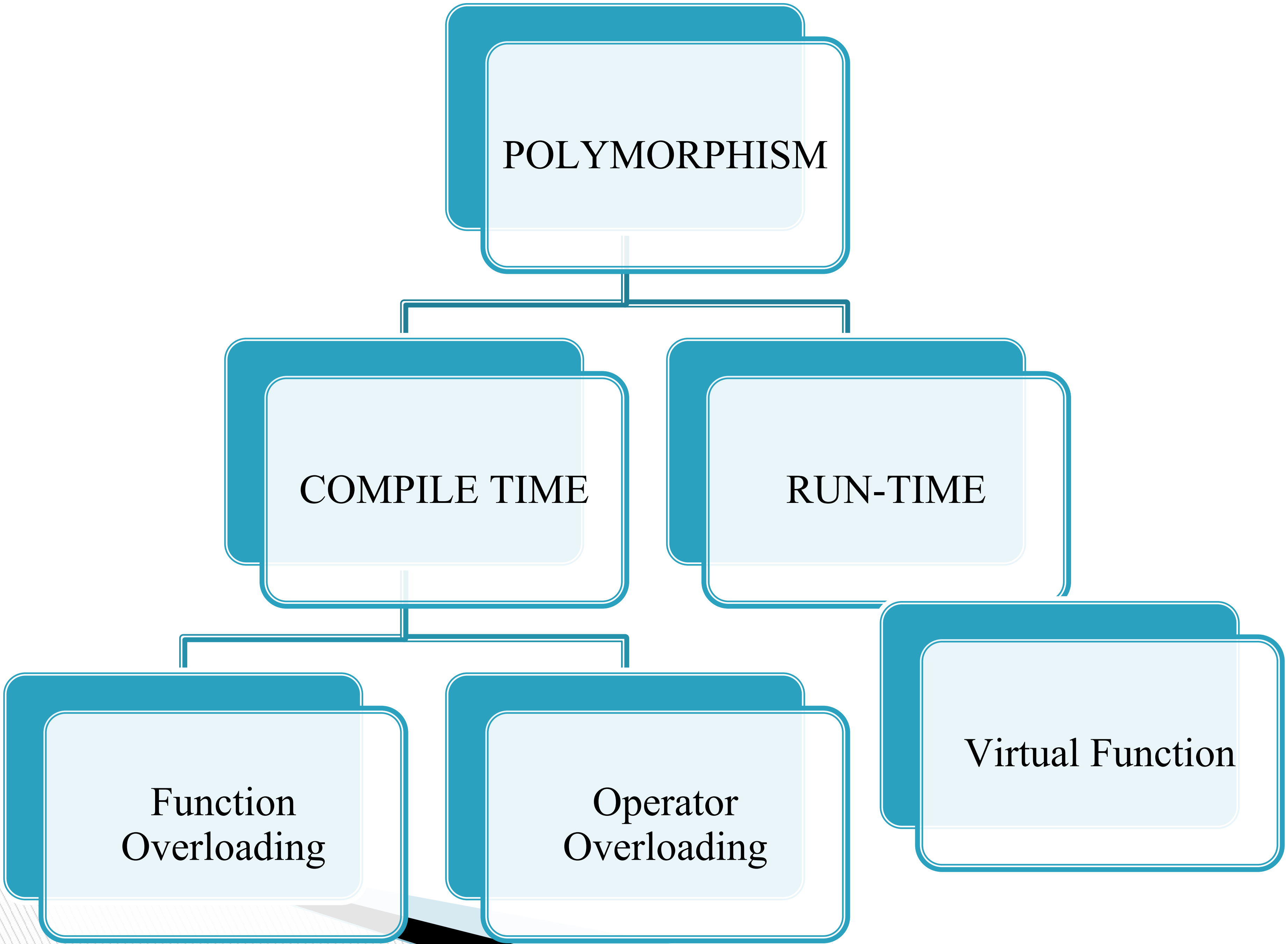
Run-time Polymorphism by Virtual Functions

- In runtime polymorphism, **the compiler resolves the object at run time and then it decides which function call should be associated with that object**.
- It is also known as dynamic or late binding polymorphism.
- This type of polymorphism is executed through virtual functions and function overriding.


