Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

September 7, 2018

Inheritance: Part I

- In this lecture we shall learn about the important concept of **inheritance**.
- Inheritance is an important way that a C++ class makes use of data and methods which have already been written in another class.
- It is not necessary to start from scratch and write all the data and/or functionality ourselves.
- Instead we can make use of the already available data and/or functionality in another class and add or modify it to suit our needs.
- We shall study the inheritance tree, which is a useful aid to visualize inheritance.

1 Introduction

- As with much else in object oriented programming (OOP), we observe the fundamental concepts all around us in everyday life.
- We can explain, or at least motivate, the concept of **inheritance** without reference to computer programming at all.
- Let us employ the example a rectangle and a square.
  1. Every square is a rectangle.
  2. Every square has, or **inherits**, all the properties of a rectangle.
  3. However, a square also has specialized properties which are not shared by all rectangles in general.
  4. A square is therefore a specialization of the "parent" or "base case" of a rectangle.
- We can go one step further.
  1. We can consider a "polygon" as the base case.
  2. Every triangle is a polygon.
  3. Every rectangle is also a polygon.
  4. Every equilateral triangle is a triangle.
  5. Every square is a rectangle.
  6. However, a rectangle is not a triangle (and vice-versa).
- Hence the concept of inheritance forms a tree.
- Each shape (in the above example) inherits from its parent, and the parent of its parent, etc.
- The inheritance tree can go many levels deep, there is no upper limit.
- Each branch can have sub-branches.
- Some branches may have many sub-branches, others not so many or none at all.
- The concept of **inheritance** in OOP attempts to capture these relationships.
- In this course, we shall learn how inheritance is implemented in C++.
- However, it is important to bear in mind that **inheritance is a computer science concept**, and is *independent of any specific programming language.*
- Inheritance is a major feature of Java, for example.

2 Example: class `A`

- Consider the following very simple class `A`.

```
class A {
public:
  string name() { return "class is A"; }
  void f1()  { cout << "A::f1" << endl; }
  void f2()  { cout << "A::f2" << endl; }
  void f3()  { cout << "A::f3" << endl; }
};
```

- The class `A` just contains a few public methods and no data.
- It is obvious what each method of the class `A` does.
- We shall use the class `A` as our "base class" to demonstrate the concept of **inheritance**.
- Here is a working C++ program to use the class `A`.

```
#include <iostream>
#include <string>
using namespace std;

class A {
  // etc
};

int main()
{
  A a;
  cout << a.name() << endl;           // prints "class is A"
  a.f1();                             // prints "A::f1"
  a.f2();                             // prints "A::f2"
  a.f3();                             // prints "A::f3"
  return 0;
}
```

3 Inheritance: introduction

- We now write a second class `B` as follows, which **inherits** from the class `A`.

```
class B : public A {
public :
    string name() { return "class is B"; }
    void f1() { cout << "B :: f1" << endl; }
};
```

- This syntax "**: public A**" signifies that the class `B` **inherits** from the class `A`.
  1. We also say that `B` is a **derived class** of `A` (or `B` **derives** from `A`).
  2. We also say that `A` is the **base class** of `B` (or `A` is the **parent** of `B`).
- What this means, in practice, is the following:
  1. **The class `B` has access to all the public methods of `A` as if they were methods of the class `B` itself.** (Just as a square has all the properties of a rectangle.)
  2. Hence an object of type `B` can call all the methods `name()`, `f1()`, `f2()` and `f3()`.
  3. *However, the class* `B` *defines its own versions of the methods* **name()** *and* **f1()**.
  4. We say that the methods `name()` and `f1()` in `B` **override** the methods in `A`.
  5. However, the class `B` does not override the methods `f2()` and `f3()`.
- It is simplest to explain by running a program and printing the output.

```
#include <iostream>
#include <string>
using namespace std;

class A {
  // etc
};
class B : public A {
  // etc
};
int main()
{
  B b;
  cout << b.name() << endl;   // prints "class is B" *** overridden,     uses class B
  b.f1();                     // prints "B::f1"      *** overridden,     uses class B
  b.f2();                     // prints "A::f2"      *** not overridden, uses class A
  b.f3();                     // prints "A::f3"      *** not overridden, uses class A
  return 0;
}
```

