C++ Programming:
Inheritance

2019년도 2학기

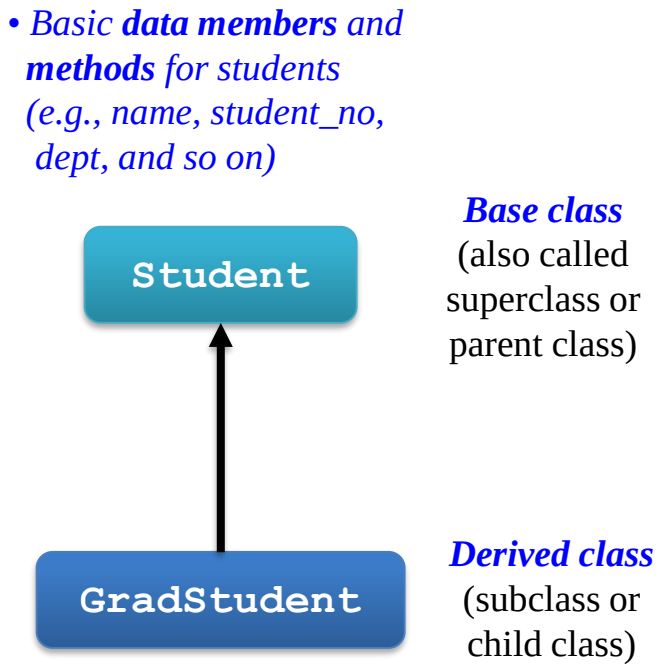
Instructor: Young-guk Ha
Dept. of Computer Science & Engineering

Contents

- Basics on inheritance
- C++ syntax for inheritance
- **`protected`** members
- Constructors and destructors in inheritance
- Multiple inheritance

Inheritance

- What is ***inheritance***?
  - The heart of OO programming
  - A mechanism to build a new class by derivation from already existing base classes
    - Single inheritance: inherits from one base class
    - Multiple inheritance: inherits from more than two base classes
- Content of derived class
  - Data members and methods inherited from the base classes
    - *Except for constructors and destructors*
  - Additional data members and methods specific to the derived class
    - Inherited methods can be overridden (cf. ***polymorphism***)



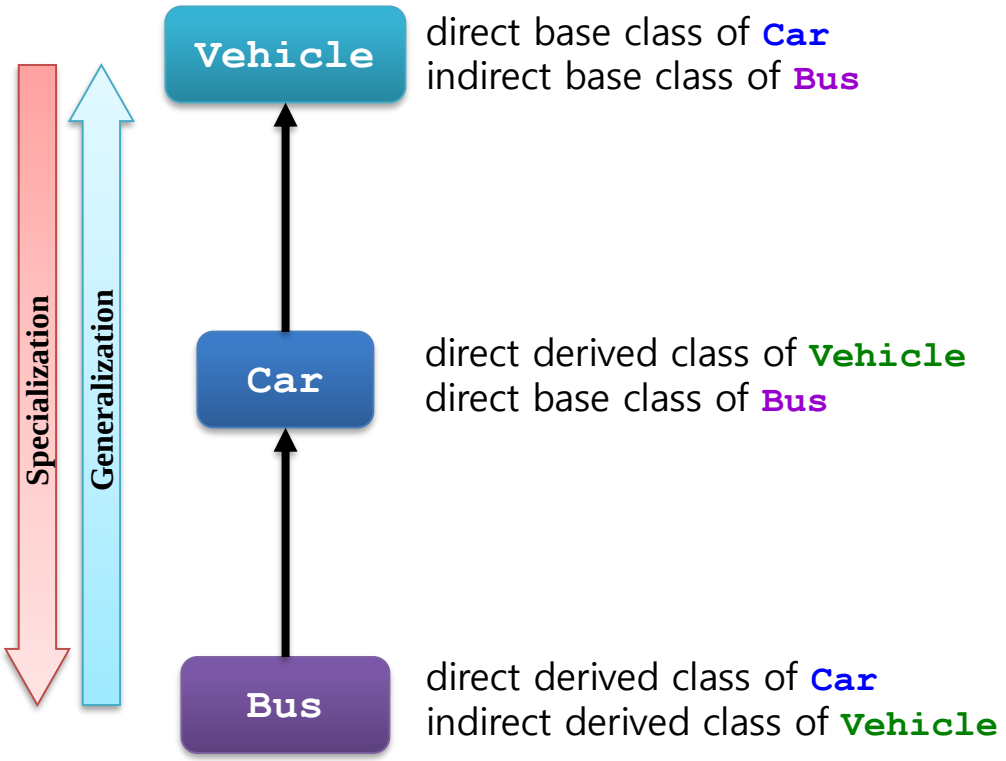
Benefits of Inheritance

- Inheritance promotes code reuse and robustness
  - The code (data and methods) in the base class is inherited to the subclass
    - Inherited code needs not be rewritten
    - Reducing the amount of code to be newly created
  - If the code from the base class is correct, it will also correct in the derived class
    - Bugs might be reduced → robustness

