



CS 202 Review

ERIN KEITH

Goals

1. OOP
2. Operator Overloading
3. Interfaces
4. Templates
5. makefiles

Homework 1

Project Goals

The goal of this project is to:

1. Review **C++**
2. Review **Templates**
3. Review **Interfaces**
4. Review **Operator Overloading**
5. Review **makefiles**

Goals

1. OOP
2. **Operator Overloading**
3. **Interfaces**
4. **Templates**
5. **makefiles**

Homework 1

Project Goals

The goal of this project is to:

1. Review C++
2. Review **Templates**
3. Review **Interfaces**
4. Review **Operator Overloading**
5. Review **makefiles**

POLYMORPHISM

Definition

- “The condition of occurring in several different forms.”

C++ tools:

- There are actually 4 types of polymorphism in C++!

POLYMORPHISM

C++ tools:

- Subtype
 - Runtime polymorphism
- Parametric
 - Compile-time polymorphism
- Ad-hoc
 - overloading
- Coercion
 - casting

Subtype Polymorphism

Definition

- Runtime Polymorphism (what everyone thinks of when they hear “polymorphism”)
- replacing base class place holders with objects of the derived type

C++ tools:

- base class pointers and references
- **virtual**

Example:

- How can we require that each derived class implement their own feed function?

Parametric Polymorphism

Definition

- Compile-time Polymorphism (what we'll use most in this class!)
- Execute the same code for any data type

C++ tools:

- **templates**

Example:

Templated Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template <typename ItemType>
class PlainBox{
    ItemType item;
public:
    PlainBox();
    PlainBox(const ItemType& theItem);
    void setItem(const ItemType& theItem);
    ItemType getItem() const;
};
#include "PlainBox.cpp"
#endif
```


Driver

```
#include "PlainBox.h"
```

```
#include <iostream>
using namespace std;
```

During compile time, the compiler to creates the code for a PlainBox class that holds a double.

```
int main() {
```

```
    PlainBox<double> dblBox(3.1415);
    PlainBox<int> intBox(42);
```

```
    cout << dblBox.getItem() << endl;
```

```
    cout << intBox.getItem() << endl;
```

```
    return 0;
```

```
}
```

Created Class

```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX

template <typename double>
class PlainBox{
    double item;
public:
    PlainBox();
    PlainBox(const double& theItem);
    void setItem(const double& theItem);
    double getItem() const;
};
#include "PlainBox.cpp"
#endif
```

Ad-hoc Polymorphism

Definition

- functions with the same name behave differently for each type
- particularly useful in conjunction with runtime polymorphism

C++ tools:

- Overloading
- Overriding

Ad-hoc Polymorphism

Consider the addition operator. It takes two operands and returns the sum. Do you think it works the same way for ints as for floats? Why does it work differently when the operands are of mixed type?

Would we ever want to add objects together? Can we? If so, how?

Operator overloading! Operators act like functions and we can define our own behavior through code.

Operator Overloading

Let's take consider another operator, the relational operator `==`. With scalar types, it checks for equality. What does that mean for objects which are instances of classes or composite data types? How should we determine equality? What should the data type of the expression be?

Generally, member variable by member variable.
Should evaluate to bool

Date Header File w/Overloaded Relational Operator

```
Date.h
#ifndef DATE_H
#define DATE_H

class Date {
    int month, day, year;

public:
    Date();
    Date(int, int, int);
    Date(const Date&);
    bool operator==(const Date& other) const;
    void getMonth(int) const;
    void getDay(int) const;
    void getYear(int) const;
    void setMonth(int);
    void setDay(int);
    void setYear(int);
    void outputAmerican();
    void outputEuropean();
};
#endif
```

Date Source File w/ Overloaded Relational Operator

Date.cpp

```
#include "Date.h"
bool Date::operator==(const Date& other) const {
    return ((month == other.month) && (day == other.day) &&
        (year == other.year));
}
```

driver.cpp

```
#include "Date.h"
#include <iostream>
using namespace std;
int main(){
    Date date1(12, 31, 21);
    Date date2(1, 1, 22);

    if(date1 == date2)
        cout << "Same date!" << endl;
}
```

Scope Resolution Operator

```
void Pet::eat(string food) {  
    cout << "I ate some " << food << "today!" << endl;  
}
```

Scope resolution operator – indicates which class the method belongs to

Tells the compiler that a function outside of the class definition “belongs” to that class. This is how we can implement class functions in the source files.

Scope Resolution Operator

```
void Pet::eat(string food) {  
    cout << "I ate some " << food << "today!" << endl;  
}
```

Scope resolution operator – indicates which class the method belongs to

```
void Pet::setLegs(int numLegs) {  
    legs = numLegs;  
}
```

Method body has direct access to Class member variables

Scope Resolution Operator

```
class Distance{  
    private:  
        int meter;  
    public:  
        Distance() ;  
  
        int addFive (Distance) ;  
};
```

Scope Resolution Operator

```
int addFive(Distance d) {  
    //compiler error!  
    d.meter += 5;  
    return d.meter;  
}
```

Friend Functions

In C++ the **friend** keyword gives access to private and protected members of the class to a non-member function.

