

EEE102

C++ Programming and Software Engineering II

Lecture 4 Classes and Objects II

Dr. Rui Lin/Dr. Fei Xue

Rui.Lin/Fei.Xue@xjtlu.edu.cn

Room EE512/EE222



Outline

- An example
- Constructor & Destructor
 - Default, normal and *copy constructor*
 - When to call constructors and destructors?
- Class composition
- **const** objects and methods
- Separate compilation
- UML description of classes and objects



An example of complex number class

```
class complexClass
{
private:
    int x;
    int y;
public:
    complexClass() Inline definition
    {
        x=0;
        y=0;
    }
    void set(int a, int b);
};

void complexClass::set(int a,int b)
{
    x=a;
    y=b;
}
```

```
int main()
{
    int num1=-9, num2=5;
    complexClass c1;
    c1.set(num1,num2);
    c1.x+=1;c1.y--; X
    return 0;
}
```

1. In the function member of the class, data members x and y can be directly used without dot operator.
2. Outside the class (eg. In main function), the data members cannot be accessed.



An example of complex number structure

```
struct complexStruct
{
    int x;
    int y;
};

complexStruct set(int a, int b)
{
    complexStruct com;
    com.x=a;
    com.y=b;
    return com;
}
```

```
int main()
{
    int num1=-9, num2=5;
    complexClass c1;
    c1.set(num1,num2);
    c1.x+=1;c1.y--; ✗

    complexStruct s1;
    s1=set(num1, num2);
    s1.x+=1;s1.y--; ✓
    return 0;
}
```

1. In the external function set(), data members x and y must be accessed using the dot operator.
2. Outside the class (eg. In main function), the data members can be accessed with the dot operator.



```
class complexClass
{
    double x;
    double y;
public:
    complexClass()
    { x=0; y=0; }
    void set(int a, int b);
};
```

```
struct complexStruct
{
    double x;
    double y;
};
```

```
void complexClass::set(int a, int b)
{
    x=a;
    y=b;
}
```

```
complexStruct set(int a, int b)
{
    complexStruct com;
    com.x=a;
    com.y=b;
    return com;
}
```

```
double set(int a, int b)
{
    double val;
    val=sqrt(a*a+b*b);
    return val;
}
```

These three functions have the same names and parameter lists, will they cause **confliction**?

Answer: The second and third will, but the first one won't. Because it is a function member of complexClass, which will only be called like: cNum.set(1,5);

2. More about constructors

- Constructor: A special method which initialise the objects.

- Default constructor

```
complexClass() {x=0; y=0;}
```

```
complexClass() { }
```

- Implicit called while declaring an object

```
complexClass cNum1;
```

- Normal constructor (Parameterised constructor)

```
complexClass(double r, double i) {x=r; y=i;}
```

- Implicit called while initialising an object

```
complexClass cNum1(1,2);
```

- Explicit called while initialising an object

```
complexClass cNum1=complexClass(1,2);
```



Questions

- 1. Is constructor always needed?
 - Implicit constructor
- 2. Can we define normal constructor only?
 - No.
- 3. For built-in datatype, it is valid to initialise a variable from existing variable. Can we do this for objects?
 - Copy constructor

```
int a=10;  
int b=a;
```

```
complexClass cNum1 (1,2) ;  
complexClass cNum2 (cNum1) ;
```



Copy Constructor

To use the data members of an existing object to initialise an object of the same class **when it is declared.**

```
class complexClass
{
    double x;
    double y;
public:
    // Default constructor
    complexClass() {}
    // Normal constructor
    complexClass(double r, double i)
    { x=r; y=i; }
    // Copy constructor
    complexClass(complexClass &cNum)
    { x=cNum.x; y=cNum.y; }
};
```

```
int main()
{
    double num1=-9, num2=5;
    complexClass c1;
    complexClass c2(num1, num2);
    complexClass c3(c2);
    return 0;
}
```

No reference sign when calling the copy constructor

Critical --- using reference in parameter declaration.



