AMA2222 Principles of Programming

Leung Man Kin, Adam
Instructor

adam.leung@polyu.edu.hk
TU720

Chapter 9: Object-oriented programming
defining a class, data encapsulation,
passing object to function, overloading operator,
array of objects, class inheritance,
data encapsulation in inheritance, multiple inheritance

Class

Object-oriented programming (OOP) involves programming using objects. An **object** represents an entity in the real world that can be distinctly identified. An object has a unique identity, **state**, and **behaviour**.

The state of an object is known as properties or attributes of the object. It is represented by **data fields** with their current values.

The behaviour of an object is defined by **functions**. To invoke a function on an object is to ask the object to perform an action.

A **class** is a template that defines what an object's data fields and functions will be.

Class: bubbletea

(there are different objects with different names in the same class)

Data field:

int ice *(1, 2, 3, 4, 5)*

double sugar *(1, 0.7, 0.5, 0.3, 0)*

bool skim *(0, 1)*

Constructor: bubbletea(int ice, double sugar, bool skim)

Default constructor: bubbletea() using regular ice and sugar level with whole milk (not skim milk)

Function: addsugar(), calorie(), cost(), etc…

What other classes can you think of?
What should be included in their data field?
What are the variable types?

class person: name (string), age (int)

class student: name (string), student id (string), year (int), GPA (double), etc.

class employee: name (string), year of service (int), salary (double), etc.

class rectangle: width (double), height (double).

class rational number: numerator (int), denominator (int).

For example, we would like to define a class of rectangles.

```
 4 class rectangle
 5 {
 6 public:
 7         double width;
 8         double height;
 9
10         rectangle()
11         {
12                 width = 1;
13                 height = 1;
14         }
15         rectangle(double w, double h)
16         {
17                 width = w;
18                 height = h;
19         }
20
21         double getArea()
22         {
23                 return width*height;
24         }
25 };
```

Data field, object under the class rectangle should carries two properties: width and height.

Constructor, define a default rectangle with width and height both 1 unit.

Constructor, define a rectangle with two input values as the width and height.

Function, return the area of the rectangle based on its width and height.

Example program 9.1 Class of rectangles

```
 1 // 9.1 Class of rectangles
 2 #include <iostream>
 3 using namespace std;
   // insert the class on previous page
26 int main()
27 {
28          rectangle r1;
29          rectangle r2(3,4);
30
31          cout << "The width of r1 is " << r1.width << endl;
32          cout << "The height of r1 is " << r1.height << endl;
33          cout << "The area of r1 is " << r1.getArea() << endl;
34
35          cout << "The width of r2 is " << r2.width << endl;
36          cout << "The height of r2 is " << r2.height << endl;
37          cout << "The area of r2 is " << r2.getArea() << endl;
38
39          return 0;
40 }
```

```
The width of r1 is 1
The height of r1 is 1
The area of r1 is 1
The width of r2 is 3
The height of r2 is 4
The area of r2 is 12
```

A **constructor** is a special kind of function which is designed to perform initializing actions, such as initializing the data fields of objects. Constructors have three properties:

i. Constructors must have the same names as the class itself.
ii. Constructors do not have a return type – not even void.
iii. Constructors are invoked when an object is created.

A class may be defined without constructors. In this case, a **no-argument** constructor with an empty body is automatically defined in the class, called a **default constructor**.

After an object is created, its data can be accessed and its functions can be invoked using the **dot operator**:

- `objectName.dataField` references a data field in the object.
- `objectName.function(arguments)` invokes a function on the object.

You can use the assignment operator = to copy the contents from one object to the other. By default, each data field of one object is copied to its counterpart in the other object. For example,

```
r1 = r2;
```

is equivalent to

```
r1.width = r2.width;
r1.height = r2.height ;
```

Classwork exercise 9.1:

In the class of rectangle, write the following functions:
`(a)getPeri()` that calculates the perimeter of the rectangle
`(b)isSq()` that check whether the rectangle is a square
`(c)flip()` that flips the rectangle (swaps height and width)
`(d)enlarge(x)` that multiplies both width and height by x

Paste them inside the example program and run to check.

Solution:

```
double getPeri()
{
        return (width + height) * 2;
}
```

```
bool isSq()
{
        return (width == height);
}
```

```
void flip()
{
        swap(width, height);
}
```

```
void enlarge(double x)
{
        width *= x;
        height *= x;
}
```

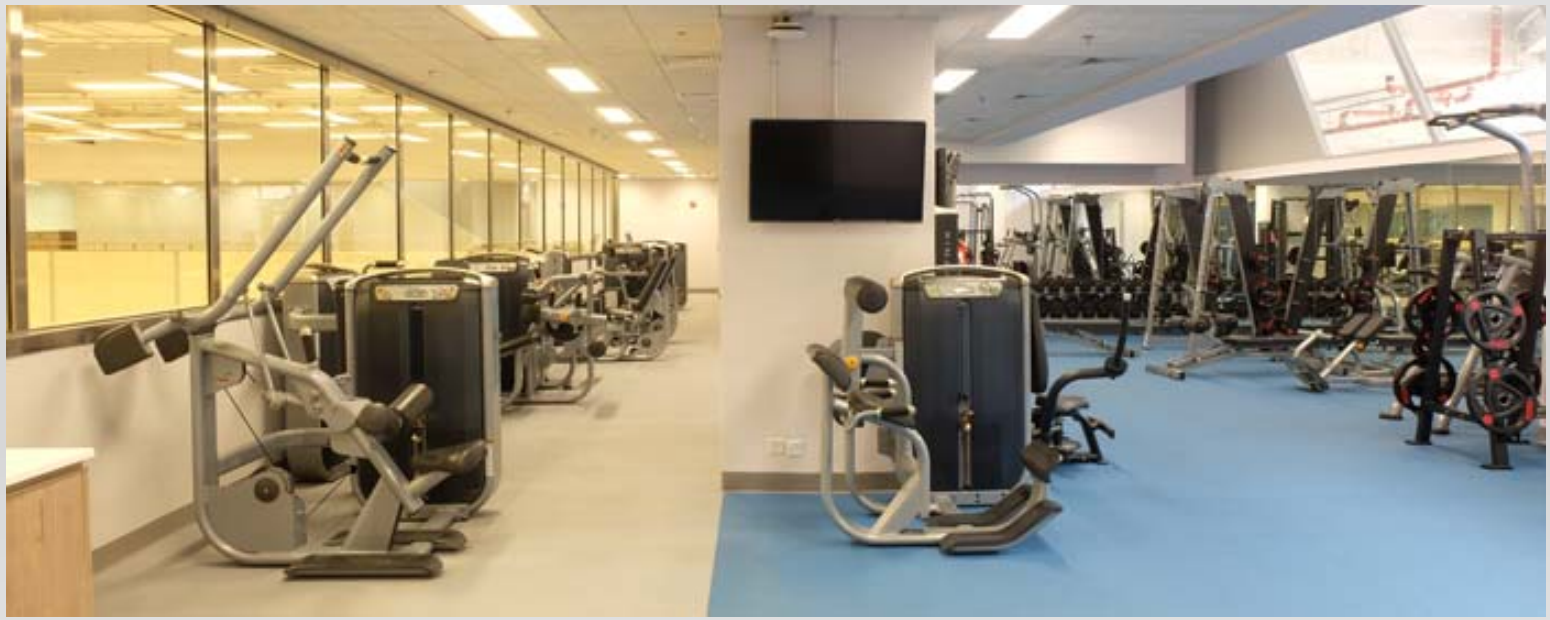
Notice that these functions are under the class rectangle with or without input arguments.

