07: INHERITANCE

Programming Technique II

(SECJ1023)

Adapted from Tony Gaddis and Barret Krupnow (2016), Starting out with C++: From Control Structures through Objects

8.1: Introduction to Inheritance

What is Inheritance?

- **Inheritance** provides a way to create a new class from an existing class.
- The new class is a specialized version of the existing class.
- Classes organised into a 'classification hierarchy'.
- Classes can inherit attributes and methods from other classes, and add extra attributes and methods of its own.

What is the Purpose of Inheritance?

- Generalisation: sharing commonality between two or more classes

- Specialisation: Extending the functionality of an existing class

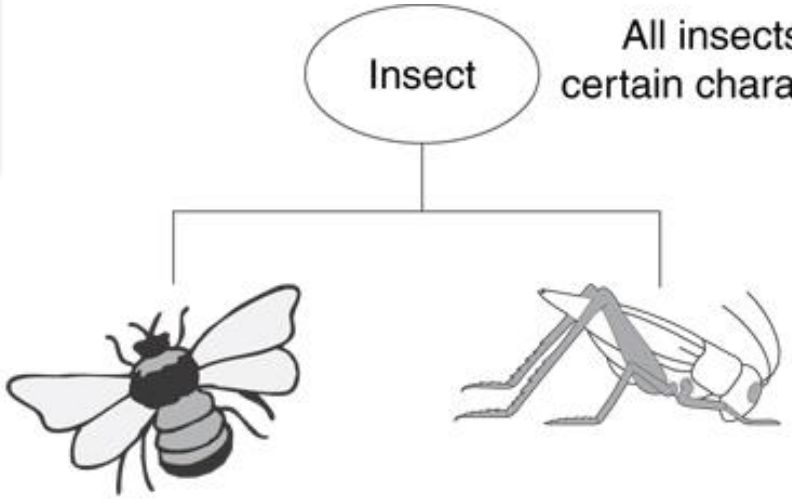
- Generalisation: also applies in Encapsulation. A class is a generalization of objects sharing the same structure. However each object has different data. For example, all students have the same structure, e.g. each of them has a name. Thus, Student is the class. However, each student has their unique name. e.g. “Ali”. Thus student “Ali” is an object

- Specialisation: Objects are specific entities from the same class but with their own data

Example: Insects

Generalisation: Insect represents all of the generic attributes and methods shared by the Bee and Grasshopper. Both Bee and Grasshopper are Insect.

Specialisation: Bee is a specialized version of Insect, which is different from Grasshopper.



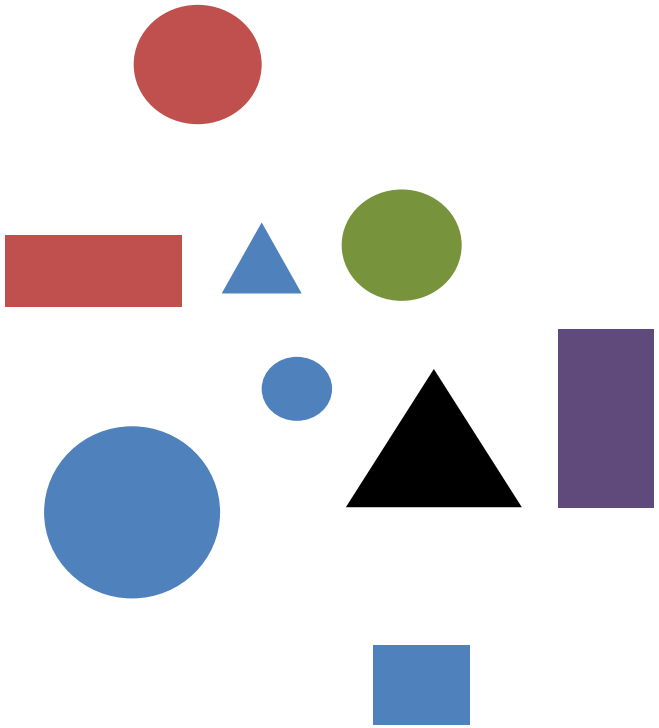
Specialisation: Grasshopper is another specialized version of Insect, which is different from Bee.

More on Generalization Concept

Generalizing objects to classes

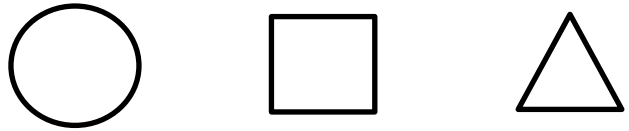
In real world, objects are unique and have their own characteristics

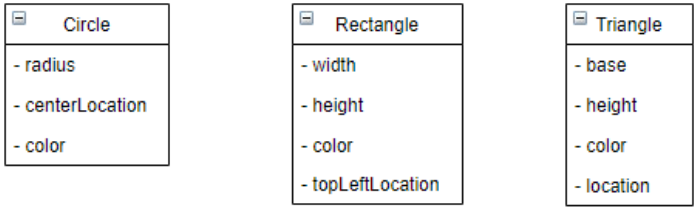
However, they share similar TYPES of characteristics



Generalizing

Three types of objects



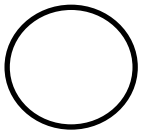


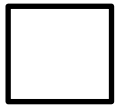
Generalization from the perspective of the encapsulation and class concepts

More on Generalization Concept (2)

Generalizing classes

Even after classifying objects into their types, we still can see they share something in common