Reference &

- Declaration

int m; int &n = m;

float w1;

float &w2=w1;

- Rules:

(1) Any reference should be initialized when declared (pointer can be initialized anytime) 。

(2) Reference must be associated with some legal variables。

(3) Once initialized, reference variable cannot directed to other variables
(Pointer can be pointed to other variables if necessary) 。

int i = 5;

int j = 6;

int &k = i;

int &w; // This is wrong, must be initialized.

k = j; // both k and i will be equal to 6;



Reference and Pointer

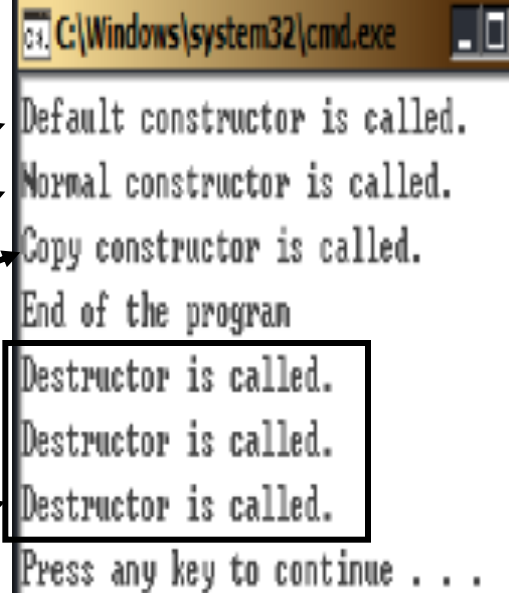
- `void Func1(int x) { x = x + 10; } ...`
`void main()`
`{ int n = 0; Func1(n); cout << "n = " << n << endl; // n = 0 }`
- `void Func2(int *x) { (* x) = (* x) + 10; }`
`void main() { int n = 0; Func2(&n); cout << "n = " << n; // n = 10 }`
- `void Func3(int &x) { x = x + 10; }`
`void main() { int n = 0; Func3(n); cout << "n = " << // n = 10 }`
- **Reference can achieve the same effect as pointer**



When will the constructor and destructor be called?

```
class complexClass
{
    double x;
    double y;
public:
    complexClass()
    {   x=0; y=0;
        cout<<"Default constructor is called."<<endl;}
    complexClass(double r, double i=0)
    {   x=r; y=i;
        cout<<"Normal constructor is called."<<endl;}
    complexClass(complexClass &cNum)
    {   x=cNum.x; y=cNum.y;
        cout<<"Copy constructor is called."<<endl;}
    ~complexClass()
    {   cout<<"Destructor is called."<<endl;}
};

void main()
{
    complexClass c1;
    complexClass c2(-9,5);
    complexClass c3(c2);
    cout<<"End of the program"<<endl;
}
```



C:\Windows\system32\cmd.exe

Default constructor is called.
Normal constructor is called.
Copy constructor is called.
End of the program
Destructor is called.
Destructor is called.
Destructor is called.
Press any key to continue . . .

```

class complexClass
{...};
double complexClass::real(void) // get real part
{
    return x;}
double complexClass:: imag(void) // get imaginary part
{
    return y;}
bool isCEqual(complexClass cNum1, complexClass cNum2)
{
    bool temp;
    if((cNum1.real()==cNum2.real()) && (cNum1.imag()==cNum2.imag()))
        temp=1;
    else
        temp=0;
    cout<<"End of compare functions."<<endl;
    return temp;
}
void main()
{
    complexClass c2(-9,5);
    complexClass c3(9,5);
    if (isCEqual(c2,c3))
        cout<<"c2 equals to c3"<<endl;
    else
        cout<<"c2 doesn't equal to c3"<<endl;
    cout<<"End of the program"<<endl;
}

```

C:\Windows\system32\cmd.exe

```

Normal constructor is called.
Normal constructor is called.
Copy constructor is called.
Copy constructor is called.
End of compare functions.
Destructors are called.
Destructors are called.
c2 doesn't equal to c3
End of the program
Destructors are called.
Destructors are called.
Press any key to continue . . .


```

Constructors with default arguments

- C++ allows calling a function without specifying all argument.
 - A default value to the parameter will be assigned;
 - The default value should be specified in the function declaration;
 - The default values are added *from right to left*.

```
class complexClass
{
    double x;
    double y;
public:
    complexClass() {}
    complexClass(double r, double i=0);
};
complexClass::complexClass(double r, double i)
{
    x=r;
    y=i;
}
```

```
int main()
{
    double a=-9;
    complexClass c2(a);
    return 0;
}
```



Multiple constructors in one class

- Default constructor, normal constructor and copy constructor share the same name;
- Which one to call is determined by the argument in declaration statement.

```
class complexClass
{
    double x;
    double y;
public:
    // Default constructor
    complexClass() {}
    // Normal constructor
    complexClass(double r, double i)
    { x=r; y=i; }

    // Copy constructor
    complexClass(complexClass &cNum)
    { x=cNum.x; y=cNum.y; }
};
```

```
int main()
```

```
double num1=-9, num2=5;
```

```
complexClass c1;
```

```
complexClass c2(num1, num2);
```

```
complexClass c3(c2);
```

```
return 0;
```