- Expression of class relationship
  - Provides a mechanism to naturally express "***is a***" relationships among classes that are modules of OO programing
  - E.g., a **`GradStudent`** ***is a*** **`Student`**
    - "***is a***" relationship is precisely mirrored in the code with inheritance

Class Hierarchy

- A derived class can itself serve as a base class for other classes
  - And those classes can also be base classes, …
  - This relationship is called ***class hierarchy*** (a.k.a. ***inheritance hierarchy***)
  - When we move down the hierarchy, each class is a *specialized* version of its base class
  - When we move up the hierarchy, each class is a *generalized* version of its derived class



C++ Syntax of Inheritance

- Three types of inheritance to derive class ***DC*** from class ***BC***
  - Private inheritance (default)

```
Class DC : BC {
    ...
};
```

=

```
Class DC : private BC {
    ...
};
```

❖ **private** can be specified explicitly

  - Public inheritance (most frequently used)

```
Class DC : public BC {
    ...
};
```

  - Protected inheritance

```
Class DC : protected BC {
    ...
};
```

Access Modifiers in Inheritance

- Public inheritance

In the Base Class	In the Derived Class
private	**invisible**
protected	protected
public	public

- Private (default) inheritance

In the Base Class	In the Derived Class
private	**invisible**
protected	private
public	private

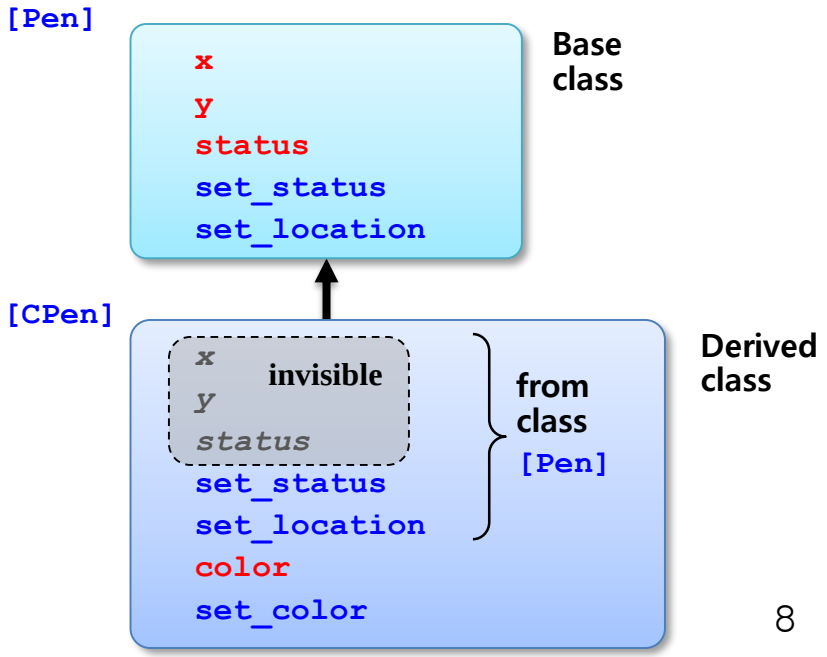
- Protected inheritance

In the Base Class	In the Derived Class
private	**invisible**
protected	protected
public	protected

Public Inheritance Example

```
class Pen {
 public:
   void set_status(bool s);
   void set_location(int x, int y);
 private:
   int x;
   int y;
   bool status;
};

class CPen : public Pen {
 public:
   void set_color(int c);
 private:
   int color;
};

CPen cp; // creates a CPen object cp

cp.set_location(100, 200);
cp.set_status(true);
cp.set_color(123)
```

- **CPen** (colored pen) class
  - Inherits all the data members and methods from class **Pen**
  - Public methods **set_status** and **set_location** of **Pen** are also public in **CPen**
  - Defines data member **color** and method **set_color**



Protected and Private Inheritance Examples

```
class BC {
 public:
   void set_x(int a) { ... }
 protected:
   int get_x() const { ... }
 private:
   int x;
};

// protected inheritance
class DC1 : protected BC {
   ...
};

// private inheritance
class DC2 : private BC {
   ...
};
```

