CS2310 Computer Programming

LT12 Object Oriented Programming-II

Computer Science, City University of Hong Kong

Semester A 2023-24

Outline

- Friend function
- Operator overloading
- Inheritance
- Polymorphism

Friend Function

- Not all functions could logically belong to a class, and sometimes, it is more natural to implement an operation as ordinary (nonmember) functions,
  - e.g. Equality (==) function that test if 2 objects are equal
- Equality operator = = cannot be applied directly on objects or structures
- Defining it as a member function will lose the symmetry
- It is more natural to define such function as an ordinary (nonmember) function

Equality testing: ordinary function

```
#include <iostream>
using namespace std;
class Rectangle
{
public:
    Rectangle(int w, int h);
    int getArea();
    int getWidth();
    int getHeight();
private:
    int width;
    int height;
};
```

```
Rectangle::Rectangle(int w,int h){
    width=w;
    height=h;
}
int Rectangle::getWidth(){
    return width;
}
int Rectangle::getHeight(){
    return height;
}
int Rectangle::getArea(){
    return width*height;
}
```

Equality testing: ordinary function

```
bool equal(Rectangle r1, Rectangle r2){
    if (r1.getWidth()==r2.getWidth() && r1.getHeight()==r2.getHeight())
        return true;
    else
        return false;
}

int main() {
    Rectangle ra(10,22), rb(10,21);
    if (equal(ra, rb))
        cout << "They are the same\n");
    return 0;
}
```

- Equality function needs to call access functions several times
- However, declare the member variable as public and direct access them are not recommend

Friend function

- Solution: Define a friend function!
- A friend function of a class is *not* a member function of the class but has access to the private members of that class
- A friend function doesn’t need to call access functions → more efficient
- Also the code looks simpler
- A friend function will be public no matter it is defined under “public:” or not

Equality testing: friend function

```
#include <iostream>
using namespace std;
class Rectangle
{
public:
    Rectangle(int w, int h);
    friend bool equal(Rectangle r1,Rectangle r2);
    int getArea();
    int getWidth();
    int getHeight();
private:
    int width;
    int height;
};
```

```
Rectangle::Rectangle(int w,int h){
    width=w;
    height=h;
}
int Rectangle::getWidth(){
    return width;
}
int Rectangle::getHeight(){
    return height;
}
int Rectangle::getArea(){
    return width*height;
}
```

Equality testing: friend function

```
/*Note the friend function is not implemented in Rectangle class*/

bool equal(Rectangle r1, Rectangle r2){
    if (r1.width == r2.width && r1.height == r2.height) // granting access of private members
        return true;
    else
        return false;
}

int main()
{
    Rectangle ra(10,22), rb(10,22);
    if (equal(ra, rb))
        cout << "They are the same\n";
    return 0;
}
```

Equality testing with Reference Object

- Pass by Reference
  - the original data, not the copy is passed to a function
  - Add ‘&’ before the parameter name in function prototype and definition.

```
class Rectangle
{  ……
   friend bool equal(Rectangle &r1, Rectangle &r2);
   ……
};
bool equal(Rectangle &r1, Rectangle &r2){
   if (r1.width == r2.width && r1.height == r2.height)
   ……
}
```

Outline

- Friend function
- Operator overloading
- Inheritance
- Polymorphism

Operator Overloading

- Enabling C++'s operators to work with class objects
- Using traditional operators with user-defined objects
- Examples of already overloaded operators
  - Operator << is both the stream-insertion operator and the bitwise left-shift operator
  - **+** and **–**, perform arithmetic on multiple types

Operator Overloading

```
class Triangle {
private:
    double s1, s2, s3;
    double area;
public:
    Triangle() {}
    void setSides();
    void computeArea();
    double getArea();
};
```

```
Triangle a, b;

a.setSides();
b.setSides();
```

lhs = Left Hand Side

rhs = Right Hand Side

```
if (a < b) {
    cout << "Triangle a is smaller than triangle b\n";
}
```

Operator Overloading

\+ - * / % ^ & | ~ ! , = < > <= >=

++ -- << >> == != && || += -= *= /=

%= ^= &= |= <<= >>= [] () ->

->* new new[] delete delete[]

