* Polymorphism in c++

* What is polymorphism in c++?

- In c++, polymorphism causes a member fun to behave differently based on the object that calls / invokes it.
- Polymorphism is a Greek word that means to have many forms.
- It occurs when you have a hierarchy of classes related through inheritance.

—Example:

1) The "+" operator in c++ can perform two specific functions at two different scenarios i.e. when the "+" operator is used in numbers, it performs addition.

```
int a=6;
int b=6;
int sum = a+b;
```

Output : sum=12

And the same "+" operator is used in the string, it performs concatenation.

```
string firstName = "ShreeSoft";
string lastName = "Informatics";

string name = firstName + lastName
```

output : name = "ShreeSoft Informatics"

* Types of polymorphism:
C++ supports two types of polymorphism:
1) Compile-time polymorphism
2) Runtime polymorphism

1) Compile-time polymorphism:

- This type of polymorphism is also referred to as static binding on early binding.
- It takes place during compilation.
- We use fun overloading & operator overloading to achieve compile-time polymorphism.

i) Function overloading:
- Fun overloading means one fun can perform many tasks.
- In c++, a single fun is used to perform many tasks with the same name & different types of arguments.
- In the fun overloading fun will call at the time of program compilation. It is an example of compile-time polymorphism.
- Redability of the program increases by fun overloading. It is acheived by using the same name for the same action.

```
#include<iostream>
using namespace std;
int main() class Addition
{
    public:
        int sum(int x, int y)
        {
            return x+y;
        }
        int sum(int x, int y, int z)
        {
            return x+y+z;
        }
        double sum(double x, double y)
        {
            return x+y;
        }
        double sum(int x, double y)
        {
            return x+y;
        }
};
int main()
{
    Additin add;
    cout<<endl<<"sum of 2 integer:"<<add.sum(7, 13);
    cout<<endl<<"sum of 3 integer:"<<add.sum(7, 13, 23);
    cout<<endl<<"sum of 2 float:"<<add.sum(7.7, 13.13);
    cout<<endl<<"sum of 1 int and 1 float:"<<add.sum(7, 13.13)
```

Output → Sum of 2 integer : 20
sum of 3 integer : 43
sum of 2 float : 20.83 sum of 1 int and 1 float : 20.13

ii) Operator overloading:

- Operator overloading mean defining additional tasks to operators without changing its actual meaning.
- We do this by using operator fun.
- The purpose of operator overloading is to provide a special meaning to the user-defined data types.
- The advantage of operator overloading is to perform different operations on the same operand.

```
#include<iostream>
using namespace std;
class Base
{
   private:
            string s;
   public:
        // Base(){}  // Default
         Base(string str)
         {
           s=str;
         }
   void operator+(Base a)
   {
      string m = s+a.s;
      cout<<endl<<"Concatenated String:"<<m;
   }
};
int main()
{
   Base a1("Welcome");
```

```
    Base a2("ShreeSoft Informatics");
    a1+a2;
}
```

Output:

Concatenated string: Welcome ShreeSoft Informatics

(2) Run time polymorphism:

- In a run-time polymorphism, functions are called at the time of program execution. Hence, it is known as late binding or dynamic binding.
- function overriding is a part of runtime polymorphism. In fun overriding, more than one method has the same name with different types of the parameter list.
- It is achieved by using virtual functions & pointers. It provides slow execution as it is known at the run time. Thus, it is more flexible as all the things executed at the run time.

i) function overloading:

- In fun overriding, we give the new defn to base class fun in derived class. At that time, we can say the base fun has been overridden. It can be only possible in the 'derived class'.
- In fun overriding, we have two definitions of the same fun, one in the super class & one in the derived class.
- The decision about which fun definition requires calling happens at runtime. That is the reason we call it 'Runtime polymorphism'.

```
#include<iostream>
using namespace std;
class A
{
    public:
        void disp()
        {
         cout<<endl<<"Super class function";
        }
};
class B : public A
{
    public:
        void disp()
        {
          cout<<endl<<"Sub class function";
        }
};
int main()
{
   B obj;
  obj.disp();
  A obj1;
  obj1.disp();
  return 0;
}
```

Output →

```
Sub class function
Super class function
```

ii) Virtual function:

