Lecture Four

Inheritance

© **Dr. Mohammad Mahfuzul Islam, PEng**
Professor, Dept. of CSE, BUET

Access Specifier

There 3 access specifiers:

- Private
- Protected
- public

Three ways of using access specifiers:

- For a member of a class
- During Inheritance
- Before class declaration

Access Specifier for a member of a class:

Access Specifier	Scope
private	Within the class only.
protected	**C++:** with the class and it's subclasses. **Java:** classes in package and subclasses inside or outside the package.
public	Open for All.
No specifier (Default)	**C++**: within the class only. **Java**: classes in the package.

Access Specifier for a member of a class

For C++:

	private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Subclass	No	No	Yes	Yes
Other class	No	No	No	Yes

For Java:

	private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access Specifier during Inheritance

- **The general form of Inheritance is:**

C++ class derived-class name: access base-class name{
}

Java class derived-class name extends base-class name{
}

C++

```
Class Base{
      ----------------
};

class Derived: public Base {
      ----------------------
};
```

Java

```
Class Base{
      ----------------
}

class Derived extends Base {
      ----------------------
}
```

For C++

Access Specifier	Change of Access Specifier from Base to Derived
Private / No Specifier	public -> private, protected -> private
protected	public -> protected, protected -> protected
public	public -> public, protected -> protected

✓ **Java doesn't use any access specifier during inheritance.**

Access Specifier before a Class Name

For Java

- ✓ A non-nested class has only two possible access levels: default and public.
- ✓ When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.
- ✓ Access Summary for Java:

✓ C++ doesn't use any access specifier before a Class Name.

Java Code

```
class Base{
    public int x;
    private int y;
    protected int z;
    public void setY(int n){ y = n;}
    public int getY(){ return y;}
    protected int getSum() { return x + y; }
}
class Derived extends Base{
    public void setZ(int n){ z = n;}
    public int getZ(){ return z;}
    public void showSum(){
        System.out.println( getSum() + z);}
}

public class Main {
    public static void main(String[] args) {
        Derived myObj = new Derived();

        myObj.x = 5;
        myObj.setY(4);
        myObj.setZ(7);
        myObj.showSum();
    }
}
```

Output:
16

Constructor and Destructor during Inheritance

- Constructors are executed in order of derivation (i. e., Base class constructor is executed first, then Derived class constructor is executed) and destructors are executed in reverse order.

- For passing arguments to the constructor of Base class, a chain of argument passing is necessary from Derived class to Base class.

- The general form of argument passing from derived class to base class in C++ is as follows:

```
Derived-constructor(arg-list): Base(arg-list){
}
```

In Java, super() is used to passed arguments from Derived class to Base class.

- Same arguments can be used in both base class and derived class. Derived class may simply pass the argument to the Base class.

Constructor and Destructor during Inheritance

C++ Code

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

class base {
     int x, y;
public:
     base(int n, int p){
          x = n;
          y = p;
          cout << "Constructing base class\n";
      }
      ~base(){
          cout <<  "Destructing base class\n";
      }
      void showxy(){
          cout << x << " " << y << '\n';
      }
};
```

```
class derived: public base {
     int i, j;
public:
     derived(int n, int m, int p): base (n, p){
          i = n;
          j = m;
          cout << "Constructing derived class\n";
     }
     ~derived(){
          cout <<  "Destructing derived class\n";
     }
     void showij(){
          cout << i << " " << j << '\n';
     }
};
```

```
int main(){
     derived ob(10, 20, 30);

     ob.showxy();
     ob.showij();
     return 0;
}
```

OUTPUT:

```
Constructing base class
Constructing derived class
10 30
10 20
Destructing derived class
Destructing base class
```

Constructor and Destructor during Inheritance

Java Code

```
class Base {
   final private int x;
   final private int y;
   Base(int n, int p){
     x = n;    //Final variables are initialized
     y = p;    //in Constructor
     System.out.println("Constructing Base class.");
   }
   public void showXY(){
     System.out.println(x + " "+y);
   }
}
```