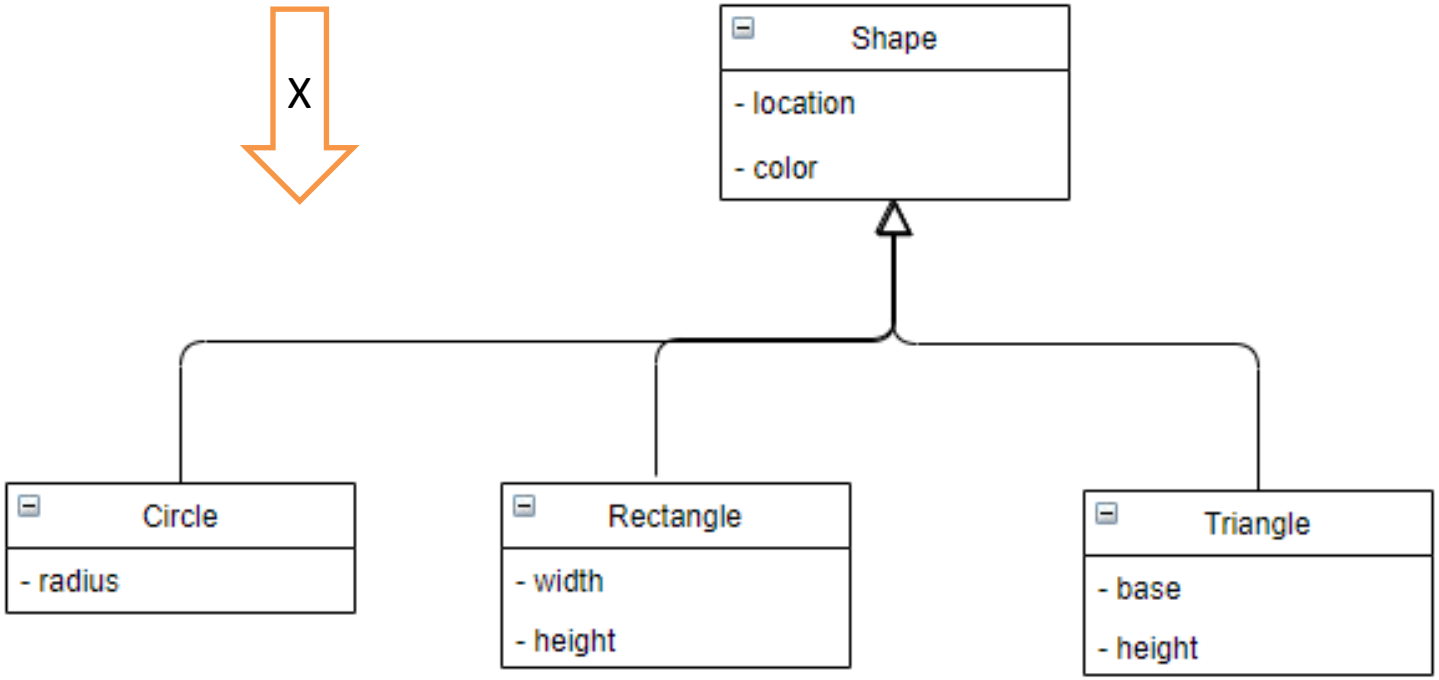
Circle
- radius
- centerLocation
- color

Rectangle
- width
- height
- color
- topLeftLocation

Triangle
- base
- height
- color
- location

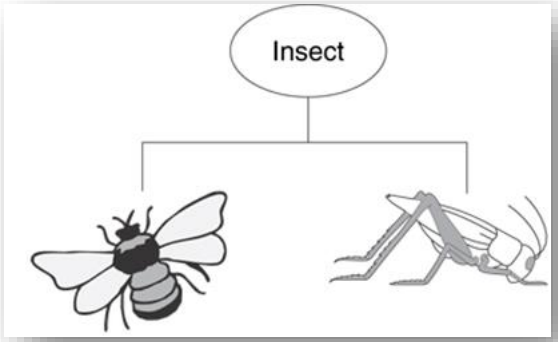
More on Generalization Concept (3)

Generalizing classes



Generalization from the perspective of the inheritance concept

Active Learning Activity




```
class Insect
{
};

class Bee : public Insect
{
};

class GrassHopper : public Insect
{
};
```

```
int main()
{
    Insect insect;
    Bee b1, b2;
    GrassHopper gh;

    // Remaining code will go here

    return 0;
}
```

Question:

Assume a parent class called **`Insect`** and two child classes, **`Bee`** and **`GrassHopper`** have been defined, and several objects have been created from the classes as shown in the above figure. Determine whether each of the following code can compile. If not justify the reason

a). `insect = b1;`	b). `b2 = gh;`
c). `gh = insect;`	d). `insect = gh`

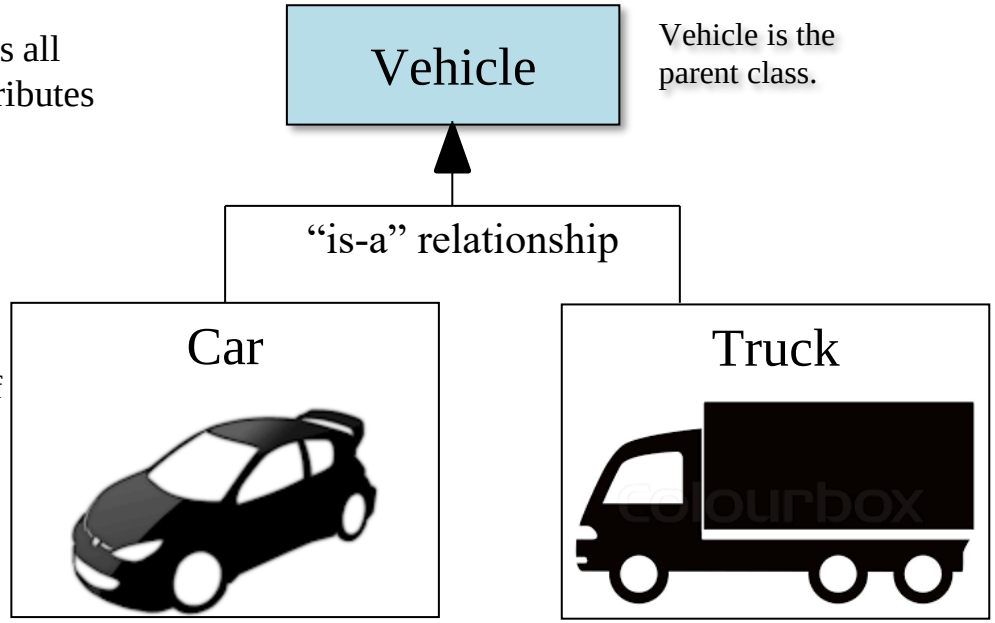
Terminology

- **Base class** or **parent class** or **super class** – A class from which another class inherits.
- **Derived class** or **child class** or **subclass** – A class which inherits some of its attributes and methods from another class.
- The base class represents general characteristics shared by the derived classes.
- Inheritance establishes an **"is a"** relationship between classes.

Example:

- A car is a vehicle. A truck is also a vehicle.
- Vehicle is the base class. Car and Truck are the derived classes.

Vehicle represents all of the generic attributes and methods of a vehicle.



Vehicle is the parent class.