Access status in **`protected`** inheritance

Member of BC	Access status in class BC	**Access status in DC1**
`set_x`	`public`	**`protected`**
`get_x`	`protected`	**`protected`**
`x`	`private`	***Not accessible***

Access status in **`private`** inheritance

Member of BC	Access status in class BC	**Access status in DC2**
`set_x`	`public`	**`private`**
`get_x`	`protected`	**`private`**
`x`	`private`	***Not accessible***

Accessing `private` Members of Base Classes

- Each **`private`** member in a base class is visible only in the base class
  - I.e., Even though **`private`** members are inherited by derived classes, they are **NOT** visible in the derived classes

```
class Point {
 public:
  void set_x(const int x1) { x = x1; }
  void set_y(const int y1) { y = y1; }
  int get_x() const { return x; }
  int get_y() const { return y; }
 private:
  int x;
  int y;
};

class Intense_Point : public Point {
 public:
  void set_intensity(const int i) { intens = i; }
  int get_intensity() const { return intens; }
 private:
  int intens;
};
```

- ✓ Class **`Intense_Point`**
  - It inherits data member **`x`** and **`y`** from **`Point`**, which are only visible in **`Point`**
  - One can indirectly access **`x`** and **`y`** through the public methods **`set_x`**, **`set_y`**, **`get_x`**, and **`get_y`** inherited from **`Point`**

Members of `Intense_Point`

Members	*Access status within* `Intense_Point`	*How obtained*
`x`	*Not accessible**	Inherited from class `Point`
`y`	*Not accessible***	Inherited from class `Point`
`set_x`	`public`	Inherited from class `Point`
`set_y`	`public`	Inherited from class `Point`
`get_x`	`public`	Inherited from class `Point`
`get_y`	`public`	Inherited from class `Point`
`intensity`	`private`	Defined in class `Intense_Point`
`set_intensity`	`public`	Defined in class `Intense_Point`
`get_intensity`	`Public`	Defined in class `Intense_Point`

* Indirectly accessible through method **`set_x`** and **`get_x`**
** Indirectly accessible through method **`set_y`** and **`get_y`**

Restricting Access to Inherited `public` Members

- Normally, **`public`** members in a base class would be inherited as **`public`**
  - However, sometimes access to those inherited **`public`** members from outside needs to be disabled
  - This can be done by changing the access status of the inherited members to **`private`** with using declaration

```
class BC {
 public:
   void set_x(float a) { x = a; }
 private:
   float x;
};

class DC : public BC {
 public:
   void set_y(float b) { y = b; }
 private:
   float y;
   using BC::set_x;
};
```

```
int main() {
   DC d;

   d.set_y(4.31);  // OK
   d.set_x(-8.03);
    // error: set_x is private in DC

   ...
}
```

Name Hiding

- ***Name hiding***: when a derived class adds a new method with the same name as one of inherited methods (but with different signatures), the new method hides the inherited method
  - Name hiding a kind of name conflict
  - Cf. ***method overriding***: when both the new method and the inherited method have the same signature

```
class BC {
public:
   void h(float);  // BC::h
};

class DC : public BC {
public:
   void h(char *);
     // hides h(float) from BC
};
```

```
void main() {
   DC d1;

   d1.h("Boffo!");
     // DC::h, not BC::h

   d1.h(707.7);
     // ERROR: DC::h hides BC::h
     //          (not overloading)
   d1.BC::h(707.7);
     // OK: BC::h is invoked
}
```

