

## What is Inheritance

- A central part of OOP, actually one of its 3 pillars:
  - Encapsulation
  - Inheritance
  - Polymorphism
- It's all about code reuse.
- As opposed to ~~code duplication~~.

## But, what is inheritance ?

- The ability of other classes to use (as opposed to invoke) the code of other classes w/o copying it.
- Whenever you find yourselves copying/pasting code from other classes, you need inheritance.

## When use inheritance? (1)

- When we want to write a new class B that has similar functionality to an existing class A.
  - B is different from A in a few aspects, e.g. invocation a some functions.
  - B upgrades A by adding new functionality, but A has still uses.
  - B is a special case of A.
  - We do not want to change A, because
  - We need it as is.
  - We do not want to touch it. (why?)
  - We can not touch it. (why?)

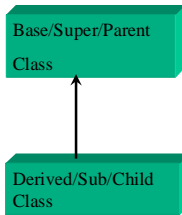
## When use inheritance (2)

- We have to write a set of related classes, e.g.
  - The classes have common functionality.
  - There is some hierarchy between the classes.
  - We want to treat different classes, similarly (Polymorphism).

## When use inheritance? (3)

- Technical/Commercial/Licensing limitations:
  - We do not have the implementation (the .cpp file) of the class, but we want to extend it.
  - The licensing policy allows us to distribute the class, but not its source code.

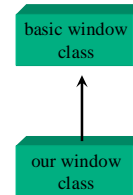
## Basic Concepts



- The Derived Class **extends** the Base class.
- The Derived Class **inherits** from the Base class.

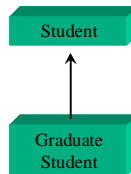
## Example

Base class is designed for inheritance:



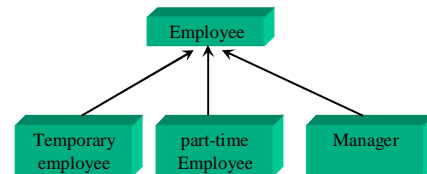
## Example

Inheritance from a usable class:



## Example

Classes having much in common:



## How do we do it? (syntax)

```
class base_class_name {  
  // class code here  
};  
  
class derived_class_name: public base_class_name {  
  // class code here  
};
```

- It is almost always public. We will discuss the other cases later.
- Java does not support the other cases.

## What does it mean? (semantics)

- The instance of the sub class, gets (inherits) automatically all the data members and member functions of the super class.
- Additionally it has its own data members and member functions.
- Therefore `sizeof(Derived) >= sizeof(Base)`

## Hiding of super class members

- A sub-class can override members of the super class by redefining them.
- We will see such an example soon.

## A comment for Java programmers

- There is no such class "Object" in C++
- If a class does not extend any class, than it is the top of its hierarchy.

## Member Access Modes

- We already know 2 member access modes:
  - private:
    - A private member\* is accessible only by the code of its own class.
  - public:
    - A public member\* is accessible everywhere in the code.

\* A data member or a member function

## Member Access Modes-cont'd

- There is a third access mode, namely
  - protected
    - A protected member is accessible only:
      - by the code of its own class, and
      - by the code of inheriting classes.

## A warning about protected

- Defining protected data members is a deviation from the Encapsulation principle.
- But, it sometimes makes the code more efficient.
- Therefore protected data members can be used only if both following conditions hold:
  - The team that is writing the sub class is the team that is writing the super class.
  - Efficiency is a major issue.

## Simple Examples

## SportsCar: public Car

```
class Car{
public:
    Car(char *s="Mazda 3", uint y=2007);
    Car(const Car&);
    ...
protected:
    char *make;
    uint year;
};

class SportsCar: public Car{
public:
    SportsCar();
    SportsCar(char *name, uint y=2007, int s=250);
    ...
private:
    int maxSpeed;
};
```

