Virtual function

A virtual function is a member function that is declared within a base class and is meant to be overridden in derived classes.

Virtual functions allow you to override a function in a derived class and call the correct function using a base class reference or pointer.

Virtual functions are essential for implementing runtime polymorphism.

You can declare virtual function by preceding the keyword 'virtual' with the function of the base class.

Declaration:

```
Class Base
{
 public:
   virtual void show()
   {
     cout<< "Base class function";
   }
};
// overriding virtual function in derived class
class Derived: public Base
{ public:
    void show() override { //'override' is optional
                           but good practice
      cout<< "Derived class function";
};
```

- Runtime polymorphism: virtual functions enable runtime polymorphism. when a base class pointer (or reference) points to a derived class object, the derived class's version of the function is called.

```
Base *b;
Derived d;
b = &d;
b->show(); // calls Derived class's show function but not Base class's function.
```

Program Example of virtual function

```
#include <iostream>
using namespace std;
class Animal {
public:
virtual void sound()
{ cout<<" Animal sounds ";
}};
class Dog : public Animal {
public:
void sound() override
{ cout<<"woof! woof!";
};
class cat : public Animal {
public:
void sound() override {
}} cout<<" meow!! meow!! "}
```

```
int main() {
Animal * ptr;
Dog dog;
Cat cat;
ptr = &dog;
~~ptr = &cat;~~
ptr->sound(); // calls Dog's sound()
ptr = &cat;
ptr->sound(); // calls cat's sound()
return 0;
}
```

Output: woof! woof! meow!! meow!!

```
int main() {
Animal * ref1 = new Dog();
ref1->sound();
Animal * ref2 = new cat();
ref2->sound();
return 0;
}
```

Need of Virtual function

Flexibility in Code Design:

Virtual functions allows you to write more general and reuseable code. You can write functions that operate on base class referenle or pointers and still achieve specific behavior based on the actual derived class objects.

Polymorphism:-

Virtual function is implementing polymorphisms at runtime.

Virtual function allowing different Clowes to define different behaviors for the same function.

Extensibility:-

Virtual functions make it easy to extend a system. You can add new derived Clases that override the Virtual function without changing existing code that uses the base class pointers or referenle.

Dynamic Binding:-

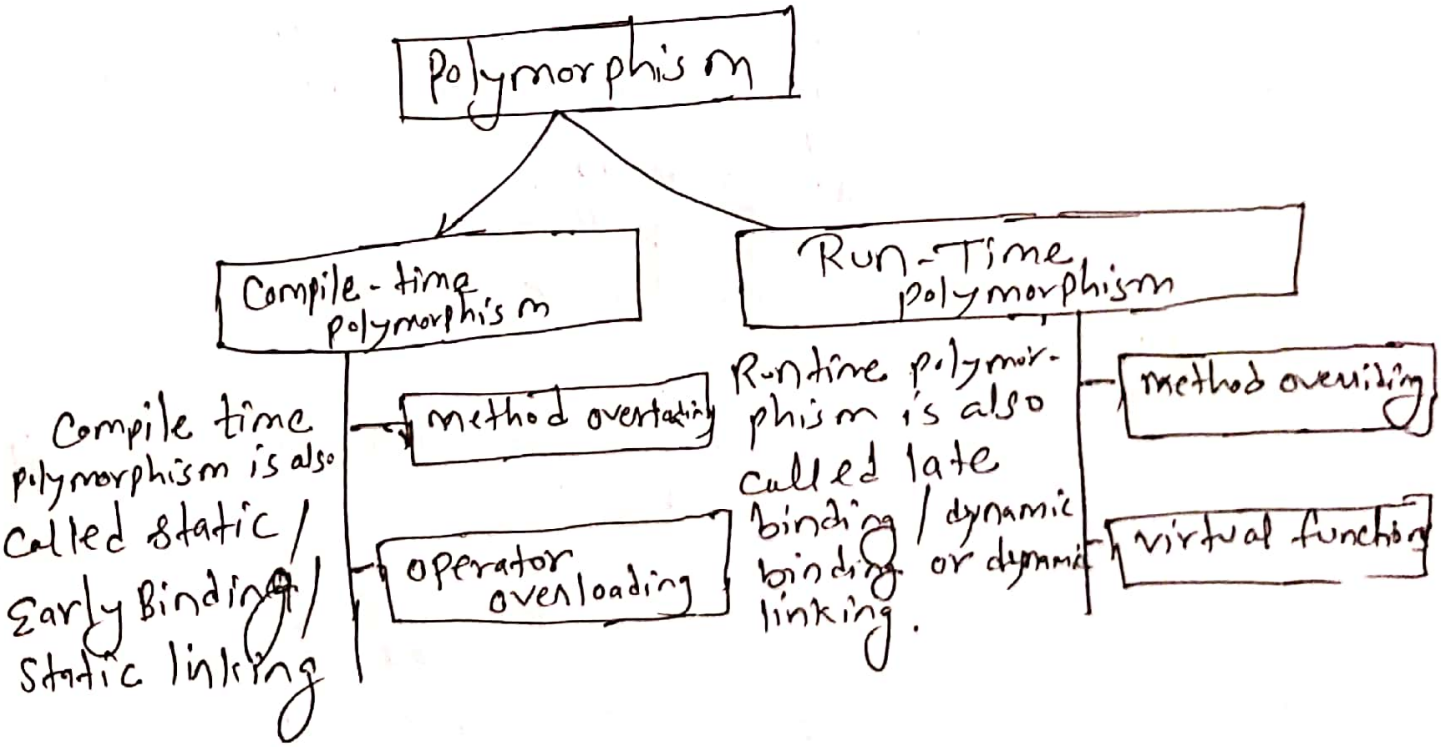
Virtual functions ensure that the method is called that belongs to the actual object type at runtime. rather than the type of the pointer or referenle used to call the method.

