CSE 1325: Object-Oriented Programming

Lecture 05 – Chapter 8

Eclectic Topics

Declarations vs Definitions, Header Files, Static Class Members, Delegated Constructors, References, Namespaces, and Makefiles (Again)

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup
www.stroustrup.com/Programming

ERB 402
Office Hours:
Tuesday Thursday 11 - 12
Or by appointment

Lecture #4

Quick Review

- Common types of errors include which of the following: **b, c, d, and e**

  (a) Edit-time errors **(b) Compile-time errors**
  (c) Link-time errors **(d) Run-time errors**
  (e) Logic errors (f) Math errors

- To handle bad assumptions, a method should usually **c**

  (a) Document the assumptions and expect callers to comply

  (b) Force compliant types on parameters so the compiler can detect

  (c) Detect the invalid input and throw an exception

- What are the pros and cons of each strategy for handling errors?

  (a) Return an error code Easy to implement – **hard to manage unique error codes, caller may not check, checking return code obscures the program's logic**
  (b) Set a global error variable **Can check after several operations, but otherwise same cons as (a)**
  (c) Throw an exception **Handler code can be centralized, if not handled exception propagates, fits well in the object-oriented paradigm**

Lecture #4

Quick Review

- Why should error messages be printed to cerr instead of cout? **They will be mixed with data if sent to cout**
- When should assert be used instead of throwing an exception? **Use assert for interface errors, and exceptions for potentially recoverable errors.**
- Match the name to the associated type of testing
  - Interactive testing **(c)** (a) Write numerous automated unit tests
  - Regression testing **(a)** (b) Write the test before writing the code
  - Test-driven development **(b)** (c) Run the program like a "nightmare user"
- What are some common useful debugger capabilities? **Breakpoints and watchpoints, single step over or into method calls, view (and change) variable values and raw memory, view the assembly code**

Overview

- Declarations
  - Definitions
  - Headers and the preprocessor
  - Scope
- Functions
  - Declarations and definitions
  - Arguments
  - Call by value, reference, and **const** reference
- Namespaces
  - "Using" declarations
- Makefiles

Pre-C++ 11 Initialization

- Prior to C++ 11, 4 different initializers were used
  - int n=0; // but not for fields! Only initialization lists for those
  - int n(0); // equivalent to =, but parentheses were confusing
  - class1 c(“hello”, 3); // call a constructor
  - int m[2] = {0, 42}; // for aggregates only
// NOT classes or primitives
- Some initializations weren't possible w/o executing code
  - class C {int x[100]; public: C( ); /* can't directly initialize x */ };
  - char *buff=new char[1024]; // can't directly initialize the buffer
  - vector <string> v; // can't directly initialize the vector

C++ 11 Initialization

- With C++ 11, all initialization (not assignment) can be accomplished consistently with { }
  - int a{0};
  - string s{"hello"};
  - string s2{s}; //copy constructor – s2 contains a copy of s
  - vector <string> vs{"alpha", "beta", "gamma"};
  - map<string, string> stars

    { {"Superman", "+1 (212) 545-7890"},

    {"Batman", "+1 (212) 545-0987"}};
  - double *pd= new double[3] {0.5, 1.2, 12.99};
  - class C {int x[4]; C( ): x{0,1,2,3} { }};

In CSE1325, always use C++ 11 style initialization

C++ Declarations

- A declaration introduces a name into a scope.
- A declaration also specifies a type for the named object.
- Sometimes a declaration includes an initializer.
- **A name must be declared before it can be used.**

```
int a = 7;              // an int variable named ‘a’ is declared
const double cd = 8.7; // a double-precision floating-point constant
double sqrt(double);   // a function taking a double argument and
                       //  returning a double result
vector<Token> v;       // a vector containing Tokens
```

Declaration – A statement that introduces a name with an associated type into a scope

C++ Declarations

- Declarations are frequently introduced into a C++ (and C) program through “headers”
  - A header is a file containing declarations providing an **interface** to other parts of a program
- This allows for **abstraction** – you don’t have to know the details of a function you (or someone else) wrote in order to use it. When you add

  #include "my_header.h"

  to your code, the declarations in the file **my_header.h** in the local directory become available
  - **#include "my_header.h"** checks the local directory first for my_header.h, then the system directories
  - **#include <my_header.h>** checks only the system directories