Additional data member

## TaggedPoint: public Point

```
class Point
{
public:
    Point(int a=0, int b=0); // includes default c'tor
    void Init(){x=0;y=0;}
    int GetX() const {return x;}
    int GetY() const {return y;}
    void SetX(int t) {x=t;}
    void SetY(int t) {y=t;}

    const Point operator+(const Point& p) const;
    const Point operator-(const Point& p) const;
    const Point operator*(int m) const;
    const Point& operator()(int a, int b);
    friend ostream& operator<<(ostream&, const Point&);
    friend istream& operator>>(istream&, Point&);
private:
    int x,y;
};

const Point operator*(int m, const Point& p);
```

## TaggedPoint: public Point

```
class TaggedPoint: public Point {
public:
    TaggedPoint(); //default c'tor
    TaggedPoint(char *name); // default coordinates
    TaggedPoint(int a, int b); // default tag
    TaggedPoint(int a, int b, char *name);
    TaggedPoint(const TaggedPoint&); // copy c'tor
    ~TaggedPoint();
    void SetTag(char *t);
    const TaggedPoint operator+(const TaggedPoint& p) const;
    const TaggedPoint operator-(const TaggedPoint& p) const;
    const TaggedPoint operator*(int m) const;
    TaggedPoint& operator=(const TaggedPoint& p);
    const TaggedPoint& operator()(int a, int b, char *name);
    friend ostream& operator<<(ostream&, const TaggedPoint&);
    friend istream& operator>>(istream&, TaggedPoint&);
    static int GetCount(); //return count;
private:
    static int defaultTag; // counts points with default tag
    static int count; // counts existing instances of TaggedPoint
    char *tag; // the name of the point
    void CreateDefaultTag();
};

const TaggedPoint operator*(int m, const TaggedPoint& p);
```

Overloaded operators

## Student : public Person

```
class Person
{
public:
    Person( int s, const string& n = "" )
        : ssn( s ), name( n ) { }
    const string & getName() const { return name; }
    int getSsn() const { return ssn; }
    void print( ostream & out = cout ) const
        { out << ssn << " , " << name; }
private:
    int ssn; // social security number
    string name;
};

ostream & operator<<( ostream & out, const Person& p )
{
    p.print( out );
    return out;
}
```

## Student : public Person

```
class Student : public Person
{
public:
    Student( int s, const string& n = "", double g = 0.0 )
        : Person( s, n ), gpa( g )
    { }
    double getGpa() const
    { return gpa; }
    void print( ostream & out = cout ) const
    { Person::print( out ); out << " , " << gpa; }
private:
    double gpa; // grade point average
};

ostream & operator<<( ostream & out, const Student & s )
{
    s.print( out );
    return out;
}
```

On c'tors, later.

Adding functionality

Overriding functionality

## Student : public Person

```
int main()
{
    Student s( 123456789, "Jane", 4.5 );

    Person *pPerson = &s; // *pPerson and s are same object
    s.print(); // calls Student::print
    cout << endl; // calls Student::print
    pPerson->print(); // calls Person::print
    cout << endl;
    cout << s << endl; // calls Student::print
    cout << *pPerson << endl; // calls Person::print
    return 0;
}
```

## C'tors and d'tors of subclass

- When a subclass is instantiated, first a c'tor of the base class is invoked.
  - The c'tor of the subclass can invoke the c'tor of the base class in the c'tor initializer list.
  - Otherwise the default c'tor is invoked.
- When a subclass object is destroyed, the d'tor of the subclass is invoked first

## Example: TaggedPoint

```

TaggedPoint::TaggedPoint()
{
    CreateDefaultTag();
    count++;
}

TaggedPoint::TaggedPoint(char *name)
: tag(NULL)
{
    count++;
    SetTag(name);
}
    
```

No explicit invocation, default c'tor used.

