

Inheritance: Generalization / Specialization

- Other terms for the same thing:
 - Inheritance
 - General case / Special case
 - Super class / Sub class
 - Base class / Derived class
 - Base class / Inherited class
 - "is" relation
 - The object of derived class is an object of base class

Example (Inheritance)

A circle is a special case of a point.
A circle inherits a point.

Base class

Derived class

```
class Circle: public Point {
public:
    Circle(float x0=0, float y0=0, float r0=0);
    Circle(const Point &cp0, float r0=0);
    Circle(const Circle &c);
private:
    float radius;
};
```

See a separate example later (page 10)

Constructors for derived class

- There is no data member of type Point in the class definition of Circle
 - Each circle instance in memory still contains the point object with all data members of Point
- The “problem” is, how to construct the base class data members “inside” the derived class
- The solution is that the constructor of the base class is called in the initializer list of the derived class constructor

Component:

Circle:

- cp (Point)
- radius

Inheritance:

Circle:

- Point:
 - x
 - y
- radius

Constructors for derived class

- The solution is that the constructor of the base class is called in the initializer list of the derived class constructor
 - So far we have seen the initializer list in a format
`datamember1 (initvalue1), datamember2 (params), ...`
- Now the point “inside” the circle has no name at all
 - The format we now have to use in the initialization list to construct the base class part in the derived class is
`BaseClassName (parameters)`
 - If the base class constructor is not called explicitly in the constructor of derived class, the compiler generates the call of the default constructor of base class
- When an instance of derived class is constructed the order is as follows:
 - First the base class part of the object is constructed
 - Then the new parts of the derived class are constructed

Example: Constructors of derived class

Example. (Circle is inherited from point)

```
class Circle: public Point {
public:
    Circle(float x0=0, float y0=0, float r0=0);
    Circle(const Point &cp0, float r0=0);
    Circle(const Circle &c);
    float area() const;
private:
    float radius;
};
```

Constructor of Point in the
initialization list

Constructor implementations:

```
Circle::Circle(float x0, float y0, float r0): Point(x0, y0) {
    radius = r0;    // But we cannot make assignment x = x0;
}
Circle::Circle(const Point &cp0, float r0 ): Point(cp0) {
    radius = r0;
}
//Copy constructor
Circle::Circle(const Circle &c0 ): Point(c0) {
    radius = c0.radius;
}
```

Copy constructor of Point

Note: Implementation of the copy constructor is not really needed here. The synthesized version will do exactly the same thing. It is done for the learning purpose only.

(See a separate complete example on page 10)

Component / inheritance relation: constructors

Now when we have studied how to write constructors in component relation and in inheritance relation, let's compare them together to see differences:

Constructor of main component class:

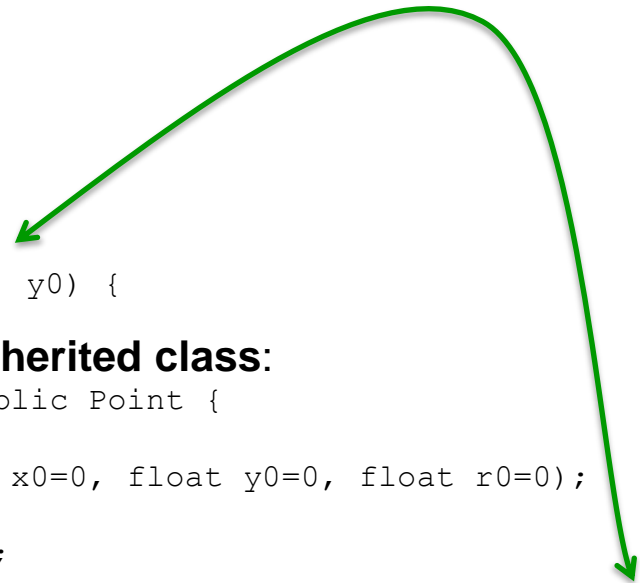
```
class Circle {
public:
    Circle(float x0=0.0, float y0=0.0, r0=0.0);
private:
    float radius;
    Point cp; //sub component
};
Circle::Circle(float x0, float y0, float r0): cp(x0, y0) {
    radius = r0;
}
```

