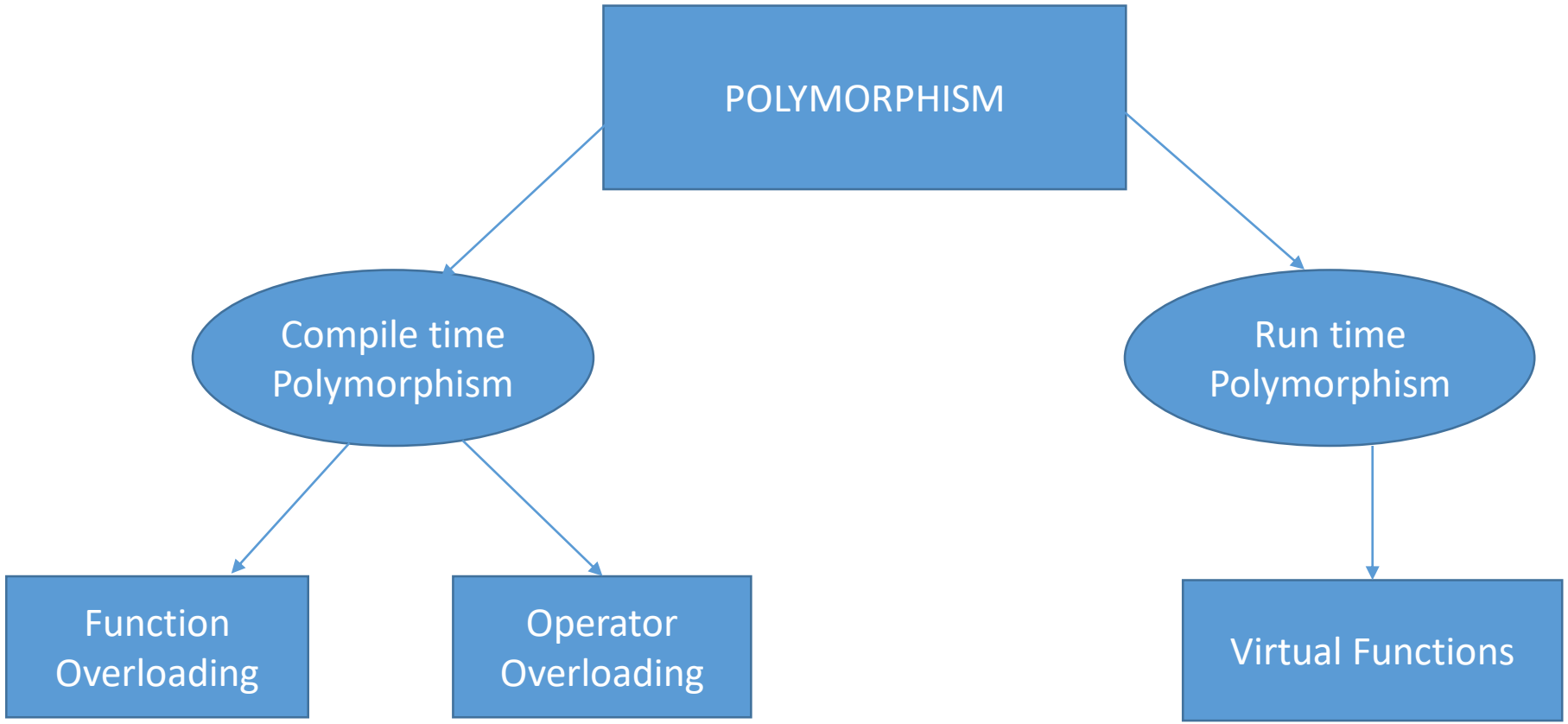
ABSTRACT CLASSES

- An abstract class is a class that is not used to create objects.
- It is designed only to act as a base class.
- It is a design concept in program development and provides a base upon which other classes may be built.

