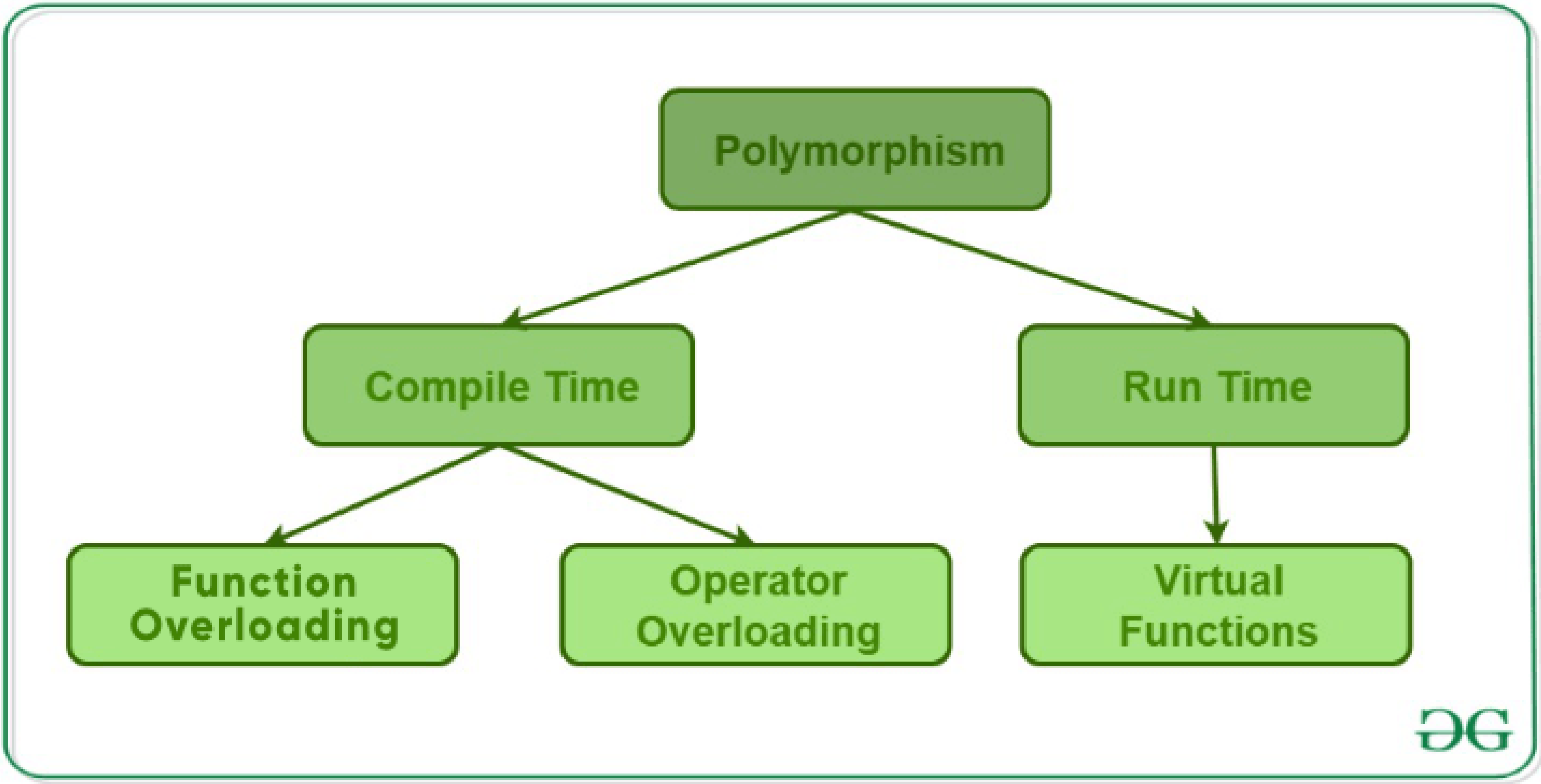
Polymorphism

Module -VII

Polymorphism – An Introduction

- *noun, the quality or state of being able to assume different forms* - Webster
- An essential feature of an OO Language
- It builds upon Inheritance