Concept of polymorphism

Polymorphism is the feature of object oriented programming (OOP). The word "polymorphism" comes from the Greek words "poly" means many and "morph" means form, meaning "many forms".

The different ways of using same function or operator depending on what they are operating on is called polymorphism.

Example:- A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.



Polymorphism example

Method overloading Program using class

```
#include <iostream>
class calculator
{
public:
 int add(int a, int b)
 {
  return a+b;
 }
 int add (int a, int b, int c)
 {
  return a+b+c;
 }
};
 int main ()
 {
  Calculator calC;
  int sum1 = calC.add (10, 20);
  int sum2 = calc.add (5, 10, 15);
  cout<<" sum of a & b is: "<< sum1;
  Cout<<" sum of a, b, & C is : "<<sum2;
  return 0;
 }
```

output:

sum of a & b is: 30
sum of a, b & C is: 30

Example of polymorphism by function overloading

Method overloading → same function name with different parameters

```
#include <iostream>
using namespace std;
int sum(int a, int b)
{
    return a+b;
}
int sum(int a, int b, int c)
{
    return a+b+c;
}
int main()
{
    int result1 = sum(5, 10);
    int result2 = sum(5, 10, 15);
    cout<<"sum of a & b is: "<<result1;
    cout<<"sum of a, b & c is: "<<result2;
    return 0;
}
```

output:

```
sum of a & b is: 15
sum of a, b & c is: 30
```

Pointers and objects:

Pointer can also be used to point to objects of a class. pointer enables the implementation of polymorphism

Example :-

Base* b = new Derived(); where b is a pointer to a Base class, but it points to an object of the Derived class.

Dynamic polymorphism is achieved using pointers to base class objects and virtual functions. pointers enabling runtime determination of the appropriate function to call. This is a powerful feature of c++ that supports flexible and reusable code.

Example:

```
#include <iostream>
using namespace std;
class Animal
{ public:
virtual void eat()
{ cout<< "Animal eat";
}
virtual ~Animal(){ }
};
class Dog: public Animal
{ public:
  void eat()
  { cout<< "Bone eating";
  }
};
class Cat: public Animal{
 public:
  void eat(){
cout<<"Eating milk";
};
```

```
int main()
{
 Animal * ptr = new Dog();
 Animal *ptr2 = new Cat();
  ptr -> eat();
  ptr2 -> eat();
return 0;
}
```

output: Bone eating
Eating milk.

Pointer

A pointer is a variable that stores the memory address of another variable. pointers provide direct access to memory and are used in various operations such as dynamic memory allocation, array manipulation and function pointers.

