

# C++ Classes odds and Ends, Operators

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

# Outline

- Review Classes - Stack verses Heap
- Review Shallow verses deep copies
- Review Objects and Dynamic data
- The 'this' pointer
- Making constructors/operators functions inaccessible
- Static Members
- Operator overloading

# Review – Classes –stack verses Heap

- If created with new (on heap)

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

- If created on stack

- MyClass mc;  
mc.grade = "A";

# 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. (or stateful objects like DB or network connections)
- 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

See sample program I gave you

# The this Pointer

- Member function definitions might need to refer to calling object
- Use predefined *this* pointer

- Automatically points to calling object:

```
Class Simple
```

```
{
```

```
public:
```

```
    void showStuff() const;
```

```
private:
```

```
    int stuff;
```

```
};
```

- Two ways for member functions to access:

```
cout << stuff;
```

```
cout << this->stuff;
```

# Static Members

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

**This object used to track number of active Instances of staticDemo object**

## Static members

Do not need instance of class

Cannot access non static member variables

## Static functions

Can only access static mem vars and other static functions

```
#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; ←
```

# 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

```
//assignment operator  
HoldsDynamicData & operator= (const HoldsDynamicData & other);
```

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

Implement this one to simplify  
sorting using `std::sort`

# Sorting – what < overload buys you

- Remember using a get function or a friend function to help with sorting?
- Implement < operator. Then the object knows how to sort itself

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

```
vector<sortable> myVector;
//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());
```

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

Just 2 arguments

implementation

# Sorting— as opposed to using getters

- Remember using a get function or a friend function to help with sorting?

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

← Exposed data here

```
vector<sortable> myVector;
std::sort(myVector.begin(), myVector.end(), compareVal);
```

```
bool compareVal(const sortable &l, const sortable &r){
    return l.getValue() < r.getValue();
}
```

# Summary

- Static Members
- C++ built-in operators can be overloaded
  - To work with objects of your class
- Operators are really just functions