EECS402 Lecture 16

Andrew M. Morgan

Savitch Ch. 10.3, 8.2

The "this" Pointer

Friend Functions

Friend Classes

Consider The Following Program


```
class Point2Class
{
  private:
    float x;
    float y;
  public:
    Point2Class(float inX, float inY):x(inX), y(inY)
    { ; }
    float getX() const
    { return x; }    //Bad style
    float getY() const
    { return y; }    //Bad style
};

ostream &operator<<(ostream &os, const Point2Class &rhs)
{
  os << "Point2Class attrs: x: " << rhs.getX() <<
        " y: " << rhs.getY();
  return os;
}

int main()
{
  Point2Class p2a(6.5, 9.1);
  Point2Class p2b(7, -7);
  cout << p2a << endl;
  cout << p2b << endl;
  return 0;
}
```

```
Point2Class attrs: x: 6.5 y: 9.1
Point2Class attrs: x: 7 y: -7
```

Program Discussion

- The previous program works fine and was designed well, providing an overloaded insertion operator (<<) for the class user
  - What if an additional requirement is made to have a member function called "printInfo" that provides the same functionality as the overloaded insertion operator?
  - You don't want to duplicate the functionality in the new function
    - Having duplicate code is problematic if a change needs to be made. Often, the change is made in only one location, and the duplicate code no longer works the same way
  - You don't want to have to move the functionality in the existing function to the new function
    - This can be time consuming, especially for a function much larger than the example
- Ideally, your new member function would make use of the existing functionality in the insertion operator

Addition Of printInfo Function

```
class Point2Class
{
  private:
    float x;
    float y;
  public:
    Point2Class(float inX, float inY):x(inX), y(inY)
    { ; }
    float getX() const
    { return x; }
    float getY() const
    { return y; }
    void printInfo() const;
};

ostream &operator<<(ostream &os, const Point2Class &rhs)
{
  os << "Point2Class attrs: x: " << rhs.getX() <<
          " y: " << rhs.getY();
  return os;
}

void Point2Class::printInfo() const
{
  cout << ???????? << endl;
}
```

The << operator expects a reference to an object of type Point2Class on the right-hand-side. What do you write in place of the question marks?

Intro To "this" Pointer

- In order for a member function to be called, it must be called "on and object" using the dot or arrow operator
  - i.e. "p2a.printInfo();" calls the printInfo function on the p2a object
- The printInfo() function doesn't seem to take any parameters
  - However, ***there is an implicit parameter that is the first parameter to every member function***
  - This parameter doesn't appear in the prototype, header, or call, but it IS there
  - It is a pointer to the object that the function was called on, and is called "this"

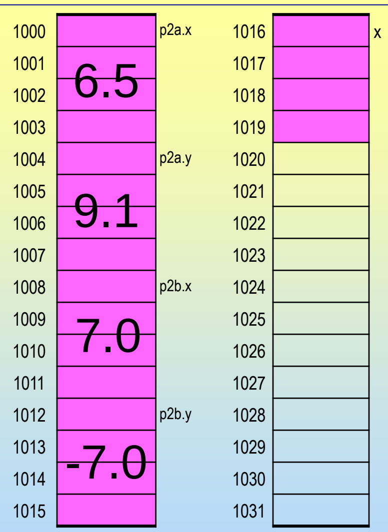
```
class Point2Class
{
  private:
    float x;
    float y;
  public:
    //...
    float getX() const
    { return x; }
};
```

These two examples are interchangeable, since the "this" parameter is implicit to the getX (and all other) member function.

```
class Point2Class
{
  private:
    float x;
    float y;
  public:
    //...
    float getX(
         //Point2Class *this
         ) const
    { return this->x; }
};
```

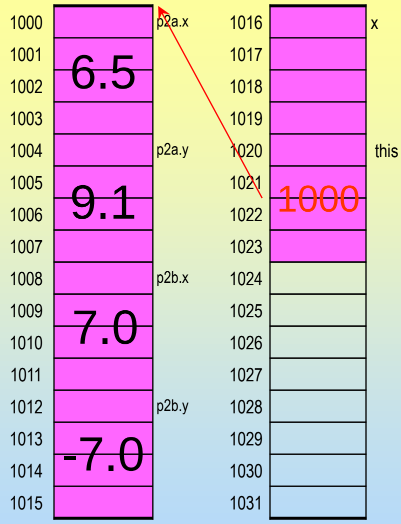

This Pointer Example, p.1

```
class Point2Class
{
  private:
    float x;
    float y;
  public:
    //...
    float getX(
         //Point2Class *this
         ) const
    {
      return this->x;
    }
};
int main()
{
  Point2Class p2a(6.5, 9.1);
  Point2Class p2b(7, -7);
  float x;

  x = p2a.getX();
  x = p2b.getX();
```