Name of member

Constructor of inherited class:

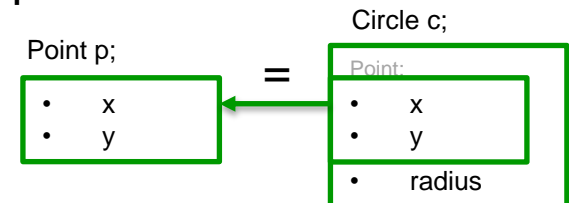
```
class Circle: public Point {
public:
    Circle(float x0=0, float y0=0, float r0=0);
private:
    float radius;
};
Circle::Circle(float x0, float y0, float r0): Point(x0, y0) {
    radius = r0;
}
```

Name of base class



Assignment compatibility

- In the initialization list of the copy constructor of circle we pass circle object to the copy constructor of the Point!
- This can be done because of assignment compatibility
 - If we have declared: `Point p; Circle c;`
 - Assignment `c = p;` is not legal, and compiler gives an error message, because point “is not” a circle
 - i.e. compiler does not know what values should be given for those circle data fields that don’t exist in point
 - On the other hand, the assignment `p = c;`
 - is OK and legal, because circle “is ” a point
 - Data member radius is dropped out in the assignment above, because p has no such data member
 - If you try to calculate the area after the assignment in the form `p.area()`, you get an error message, because point does not have a method area (or data member radius)



Assignment compatibility

- Because circle is a point, everything that we can do with a point can be done with circle
- All the following ways to call the function distance are legal (p1 and p2 are points and c1 and c2 are circles, and distance is a member function of Point that calculates the distance between two points):

```
p1.distance(p2);  
p1.distance(c2); // assignment compatibility  
c1.distance(p2); // Circle inherits distance  
c1.distance(c2); // Both things are needed
```

Provided that assignment is not overloaded and
there is no suitable conversion constructor

How features of base class can be used 1

- How to use properties of base class in derived class:
 - The inherited feature can be used as such (methods `getx`, `gety` and `move` of class `Point` are valid and useful for circles as such) (See the example on the next page)
 - Overriding member functions
 - We can add additional functionality to the inherited method in the derived class (methods `read` and `distance` in the case of circle) (See example on the page 10)
 - We can replace the inherited method totally with a new and different method
 - We can add a totally new methods (and data members) to the inherited class (data member `radius` and member function `area` and `enlarge`)

How features of base class can be used 2

- The inherited feature can be used as such

- If

```
float getx();
```

```
void move(float delta_x, float delta_y);
```

are member functions of Point, and Circle is inherited from the Point then:

```
Circle c(1.0, 2.0, 10);
```

```
cout << c.getx();    // getx returns x coordinate  
                      // of the center point
```

```
c.move(0.5, 0.5);    // moves the circle!
```

- Functions getx and move need no declaration in the class definition of Circle
- A complete example follows on page 11

How features of base class can be used 3

- We can add additional functionality to the inherited method in the derived class or we can replace the inherited method totally with a new and different method without using any functionality from the base class
- Let's assume that
`virtual float distance(const Point &p2) const;`
is a member of `Point` and we inherit `Circle` from the `Point`. We need a distance function for the class too, but the formula is a bit different. That's why we have to modify the behavior of function distance in the class `Circle`. How do we do this?

- Function prototype inside class definition of `Circle`:

```
float distance(const Circle &c2) const override;
```

- Implementation of this function

```
float Circle::distance(const Circle &c2) const {  
    float dist;  
    dist = Point::distance(c2); // call base class function  
    dist = dist - radius - c2.radius; // add some  
    return dist < 0 ? 0 : dist; // new functionality  
}
```

Complete example (1/4)

```
// Class Circle is derived (inherited) from class
// Point. This means that Circle is a specialised
// point. It has all properties of point and
// some more
#include <iostream>
#include <math.h>
#include <stdio.h>
```

