Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

September 7, 2018

Friendship

- In this lecture we shall learn about **friend functions.**
- We can also declare **friend classes.**

Recapitulation of C++ class for use in coding examples

- We shall employ the class Point1 in the coding examples in this document.

```
class Point1 {
public:                                          // keyword public
  // public methods
  void set(const double &a, const double &b)
  {
    x = a;
    y = b;
  }

  const double& getx() const { return x; }
  const double& gety() const { return y; }

  void print() const
  {
    cout << "print x,y    " << x << "    " << y << endl;
  }

private:                                         // keyword private
  // data
  double x, y;
};
```

Function overloading

Recall the code for functions `P1plus`, `P1minus` and two versions of `P1times`.

```
Point1 P1plus(const Point1 &u, const Point1 &v)  // function
{
  double xsum = u.getx() + v.getx();             // accessor methods
  double ysum = u.gety() + v.gety();
  Point1 p;
  p.set(xsum, ysum);
  return p;
}

Point1 P1minus(const Point1 &u, const Point1 &v) // operator
{
  double xdiff = u.getx() - v.getx();            // accessor methods
  double ydiff = u.gety() - v.gety();
  Point1 p;
  p.set(xdiff, ydiff);
  return p;
}

Point1 P1times(double c, const Point1 &u)        // function
{
  double cx = c * u.getx();
  double cy = c * u.gety();
  Point1 p;
  p.set(cx, cy);
  return p;
}

Point1 P1times(const Point1 &u, double c)        // function
{  return P1times(c,u); }                        // just call the other function
```

Overloaded operators

Recall the code for the overloaded operators +, − and two versions of ∗.

```
Point1 operator+ (const Point1 &u, const Point1 &v)  // operator
{
  double xsum = u.getx() + v.getx();                 // accessor methods
  double ysum = u.gety() + v.gety();
  Point1 p;
  p.set(xsum, ysum);
  return p;
}

Point1 operator- (const Point1 &u, const Point1 &v)  // operator
{
  double xdiff = u.getx() - v.getx();
  double ydiff = u.gety() - v.gety();
  Point1 p;
  p.set(xdiff, ydiff);
  return p;
}

Point1 operator* (double c, const Point1 &u)         // operator
{
  double x = u.getx() * c;
  double y = u.gety() * c;
  Point1 p;
  p.set(x, y);
  return p;
}

Point1 operator* (const Point1 &u, double c)         // operator
{ return (c*u); }
```

1 Friend functions and friend operators

- Let us review the code for `operator+`.

  ```
  Point1 operator+ (const Point1 &u, const Point1 &v)  // operator
  {
    double xsum = u.getx() + v.getx();                 // accessor methods
    double ysum = u.gety() + v.gety();
    Point1 p;
    p.set(xsum, ysum);
    return p;
  }
  ```

- The code is still a bit clumsy.

- It is likely that the + operator will be called many times for `Point1` objects.

- Since the accessor functions `getx()` and `gety()` simply return the values of x and y in the object, and moreover since `operator+` performs calculations that are not suspicious or risky (from the viewpoint of data security), it would be nice if the code for `operator+` did not have to use the accessor functions.

- C++ provides a mechanism known as **friendship.**

1.1 Revised class declaration

- We must edit the `Point1` class.
- We declare `operator+` as a **friend** of the class `Point1`.
- Let us also declare the function `P1plus` as a friend of the class `Point1`.
- The revised class declaration is given below.

```
class Point1 {
public:
  // public methods
  void set(const double &a, const double &b)
  {
    x = a;
    y = b;
  }

  const double& getx() const { return x; }
  const double& gety() const { return y; }

  void print() const
  {
    cout << "print x,y    " << x << "    " << y << endl;
  }

private:
  // data
  double x, y;

  friend Point1 P1plus(const Point1 &u, const Point1 &v);         // friend function
  friend Point1 operator+ (const Point1 &u, const Point1 &v);     // friend operator
};
```

1.2 Revised code for friend function and friend operator

- The revised code for P1plus and operator+ is given below.

```
Point1 P1plus(const Point1 &u, const Point1 &v)  // function
{
  double xsum = u.x + v.x;         // friend function can access private data
  double ysum = u.y + v.y;
  Point1 p;
  p.set(xsum, ysum);
  return p;
}

Point1 operator+ (const Point1 &u, const Point1 &v)
{
  double xsum = u.x + v.x;         // friend function can access private data
  double ysum = u.y + v.y;
  Point1 p;
  p.set(xsum, ysum);
  return p;
}
```

- **A friend function and/or friend operator can access all the private data in a class.**

2 Notes on friendship

- Friendship is actually quite a picky or peculiar thing.
- **The "friend" statement can appear anywhere in the class declaration.**
  1. The friend statement does not have to appear at the end of the class declaration.
  2. The friend statement does not have to appear in the `private` section.
  3. The friend statement can be written in the `public` section of the class declaration.
  4. The function/operator would still have access to all the private data.
- We can grant friendship to almost anything.
  1. **We can grant friendship to an entire class.**
  2. Suppose we have two classes `A` and `B`.
  3. We can declare that `A` is a friend of `B`.
  4. **Then all data and methods of `A` have access to the private data of `B`.**
  5. We can also declare that `B` is a friend of `A`.
  6. They can both be friends of each other.

```
class A
{
  friend class B;
  // etc
};

class B
{
  friend class A;
  // etc
};
```

- In addition to a friend function, operator and class, we can also declare a **friend method.**
  1. *Oh yes!*
  2. *Class B can grant friendship* **to only one method of class `A`** *not to the entire class* `A`*.*
  3. This is basically a friend function, where the function is a method of another class.
  4. See the next page for an example of two classes `AFriend` and `BFriend`.
  5. One method of `AFriend` is declared as a friend of `BFriend`.
  6. If the commented line is uncommented, it will generate a compilation error, because `readB` is not a friend of the class `BFriend` therefore `readB` has no access to the private data of `BFriend`.

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

class BFriend;

class AFriend
{
public:
  void writeB(BFriend& bf, int i, int j);
  void printB(const BFriend& bf);
};

class BFriend
{
public:
  friend void AFriend::writeB(BFriend& bf, int i, int j);
  int get_a() const { return a; }
  int get_b() const { return b; }

private:
  int a, b;
};

void AFriend::writeB(BFriend& bf, int i, int j)
{
  bf.a = i;                    // friend method has access to private data
  bf.b = j;
}

void AFriend::printB(const BFriend& bf)
{
  cout << "readB: " << bf.get_a() << "   " << bf.get_b() << endl;
  //cout << "readB: " << bf.a << "   " << bf.b << endl;            // COMPILATION ERROR
}

int main()
{
  int i = 3;
  int j = -4;
  AFriend af;
  BFriend bf;
  af.writeB(bf, i, j);
  af.printB(bf);
  return 0;
}
```

3 Comments on friendship

- If a class `A` has already been written, and at a later date new code is written for the project and it is decided to add friends to the class `A`, *then the declaration of the class* `A` *must be edited and all the code for the class* `A` *must be recompiled.*
- This can be a nuisance in large projects, especially if the class `A` was written long ago. A new round of software testing of old code must be performed.
- When an important new function is written, and the code for it becomes too cumbersome, then a decision may have to be made to make it a friend of the classes to which it needs access. Granting friendship should be employed judiciously, when the alternative choices for the software design are bad.
- Friendship is not inherited. (We shall study the important concept of inheritance later in this course.)