Operator Overloading

- Overloading an operator
  - Write function definition as normal
  - Function name is keyword **operator** followed by the symbol for the operator being overloaded
  - **operator+** used to overload the addition operator (**+**)
- Special operators
  - To use an operator on a class object it must be overloaded except the assignment operator**(=)** or the address operator**(&)**
    - Assignment operator by default performs member-wise assignment
    - Address operator (&) by default returns the address of an object

Operator Overloading: Member Function

- Add a function called operator _ (e.g., <, +, !) to your class:

```
class Circle {
private:
    int radius;
public:
    Circle(int radius): radius(radius) {};
    bool operator< (Circle& rhs); // unary parameter
    bool operator> (Circle& rhs); // unary parameter
    //  lhs (left hand side) of each operator is this.
};

bool Circle::operator<(Circle& rhs) {
    if (radius < rhs.radius) return true;
    else return false;
}
```

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

int main() {
    Circle a(3);
    Circle b(5);
    cout << (a < b);
    return 0;
}

// a < b <-> a.operator<(b)
```

Operator Overloading: Friend Function

- *Friend function:* a special function which is a non-member function of a class but has privilege to access private and protected data of that class

- Friend function can be declared in any section of the class i.e. public or private or protected

- When friend function is called neither name of object nor dot operator is used

Operator Overloading: Friend Function

```
class Triangle {
private:
    double s1, s2, s3;
public:
    Triangle() { s1=0; s2=0; s3=0; }
    Triangle(double s1, double s2, double s3): s1(s1), s2(s2), s3(s3) {}
    double getArea();
    friend bool operator< (Triangle &lhs, Triangle &rhs); // binary parameters
    friend bool operator> (Triangle &lhs, Triangle &rhs); // binary parameters
    friend ostream& operator<< (ostream &outs, Triangle &c); // binary parameters
};
double Triangle::getArea() {
    double s = (s1+s2+s3)/2;
    return sqrt(s*(s-s1)*(s-s2)*(s-s3));
}
```

Operator Overloading: Friend Function

```cpp
bool operator<(Triangle &lhs, Triangle &rhs) {
    return lhs.getArea() < rhs.getArea();
}

bool operator>(Triangle &lhs, Triangle &rhs) {
    return lhs.getArea() > rhs.getArea();
}

ostream &operator << (ostream &outs, Triangle &t) {
    outs << "The sides are: ";
    outs << t.s1 << " " << t.s2 << " " << t.s3 << " ";
    outs << "The area is: ";
    outs << t.getArea() << endl;
    return outs;
}
```

```cpp
int main() {
    Triangle t1(3, 4, 5);
    Triangle t2(5, 6, 7);
    cout << t1;
    cout << t2;
    if (t1 < t2) {
        cout << "t1 is smaller\n";
    } else {
        cout << "t2 is smaller\n";
    }
    return 0;
}
```

Operator Overloading: Copy assignment

```
class Date {
public:
    int year;   int month;    int day;
    // Default constructor
    Date() : year(0), month(0), day(0) {}
    // Parameterized constructor
    Date(int y, int m, int d) : year(y), month(m), day(d) {}
    // Copy constructor
    Date(const Date& other) : year(other.year), month(other.month), day(other.day) {
        // You can include additional logic here if needed
    }
    // Copy assignment operator
    Date& operator=(const Date& other) {
        if (this != &other) { // Protect against invalid self-assignment
            year = other.year;
            month = other.month;
            day = other.day;
            // You can include additional logic here if needed
        }
        return *this; // Dereference to enable chaining of assignments
    }
};
```

