OOP_Inheritance

November 2, 2018

In [1]:
```
#include <iostream>

using namespace std;
```

1 Object Oriented Programming (OOP): Inheritance

FE 522 C++ Programming in Finance

Thiago W. Alves

1.1 Copy Constructors

- A copy constructor is a constructor for which the only parameter is an object of the same class.
- A copy constructor is used to create a new object from an already existing object.
- The only parameter of a copy constructor is a constant reference to another object of the same class.

In [2]:
```
enum OptionType {
    CALL,
    PUT
};
```

In [3]:
```
class EuropeanOption {
public:
    // default constructor (with no parameters)
    EuropeanOption();

    // constructor with parameters
    EuropeanOption(OptionType type, double s, double k, double r, double t, double vol

    // copy constructor
    EuropeanOption(const EuropeanOption& option);

private:
    OptionType m_optionType; // option type
```

```
        double m_s;             // spot price
        double m_k;             // strike price
        double m_r;             // interest rate
        double m_t;             // time to maturity
        double m_vol;           // volatility
        double m_price;
};
```

In [4]:
```
// default constructor (with no parameters)
EuropeanOption::EuropeanOption() {
    m_optionType = CALL;
    m_s = 100.0;
    m_k = 100.0;
    m_r = 0.12;
    m_t = 3.0;
    m_vol = 0.20;
    m_price = -1.0;      // invalid value
}
```

In [5]:
```
// copy constructor
EuropeanOption::EuropeanOption(const EuropeanOption& option) {
    m_optionType = option.m_optionType;
    m_s = option.m_s;
    m_k = option.m_k;
    m_r = option.m_r;
    m_t = option.m_t;
    m_vol = option.m_vol;
    m_price = -1.0;      // invalid value
}
```

- What would happen if we had used *call-by-value* instead of *call-by-reference* in the copy constructor?
- *call-by-value* creates an entirely new object as a copy of the original object, i.e. it calls the copy constructor of such object:
  - In a copy constructor, this would create infinite recursive calls.
  - Thankfully, trying to pass an object of the same class as call-by-value in a constructor raises a compilation error.
- Using a *const call-by-reference* is not a requirement, but is recommended:
  - Since the objective of a copy constructor is to initialize a new object from an existing one, there is no point in modifying the original object.
- If a copy constructor is not explicitly declared, the compiler will provide a *default* copy constructor.

In [6]:
```
class Foo {
public:
    int f;
};
```

```
In [7]: int sumFoos(Foo a, Foo b)
        {
            return a.f + b.f;
        }
```

- A *default* copy constructor copies the values of all data members of the original object to the new object that is being created.
- **This form of assignment works well for objects that do not allocate memory dinamically (with pointers).**

1.2 Copy Assignment

- An object can be assigned to another using the assignment (=) operator:

```
In [8]: EuropeanOption option1;
        EuropeanOption option2;
        option2 = option1;
```

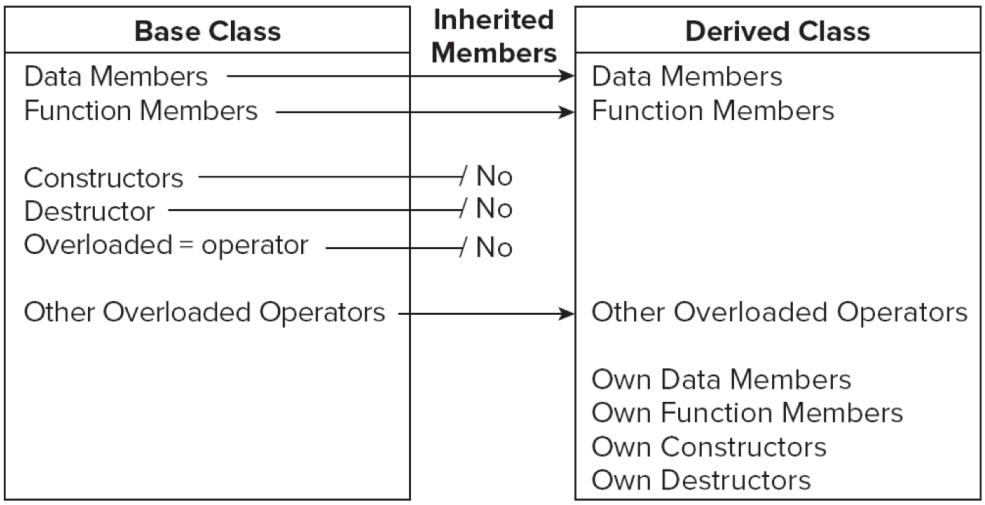
- In this case, the values of all data members are copied from one object to the other (one by one).
- **This form of assignment works well for objects that do not allocate memory dinamically (with pointers).**
- We will study the overload of copy assignments when we study pointers.
- However, when we use the following:

```
In [9]: EuropeanOption option3;
        EuropeanOption option4 = option3;
```

- We are not using a copy assignment, but constructing an entirely new object from another. That is, we are calling the copy constructor.