For example, if we want to find the perimeter of r2, we should use `r2.getPeri()` but **NOT** `getPeri(r2)`. If we want to enlarge r2 by 10 times, we should use `r2.enlarge(10)`.

Example program 9.2: Gym room booking

The gym room is available for teachers and students to book. Each booking must start and end at an integer hour of time. Moreover, the charge is $6/h and $3/h respectively.

We are going to write a class called `gymbook` to record the details of booking of that day.



```
 1 // 9.2a Gym room booking
 2 #include <iostream>
 3 #include <string>
 4 using namespace std;
 5 class gymbook
 6 {
 7 public:
 8           string name;
 9           char status;
10           int start, end;
11           gymbook(string n, char s, int st, int et)
12           {
13                     name = n;
14                     status = s;
15                     start = st;
16                     end = et;
17           }
18           int duration()
19           {
20                     return (end - start);
21           }
22           int charge()
23           {
24                     if (status == 'T') return 6*(end - start);
25                     else return 3*(end - start);
26           }
27 };
```

Constructor, define a booking with several field for data input.

Function that calculates the number of hours booked.

Function that calculates the $ to charge

```
28 int main()
29 {
30         string name;
31         char status;
32         int start, end;
33         cout << "Please enter your name: ";
34         getline(cin,name);
35         cout << "Please enter your status (T:Teacher / S:Student): ";
36         cin >> status;
37         cout << "Please enter your starting and ending time (hour): ";
38         cin >> start >> end;
39
40         gymbook b(name, status, start, end);
41
42         cout << "Hello " << b.name << " !" << endl;
43         cout << "You have booked " << b.duration() << " hours." << endl;
44         cout << "The charge is " << b.charge() << " dollars." << endl;
45
46         return 0;
47 }
```

Prompt user to enter the required data field for gym room booking.

Make the booking with the input information

Functions in a class

In the previous example, we have seen some functions defined within a class for example:

In 9.1 rectangle: `getArea(), gerPeri(), isSq()`

In 9.2 gymbook: `duration(), charge()`

They will return a value based on the stored value in the data field of the object.

Actually we can construct some functions that a user can enter some extra input values, and change the stored value of the object.

Example 9.2b Gym room booking

In the class of gymbook, we would like to define two more functions:

`delay(int h)`

delay the booking by h hours

```
void delay(int h)
{
        start += h;
        end += h;
}
```

`transfer(string new_name, char new_status)`

transfer the booking to another person

```
void transfer(string new_name, char new_status)
{
        name = new_name;
        status = new_status;
}
```

```
int main()
{
        gymbook b("Amy", 'T', 13, 15);

        string new_name;
        char new_status;

        cout << "Please the name of person to transfer: ";
        getline(cin,new_name);
        cout << "Please the status of person to transfer(T:Teacher / S:Student): ";
        cin >> new_status;
        b.transfer(new_name, new_status);
        cout << "The booking has been transferred to " << b.name <<endl;

        int h;
        cout << "Please enter the hours to delay: ";
        cin >> h;
        b.delay(h);

        cout << "The delayed booking is " << b.start << " to " << b.end << endl;

        return 0;
}
```

Data field encapsulation

In the previous example, the data fields `width` and `height` in the `rectangle` class in public section can be modified directly. This is not a good practice because it makes the class difficult to maintain and vulnerable to bugs, e.g. `r1.width = -3` results in negative width which may not be easily found out.

Data field encapsulation prevents direct modifications of data field. This is accomplished by declaring data fields in **private** section instead of public section.

To make a private data field accessible, provide a get function to return the field's value. To enable a private data field to be updated, provide a set function to set a new value:

```
returnType getDataFieldName()
void setDataFieldName(dataType dataFieldValue)
```

```
1  // 9.1b Data field encapsulation
2  #include <iostream>
3  using namespace std;
4  class rectangle
5  {
6  public:
7          rectangle()
8          {
9                  width = 1;
10                 height = 1;
11         }
12         double getWidth()
13         {
14                 return width;
15         }
16         void setWidth(double w)
17         {
18                 if (w > 0)
19                         width = w;
20         }
21 private:
22         double width;
23         double height;
24 };
```

```
25 int main()
26 {
27         rectangle r1;
28         r1.setWidth(-1);
29         cout << r1.getWidth() << endl;
30         r1.setWidth(4);
31         cout << r1.getWidth() << endl;
32         return 0;
33 }
```

Constructor, define a default rectangle with width and height both 1 unit.

Function that returns the width of a certain rectangle.

Function that inputs a value and set it as the width of a certain rectangle, only if such value is positive.

The two data fields, width and height, of a rectangle cannot be accessed or modified in the main program directly.

In the gym room booking example, we have to make sure the input data are valid, i.e.

- The status is either 'T' or 'S'
- The starting time and ending time are integers in 7 to 23
- The starting time is less than the ending time.

```
 1 // 9.2c Gym room booking
 2 #include <iostream>
 3 #include <string>
 4 using namespace std;
 5 class gymbook
 6 {
 7 private:
 8           string name;
 9           char status;
10           int start=0, end=0;
```

```
public:
        void setUser(string n, char s)
        {
                name = n;
                if (s=='T' || s=='S') status = s;
        }
        void setTime(int st, int et)
        {       if (st>=7 && et <=23 && st<et) {
                        start = st;
                        end = et;
                }
        }
        string getName()
        {       return name;
        }
        int duration()
        {        if (status == 'T'||status == 'S') return (end - start);
                 else return -1;
        }
        int charge()
        {       if (status == 'T') return 6*(end - start);
                else if (status == 'S') return 3*(end - start);
                else return -1;
        }
};
```

```
36 int main()
37 {
38          string name;
39          char status;
40          int start, end;
41          cout << "Please enter your name: ";
42          getline(cin,name);
43          cout << "Please enter your status (T:Teacher / S:Student): ";
44          cin >> status;
45          cout << "Please enter your starting hour and ending hour: ";
46          cin >> start >> end;
47
48          gymbook b;
49          b.setUser(name, status);
50          b.setTime(start, end);
51
52          cout << "Hello " << b.getName() << " !" << endl;
53          cout << "You have booked " << b.duration() << " hours." << endl;
54          cout << "The charge is " << b.charge() << " dollars." << endl;
55
56          return 0;
57 }
```

Define b as a object of class gymbook first, then use the set function to enter the data.

