C++ Programming

Taking Classes Further

Taking Classes Further

- Inheritance
- Multiple Inheritance
- Friends

Inheritance

- One of the most important concepts in object-oriented programming is that of inheritance
- Inheritance allows us to define a class in terms of another class, which makes it easier to create, organize, and maintain an application
- This also provides an opportunity to reuse code functionality and accelerate implementation time (less time to code, less time to test, and so on)

Inheritance

- When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class
- This existing class is called the base class or superclass, and the new class is referred to as the derived class or a subclass of the base class

Inheritance

- A class can be derived from more than one class, which means it can inherit data and functions from multiple base classes (multiple inheritance)
- To define a derived class, we use a class derivation list to specify the base class(es); a class derivation list has the form:

```
class derived-class: access-specifier base-class
```

- In this case, `access-specifier` is one of `public`, `protected`, or `private`, and `base-class` is the name of a previously defined class
- If the access specifier is not used, then it is `private` by default

Inheritance

- If instead of

```
class B { ... };
```

  we write:

```
class B: public A { ... };
```

  … it means: B is a subclass of A

- For example:

```
class Mammal: public Animal { ... };
```

  which means Mammal is a subclass of Animal

Inheritance – Example

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

// Base class
class Shape
{
protected:
   int sides;
public:
   int getSides()
        { return sides; }
   void setSides(int s)
        { sides = s; }
};
```

```
// Derived class
class Rectangle: public Shape
{
public:
    Rectangle()
    { sides = 4; }
};
```

```
int main()
{
   Rectangle rect;
   cout << "Sides: " << rect.getSides() << endl;
}
```

Inheritance and Access

	public member	protected member	private member
Access from members of own class	Yes	Yes	Yes
Access from members of a derived class	Yes	Yes	No
Access from non members (other code)	Yes	No	No

Inheritance and Access

- In public inheritance, all members are inherited with the same access levels as they had in the base class
- In protected inheritance, all public members in the base class are inherited as protected
- In private inheritance, all members in the base class are inherited as private
- Generally, most use cases for inheritance should follow public inheritance, even though in C++ private would be the default

Inheritance and Access – Example 1

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

// Base class
class Shape
{
protected:
   int sides;
public:
   int getSides()
        { return sides; }
   void setSides(int s)
        { sides = s; }
};
```

```
// Derived class
class Rectangle: public Shape
{
public:
    Rectangle()
    { sides = 4; }
};
```

```
int main()
{
   Rectangle rect;
   cout << "Sides: " << rect.getSides() << endl;
}
```

Inheritance and Access – Example 2

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

// Base class
class Shape
{
private:
   int sides;
public:
   int getSides()
        { return sides; }
   void setSides(int s)
        { sides = s; }
};
```

```
// Derived class
class Rectangle: public Shape
{
public:
    Rectangle()
    { setSides(4); }
};
```

Because `sides` is private, we can't reference it directly in the derived class. We need to use `setSides()` instead.

```
int main()
{
   Rectangle rect;
   cout << "Sides: " << rect.getSides() << endl;
}
```

Inheritance and Access – Example 3

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

// Base class
class Shape
{
protected:
    int sides;
public:
    int getSides()
         { return sides; }
    void setSides(int s)
         { sides = s; }
};
```

```
// Derived class
class Rectangle: private Shape
{
public:
     Rectangle()
     { sides = 4; }
};
```

Because we inherited privately, the `getSides()` method is no longer publicly visible.

```
int main()
{
   Rectangle rect;
   cout << "Sides: " << rect.getSides() << endl;
}
```

As a result, this call will fail.

Inheritance of Constructors and Destructors

- Like other members, a base class's constructors and destructor are also passed down to derived classes during inheritance
- They are automatically called by constructors and destructor of the derived class when its constructors or destructor are called
  - Base class constructors are called first, but their destructors are called last
- Unless otherwise specified, the default constructor (the one taking no parameters) is called

Inheritance of Constructors and Destructors

```
#include <iostream>
using namespace std;
class Parent {
public:
    Parent() {
      cout << "In Parent default constructor" << endl;
    }
    ~Parent() {
      cout << "In Parent destructor" << endl;
    }
};

class Child: public Parent {
public:
    Child() {
      cout << "In Child default constructor" << endl;
    }
    ~Child() {
      cout << "In Child destructor" << endl;
    }
};

int main() {
    Parent p;
    Child c;
}
```

When executed, we get this output. Note the order in which things are printed.

```
In Parent default constructor
In Parent default constructor
In Child default constructor
In Child destructor
In Parent destructor
In Parent destructor
```

call parent before the construction of child.

destruct child would also call the destructor of parent.

