Today …

• Quiz 2
• Homework 1 is due on Thursday
• Topics
  – Using const
  – Operator overloading
Organizing your files ...

Organizing Header Files

Within a header file myclass.h ...

```c
#ifndef MYCLASS_H_
#define MYCLASS_H_
...

class MyClass {
    ...
};
#endif
```

These commands ensure your header files are only read once

Also often used to turn on/off debug statements
### ifndef define endif

- These are called “preprocessing” commands
  - give directions to the compiler, before compilation

  ```c
  ifndef <variable>
  ```
  - True if the preprocessing variable has not been defined

  ```c
  ifndef <variable>
  ```
  - True if the preprocessing variable has not been defined

  ```c
  define <variable>
  ```
  - Define a preprocessing variable (as a “token”)

  ```c
  endif
  ```
  - Ends the section started by the #ifdef or ifndef

---

### When compiling …

You should be able to compile your programs using:

```
g++ class1.cpp class2.cpp main.cpp
```

where:
- class1.cpp #includes class1.h
- class2.cpp #includes class2.h
- etc.

- Using the #ifndef #define #endif conventions …
  - Compiler won’t read .h files multiple times
  - Reading header files multiple times can lead to problems

- A caveat is with templates (more later)
Using the `const` keyword ...

The various uses of `const`

A simple class for managing stock shares ...

class Stock {
public:
    Stock();
    Stock(string aCompany, int theShares);
    int getShares();
    string getCompany();
    void setShares(int newShares);
    Stock worthMore(Stock s);
private:
    string company;
    int shares;
};
The various uses of \texttt{const}

Lets say we want to do the following

\begin{verbatim}
const Stock s1("GOOG", 1000);
s1.setShares(900); // sell shares
\end{verbatim}

- What will happen when we compile?

\textbf{Compile Error!}

"error: passing 'const Stock' as 'this'
argument of 'void Stock::setShares(int)'
discards qualifiers"

- Why is the compiler complaining?

... Because \texttt{s1} is a constant, and we are modifying it!

The compiler is our friend!
It is saving us from introducing bugs later

The various uses of \texttt{const}

Lets say now we want to do the following

\begin{verbatim}
const Stock s1("GOOG", 1000);
cout << s1.getShares() << endl; // check shares
\end{verbatim}

- What will happen when we compile?

\textbf{Compile Error!}

"error: passing 'const Stock' as 'this'
argument of 'void Stock::getShares()'
discards qualifiers"

- Why is the compiler complaining now?

... The compiler can't tell we aren't modifying \texttt{s1}

The compiler is our friend!
But we have to help it...
The various uses of `const`

We need to tell the compiler `getShares()` won’t change the invoking object

```cpp
int getShares() const;
```

• Now when we compile

```cpp
const Stock s1("GOOG", 1000);
cout << s1.getShares() << endl; // check shares
```

– The compiler is happy

• All “getters” should be `const` functions

The various uses of `const`

What happens after we call this function?

```cpp
Stock s = s1.worthMore(s2);
```

• A `copy` of either `s1` or `s2` is made
  – In general, creating lots of copies is `inefficient`
  – Many copies are actually being made here …

• How can we avoid copying?
  – Pass objects by reference …

```cpp
Stock& worthMore(const Stock& s) const; // by ref
```
The various uses of `const`

Now we return the Stock by reference

```
Stock s = s1.worthMore(s2); // '==' still copying
```

- What will happen when we compile?

  **Compile Error!**

  "error: invalid initialization of reference of type 'Stock&' from expression of type 'const Stock'"

  - Why is the compiler complaining now?

The various uses of `const`

The compiler is complaining because we could do this

```
Stock& s = s1.worthMore(s2);
            s.setShares(s.getShares() - 100); // modify s
```

- But `worthMore` (i.e., `s1`) and its argument (`s2`) are `const`!

- Luckily the compiler caught our mistake!

- Are we stuck?

- Nope ... we can add more `const` keywords
The various uses of `const`

We can define `worthMore` to return a `const` reference.

```cpp
const Stock& worthMore(const Stock& s) const;
```

The compiler is happy again:

```cpp
Stock s = s1.worthMore(s2);
```

But what about now:

```cpp
Stock& s = s1.worthMore(s2);
```

What is the difference? How can we fix it?

```cpp
const Stock& s = s1.worthMore(s2);
```

The various uses of `const`

When should we not return by reference?