Notice that you can no longer use b.name directly as it is in the private sector.

Classwork exercise 9.2

Design a class named **`Point`** to represent a point with x- and y-coordinates. The class contains:

- Two private data fields **`x`** and **`y`** that represent the coordinates respectively.
- A no-argument constructor that creates a point (0,0).
- A constructor that constructs a **`Point`** object with specified coordinates.
- Two **`get`** functions for data fields **`x`** and **`y`**, respectively.
- A function named **`distance`** that takes a **`Point`** object as input and returns the distance from this **`Point`** to the input **`Point`** object.

Suppose the main function is

```
#include <iostream>
#include <string>
// missing class
int main()
{
        Point p1(3,4);
        Point p2;

        cout << "The distance between "
        << "(" << p1.getX() << "," << p1.getY() << ")"
        << " and "
        << "(" << p2.getX() << "," << p2.getY() << ")"
        << " is "
        << p1.distance(p2) << endl;
        return 0;
}
```

The output should be

```
The distance between (3,4) and (0,0) is 5
```

```
class Point
{
public:
        Point(double n1, double n2)
        {
                x = n1;
                y = n2;
        }
        Point()
        {
                x = 0;
                y = 0;
        }

        double getX()
                {return x;}
        double getY()
                {return y;}

        double distance(Point p)
        {
                return sqrt(pow(x-p.getX(),2)+pow(y-p.getY(),2));
        }
private:
        double x;
        double y;
};
```

Passing objects to functions

You may use an object of a defined class to be the input of a function.

Example 9.1c Difference of area of two rectangles
We would like to construct a function called “difference” that inputs two rectangles and return the difference of their area.

For example:

```
Enter the dimensions of the first rectangle: 2 4
Enter the dimensions of the second rectangle: 3 5
The difference in their area is 7
```

```
 1 // 9.1c Passing objects to function
 2 #include <iostream>
 3 using namespace std;
   // insert the class from example 9.1a
26 double difference(rectangle r1, rectangle r2)
27 {
28          if (r1.getArea()>=r2.getArea())
29                    return r1.getArea()-r2.getArea();
30          else return r2.getArea()-r1.getArea();
31 }
32 int main()
33 {
34          double w1,h1,w2,h2;
35          cout << "Enter the dimensions of the first rectangle: ";
36          cin >> w1 >> h1;
37          rectangle r1(w1,h1);
38          cout << "Enter the dimensions of the second rectangle: ";
39          cin >> w2 >> h2;
40          rectangle r2(w2,h2);
41          cout << "The difference in their area is " << difference(r1,r2);
42          return 0;
43 }
```

Overloading operator

C++ allows you to define functions for operators, such as

+ - / * % ^ < >

by defining the operator function in the class. This is called **operator overloading**.

The operator function is just like a regular function except that it must be named with keyword operator followed by the actual operator. For example, to define the * operator function, use the following function header:

```
className operator*(className objectName)
```

Example program 9.3 rational number

A rational number is a number which can be expressed as an integer numerator over an integer denominator.

```
 1 // 9.3 Rational number
 2 #include <iostream>
 3 using namespace std;
 4 class Rational
 5 {
 6 public:
 7           Rational(int a,int b)
 8           {
 9                     num = a;
10                     den = b;
11           }
12           Rational operator+(Rational r)
13           {
14                     int a = num * r.getDen() + den * r.getNum();
15                     int b = r.getDen() * den;
16                     Rational ans = Rational(a,b);
17                     return ans;
18           }
```

In a C++ program the "+" plus sign by default means addition of two numbers. But we can overload this operator and define it for the addition of two rational numbers that results in a new rational number.

```
19          int getNum()
20          {
21                  return num;
22          }
23          int getDen()
24          {
25                  return den;
26          }
27          void print()
28          {
29                  cout << num << "/" << den;
30          }
31  private:
32          int num;
33          int den;
34  };
35  int main()
36  {
37          Rational r1(1,2);
38          Rational r2(2,3);
39          Rational r3 = r1 + r2;
40          r3.print();
41          return 0;
42  }
```

Classwork 9.3: from the previous example, write the following operators to be included in class Rational:

a) Suppose **r1** and **r2** are objects of **Rational**. Define the operator function **operator<** which return **true** if **r1<r2**; otherwise, it returns **false**.

b) Suppose **r1** is an object of **Rational**. Define the operator function **operator^** which return the rational of r1 to the power n **r1^n** is called.

The main function and the sample output is as follows:

```
int main()
{
        Rational r1(1,2);
        Rational r2(2,3);
        cout << boolalpha << (r1<r2) << endl;
        Rational r4 = r2^3;
        r4.print();
        return 0;
}
```

```
true
8/27
```

Solution to (a)

```
bool operator<(Rational r)
{
        double a = (double) num / den;
        double b = (double) r.getNum() / r.getDen();
        if (a<b) return true;
        else return false;
}
```

Solution to (b)

```
Rational operator^(int n)
{
        int a = pow(num,n);
        int b = pow(den,n);
        Rational ans = Rational(a,b);
        return ans;
}
```

Array of objects

An array of objects can be created just like an array of primitive values. For example, the following statement declares an array of 10 `rectangle` objects:

```
rectangle r[10];
```

The no-argument constructor is called to initialize each element in the array. So all the 10 `rectangle` objects defined above have the default width 1 and height 1.