```
class Derived extends Base {
   final private int i;
   final private int j;
   Derived(int n, int m, int p){
      super(n, p);
      i = n;
      j = m;
      System.out.println("Constructing Derived class.");
   }
   public void showIJ(){
      System.out.println(i + " "+ j);
   }
}
```

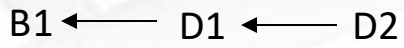
```
public class Main{
   public static void main(String [] args){
      Derived ob = new Derived(10, 20, 30);

      ob.showXY();
      ob.showIJ();
   }
}
```

OUTPUT:

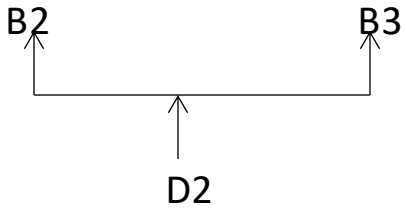
```
Constructing Base class.
Constructing Derived class.
10 30
10 20
```

Multiple Inheritance

- There are two ways that a derived class can inherit more than one base class: the hierarchical inheritance and the multiple base inheritance.
- The hierarchical inheritance: a derived class can be used as a base class for another derived class, creating a multilevel class hierarchy.



- The multiple base inheritances: a derived class can directly inherit more than one base class.



- The general for multiple base inheritance is as follows:

```
derived-constructor(arg-list): base1(arg-list), base2(arg-list), …, baseN(arg-list){
}
```

- When multiple base classes are inherited, constructors are executed from left to right, and destructors are executed in opposite order.

Java doesn't Support Multiple base Inheritance.

Multiple Inheritance

C++ Code

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

class B1 {
    int x;
public:
    B1(int i) {
        x = i;
        cout << "Constructing B1\n";
    }
    ~B1(){ cout << "Destructing B1\n";}
    int getx(){ return x;}
};

class B2 {
    int y;
public:
    B2(int j) {
        y = j;
        cout << "Constructing B2\n";
    }
    ~B2(){ cout << "Destructing B2\n";}
    int gety(){ return y;}
};
```

```
class B3 {
    int z;
public:
    B3(int k) {
        z = k;
        cout << "Constructing B3\n";
    }
    ~B3(){ cout << "Destructing B3\n";}
    int getz(){ return z;}
};
```

```
class D1: public B1 {
public:
    D1(int j):B1(j) {
        cout << "Constructing D1\n";
    }
    ~D1(){ cout << "Destructing D1\n";}
};
```

```
int main(){
    D2 ob(10, 20, 30);

    ob.show();
    return 0;
}
```

```
class D2: public D1, public B2, public B3 {
public:
    D2(int i, int j, int k): D1(i), B2(j),B3(k) {
        cout << "Constructing D2\n";
    }
    ~D2(){ cout << "Destructing D2\n";}
    void show(){
      cout << getx() << " " << gety() << " ";
      cout << getz() << '\n';
    }
};
```

OUTPUT:

```
Constructing B1
Constructing D1
Constructing B2
Constructing B3
Constructing D2
10 20 30
Destructing D2
Destructing B3
Destructing B2
Destructing D1
Destructing B1
```

Pointer to Derived Class

- A pointer declared as a pointer to a base class can also be used to point to any class derived from that base; however, the reverse is not true.
- A type casting can be used, but not recommended.

```
base *p;                  //base class pointer
base base_ob;             // object of base type
derived derived_ob;       // object of type derived
p = &base_ob;             // p points to base object
p = &derived_ob;          // p points to derived object
```

- Java doesn't Support Pointer, but still Base class reference can be used as-

```
Base ob = new Derived();
```

Method Overriding

- ***Method Overriding*** **is not possible if the method is not virtual.**
  - ✓ **In Java, the keyword virtual is not required to write explicitly. When a method with the same name is declared in Derived class, the method in Base class automatically becomes virtual.**
  - ✓ **In C++, the keyword virtual is used in base class and the keyword is not needed in derived class.**
- **A *virtual function* is a member function that is declared within a base class and redefined by a derived class.**
- **The determination of the type of object being pointed to by the pointer is made at run time.**
- **Only methods are overridden, variables cannot be virtual or overridden.**

Virtual destructor in C++ is mainly responsible for resolving the problem of memory leaks. When we use a virtual destructor inside the base class, it will call the destructor of the child class, ensuring that the child class's object should be deleted so there might be no memory leakage.