We can repeat virtual in the derived class but it is not mandatory

We must have a virtual destructor if we want this class to be base class of other classes

```
//definition of class Point
class Point {
public:
    //constructor
    Point(float xcoord=0.0, float ycoord=0.0);
    //copy constructor
    Point(const Point &p);
    //destructor
    virtual ~Point() = default;
    virtual void read(const char *prompt);
    float getx() const;
    float gety() const;
    void move(float deltax, float deltay);
    virtual float distance(const Point &p2) const;
private:
    float x;
    float y;
};
```

Member functions that are to be overridden in derived class must be declared virtual

```
//definition of class Circle
class Circle: public Point {
public:
    Circle(float cpx=0.0, float cpy=0.0, float r=1.0);
    Circle(const Point &cp, float r); //constructor 2
    Circle(const Circle &c); //copy constructor
    virtual void read(const char *prompt) override;
    float getRadius() const;
    float distance(const Circle &p) const override;
    float area() const;
private:
    float radius;
};
```

C++11 allows us to explicitly specify our intention

```
//prototypes of input and output operators
ostream &operator<<(ostream &out, const Point&p);
ostream &operator<<(ostream &out, const Circle &c);
```

Complete example (2/4)

```
//Test application program
void main(void) {
    Point  p, p3(3.0, 3.0);
    Circle c1, c2(2.0, 2.0, 20.0) , c3(p3,30);

    c1.read("Enter circle 1");;
    cout << c1 << endl;
    cout << "Center point of c1 is " << "(" <<  c1.getx() << ", " << c1.gety() << ")" << endl;
    cout << c3 << endl;
    cout << " Area of circle 3 is " << c3.area() << endl;

    cout << "\nDistance between circles is " << c1.distance(c3) << endl;

    c3.move(0.5 , 0.5);
    cout << c3 << endl;

    //c1 = p; This is not possible. It causes compilation error
    p = c1; //circle can be assigned to a point (though it makes no sense usually)
    cout << "Circle 1 as a point is " << p << endl;
    p.move(0.5, 0.5);
    cout << c1 << endl ; // c1 is not moved
    cout << "Point is moved" << p << endl ; //p is moved
    //cout << p.area(); This is not possible. It causes compilation error
}
```

Complete example (3/4)

```
// Methods of class Point
Point::Point(float xcoord, float ycoord) {
    x = xcoord;
    y = ycoord;
}
Point::Point(const Point &p) { //copy constructor
    x = p.x;
    y = p.y;
}
void Point::read(const char *prompt) {
    cout << prompt;
    cout << "Enter x:";
    cin >> x;
    cout << "Enter y:";
    cin >> y;
}
```

```
float Point::getx(void) const {
    return x;
}
float Point::gety(void) const {
    return y;
}
void Point::move(float deltax, float deltay) {
    x+=deltax;
    y+=deltay;
}
float Point::distance(const Point &p2) const {
    return sqrt((p2.x-x)*(p2.x-x) +
                (p2.y-y)*(p2.y-y));
}
```

Complete example (4/4)

```
// Methods for class Circle
Circle::Circle (float cpx, float cpy, float r0 ): Point(cpx, cpy), radius(r0) { }
Circle::Circle(const Point &cp, float r0): Point(cp), radius(r0) { }
Circle::Circle(const Circle &c): Point (c), radius(c.radius) { }
void Circle :: read(const char *prompt){
    cout << prompt;
    Point::read("\nEnter center point: ");
    cout << "Enter radius ";
    cin >> radius;
}
float Circle::getRadius() const{
    return radius;
}
float Circle::distance(const Circle &c2) const {
    float dist;
    dist = Point::distance(c2); //use functionality from the base class
    dist = dist - radius - c2.radius; // add some new functionality
    return dist < 0 ? 0 : dist ;
}
float Circle::area(void) const {
    float area;
    area = M_PI* radius * radius;
    return area;
}
// Output operator functions (They are not friends now)
ostream &operator<<(ostream &out, const Point&p) {
    out << "(" << p.getx() << "," << p.gety() << ")";
    return out;
}
ostream &operator<<(ostream &out, const Circle &c) {
    out << "\nCircle: Radius is " << c.getRadius() << " Center point is " << (Point) c; //or static_cast<Point>(c)
    return out;
}
```