Declaration and initialization of an array can also use a constructor with arguments. For example,

```
rectangle r[2] = {rectangle(1,2), rectangle(3,4)};
```

Example program 9.4 Array of rectangles

```
 1 // 9.4 Array of rectangles
 2 #include <iostream>
 3 using namespace std;
   // insert the class from example 9.1a
26 int main()
27 {
28          rectangle list[10];
29          double sum = 0;
30          for (int i=0; i<10; i++)
31          {
32                   list[i] = rectangle(i+1,i+1);
33                   sum += list[i].getArea();
34          }
35          cout << "The sum of first 10 squares is " << sum << endl;
36          return 0;
37 }
```

Recall the formula:

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{1}{6}n(n+1)(2n+1)$$

Example program 9.5 Student record

This is a very practical example useful for database management.
Define the **`Student`** class that contains:

- Two **`string`** data fields named **`name`** and **`major`** that specify the name and the major of a student.
- A **`string`** array called **`courses`** of size 10 which stores the names of the courses taken by a student.
- A **`char`** array called **`grades`** of size 10 which stores the grade of the course taken by a student.
- A **`int`** data field **`courseCount`** which stores the number of courses taken by the student.
- A constructor that creates a **`Student`** object with the specified **`name`**, and **`major`**, and **`courseCount`** to be zero.
- A function **`getGPA()`** that returns the grade point average of a **`Student`** object .
- A function **`addCourse(string course, char grade)`** that adds a course and a grade to the **`course`** array and the **`grade`** array.

```
#include <iostream>
#include <string>
using namespace std;
class Student
{
public:
        Student(string n, string m)
        {
                name = n;
                major = m;
                courseCount = 0;
        }
        double getGPA()
        {
                double gpa = 0;
                for (int i=0; i<courseCount; i++)
                {
                        switch(grades[i])
                        {
                                case 'A': gpa += 4; break;
                                case 'B': gpa += 3; break;
                                case 'C': gpa += 2; break;
                                case 'D': gpa += 1; break;
                                case 'F': gpa += 0; break;
                        }
                }
                return gpa / courseCount;
        }
```

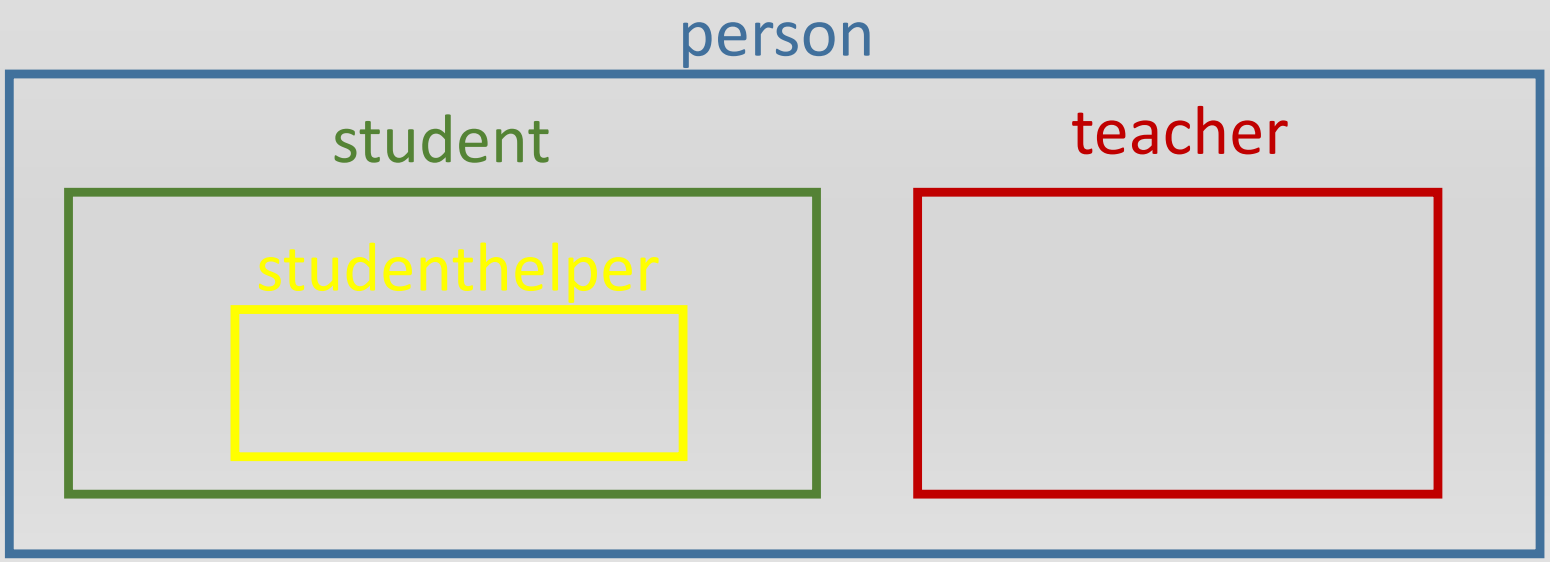
```
29          void addCourse(string c, char g)
30          {
31                  courses[courseCount] = c;
32                  grades[courseCount] = g;
33                  courseCount++;
34          }
35  private:
36          string name;
37          string major;
38          string courses[10];
39          char grades[10];
40          int courseCount;
41  };
42  int main()
43  {
44          Student s("John","Data Science");
45          s.addCourse("Principle of Programming",'A');
46          s.addCourse("Econometrics",'B');
47          cout << s.getGPA();
48          return 0;
49  }
```

The program creates a Student object with the name John and major Investment Science who has taking two courses: Principle of Programming with grade A and Econometrics with grade B; and output the student's grade point average.