Method Overriding in C++

```
class Father {
   char name[20];
public:
   Father(char *fname){ strcpy(name,fname);}
   void show(){
      cout << "Father: " << name <<endl;}
};
class Son: public Father {
   char name[20];
public:
   Son(char *sname, char *fname):
       Father(fname){ strcpy(name, sname);}
   void show(){
       cout << "Son: " << name << endl;}
};
int main(){
   Father *fp, father("Rashid");
   Son son("Robin", "Rashid");
   fp = &father; fp->show();
   fp = &son; fp->show(); //without virtual
}
```

OUTPUT: Father: Rashid
Father: Rashed

C++ Code

```
class Father {
   char name[20];
public:
   Father(char *fname){ strcpy(name,fname);}
   virtual void show(){
      cout << "Father: " << name <<endl;}
};
class Son: public Father {
   char name[20];
public:
   Son(char *sname, char *fname):
       Father(fname){ strcpy(name, sname);}
   void show(){
       cout << "Son: " << name << endl;}
};
int main(){
   Father *fp, father("Rashid");
   Son son("Robin", "Rashid");
   fp = &father; fp->show();
   fp = &son; fp->show(); //with virtual
}
```

OUTPUT: Father: Rashid
Son: Robin

Method Overriding in Java

- Java allows Base class reference to declare an object of Derived class.

  Base ob = new Derived();

- The object can access all members of Base class. However, if the member is not present in Base class, then casting is required.

```
class Father{
   private String name;
   public int age;
   Father(String fName){
      name = fName;
      age = 60;
   }
   public void show(){
      System.out.println("Father:" + name);
   }
}
```

Java Code

```
class Son extends Father{
   private String name;
   public int age;
   Son(String sName, String fName){
      super(fName);
      age = 20;
      name = sName;
   }
   public void show(){
      super.show();
      System.out.println("Son:" + name);
   }
   public void showAge(){
      System.out.println("Son's Age: "+ age);
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      Father father = new Father("Rashid");
      Father son = new Son("Habib", "Jashim");

      father.show();
      son.show();
      ((Son)son).showAge();
      System.out.println(son.age);
      father = son;
      father.show();
   }
}
```

OUTPUT:

```
Father:Rashid
Father:Jashim
Son:Habib
Son's Age: 20
60
Father:Jashim
Son:Habib
```

Virtual Destructor

C++ Code

```
class Father {
    char *name;
public:
    Father(char *fname){
        name = new char[ strlen(fname)+1 ];
        strcpy(name, fname);
    }
    virtual ~Father(){
        delete name;
        cout << "Father destroyed" << endl;
    }
    virtual void show(){
        cout << "Father: " << name <<endl;
    }
};
```

```
class Son: public Father {
    char *name;
public:
    Son(char *sname, char *fname): Father(fname){
        name = new char[ strlen(sname)+1 ];
        strcpy(name, sname);
    }
    virtual ~Son(){
        delete name;
        cout << "Son destroyed" << endl;
    }
    virtual void show(){
        cout << "Son: " << name << endl;
    }
};
```

```
int main(){
   Father *fp = new Father("Rashid");
   fp->show();
   delete fp;
   fp = new Son("Robin", "Rashid");
   fp->show();
   delete fp;
   return 0;
}
```

OUTPUT:

Father: Rashid
Father destroyed
Son: Robin
Son destroyed
Father destroyed

Java doesn't support Destructor

Abstract or Pure Virtual Method

- A pure *virtual function* has no definition relative to the base class. Only the function's prototype is included. The general form is:

  virtual type func-name (parameter-list) = 0;

- If a class contains a pure virtual function, then the class is referred to abstract class. No object can be created for abstract class.

C++ Code

```
class area {                    //Abstract class
    double dim1, dim2;
public:
    void setarea( double d1, double d2){
        dim1 = d1;    dim2 = d2;
    }
    void getdim( double &d1, double &d2){
        d1 = dim1;    d2 = dim2;
    }
    virtual double getarea() = 0;
};
```

```
class rectangle: public area {
public:
    double getarea(){
        double d1, d2;
        getdim(d1, d2);
        return d1 * d2;}
};
```

```
class triangle: public area {
public:
    double getarea(){
        double d1, d2;
        getdim(d1, d2); return 0.5*d1 * d2; }
};
```

```
int main(){
    area *p;      //area p; -> not permitted
    rectangle r;
    triangle t;

    r.setarea( 3.3, 4.5);
    t.setarea( 4.0, 5.0);

    p = &r; cout << p->getarea() << '\n';
    p = &t; cout << p->getarea() << '\n';
    return 0;
}
```

OUTPUT:

```
14.85
10
```

Abstract or Pure Virtual Method

- Java support *abstract function* instead of pure virtual function and the class is known as abstract class. No object can be created based on abstract class.
- It's subclasses' responsibility to override abstract method.
- Constructor or static method cannot be abstract.