For example

- Define your own functions and variables

```
#include <iostream>            // We find the declaration of cout in here
using namespace std;           // Explanation coming shortly!

int f(int x);                  // This DECLARES f(int) – you can declare it as often
                               // as you like, but you must declare it before calling it

int main() {
     int i{7};                 // This declares and defines i
     cout << f(i) << '\n';     // This INVOKES (calls) f(int)
}

int f(int x) {return x*x;}     // This DEFINES f(int) – you must define it exactly once
```

```
ricegf@pluto:~/dev/cpp/07$ g++ -w decl_correct.cpp
ricegf@pluto:~/dev/cpp/07$ ./a.out
49
```

C++ Definitions

Definition – A declaration that (also) *fully* specifies the entity declared

- Examples of definitions

```
int a = 7;                        // an (initialized) int
int b;                            // an (uninitialized) int
vector<double> v;                 // an empty vector of doubles
double sqrt(double) { … };        // a function with a body
struct Point { int x; int y; }; // a struct
```

- Examples of declarations that are <u>not</u> definitions

```
double sqrt(double);  // function body missing (define it later)
class Point;          // class members specified elsewhere
extern int a;         // extern means “defined in another file”
                      // “extern” is archaic; we will rarely use it
```

Memory is allocated when a variable is **defined**,
not when it is **declared**!

C++ Declarations and Definitions

- You can't *define* something twice
  - A definition says what something is
  - Examples

```
int a;        // definition
int a;        // error: double definition
double sqrt(double d) { … }   // definition
double sqrt(double d) { … }   // error: double definition
```

- You can *declare* something twice
  - A declaration says how something can be used
  - That is, its interface – perfect for a header file!

```
int a = 7;           // definition (also a declaration)
extern int a;             // declaration
double sqrt(double);      // declaration
double sqrt(double d) { … }   // definition (also a declaration)
```

Why both declarations and definitions?

- To refer to something, we need (only) its declaration
- Often we want the definition “elsewhere”
  - Later in a file
  - In another file
    - Preferably written by someone else
- Declarations are used to specify interfaces
  - To your own code
  - To libraries: Libraries are key!
    - We can’t write everything ourselves
    - We don't want to write everything ourselves!
- In larger programs
  - Place all declarations in header files to ease sharing and save compilation time

Key Point!

Kinds of Declarations

- The most interesting are
  - Variables
    - **int x;**
    - **vector<int> vi2 {1,2,3,4};**
  - Constants
    - **void f(const X&);**
    - **constexpr int = isqrt(2);**
  - Functions
    - **double sqrt(double d) { /*** *…* ***/ }**
  - Types (classes and enumerations)
    - **class Foo {public: Foo(int x); int get_x();}**
    - **enum Animal {cat, dog, squirrel, duck, snake};**
  - Namespaces (which we'll cover this lecture)
  - Templates (such as vector – we'll cover in a later lecture)

Classes

- A class encapsulates data and associated code
  - A class directly represents a concept in a program
    - If you can think of “it” as a separate entity, it is plausible that it could be a class or an object of a class
    - Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
  - A class is a (user-defined) type that specifies how objects of its type can be created and used
  - In C++ (as in most modern languages), a class is the key building block for large programs
    - And very useful for small ones, too!
  - The concept was originally introduced in Simula67
  - **The classes public methods and variables are it's *interface***

Class Interfaces

- What makes a good class interface?
  - Minimal
    - As small as possible
    - Simple and easy to use – or better, hard to **misuse**!
  - Complete
    - And no smaller
  - Type safe
    - Beware of confusing argument orders
    - Beware of over-general types (e.g., int to represent a month)

Header Files as Class Interface Definitions

- Until now, we have simply included (#include) the .cpp files that we need
- This is a Very Bad Idea
  - Your implementation is compiled as part of a different file
  - If a .cpp file is included twice, you'll get redefinition errors – making it difficult to determine a valid compilation order
- All the including file needs is the <u>interface</u> <u>definition</u>
  - The header file is that interface definition!