No Argument

Two double Arguments

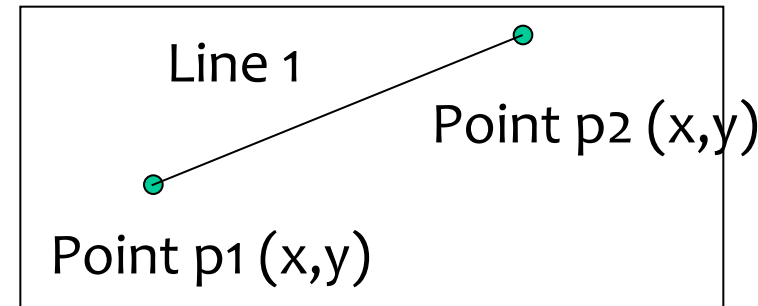
One complexClass Argument

Multiple functions sharing the same name
– *function overloading*



3. Class Composition

- Class composition
 - Class A contains the objects of class B.
 - Class A: Line
 - Class B: Point



- When initialising a composite class (class A):
 - The embedded objects have to be initialised first
 - Syntax:

```
ClassName::ClassName(parameter list):embeddedObject1(parameter list),  
embeddedObject2(parameter list).....  
{  
    Class A initialisation;  
}
```



Class Composition

```
class Point
{
    int x,y;
public:
    Point(int xx=0, int yy=0)
    {
        x=xx;y=yy;}
    Point (Point &p);
    int GetX()
    {
        return x;}
    int GetY()
    {
        return y;}
};

Point::Point(Point &p)
{
    x=p.x;
    y=p.y;
    cout<<"Point copy
constructor is called."<<endl;
}
```

```
class Line
{
    public:
        Line(Point xp1, Point xp2);
        Line(Line &L);
        double GetLen()
        {
            return len;}
    private:
        Point p1,p2;
        double len;
};

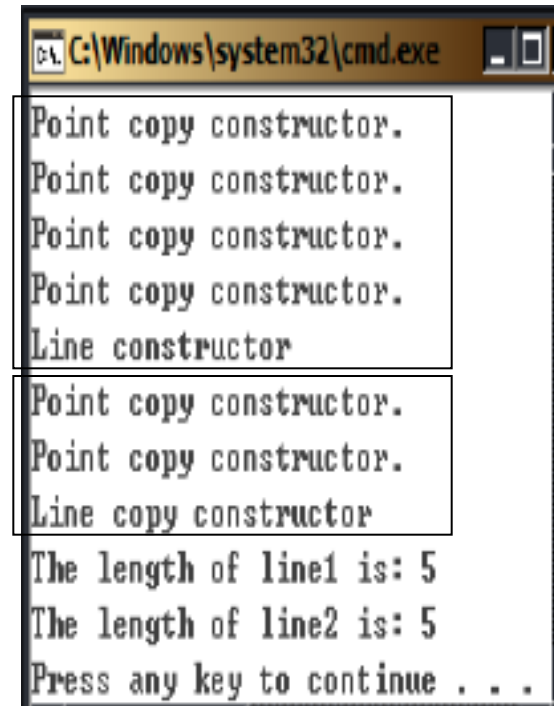
// Constructor of combined class
Line::Line(Point xp1, Point
xp2):p1(xp1),p2(xp2)
{
    cout<<"Line constructor is called"<<endl;
    double x=p1.GetX()-p2.GetX();
    double y=p1.GetY()-p2.GetY();
    len=sqrt(x*x+y*y);
}

Line::Line(Line &L):p1(L.p1),p2(L.p2)
{
    cout<<"Line copy constructor is called"<<endl;
    len=L.len;
}
```



Constructor and destructor in composite class

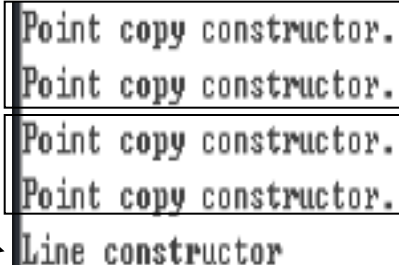
```
int main()
{
    Point myp1(1,1), myp2(4,5);
    Line line1(myp1, myp2);
    Line line2(line1);
    cout <<"The length of line1 is: ";
    cout <<line1.GetLen()<<endl;
    cout <<"The length of line2 is: ";
    cout <<line2.GetLen()<<endl;
}
```



```
C:\Windows\system32\cmd.exe
Point copy constructor.
Point copy constructor.
Point copy constructor.
Point copy constructor.
Line constructor
Point copy constructor.
Point copy constructor.
Line copy constructor
The length of line1 is: 5
The length of line2 is: 5
Press any key to continue . . .
```