1.3 Inheritance

- The second of the three main mechanisms of OOP.
- Inheritance is the process of creating new classes (*derived classes*) based on existing classes (*base classes*).
- The derived class inherits all the characteristics of the base class, in addition to its own characteristics.
- One of the major benefits of the inheritance mechanism is code reutilization. After defining a class, it is not necessary to change it in order to adapt it to new use cases.
- Code reutilization also allows advanced use of class libraries created by others. Without modifying the original classes, it is possible to derive other classes from them according to the problem being faced.



Inheritance.png

- Particular to C++ is the fact that a derived class may inherit from more than one base class. **Multiple inheritance is, however, not recommended, and will not be covered in this course.**
- A derived class automatically contains all data members of its base class.
- Methods are also inherited, except:
  - Constructors (including copy and move constructors)
  - Copy (and move) assignment
  - Destructor
- The derived class can, however, substitute the derived methods with its own version of them.

1.4 Defining a Derived Class

- To derive a class from an existing one, put the name of the base class after the name of the new class, separated by a colon (:).
- The name of the base class may also be preceded by the keyword *public*, *protected* or *private*: we will study their effects in next slides.
- Box.h:

```
In [10]: #pragma once

         class Box {
         public:
             double m_length;
             double m_width;
             double m_height;
         };
```

- CandyBox.h:

```
In [11]: #pragma once
         // #include "Box.h"

         class CandyBox : Box {
         public:
             string m_contents;
         };
```

- A derived class automatically contains all data members of its base class:

```
In [12]: CandyBox cb;

In [13]: cb.m_length = 3;

input_line_20:2:2: error: cannot cast 'CandyBox' to its private base class 'Box'
 cb.m_length = 3;
 ^
input_line_18:3:18: note: implicitly declared private here
class CandyBox : Box {
                 ^~~
input_line_20:2:5: error: 'm_length' is a private member of 'Box'
 cb.m_length = 3;
    ^
input_line_18:3:18: note: constrained by implicitly private inheritance here
class CandyBox : Box {
                 ^~~
input_line_17:4:12: note: member is declared here
    double m_length;
           ^



        Interpreter Error:
```

- In the previous example, the compiler gives an error when we try to access the (*public*) data member *m_length* from an object of the derived class CandyBox.
- This happens because *m_length* became *private* in the derived class, since there is an implicit default private access modifier for the base class in the definition of the derived class.
- The access modifier for the base class is always present. It determines the status of the inherited members in the derived class.
- If the access modifier is not explictly indicated, the compiler implictly uses *private* as default.
- That is, the following two class declarations are equivalent:

```
class CandyBox : Box
class CandyBox : private Box
```

- Private members of the base class are also members of the derived class, but cannot be accessed by any method of the derived class.

- Box.h:

```
In [14]: #pragma once

         class Box2 {
         private:
             double m_length;
             double m_width;
             double m_height;
         };
```

- CandyBox.h:

```
In [15]: #pragma once
         // #include "Box.h"

         class CandyBox2 : Box2 {
         public:
             string m_contents;

             double Volume() const;
         };
```

- Private members of the base class are also members of the derived class, but cannot be accessed by any method of the derived class.

```
In [16]: // Function to calculate the volume
         // of a CandyBox object
         // Error - members not accessible
         double CandyBox2::Volume() const {
             return m_length * m_width * m_height;
         }
```

```
input_line_23:5:12: error: 'm_length' is a private member of 'Box2'
    return m_length * m_width * m_height;
           ^
input_line_21:4:12: note: declared private here
    double m_length;
           ^
input_line_23:5:23: error: 'm_width' is a private member of 'Box2'
    return m_length * m_width * m_height;
                      ^
input_line_21:5:12: note: declared private here
```

```
    double m_width;
           ^
input_line_23:5:33: error: 'm_height' is a private member of 'Box2'
    return m_length * m_width * m_height;
                                ^
input_line_21:6:12: note: declared private here
    double m_height;
           ^



        Interpreter Error:
```

- Other than *public* and *private* access modifiers, it is also possible to declare members as *protected*.
- In the class where the members are declared, *protected* has exactly the same effect as *private*: *protected* members may only be accessed by methods of the class (and not directly from its objects).
- *protected* ensures that members are effectively private, but makes them accessible to methods of derived classes.
- Box.h:

```
In [17]: #pragma once

         class Box3 {
         protected:
             // protected members: as private members, they may
             // not be accessed directly from objects of the class,
             // but are visible to the methods of derived classes
             double m_length;
             double m_width;
             double m_height;
         };
```