JAVA Code

```
abstract class Figure {
    double dim1, dim2;
    Figure(double a, double b){ dim1 = a; dim2 = b;}
    abstract double area();
}
class Rectangle extends Figure {
    Rectangle(double a, double b) { super(a, b);}
    double area(){ return dim1*dim2;}
}
class Triangle extends Figure {
    Triangle(double a, double b) {super(a, b);}
    double area(){ return 0.5*dim1*dim2;}
}
```

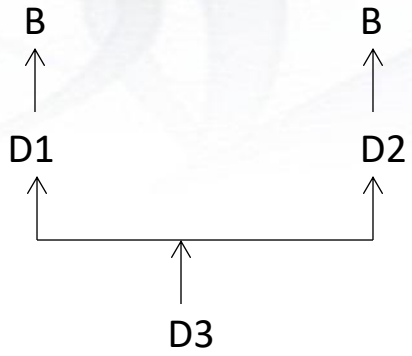
```
public class FigureDemo {
  public static void main(String[] args) {
      Rectangle r = new Rectangle(4, 5);
      Triangle t = new Triangle(4, 3);
       Figure figref;

       figref = r;
       System.out.println("Area: "+figref.area());
       figref = t;
       System.out.println("Area: "+ figref.area());
  }
}
```

OUTPUT:

```
Area: 20.0
Area: 6.0
```

Diamond Problem: Virtual Inheritance

- In the following scenario, B is inherited by D1 and D2 and D3 directly inherits D1 and D2. This causes ambiguity (whether calling through D1 or D2) when a member of B is inherited in D3, because two copies of B is included in D3.



- The keyword "virtual" precedes the base class in both cases inherits only one copy in D3.

Java doesn't support multiple base Inheritance. So, no such ambiguity.

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

class B{
public:
    int i;
};

class D1: virtual public B{
public:
    int j;
};

class D2: virtual public B{
public:
    int k;
};

class D3: public D1, public D2{
public:
    int product(){return i *j;}
};

int main(){
    D1 ob1;
    D3 ob3;

    ob1.i = 100;
    ob3.i = 10;
    ob3.j = 3;
    cout << ob1.i << endl;
    cout << ob3.i << endl;
    cout << ob3.product() <<endl;

    return 0;
}
```

C++ Code

OUTPUT:

```
100
10
30
```

Interface in Java

- **Interfaces** are designed to support **dynamic method resolution** at **run-time.**
- **Interface** is fully abstract:
  - ✓ **No instance variable** is allowed, but may contain **variables that are initialized** to desired values. These act as **shared constants** into multiple classes.
  - ✓ Methods are declared with **without any body**. Exceptions: Default and private methods.
- An interface can be implemented by **any number of classes**; and one class can implement **any number of interfaces.** (Note that A class can't inherit more than one class, which is also true for abstract class.)
- To implement an interface, a class must provide the **complete set of methods** required by the interface. However, **abstract method** can implement methods partially.
- There can be **only one public interface** in a file and the file name must be **the same** as the interface.
- If a class implements two interfaces that declare the **same method**, then the same method will be used by clients of either interface.
- One interface can **inherit** another interface by the use of the keyword **extends.**

Interface in Java

```
interface InterfaceDemo{
   public double PI = 3.14159;
   public void volume();
   default double circleArea(double radius) {
      System.out.println(getName());
      return PI*radius*radius;
   }
   private String getName(){
      return "Area is calculated using default method.";
   }
}

class Cone implements InterfaceDemo{
   private String name;
   private double height;
   private double radius;
   Cone(double nh, double nr){
      height = nh;
      radius = nr;
   }
   public void volume(){
      double vol = height * circleArea(radius) / 3;
      System.out.println(name + " Volume: "+ vol);
   }
}
```