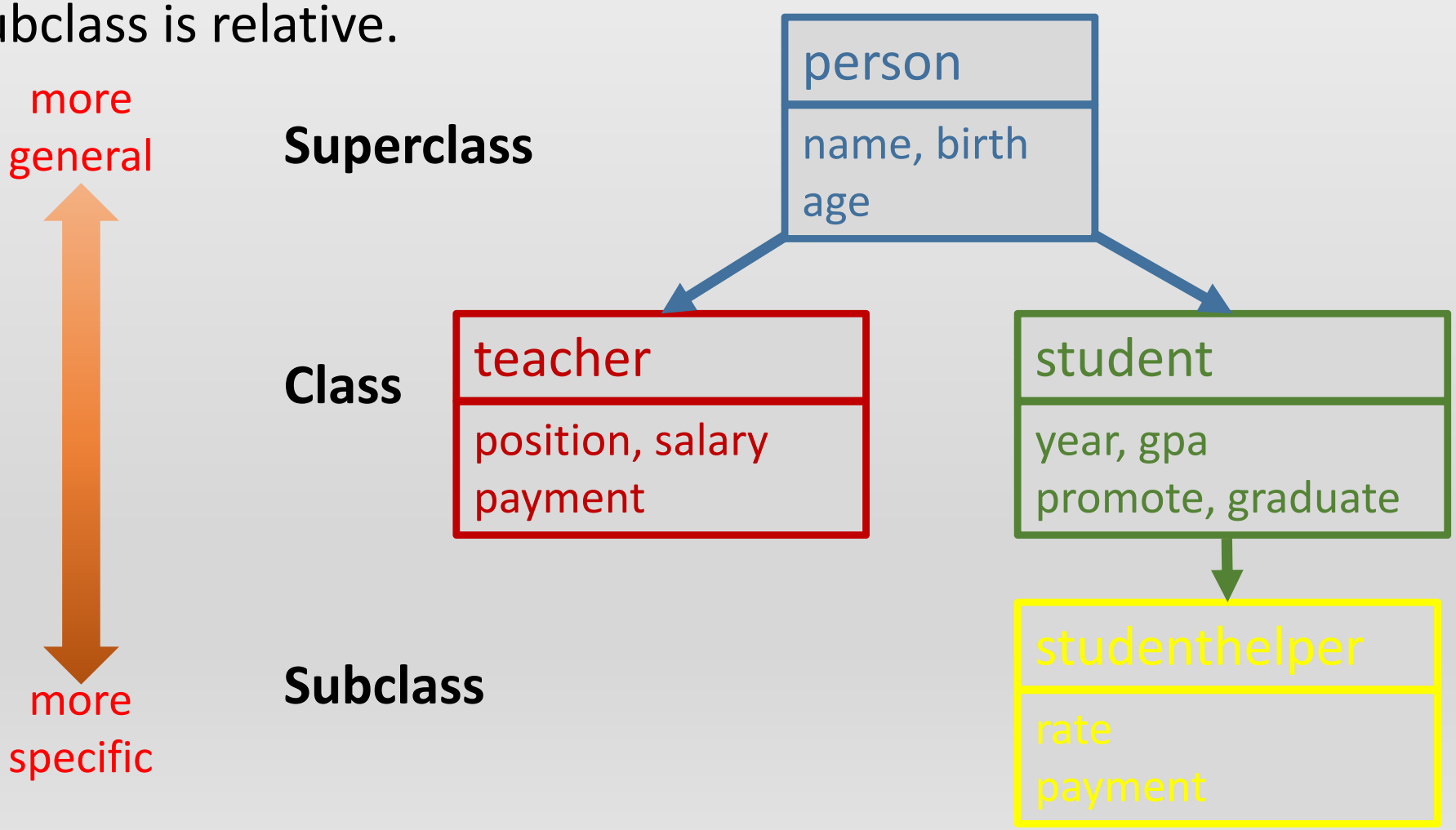
Class inheritance

Inheritance is a core feature of Object-Oriented Programming. It allows one class to inherit data or behaviour from another class and is one of the key ways in which reuse is enabled within classes. The advantage of using inheritance is to make coding more efficient and easier to maintain for any updates in future.

To illustrate this idea, we will use the example below. The Euler's diagram illustrates the relationship between different sets.

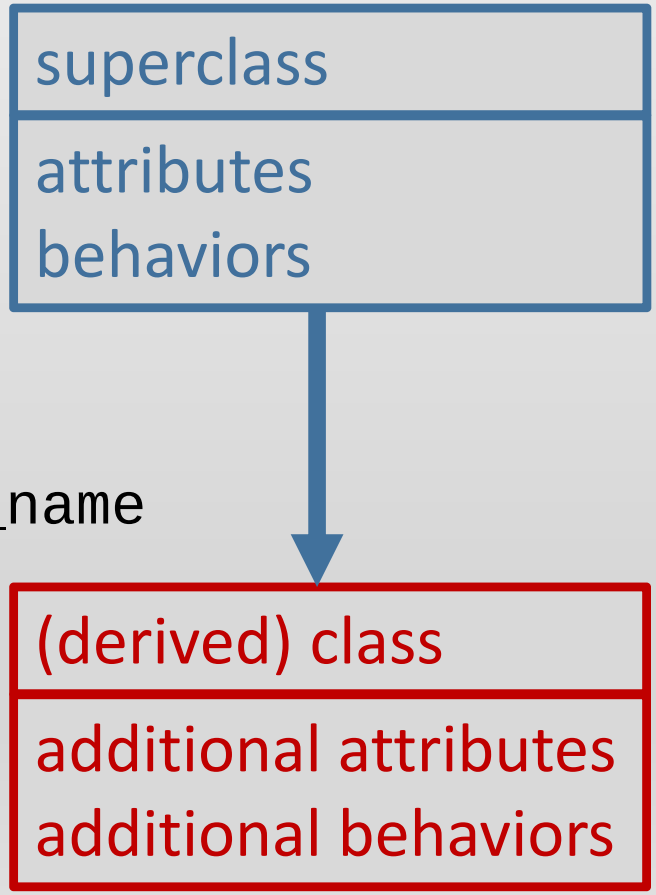


In object-oriented programming, we would consider each set as a class and each element as an object. However, an object can only belong to one class. The tree diagram below illustrates the relationship between different classes. Notice that a class can inherit data from another class. The relationship of superclass and subclass is relative.



The syntax of defining a class derived from a superclass is shown below. The superclass is indicated after a colon. Notice that the constructors in the derived class will call the default constructor of the superclass.

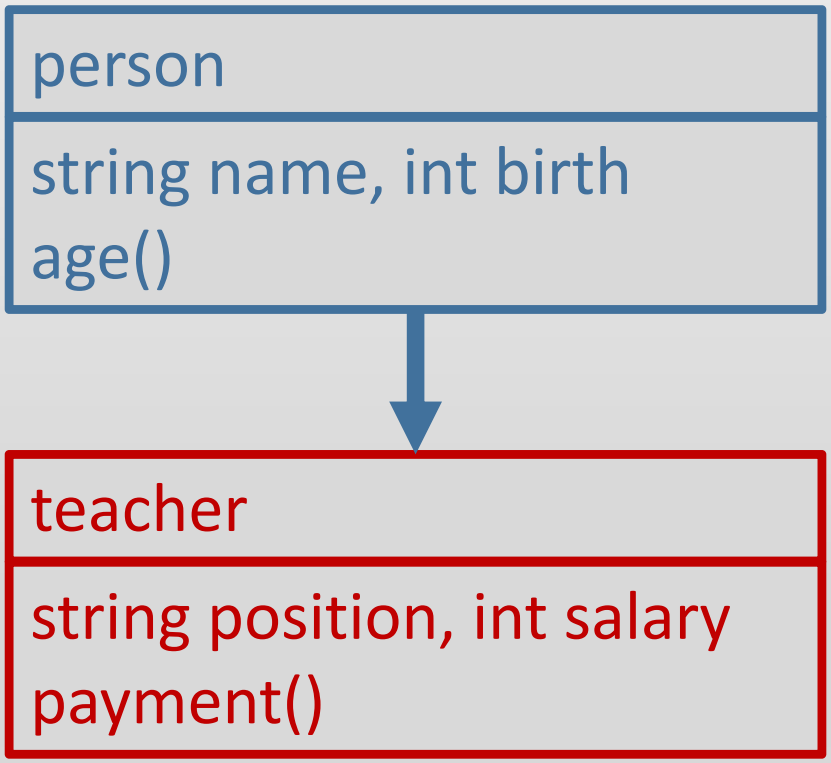
```
class superclass_name
{
   attributes
   behaviors
   default_constructor
   parametrized_constructor
};

class class_name: public superclass_name
{
   attributes
   behaviors
   default_constructor
   parametrized_constructor
};
```