Polymorphism

Compile Time

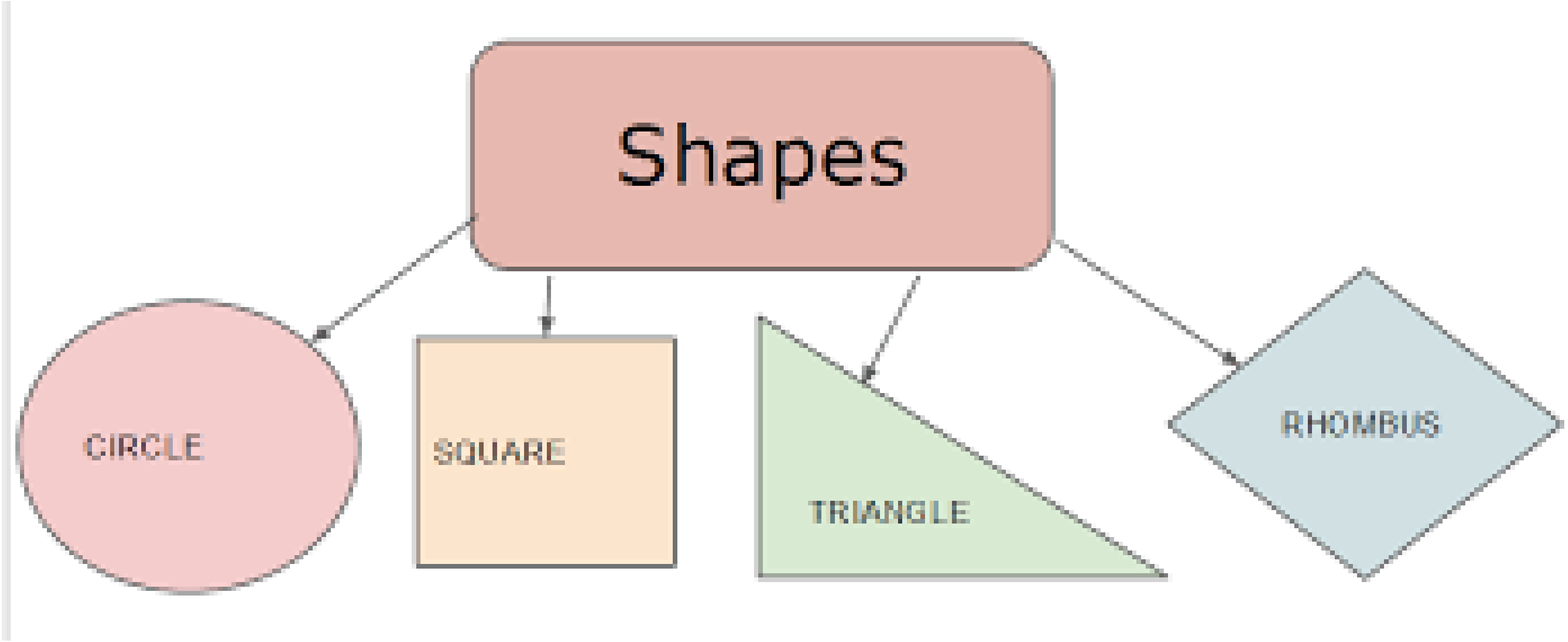
Run Time

Function Overloading

Operator Overloading

Virtual Functions

Polymorphism-Shapes



A Real Time Example

- A real-life example of polymorphism, 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 posses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

Before we proceed….