```
class Distance{  
    private:  
        int meter;  
    public:  
        Distance(): meter(0) { }  
        //friend function  
        friend int addFive(Distance) ;  
};
```

Friend Functions

The **friend** keyword is not necessary in the source file.

```
// friend function definition
int addFive(Distance d) {
    //accessing private data from non-member function
    d.meter += 5;
    return d.meter;
}
```

Overloading Extraction and Insertion Operators

You're calling the function on the stream object, not your class object, so it is not a member function.

Since you still want the function to have access to the classes private and protected members, it should be a friend function.

Also, these functions should return the stream object so you can chain together calls to the operators.

Overloading Extraction and Insertion Operators

In header file:

```
friend ostream& operator<<(ostream&, Address);
```

In source file:

```
ostream& operator<<(ostream& out, Address address){  
    out << address.houseNumber << " " << address.street;  
  
    return out;  
}
```

*no “friend” keyword in source
return ostream object for chaining.

Overloading Extraction and Insertion Operators

In driver:

```
Address myAddress;  
cout << myAddress << endl;
```

Chaining: return the iostream object so the operator can continuously be called on that object.

Interfaces

similar to inheritance BUT

Inheritance is primarily a tool to share *implementation* across a set of classes

Interfaces are primarily a mechanism to share *behavior* across a set of classes

THEREFORE, interfaces do not have attributes.

Interfaces

in C++ support runtime polymorphism through

inheritance and function **overloading**

- have only a .h header file
- only pure virtual functions are included in the interface definition
- the syntax is the same as a regular class definition
- the difference is in the conventions

Pure Virtual Methods

Also known as an “abstract function”

Doesn't allow for implementation in the base class, because we can't know what it's going to be

A pure virtual function is created by assigning 0 in declaration

A class is abstract if it has at least one pure virtual function

We cannot create objects of abstract classes

Example

Balls

- Football
- Basketballs
- Tennis balls
- Baseballs

What do they do?

Interface

```
class IBall{  
    public:  
        virtual void bounce() = 0;  
        virtual void roll() = 0;  
        virtual void fly() = 0;  
};
```

Derived Classes Example

```
// Derived classes
class Football: public IBall{
private:
    double length;
    double diameter;
public:
    Football();
    Football(double, double);
    Football(const Football&);

    void bounce();
    void roll();
    void fly();

    void setLength(double);
    double getLength() const;
};
```

Derived Classes Example

```
#include "football.h"

Football::Football() {
    length = 0;
    diameter = 0;
}

Football::Football(double l, double d) {
    length = l;
    diameter = d;
}

Football::Football(const Football& rhs) {
    length = rhs.length;
    diameter = rhs.diameter;
}

void Football::setLength(double l) {
    length = l;
}

double Football::getLength() const {
    return length;
}

void Football::bounce() {
    cout << "Bouncing all wonky";
}

void Football::roll() {
    cout << "Rolling haphazardly";
}

void Football::fly() {
    cout << "Whistling through the air";
}
```

Derived Classes Example

```
// Derived classes
class Tennisball: public IBall{
    private:
        bool fuzz;
        double diameter;
    public:
        Tennisball();
        Tennisball(bool, double);
        Tennisball(const Tennisball&);

        void bounce();
        void roll();
        void fly();

        void setFuzz(bool);
        bool getFuzz() const;
};
```


Derived Classes Example

```
#include "tennisball.h"

Tennisball::Tennisball() {
    fuzz = 0;
    diameter = 0;
}

Tennisball::Tennisball(bool f, double d) {
    fuzz = f;
    diameter = d;
}

Tennisball::Tennisball(const Tennisball& rhs) {
    fuzz = rhs.fuzz;
    diameter = rhs.diameter;
}

void Tennisball::setFuzz(bool f) {
    fuzz = f;
}

bool Tennisball::getFuzz() const {
    return fuzz;
}

void Tennisball::bounce() {
    cout << "Bouncing so high";
}

void Tennisball::roll() {
    cout << "Rolling slowly";
}

void Tennisball::fly() {
    cout << "Zooming at your face";
}
```

Interfaces

Used a LOT moving forward

Design Principles

identify the aspects of your application that vary and separate them from what stays the same

program to an interface not an implementation

Compiler

g++

- GNU compiler for C++
- Using g++ to link the object files, files automatically link in the std C++ libraries.

Compiling Steps

1. Preprocessing
2. Compilation
3. Assembly
4. Linking

Compiling Steps

Preprocessing

- Removal of comments
- Expansion of macros
- Expansion of include files

Leaves us with just the code.

Compiling Steps

Compilation

- Converts the preprocessed code to assembly code
- Assembly code is machine dependent

Compiling Steps

Assembly

- Convert assembly code to binary **object** files
 - Object file formats are operating system dependent

Compiling Steps

Linking

- Links all the function calls together with addresses of function definitions
- Produces the final executable

Wind Analysis

```
#include <iostream>
using namespace std;

#define MAX_WIND 50

int getWindData(double[]);

int main(){
    int numWind;
    double windData[MAX_WIND];

    numWind = getWindData(windData);

    return 0;
}

int getWindData(double data[]){
    int numItems;

    cout << "Enter number of wind data: ";
    cin >> numItems;

    for(int i = 0; i < numItems; i++){
        cout << "Item " << i + 1 << ": ";
        cin >> data[i];
    }
    return numItems;
}
```

Wind Analysis

Let's add a function to calculate the average of the speeds.