```
// Normal constructor
Date date1(2022, 11, 8);
// Copy constructor
Date date2 = date1;
```

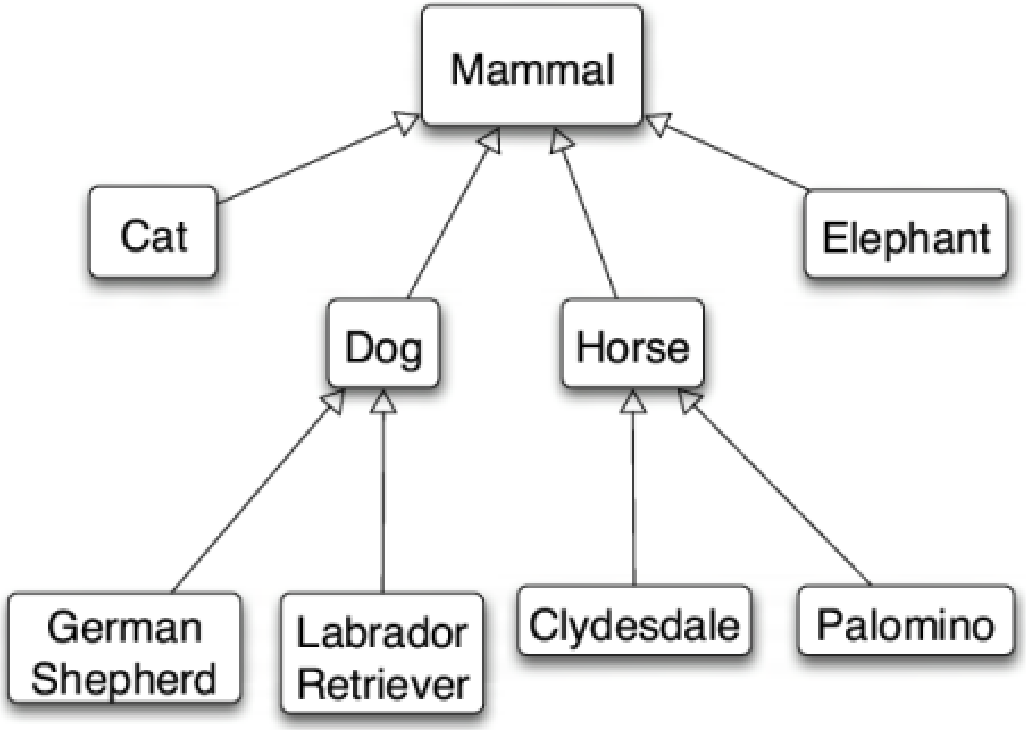
```
// Normal constructor
Date date1(2022, 11, 8);
// Default constructor
Date date3;
// Copy assignment
date3 = date1;
```

Outline

- Friend function
- Operator overloading
- Inheritance
- Polymorphism

What is Inheritance

- **is-a relationship**: A hierarchical connection where one category can be treated as a specialized version of another.
  - every rectangle *is a* shape
  - every lion *is an* animal
  - every lawyer *is an* employee
- **class hierarchy**: A set of data types connected by is-a relationships that **can share common code**.
  - **Re-use**



Basic Concepts

- **Inheritance**: A way to create new classes by extending existing classes

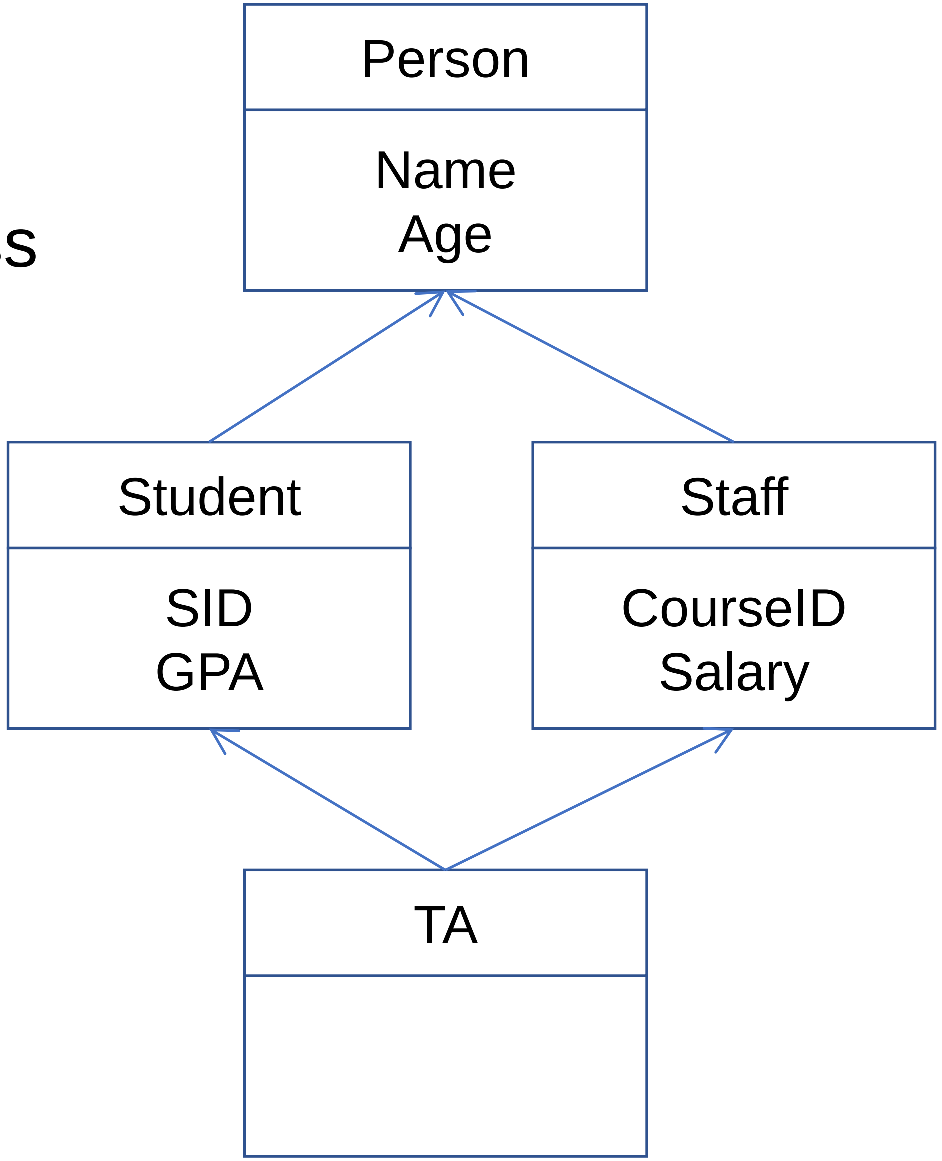
- **Base class**: Parent class that is being extended

- **Derived class**: Child class that inherits from base class(es)
  - A derived class gets a copy of every fields and methods from base class(es).
    - Note: gets a copy does NOT mean can access (details later)
  - A derived class can add its own behavior, and/or change inherited behavior

Basic Concepts

- Multiple inheritance: When one derived class has multiple base classes

- Forbidden in many object-oriented languages (e.g. Java) but allowed in C++.
- Convenient because it allows code sharing from multiple sources.
- Can be confusing or buggy, e.g. when both base classes define a member with the same name.



Syntax