Indirect Inheritance

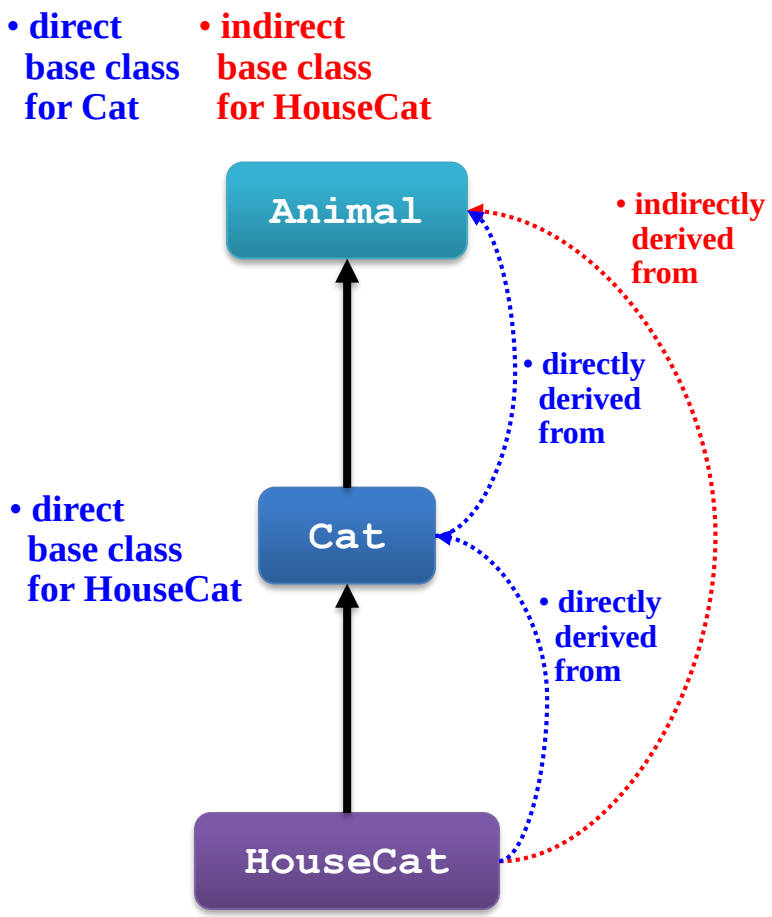
- Data members and methods may traverse several inheritance links as they are included from a base to a derived class

- Thus, inheritance may be either *direct* or *indirect*
  - Direct inheritance
    - From a direct base class to a derived class
  - Indirect inheritance
    - From an indirect base class to a derived class

❖ Note that **`public`** members remain **`public`** throughout the chain of **`public`** inheritance links

Indirect Inheritance Example

```
class Animal {
public:
   string   species;
   bool     warmBlooded;
   ...
};

// direct derived class of Animal
class Cat : public Animal {
public:
   string range[100];
   float favoritePrey[100][100];
   ...
};

// direct derived class of Cat
// indirect derived class of Animal
class HouseCat : public Cat {
public:
   string toys[10000];
   string catDoctor;
   string apparentOwner;
   ...
};
```



Protected Members

- In addition to **`public`** and **`private`** members, C++ provides **`protected`** members
  - Without inheritance, a **`protected`** member is just like a **`private`** member
    - I.e., only visible within the same class
  - In public inheritance, a **`protected`** member of a base class is also **`protected`** in its derived classes
    - I.e., also visible in the derived classes
  - Consequently, methods of directly or indirectly derived classes can access **`protected`** members from their base classes
    - Note that a **`friend`** function of a class can also access **`private`** and **`protected`** members of the class (to be discussed later)