Source files

Interface / Declarations

complex.h:

```
class Complex {
    int _x, _y;
  public:
    Complex(int x, int y);
    double magnitude();
    …
};
…
```

Defines

Uses

complex.cpp:

```
#include "complex.h"
//definitions:
Complex::Complex(int x, int y)
       : _x{x}, _y{y} { }
double Complex::magnitude() {
  /* … */
}
…
```

use.cpp:

```
#include "complex.h"
…
Complex c{3.0, 4.0};
Cout <<
…
```

- A header file (here, **complex.h**) defines an interface between user code and implementation code (usually in a library)
- The same **#include** declarations in both **.cpp** files (definitions and uses) ease consistency checking

Yes, I *know* C++ has a <complex> template type in its library.
This is just a quick and simple demo!

Header Files

- An interface can be specified in a header file
  - The header file / interface

select_floor.h

Specify the function's interface, but not it's implementation

```
#include "elevator.h"

int select_floor(Elevator elevator, vector<int> request);
```

  - The cpp file / implementation

select_floor.cpp

```
#include "select_floor.h"
#include "elevator.h"

int select_floor(Elevator elevator, Vector<int> request) {
  int direction = elevator.is_going_up() ? 1 : -1;
  int floor = elevator.get_current_floor();
  for (int floors = 0; floors < max_floor; ++floors) {
    floor += direction;
    if (floor < 0 || floor > max_floor) {
      direction = -direction;
      floor = elevator.get_current_floor() + direction;
    }
    if (request[floor] == TRUE) {
      return floor;
    }
```

Include the associated header so the compiler can verify consistency

Repeating an include from the header isn't *required*, but is *allowed*

Class Header Files

- Class methods are specified w/o implementation

elevator.h

```
#ifndef __ELEVATOR_H
#define __ELEVATOR_H
class Elevator {
  public:
    Elevator() : top_floor{9} {}
    Elevator(int max_floors) : top_floor{max_floors} {}

    class Invalid_floor: global exception {}; // Exception
    void goto_floor(int floor);
    void move();

    int get_current_floor();
    int get_desired_floor();
    bool is_going_up();
    bool has_arrived();
    bool is_idle();

  private:
    int top_floor;
    int current_floor = 1;
    int desired_floor = 1;
    bool going_up = true;
    bool idle = false;
};
#endif
```

These statements ensure we don't redefine anything if the header file is included in more than one file*

The constructors – one default and one non-default

The methods declared here must be defined *somewhere* – typically in the associated implementation (.cpp) file

The private data for this class

Including a header is exactly equivalent to including the text of the header file into the file that includes it, at the point of the #include

* "#pragma once" is also often used for this purpose, but it is NOT part of the C++ standard

Class Implementation Files

- If methods are define in a header, the method bodies are implemented separately in the .cpp

elevator.cpp

```
#include "elevator.h"

void Elevator::goto_floor(int floor) {
   // Your code here
}

void Elevator::move() {
  // Your code here
}

int Elevator::get_current_floor() {
   // Your code here
}

int Elevator::get_desired_floor() {
   // Your code here
}

bool Elevator::is_going_up() {
   // Your code here
}

bool Elevator::has_arrived() {
```

Include the associated header so the compiler can verify consistency

Elevator::goto_floor means the method goto_floor in the class Elevator

This file is compiled separately into a .o file for linking (use g++ -c to avoid an error for a missing main())

Constructors

- A constructor is a special kind of member function that initializes an instance of its class
  - Initialize private variables, allocate memory, update static variables, etc.
- A constructor has the same name as the class and no return value
  - Constructors are usually the first methods in the public section
- A constructor can have any number of parameters
  - The default constructor has zero parameters
  - If no constructor is specified, the compiler defines the default constructor that just initializes the object based on the class definition
  - **If only non-default constructors are specified, the compiler will NOT provide a default constructor**
- A class may have any number of constructors
  - Each must have a unique parameter signature – the number of parameters, and the type of each

Initialization Lists

- C++ allows specification of private variable values as part of the constructor declaration
  - This avoids invoking the default constructor for the class' private variables, and then overwriting the default variable in the constructor body

```
class Elevator {
  public:
    Elevator() : top_floor{9} {}
    Elevator(int max_floors) : top_floor{max_floors} {}

  private:
    int top_floor;
};
```