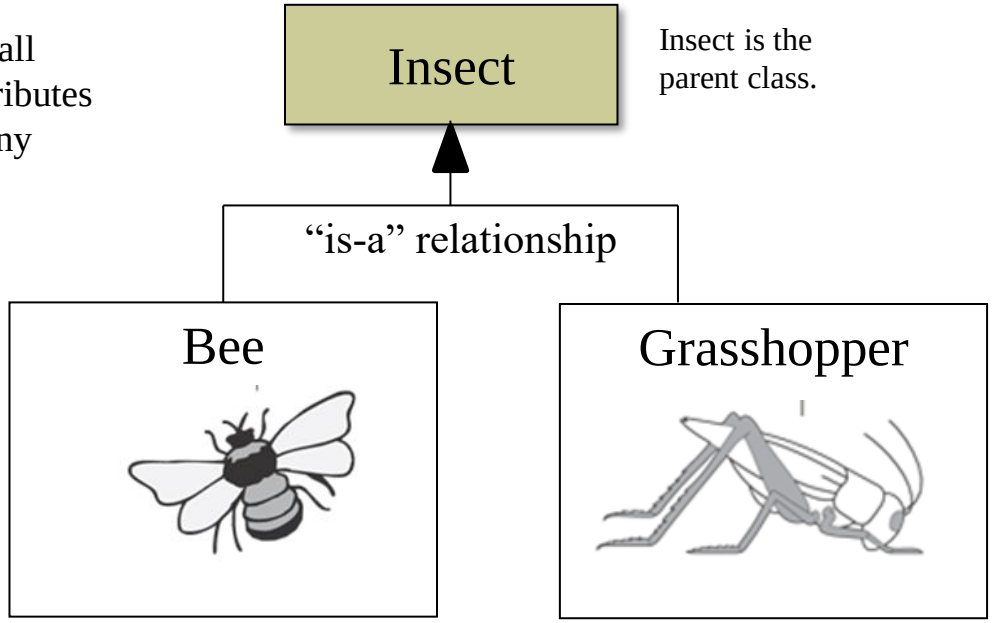
Car and Truck are Specialized versions of a Vehicle.

Car and Truck are child classes of Vehicle.

Example:

- A bee is an insect. A grasshopper is also an insect.
- Insect is the base class. Bee and Grasshopper are the derived classes.

Insect represents all of the generic attributes and methods of any insect.



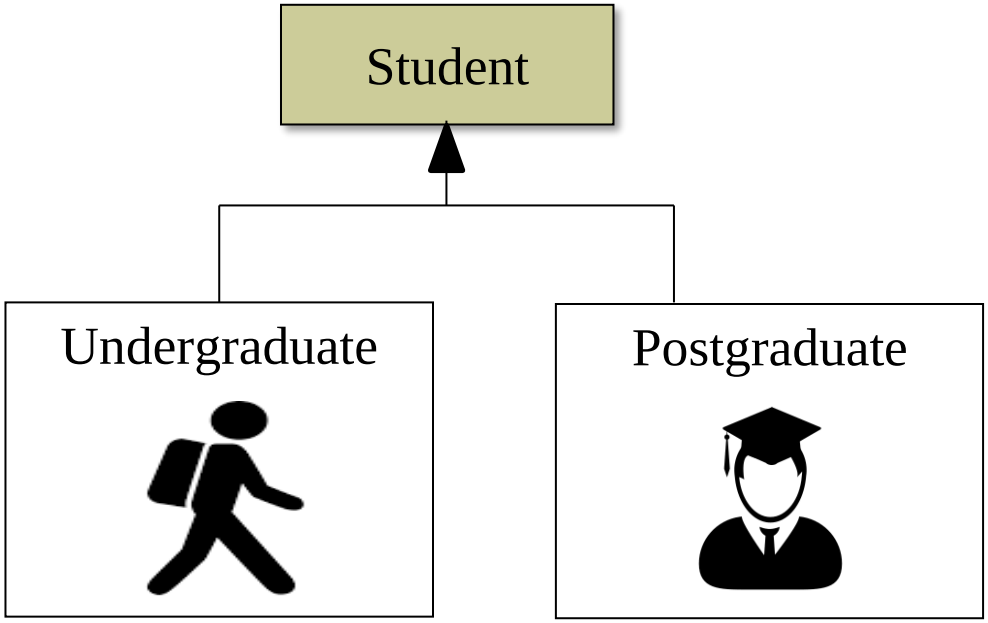
Insect is the parent class.

Bee and Grasshopper are Specialized versions of an Insect.

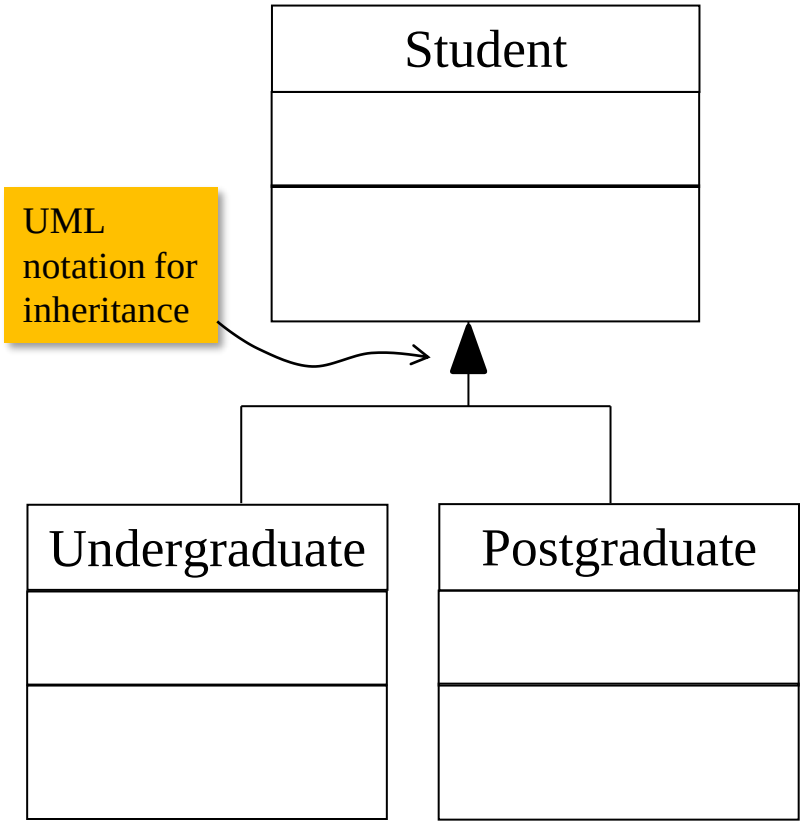
Bee and Grasshopper are child classes of Insect.

Example:

- A student can be an undergraduate or a postgraduate student.
- The base class is Student, and the derived classes are Undergraduate and Postgraduate.