```
double getAverage(int, double[]);

double getAverage(int numItems, double data[]){
    double sum = 0;

    for(int i = 0; i < numItems; i++){
        sum += data[i];
    }
    return sum / numItems;
}
```

Wind Analysis

We can add this to the main driver, but what if we want more statistical functions, such as finding the minimum or the maximum?

We could have a whole set of statistical functions, which we could use in other programs, but kept in a separate pair of files.

stats.cpp

We can add the `getAverage` function to another `.cpp` file, but how do we get them to match up when we compile?

Header files that are included in other source files become the bridges between source files.

stats.h

```
#ifndef STATS_H  
#define STATS_H
```

← Include guard

```
double getAverage(int, double[]);  
double findMinimum(int, double[]);  
double findMaximum(int, double[]);
```

```
#endif
```

Include Guard

- these header files are what get included in other files
- used to avoid the problem of double inclusion when dealing with the include directive.
- each guard must test and conditionally set a different preprocessor macro
- often this is `FILENAME_H`

stats.h

```
#ifndef STATS_H  
#define STATS_H
```

list of function prototypes



```
double getAverage(int, double[]);  
double findMinimum(int, double[]);  
double findMaximum(int, double[]);
```

```
#endif
```

The new project structure

- a main driver file
 - often called “driver.cpp”
 - a package of files containing related functions
 - header file
 - filename.h
 - source file
 - filename.cpp
- *a project can have multiple “packages”

The new project structure

We have all our files, but how to we connect them altogether when we compile?

makefile

a file (by default named “makefile”) containing a set of directives used by a make build automation tool to generate a target/goal.

Most often, the makefile directs Make on how to compile and link a program

makefile

A makefile consists of “rules” in the following form:

```
target: dependencies  
    system command(s)
```

A **target** is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as "clean".

A **dependency** is a file that is used as input to create the target. A target often depends on several files.

The **system command(s)** (also called recipe) is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line. Note that the indentation must consist of a single <tab> character.

makefile

```
wind: wind.o stats.o  
    g++ -o wind wind.o stats.o
```

← executable rule

```
wind.o: wind.cpp stats.h  
    g++ -c wind.cpp
```

← driver package rule

```
stats.o: stats.cpp stats.h  
    g++ -c stats.cpp
```

← stats package rule

```
clean:  
    rm *.o
```

← forces recompile

makefile

To compile any files that have changed:

make

To delete any object files and force everything to recompile the next time you make:

make clean

Class Templates

A template is not a class, it is a pattern that the compiler uses to generate a family of classes. For the compiler to generate the code, it must see both the template definition and the types to fill in the template.

Hence the term “compile-time polymorphism”

Your compiler can't remember the details of one .cpp file while it's compiling another.

Class Templates

Options

- Include the .cpp file at the end of the header file
- Keep declarations and implementation (normally placed in the .cpp source file) all in a single header file!

Example


```
#ifndef _PLAIN_BOX
#define _PLAIN_BOX
```

```
template <typename ItemType>
class PlainBox{
    ItemType item;
public:
    PlainBox();
    PlainBox(const ItemType& theItem);
    void setItem(const ItemType& theItem);
    ItemType getItem() const;
};
```

```
#include "PlainBox.cpp"
```

```
#endif
```

allows for source to be
saved in a different file



Example

```
//DON'T NEED THE HEADER HERE
```

```
template<typename ItemType>
```

```
PlainBox<ItemType>::PlainBox(): item() { }
```

```
template<typename ItemType>
```

```
PlainBox<ItemType>::PlainBox(const ItemType& theItem)
```

```
: item(theItem) { }
```

```
template<typename ItemType>
```

```
void PlainBox<ItemType>::setItem(const ItemType& theItem) {
```

```
    item = theItem;
```

```
}
```

```
template<typename ItemType>
```

```
ItemType PlainBox<ItemType>::getItem() const{
```

```
    return item;
```

```
}
```


makefile

```
box: driver.o
    g++ -o box driver.o

driver.o: driver.cpp plainbox.h
    g++ -c driver.cpp

clean:
    rm *.o box
```

Class Templates

A template is not a class, it is a pattern that the compiler uses to generate a family of classes.

For the compiler to generate the code, it must first know what data type to use when replacing the template label.

Therefore, you cannot directly compile a templated class. Instead it occurs when the declaration for a templated object is encountered during compilation.

Next Class

Module:

Week 2: Review

Topic:

Finish Review

Practice