Directly initialize top_floor to 9

Directly initialize top_floor to the parameter value

Note the curly braces (not parentheses)

Delegated Constructors

- Sometimes called "constructor chaining", a similar syntax permits one constructor to rely on another in the same class
  - This avoids code duplication

```
class Elevator {
  public:
    Elevator() : Elevator{9} {}
    Elevator(int max_floors) : top_floor{max_floors} {}

  private:
    int top_floor;
};
```

Delegate to the 1 int parameter constructor

Directly initialize top_floor to the parameter value

Static Class Members

- A static method or variable exists as part of the class, shared among all objects
  - A static method may be called without instancing an object
  - A static variable must be defined outside the class

```
class Elevator {
   public:
     Elevator();
     void move(int target_floor);  // Unique to each object
     static int get_floors_traversed(); // Declaration (at class level) only
   private:
     int current_floor;
     static int floors_traversed;  // declaration (at class level) only
};

Elevator::Elevator() : current_floor{1} { }
int Elevator::floors_traversed = 0;
void Elevator::move(int target_floor) {
    floors_traversed += abs(target_floor - current_floor);
    current_floor = target_floor;
}
int Elevator::get_floors_traversed() {return floors_traversed;}
```

Static Class Members

```
class Elevator {
   public:
     Elevator();
     void move(int target_floor);  // Unique to each object
     static int get_floors_traversed(); // Declaration (at class level) only
   private:
     int current_floor;
     static int floors_traversed;  // declaration (at class level) only
};

Elevator::Elevator() : current_floor{1} { }
int Elevator::floors_traversed = 0;
void Elevator::move(int target_floor) {
    floors_traversed += abs(target_floor - current_floor);
    current_floor = target_floor;
}
int Elevator::get_floors_traversed() {return floors_traversed;}

int main() {
    Elevator a, b, c;  // 3 elevators
    vector<int> requests = {3,2,9,1,11,5,1,1,1};  // Must be multiple of 3!
    for (int i; i < requests.size() ; i += 3) {
        a.move(requests[i]);
        b.move(requests[i+1]);
        c.move(requests[i+2]);
    }
    cout << Elevator::get_floors_traversed() << " floors traversed" << endl;
}
```

```
ricegf@pluto:~/dev/cpp/201801/05$ make static
g++ --std=c++14 -o static static.cpp
ricegf@pluto:~/dev/cpp/201801/05$ ./static
40 floors traversed
```

Scope

- A **scope** is a region of program text
  - Global scope (outside any language construct)
  - Class scope (within a class)
  - Local scope (between any { … } braces)
  - Statement scope (unique to the 3-term for statement)

- A name in a scope can be seen from within its scope and within scopes nested within that scope
  - Only after the declaration of the name (“can’t look ahead” rule)
  - However, class members can be used *within the class* before they are declared
- A scope keeps “things” local
  - Prevents my variables, functions, etc., from interfering with yours
  - Remember: real programs have **many** thousands of entities
  - Locality is good!

Keep names as local as possible

Scope Example

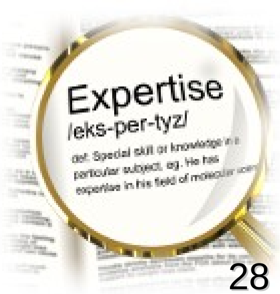
```
// no r, i, or v here

class My_vector {
    vector<int> v;                        // v is in class scope
public:
    int largest() {                       // largest is in class scope
        int r = 0;                        // r is local
        for (int i = 0; i<v.size(); ++i)   // i is in statement scope
            r = max(r,abs(v[i]));
        // no i here
        return r;
    }
    // no r here
};
// no v here
```

Nested Scopes

```
int x;   // global variable – avoid those where you can
int y;   // another global variable

int f()
{
    int x;       // local variable (Note – now there are two x's)
    x = 7;       // local x, not the global x
    {
        int x = y;  // another local x, initialized by the global y
                    // (Now there are three x's)
        ++x;    // increment the local x in this scope
    }
}

// avoid such complicated nesting and hiding: keep it simple!
```

The code attached to these slides prints the various x values throughout the program