Component / inheritance relation: functions

- Now when we have studied how to write member functions in component relation and in inheritance relation, let's compare them together to see differences:

- Member function of **main component class**:

```
class Circle {
public:
    virtual void read();
private:
    float r;
    Point cp; // sub component
};
void Circle::read( ) {
    cp.read(); // how to use sub
               // component function
    cin >> r;
}
```

- Overridden member function of **inherited class**:

```
class Circle: public Point {
public:
    virtual void read() override;
private:
    float r;
};
void Circle::read( ) {
    Point::read(); // how to use base
                  // class function
    cin >> r;
}
```

Base class name and scope operator are needed only because we are overriding the function

Access specifier protected

- When the meaning of access specifiers is considered we have two different viewpoints:
 1. Using an existing class as such
 2. Using an existing class by inheriting from it
- If the access specifier for a member is private, there is no way to access that member in both cases above
- If the access specifier for a member is public, it is accessible in both of cases above
- So far we have seen only access specifiers private and public
- There is also a third access specifier **protected**
 - The effect of access specifier protected is as follows:
 - a) It is not allowed to refer to the member in the application (case 1 above). In that sense it is same than private
 - b) It is allowed to refer to the member in member functions of that class. In that sense it is same than private
 - c) The difference (compared to private member) is that it is possible to refer to the protected member of the base class in the member function of the derived class (case 2)
- See complete example starting on the next page

Example (1 / 3)

```
/* Class Circle is derived (inherited) from class point.
   The meaning of access specifier protected is illustrated in bold font */
#include <iostream.h>
#include <math.h>
//definition of class Point
class Point {
public:
    Point(float xcoord=0.0, float ycoord=0.0); //constructor
    Point(const Point &p); //copy constructor
    virtual void read(const char *prompt);
    float getx() const;
    float gety()const;
protected: // This is the only difference in the class definition
    float x; // compared to the example on the page 10.
    float y; // See the effects in the implementations of
             // member function
};
//definition of class Circle
class Circle: public Point {
public:
    Circle(float cpx=0.0, float cpy=0.0, float r=1.0); //constructor 1
    Circle(const Point &cp, float r); //constructor 2
    Circle(const Circle &c); //copy constructor
    virtual void read(const char *prompt) override;
    float getRadius() const {return radius;}
private:
    float radius;
};

//prototypes of input and output operators
ostream &operator<<(ostream &out, const Point&p);
ostream &operator<<(ostream &out, const Circle &c);
```

Example (2 / 3)

```
// Test application program
void main(void) {
    Point p2(2.0, 2.0);
    Circle c1(1.0, 1.0, 10.0), c2(p2, 20.0), c3(c2), c4;

    //cout << p2.x is not allowed here because x (and y) is protected
    //member (not public or private private member)
    //you have to use public access functions
    cout << "(" << p2.getx() << "," << p2.gety() << ")" << endl;

    c4.read("Enter circle 4");
    cout << "Circle 1 is " << c1 << endl;
    cout << "Circle 2 is " << c2 << endl;
    cout << "Circle 3 is " << c3 << endl;
    cout << "Circle 4 is " << c4 << endl;
}

// Methods of class Point are exactly as they were before

// Output operator functions (They are not friends now)
ostream &operator<<(ostream &out, const Point&p) {
    cout << "(" << p.getx() << "," << p.gety() << ")" << endl;
    return out;
}

ostream &operator<<(ostream &out, const Circle &c) {
    cout << "\nCircle: Radius is " << c.getRadius() << " Center point is " << (Point)c;
    return out;
}:
```

