Week 13 - Wednesday

COMP 2400

Last time

- What did we talk about last time?
- OOP in C++
- Dividing code into headers and implementation files
- Operator overloading

Questions?

Project 6

Quotes

C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg.

Bjarne Stroustrup
Creator of C++

C++ Operator Overloading

Dividing up code header

```
class Complex
{
  double real;
  double imaginary;

public:
  Complex(double realValue = 0, double
  imaginaryValue = 0);
  ~Complex(void);

  double getReal();
  double getImaginary();
};
```

Dividing up code implementation

```
Complex::Complex(double realValue, double imaginaryValue)
{
  real = realValue;
  imaginary = imaginaryValue;
}

Complex::~Complex(void)
{}

double Complex::getReal()
{ return real; }

double Complex::getImaginary()
{ return imaginary; }
```

(Partial) overloading operators header

```
Complex& operator=( const Complex& complex );

Complex operator+( const Complex& complex ) const;

Complex operator-( const Complex& complex ) const;

Complex operator-() const;

Complex operator*( const Complex& complex ) const;
```

(Partial) overloading operators implementation

```
Complex& Complex::operator=
(const Complex& complex)
{
  real = complex.real;
  imaginary = complex.imaginary;

  return *this;
}
```

Overloading << for output

- In Java, every object has a **`toString()`** method
- Whenever *you* create a Java class, it's a good idea to override the default **`toString()`** so that it gives meaningful output
- In C++, the standard approach for output is to overload the **<<** operator for **`ostream`** objects and your class
- Unfortunately, it's not a method in *your* class … because it would actually have to be in the **`ostream`** class because the first object is the one the operator is "called" on

`ostream` (calling object)	**<<** (operator/method)	**`object`** (argument)

C++ approach

- For situations like this one, C++ lets a class declare a **`friend`** method
- A friend method is a method that's not actually inside the class, but it is allowed to access private and protected member variables
- For example, the **`Complex`** class would contain the following method declaration for output:

```
friend ostream& operator<<(ostream& out, const Complex& complex);
```

Programming practice

- Let's finish the **`Complex`** type
- Then, we can do operations on some **`Complex`** objects and output the result

What's all that `const`?

- **`const`**, of course, means constant in C++
- In class methods, you'll see several different usages
- Const methods make a guarantee that they will not change the members of the object they are called on
  - `int countCabbages() const;`
- Methods can take const arguments
  - `void insert(const Coin money);`
- Methods can take const reference arguments
  - `void photograph(const Castle& fortress);`
- Why take a **`const`** reference when references are used to change arguments?

Templates

Templates

- Allow classes and functions to be written with a generic type or value parameter, then instantiated later
- Each necessary instantiation is generated at compile time
- Appears to function like generics in Java, but works very differently under the covers
- Most of the time you will **use** templates, not create them

Template method example

```
template<class T> void exchange(T& a, T& b )
{
  T temp = a;
  a = b;
  b = temp;
}
```

Template classes

- You can make a class using templates
- The most common use for these is for container classes
  - e.g. you want a `list` class that can be a list of anything
- The STL is filled with such templates
- Unfortunately, template classes **must** be implemented entirely in the header file
  - C++ allows template classes to be separate from their headers, but most compilers don't fully support this feature

Template class example

```
template<class T> class Pair {
  private:
      T x;
      T y;
  public:
      Pair(const T& a, const T& b) {
            x = a;
            y = b;
      }

      T getX() const { return x; }

      T getY() const { return y; }

      void swap() {
            T temp = x;
            x = y;
            y = temp;
      }
};
```

Programming practice

- Let's write an **`ArrayList`** class with templates!
- Methods:
  - **`void add(T element)`**
  - **`T get(int index)`**
  - **`T remove(int index)`**

STL

Standard Template Library

Containers

- `list`
- `map`
  - `multimap`
- `set`
  - `multiset`
- `stack`
- `queue`
  - `deque`
- `priority_queue`
- `vector`

Iterators

- Generalization of pointers
- No iterators for:
  - **`stack`**
  - **`queue`**
  - **`priority_queue`**
- Regular iterator operations:
  - Postfix and prefix increment and decrement
  - Assignment
  - == and !=
  - Dereference
- **`deque`** and **`vector`** iterators also have <, <=, >, >=, +, −, +=, and −=, and these containers also support [ ] access

STL example part 1

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
	int count;
	vector<string> words;
	vector<string>::iterator index;
	string word;
```

STL example part 2

```
        cout << "How many words will you enter? ";
        cin >> count;

        for(int i = 0; i < count; i++ )
        {
                cin >> word;
                words.push_back( word );
        }
        for(index = words.begin(); index != words.end(); index++)
                cout << *index << endl;
        return 0;
}
```

Algorithms

- Shuffle
- Find
- Sort
- Count
- Always use the ones provided by the container, if available
- Functors provided in **`<functional>`**

Ticket Out the Door

Upcoming

Next time…

- Review up to Exam 1

Reminders

- Keep working on Project 6
  - Due next Friday