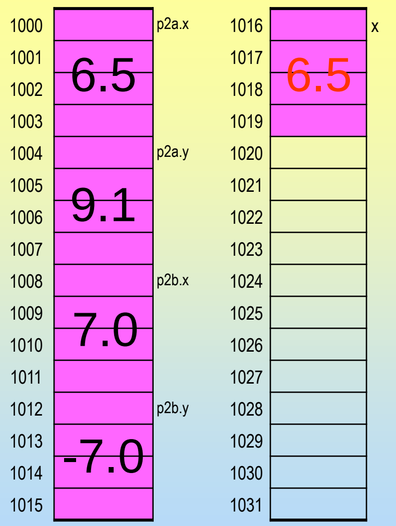
This Pointer Example, p.2


```
class Point2Class
{
  private:
    float x;
    float y;
  public:
    //...
    float getX(
         //Point2Class *this
         ) const
    {
      return x; //actually this->x
    }
};
int main()
{
  Point2Class p2a(6.5, 9.1);
  Point2Class p2b(7, -7);
  float x;

  x = p2a.getX();
  x = p2b.getX();
```



This Pointer Example, p.3


```
class Point2Class
{
  private:
    float x;
    float y;
  public:
    //...
    float getX(
         //Point2Class *this
         ) const
    {
      return x; //actually this->x
    }
};
int main()
{
  Point2Class p2a(6.5, 9.1);
  Point2Class p2b(7, -7);
  float x;

  x = p2a.getX();
  x = p2b.getX();
```



This Pointer Example, p.4


```
class Point2Class
{
  private:
    float x;
    float y;
  public:
    //...
    float getX(
         //Point2Class *this
         ) const
    {
      return x; //actually this->x
    }
};
int main()
{
  Point2Class p2a(6.5, 9.1);
  Point2Class p2b(7, -7);
  float x;

  x = p2a.getX();
  x = p2b.getX();
```

1000		p2a.x	1016		x
1001	6.5		1017	6.5	
1002			1018		
1003			1019		
1004		p2a.y	1020		this
1005	9.1		1021	1008	
1006			1022		
1007			1023		
1008		p2b.x	1024		
1009	7.0		1025		
1010			1026		
1011			1027		
1012		p2b.y	1028		
1013	-7.0		1029		
1014			1030		
1015			1031		

This Pointer Example, p.5


```
class Point2Class
{
  private:
    float x;
    float y;
  public:
    //...
    float getX(
         //Point2Class *this
         ) const
    {
      return x; //actually this->x
    }
};
int main()
{
  Point2Class p2a(6.5, 9.1);
  Point2Class p2b(7, -7);
  float x;

  x = p2a.getX();
  x = p2b.getX();
```

1000		p2a.x	1016		x
1001	6.5		1017	7.0	
1002			1018		
1003			1019		
1004		p2a.y	1020		
1005	9.1		1021		
1006			1022		
1007			1023		
1008		p2b.x	1024		
1009	7.0		1025		
1010			1026		
1011			1027		
1012		p2b.y	1028		
1013	-7.0		1029		
1014			1030		
1015			1031		

Back To The Original Problem (printInfo)


```
//Point2Class definition as before

ostream &operator<<(ostream &os, const Point2Class &rhs)
{
  os << "Point2Class attrs: x: " << rhs.getX() <<
          " y: " << rhs.getY();
  return os;
}

void Point2Class::printInfo() const
{
  cout << *this << endl;
}

int main()
{
  Point2Class p2a(6.5, 9.1);
  Point2Class p2b(7, -7);

  cout << p2a << endl;
  cout << p2b << endl;

  p2a.printInfo();
  p2b.printInfo();

  return 0;
}
```

```
Point2Class attrs: x: 6.5 y: 9.1
Point2Class attrs: x: 7 y: -7
Point2Class attrs: x: 6.5 y: 9.1
Point2Class attrs: x: 7 y: -7
```

Dereferencing the "this" pointer represents the object that the function was called on.

"this" type is: Point2Class *. By using dereference, resulting type is "Point2Class" and can be passed as the second parameter to the insertion operator.

Restriction Due To The "this" Pointer

- The "this" pointer will ***always*** be the first parameter to any member function
- Consider overloaded operators such as the insertion operator:
  - cout << p2a;
  - The left-hand-side argument is type ostream
  - Since the left-hand-side argument is the first parameter to the operator, this operator can NOT be a member of Point2Class
    - You an only have one first parameter – the "this" pointer of type Point2Class*
  - Potentially it could be a member of the ostream class, but you aren't allowed to add members to that class
  - Therefore, it must be declared as a global function (always)
- Consider commutative operations, such as addition:
  - obj + 8 == 8 + obj
  - The first version, "obj + 8" can be a member of obj's class, since the object is the left-hand-side parameter
  - The secone version, "8 + obj" can NOT be a member of obj's class, though, since the first parameter is not an int). Must be written as a global function!

