Algorithms -

→ • Algorithms are used to process contents of container

• Functionalities in container are not sufficient to perform complex operations, hence algorithms are used to support complex operations

• Various categories of algorithm are -

▲ Sorting Algorithm - Related to sorting of list

▲ Mutating sequence Algorithm - Modify contents of container

▲ Nonmutating sequence Algorithm - Do not modify contents of container as they work

▲ Numerical Algorithm - Useful for performing computations

* Iterators –

→ • Iterators are basically objects but sometimes they can be pointers & hence they specify positions in container

• Iterators are used to traverse the contents of container

There are 5 types of iterators:

i) Random Access –
Elements can be stored or retrieved randomly

ii) Forward –
Elements can be stored or retrieved but only forward moving is allowed

iii) Bidirectional –
Store and retrive elements and forward/backward moving is allowed

iv) Input –
Element retrieving is allowed with forward moving

v) Output –
Elemint storing is allowed with forward moving.

	Sequential Container	Associative Cont	Derived Container
1)	These are ordered collection in which each element has a position	1) These are sorted collection in which position depends on its value	1) These are un-ordered collection in which position doesn't matter
2)	Position depends on time and place of insertion	2) Value of element determine position Order of insertion doesn't matter	2) Neither does order of insertion nor value of element, only existence matters
3)	Examples - Vectors, Deque, Array, etc	3) Examples - set, multiset, map, etc	3) Example - Stack, Queue, etc