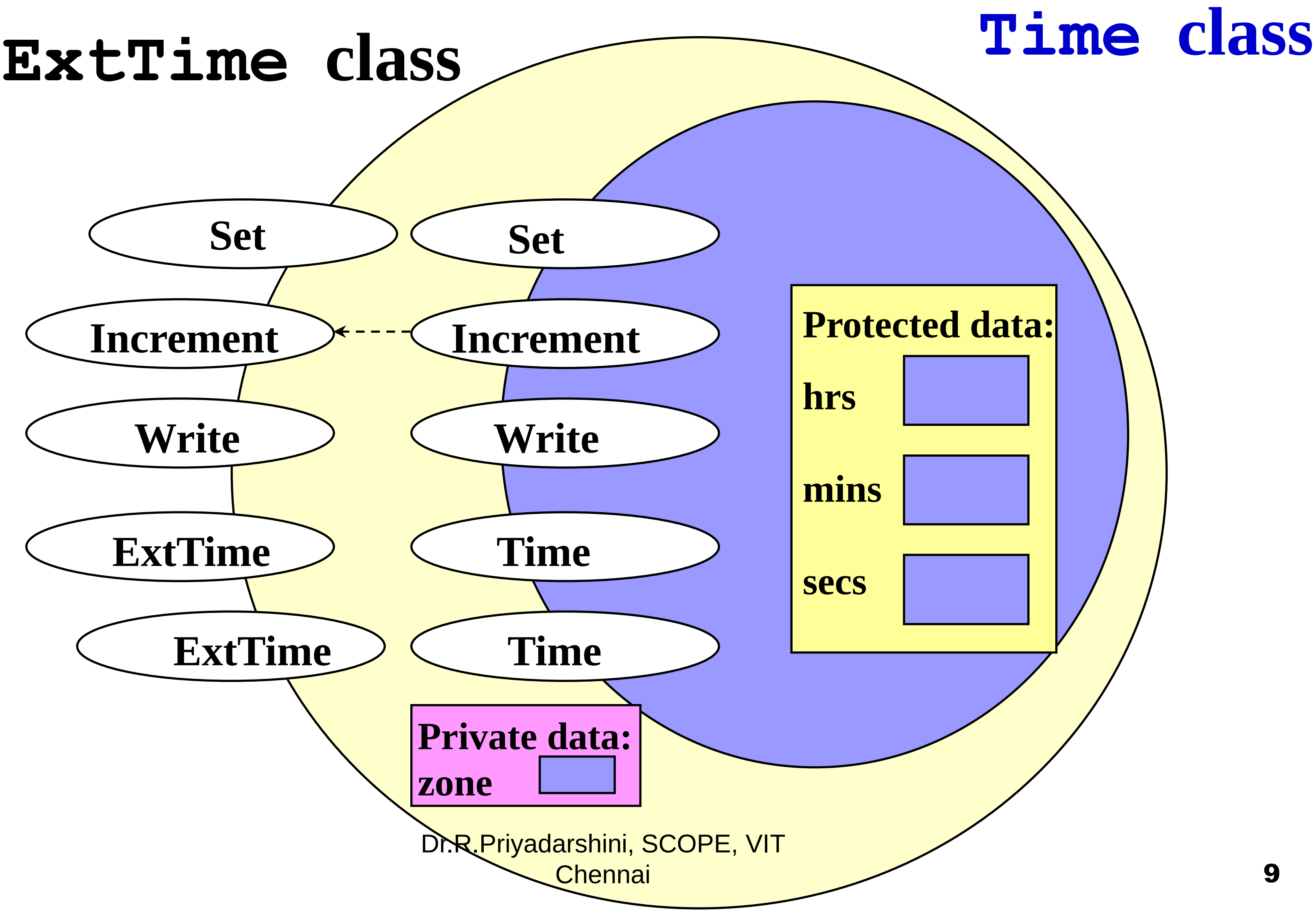
- Inheritance – Basic Concepts
  - Class Hierarchy
    - Code Reuse, Easy to maintain
  - Type of inheritance : public, private
  - Function overriding