Notations




```
//base class
class Student
{
    . . .
};
```

C++ code for inheritance

```
// derived classes
class Undergraduate:public Student
{                   . . .
};

class Postgraduate : public Student
{   . . .
};
```

What Does a Child Have?

- An object of the derived class has:
  - all members defined in child class
  - all members declared in parent class

- An object of the derived class can use (or access to):
  - all `public` members defined in child class
  - all `public` members defined in parent class

8.2: Protected Members and Class Access

Protected Members and Class Access

- `protected` member access specification: like `private`, but accessible by derived classes.
  - Only the derived classes can access to `protected` members in the base class, but not their objects.

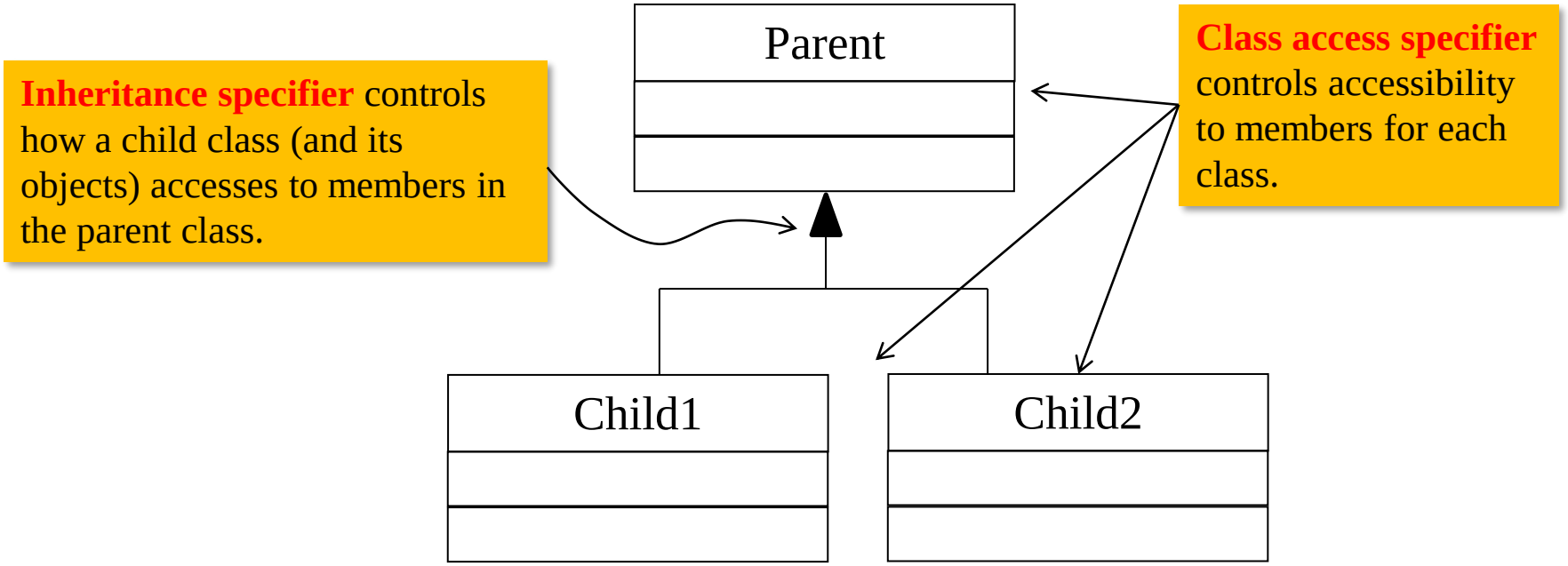
- Class access specification: determines how `private`, `protected`, and `public` members of base class are inherited by the derived class.

Access Specifiers

- `public` – object of derived class can be treated as object of base class (not vice-versa)
- `protected` – more restrictive than `public`, but allows derived classes to know details of parents
- `private` – prevents objects of derived class from being treated as objects of base class.