POLYMORPHISM
COMPILE TIME
RUN-TIME
Function Overloading
Operator Overloading
Virtual Function

POINTER TO OBJECT

- Pointer to object is a variable containing an address of an object.
- It is similar to a pointer to any other variable. The normal address-of operator has to be used to get the address of an object.

```
class Demo
{
// Body of the class
}
Demo DemoObj;
Demo *PtrDemoObj;
PtrDemoObj = &DemoObj;
```

this POINTER

- The this pointer is a pointer to an object invoked by the member function.
- Suppose one writes DemoObj.DispDemo(), then a function call to this pointer will return the address of DemoObj.
- In other words, this pointer is a pointer to DemoObj.

```
person Elder(person OtherPerson)
{
      if(Age > OtherPerson.Age)
       return *this;
      else
       return OtherPerson;
}

main()
{
      person p1,p2,p3;
      p1=p2.Elder(p3);
}
```

COMPATIBILITY OF DERIVED AND BASE CLASS POINTERS

```
BaseClass BC;
DerivedClass DC : public BaseClass;
BaseClass *PtrBC;
DerivedClass PtrDC;
```

```
PtrBC = &BC; // Obvious, pointer and content are of similar types
PtrDC = &DC; // Here too
PtrBC = &DC; // This is done without any casting
// The following is not allowed
PtrDC = &BC
```

Subobject Concept

```
BaseClass1 BC1;
BaseClass2 BC2;
DerivedClass DC : public BaseClass1, BaseClass2;
BaseClass1 *PtrBC1;
BaseClass2 *PtrBC2;
DerivedClass PtrDC;
```

Now, consider the following statements:

```
PtrBC1 = &DC;
PtrBC2 = &DC;
```

BASE CLASS AND DERIVED CLASS MEMBER FUNCTIONS

- It is possible for two different member functions, one defined in the base class and another defined in the derived class, to have the same name. The function in the derived class is known as an *overridden function* in this case. The process is known as overriding the said function.

```
#include <iostream>
using namespace std;
class Shape
{
int LineStyle;
int FillColour;
public:
// Non-virtual function
void draw()
{
cout << "Shape is drawn \n";
}
};
class Circle : public Shape
{
int Radius;
int CentrePointX;
int CentrePointY;
public:
void draw()
{
cout << "Circle is drawn \n";
}
};
```

```
int main()
{
Shape SomeShape, *PtrShape;
Circle Ring;
SomeShape.draw();
Ring.draw();
PtrShape = &Ring;
PtrShape->draw();
/* This calls the draw function of the base class and not the
derived class where the pointer is pointing. */
}
```

Output

```
Shape is drawn
Circle is drawn
Shape is drawn
```

If Virtual Function in base class then

```
virtual void draw()
{
    cout << "Shape is drawn \n";
}
```

```
Shape is drawn
Circle is drawn
Circle is drawn
```

Virtual Function

- A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.
- **Some Key Points About Virtual Functions:**
- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword “**virtual**” inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime
- Below is the C++ program to demonstrate virtual function:

```
// C++ Program to demonstrate
// the Virtual Function
#include <iostream>
using namespace std;

// Declaring a Base class
class GFG_Base {

public:
    // virtual function
    virtual void display()
    {
        cout << "Called virtual Base Class function" <<
                "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Base print function" <<
                "\n\n";
    }
};
```

```
// Declaring a Child Class
class GFG_Child : public GFG_Base {

  public:
    void display()
    {
        cout << "Called GFG_Child Display Function" <<
                "\n\n";
    }


    void print()
    {
        cout << "Called GFG_Child print Function" <<
                "\n\n";
    }
};

  // Driver code
int main()
{
    // Create a reference of class bird
    GFG_Base* base;


    GFG_Child child;


    base = &child;


    // This will call the virtual function
    base->GFG_Base::display();


    // this will call the non-virtual function
    base->print();
}
```

- **Output:**

```
Called virtual Base Class function
Called GFG_Base print function
```

Virtual Destructors

```
class BaseClass
{
public:
BaseClass()
{
cout << "BaseClass constructor ..." << endl;
}
~BaseClass()
{
cout << "BaseClass destructor ..." << endl;
}
};

class DerivedClass : public BaseClass
{
public:
DerivedClass()
{
cout << "DerivedClass constructor ..." << endl;
}
~DerivedClass()
{
cout << "DerivedClass destructor ..." << endl;
}
};
```

```
void main()
{
BaseClass* ptrBase;
ptrBase = new DerivedClass();
delete ptrBase;
/* This should ideally call for DerivedClass destructor, but will
call only BaseClass destructor, because the destructor is not virtual.*/
getchar();
}
```

Output

```
BaseClass constructor ...
DerivedClass constructor ...
BaseClass destructor ...
```

Cont…

```
virtual ~BaseClass()
{
cout << "BaseClass class destructor ..." << endl;
}
```

```
BaseClass constructor ...
DerivedClass constructor ...
DerivedClass destructor ...
BaseClass destructor ...
```

PURE VIRTUAL FUNCTIONS

```
class Shape
{
        public:
        virtual void draw()=0;
};
```

- Pure virtual functions do not have the body where they are defined, but they can still have it outside the class.
- Pure virtual functions of base class force derived class to implement body of pure virtual function.