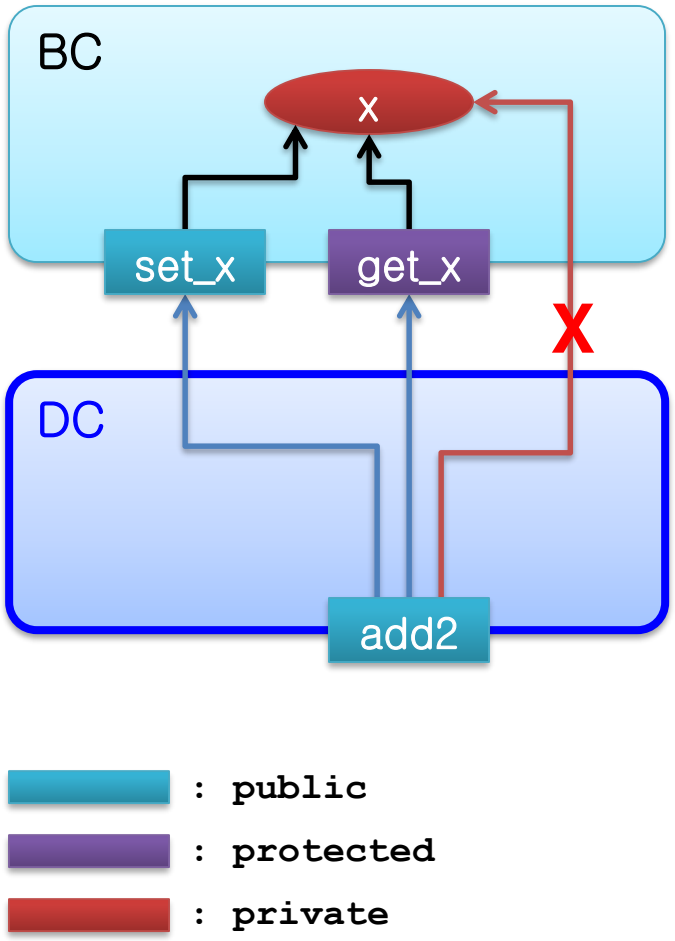
Example of `protected` Members

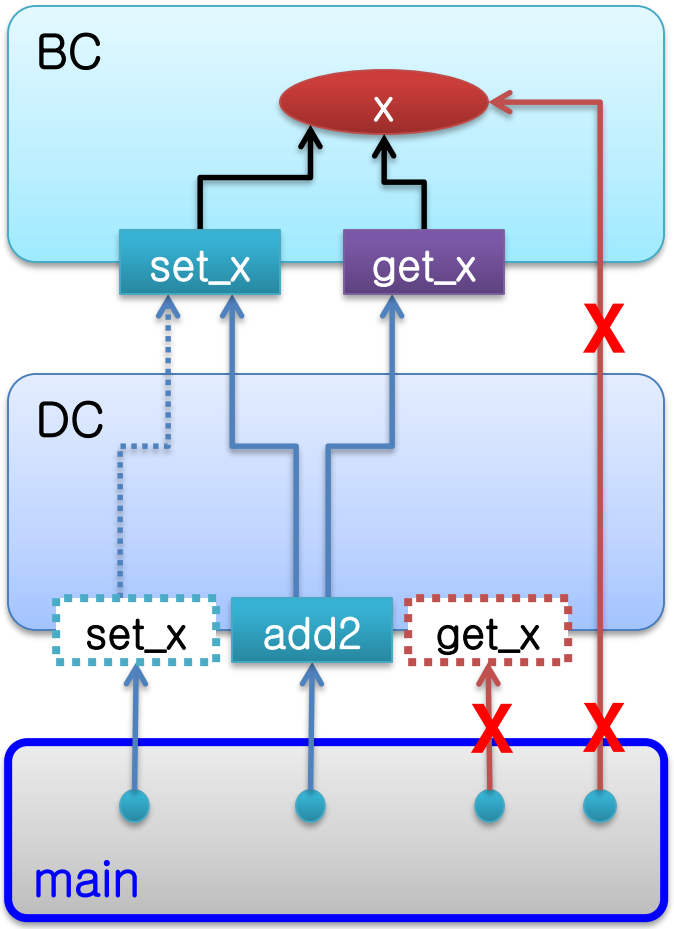
```
// base class
class BC {
 public:
   void set_x(int a) { x = a; }
 protected:
   int get_x() { return x; }
 private:
   int x;
};

class DC : public BC {
 public:
   void add2() {
     x += 2; // ERROR: x is
             //   private in BC

     int c = get_x(); // OK
     set_x(c + 2);
   }
};
```

```
int main() {
  DC d;

  d.set_x(3);
  // OK: set_x is public in DC

 cout << d.get_x() << '\n';
  // ERROR: get_x is protected in DC

 d.add2();
  // OK: add2 is public in DC
}
```

Member	Access status in class DC	Access status in main function
`set_x`	`public`	*Accessible*
`get_x`	`protected`	*Not accessible*
`x`	*Not accessible*	*Not accessible*
`add2`	`public`	*Accessible*

Example of `protected` Members (cont.)





Protected Members of Base Class Object

- A derived class …
  - can access **`protected`** members inherited from a base class,
  - but can **NOT** access **`protected`** members of an object that belongs to the base class

```
class BC {
 protected:
   int get_w() const;
   ...
};

class DC : public BC {
 public:
   void get_val() const
      { return get_w(); }
```

```
   void base_w() const {
      BC b;
      cout << b.get_w()
           << '\n';
      // ERROR: b.get_w is not
      //  visible in DC since
      //  it is a member of
      //  a BC object, not one
      //  inherited to DC
   }
};
```

Better Class Design for Protected Data Members

- In general, **protected** data members should be avoided in strict terms of information hiding
  - They can usually be made **private** and accessible by **protected** accessor methods
  - Exceptively, some complex data members would be declared **protected** instead of providing complicated accessor methods
    - E.g., a multi-dimensional array of complicated structures