Function Overriding

- Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class.
- A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding.
- It is like creating a new version of an old function, in the child class.

```
#include <iostream>
using namespace std;
class BaseClass {
public:
   void disp(){
      cout<<"Function of Parent Class";
   }
};
class DerivedClass: public BaseClass{
public:
   void disp() {
      cout<<"Function of Child Class";
   }
};
int main() {
   DerivedClass obj = DerivedClass();
   obj.disp();
   return 0;
}
```

Class Interface Diagram



Why Polymorphism?--Review:
Time and ExtTime Example by Inheritance

```
void  Print  (Time   someTime )  //pass an object by value
{
        cout   <<   "Time is  " ;
        someTime.Write (  ) ;                    // Time :: write()
        cout   <<   endl ;
}
```

CLIENT CODE

```
Time          startTime ( 8, 30, 0 ) ;
ExtTime       endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

```
Time is   08:30:00
Time is   10:45:00
```

Static Binding

- When the type of a formal parameter is a parent class, the argument used can be:

  the same type as the formal parameter,

  or,

  any derived class type.

- Static binding is the compile-time determination of which function to call for a particular object based on the type of the formal parameter
- When pass-by-value is used, static binding occurs

Can we do better?

```
void  Print  (Time   someTime )  //pass an object by value
{
          cout  <<  “Time is  “ ;
          someTime.Write (  ) ;                    // Time :: write()
          cout  <<  endl ;
}
```

```
Time          startTime ( 8, 30, 0 ) ;
ExtTime     endTime (10, 45, 0, CST) ;

Print ( startTime ) ;
Print ( endTime ) ;
```

OUTPUT

```
Time is  08:30:00
Time is  10:45:00
```

Polymorphism – An Introduction

- *noun, the quality or state of being able to assume different forms* - Webster
- An essential feature of an OO Language
- It builds upon Inheritance

- Allows run-time interpretation of object type for a given class hierarchy
  - Also Known as “Late Binding”

- Implemented in C++ using virtual functions

Dynamic Binding

- Is the run-time determination of which function to call for a particular object of a derived class based on the type of the argument
- Declaring a member function to be virtual instructs the compiler to generate code that guarantees dynamic binding
- Dynamic binding requires pass-by-reference

Virtual Member Function

```
class  Time
{
public :

    .   .   .

    virtual   void   Write ( ) ;              //  for dynamic binding
    virtual ~Time();                         // destructor
private :
    int                 hrs ;
    int                 mins ;
    int                secs ;
} ;
```

This is the way we like to see…

```
void  Print  (Time *   someTime )
{
   cout  <<  "Time is  " ;
   someTime->Write (  ) ;
   cout  <<  endl ;
}
```

```
Time        startTime( 8, 30, 0 ) ;
ExtTime    endTime(10, 45, 0, CST) ;

Time *timeptr;
timeptr = &startTime;
Print ( timeptr ) ;

timeptr = &endTime;
Print ( timeptr ) ;
```

Time::write()

ExtTime::write()

OUTPUT

```
Time is  08:30:00
Time is  10:45:00  CST
```

Virtual Functions

- Virtual Functions overcome the problem of run time object determination
- Keyword virtual instructs the compiler to use late binding and delay the object interpretation
- How ?
  - Define a virtual function in the base class. The word virtual appears only in the base class
  - If a base class declares a virtual function, it must implement that function, even if the body is empty
  - Virtual function in base class stays virtual in all the derived classes
  - It can be overridden in the derived classes
  - But, a derived class is not required to re-implement a virtual function. If it does not, the base class version is used

Polymorphism Summary:

- When you use virtual functions, compiler store additional information about the types of object available and created
- Polymorphism is supported at this additional overhead
- Important :
    - virtual functions work only with pointers/references
    - Not with objects even if the function is virtual
    - If a class declares any virtual methods, the destructor of the class should be declared as virtual as well.