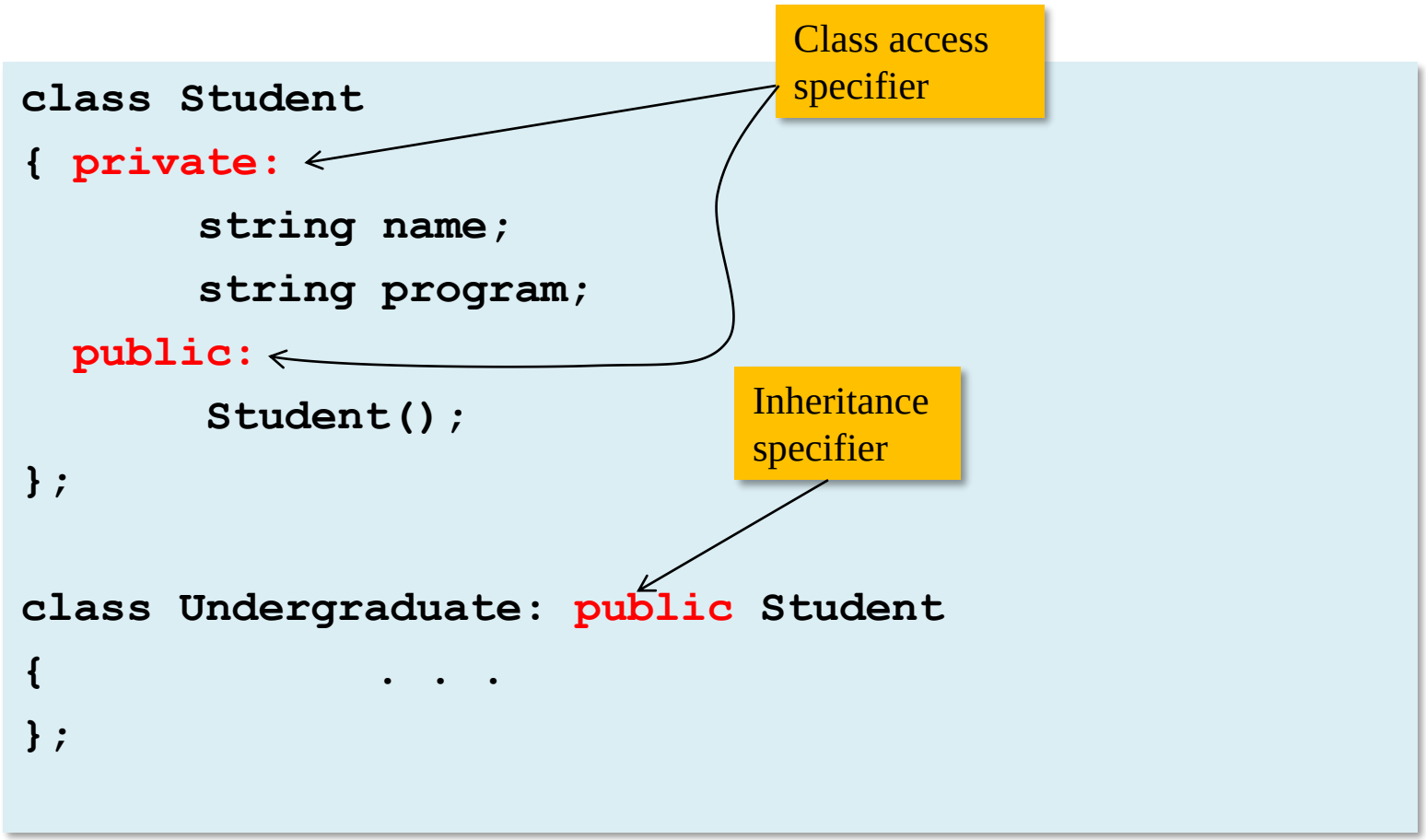
Inheritance vs. Class Access Specifiers

- Member accessibility can be specified at two area:
  - Inside each class. (called Class Access Specifier)
  - When a derived class extends the base class (called Inheritance Specifier).

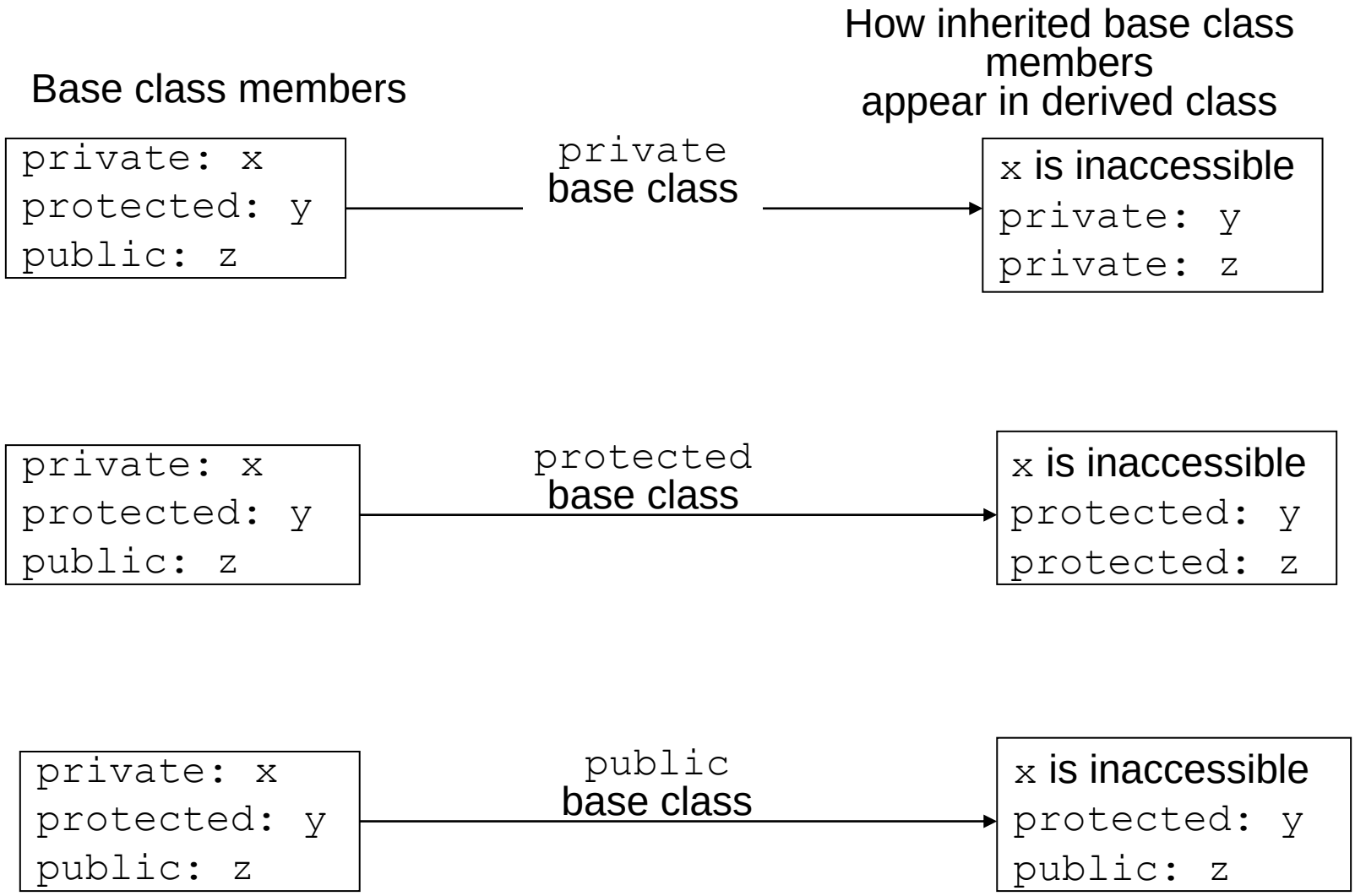


Inheritance vs. Class Access Specifiers

Example:



Inheritance vs. Class Access Specifiers



Accessibility vs. Ownerships

- An object owns all members from the class it was created from, regardless of `private`, `public` or `protected`.

- However, the object can only access to the `public` members.

- A class can access all its own members.

Accessibility vs. Ownerships

Example:

Ownership

```
1  class Student{
2      private:
3          string name;
4          string program;
5
6      public:
7          Student(){
8            ....
9          }
10
11         void print() const{
12             cout << "Name:     " << name    << endl;
13             cout << "Program: " << program << endl;
14         }
15 };
```

```
17 class Undergraduate: public Student{
18     private:
19         double cgpa;
20
21     public:
22         Undergraduate(){
23             ....
24         }
25
26         void read(){
27             ....
28         }
29 };
30
31 Student s;
32 Undergraduate u;
33
```

	name	program	print()	cgpa	read()
Student	✓	✓	✓	-	-
Undergraduate	✓	✓	✓	✓	✓
Object **s**	✓	✓	✓	-	-
Object **u**	✓	✓	✓	✓	✓

Accessibility vs. Ownerships

Example:

Accessibility

```
1  class Student{
2      private:
3          string name;
4          string program;
5
6      public:
7          Student(){
8            ....
9          }
10
11         void print() const{
12             cout << "Name:     " << name    << endl;
13             cout << "Program: " << program << endl;
14         }
15 };
```

```
17 class Undergraduate: public Student{
18     private:
19         double cgpa;
20
21     public:
22         Undergraduate(){
23             ....
24         }
25
26         void read(){
27             ....
28         }
29 };
30
31 Student s;
32 Undergraduate u;
33
```

	name	program	print()	cgpa	read()
Student	✓	✓	✓	-	-
Undergraduate	-	-	✓	✓	✓
Object **s**	-	-	✓	-	-
Object **u**	-	-	✓	-	✓

Accessibility vs. Ownerships

- It is a good idea to specify data members as `protected` rather than `private`.
  - Thus, child classes can directly access to them,
  - while the objects still cannot access to them, i.e., the concept of data hiding remains.