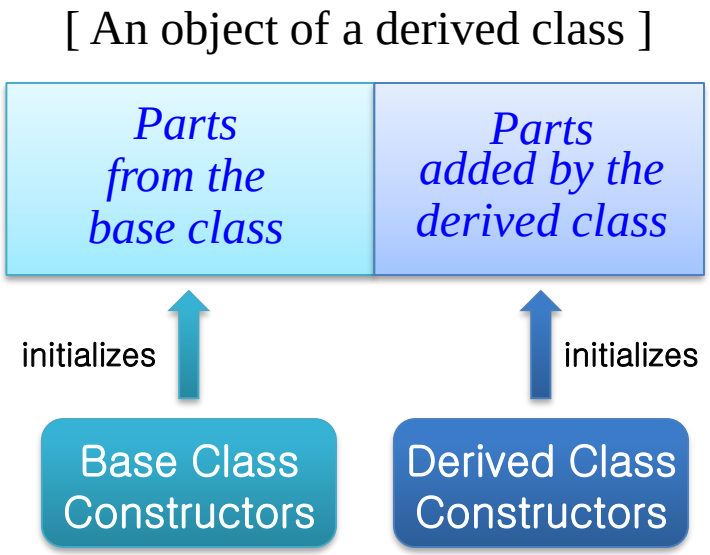
```
class BC {
    ...
   protected:
      int y;
};
```

```
class BC {
   ...
 protected:
   int get_y() const { return y; }
   void set_y(int a) { y = a; }
 private:
   int y;
};
```

Constructors under Inheritance

- A derived class is a specialization of base class
  - Basically, an object of a derived class has characteristics inherited from the base class
  - In addition, it has characteristics specific to the derived class

- Thus, a base class constructor needs to be invoked when a derived class object is created
  - The base class constructor handles initialization and other matters for the "***from base class***" part of the object
  - And the derived class constructor handles the "***added by derived class***" part of the object



Example of Derived Class Constructor Invocation

```
class Animal {
 public:
   Animal() { species = "Animal"; }
   Animal(const char* s) { species = s; }
 private:
   string species;
};

class Primate : public Animal {
 public:
   Primate() : Animal("Primate") {}
   Primate(int n) : Animal("Primate") { heart_cham = n; }
 private:
   int heart_cham;
};

Animal  slug;              // invokes Animal()
Animal  tweety("canary"); // invokes Animal(const char*)
Primate godzilla;          // Animal("Primate")→Primate()
Primate human(4);          // Animal("Primate")→Primate(int)
```

In the initialization section, this indicates to invoke **Animal** constructor before executing its body

Basic Rules for Constructors under Inheritance

[**Rule #1**] If no constructor is defined explicitly

⇒ The system provides the default constructor

[**Rule #2**] If no specific constructor of the base class is invoked in a constructor of the derived class

⇒ The default constructor of the base class is automatically invoked whenever a derived class object is created

Application of the Basic Rules When a DC Object is Created

Base Class (BC) \ Derived Class (DC)	**No constructor is defined explicitly** (system provides default constructor)	**At least one constructor is defined** (system does **NOT** provide default constructor for DC)
No constructor is defined explicitly (system provides default constructor)	① The system-provided default constructors of **BC** and **DC** are invoked automatically	② The system-provided default constructor of **BC** and appropriate **DC** constructors are invoked
At least one constructor is defined (system does **NOT** provide default constructor for BC)	③ Programmer MUST provide the default constructor for **BC** (which is automatically invoked by the system provided constructor of **DC**)	④ Appropriate **BC** and **DC** constructors are invoked

Constructor Example (1)

```
class BC {                          // base class
 public:
   BC() { x = y = -1; }             // default constructor
 protected:
   int get_x() const { return x; }
   int get_y() const { return y; }
 private:
   int x;
   int y;
};

class DC : public BC {              // derived class has no constructor
 public:
   void write() const { cout << get_x() * get_y() << '\n'; }
};

int main() {
   DC d1;                  // BC()→DC() is invoked automatically
   d1.write();             // "1" is written to the standard output
   ...
}
```

Constructor Example (2)

```
// BC has constructors, but
//   no default constructor
class BC {
 public:
   BC(int a)
        { x = a, y = 999; }
   BC(int a1, int a2)
        { x = a1; y = a2; }
 private:
   int x;
   int y;
};

// DC has a constructor
class DC : public BC {
 public:
   DC(int n) { z = n; }
        // *** ERROR
 private:
   int z;
};
```



```
// Solution 2: add default
//   constructor to BC
class BC {
 public:
   BC() {  ...  }
   BC(int a)
        { x = a, y = 999; }
   ...
};
```

```
// Solution 1: explicitly
//   invoke a BC constructor
class DC : public BC {
 public:
   DC(int n) : BC(n, n+1)
        { z = n; }
   ...
};
```

Constructor Example (3)

```
class BC {
 public:
   BC() { cout << "BC() executed ...\n"; }
 private:
   int x;
};

class DC : public BC {
 public:
   DC() : BC() { cout << "DC() executed ...\n"; }
        // legal but unnecessary
 private:
   int y;
};

int main() {
   DC d;
   ...
}
```