```
#include <iostream>
using namespace std;
class base {
   public:
   virtual void print(){
      cout << "print base class" << endl;
   }
   void show(){
      cout << "show base class" << endl;
   }
};
class derived : public base {
   public:
   void print(){
      cout << "print derived class" << endl;
   }
   void show(){
      cout << "show derived class" << endl;
   }
};
```

```
int main(){
   base* bptr;
   derived d;
   bptr = &d;
   //calling virtual function
   bptr->print();
   //calling non-virtual function
   bptr->show();
}
```

Output:

```
print derived class
show base class
```

Abstract Classes & Pure Virtual Functions

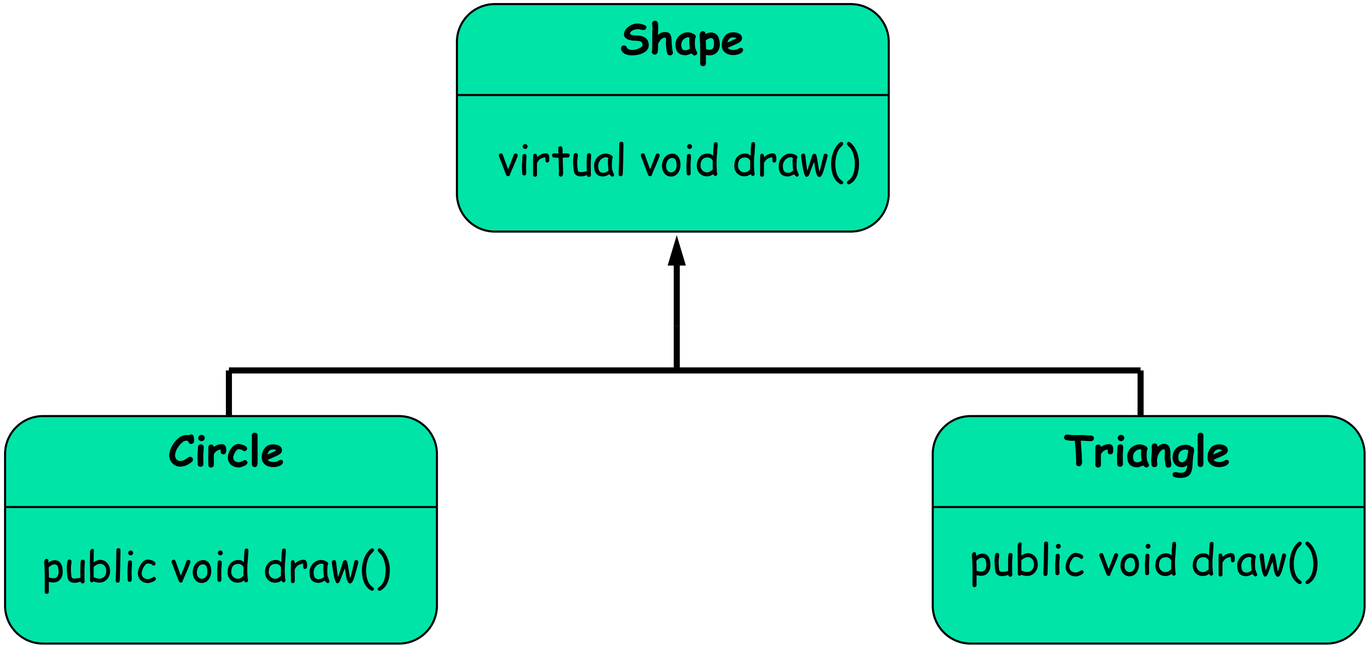
- Some classes exist logically but not physically.
- Example : Shape
  - `Shape s; // Legal but silly..!! : "Shapeless shape"`
  - Shape makes sense only as a base of some classes derived from it. Serves as a "category"
  - Hence instantiation of such a class must be prevented

```
class Shape    //Abstract
{
  public :
  //Pure virtual Function
  virtual void draw() = 0;
}
```

- A class with one or more pure virtual functions is an Abstract Class
- Objects of abstract class can't be created

```
Shape s; // error : variable of an abstract class
```

Example



Shape

virtual void draw()

Circle

public void draw()

Triangle

public void draw()

- A pure virtual function not defined in the derived class remains a pure virtual function.
- Hence derived class also becomes abstract