Shadowing – A variable declared in a narrower scope than that of a variable of the same name declared in a broader scope



Functions: Call by Value

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

// call-by-value (send the function a copy of the argument’s value)
int f(int a) { a = a+1; return a; }


int main() {
     int xx = 0;
     cout << f(xx) << '\n';   // writes 1
     cout << xx << '\n';      // writes 0; f() doesn’t change xx
     int yy = 7;
     cout << f(yy) << '\n';   // writes 8;
     cout << yy << '\n';      // writes 7; f() doesn’t change yy
}
```

```
ricegf@pluto:~/dev/cpp/201701/05$ g++ -std=c++11 call_by_value.cpp
ricegf@pluto:~/dev/cpp/201701/05$ ./a.out
1
0
8
7
ricegf@pluto:~/dev/cpp/201701/05$
```

Functions: Call by Reference

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

// call-by-value (send the function a reference to the argument’s value)
int f(int& a) { a = a+1; return a; }


int main() {
     int xx = 0;
     cout << f(xx) << '\n';   // writes 1
     cout << xx << '\n';      // writes 0; f() changes xx
     int yy = 7;
     cout << f(yy) << '\n';   // writes 8;
     cout << yy << '\n';      // writes 7; f() changes yy
}
```

```
ricegf@pluto:~/dev/cpp/201701/05$ g++ -std=c++11 call_by_reference.cpp
ricegf@pluto:~/dev/cpp/201701/05$ ./a.out
1
1
8
8
ricegf@pluto:~/dev/cpp/201701/05$
```

Call by value/by reference/ by const-reference

```
void f(int a, int& r, const int& cr) { ++a; ++r; ++cr; } // error: cr is const
void g(int a, int& r, const int& cr) { ++a; ++r; int x = cr; ++x; } // ok

int main()
{
    int x = 0;
    int y = 0;
    int z = 0;
    g(x,y,z);       // x==0; y==1; z==0
    g(1,2,3);       // error: reference argument r needs a variable to refer to
    g(1,y,3);       // ok: since cr is const we can pass "a temporary"
}
// const references are very useful for passing large objects
```

Function / Method Arguments

- Avoid (non-const) reference arguments if possible
  - They can lead to obscure bugs when you forget which arguments can be changed

```
int incr1(int a) { return a+1; }
void incr2(int& a) { ++a; }
int x = 7;
x = incr1(x);     // pretty obvious
incr2(x);         // pretty obscure
```

- So why have reference arguments?
  - Occasionally, they are essential
    - *E.g.*, for changing several values
    - For manipulating large containers (*e.g.*, vector)
  - **const** reference arguments are very often useful

References

- "Reference" is a general concept
  - Not just for call-by-reference

```
int i = 7;
int& r = i;           // r and i point to the same memory address!
r = 9;                // i also becomes 9
const int& cr = i;
// cr = 7;            // error: cr refers to const
i = 8;
cout << cr << endl;  // write out the value of i (that's now 8)
```

- You can think of a reference as an alternative name for an object
- You can't modify an object through a const reference
- You can't make a reference refer to another object after the reference variable is initialized (unlike pointers)
  - Thus, references are never "dangling"

Copy vs Reference

- We said earlier that a For Each loop can be very inefficient for vectors of large objects
  - Because it copies each object to the loop variable
  - Now we see how to access very large vector members efficiently – using references!

```
// Range-for (for each) loops
for (string s : v) cout << s << "\n";          // s is a copy of some v[i]
for (string& s : v) cout << s << "\n";         // s is literally v[i]
for (const string& s : v) cout << s << "\n";   // s is literally v[i] BUT
                                               //   we can’t modify v
```

Compile-time functions

- You can define functions that *can be* evaluated at compile time: **constexpr** functions
  - A definition usually results in memory allocation
  - A constexpr does not – the compiler simply substitutes the value of the expression as a constant wherever referenced

```
constexpr double xscale = 10;    // scaling factors
constexpr double yscale = .8;

constexpr Point scale(Point p) { return {xscale*p.x, yscale*p.y}; };

constexpr Point x = scale({123,456});  // evaluated at compile time

void use(Point p) {
    constexpr Point x1 = scale(p);   // error: compile-time evaluation
                                     // requested for a variable argument
    Point x2 = scale(p);             // OK: run-time evaluation
}
```