```
class Parent { … };
class Child : AccessSpecifier Parent { … };

class ParentA { … };
class ParentB { … };
class Child : AccessSpecifier ParentA, AccessSpecifier ParentB { … };
```

For example:

```
class TA : public Student, public Staff { … };
```

Inheritance VS Composition

Inheritance: "is a" Composition: "has a"

```
class Engine {};          // The Engine class

class Automobile {};      // Automobile, a parent to Car class.

// Car is an Automobile, so Car class derives from Automobile class.

class Car : public Automobile {

// Car has an Engine: Car class has an instance of Engine class as its member.

    private Engine engine;

}
```

Inheritance and Access

How inherited base class members appear in derived class

Base class members

```
class Parent {
  private:   x;
  protected: y;
  public:    z;
};
```

class Child : public Parent {…}

```
class Child {
  // x is inaccessible
  protected: y;
  public:    z;
};
```

class Child : protected Parent {…}

```
class Child {
  // x is inaccessible
  protected: y;
  protected: z;
};
```

class Child : private Parent {…}

```
class Child {
  // x is inaccessible
  private:   y;
  private:   z;
};
```

Public Inheritance: Example

```cpp
class A {
private:
    int x;
protected:
    int y;
public:
    int z;
};
```

```cpp
class B : public A {
public:
    void print() {
        cout << z;  //  allowed
        y = 0;      //  allowed
        cout << x;  //  NOT allowed
    }
};
int main() {
    B obj;
    obj.y = 0;     //  NOT allowed, y is protected in B
    obj.z = 0;     //  allowed, z is public in B
    obj.print();   //  allowed, print is public in B
    return 0;
}
```

Protected Inheritance: Example

```
class A {
private:
    int x;
protected:
    int y;
public:
    int z;
};
```

```
class B : protected A {
public:
    void print() {
        cout << z;  //  allowed
        y = 0;      //  allowed
        cout << x;  //  NOT allowed
    }
};
int main() {
    B obj;
    obj.y = 0;     //  NOT allowed, y is protected in B
    obj.z = 0;     //  NOT allowed, z is protected in B
    obj.print();  //  allowed, print is public in B
    return 0;
}
```

Private Inheritance: Example

