# COMP 322 Lecture 8 - Classes and inheritance

Junji Duan

2024/3/1

## Today's Outline

- Friendship
- Inheritance
- Construction/Destruction order
- Types of inheritance
- Is-a VS Has-a
- Virtual methods
- Abstract classes

## Classes - Behind the scenes

- How many methods does the following class have?

```
class SomeAwesomeClass
{

};
```

- Prior to C++11, the compiler would provide 4 methods for you unless you explicitly define them yourself:
  - Default constructor
  - Default destructor
  - Copy constructor
  - Copy assignment operator
- Since C++11, compiler will also generate 2 extra methods (so total now is 6):
  - Move constructor
  - Move assignment operator
- Probably other methods were being added in C++20

- Default constructor
- Default destructor
- Copy constructor
- Copy assignment operator

```
class SomeAwesomeClass
{

};

int main()
{
    SomeAwesomeClass sac1;
    SomeAwesomeClass sac2 = sac1;
    SomeAwesomeClass sac3(sac2);
    SomeAwesomeClass sac4;
    sac4 = sac1;
}
```

## Classes - Copy Constructor

- SomeAwesomeClass( const SomeAwesomeClass & obj);
  - Instantiate and Initialize an object from another object having the same type
  - In Java we can obtain similar behavior by simply inheriting from "Cloneable" (however the way Cloneable works is very different from C++ copy constructor)

```
class SomeAwesomeClass
{

};

int main()
{
    SomeAwesomeClass sac1;
    SomeAwesomeClass sac2 = sac1;
    SomeAwesomeClass sac3(sac2);

}
```

## Classes - Copy Assignment Operator

- SomeAwesomeClass & operator= ( const SomeAwesomeClass & obj);
  - Assign an object from another object having the same type

```
class SomeAwesomeClass
{

};

int main()
{
    SomeAwesomeClass sac1;
    SomeAwesomeClass sac2 = sac1;
    SomeAwesomeClass sac3(sac2);
    SomeAwesomeClass sac4;
    sac4 = sac1;
}
```

## Classes - friends

```
class GPS
{
public:
    GPS(double altitude, double longitude, double latitude):
        altitude(altitude),
        longitude(longitude),
        latitude(latitude)
    {
        cout << "GPS Constructor" << endl;
    }

    ~GPS()
    {
        cout << "GPS Destructor" << endl;
    }

    friend void setLongitude(GPS& gps);

private:
    double altitude;
    double longitude;
    double latitude;
};

void setLongitude(GPS& gps)
{
    gps.longitude = 42;
}
```

- Functions and classes can be declared "friends" using the friend keyword
- A friend function or class can have access to a class's private and protected members

## What is class inheritance?

- Capability of a class to inherit (or extend) the members (data and methods) of another class
- Reuse of functionalities and characteristics of a base class by a derived class
- Multiple classes can derive from the same base class
- One class may derive from multiple base classes (unlike Java)

- Derived classes inherit all the accessible members of their base classes: public and protected members
- Derived classes can extend the inherited members by adding their own members
- Base class cannot access extended members defined within inherited classes

## Class inheritance: example

```
class Aircraft
{
public:
    Aircraft() {cout << "Aircraft ctor" << endl;}
    ~Aircraft(){cout << "Aircraft ~dtor" << endl;}

    void setCapacity(int i) {capacity = i;}
    void fly() {cout << "Aircraft flying: " << capacity << endl;}
    // ...
protected:
    int capacity; //nbre of pass.
};

class Boeing: public Aircraft
{
public:
    Boeing() {cout << "Boeing ctor" << endl;}
    ~Boeing(){cout << "Boeing ~dtor" << endl;}
};

// main function
int main()
{
    Aircraft a;
    a.setCapacity(50);
    a.fly();

    Boeing b;
    b.setCapacity(100);
    b.fly();
}
```

```
Aircraft ctor
Aircraft flying: 50
Aircraft ctor
Boeing ctor
Aircraft flying: 100
Boeing ~dtor
Aircraft ~dtor
Aircraft ~dtor
```