Guidance for Passing Variables

- Use call-by-value for very small objects
- Use call-by-const-reference for large objects
- Use call-by-reference only when *absolutely necessary*
- Return a result rather than modify an object through a reference argument if practical – experience will help
- For example

```
class Image { /* objects are potentially huge */ };
void f(Image i);  … f(my_image);      // oops: this could be s-l-o-o-o-w
void f(Image& i); … f(my_image);      // no copy, but f() can modify my_image
void f(const Image&); … f(my_image);  // f() won't mess with my_image
Image make_image();                   // most likely fast!
                                      //   ("move semantics" – later)
```

Namespaces

- Consider this code from two programmers Jack and Jill

jack.h

```
class Glob { /*…*/ };       // in Jack’s header file jack.h
class Widget { /*…*/ };
```

jill.h

```
class Blob { /*…*/ };       // in Jill’s header file  jill.h
class Widget { /*…*/ };
```

awesome_app.h

```
#include "jack.h";          // this is in your code
#include "jill.h";

void my_func(Widget p) {   // oops! – error: multiple definitions of Widget
     // …
}
```

Namespaces

- The compiler will not compile multiple definitions; such clashes can occur from multiple headers.
- One way to prevent this problem is with namespaces:

jack.h

```
namespace Jack {
  class Glob { /*…*/ };       // in Jack’s header file jack.h
  class Widget { /*…*/ };
}
```

jill.h

```
Namespace Jill {
  class Blob { /*…*/ };       // in Jill’s header file  jill.h
  class Widget { /*…*/ };
}
```

awesome_app.h

```
#include "jack.h";         // this is in your code
#include "jill.h";

void my_func(Jack::Widget p) {   // No collision!
     // …
}
```

`::` means *membership* – Jack’s Widget!

Namespaces

- A namespace is simply a named scope
- The :: operator is used to specify which namespace you are using and which (of many possible) objects of the same name to which you are referring
- For example, since **cout** is in namespace **std**, you could write:

```
std::cout << "Please enter stuff… \n";
```

- In fact, you would have to if you didn't specify that you are “using namespace std;”!

Namespace – a named scope

using Declarations and Directives

- To avoid the tedium of

```
std::cout << "Please enter stuff… \n";
```

you could write a "using declaration"

```
using std::cout;                     // "when I say cout, I mean std::cout"
cout << "Please enter stuff… \n";   // std::cout is also fine
cin >> x;                            // error: cin not in scope
```

- or you could write a "using directive"

```
using namespace std;                 // "make all names from std available"
cout << "Please enter stuff… \n";   // std::cout is also fine
cin >> x;                            // std::cin is also fine
```

Using *Using* Well

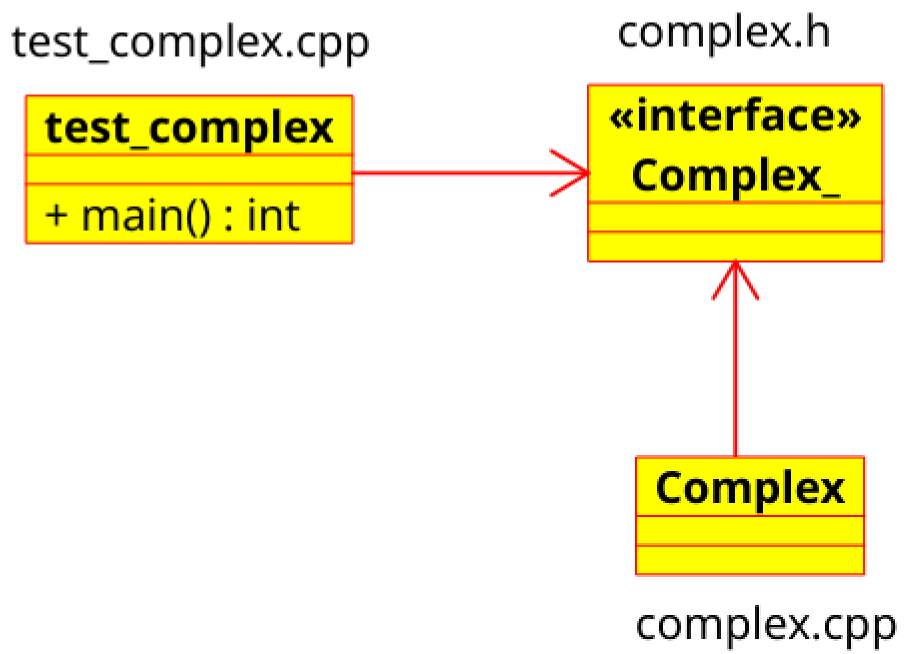
- “using namespace std;” puts EVERYTHING from std into our current namespace
  - This is a <u>lot</u> of declarations, which may collide with our own declarations accidentally (remember std::to_string for double?)
  - This is sometimes called “namespace pollution”
- You can “using namespace” for multiple namespaces, but that increases the chances for collisions
  - This is (ahem) namespace mega-pollution!*
- But it's better to “using namespace::member” rather than the entire namespace
  - Explicit *using* places only the specific members you want into your namespace, and also makes clear to which namespace member belongs
  - But you're still permitted to “using namespace std;” as you like