Example 9.6 Class of teacher

Write a class `person` and its subclass `teacher`. The classes should include the attributes and behaviours below, and also the default and parametrized constructors.

Attributes:
Behaviors:



additional attributes:
additional behaviors:

```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4 class person
 5 {
 6 public:
 7         string name;
 8         int birth;
 9         person()
10         {
11                 name = "unknown";
12                 birth = 0;
13         }
14         person(string n, int b)
15         {
16                 name = n;
17                 birth = b;
18         }
19         int age()
20         {
21                 return 2023-birth;
22         }
23 };
```

data field of the superclass person, also inherited to the teacher class.

default constructor of person, called when a teacher class object is constructed. **(necessary for class inheritance)**

parametrized constructor of person, not called in the teacher class constructor.

return the age of the person in 2023

```
24 class teacher: public person
25 {
26 public:
27         string position;
28         int salary;
29         teacher(string n, int b, string p, int s)
30         {
31                 name = n;
32                 birth = b;
33                 position = p;
34                 salary = s;
35         }
36         void payment()
37         {
38                 cout << "$" << salary << " is transferred to "
39                 << name << "'s account." << endl;
40         }
41 };
42 int main()
43 {
44         teacher t1("Peter", 1973, "professor", 90000);
45         cout << t1.name << " is " << t1.age() << " years old." << endl;
46         cout << t1.name << " is a " << t1.position << endl;
47         t1.payment();
48         return 0;
49 }
```

derive teacher class under the superclass person

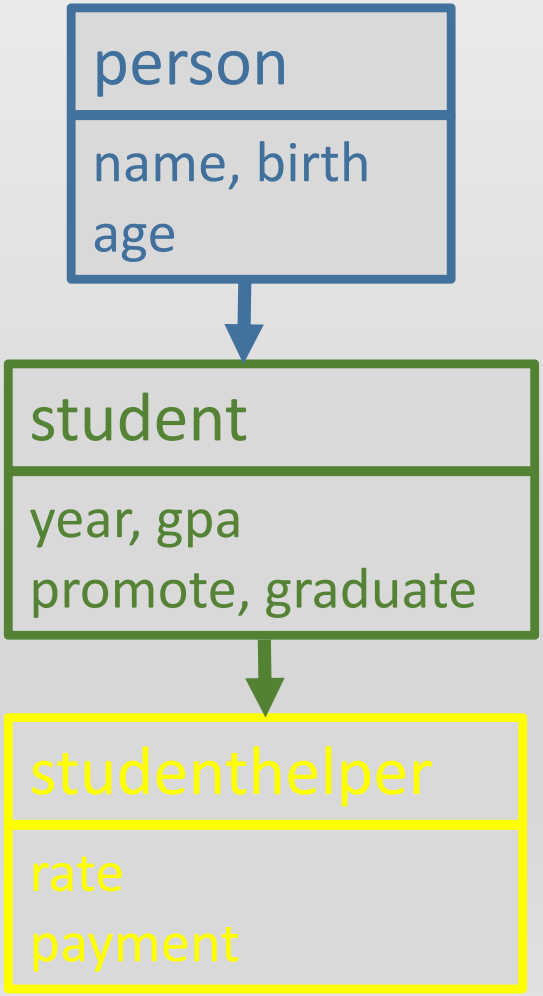
data field in addition to the superclass

the default constructor of person class is called first, then update the attributes from input parameters

make use of name and salary from the data field of person and teacher respectively

The hierarchy of class inheritance can be extended to more than two classes. This example illustrates a superclass-class-subclass relationship between three classes.

Example 9.7 Class of `student` and `studenthelper`.



```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4 class person
 5 {
 6 public:
 7         string name;
 8         int birth;
 9         person()
10         {
11                 name = "unknown";
12                 birth = 0;
13         }
14         person(string n, int b)
15         {
16                 name = n;
17                 birth = b;
18         }
19         int age()
20         {
21                 return 2023-birth;
22         }
23 };
```

We will directly use the person class written in the previous example program.

data field of the superclass person, also inherited to the student class.

default constructor of person, called when a student class object is constructed. **(necessary for class inheritance)**

```
24 class student: public person
25 {
26 public:
27         double gpa;
28         student()
29         {
30                 name = "unknown";
31                 birth = 0;
32                 gpa = 0;
33         }
34         student(string n, int b, double g)
35         {
36                 name = n;
37                 birth = b;
38                 gpa = g;
39         }
40 };
```

derive student class under the superclass person

data field in addition to the superclass

default constructor of student, called when a studenthelper class object is constructed.
(necessary for class inheritance)

the default constructor of person class is called first, then update the attributes from input parameters

```
41 class studenthelper: public student
42 {
43 public:
44         int rate;
45         studenthelper(string n, int b, double g, int r)
46         {
47                 name = n;
48                 birth = b;
49                 gpa = g;
50                 rate = r;
51         }
52         void payment(int t)
53         {
54                 cout << "$" << rate*t << " is transferred to "
55                 << name << "'s account." << endl;
56         }
57 };
```