```
class Cylindar implements InterfaceDemo{
   private String name;
   private double height;
   private double radius;
   Cylindar(double nh, double nr){
      height = nh;
      radius = nr;
   }
   public void volume(){
      double vol = height * circleArea(radius);
      print(vol);
   }

   private void print(double vol){
      System.out.println(name + " Volume: " +
vol);
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      Cylindar cylindar = new Cylindar(5.6, 4.2);
      Cone cone = new Cone(5.6, InterfaceDemo.PI);

      cylindar.volume();
      cone.volume();
   }
}
```

OUTPUT:

Area is calculated using default method.
null Volume: 310.33882656
Area is calculated using default method.
null Volume: 57.87823647334713

Nested Interface

- A nested interface is nested to another **interface** or **class**.
- **Top level Interface** must be **public** or **default**. A nested interface can be **public**, **private** or **protected**.
- Outside the enclosing scope, the **Nested Interface** must be fully **qualified** with name of class or interface.

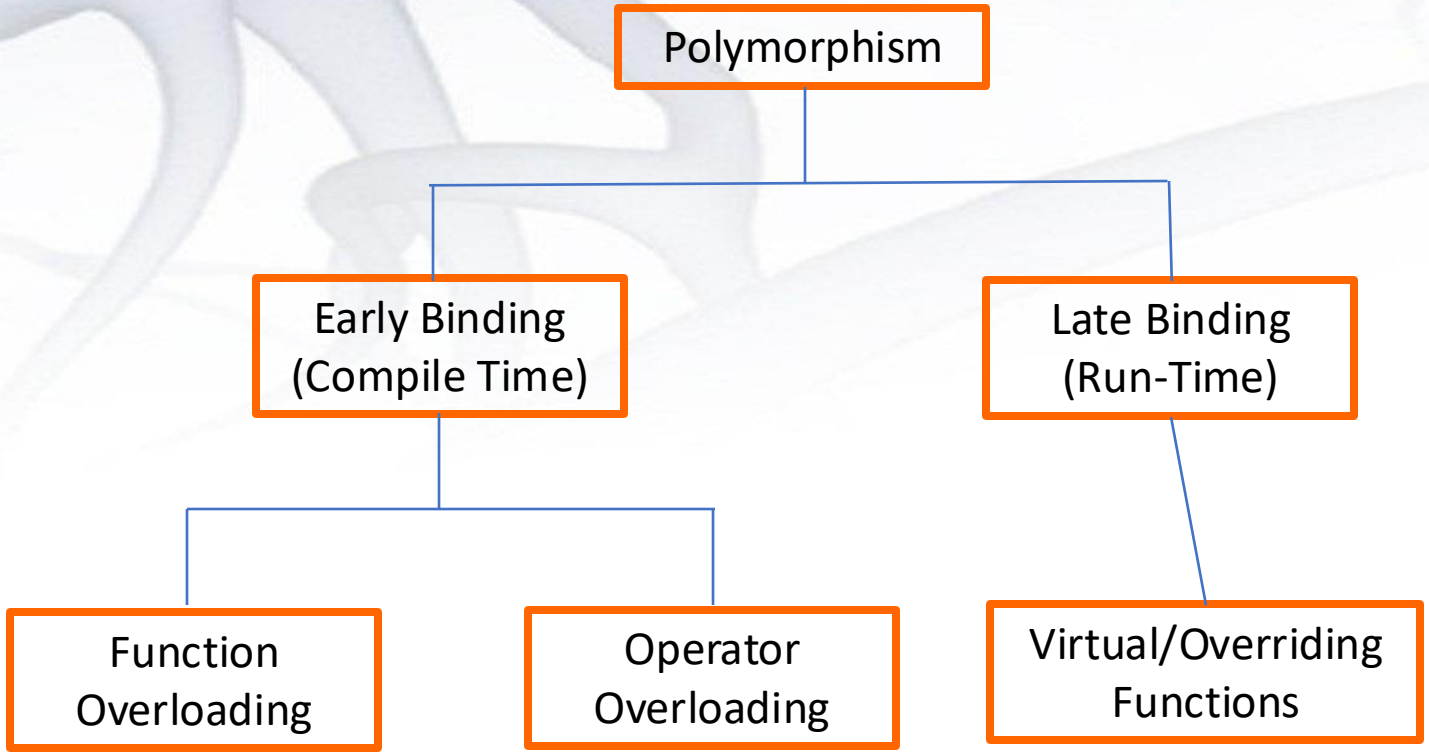
```
class A{
    public interface Nested{
        public boolean isNegative(int x);
    }
}