* Yes, I just made that one up

Writing a New Makefile

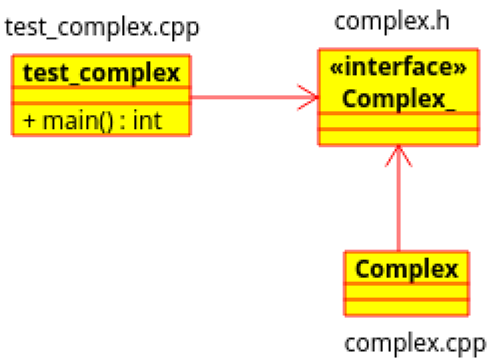
http://www.cprogramming.com/tutorial/makefiles.html

- Write a Makefile for this program



Makefiles, Step by Step

#1: Define your C++ version

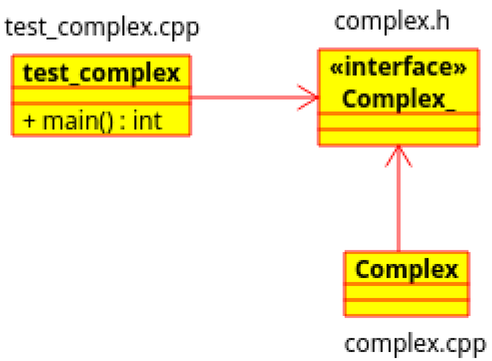



```
# Makefile for Complex
CXXFLAGS += --std=c++14
```

The CXX variable is usually pre-defined as your default compiler (“g++”)
The CXXFLAGS variable contains flags for all invocations of $(CXX)

Makefiles, Step by Step

#2: Define your link rule



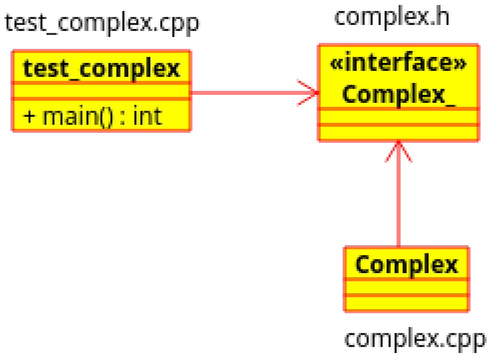

```
# Makefile for Complex
CXXFLAGS += --std=c++14

complex: test_complex.o complex.o
        $(CXX) $(CXXFLAGS) -o complex test_complex.o complex.o
```

Usually the rule name matches the executable name that you’re building.
The dependencies are also the object files on the linker line.