Inheritance of Constructors and Destructors

```
#include <iostream>
using namespace std;
class Parent {
public:
    Parent() {
      cout << "In Parent default constructor" << endl;
    }
    Parent(int x) {
      cout << "In Parent other constructor" << endl;
    }
};

class Child: public Parent {
public:
    Child() {
      cout << "In Child default constructor" << endl;
    }
    Child(int x) {
      cout << "In Child other constructor" << endl;
    }
};

int main() {
    Child c1;
    Child c2(0);
}
```

Note that because we didn't specify which base class constructor to use, the second case still used the default even though a better match was there.

```
In Parent default constructor
In Child default constructor
In Parent default constructor
In Child other constructor
```

Inheritance of Constructors and Destructors

```
#include <iostream>
using namespace std;
class Parent {
public:
   Parent() {
     cout << "In Parent default constructor" << endl;
   }
   Parent(int x) {
     cout << "In Parent other constructor" << endl;
   }
};

class Child: public Parent {
public:
   Child() : Parent() {
     cout << "In Child default constructor" << endl;
   }
   Child(int x) : Parent(x) {
     cout << "In Child other constructor" << endl;
   }
};

int main() {
   Child c1;
   Child c2(0);
}
```

to call the other constructor we have to manually specify the constructor use in each child constructor.

We specifically identified which constructor from the base class to call from the child class, so we now get the desired behaviour.

```
In Parent default constructor
In Child default constructor
In Parent other constructor
In Child other constructor
```

For the readability, it is better to write the constructor even if it is the default one.

Multiple Inheritance

- Until now we have seen inheritance from one base class to one or more derived classes
- It is possible in C++ for one class to inherit from more than one class; this is referred to as multiple inheritance
- This can be done by simply listing multiple classes, separated by commas, in the base class section of the class definition (after the :)

Multiple Inheritance

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

class House {
protected:
   int residents;
public:
   int getResidents() {
      return residents;
   }
   void setResidents(int num) {
   residents = num;  }
};

class Boat {
protected:
   float maxSpeed;
public:
   float getMaxSpeed() {
      return maxSpeed;
   }
   void setMaxSpeed(float max) {
      maxSpeed = max;
   }
};
```

A HouseBoat is both a House and a Boat, inheriting the members of both.

```
class HouseBoat: public House, public Boat {
};

int main() {
   HouseBoat hb;

   hb.setResidents(4);
   hb.setMaxSpeed(10.0);

   cout << "This dwelling has " << hb.getResidents();
   cout << " residents and a max speed of ";
   cout << hb.getMaxSpeed() << endl;
}
```

these two are accessible for: house; boat; and houseboat.

Friends of Classes

- Private and protected members of a class cannot be accessed from outside that class; however, this rule does not apply to "friends"
- Friends are functions or classes declared with the `friend` keyword
- A couple of things to keep in mind:
  - Friendship is not reciprocal; just because X is a friend of Y, that does not mean that Y is a friend of X, unless it is explicitly declared
  - Friendship is not transitive; a friend of a friend is not considered a friend, again unless explicitly declared

Friend Functions

- A non-member function can access the private and protected members of a class if it is declared a friend of that class
- That is done by including a declaration of this external function within the class, and preceding it with the keyword `friend`
  - Because of the way they are declared, such friend functions can appear as though they are members of the the class but they are not; they simply have access to private and protected members that they ordinarily wouldn't have access to

Friend Functions

the friend is external.

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

class Rectangle {
private:
    int width, height;
public:
    Rectangle() {}
    Rectangle (int x, int y) : width(x), height(y) {}
    int area() {return width * height;}
    friend Rectangle doubleInSize(const Rectangle&);
};

Rectangle doubleInSize(const Rectangle& param){
    Rectangle res;
    res.width = param.width*2;
    res.height = param.height*2;
    return res;}

int main () {
    Rectangle foo;
    Rectangle bar(2,3);
    foo = doubleInSize(bar);
    cout << foo.area() << endl;
}
```

bar.x are private .y variables.

since it is outside of the class we're not using Rec::doubleSize()

Note the interesting shorthand used to initialize the `width` and `height` data members. This still works!

Here we denote this function to be a friend of the class. We're not saying it's a member … it's just a friend.

The friend function has direct access to private data members but is itself not a member of the class.

Friend Classes

- Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class

Friend Classes

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

class Square;

class Rectangle {
private:
   int width, height;
public:
   int area () {return (width * height);}
   void convert (Square a);
};

class Square {
friend class Rectangle;
private:
   int side;
public:
   Square (int a) {
     side = a;
   }
};

void Rectangle::convert (Square a) {
   width = a.side;
   height = a.side;
}
```

```
int main () {
   Rectangle rect;
   Square sqr(4);

   rect.convert(sqr);
   cout << rect.area() << endl;
}
```

Note the forward declaration of the class `Square`. This is because of a circular dependency/chicken-and-the-egg situation here. The `Rectangle` class needs to know about `Square` to refer to it in its `convert()` method. But, `Square` needs to know about `Rectangle` to make it a friend. Since they both can't be declared first, we give the compiler an empty declaration of `Square` so it knows to expect it later on. We also have to defer the definition of `convert()` until `Square` has been defined, as this method needs `Square`'s details.

rec) class.