## Example: TaggedPoint-cont'd

```

TaggedPoint::TaggedPoint(int a, int b)
:Point(a,b)
{
    CreateDefaultTag();
    count++;
}

TaggedPoint::TaggedPoint(int a, int b, char *name)
:Point(a,b), tag(NULL)
{
    count++;
    SetTag(name);
}
    
```

Invocation in c'tor initializers

## Student extending Person

```

class Student : public Person
{
public:
    Student(int s, const string & n = "", double g = 0.0)
    : Person(s, n), gpa(g)
    {
    }
    double getGpa() const
    { return gpa; }
    void print(ostream & out = cout) const
    : Person::print(out); out << " " << gpa; }
private:
    double gpa; // grade point average
};
    
```

Invocation of the c'tor of Person

Invocation of base class method

## Base class' method invocation

- A sub class code can access super class members by specifying the super class name as below:

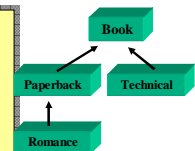
```
base_class_name::function_name(...);
```

## Base class' method invocation

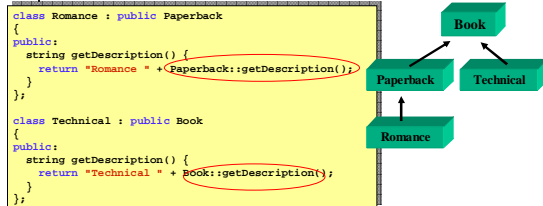
```

class Book
{
public:
    string getDescription() { return "Book"; }
};

class Paperback : public Book
{
public:
    string getDescription() {
        return "Paperback " + Book::getDescription();
    }
};
    
```



## Base class' method invocation



## The Big Three in a derived class

- The derived class defines
    - a copy c'tor
    - a d'tor, and
    - an operator=
- if there is dynamic allocation in it's code (not in it's base class' code)

## The Big Three in a derived class

- If the derived class does not define a copy c'tor (resp. an operator=), then the default implementations invoke the defaults of the base class, automatically.
- Otherwise, there is no automatic invocation, the derived class should make the invocations explicitly.
- The d'tor of the base class is invoked automatically in both cases.

## TaggedPoint – copy c'tor

```

// copy c'tor copies the coordinates,
// but sets a new default tag
TaggedPoint::TaggedPoint(const TaggedPoint& p)
:Point((Point)p)
{
    CreateDefaultTag();
    count++;
}
    
```

W/o this invocation, the default c'tor (not the copy c'tor) of Point will be invoked

## Car and SportsCar

```

class Car{
public:
    Car(char *s="Mazda 3", uint ys=2007);
    Car(const Car&);
    ...
protected:
    char *make;
    uint year;
};

class SportsCar: public Car{
public:
    SportsCar();
    SportsCar(char *name,
               uint ys=2007, int s=250);
    ...
private:
    int maxSpeed;
};
    
```

- Car needs a copy c'tor
- No copy c'tor in SportsCar, therefore there is a default copy c'tor
- It invokes the copy c'tor of Car

## Copy assignment operators

```

TaggedPoint& TaggedPoint::operator=(const TaggedPoint& p)
{
    if (&p == this)
        return *this;
    Point::operator()(p.GetX(),p.GetY());
    ...
    return *this;
}
    
```

## Modifying base class functionality

- We want a BoundVect class that it is essentially same as Vect class,
- But the access is limited to indices between some lower bound and some upper bound.
- BoundVect is same as Vect except the indexing operators

## BoundVect – modifying functionality

```
class BoundVect: public Vect
{
public:
    BoundVect(); //create a size 10 array
    BoundVect(int lBnd, int uBnd); //create an array between two
                                //bounds
    BoundVect(int a[], int lBnd, int uBnd); //initialization by
                                //array
    int& operator[](int i); //obtain range checked element
    const int& operator[](int i) const; //obtain range checked
                                //element
private:
    int lowerBound, upperBound;
};
```

## BoundVect – c'tors

```
// default ctor
BoundVect::BoundVect()
:Vect(10), lowerBound(0), upperBound(9)
{}

// initialization by bounds
BoundVect::BoundVect(int lBnd, int uBnd)
:Vect(uBnd-lBnd+1), lowerBound(lBnd), upperBound(uBnd)
{}

// initialization by array
BoundVect::BoundVect(int a[], int lBnd, int uBnd)
:Vect(a, uBnd-lBnd+1), lowerBound(lBnd), upperBound(uBnd)
{}
```