Example (3 / 3)

```
// Methods for class Circle
//Constructor version 1
/* Circle::Circle (float cpx, float cpy, float r ): Point(cpx, cpy), radius(r) {} */
//This is possible too, because x and y are protected
//Constructor version 2
Circle::Circle(float cpx, float cpy, float r ) {
    x = cpx; // this is now possible because
    y = cpy; // x and y are protected (not private)
    radius = r;
}
Circle::Circle(const Point &cp, float r): Point(cp) {
    //x = cp.x; // this is not allowed
    //y = cp.y; // You can access only protected members
    radius = r; // of Point that is inherited in Circle.
                // Parameter cp is independent point object
}
Circle::Circle(const Circle &c) /* : Point(c) */{
    x = c.x; // This is now allowed
    y = c.y;
    radius = c.radius;
}
void Circle::read(const char *prompt){
    cout << prompt;
    cout << "\nEnter center point: ";
    cin >> x; //you can access protected data member
    cin >> y; //of base class in a member function of derived class
    cout << "Enter radius ";
    cin >> radius;
}
```

Inheritance types

- Keywords private, protected and public are used as access specifiers
- The same keywords can also be used as a type of inheritance
- In our first example of inheritance we used public inheritance

```
class Circle: public Point { ... };
```

 Inheritance type

- This means that all public members of Point are available for the user of Circle:

```
Circle c;  
cout << c.getx(); // getx is public member of Point
```
- To be more precise it means that all members of Point still have the same access specifications in Circle
 - The instance of derived class is (is-relation) an instance of the base class, because it can be used like an instance of base class
- The inheritance type can be used to **restrict** the access rights to the members of base class in the derived class

Inheritance types

- The inheritance type can be used to **restrict** the access rights to the members of base class in the derived class
- The examples in pages 23-27 demonstrate the effects of inheritance types protected and private
 - A separate table there illustrates the effects of inheritance types to different data members from base class
- Inheritance type **public** is used to **inherit interface**
- Inheritance type **protected** and **private** are used to **inherit implementation**

Example of protected inheritance

Let's assume that class List is defined as follows:

```
class List {
public:
    List();
    bool is_empty() const;
    int number_of_items() const;
    bool retrieve_ith(int i, int *item) const;
    bool store_item(int item);
    bool find_pos(int item, int *pos) const;
    bool remove_item(int orderNo);
private:
    int count;
    int array[100];
};
```

We now want to implement class Stack. The Stack has the methods push, pop and is_empty.

Then we could use protected or private inheritance as follows:

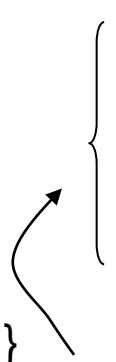
```
class Stack: protected List {
public:
    bool push(int item);
    int pop();
    bool is_empty() const;
};
```

Remark: The same goal could be achieved with letting the List be as a component of Stack

The effect of access specifiers 1

<p>Base class definition</p> <pre>class BClass { private: int a; protected: int b; public: int geta(); void f(BClass p); }</pre>	<p>Derived class definition</p> <pre>class DClass: public BClass { ... public: void f(DClass p); };</pre>
--	---

The effect of access specifiers 2

Using BClass	Using DClass
<pre> BClass bc; bc.geta(); //OK bc.a //NOT OK bc.b //NOT OK </pre>	<pre> DClass dc; dc.geta(); //OK dc.a //NOT OK dc.b //NOT OK </pre>
Implementing Bclass	Implementing Dclass
<pre> void BClass::f(BClass p) { a //OK p.a //OK b //OK p.b //OK geta(); //OK } </pre>	<pre> void DClass::f(DClass p) { { a // NOT OK p.a // NOT OK } { b // OK p.b // OK } geta(); //OK } </pre> 
Protected and private are same here	Difference between private and protected