- Inheritance access specifier is commonly specified as `public`.

Accessibility vs. Ownerships

Example:

```
1  class Student{
2      protected:
3          string name;
4          string program;
5
6      public:
7          Student(){
8            ....
9          }
10
11         void print() const{
12             cout << "Name:     " << name    << endl;
13             cout << "Program: " << program << endl;
14         }
15 };
```

```
17 class Undergraduate: public Student{
18     protected:
19         double cgpa;
20
21     public:
22         Undergraduate(){
23             ....
24         }
25
26         void read(){
27             ....
28         }
29 };
30
31 Student s;
32 Undergraduate u;
```

Accessibility

	name	program	print()	cgpa	read()
Student	✓	✓	✓	-	-
Undergraduate	✓	✓	✓	✓	✓
Object **s**	-	-	✓	-	-
Object **u**	-	-	✓	-	✓

8.3: Constructors and Destructors in Base and Derived Classes

Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors

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

- When an object of a derived class is destroyed, its destructor is called first, then that of the base class

Example:

```
1  // This program demonstrates the order in which base and
2  // derived class constructors and destructors are called.
3  #include <iostream>
4  using namespace std;
5
6  //*************************************
7  // BaseClass declaration              *
8  //*************************************
9
```

```
10  class BaseClass
11  {
12  public:
13     BaseClass()  // Constructor
14        { cout << "This is the BaseClass constructor.\n"; }
15
16     ~BaseClass() // Destructor
17        { cout << "This is the BaseClass destructor.\n"; }
18  };
19
20  //*********************************
21  // DerivedClass declaration      *
22  //*********************************
23
24  class DerivedClass : public BaseClass
25  {
26  public:
27     DerivedClass()  // Constructor
28        { cout << "This is the DerivedClass constructor.\n"; }
29
30     ~DerivedClass()  // Destructor
31        { cout << "This is the DerivedClass destructor.\n"; }
32  };
33
```

```
34  //*********************************
35  // main function                  *
36  //*********************************
37
38  int main()
39  {
40     cout << "We will now define a DerivedClass object.\n";
41
42     DerivedClass object;
43
44     cout << "The program is now going to end.\n";
45     return 0;
46  }
```

Program Output

```
We will now define a DerivedClass object.
This is the BaseClass constructor.
This is the DerivedClass constructor.
The program is now going to end.
This is the DerivedClass destructor.
This is the BaseClass destructor.
```

Passing Arguments to Base Class Constructor

- Allows selection between multiple base class constructors
- Specify arguments to base constructor on derived constructor heading.
- Must be done if base class has no default constructor

Example:

```
 6  class Rectangle{
 7      protected:
 8          int width;
 9          int height;
10      public:
11          Rectangle(int _width, int _height){
12              width  = _width;
13              height = _height;
14          }
15  };
16
17  class Square : public Rectangle {
18      public:
19          Square(int length) : Rectangle(length, length)
20          {}
21  };
```

base class constructor

derived class constructor

Example:

if not using inline style

```
 6  class Rectangle{
 7      protected:
 8          int width;
 9          int height;
10      public:
11          Rectangle(int, int);
12  };
13
14  class Square : public Rectangle {
15      public:
16          Square(int);
17  };
18
19  Rectangle::Rectangle(int _width, int _height){
20              width  = _width;
21              height = _height;
22  }
23
24
25
26
27  Square::Square(int length) : Rectangle(length, length)
28  {}
29
30
```

derived class constructor

base class constructor

8.4: Redefining Base Class Functions

Redefining Base Class Functions

- To redefine a **`public`** member function of a base class
  - Corresponding function in the derived class must have the same name, number, and types of parameters

- If derived class overrides a **`public`** member function of the base **`class`**, then to call to the base class function, specify:
  - Name of the base class
  - Scope resolution operator (`::`)
  - Function name with the appropriate parameter list

Redefining Base Class Functions

- Not the same as overloading – with overloading, parameter lists must be different

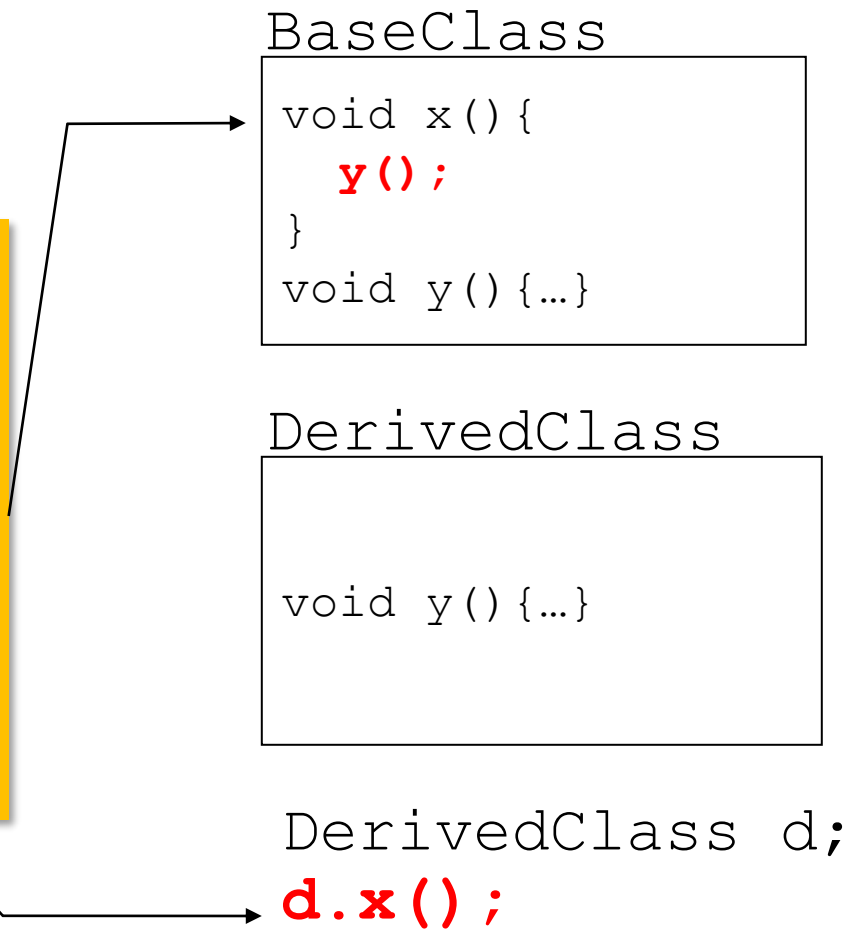
- Objects of base class use base class version of function; objects of derived class use derived class version of function