- A virtual fun is declared by keyword virtual.
- The return type of virtual fun may be int, float, void.
- A virtual fun is a member fun in the base class. We can redefine it in a derived class. It is part of run time polymorphism.
- The declaration of virtual fun must be in the base class by using the keyword virtual.
- A virtual fun is not static.
- The virtual fun helps to tell the compiler to perform dynamic binding or late binding on the fun.
- If it is necessary to use a single pointer to refer to all the different classes objects. This is because we will have to create a pointer to the base class that refers to all the derived objects.
- But when the base class pointer contains the derived class address, the object always executes the base class fun. For resolving this problem, we use the virtual fun.
- When we declare a virtual fun, the compiler determines which fun to invoke at runtime.

```
#include<iostream>
using namespace std;
class Add
{
    int x=5, y=20;
    public:
        void display()  //overridden fun
        {
            cout<<"Value of x is:"<<x+y<<endl;
        }
```

```
class Substract:public Add
{
   int y=10, z=30;
   public:
        void display()   //overridden fun
        {
          cout<<"Value of y is:"<<y-z<<endl;
        }
};
int main()
{
    Add *m        //base class pointer. it can only access base class member
    Substract s;  // making object of derived class.
    m=&s;
    m->display();  //Accessing fun by using base class pointer
    return 0;
}
```

* with use of virtual fun

```
#include<iostream>
using namespace std;
class Add
{
    public:
          virtual void print()
          {
            int a=20, b=30;
            cout<<" base class Action is:"<<a+b<<endl;
          }
          void show()
          {
```

```
        cout<<"show base class"<<endl;
        }
};
class Sub: public Add
{
public:
        void print() //print() is already virtual fun in
                        derived class, we could also declared
                        as virtual void print() explicitly
        {
            int x=20, y=10;
            cout<<"derived class Action:"<<x-y<<endl;
        }
          void show()
          {
            cout<<"show derived class"<<endl;
          }
};
int main()
{
  Add *aptr;
  Sub s;
  aptr=&s;
  //virtual fun, binded at runtime (runtime polymorphism)
  aptr->print();

  //Non-virtual fun, binded at compile time
  aptr->show();
  return 0;
}
```

* File Handling in C++

* Introduction:

Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file & to perform various operations on it.

A stream is an abstraction that represent a device on which operations of I/P & O/P are performed. A stream can be represented as a source of destination of characters of indefinite length depending on its usage.

In C++, we have a set of file handling methods. These include ifstream, ofstream & fstream. These classes are derived from fstreambase & from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream & therefore we must include fstream & therefore we must include this file in any program that uses files.

* The fstream library:

- The fstream library provides C++ programs with three classes for working with files. These classes include:

- fstream: This class generally represents a file stream. It comes with ofstream/ifstream capabilities. This mean it's capable of creating files, writing to files, reading from data files.

- ifstream: This class represents an I/P stream. It is used for reading inform° from the data files.

• ofstream: This class represents an o/p stream. It is used for creating files & writing informn to files.

All the above three classes are derived from fstreambase & from the corresponding iostream class & they are designed specifically to manage disk files.

* Operations in file Handling:

- Creating file
- Reading data from file
- Writing new data into file
- Closing a file

* Creating / opening a file:

We can open a file using any one of the following methods:

1) Opening file using constructor
2) Opening file using the open() fun.

① Opening file using constructor:

We know that a constructor of class initializes an object of its class when it (the object) is being created. Same way, the constructors of stream classes (ifstream, ofstream or fstream) are used to initialize file stream objects with the filenames passed to them.

— Example:

- ofstream fout("myfile.txt");→ This creates object fout of ofstream class for output only.
- ifstream fin("myfile.txt");→ This creates object fin of ifstream class for input only.
- fstream finout("myfile.txt");→ This creates object finout of fstream class input foutput.

* File State Slags:

There are some member fun's that are used to check the state of the file. All these functions return a Boolean value.

Function	Description
eof()	Returns true if the end of file is reached while reading the file.
fail()	Returns true when read/write operation fails or format error occurs.
bad()	Returns true if reading from or writing to a file fail.
good()	Returns false in the same cases in which calling any of the above functions would return true.

* General function used for file handling:

1) open(): To create file
2) close(): To close an existing file.
3) get(): to read a single character from the file.
4) put(): to write a single character in the file
5) read(): to read data from a file.
6) write(): to write data into a file.