```
class A {
private:
    int x;
protected:
    int y;
public:
    int z;
};
```

```
class B : private A {
public:
    void print() {
        cout << z;  //  allowed
        y = 0;      //  allowed
        cout << x;  //  NOT allowed
    }
};
int main() {
    B obj;
    obj.y = 0;     //  NOT allowed, y is private in B
    obj.z = 0;     //  NOT allowed, z is private in B
    obj.print();   //  allowed, print is public in B
    return 0;
}
```

Constructors in Inheritance

- Derived classes can have their own constructors

- When an object of a derived class is created, the base class's *default constructor* is executed first *at the beginning* of derived class's constructor, followed by executing the derived class's constructor

```
class A {
public:
    A() { cout << "A's default constructor\n"; }
};
class B : public A {
public:
    B() {
        cout << "B's constructor\n";
    }
};
int main() {
    B b;
}
```

Constructors in Inheritance

- Derived classes can have their own constructors

- When an object of a derived class is created, the base class's *default constructor* is executed first *at the beginning* of derived class's constructor, followed by executing the derived class's constructor

```
class A {
public:
   A() { cout << "A's default constructor\n"; }
   A(int a) {
      cout << "A's non-default constructor\n";
   }
};
class B : public A {
public:
   B() {
      cout << "calling A(2310) in B()\n"; A(2310);
      cout << "calling A() in B()\n";     A();
      cout << "B's constructor\n";
   }
};
int main() {
   B b;
}
```

Passing Arguments to Constructors

```cpp
class Student {
protected:
    int sid;
public:
    Student(int sid=0) : sid(sid) {}
    int getSid() { return sid; }
};
class TA: public Student {
protected:
    int courseid;
public:
    TA(int courseid =0) : courseid(courseid) {}
    int getCourseid() { return courseid; }
};
```

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

int main() {
    Student alice(12345);
    cout << alice.getSid() << endl;

    TA bob(2311);
    cout << bob.getSid() << ": ";
    cout << bob.getCourseid() << endl;

    return 0;
}
```

*How to pass parameters to **base** constructor?*

Passing Arguments to Constructors

```
class A {
public:
  // Constructor for class A
  A(Type1 arg1, Type2 arg2, ...) {
    // Initialization for A
  }
};
```

- augment the parameter list of child constructor to include parent constructor parameters in the beginning

- pass parent constructor arguments in initializer list first

```
class B : public A {
public:
  // Constructor for class B that includes A's
  // constructor parameters followed by B's own parameters
  B(Type1 arg1, Type2 arg2, ..., TypeX argX, TypeY argY, 
...)
  : A(arg1, arg2, ...),  // Pass A's constructor arguments
    memberX(argX),       // Initialize B's own members
    memberY(argY) {
    // Additional initialization for B
  }
private:
  TypeX memberX;
  TypeY memberY;
  // ...
};
```

Passing Arguments to Constructors

```
class Student {
protected: int sid;
public:    Student(int sid=0) : sid(sid) {}
           int getSid() { return sid; }
};
class TA: public Student {
protected: int courseid;
public:    TA(int sid=0, int courseid=0) : Student(sid), courseid(courseid) {}
           int getCourseid() { return courseid; }
};
int main() {
    int sid=12345, courseid=2311;
    TA bob(sid, courseid);
    cout << bob.getSid() << ": " << bob.getCourseid() << endl;
    return 0;
}
```

Destructors in Inheritance

- Derived classes can have their own destructors
- When an object of a derived class is destroyed, the derived class's destructor is executed first, followed by the base class's destructor

```
class A {
public:
    ~A() { cout << "A's destructor\n"; }
};
class B : public A {
public:
    ~B() { cout << "B's destructor\n"; }
};
int main() {
    B* b = new B();
    delete b;
    return 0;
}
```

Outline

- Friend function
- Operator overloading
- Inheritance
- Polymorphism

Type Casting in Class Inheritance

```
class Animal {
    …
};
class Human : public Animal {
    …
};
// Only down-casting to subtype is allowed in class type conversion
// You can say a human is an animal, but not vice versa
Animal *a = new Human();     // legal
Human *b = new Animal();     // illegal
```