Syntax of pointer:

```
int var;
int* varptr = &var; // holding the address of another variable.
```

Dereferencing a pointer:-

Using '*' operator, you can access the value stored at the memory address pointed to by 'ptr'.

```
*ptr_variable
```

Example of pointer

```
#include <iostream>
using namespace std;
int main()
{
    int var = 5;
    int* ptr = &var;
    cout << var;
    cout << "address of variable is: " << &var;
    cout << "pointer variable: " << ptr;
    cout << "content of pointer variable: " << *ptr;
    return 0;
}
```

output:

```
5
address of variable is: 0x61ff08
pointer variable: 0x61ff08
content of pointer variable: 5
```

5.3 Definition of virtual function

A virtual function is a member function in a base class that you expect to be redefined (overridden) in derived classes. When a function is declared as 'virtual' in a base class, it allows the derived classes to provide specific implementations of the function. The key feature of a virtual function is that it supports runtime polymorphism

virtual function Declaration

A function in the base class is made virtual by preceding its declaration with the 'virtual' keyword.

```
class Base {
public:
  virtual void show()
  {
    cout<<"Base class show function";
  }
};
```

overridden in Derived classes

Derived classes can override the virtual function to provide a specific implementation.

```
class Derived : public Base{
public:
  void show() override{
    cout<<"Derived class show function";
  }
};
```

supports Runtime Polymorphism:

When a base class pointer or reference points to a derived class object, the derived class's version of the virtual function is called, even though the function call is made

5.3 Definition of virtual function

A virtual function is a member function in a base class that you expect to be redefined (overridden) in derived classes. When a function is declared as 'virtual' in a base class, it allows the derived classes to provide specific implementations of the function. The key feature of a virtual function is that it supports runtime polymorphism

virtual function Declaration

A function in the base class is made virtual by preceding its declaration with the 'virtual' keyword.

```
class Base {
public:
  virtual void show()
  { cout<<"Base class show function";
};
```

overridden in Derived classes

Derived classes can override the virtual function to provide a specific implementation.

```
class Derived : public Base{
public:
  void show() override{
    cout<<"Derived class show function";
};
```

supports Runtime Polymorphism:

When a base class pointer or reference points to a derived class object, the derived class's version

5.4 pure virtual function

A pure virtual function is a virtual function that does not have an implementation in the base class and must be overridden by derived classes. It is declared by assigning '0' to the function declaration in the base class. A class containing at least one pure virtual function is known as an abstract class and cannot be instantiated.

Syntax:

```
class Base
{
public:
  virtual void functionName() = 0; //pure virtual function
};
```

Program Example:

```
#include <iostream>
  using namespace std;
class Animal
{
public:
  virtual void makesound() = 0;
  //virtual destructor
  virtual ~Animal() {}
};
class Dog : public Animal
{
public:
  void makesound() override
  {cout<<"woof";}
};
```

```
class Cat : public Animal
{ public:
  void makesound ( ) override
  { cout<< "meow";
  }
};
  int main ( )
  {
    Animal * animalptr;
     Dog dog;
     Cat cat;
     animalptr = &dog;
     animalptr -> makesound ( );
     animal ptr = &Cat;
     animalptr -> makesound ( );
     return 0;
  }
```

output: woof
meow

"When a class contains only pure virtual functions, it acts as an interface. All the pure virtual methods are implemented by other derived classes."

5.5 Abstract Class

An abstract class is a class that cannot be instantiated on its own and is designed to be inherited by other classes. It usually contains one or more pure virtual functions. The primary purpose of an abstract class is to provide a common interface for all of its derived classes.

↳ Cannot be instantiated :- you cannot create objects of an abstract class.

↳ Contains pure virtual functions :- At least one function is declared as pure virtual (i.e. = 0).

↳ Provides a common interface : Derived classes must implement the pure virtual functions.

Syntax :

```
Class Abstractclass_Name
{public:
   virtual void purevirtualfunction() =0;
   // other member functions and variables
};
```

Program Example

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

```
virtual ~Shape() { }
};
class Circle : public Shape
{
public:
    void draw()
    { cout<< "Drawing a circle";
};
class Rectangle : public Shape
{
public:
    void draw()
    {
        cout<< "Drawing a rectangle";
};
int main()
{
    Shape* ptr1 = new Circle();
    Shape* ptr2 = new Rectangle();
    ptr1->draw();
    ptr2->draw();
    delete ptr1;
    delete ptr2;
    return 0;
}
```

output:

```
Drawing a circle
Drawing a rectangle
```