* Closing a file in C++

Closing a file is a good practice, & it is must to close the file. Whenever the C++ program comes to an end, it clears the allocated memory, & it closes the file. We can perform the task with the help of close() Fun.

- Syntax:

```
FileName.close();
```

* Write a program to create a file & read text from file.

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
  char c, fname[10];
  cout<<" Enter file name: ";
  cin>> fname;
  ifstream file(fname);
  if(!file)
  {
    cout<<"Error! file Does not Exist";
  }
  else
  {
    while(file.eof()==0)
    {
      file.get(c);
      cout<<c;
    }
  }
}
```

* Write a program to create a file & write text in file.

```
#Include <iostream>
#Include< fstream>
using namespace std;
int main ()
{
  char ch, fname[10];}
  Cout<<"Enter file name:";
  cin>>fname;
  ofstream out (fname);
  if (!out)
  {
   cout<<"Error ! file Does not Exist";
  }
  else
  {
   cout<<"Enter Text:";
   cin>>ch;
   while(ch!='$')
   {
    out<<ch;
    cin>>ch;
   }
  }
}
```

8) ios:: noreplace → The file opens only if it doesn't exist.

* ifstream class contains following functions:

1) get() → use to read a single character.
2) getline() → use to read a single line.
3) read() → use to read single record.
4) seekg() → Moves get pointer to a specified location.
5) tellg() → Tell the current position of get pointer.

* ofstream class contains following functions:

1) put() → use to write a single character.
2) write() → use to write a single record.
3) seekp() → Moves put pointer to a specified location.
4) tellp() → Tell the current position of put pointer.

* Seek Function	Action
• fout.seekg(0, ios::beg) or fout.seekg(0)	Go to beginning of the file.
• fout.seekg(m, ios::cur)	Move m+1st byte in the file.
• fout.seekg(0, ios::end)	Go to end of file.

1024 byte = 1 kB
1024 kB = 1 MB
1024 MB = 1 GB
1024 GB = 1 TB

Write a program to count the no. of vowels in a file.

Write a program to find size of file.

Write a program to copy the content of one file into the another file.

* Template

* Introduction:

- Templates are one of the most powerful feature in c
- Templates provide us the code that is independent of the data type. In other words, using template we can write generic code that works on any data type; we just need to pass the data type as a parameter. This parameter which passes the data-type is also called a typename.
- Templates are ~~rarely~~ used mainly to ensure code resuability & flexibility of the program.
- We can just create simple fun or a class that takes the data type as a parameter & implement the code that works for any data type.

* Types of Template:-

1) Function Template
2) Class Template

(1) Function Template:-

- Fun template is just like a normal fun but the only difference is while normal fun can work only on one data type & a fun template code can ~~even~~ work on multiple data types.
- While they can actually overload a normal fun to work on various data type, fun template are always more useful, as ~~taking~~ we have to write the only program & it can work on all data types.

- Syntax:

```
template<class T>
T function_name (T argument)
{
    ======   // } Function body
}
```

```
#include<iostream>
using namespace std;
template <class T>
T sum (T x, T y)
{
    return x+y;
}
int main()
{
    int a,b;
    float p,q;
    cout<< endl<<" Enter integer numbers";
    cin>>a>>b;
    cout<<"sum:"<<sum(a,b);
    cout<<endl<<" Enter float numbers:";
    cin>>p>>q;
    cout<<"sum:"<<sum(p,q);
}
```

(2) Class Template:-

- Like in fun template, we might have a requirement to have a class that is similar to all other aspects but only different data types.
- In this situation, we can have different classes for different data type or different implementation for different data type in same class. But doing this, will make our code bulky.
- The best soln for this is to use a template class.
- Template class also behaves similar to fun template. we need to pass data type as a parameter ~~as~~ to the class while creating object or calling member fun.

```
#include<iostream>
using namespace std;
template <class T>
class MaxMin{
{
   private:
         a,b;
   public:
         MaxMin (T x, T y)
        {
          a=x;
          b=y;
        }
        T getMax()
        {
          return a>b ?a:b;
        }
};
```

```
int main()
{
  MaxMin <int> mx(7,13);
  cout<<endl<<"Maximum:"<<mx.getMax();
  MaxMin <char> mn ('M','m');
  cout<<endl<<"Maximum:"<<mn.getMax();
}
```