Friends, Motivation

- Recall overloading the insertion operator
  - Since first parameter is an ostream, the operator can not be a member function of your user-defined class

```
class IntClass
{
  private:
    int val;
  public:
    //Assign to an integer var
    void operator=(int iVal)
    {
      val = iVal;
    }
    //Reader function for val
    int getVal() const
    {
      return val;
    }
};
ostream& operator<<(ostream &os, const IntClass &iObj)
{
  os << iObj.getVal();
  return os;
}
```

Notes about this code:

1. Function calls are always fairly inefficient. operator<< has to call getVal in order to access private data member from class

2. In this example, the existence of the getVal function was only to allow operator<< access to the private data.

3. Having many extra functions, such as getVal, that are otherwise unnecessary, and requiring operator<< to call (possibly many) reader functions, are both undesirable side effects of desirable encapsulation

Friend Functions

- Making a function a "friend function" allows that function direct access to private data, while keeping the data private to others
- Friend functions should not be used often – only in special circumstances
  - In general, you would consider making a function a friend function when:
    - The function is delivered to the class user along with the class
    - It needs access to private data and could represent an efficiency problem by calling many functions
- A function that is a friend function to a class ***is not*** a member function!
  - It is still a global function, it just has special privileges
- Syntax is easy: Just add keyword "friend" and function prototype inside the class definition

New Version Of operator<<


```
class IntClass
{
  private:
    int val;
  public:
    //Assign to an integer var
    void operator=(int iVal)
    {
      val = iVal;
    }

    //NOTE: getVal function may no longer be necessary

    friend ostream& operator<<(ostream &os, const IntClass &iObj);
};

ostream& operator<<(ostream &os, const IntClass &iObj)
{
  os << iObj.val;  //No function call required here
  return os;
}
```

is there an example of this?

This one function is allowed to access private data directly (without using the class interface). All other global functions must still use interface, however. Therefore, the getVal function may still be necessary to support other functions.

Using Friend Functions

- Calls to friend functions are no different than calls to non-friends
  - The function only needs to be declared a friend inside the class definition in order to be made a friend.
  - Since this declaration must be inside the class, other class users don't have the ability to add friend functions
  - Therefore, you can still make guarantees about your class' behavior
- Friend functions should be used only sparingly, and only after consideration of other possibilities
  - Can the function be made a member function? If so, that is likely a better solution
- Friend functions should not be used simply as a convenience
  - Data is made private for a reason, and unless there is a very good reason for making a function a friend, its access should remain restricted

Friend Member Functions

- Member functions of other classes may also be declared friend functions
  - Scope resolution required to indicate function is a member

```
class AClass; //Forward declarations.
class BClass; //Similar to function
              //prototypes, but for classes

class BClass
{
  private:
    int b;
  public:
    BClass(int inB):b(inB)
    { ; }
    void addA(const AClass &aObj);
    void print();
};
class AClass
{
  private:
    int a;
  public:
    AClass(int inA):a(inA)
    { ; }
    friend void BClass::addA(const AClass &aObj);
};
```

```
void BClass::print()
{
  cout << "b: " << b << endl;
}
void BClass::addA(const AClass &aObj)
{
  b += aObj.a; //has direct access!
}

int main()
{
  AClass aObj(7);
  BClass bObj(6);
  bObj.addA(aObj);
  bObj.print();
  return 0;
}
```

```
b: 13
```


Friend Classes

- Sometimes, two classes are created, and work together
  - For example, consider two classes EECS280Class, and EECS280StudentClass.
  - EECS280Class is simply a collection (i.e. a "container") of EECS280StudentClass objects
  - Separated for logical design, but EECS280Class member functions will need to access EECS280StudentClass objects
- When all, or most, member functions of one class need direct access to another class' data, each can be declared friends
- An easier syntax is simply to make the whole class a friend
  - This implies that all member functions become friend functions
- Declare the class as a friend, just like declaring a friend function

Declaring A Class A Friend


```
class AClass; //Forward declarations.
class BClass; //Similar to function
              //prototypes, but for classes

class BClass
{
  private:
    int b;
  public:
    BClass(int inB):b(inB)
    { ; }
    void addA(const AClass &aObj);
    void subA(const AClass &aObj);
    void print();
};
class AClass
{
  private:
    int a;
  public:
    AClass(int inA):a(inA)
    { ; }
    friend class BClass;
};
```

All BClass member functions are granted unrestricted access to private data of AClass

```
void BClass::print()
{
  cout << "b: " << b << endl;
}
void BClass::addA(const AClass &aObj)
{
  b += aObj.a; //has direct access!
}
void BClass::subA(const AClass &aObj)
{
  b -= aObj.a; //has direct access!
}

int main()
{
  AClass aObj(7);
  BClass bObj(6);
  bObj.addA(aObj);
  bObj.print();
  return 0;
}
```

```
b: 13
b: 6
```


When To Use Friends

- Friend functions are normally used:
  - To support overloaded operators that ***can't*** be member functions
- Friend classes are normally used:
  - When you create a container class that contains objects of another class type
- There are rarely other circumstances in which friends are a good design decision
- Do ***not*** use friend functions or classes without serious design considerations first!
- In this class, friends are never allowed unless specifically allowed in the program specifications