## BoundVect: indexing operators

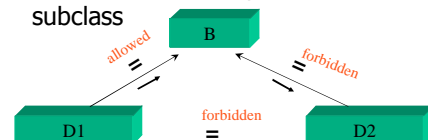
```
// implementation of operator[]:
inline int& BoundVect::operator[](int i)
{
    if(i<lowerBound || i>upperBound)
    {
        cerr<<"index out of bounds: "<<i<<"\n";
        exit(1);
    }
    Vect *v = (Vect *) this;
    return v[i-lowerBound];
}
```

## Another alternative

```
// implementation of operator[]:
int& BoundVect::operator[](int i)
{
    if(i<lowerBound || i>upperBound)
    {
        cerr<<"index out of bounds: "<<i<<"\n";
        exit(1);
    }
    return Vect::operator[](i-lowerBound);
}
```

## Assignment within inheritance hierarchy

- A subclass instance can be assigned to a superclass variable, but not vice versa
- By default, all the relevant data members will be copied from the subclass



## Inheritance access mode

- All the inheritances we saw were public.
- It can be (rarely) private.
- protected inheritance is almost useless.

## Inheritance access mode-cont'd

- The inheritance access mode does not affect the sub class, it affects only client programs.
  - Subclasses of the subclass.
  - Client code invoking the subclass' methods.
- As explained by the following example

```
class B{
public:
    void f();
    //...
};

class A: private B{
public:
    // f is not redefined
    void g() {f();}
    //...
};

int main() {
    A a;
    B b;
    //...
    b.f(); // ok
    a.f(); // error
    a.g(); // ok
    //...
}
```

OK, because f() is public in B

OK, because f() is public in B

ERROR B is inherited privately

OK, because g() is public in A

## The philosophy behind ...

- A public inheritance has the meaning of "is a".
- If A inherits from B, then "A is a B", or "an object of type A is also an object of type B". (But not vice-versa. This also a good explanation of the assignment rule).
  - A SportsCar is a Car.
  - A GraduateStudent is a Student.
  - A Student is a Person.

## The philosophy – cont'd

- The sub class is a special case of the base class.
- The base class is a generalization of the base class.

## The philosophy – cont'd

- A private inheritance has the meaning of "is implemented by".
- As the implementation should be hidden to clients, the inheritance is hidden.
- A Rectangle class may be (partly) implemented using a Line class.

## What is an "is a" inheritance?

The criterion is:

- Are all the functions of the base class relevant to the subclass too?

in other words

- Do all the functions of the base class make part of the interface of the subclass?

## Is a Square a Rectangle ?

- A priori, yes.
- But actually, the answer might be no.
  - Assume that the Rectangle has a method allowing to set the length of one of the edges, regardless of the other - setX(), setY(). We don't want them to be in the I/F.
- But, ... one may argue also otherwise, how?

## An important consequence



- The assignment rule holds only for public, or "is a" inheritance.
- As "A is a B", everywhere a B is required, but an A is supplied, A is automatically converted to B.
- Polymorphism will be available in public inheritance only.

## Containment vs. private inheritance

```
class Engine {
public:
    Engine(int n_Cylinders);
    void start(){cout<<"Engine starting..."<<endl;}
private:
    int numCilyndrs;
};

class Car {
public:
    Car() : e(8) { }
    // Initializes this Car with 8 cylinders
    void start() { e.start(); }
    // Start this Car by starting its Engine
private:
    Engine e; // Car has-a Engine
};
```

Let us use private inheritance instead:

```
class Engine {
public:
    Engine(int n_Cylinders):numCilyndrs(n_Cylinders){}
    void start(){cout<<"Engine starting..."<<endl;}
private:
    int numCilyndrs;
};

class Car : private Engine { // Car has-a Engine
public:
    Car() : Engine(8) { }
};
```

But, this client will not work:

```
Car c;
c.start();
```

## Private inheritance example

We can add a start method:

```
class Car : private Engine { // Car has-a Engine
public:
    Car() : Engine(8) { }
    void start(){Engine::start();}
};
```

or use the using keyword:

```
class Car : private Engine {
public:
    Car() : Engine(8) { }
    using Engine::start;
};
```

## When to use private inheritance?

- The subclass needs access to protected members of the base class.
- It's important that the base class instance is created before the sub class instance.
- Containment is does not make sense, conceptually.