## Construction / Destruction call order

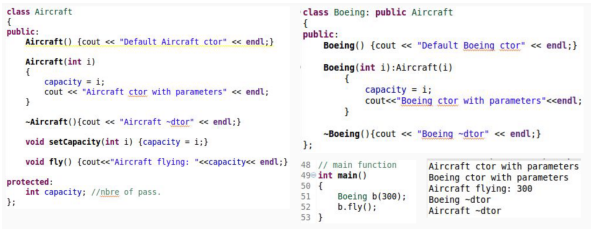
- Construction
  - Base class constructor is called first then the constructor of the derived class
  - Whenever any constructor of a derived class (either default or with parameters) is called, the default constructor of the base class is called automatically and executed first
- Destruction
  - It works in exactly the opposite order of construction
  - Derived class destructor is called first then the destructor of the base class

## Construction / Destruction order: example 1

```
class Aircraft
{
public:
    Aircraft() {cout << "Default Aircraft ctor" << endl;}

    Aircraft(int i)
    {
        capacity = i;
        cout << "Aircraft ctor with parameters" << endl;
    }

    ~Aircraft(){cout << "Aircraft ~dtor" << endl;}

    void setCapacity(int i) {capacity = i;}

    void fly() {cout<<"Aircraft flying: "<<capacity<< endl;}

protected:
    int capacity; //nbre of pass.
};

class Boeing: public Aircraft
{
public:
    Boeing() {cout << "Default Boeing ctor" << endl;}

    Boeing(int i)
        {
            capacity = i;
            cout<<"Boeing ctor with parameters"<<endl;
        }

    ~Boeing(){cout << "Boeing ~dtor" << endl;}
};

// main function
int main()
{
    Boeing b(300);
    b.fly();
}
```

```
Default Aircraft ctor
Boeing ctor with parameters
Aircraft flying: 300
Boeing ~dtor
Aircraft ~dtor
```

## Construction / Destruction order: example 2



```
class Aircraft
{
public:
    Aircraft() {cout << "Default Aircraft ctor" << endl;}

    Aircraft(int i)
    {
        capacity = i;
        cout << "Aircraft ctor with parameters" << endl;
    }

    ~Aircraft(){cout << "Aircraft ~dtor" << endl;}

    void setCapacity(int i) {capacity = i;}

    void fly() {cout<<"Aircraft flying: "<<capacity<< endl;}

protected:
    int capacity; //nbre of pass.
};

class Boeing: public Aircraft
{
public:
    Boeing() {cout << "Default Boeing ctor" << endl;}

    Boeing(int i):Aircraft(i)
        {
            capacity = i;
            cout<<"Boeing ctor with parameters"<<endl;
        }

    ~Boeing(){cout << "Boeing ~dtor" << endl;}
};

// main function
int main()
{
    Boeing b(300);
    b.fly();
}
```

```
Aircraft ctor with parameters
Boeing ctor with parameters
Aircraft flying: 300
Boeing ~dtor
Aircraft ~dtor
```

## Types of inheritance

- Derived classes can inherit a base class in three different fashions
  - Public
    - Derived class keeps the same access rights to the inherited members
    - Public members in base class remain public in derived class
    - Protected members in base class remain protected in derived class
  - Private
    - Derived class changes the accessibility rights to the inherited members
    - Public and protected members in base class become private in derived class
  - Protected
    - Derived class changes the accessibility rights to the inherited members
    - Public and protected members in base class become protected in derived class

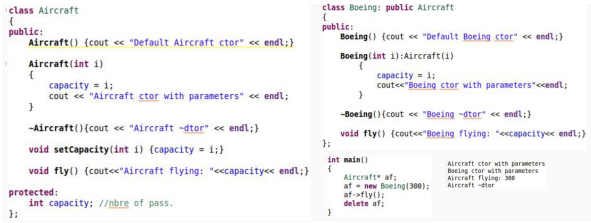
## Architecture dilemma: is-a VS has-a

- When designing the classes of a software you should define carefully the relationship between those classes
  - Should class A inherit from class B or should it contain a pointer to class B?
  - Should class Aircraft inherit from class Engine since every aircraft has an engine?
- If $A$ is $B$ then $A$ should inherit from $B$
- If $A$ has $B$ as one of its components then $A$ should contain $B$ and not inherit from it

## Few words about multiple inheritance

- C++ allows a class to inherit from multiple other classes
  - class FighterJet : public Aircraft, public Fighter
- Order of construction follows the same order of declaration
  - Aircraft ctor then Fighter ctor, then FighterJet ctor
- Beware the diamond problem
  - Use virtual inheritance to avoid the headache