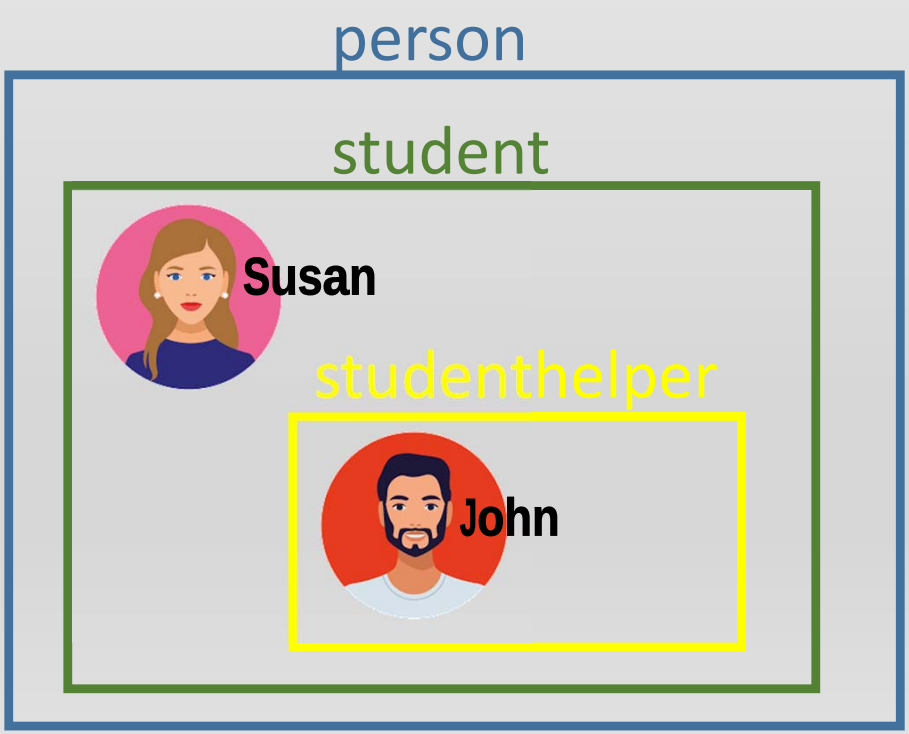
derive studenthelper class under the superclass student

data field in addition to the superclass

the default constructor of student class is called first, then update the attributes from input parameters

evaluate the payment based on hourly rate times the number of hours

```
58 int main()
59 {
60          student s1("Susan", 2003, 3.7);
61          cout << s1.name << " is " << s1.age() << " years old." << endl;
62          cout << s1.name << " has gpa " << s1.gpa << endl;
63          studenthelper s2("John", 2001, 2.8, 70);
64          cout << s2.name << " is " << s2.age() << " years old." << endl;
65          cout << s2.name << " has gpa " << s2.gpa << endl;
66          s2.payment(10);
67          return 0;
68 }
```



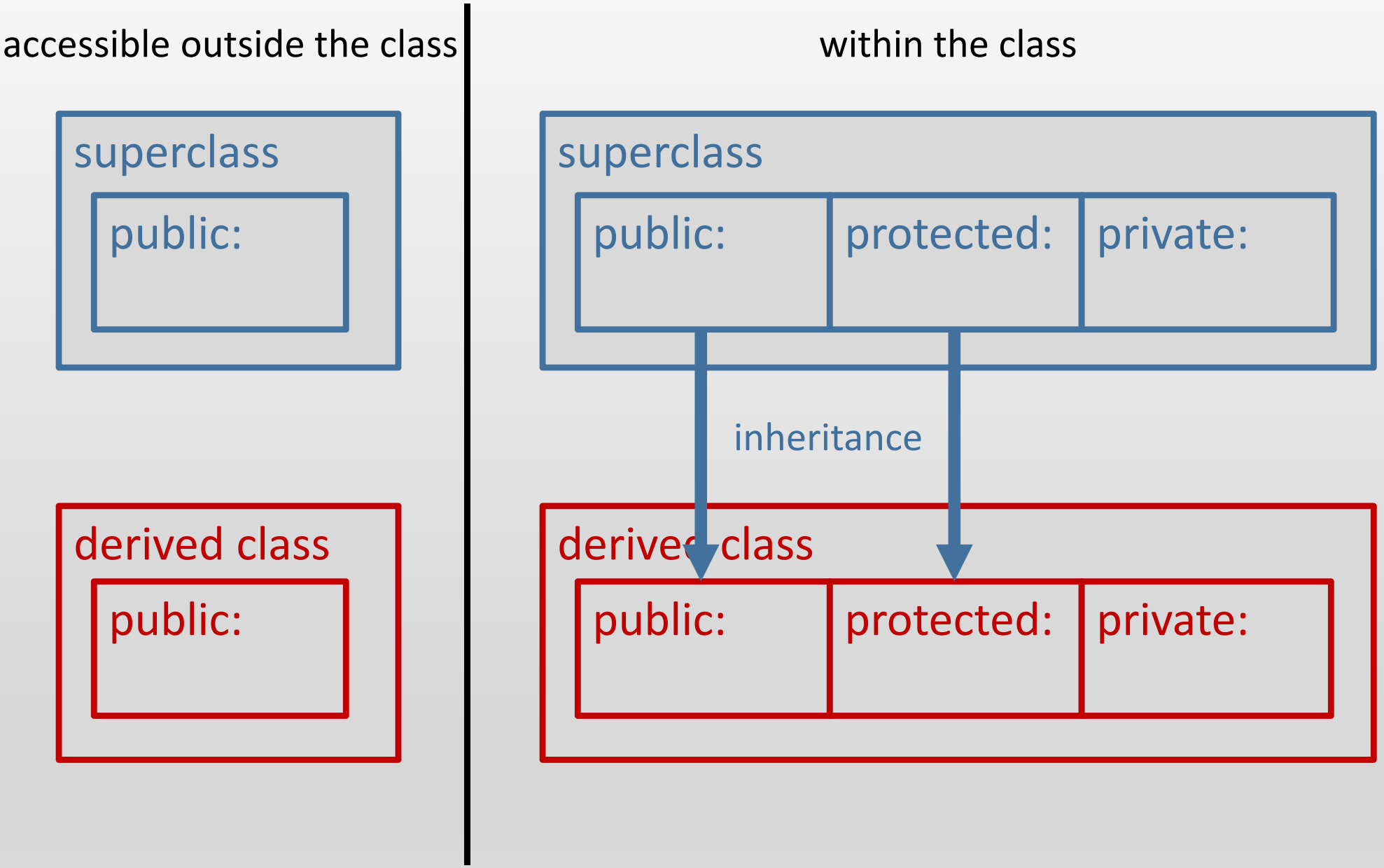

```
Susan is 20 years old.
Susan has gpa 3.7
John is 22 years old.
John has gpa 2.8
$700 is transferred to John's account.
```

Data encapsulation under inheritance

In defining a class, we can put some attributes or behaviors under the private section so that they cannot be accessed or modified directly from outside. This feature however, may cause some trouble in class inheritance. Attributes or behaviors in the private section of the superclass cannot be inherited to the derived class.

To remedy this problem, we can partition the superclass into public, private and protected sections.