The effect of inheritance types

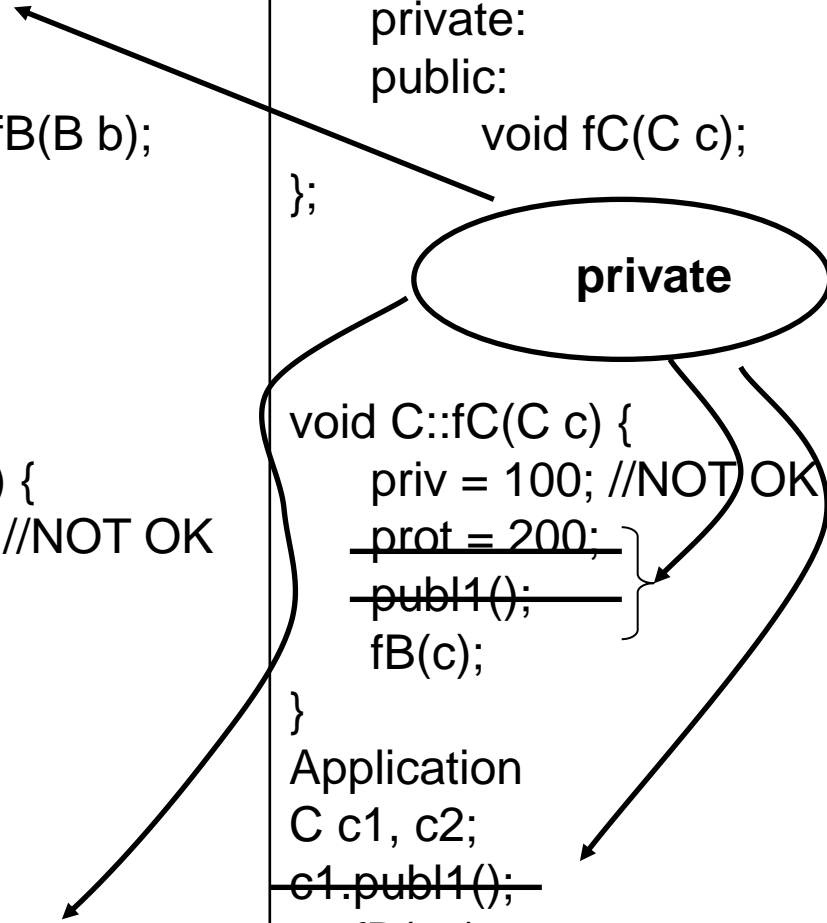
	Member of base class	Member of derived class
Inheritance type public:	Private Protected Public	No access Protected Public
Inheritance type protected:	Private Protected Public	No access Protected Protected
Inheritance type private:	Private Protected Public	No access Private Private

The effect of inheritance type protected

<p>Base class definition</p> <pre>class A { private: int priv; protected: int prot;; public: int publ1(); int publ2(); } int A::publ1() { priv = 1; prot = 2; publ2(); }</pre> <p>Application</p> <pre>A a; a.publ1();</pre>	<p>Derived class definition</p> <pre>class B : protected A { private: public: void fB(B b); }; void B::fB(B b) { priv = 10; //NOT OK prot = 20; publ1(); }</pre> <p>Application</p> <pre>B b1, b2; b1.publ1(); b2.fB(b1)</pre>	<p>Derived class definition</p> <pre>class C : public B { private: public: void fC(C c); }; void C::fC(C c) { priv = 100; //NOT OK prot = 200; publ1(); fB(c); }</pre> <p>Application</p> <pre>C c1, c2; c1.publ1(); c2.fB(c1); c1.fC(c2);</pre>
---	---	--

protected

The effect of inheritance type private

<p>Base class definition</p> <pre>class A { private: int priv; protected: int prot;; public: int publ1(); int publ2(); } int A::publ1() { priv = 1; prot = 2; publ2(); }</pre> <p>Application</p> <pre>A a; a.publ1();</pre>	<p>Derived class definition</p> <pre>class B : private A { private: public: void fB(B b); }; void B::fB(B b) { priv = 10; //NOT OK prot = 20; publ1(); }</pre> <p>Application</p> <pre>B b1, b2; b1.publ1(); b2.fB(b1)</pre>	<p>Derived class definition</p> <pre>class C : public B { private: public: void fC(C c); }; void C::fC(C c) { priv = 100; //NOT OK prot = 200; publ1(); fB(c); }</pre> <p>Application</p> <pre>C c1, c2; c1.publ1(); c2.fB(c1); c1.fC(c2);</pre> 
---	---	--