Static Type vs Dynamic Type

- *Static type*: the declared type; compilation-time determined
- *Dynamic type*: the actual type assigned; determined at program runtime

```
class Animal {
    …
};

class Human : public Animal {
    …
};

class Dog : public Animal {
    …
};
```

```
int main() {
  Human *human = new Human();
  Dog *dog = new Dog();

  Animal *a;     //  the static type of a is Animal
  a = human;     //  the dynamic type of a is Human
  a = dog;       //  the dynamic type of a is Dog

  delete human;
  delete dog;
  return 0;
}
```

Override

- To re-implement a base class's member function by writing a new version of that function (with the same function prototype) in a derived class

```
class Shape {
public:
    void print() { cout << "I am a shape\n"; }
};
class Circle: public Shape {
private:
    double radius;
public:
    Circle(double radius=0):radius(radius) {};
    void print() { cout << "I am a circle and my radius is " << radius << "\n"; }
};
```

Override vs Overload

- Overload

```
double sum(double, double, double);
double sum(double, double);
```

- Override

```
void Animal::makeSound();
void Human::makeSound();
void Dog::makeSound();
```

Polymorphism

- Polymorphism means "many forms"

- In inherited classes, the same function behaves differently depending on types

```
void Animal::makeSound();
void Human::makeSound();
void Duck::makeSound();
void Dog::makeSound();
void Cat::makeSound();
```

Polymorphism: Static Binding

- The called function is determined by *static type*

```
class Animal {
  public:
  void sayHi() {
    cout << "…\n";
  }
};
```


```
class Human : public Animal {
  public:
  void sayHi() {
    cout << "hi\n";
  }
};
```

```
class Dog : public Animal {
  public:
  void sayHi() {
    cout << "wow wow\n";
  }
};
```

```
int main() {
  Human *human = new Human();
  Dog *dog = new Dog();
  Animal *a;

  // the static type of a is Animal
  a = human;
  a->sayHi();  // will print "…"
  a = dog;
  a->sayHi();  // will print "…"

  delete human;
  delete dog;
  return 0;
}
```

Polymorphism: Dynamic Binding

- We want the called function to be determined by *dynamic type*

```
int main() {
  Human *human = new Human();
  Dog *dog = new Dog();

  Animal *a;          // the static type of a is Animal
  a = human;          // the dynamic type of a is Human
  a->sayHi();         // we want it to print "Hi"
  a = dog;            // the dynamic type of a is Dog
  a->sayHi();         // we want it to print "wow wow"

  delete human;
  delete dog;
  return 0;
}
```

Dynamic Binding: Virtual Function

- A *virtual function* is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class
- Allows dynamic binding at runtime

To achieve run-time polymorphism in C++ requires BOTH the following two:

1. The call should be made using a **pointer** or a **reference** to the Base class.
2. Declaring a member function in the **base** class to be **virtual** instructs the compiler to generate code that guarantees dynamic binding

Important: If **ANY** of the above two conditions is not met, then we will have static binding!!!

Polymorphism: Dynamic Binding

```cpp
class Base {
public:
  virtual void print() {
    cout << "print base\n";
  }
  void show() {
    cout << "show base\n";
  }
};
```

```cpp
class Derived : public Base {
public:
  void print() {
    cout << "print derived\n";
  }
  void show() {
    cout << "show derived\n";
  }
};
```

```cpp
int main() {
  Base *base;
  Derived *derived = new Derived();

  base = derived;
  base->print(); //  dynamic binding
                 //  will print "print derived"

  base->show();  //  static binding
                 //  will print "show base"

  delete derived;
  return 0;
}
```

Polymorphism: Dynamic Binding

```cpp
class Base {
public:
  virtual void print() {
    cout << "print base\n";
  }
  void show() {
    cout << "show base\n";
  }
};
```

```cpp
class Derived : public Base {
public:
  void print() {
    cout << "print derived\n";
  }
  void show() {
    cout << "show derived\n";
  }
};
```

```cpp
int main() {
  Derived derived;
  Base &base = derived;


  base.print();  // dynamic binding
                 // will print "print derived"

  base.show();   // static binding
                 // will print "show base"


  return 0;
}
```