- **Public**: accessible in the whole program
- **Private**: accessible only within the class
- **Protected**: accessible only within the class and its derived classes


accessible outside the class
within the class
superclass
public:
derived class
public:
superclass
public:
protected:
private:
inheritance
derived class
public:
protected:
private:

Example 9.8 Class of teacher (with data encapsulation)

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class person {
5  private:
6           string name;
7           int birth;
8  public:
9           person(){
10                  name = "unknown";
11                  birth = 0;
12          }
13          person(string n, int b){
14                  name = n;
15                  birth = b;
16          }
17          int age(){
18                  return 2023-birth;
19          }
20          string getname(){
21                  return name;
22          }
23 };
```

We will modify the person class written in the previous example program with data encapsulation.

two attributes in the data field are encapsulated, they cannot be accessed directly outside the class

since the name is encapsulated, need to use a getname function to retrieve the name

```
24 class teacher: public person {
25 public:
26          string position;
27          int salary;
28          teacher(string n, int b, string p, int s){
29                    name = n;
30                    birth = b;
31                    position = p;
32                    salary = s;
33          }
34          void payment(){
35                    cout << "$" << salary << " is transferred to "
36                    << name << "'s account." << endl;
37          }
38 };
39 int main()
40 {
41          teacher t1("Peter", 1973, "professor", 90000);
42          cout << t1.getname() << " is " << t1.age() << " years old." << endl;
43          cout << t1.getname() << " is a " << t1.position << endl;
44          t1.payment();
45          return 0;
46 }
```

The name attribute is stored in the private section of the person class and it is not inherited to the teacher class. Therefore it doesn't make sense to apply getname function to a teacher class object.

```
4 class person {
5 private:
6           string name;
7           int birth;
```

```
4 class person {
5 protected:
6           string name;
7           int birth;
```

To remedy the problem, we can simply put the data field into the protected section. The name attribute from person class will be inherited to the derived class teacher. And it is not accessible from outside

Multiple inheritance

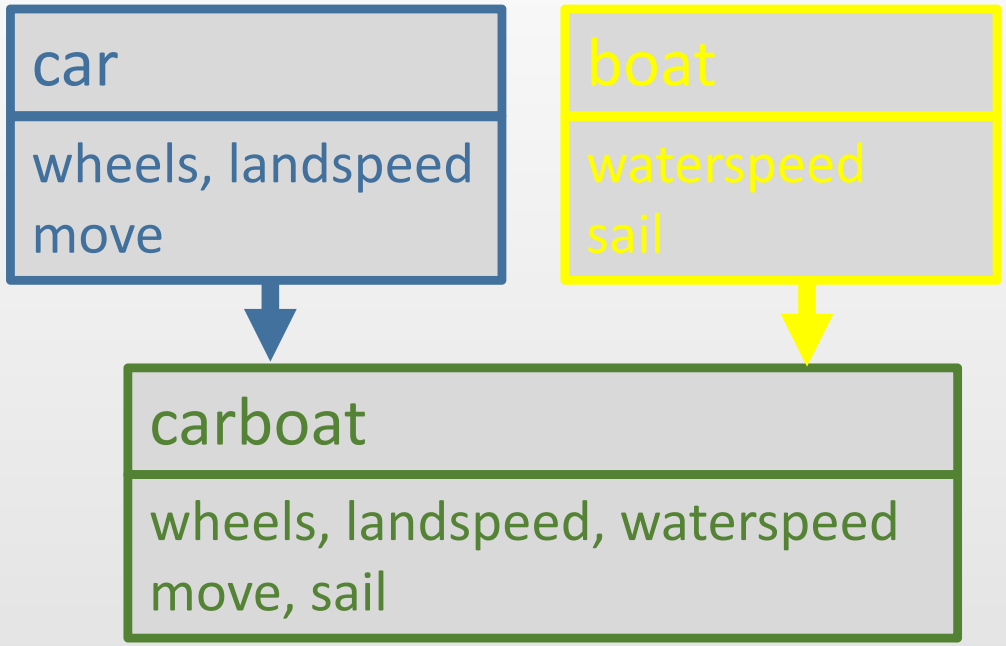
In OOP, a class might inherit data from several superclasses, in other words, it is the subclass of more than one class. This is called **multiple inheritance**. To illustrate this idea, consider the following.

Example 9.9 Carboat

Consider the sets of cars and boats. Their intersection is a set of carboats (or boatcars) as shown in the Venn's diagram:

`car` contains two attributes `wheels`, `landspeed` and one behavior `move`.

`boat` contains one attributes `waterspeed` and one behavior `sail`.



The tree diagram shows the inheritance relation between the classes. Notice that all the attributes and behaviors in carboat are inherited from its superclasses.

To avoid ambiguity, we need to distinguish the two attributes `landspeed` and `waterspeed`, also the two behaviors `move` and `sail`. Otherwise the program cannot refer to the designated superclass accordingly.

```
1  #include<iostream>
2  using namespace std;
3  class car
4  {
5  public:
6          int wheels;
7          double landspeed;
8          car()
9          {
10                 wheels = 0;
11                 landspeed = 0;
12         }
13         car(int w, double s)
14         {
15                 wheels = w;
16                 landspeed = s;
17         }
18         void move()
19         {
20                 cout << "moving with (km/h)"
21                 << landspeed << endl;
22         }
23 };
```

data field of the superclass car, also inherited to the carboat class

default constructor of car, called when a carboat class object is constructed. **(necessary for class inheritance)**

```
24 class boat
25 {
26 public:
27         double waterspeed;
28         boat()
29         {
30                 waterspeed = 0;
31         }
32         boat(double s)
33         {
34                 waterspeed = s;
35         }
36         void sail()
37         {
38                 cout << "sailing with (km/h)"
39                 << waterspeed << endl;
40         }
41 };
```

data field of the superclass boat, also inherited to the carboat class

default constructor of boat, called when a carboat class object is constructed. **(necessary for class inheritance)**

```
42 class carboat: public car, public boat
43 {
44 public:
45          carboat(int w, double ls, double ws)
46          {
47                    wheels = w;
48                    landspeed = ls;
49                    waterspeed = ws;
50          }
51 };
52 int main()
53 {
54          car x(4,70);
55          x.move();
56          boat y(40);
57          y.sail();
58          carboat z(4,50,30);
59          z.move();
60          z.sail();
61          return 0;
62 }
```

derive carboat class under two superclasses car and boat

the default constructors of car and boat classes are called, then update the attributes from input parameters

```
moving with (km/h)70
sailing with (km/h)40
moving with (km/h)50
sailing with (km/h)30
```

a carboat named z proceeds both attributes of a car and a boat, it is able to move and sail