Problem with Redefining

- Consider this situation:
  - Class **`BaseClass`** defines functions **`x()`** and **`y()`**. **`x()`** calls to **`y()`**.
  - Class **`DerivedClass`** inherits from **`BaseClass`** and redefines function **`y()`**.
  - An object **`d`** of class **`DerivedClass`** is created and function **`x()`** is called to.
  - When **`x()`** is called to, which **`y()`** is used?, the one defined in **`BaseClass`** or the the redefined one in **`DerivedClass`**?

Problem with Redefining

- Object `d` invokes function `x()` of `BaseClass`.
- Function `x()` invokes function `y()` of `BaseClass`, not function `y()` of `DerivedClass`, because function calls are bound at compile time. This is static binding.



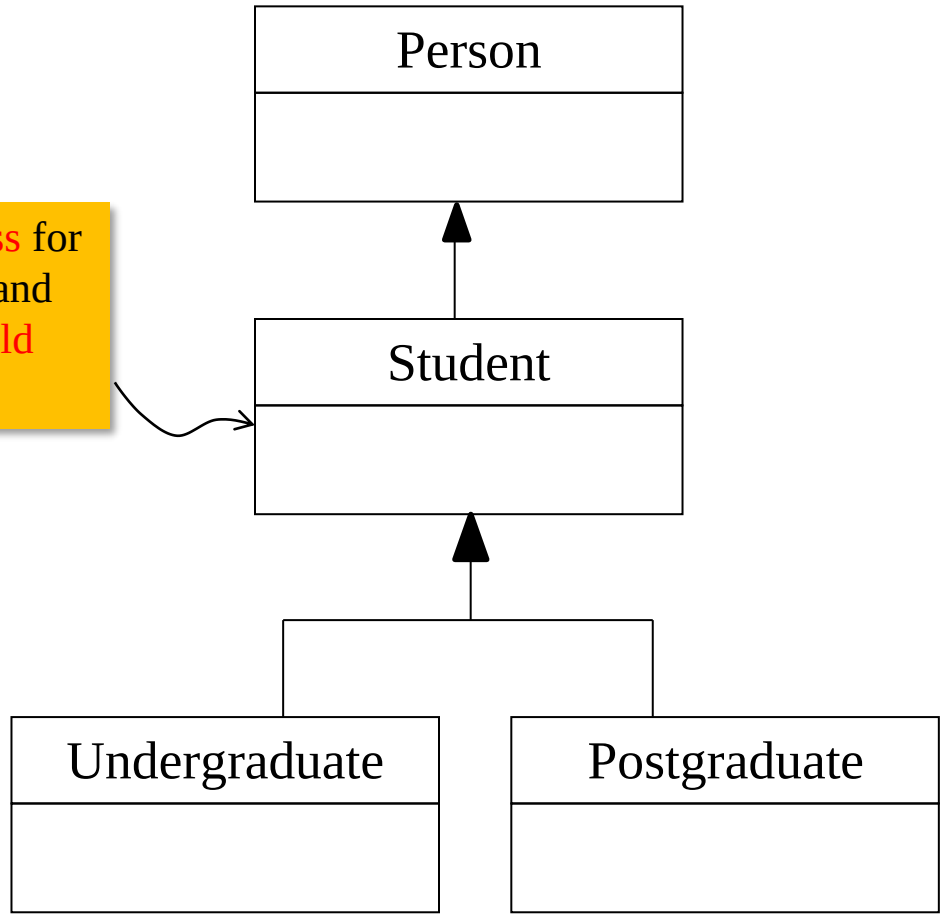
8.5: Class Hierarchies

Class Hierarchies

- A base class can be derived from another base class.

Example:

`Student` is the parent class for classes `Undergraduate` and `Postgraduate`, and a child class from class `Person`



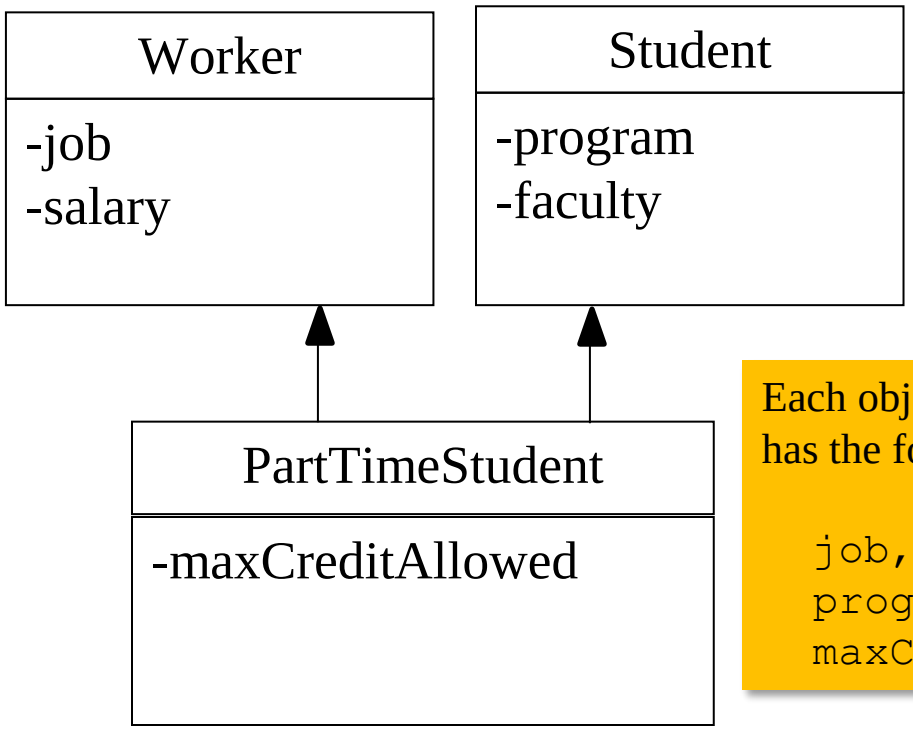
8.6: Multiple Inheritance

Multiple Inheritance

- A derived (child) class can have more than one base (parent) class.
- Each base class can have its own access specification in derived class's definition
- Multiple inheritance allows a derived class to inherit features from different classes.