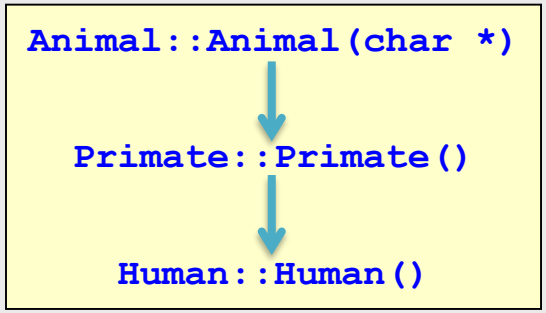
```
BC() executes...
DC() executes...
```

Constructor Example (4)

```
class Animal {
 public:
   Animal(const char* s) { species = s; ... }
   ...
 private:
   string species;
};

class Primate : public Animal {
 public:
   Primate() : Animal("Primate") { ... }
   ...
 private:
   int heart_cham;
};

class Human : public Primate {
 public:
   Human() : Primate() { ... }
   ...
};

Human jill();
Human fred();
```

[Execution Sequence]

Base Class → Derived Class



Constructor Example (5)

```
class Team {    // base class
 public:
   Team(int len = 100) {
      names = new string[maxno = len];
   }  // dynamically allocates storage for a pointer member
 protected:
   string* names;
   int     maxno;
   ...
};

class BaseballTeam : public Team {    // derived class
 public:
   // make sure to invoke BC constructor Team(int)
   //   to allocate storage for "names" before using it
   BaseballTeam(const string s[], int si) : Team(si) {
      for (int i = 0; i < si; i++) names[i] = s[i];
   }
   ...
};
```

Destructors under Inheritance

- Constructors in an inheritance hierarchy are invoked in a "base class to derived class" order

- On the other hand, destructors in an inheritance hierarchy are invoked in the reverse order, i.e., a "derived class to base class" order
  - This ensures that the most recently initialized part of an object is the first to be finalized (i.e., the most recently allocated storage is freed first)
  - Note a destructor **NEVER** explicitly invokes another destructor, since each class has at most one destructor

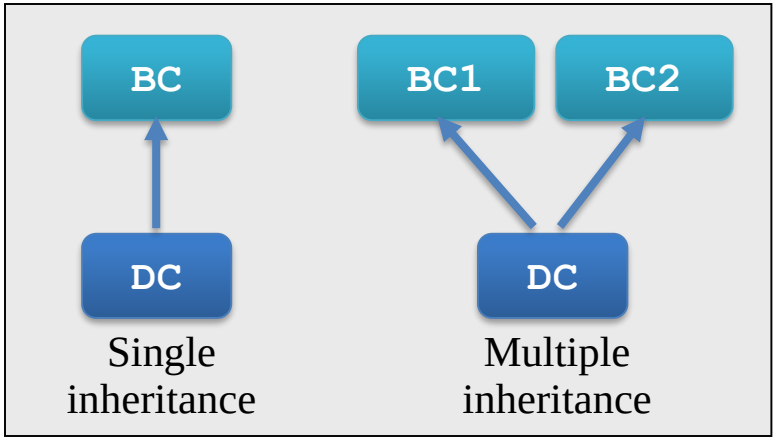
Destructor Example

```
class BC {
 public:
   BC() { cout << "BC's constructor\n"; }
   ~BC() { cout << "BC's destructor\n"; }
};

class DC : public BC {
 public:
   DC() : BC() { cout << "DC's constructor\n"; }
   ~DC() : ~BC() { cout << "DC's destructor\n"; }
   ~DC() { cout << "DC's destructor\n"; }
};

int main() {
   DC d;
   return 0;
}
```

```
BC's constructor
DC's constructor
DC's destructor
BC's destructor
```

Multiple Inheritance

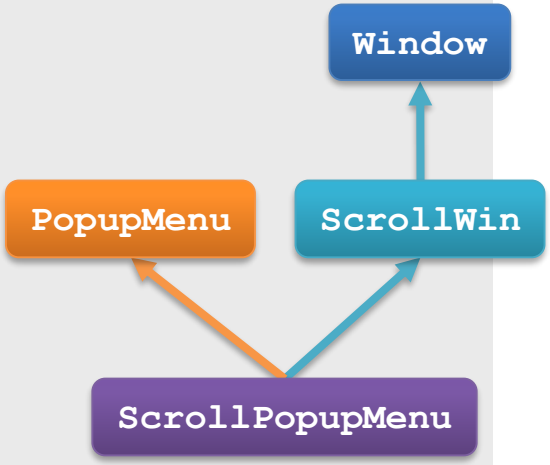
- With multiple inheritance, a derived class may have two or more base classes
  - Inheritance hierarchy forms a *tree* with single inheritances, whereas it may form a *graph* with multiple inheritances
- Meanings of inheritance
  - In a single inheritance hierarchy, a derived class typically represents a ***specialization*** of its base class
  - In a multiple inheritance hierarchy, a derived class typically represents a ***combination*** of its base classes