interface inNested extends A.Nested{
    public void print(String str);
}

class B implements inNested, A.Nested{
    public boolean isNegative(int x){
        return  x < 0 ? true: false;
    }
    public void print(String str){
        System.out.println(str);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        A.Nested nes = new B();
        inNested ines = new B();

        if(!nes.isNegative(10))
            ((B) nes).print("Positive");
        if (ines.isNegative(-5))
            ((B) ines).print("Negative.");
    }
}
```

OUTPUT:

Positive
Negative.

Run-Time Polymorphism
(Dynamic Method Dispatch)



There are two terms linked with OOP:

- **early binding and**
- **late binding.**

➢ **Early binding refers to those function calls that can be resolved during compilation. This method is faster but not flexible.**

➢ **Late binding refers to those function calls that can be resolved during run time. This method is slower but flexible.**

Run-Time Type Identification (RTTI)

C++ Code

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

class Animal{
public:
    virtual void sound() = 0;
};

class Dog: public Animal{
    void sound(){
        cout << "Bark" << endl;
    }
};

class Cat: public Animal{
    void sound(){
        cout << "Meow" << endl;
    }
};
```

```
int main() {
    Dog* dog = new Dog();
    Cat* cat = new Cat();
    double length = 25.5;
    int num = 10;
    Animal *animal;

    const type_info &type_num = typeid(num);
    cout << "Type(num): " << type_num.name() << endl;
    cout << "Type(dog): " << typeid(dog).name() << endl;
    cout << "Type(*dog): " << typeid(*dog).name() << endl;
    cout << "Type(cat): " << typeid(cat).name() << endl;
    // cout << "Type(*num): " << typeid(*num).name() << endl;
    cout << "Type(length): " << typeid(length).name() << endl;
    // cout << "Type(*length): " << typeid(*length).name() << endl;
    animal = dog;
    cout << "Type(animal): " << typeid(animal).name() << endl;
    cout << "Type(*animal): " << typeid(*animal).name() << endl;
    animal = cat;
    cout << "Type(animal): " << typeid(animal).name() << endl;
    cout << "Type(*animal): " << typeid(*animal).name() << endl;
    cout << "Type(length+num): " << typeid(length+num).name() << endl;
    return 0;
}
```

OUTPUT:

```
Type(num): i
Type(dog): P3Dog
Type(*dog): 3Dog
Type(cat): P3Cat
Type(length): d
Type(animal): P6Animal
Type(*animal): 3Dog
Type(animal): P6Animal
Type(*animal): 3Cat
Type(length+num): d
```

Run-Time Type Identification (RTTI)

```
abstract class Animal{
   abstract public void sound();
}

class Dog extends Animal{
   public void sound(){
      System.out.println("Bark");
   }
}
class Cat extends Animal{
   public void sound(){
      System.out.println("Meow");
   }
}

class PrintType{
   public void printType(Object ob){
      System.out.println(ob.getClass());
   }
}
```

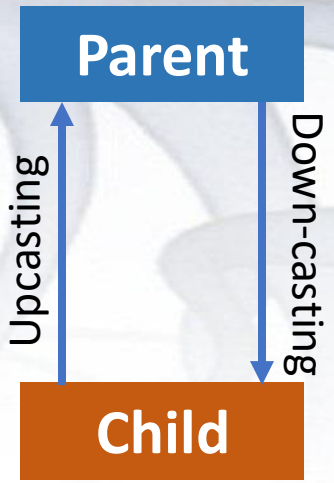
Java Code

```
public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();
      Cat cat = new Cat();
      Double length = 25.5;
      Integer num = 10;
      PrintType pt = new PrintType();

      if (dog instanceof Dog)
         System.out.println("Yes, Dog.");

      pt.printType(dog);
      pt.printType(cat);
      Animal animal = dog;
      pt.printType(animal);
      animal = cat;
      pt.printType(animal);
      Object ob = length;
      pt.printType(ob);
      ob = num;
      pt.printType(ob);
      ob = dog;
      pt.printType(ob);
   }
}
```

OUTPUT:

```
Yes, Dog.
class Dog
class Cat
class Dog
class Cat
class java.lang.Double
class java.lang.Integer
class Dog
```

Type Casting Operator in C++



Consider:

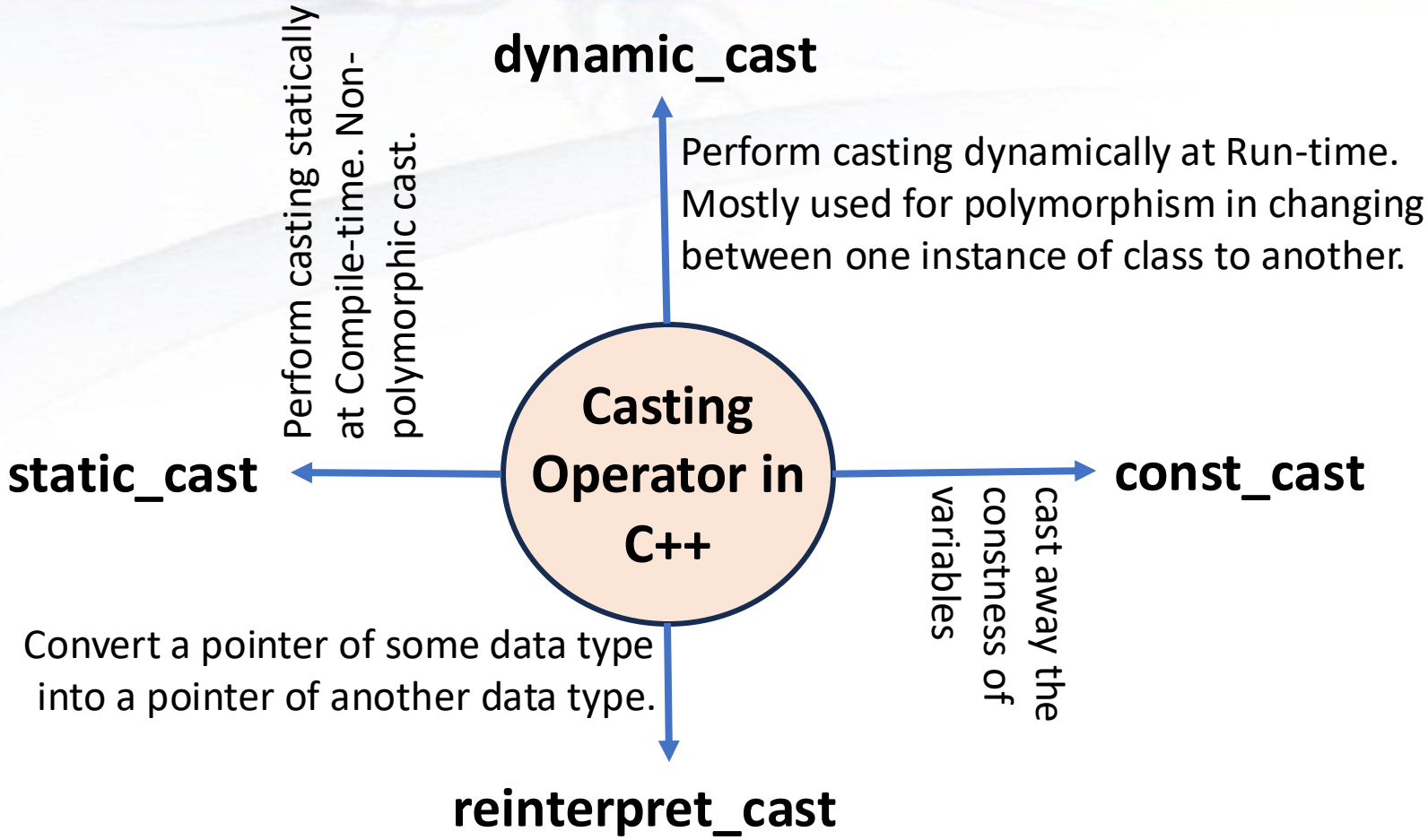
```
float length = 12.5;
int num = 10;
```

Upcasting: Implicit or Explicit

```
length = num;
length = (float) num;
```

Down-casting: Always explicit

```
num = (int) length;
```



Java supports only C like representation of static and dynamic casting.

Type Casting Operator in C++

static_cast:

- **Perform casting statically at Compile-time. Non-polymorphic cast.**
  - ✓ Static_cast is normal/ordinary type conversion.
  - ✓ Does like implicit and also explicit type conversion.