Multiple Inheritance

Example:



Each object of PartTimeStudent has the following attributes:

```
job, salary,
program, faculty
maxCreditAllowed
```

Multiple Inheritance

Example:

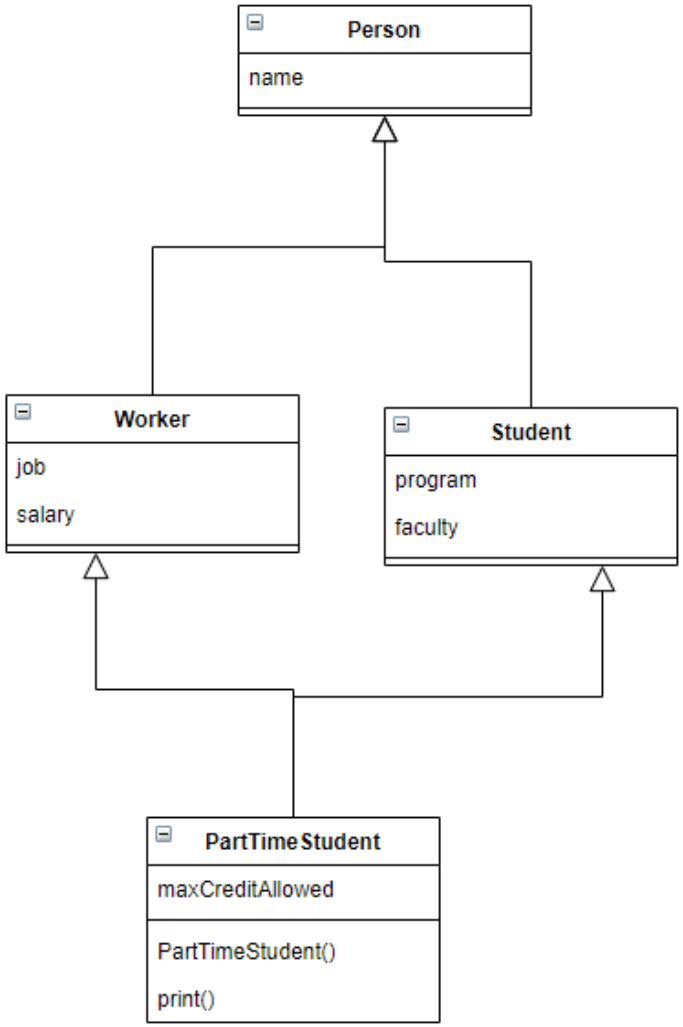
```
6  class Worker{
7      protected:
8          string job;
9          double salary;
10
11     public:
12         Worker(string _job="", double _salary=0.0){
13             job = _job;
14             salary = _salary;
15         }
16 };
17
18
19 class Student{
20     protected:
21         string program;
22         string faculty;
23
24     public:
25         Student(string _program="", string _faculty=""){
26             program = _program;
27             faculty = _faculty;
28         }
29 };
```

Multiple inheritance

```
32  class PartTimeStudent: public Student, public Worker
33  {
34      protected:
35          int maxCreditAllowed;
36
37      public:
38          PartTimeStudent( int _maxCreditAllowed=0,
39                           string _program="", string _faculty="",
40                           string _job="", double _salary=0.0
41                         ) : Student(_program, _faculty) , Worker(_job, _salary)
42          {
43              maxCreditAllowed = _maxCreditAllowed;
44              program = _program;
45              faculty = _faculty;
46          }
47
48          void print() const{
49              cout << "Part Time Student Information: " << endl << endl;
50
51              cout << "Program: " << program << endl;   // inherited from Student
52              cout << "Faculty: " << faculty << endl;   // inherited from Student
53
54              cout << "Job    : " << job << endl;       // inherited from Worker
55              cout << "Salary : " << salary << endl;    // inherited from Worker
56
57              cout << "Max Credit Allowed: " << maxCreditAllowed << endl;  // its own member
58          }
59  };
```

Virtual Parent Class

Consider the following multiple inheritance:




```
6   class Person
7   {
8   protected:
9       string name;
10  };
11
12  class Worker : public Person
13  {
14  protected:
15      string job;
16      double salary;
17  };
18
19  class Student : public Person
20  {
21  protected:
22      string program;
23      string faculty;
24  };
25
26  class PartTimeStudent : public Student, public Worker
27  {
28  protected:
29      int maxCreditAllowed;
30
31  public:
32      void print() const
33      {
34          cout << "Name: " << name << endl;
35      }
36  };
```

Virtual Parent Class

```
6   class Person
7   {
8   protected:
9       string name;
10  };
11
12  class Worker : public Person
13  {
14  protected:
15      string job;
16      double salary;
17  };
18
19  class Student : public Person
20  {
21  protected:
22      string program;
23      string faculty;
24  };
25
26  class PartTimeStudent : public Student, public Worker
27  {
28  protected:
29      int maxCreditAllowed;
30
31  public:
32      void print() const
33      {
34          cout << "Name: " << name << endl;
35      }
36  };
```

This results in a compilation error due to ambiguity

Problem with multiple inheritance:

- What if you want to print the **`name`** from the class **`PartTimeStudent`**?
- The attribute `name` of class `Person` is inherited twice to class `PartTimeStudent`, i.e., through class `Worker` and class `Student`.
- When class `PartTimeStudent` accesses the attribute `name`, it will result in a compilation error due to an ambiguity.

Virtual Parent Class

- Solution: **virtual parent class**
- Virtual parent classes prevent the child class from inheriting multiple instances of the same member.

Note: the `virtual` keyword can be written before or after the access modifier

```
6   class Person
7   {
8   protected:
9       string name;
10  };
11
12  class Worker : public virtual Person
13  {
14  protected:
15      string job;
16      double salary;
17  };
18
19  class Student : public virtual Person
20  {
21  protected:
22      string program;
23      string faculty;
24  };
25
26  class PartTimeStudent : public Student, public Worker
27  {
28  protected:
29      int maxCreditAllowed;
30
31  public:
32      void print() const
33      {
34          cout << "Name: " << name << endl;
35      }
36  };
```

Inherit the parent class virtually

This should work now