❖ For reference, Java does **NOT** allow multiple inheritance for classes (c.f., only allows interfaces to inherit from multiple interfaces)

Multiple Inheritance Example (1)

```
// Popup menu class - no scroll bars
class PopupMenu {
 private:
   int  menuChoices;
   Win* menuSubWins;
   ...
};

// Scrolled window class - not a popup
class ScrollWin : public Window {
 private:
   Widget horizontalSB;
   Widget verticalSB;
   ...
};

// Multiple inheritance: combination of popup & scroll bars
class ScrollPopupMenu : public PopupMenu, public ScrollWin {
   ...
};
```



Multiple Inheritance and Access to Members

- The basic rules of inheritance and access in single inheritance are intactly applied to multiple inheritance
  - A derived class inherits data members and methods from all its base classes
- In consequence, multiple inheritance increases opportunity for name conflict (e.g., name hiding)
  - Because the conflict can occur …
    - not only between the derived class and its multiple base classes
    - but between the base classes
  - It is up to the programmer to prevent or resolve those kinds of name conflict

Multiple Inheritance Example (2)

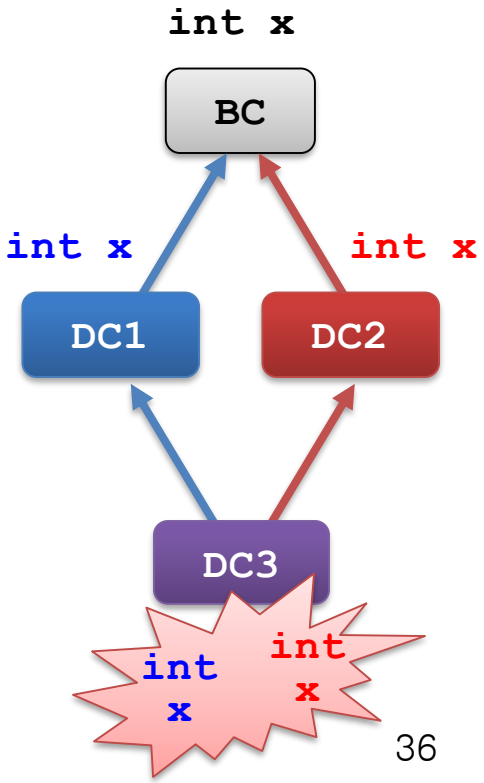
```
class BC1 {  // base class 1
 public:
   void set_x(float a) { x = a }
   ...
};

class BC2 {  // base class 2
 public:
   void set_x(char a) { x = a }
   ...
};

class DC     // derived class
 : public BC1, public BC2 {
 public:
   void set_x(int a) { x = a }
   ...
};
```

```
void tester() {
   DC d1;

   // DC::set_x
   d1.set_x(137);

   // Error: DC::set_x hides
   //    set_x of BC1 and BC2
   d1.set_x(1.23);
   d1.set_x('c');

   // BC1::set_x
   d1.BC1::set_x(1.23);

   // BC2::set_x
   d1.BC2::set_x('c');

   ...
};
```

Multiple Inheritance Originated from the Same Base Class

- Multiple inheritance hierarchy may be complicated
  - This may lead to the situation in which a derived class inherits multiple times from the same indirect base class (e.g., a cycle)
  - It is wasteful and confusing

```
class BC {       // indirect base class of DC3
   int x;
   ...
};

class DC1 : public BC { // inheritance path 1
   ...
};

class DC2 : public BC { // inheritance path 2
   ...
};

class DC3 : public DC1, public DC2 {
   ...           // x comes twice in class DC3
};
```



Virtual Base Classes

- The previous problem can be solved by declaring **DC1** and **DC2** as **virtual** base classes for **DC3**
  - This tells **DC1** and **DC2** to send only one copy of whatever they inherit from there common ancestor **BC** to **DC3**

```
class BC {
   int x;
   ...
};

class DC1 : virtual public BC {
   ...
};

class DC2 : virtual public BC {
   ...
};

class DC3 : public DC1, public DC2 {
   ...          // x comes once
};
```