## Polymorphism: having different forms

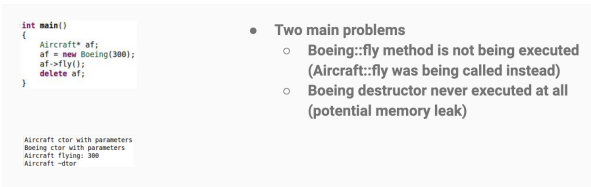


```
class Aircraft
{
public:
    Aircraft() {cout << "Default Aircraft ctor" << endl;}

    Aircraft(int i)
    {
        capacity = i;
        cout << "Aircraft ctor with parameters" << endl;
    }

    ~Aircraft(){cout << "Aircraft ~dtor" << endl;}

    void setCapacity(int i) {capacity = i;}

    void fly() {cout<<"Aircraft flying: "<<capacity<< endl;}

protected:
    int capacity; //nbre of pass.
};
```

```
class Boeing: public Aircraft
{
public:
    Boeing() {cout << "Default Boeing ctor" << endl;}

    Boeing(int i):Aircraft(i)
        {
            capacity = i;
            cout<<"Boeing ctor with parameters"<<endl;
        }

    ~Boeing(){cout << "Boeing ~dtor" << endl;}

    void fly() {cout<<"Boeing flying: "<<capacity<< endl;}
};
int main()
{
    Aircraft* af;
    af = new Boeing(300);
    af->fly();
    delete af;
}
```

```
Aircraft ctor with parameters
Boeing ctor with parameters
Aircraft flying: 300
Aircraft ~dtor
```

## Polymorphism

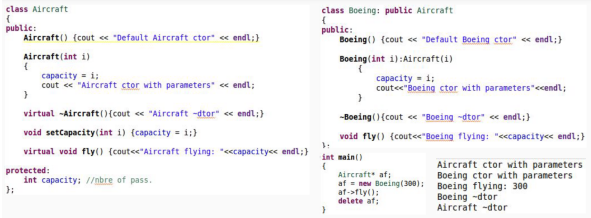


```
int main()
{
    Aircraft* af;
    af = new Boeing(300);
    af->fly();
    delete af;
}
```

```
Aircraft ctor with parameters
Boeing ctor with parameters
Aircraft flying: 300
Aircraft ~dtor
```

- Two main problems
  - Boeing::fly method is not being executed (Aircraft::fly was being called instead)
  - Boeing destructor never executed at all (potential memory leak)

## Polymorphism: virtual methods



```
class Aircraft
{
public:
    Aircraft() {cout << "Default Aircraft ctor" << endl;}

    Aircraft(int i)
    {
        capacity = i;
        cout << "Aircraft ctor with parameters" << endl;
    }

    virtual ~Aircraft(){cout << "Aircraft ~dtor" << endl;}

    void setCapacity(int i) {capacity = i;}

    virtual void fly() {cout<<"Aircraft flying: "<<capacity<< endl;}

protected:
    int capacity; //nbre of pass.
};
```

```
class Boeing: public Aircraft
{
public:
    Boeing() {cout << "Default Boeing ctor" << endl;}

    Boeing(int i):Aircraft(i)
        {
            capacity = i;
            cout<<"Boeing ctor with parameters"<<endl;
        }

    ~Boeing(){cout << "Boeing ~dtor" << endl;}

    void fly() {cout<<"Boeing flying: "<<capacity<< endl;}
};
int main()
{
    Aircraft* af;
    af = new Boeing(300);
    af->fly();
    delete af;
}
```

```
Aircraft ctor with parameters
Boeing ctor with parameters
Boeing flying: 300
Boeing ~dtor
Aircraft ~dtor
```

## Polymorphism: virtual keyword

- Always mark destructor virtual if the class is meant to be inherited
- You only need to mark the destructor of the base class virtual. By doing so, the compiler will automatically consider all subclasses' destructors as virtual as well.
- You only need to mark the polymorphic methods in the base class as virtual. However, it is common to mark them virtual in the derived classes as well for readability.
- C++11 introduced the keyword "override" to enhance the readability of the polymorphic methods

## Virtual methods VS pure virtual methods

- Virtual method has an implementation in the base class and can be overridden by a derived class to obtain polymorphic behavior
- Pure virtual method does not have an implementation in the base class and should necessarily be implemented in the derived classes
  - virtual void fly ()= 0;
- Class that does have at least one pure virtual method is called an abstract base class (similar to Java's interface classes)
- Abstract base classes cannot be instantiated. Only derived classes can