C++ Classes, Operators, Friends

Some content adapted from 'Absolute C++' by Walter Savitch

Administrative

- Next project soon
- It may be the basis for the following project
- I have not graded Project 4 yet.
- Quiz Friday

Outline

- Class Object Review
- Simple Class skeleton
- Making constructors/operators functions inaccessible
- Static Members
- Operator overloading
- Friends

Review Classes and const

- For parameters

  void add (const myClass& f2);

  (call would be f1.add(f2))
  - cannot change f2
- For objects

  void write( ) const;
  - cannot change the object
- For return types

  const Fraction copy( )
  - cannot modify the result returned (but if assigned to a variable, can modify)

Review-Shallow and Deep Copies

- Shallow copy
  - Assignment copies only member variable contents over (so only pointer addresses copied, NOT the data pointed to)
  - This is how default (compiler generated) assignment and copy constructors work
  - Fine if No dynamic memory involved
- Deep copy
  - Pointers, dynamic memory involved
  - Must dereference pointer variables to "get to" data for copying.

Review-When your object holds dynamic data

- YOU MUST IMPLEMENT
  - **Class destructor**
    - Special member function
    - Automatically destroys objects
  - **Copy constructor**
    - Single argument member function
    - Called automatically when temp copy needed
    - MUST DO DEEP COPY
  - **Assignment operator**
    - Must be overloaded as member function
    - MUST DO DEEP COPY

- Assignment operator Returns reference for chaining

Review - Using Classes

- If created with new (on heap)
  - Object->method()
  - Combines dereference and . Operator
  - Example:

```
MyClass *p;
p = new MyClass;
p->grade = "A";  // Equivalent to:
(*p).grade = "A";
```

- If created on stack
  - MyClass mc;
    mc.grade = "A";

- **Stack usually faster but limited in max size**
- **Heap allocations can be much larger**

Simple Class Skeleton for objects that will hold dynamic data

- Use as basis for objects that hold dynamic data
- See 17_Classes CopyAndAssign class.

What if you do not want to allow copy and/or assignment?

- Declare function as private in header file
- But wait, friends and member functions can still get to them.

- Solution: do not define them in the cpp file.
- If you try to call them you get a linker error

- This works for any function BTW
- See cantCallMe(); in StaticDemo class in 17_Classes

Static Members

```cpp
#pragma once
class staticDemo
{
public:
    staticDemo(void);
    virtual ~staticDemo(void);
    static int getNumberInstances();
private:
    static int numberInstances;
};
```

```cpp
#include "stdafx.h"
#include "staticDemo.h"

staticDemo::staticDemo(void)
{
    staticDemo::numberInstances++;
}


staticDemo::~staticDemo(void)
{
    staticDemo::numberInstances--;
}

int staticDemo::getNumberInstances(){
    //since this object is static only
    //static objects can be referenced
    return staticDemo::numberInstances;
}
//initialize static var
int staticDemo::numberInstances=0;
```

- **Pragma once MS equiv of include guards**
- **Syntax for function and var**
- **This class tracks the number of instances of itself**

- **Do not need calling object**
- **So not part of particular object, cannot access non static object data**
- **So static functions must only access static member vars**
- **Since no instance of class required initialize outside of class**

- **Start here 10/30**

Operator Overloading Introduction

- Operators <,+, -, %, ==, etc.
  - Really just functions!
- Simply "called" with different syntax:
  x < 7
  - “<" is binary operator with x & 7 as operands
  - We "like" this notation as humans
- Think of it as:
  <(x, 7)
  - “<" is the function name
  - x, 7 are the arguments
  - Function “<" returns bool of it’s arguments
- Can be done 2 ways
  - Overload as an object member function
  - Overload as a non member function

Operator Overloading Why

- Already work for C++ built-in types (int, double, etc.)
- Our types get same built in behavior. But we can (and usually need to) customize it programmatically.

Did this already for objects with dynamic data

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

Implement this one to simplify sorting using std::sort

- **There are some rules:**
  - Cannot overload: . :: sizeof ?: .*
  - Overloaded operators cannot have default parameters
  - You cannot invent operators and cannot change the precedence

Overload < operator – member function

- Remember vector sort needed a special compare function

```
bool compareName(const Student_info& x, const Student_info& y)
{
    return x.name < y.name;
}

sort(students.begin(), students.end(), compareName);
```

- Do not need this if overload < operator (See 17_Classes)

```
#pragma once
class sortable
{
public:
    sortable();
    ~sortable(void);
    void setValue(int value);
    int getValue();
    bool operator< (const sortable& param);
private:
    int value;
};

bool sortable::operator< (const sortable& param)
{
  return value<param.value;
}
```

```
//sort using sortables operator <
//no more custom sort functions needed
//its all encapsulated, the object knows
//how to sort itself
sort(myVector.begin(),myVector.end());
```

Overload < operator – non member function

```
#pragma once
class sortable
{
public:
    sortable();
    ~sortable(void);
    void setValue(int value);
    int getValue() const;
private:
    int value;
};

bool operator< (const sortable& param1,const sortable& param2);
```

```
//sort using sortables operator <
//no more custom sort functions needed
//its all encapsulated, the object knows
//how to sort itself
sort(myVector.begin(),myVector.end());
```

```
//non member operator overloading
bool operator< (const sortable& param1,const sortable& param2)
{
  return (param1.getValue() <param2.getValue());
}
```

- 2 parameters instead of 1
- Requires access to private data
- Which means Getters and Setters
- Violates encapsulation, inefficient

Overloading Operators: Which Method?

- Object-Oriented-Programming
  - Principles suggest member operators
  - Many agree, to maintain "spirit" of OOP
- Member operators more efficient
  - No need to call getters and setters

Friend Functions

- Sometimes a function may require access to private parts of an object (like non member operator overloads do).
- Declaring the function to be a friend of the class, gives it access to protected and private members of the class without opening up access to anyone else.

Friend Functions - fixes getter and setter operator problem

```
#pragma once
class sortable
{
public:
    sortable();
    ~sortable(void);
    void setValue(int value);
    int getValue() const;
    friend bool operator< (const sortable& param1,const sortable& param2);
private:
    int value;
};
```

```
bool operator< (const sortable& param1,const sortable& param2)
{
  return (param1.value <param2.value);
}
```

Friend Functions

- Friend function of a class
  - Not a member function
  - Has direct access to private members
    - Just as member functions do
- Use keyword *friend* in front of function declaration
  - Specified IN class definition
  - But they're NOT member functions!

Friend Function Purity

- Friends not pure?
  - "Spirit" of OOP dictates all operators and functions be member functions
  - Many believe friends violate basic OOP principles
- Advantageous?
  - For non member operators: very!
  - Still encapsulates: friend is in class definition
  - Improves efficiency (no getters or setters)

Friend Classes

- Entire classes can be friends
  - Similar to function being friend to class
  - Example:
    class F is friend of class C
    - All class F member functions are friends of C
    - NOT reciprocated
    - Friendship granted, not taken
- Syntax: friend class F
  - Goes inside class definition of "authorizing" class

Summary

- Simple class skeleton for classes utilizing dynamic memory
- Static Members
- C++ built-in operators can be overloaded
  - To work with objects of your class
- Operators are really just functions
- Friend functions have direct private member access
- Friend functions add efficiency only
  - Violate OOP?