- CandyBox.h:

```
In [18]: #pragma once
         // #include "Box.h"

         class CandyBox3 : Box3 {
         public:
             string m_contents;

             double Volume() const;
         };
```

```
In [19]: // Derived class function to calculate volume
         double CandyBox3::Volume() const {
             return m_length * m_width * m_height;
         }
```

1.5 Constructors in Derived Classes

- Even though constructors are not inherited, they still exist and are used to create the part of the object that is from the base class:
  - After all, *private* members of the base class are not accessible to the methods of the derived class, therefore the responsability of initializing them is of the base class' constructor.
- The *default* constructor of the base class is automatically called by the *default* constructor of the derived class.
- It is possible to call a particular base class' constructor from within a constructor of the derived class:

```
In [20]: class Box4 {
         public:
             double m_length;
             double m_width;
             double m_height;

             // Base class constructor
             Box4(double lv = 1.0, double wv = 1.0, double hv = 1.0) {
                 m_length = lv;
                 m_width = wv;
                 m_height = hv;
                 cout << "Box constructor called" << '\n';
             }
         };
```

```
In [21]: class CandyBox4 : public Box4 {
         public:
             string m_contents;

             CandyBox4(double lv, double wv, double hv, string str = "Candy");
             CandyBox4(string str = "Candy");

             CandyBox4(const CandyBox4& initCB);
         };
```

```
In [22]: // Constructor to set dimensions and contents
         // with explicit call of Box constructor
         CandyBox4::CandyBox4(double lv, double wv, double hv, string str) : Box4(lv, wv, hv)
             cout << "CandyBox constructor2 called" << '\n';
             m_contents = std::move(str);
         }
```

```
In [23]: // Constructor to set contents
         // calls default Box constructor automatically
         CandyBox4::CandyBox4(string str) {
             cout << "CandyBox constructor1 called" << '\n';
             m_contents = std::move(str);
         }
```

为什么一定要加std::
用处是处理str

```
In [24]: CandyBox4 cb4a;
```

为什么没有参数

```
Box constructor called
CandyBox constructor1 called
```

```
In [25]: CandyBox4 cb4b(3.0, 3.0, 3.0, "Treat");
```

```
Box constructor called
CandyBox constructor2 called
```

```
In [26]: cout << cb4a.m_length << '\t' << cb4a.m_width << '\t' << cb4a.m_height << '\t' << cb4a
         cout << cb4b.m_length << '\t' << cb4b.m_width << '\t' << cb4b.m_height << '\t' << cb4b
```

```
1        1        1        Candy
3        3        3        Treat
```

1.6 Copy Constructors in Derived Classes

- The copy constructor is called when we declare an object that is initialized from another object of the same class.
- If no copy constructor is explictly defined, the compilar will generate one automatically, copying the members of the original object one by one to the new object.
- The copy constructor of a derived class **DOES NOT** automatically call the copy constructor of a base class. It calls the default constructor instead.

```
In [27]: CandyBox4::CandyBox4(const CandyBox4& initCB) {
             cout << "CandyBox copy constructor called" << '\n';
             m_contents = initCB.m_contents;
         }
```

```
In [28]: CandyBox4 cb4c = cb4b;
         cout << cb4c.m_length << '\t' << cb4c.m_width << '\t' << cb4c.m_height << '\t' << cb4c
```

```
Box constructor called
CandyBox copy constructor called
1        1        1        Treat
```

virtual的位置：
加在baseclass里
A virtual.(B& b) C

```
In [29]: class CandyBox5 : public Box4 {
         public:
             string m_contents;

             CandyBox5(double lv, double wv, double hv, string str = "Candy");

             CandyBox5(const CandyBox5& initCB);
         };

In [30]: CandyBox5::CandyBox5(double lv, double wv, double hv, string str) : Box4(lv, wv, hv) {
             cout << "CandyBox constructor2 called" << '\n';
             m_contents = std::move(str);
         }

In [31]: CandyBox5::CandyBox5(const CandyBox5& initCB) : Box4(initCB) {
             cout << "CandyBox copy constructor called" << '\n';
             m_contents = initCB.m_contents;
         }

In [32]: CandyBox5 cb5b(3.0, 3.0, 3.0, "Treat");

Box constructor called
CandyBox constructor2 called


In [33]: CandyBox5 cb5c = cb5b;
         cout << cb5c.m_length << '\t' << cb5c.m_width << '\t' << cb5c.m_height << '\t' << cb5

CandyBox copy constructor called
3        3        3        Treat
```