`Line::Line(Point xp1, Point xp2):p1(xp1),p2(xp2)`

1. `myp1 -> xp1; myp2 -> xp2`
2. `xp1 -> p1; xp2 -> p2`
3. create line object



```
Point copy constructor.
Point copy constructor.
Point copy constructor.
Point copy constructor.
Line constructor
```



4.1 `const` Objects

Least privilege is one of the most fundamental principles of good software engineering.

- **`const` Objects**

- For some objects that do not need to be modified, we can use keyword **`const`** to specify that an object is not modifiable and that any attempt to modify the object should be resulted in a compilation error.
- For example, the **Origin** is a special **Point** whose coordinates are always **(0,0)**, unmodifiable:

```
const Point Origin(0,0) ;
```



4.2 const Methods

C++ compiler disallows a member function calls for **const** objects unless the member functions themselves are also declared **const**.

```
class Point
{
    int x,y;
public:
    Point(int xx=0, int yy=0);
    int GetX();
    int GetY();
};
Point::Point(int xx=0, int yy=0)
{
    x=xx;y=yy;}
int Point::GetX()
{
    return x;}
int Point::GetY()
{
    return y;}

int main()
{
    const Point Origin(0,0);
    a = Origin.GetX();//invalid
    return 0;
}
```

```
class Point
{
    int x,y;
public:
    Point(int xx=0, int yy=0);
    int GetX() const;
    int GetY() const;
};
Point::Point(int xx=0, int yy=0)
{
    x=xx;y=yy;}
int Point::GetX() const
{
    return x;}
int Point::GetY() const
{
    return y;}

int main()
{
    const Point Origin(0,0);
    a = Origin.GetX(); //valid
    return 0;
}
```

5. Separate compilation

- It is necessary to split the code into separate source files
 - programs get larger
 - you work in a team
 - Reason
 - For separated files, only the ones changed will be recompiled
 - For group work, people work on individual files
- Files communicated through the inclusion of header files.

Header file (*.h) contains

- definitions of classes.
- declarations of constants.
- declarations of nonmember functions.
- declarations of global variables.

Source file (*.cpp) contains

- definitions of member functions.
- definitions of nonmember functions.
- definitions of global variables.



.h, .cpp and the main source file

```
/* Header file "point.h" */
#ifndef POINT_H
#define POINT_H
class Point
{
    int x,y;
public:
    Point();
    Point(int xx, int yy);
    Point (Point &p);
    void set(int xx, int yy)
    { x=xx;y=yy;}
    int GetX() const
    { return x;}
    int GetY() const
    { return y;}
};

#endif
```

```
/* Header file "point.cpp" */
#include<iostream>
#include"point.h"
using namespace std;

Point::Point()
{
    x=0;y=0;}
Point::Point(int xx, int yy)
{
    x=xx;y=yy;}
Point::Point(Point &p)
{
    x=p.x;y=p.y;}
```

```
#include<iostream>
using namespace std;
#include"point.h"

int main()
{
    Point myp1(1,1), myp2;
    return 0;
}
```



Conditional Compilation Directives

- **#ifdef** : if defined
- **#ifndef** : if not defined
- Syntax:

```
#ifndef _MACRO_NAME
#define _MACRO_NAME
statements
#endif
```
- Meaning: If **_MACRO_NAME** has not been previously defined in a **#define** statement, the block of code will be compiled.
- **#ifndef** and **#define** are used together to avoid the redefinition of classes.
 - The class will only be defined when it has not been defined before.



Example

```
/* Header file "point.h" */
#ifndef POINT_H
#define POINT_H
class Point
{ . . . . . };
#endif
```

```
/* Header file "point.cpp" */
#include<iostream>
#include"point.h"
using namespace std;
. . . . .
```

```
/* Header file "line.h" */
#ifndef LINE_H
#define LINE_H
#include"point.h"
class line
{
    Point p1,p2;
    double len;
public:
    Line(Point xp1, Point xp2);
};
#endif
```