- Let us work our way through the program.
- **b.name()**
  1. The compiler searches for a method "`name()`" in the declaration of the class `B`.
  2. The compiler finds the method "`name()`" in the class `B` and invokes it.
  3. Therefore "class is B" is printed to screen.
- **b.f1()**
  1. The compiler searches for a method "`f1()`" in the declaration of the class `B`.
  2. The compiler finds the method `f1()` in the class `B` and invokes it.
  3. Therefore "B::f1" is printed to screen.
- **b.f2()**
  1. The compiler searches for a method "`f2()`" in the declaration of the class `B`.
  2. The compiler **does not find** the method `f2()` in the class `B`.
  3. The compiler does not give up. The compiler knows that `B` inherits from `A`, hence the compiler searches for the method `f2()` in the declaration of the class `A`.
  4. The compiler finds the method `f2()` **in the class `A`** and invokes it.
  5. **Therefore "A::f2" is printed to screen, as if `b` were an object of the class `A`.**
- **b.f3()**
  1. The compiler searches for a method "`f3()`" in the declaration of the class `B`.
  2. The compiler does not find the method `f3()` in the class `B`, but finds the method `f3()` in the base class `A`.
  3. **Therefore "A::f3" is printed to screen, as if `b` were an object of the class `A`.**

4 Inheritance: “`IS-A`”

- Many textbooks use the “**IS-A**” buzzword to express the concept of inheritance.
- What exactly does the class `B` consist of?
- An object of type `B` implicitly contains an object of the base class `A`.
- The structure of an object of the class `B` is schematically as follows:

$$\text{object of class B} = \begin{array}{|c|}\hline \text{B component} \\ \hline \text{A component} \\ \hline \end{array}$$

- When the compiler allocates memory for an object of the class `B`, it first allocates memory for an object of the class `A`, then adds on whatever extra memory is required to complete the construction of the class `B`.
  1. This is not just symbolic.
  2. We shall see later that the compiler invokes the constructor of the class `A` first, then it invokes the constructor of the class `B`.
  3. We have not written any constructors for the classes `A` and `B`, hence the compiler generates default constructors for both `A` and `B` and invokes them in the order `A` then `B`.
  4. The compiler really does construct an object of the base class first, then adds on the construction of the derived class.
  5. There really is a base class object implicitly inside a derived class object.
- Hence a `B` object “IS-A” `A` object.
  1. For example every square “IS-A” rectangle.
  2. Every equilateral triangle “IS-A” triangle.
  3. Also, every square “IS-A” polygon.
  4. Every equilateral triangle “IS-A” polygon.

5 Digression: “`HAS-A`”

- This section is not about inheritance.
- It is about a related buzzword “**HAS-A**” which also appears in many textbooks.
- The concept is simple and we have already encountered it.
- Let us declare three classes `Cat`, `Dog` and `Owner`.
- The internal details of the classes `Cat` and `Dog` are not important.
- The class `Owner` has two data members, of type `Cat` and `Dog`.

```
class Cat {
  // etc
};

class Dog {
  // etc
};

class Owner {
public:
  // etc
private:
  Cat c;
  Dog d;
};
```

- Then we say `Owner` “**HAS-A**” `Cat` and `Owner` “**HAS-A**” `Dog`.
- Hence “**HAS-A**” indicates that a class has a data member of another class.
- This is different from inheritance.
- Basically, there are different ways in which classes can have relationships with each other.

6 Inheritance tree

- Let us make an inheritance tree.

  1. Class `A` is the base.
  2. `B` derives from `A`.
  3. `C` and `D` both derive from `B`.
  4. `E` inherits directly from the base `A`.

```
class A;
class B : public A;
class C : public B;
class D : public B;
class E : public A;
```

- The structure of the inheritance tree looks like this.

C	D	
B		E
A		

- `C` implicitly contains an object of class `B` which implicitly contains an object of class `A`.
- `D` implicitly contains an object of class `B` which implicitly contains an object of class `A`.
- `E` implicitly contains an object of class `A` only.
- `C` and `D` both know about `B` and therefore about `A`.
- `E` knows only about `A` but not about `B`, `C` and `D` (and vice-versa).
- **Each class in the inheritance tree is a base class for all the others above it.**

  1. Hence `A` is a base class for all the others.
  2. `B` is the base class for `C` and `D` (and `B` has its own base class, i.e. `A`).
  3. If an additional class, say `F`, is written and derives from `C`, then `C` is the base class for `F`.
  4. The **inheritance chain** will be F $\to$ C $\to$ B $\to$ A.

- **If a derived class does not override a method, the compiler searches downwards through the inheritance tree until it finds a match.**

  1. Suppose the compiler starts from `C` or `D`.
  2. If the compiler finds an override in `C` or `D`, it uses that override.
  3. If not, then the compiler searches for a match in `B` and next in `A`.
  4. **However the compiler will not search in `E`.**
  5. The compiler figures out the structure of the inheritance tree automatically.