```
class Circle : public Shape { //No draw() - Abstract
   public :
   void print(){
      cout << “I am a circle” << endl;
   }
class Rectangle : public Shape {
   public :
   void draw(){ // Override Shape::draw()
      cout << “Drawing Rectangle” << endl;
   }
```

```
Rectangle r; // Valid
Circle c; // error : variable of an abstract class
```

Pure virtual functions : Summary

- Pure virtual functions are useful because they make explicit the abstractness of a class
- Tell both the user and the compiler how it was intended to be used
- Note : It is a good idea to keep the common code as close as possible to the root of you hierarchy

Summary ..continued

- It is still possible to provide definition of a pure virtual function in the base class
- The class still remains abstract and functions must be redefined in the derived classes, but a common piece of code can be kept there to facilitate reuse
- In this case, they can not be declared inline

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

// OK, not defined inline
void Shape::draw(){
 cout << “Shape" << endl;
}
```

```
class Rectangle : public Shape
{
  public :
   void draw(){
     Shape::draw(); //Reuse
     cout <<“Rectangle”<< endl;
}
```

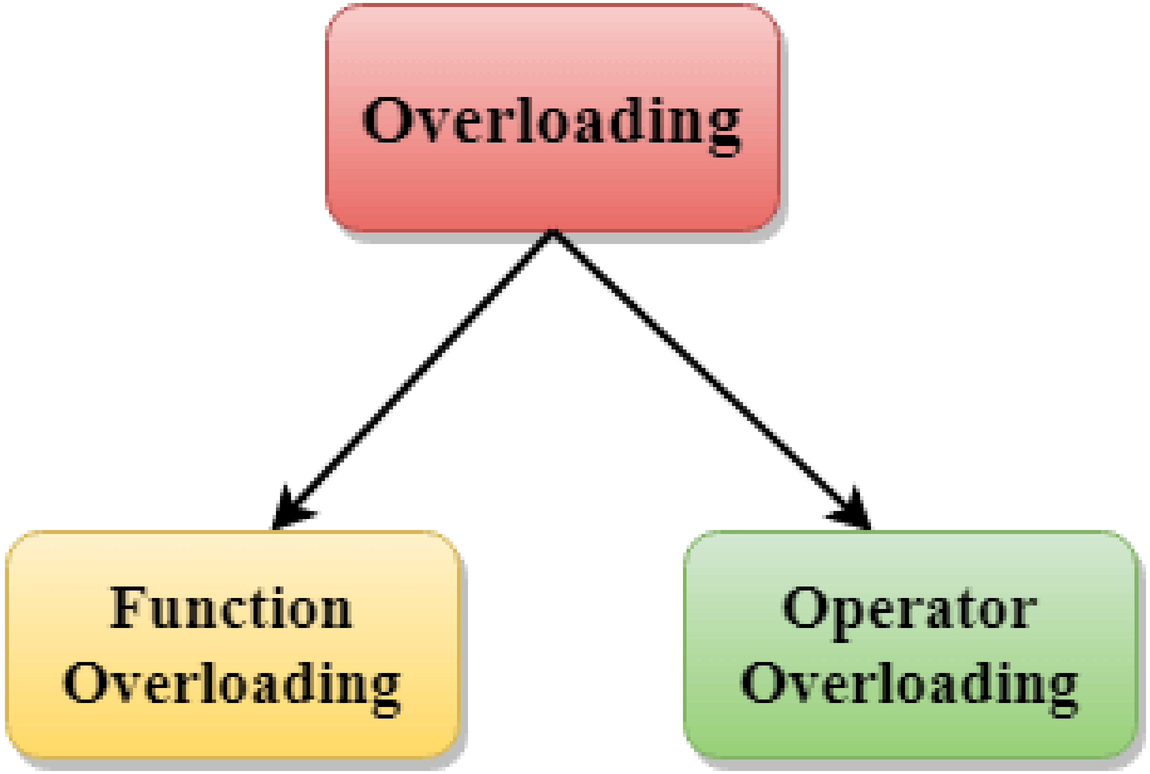
Summary

- Polymorphism is built upon class inheritance
- It allows different versions of a function to be called in the same manner, with some overhead
- Polymorphism is implemented with virtual functions, and requires pass-by-reference