Polymorphism



Compile time Polymorphism

- The concept of polymorphism is implemented using the overloaded functions and operators.
- The overloaded member functions are selected for invoking by matching arguments, both type and member.
- This information is known to the compiler during compile time and the compiler is able to select the appropriate function for a particular call at the compile time itself.
- This is called early binding/static binding/static linking and this procedure is referred to as **Compile time Polymorphism**. (an object is bound to its function call at its compile time)

Run time Polymorphism

- If the appropriate member function could be selected while the program is running, then it is known as run time polymorphism.
- In C++, run time Polymorphism is achieved by virtual functions.
- Since the function is linked with a particular class much later after the compilation, this process is termed as **late binding**.
- It is also known as **dynamic binding** because the selection of the appropriate function is done dynamically at run time.

Virtual Functions

- When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword **virtual** preceding its normal declaration.

- When a function is made **virtual**, C++ determines which function to use at run time based on the type of object pointed to by the base pointer rather than the type of the pointer.

- By making the base pointer to point to different objects, we can execute different versions of the virtual function.

Example Program

```
#include<iostream>
using namespace std;
class Base
{
 public:
        void display()
        {
        cout<<“\n Display base”;
        }
        virtual void show()
        {
        cout<<“\n Show base”;
        }
};
```

```
class Derived : public Base
{
        public:
                void display()
                    {
                            cout<<"\n Display derived";
                     }
                    void show()
                    {
                            cout<<"\n Show derived";
                    }
};
```

```
int main()
{
 Base B;
 Derived D;
 Base *bptr;
 cout<<"\n bptr points to base\n";
 bptr = &B;
 bptr -> display();
 bptr -> show();
 cout<<"\n\n bptr points to Derived\n";
 bptr = &D;
 bptr -> display();
 bptr -> show();
 return 0;
}
```

OUTPUT

bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived

- When **bptr** is made to point to the object **D**, the statement

  bptr-> display();

  calls only the function associated with the **Base** (that is the **display()** of Base), whereas the statement

  bptr-> show();

  calls the **Derived** version of **show().**

  This is because the function **display()** has not been made **virtual** in the **Base** class.

- We must access the virtual functions through the use of a base class pointer only.

Rules for Virtual Functions

- The virtual functions must be members of some class.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- Virtual function in a base class must be defined, even though it may not be used.
- The prototypes of the base class of a virtual function and all the derived class must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
- We cannot have virtual constructors but we can have virtual destructors.

- While a base pointer can point to any type of the derived object, the reverse is not true. That means, we cannot use a pointer to a derived class to access an object of the base type.
- When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will revoke the base function.

Pure virtual function

- A virtual function, equated to zero is called a pure virtual function.

  virtual void display() = 0;

- It is a function declared in the base class that has no definition relative to the base class.
- A class containing such pure virtual functions cannot be used to declare any objects of its own. Such classes are called abstract classes.
- The main aim of pure virtual functions is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Virtual destructor in C++

- A virtual destructor is used to free up the memory space allocated by the derived class object while deleting instances of the derived class using a base class pointer object.
- They maintain the hierarchy of calling destructors from derived class to base class as the virtual keyword used in the destructor follows the concept of **late binding**.
- The destructor of the base class uses a **virtual** keyword before its name and makes itself a virtual destructor to ensure that the destructor of both the base class and derived class should be called at the run time.
- The derived class's destructor is called first, and then the base class or base class releases the memory occupied by both destructors.

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

class Base {
    public: Base()
            {
                cout << "Constructor of Base class is called" << endl;
            }
   virtual ~Base()
            {
               cout << "Destructor of Base class is called" << endl;
            }
};
```

```
class Child: public Base {
    public: Child()
            {
            cout << "Constructor of Child class is called" << endl;
            }

        ~Child()
             {
              cout << "Destructor of Child class is called" << endl;
            }
};
```

```
int main()
 {
    Base * b = new Child;
    delete b;
    return 0;
 }
```

OUTPUT

```
Constructor of Base class is called
Constructor of Child class is called
Destructor of Child class is called
Destructor of Base class is called
```