7 Inheritance tree: overrides of methods

- **`C` derives from `B` and overrides `name` and `f3`.**

```
class C : public B {
public:
  string name() { return "class is C"; }
  void f3()  { cout << "C::f3" << endl; }
};
```

1. `f1` drops down to **B::f1** because `B` overrides `f1`. $\mathtt{C::f1} \longrightarrow \mathtt{B::f1}$
2. `f2` drops down to **A::f2**. $\mathtt{C::f2} \longrightarrow \mathtt{A::f2}$
3. `f3` does not drop down (overridden by `C`).

- **`D` derives from `B` and overrides `name` only.**

```
class D : public B {
public:
  string name() { return "class is D"; }
};
```

1. `f1` drops down to **B::f1** because `B` overrides `f1`. $\mathtt{D::f1} \longrightarrow \mathtt{B::f1}$
2. `f2` drops down to **A::f2**. $\mathtt{D::f2} \longrightarrow \mathtt{A::f2}$
3. `f3` drops down to **A::f3**. $\mathtt{D::f3} \longrightarrow \mathtt{A::f3}$

- **`E` derives from `A` and overrides `name`, `f1` and `f3`.**

```
class E : public A {
public:
  string name() { return "class is E"; }
  void f1()  { cout << "E::f1" << endl; }
  void f3()  { cout << "E::f3" << endl; }
};
```

1. `f1` does not drop down (overridden by `E`).
2. `f2` drops down to **A::f2**. $\mathtt{E::f2} \longrightarrow \mathtt{A::f2}$
3. `f3` does not drop down (overridden by `E`).

- The total list of the overridden methods looks like this.

class	name	f1	f2	f3
A	A	A	A	A
B	B	B	A	A
C	C	B	A	C
D	D	B	A	A
E	E	E	A	E

8 Keyword `protected`

- Up to now, our example classes have only public methods and no data.
- How does a derived class access data members in a parent class?
- *What about private data? Private methods?*
- To answer these questions, we shall employ the example of the rectangle and square.
- We declare a class `Rectangle` with private data members `h,w` for the height and width.
- We also declare a class `Square` which inherits from `Rectangle`.

```
class Rectangle {
public:
  // etc
private:
  double h, w;
};

class Square : public Rectangle {
  // etc
};
```

- **We encounter an immediate difficulty with the use of the "`private`" keyword.**
- **A derived class cannot access private data and methods.**
  1. It does not matter that a `Square` "IS-A" `Rectangle`.
  2. It does not matter that an object of type `Square` implicitly contains an object of type `Rectangle`.
  3. The keyword `private` is an absolute lock-out: ***nothing outside of a class can access its private data and methods.***
  4. The only way to access the private data of a class is via (public) accessor and mutator methods supplied in the class declaration.
  5. By the same logic, private methods are not accessible to derived classes.
  6. This situation is terribly inconvenient for an inherited class.
  7. By definition, an inherited class is related to the base class and would like to use its data and methods.
- The keyword "**protected**" is designed to remedy this situation,
- **See next page(s).**

9 Properties of keyword `protected`

- The keyword **protected** has the following properties.
  1. Derived classes have access to all the protected data and methods.
  2. Outside applications cannot access protected data and methods.
  3. Effectively, "`protected`" means:
     (a) "private" to the outside world.
     (b) "public" to derived classes.
- **Protected data and protected methods in a class are visible to all the derived classes above it in the inheritance tree.**
- In terms of an inheritance chain, a class can access all the protected data and protected methods down the inheritance chain.
- This feature makes the `protected` keyword very useful.
- Personally, I find the keyword `protected` to be much more useful than `private`.
- Let us redefine the class `Rectangle` and tag the data as `protected`.
- The class `Square` can now access the protected data and methods of the class `Rectangle`.

```
class Rectangle {
public:
    // etc
protected:                              // keyword "protected"
    double h, w;
};

class Square :  public Rectangle {
    // etc
};
```

- We shall use the `Rectangle` and `Square` class later, when we study how to write constructors, etc. for derived classes.

10 Timing

- **The material in this section is a sociological comment.**
- An important fact to bear in mind is that the base class and derived class **do not have to be written at the same time.**
- It is perfectly possible for a software library to exist for years and for a derived class to be added later.
- There are many examples of software libraries which are sold commercially, e.g. for mathematical or financial calculations.
- Customers purchase a software library and **write their own derived classes.**
- The designers of the original library have no idea what the derived classes will look like.
- Some software libraries have existed for years and the programmers who write derived classes today may not even have been born when the base classes in the library were written.
- This is an example of the enormous utility of object oriented programming.