Static Polymorphism

- C++ Overloading (Function and Operator)
- If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:
- methods,
- constructors

Types of overloading



Overloading

Function Overloading

Operator Overloading

Function Overloading

- Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++.
- In function overloading, the function is redefined by using either different types of arguments or a different number of arguments.
- It is only through these differences compiler can differentiate between the functions.

Advantages of Function Overloading

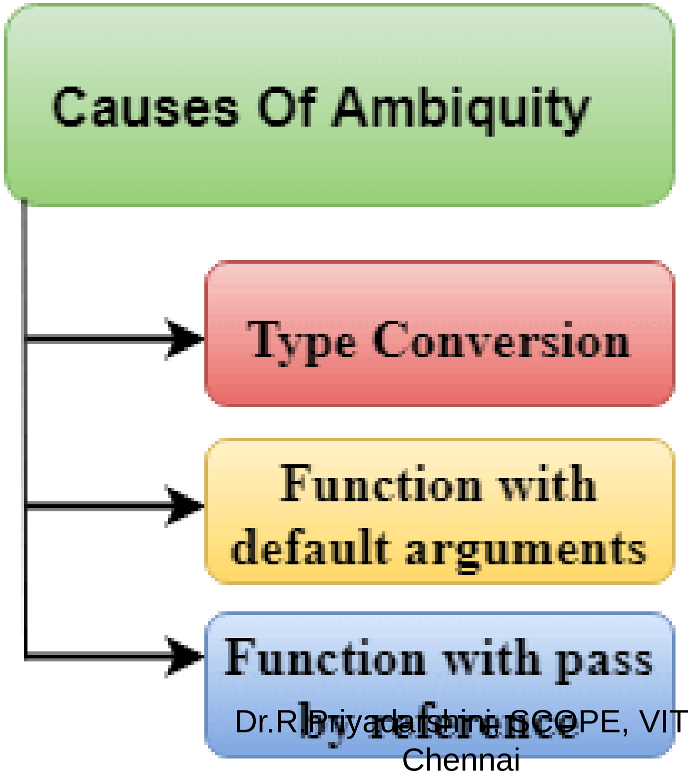
- The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

```
#include <iostream>
using namespace std;
class Cal {
    public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C;                                                                    //      class
object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
   return 0;
}
```

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);
 int mul(int a,int b)
{
    return a*b;
}
float mul(double x, int y)
{
    return x*y;
}
int main()
{
    int r1 = mul(6,7);
    float r2 = mul(0.2,3);
    std::cout << "r1 is : " <<r1<< std::endl;
    std::cout <<"r2 is : "  <<r2<< std::endl;
    return 0;
}
```

Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.



Type Conversion:

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{
    std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{
    fun(12);
    fun(1.2);
    return 0;
}
```

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
    std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
    std::cout << "Value of a is : " <<a<< std::endl;
    std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
    fun(12);

    return 0;
}
```

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

- **Rules of Operator Overloading**
- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains at least one operand of the user-defined data type.
- We cannot use friend function to overload certain operators.
- When unary operators are overloaded through a member function take no explicit arguments
- When binary operators are overloaded through a member function takes one explicit argument.

C++ Operators Overloading Example

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
      Test(): num(8){}
      void operator ++()        {
        num = num+2;
      }
      void Print() {
          cout<<"The Count is: "<<num;
      }
};
```

```
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output

```
10
```

```
#include <iostream>
using namespace std;
class A
{
    int x;
      public:
      A(){}
    A(int i)
    {       x=i;
    }
    void operator+(A);
    void display();
};
  void A :: operator+(A a)
{

    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;
  }
```

```
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Output

```
9
```