Polymorphism: Dynamic Binding

```cpp
class Base {
public:
  virtual void print() {
    cout << "print base\n";
  }
  void show() {
    cout << "show base\n";
  }
};
```

```cpp
class Derived : public Base {
public:
  void print() {
    cout << "print derived\n";
  }
  void show() {
    cout << "show derived\n";
  }
};
```

```cpp
int main() {
  Derived derived;
  Base base = derived;


  base.print();  // static binding
                 // will print "print base"

  base.show();   // static binding
                 // will print "show base"


  return 0;
}
```

Virtual Destructor

```
class Base {
public:
    Base(){ cout << "Base Constructor Called\n"; }
    virtual ~Base(){ cout << "Base Destructor called\n"; }
};
class Derived : public Base {
public:
    Derived(){ cout << "Derived constructor called\n"; }
    ~Derived(){ cout << "Derived destructor called\n"; }
};
```

```
Base *b = new Derived();
delete b;
```

```
class Base{
public:
    virtual ~Base(){}
    virtual void print(){ cout << "print base\n" << endl;}
};
```

- When using class inheritance, deleting an object through a pointer to the base class without a virtual destructor will result in undefined behavior.
- Always declare a virtual destructor in base classes if you have any virtual functions (dynamic binding) to ensure proper cleanup.

Pure Virtual Functions

- Pure virtual function: a virtual member function that MUST be overridden in a derived class that has objects
- Why use pure virtual function?
  - Make base class an abstract base class, which represents abstractions.
  - Abstract base class contains at least one pure virtual function:

We can not create objects of a type that is an abstract base class.

- Syntax:

```
virtual void pureFunc() = 0;
```

- The part `= 0` indicates a pure virtual function
  - it may appear only on the declaration of the base class

Example

```
class Shape{
public:
    virtual ~Shape(){};
    virtual double getArea() = 0;
};
class Circle: public Shape{
private:
    int radius;
public:
    Circle(int radius=0):radius(radius){};
    Circle(const Circle& c){ radius = c.radius; }
    void setRadius(int radius){ this->radius = radius; }
    double getArea(){ return 3.1416*radius*radius;}
};
```

```
int main() {
    Shape* shape = new Circle(3);
    // run-time polymorphism
    cout << shape->getArea() << endl;
    delete shape;
    Circle circle;
    Shape& shape2 = circle;
    // shape2.setRadius(4);
    // run-time polymorphism
    cout << shape2.getArea() << endl;
    return 0;
}
```

Exercise

What is printed?

(A)“Mammal”

(B)“Cat”

(C)“Siamese”

(D) Gives an error (identify compiler or crash)

(E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};
class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};
class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

 Siamese * s = new Siamese();
 s->printString();
```

Exercise

What is printed?

(A)“Mammal”

(B)“Cat”

(C)“Siamese”

(D) Gives an error (identify compiler or crash)

(E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};
class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};
class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

Siamese * s = new Mammal();
s->printString();
```

Exercise

What is printed?

(A)“Mammal”

(B)“Cat”

(C)“Siamese”

(D) Gives an error (identify compiler or crash)

(E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};
class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};
class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

 Mammal * s = new Mammal();
 s->printString();
```

Exercise

What is printed?

(A)“Mammal”

(B)“Cat”

(C)“Siamese”

(D) Gives an error (identify compiler or crash)

(E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};
class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};
class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

 Mammal * s = new Siamese();
 s->printString();
```

Exercise

What is printed?

(A)“Mammal”

(B)“scraaaatch”

(C)“Siamese”

(D) Gives an error (identify compiler or crash)

(E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};
class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};
class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

 Mammal * s = new Siamese();
 s->scratchCouch();
```

Exercise

What is printed?

(A)“rawr”

(B)“meow”

(C)“Siamese”

(D) Gives an error (identify compiler or crash)

(E) Other/none/more

```
class Mammal {
public:
    virtual void makeSound() = 0;
    void printString() { cout << "Mammal" << endl; }
};
class Cat : public Mammal {
public:
    virtual void makeSound() { cout << "rawr" << endl; }
    void printString() { cout << "cat" << endl; }
};
class Siamese : public Cat {
public:
    virtual void makeSound() { cout << "meow" << endl; }
    void printString() { cout << "Siamese" << endl; }
    virtual void scratchCouch() { cout << "scraaaatch" << endl; }
};

 Mammal * s = new Siamese();
 s->makeSound();
```