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

class Base {
public:
    virtual void print(){
        cout << "Base" << endl;
    }
};
class Derived : public Base {
public:
    void print(){
        cout << "Derived" << endl;
    }
};
class MyClass {};
```

OUTPUT:

num: 4
length: 4.26
wide: 4
Derived
Derived
Base
Base
Derived

```
int main(){
    float length = 4.26;
    int num = static_cast<int>(length);
    float wide = num;
    char c = 'A';
    cout << "num: " << num << endl;
    cout << "length: " << length << endl;
    cout << "wide: " << wide << endl;
    // int* p = static_cast<int*>(&c);

    Derived* d = new Derived;
    d->print();
    Base* b = static_cast<Base*>(d);
    b->print();
    b = new Base();
    b->print();
    Base * base = new Base();
    Derived * derived = static_cast<Derived*>(base);
    derived->print();
    derived = new Derived();
    derived->print();
    // MyClass* x = static_cast<MyClass*>(d);

    return 0;
}
```

Type Casting Operator in C++

dynamic_cast:

- ✓ cast one type of pointer into another **or** one type of reference into another at run-time.
- ✓ If the dynamic_cast involves pointer, it returns NULL if the casting fails.
- ✓ If the dynamic_cast involves reference type fail, it throws bad_cast exception

```
int main(){
   Animal* animal = new Animal;
   Dog* dog = new Dog;
   Cat* cat = new Cat;

   animal->print();
   animal = dynamic_cast<Animal*>(dog);
   animal->print();
   animal = dynamic_cast<Animal*>(cat);
   animal->print();
   Dog* d = dynamic_cast<Dog*>(animal);
   if (d == nullptr){
       cout << "Conversion failed" << endl;
   } else d->print();
   myClass* x = dynamic_cast<myClass*>(animal);
   return 0;
}
```

OUTPUT:

Animal
Dog
Cat
Conversion failed

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

class Animal {
public:
   virtual void print(){
       cout << "Animal" << endl;
   }
};
class Dog : public Animal {
public:
   void print(){
       cout << "Dog" << endl;
   }
};
class Cat : public Animal {
public:
   void print(){
       cout << "Cat" << endl;
   }
};

class myClass {};
```

Type Casting Operator in C++

const_Cast:

- **Const_cast is used to cast away the constness of variables. Used only for pointer or reference.**

```
int main(){
   const int num = 10;
   const int* ptr1 = &num;
   int* ptr2 = const_cast<int*>(ptr1);

   cout << "Value of num after: " << funct(ptr2) << endl;
   cout << "*ptr1: " << *ptr1 << endl;
   cout << "Value of num before: " << num << endl;
   //int val = const_cast<int>(num);

   Student std(3);
   cout << "Roll number before: " << std.getRoll() << endl;
   std.funct();
   cout << "Roll number after: " << std.getRoll() << endl;

   return 0;
}
```

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

int funct(int* ptr){
   *ptr = *ptr + 10;
   return *ptr;
}

class Student{
   int roll;
public:
   Student(int r){ roll = r;};
   void funct() const{
      (const_cast<Student*>(this))->roll = 5;
   }
   int getRoll() const{return roll;}
};
```

OUTPUT:

Value of num after: 20
*ptr1: 20
Value of num before: 10
Roll number before: 3
Roll number after: 5

Type Casting Operator in C++

reinterpret_Cast:

- **Convert a pointer of some data type into a pointer of another data type.**

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

struct mystruct{
    int a = 5;
    char c = 'a';
};
class A{
public:
   void display(){
       cout << "Class A" << endl;
   }
};
class B{
public:
   void display(){
       cout << "Class B" << endl;
   }
};
```

OUTPUT:

```
65
A
0x7fba70f059d0

5
32609

Class B
Class A
```

```
int main(){
   int* p = new int(65);
   char* ch = reinterpret_cast<char*>(p);
   cout << *p << endl;
   cout << *ch << endl;
   cout << p << endl << endl;

   mystruct s;
   int* iptr = reinterpret_cast<int*>(&s);
   cout << *iptr << endl;
   iptr++;
   cout << *iptr << endl << endl;

   B* b = new B();
   b->display();
   A* a = reinterpret_cast<A*>(b);
   a->display();

   return 0;
}
```

Object class in Java

- The class **Object** is a special class defined by Java.
- All other classes are subclasses of **Object**.

Methods of Object class	
Object clone()	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize()	Called before an unused object is recycled.
Class<?> getClass()	Obtains the class of an object at run time.
int hashCode()	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

The methods getClass(), notify(), notifyAll() and wait() is declared as final.