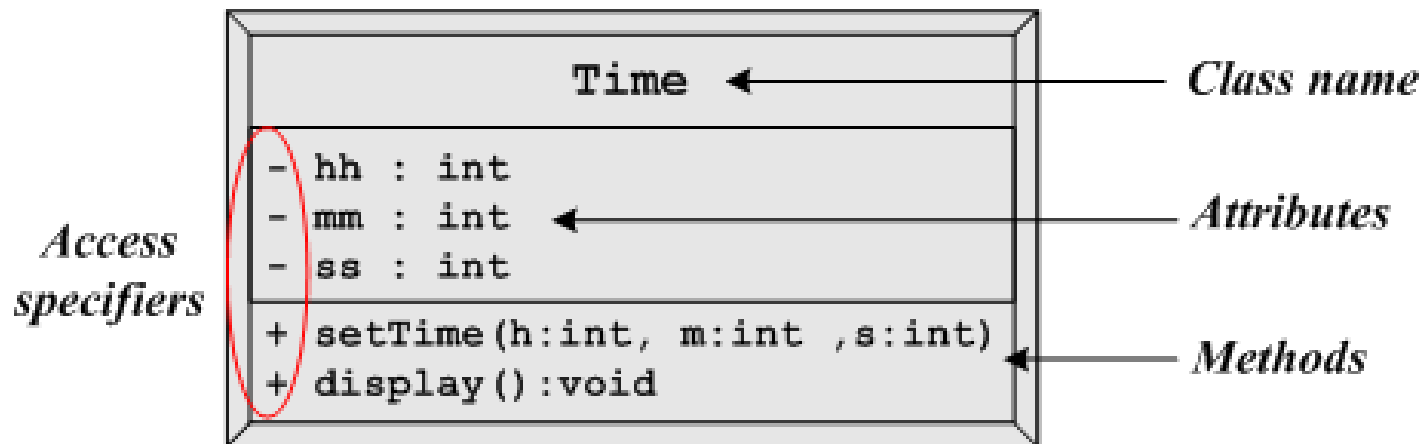
```
/* Header file "line.cpp" */
#include<iostream>
#include<cmath>
#include"line.h"
using namespace std;
Line::Line(Point xp1, Point xp2):
p1(xp1),p2(xp2)
{
    double x=p1.GetX()-p2.GetX();
    double y=p1.GetY()-p2.GetY();
    len=sqrt(x*x+y*y);
}
```

```
#include<iostream>
using namespace std;
#include "point.h"
#include "line.h"
int main()
{
    Point myp1(1,1), myp2;
    Line line1(myp1, myp2);
    return 0;
}
```

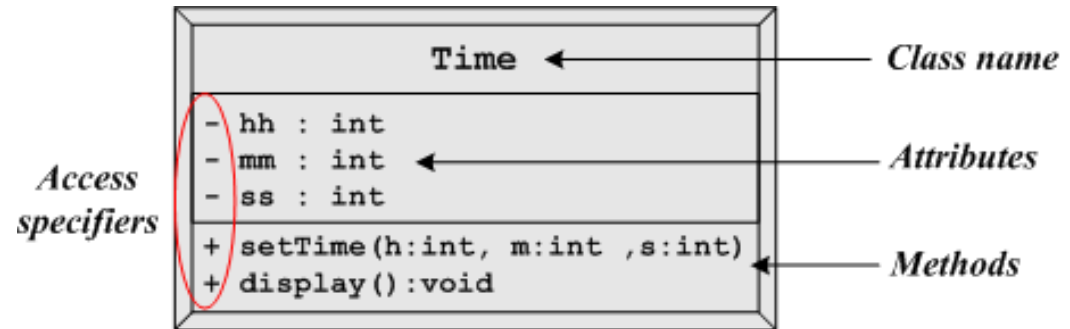


6. Unified Modelling Language (UML)

- Unified Modelling Language (UML) was developed as a standardized general-purpose graphical modelling language in the field of object-oriented software engineering.
 - UML includes a set of graphical notation techniques to create visual models of object-oriented software-intensive systems.
 - Class diagram: the rectangle illustrating a class is divided into three compartments, with the class name in the top, attributes (data members) in the middle, and methods (function members) in the bottom.



6. Unified Modelling Language (UML)



- Several simple rules to draw a standard UML class diagram:
 - To distinguish the attributes from the methods, add parentheses after function names, such as **display()**.
 - To indicate the datatype of attributes and methods, using the formal UML format is *attribute:type_name*, for example: **hh:int**.
 - To specify parameter and return types of a method, put the parameter type after each parameter, and the return type after the function name. **void** can be left out.
 - Sometimes it is useful to illustrate the accessibility of the members. UML using ‘-’ for private member, and ‘+’ for public members.

6. Unified Modelling Language (UML)

- Use UML to illustrate the class composition