- Never return a temporary variable (compiler error)
  ```cpp
  const string& badIdea1(string s1, string s2)
  {
    string tmp = s1 + s2;
    return tmp; // same if: return s1+s2
  }
  ```
  What happens to `tmp` after the function exits?

- Returning object state ...
  ```cpp
  const string& badIdea2() const
  {
    return name; // data member
  }
  ```
  What happens to `name` if the object is deallocated?
The various uses of \texttt{const}

- In general, we want to \texttt{pass by reference}
  - Especially when passing around objects
  - Can lead to more efficient programs

- Usually need to ensure these are \texttt{const} references
  E.g.: \texttt{void setCompany(const string& newCompany)};

- Sometimes inputs are outputs in C++
  - For outputs, args are passed by reference but \texttt{not} \texttt{const}
  - To make an argument an output, it \texttt{must} be a reference

---

Our revised class for managing stock shares ...

```cpp
class Stock {
public:
    Stock();
    Stock(const string& aCompany, int theShares);
    int getShares() const;
    string getCompany() const;
    void setShares(int newShares);
    void setCompany(const string& newCompany);
    const Stock& worthMore(const Stock& s) const;
private:
    string company;
    int shares;
};
```

Be sure you understand these definitions ... and the different uses of \texttt{const} here!
Exercise #1: Using `const`

Add `const` keywords and references to these functions:

class Department {
    public:
        Department();
        Department(const string& name, const Company& cmp);
        void setDeptName(const string& name);
        string getDeptName() const;
        Company getCompany() const;
        void addEmployee(const Employee& emp);
        Employee getEmployee(const string& n) const;
        void removeEmployee(const string& n);
    private:
        ...
};

Really depends on the behavior of these functions (but here we can guess since getters/setters)!
Operator overloading (first pass) ...

Basic operator overloading

• Who remembers what function overloading is?
  – Same function name, different argument types

• C++ supports overloading of functions & operators!
  – We can define, e.g., +, -, ==, <=, … for our objects

```cpp
int x, y;
... if(x < y)
    cout << "x < y\n";
else
    cout << "x >= y\n";
... Stock s1, s2;
... if(s1 < s2)
    cout << "s1 < s2\n";
else
    cout << "s1 >= s2\n";
...```
Basic operator overloading

We’ll start with the ==, <, and = operators

– You need to implement these for HW1
– More operator overloading details later

• Operators are overloaded via operator functions
  
  return-type operatorop(arg-list)

  – operator*(…), operator>(…), operator++(…), etc.
  – The number of args depends on the type of operator
  – Most operators are either unary (1-arg) or binary (2-args)

Basic operator overloading

We can call either the operator or the function …

```cpp
#include <iostream>
#include <string>   // defines operator+ for strings
using namespace std;

int main()
{
    string x, y = "foo", z = "bar";
    x = y + z;            // more readable
    cout << "x = " << x << endl; // prints "foobar"
    x = operator+(y, z);   // less readable
    cout << "x = " << x << endl; // prints "foobar"
}
```

Operator Overloading for Classes

ClassName operator-(const ClassName& arg1, const ClassName& arg2);

- Defines subtraction operator (-) on two objects of the same class, returning an object of that class
  - Note the use of const …
- Is this a unary or binary operator?

Stock example again …

Let's add a “<” operator to our Stock class …

class Stock {
    public:
        int getShares() const;
        double getShareValue() const;
        ...
    }

    // in Stock.cpp:
    bool operator<(const Stock& s1, const Stock& s2)
    {
        ... what should go here???
    }
Stock example again …

Let's add a "<" operator to our Stock class …

```cpp
class Stock {
public:
    int getShares() const;
    double getShareValue() const;
    ... 
};
```

Note similar to "worthMore" function

// in Stock.cpp:
bool operator<(const Stock& s1, const Stock& s2) {
    double v1 = s1.getShares() * s1.getShareValue();
    double v2 = s2.getShares() * s2.getShareValues();
    return v1 < v2;
}
```

Operator Overloading for Classes

```cpp
ClassName operator-(const ClassName& arg1, const ClassName& arg2);
```

We can also overload operators as member functions

```cpp
class ClassName {
public:
    ClassName operator-(const ClassName& arg2);
    ...
};
```