$(CXX) simply recalls (“dereferences”) the value of the CXX variable

Makefiles, Step by Step

#3: Define your object rules

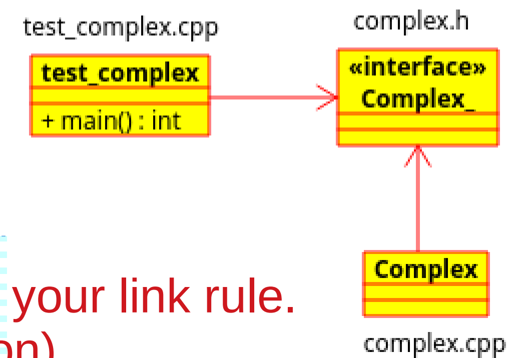



```
# Makefile for Complex
CXXFLAGS += --std=c++14

complex: test_complex.o complex.o
        $(CXX) $(CXXFLAGS) -o complex test_complex.o complex.o

test_complex.o: test_complex.cpp *.h
        $(CXX) $(CXXFLAGS) -c test_complex.cpp
complex.o: complex.cpp *.h
        $(CXX) $(CXXFLAGS) -c complex.cpp
```

You need one rule, named for a .o file, for each .cpp file.
The .o dependency is the .cpp file of the same name, and all headers (*.h)
The command simply compiles the .cpp file with the -c option

Makefiles, Step by Step

#4: Define the clean, debug, rebuild, and other miscellaneous rules




```
# Makefile for Complex
CXXFLAGS += --std=c++14

all: complex

debug: CXXFLAGS += -g
debug: complex

rebuild: clean complex

complex: test_complex.o complex.o
	$(CXX) $(CXXFLAGS) -o complex test_complex.o complex.o

test_complex.o: test_complex.cpp *.h
	$(CXX) $(CXXFLAGS) -c test_complex.cpp
complex.o: complex.cpp *.h
	$(CXX) $(CXXFLAGS) -c complex.cpp

clean:
	-rm -f *.o *.gch *~ complex
```

The first rule will be default, so jump to your link rule.
Debug builds for the debugger (-g option).
Rebuild is a convenience rule to recompile everything.
Clean deletes all build products to force a full rebuild.

- *.o are built from *.cpp; *.gch are from *.h (sometimes)
- *~ are backup files produced by some editors (Emacs)
- Also include the executable name(s), e.g., complex

Maintaining a Makefile

- Adding an #include "..." or .h file
  - No change as long as your rules include a *.h dependency
- Adding a .cpp
  - Add a new .o rule to compile it
  - Add the .o to the linker rule dependency and linker command
- Adding an executable, e.g., regression test
  - Add a completely new linker rule to link it
  - Add the executable name to the rm command in the clean rule
- Remember: Your Makefile must be managed in git!
  - It evolves along with your program code

Quick Review

- A statement that introduces a name with an associated type into a scope is a ______.

  A statement that also fully specifies that entity is a ________.
- Why should programs #include header files instead of cpp files?
- True or False: Memory is allocated for a variable when it is declared.
- True or False: It is an error to include the same file in both the header and the body of a class file.
- What is the purpose of “#ifndef __FILE_H” and “#define __FILE_H” at the top of a header file?

Quick Review

- A special kind of member function that initializes an instance of its class is a __________.
  ______ _______ is a shortcut to assign a value to a class's private variables in a constructor.
  _______ _______ implements one constructor by calling another.
- True or False: A default constructor with no parameters is always provided.
- To modify a parameter, the parameter must be passed by ________ - otherwise, the method only has a copy of the parameter.
  A _____ reference cannot modify the parameter even though it references to the original data.
- Constexpr specifies that a variable can be evaluated at _______ ______.
- A ________ is a named scope, which can be accessed via the _______ keyword or the ____ operator.
- A ________ builds a C++ program optimally with a simple “make” command.

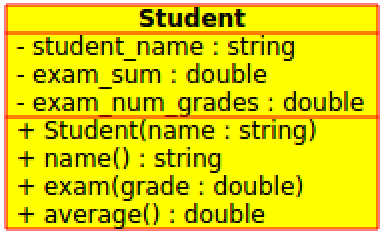
For Next Class

- (Optional) Review chapter 8 (some of today's material is found there)
  - Do the drills!
- Skim chapter 9
  - Enum classes
  - Operator overloading
  - Simple inheritance

Homework #2 Questions?

- Create an OO-based program that accepts a student’s name and exam grades, and outputs their semester average.
  - **Bonus:** Also include homework grades in their semester average.
  - **Extreme Bonus:** Read the data from a file, and write the same file in the same format but with additional information added.
- As always, details are on Blackboard.
  - Requirements are in a ZIP archive along with test data

Due Thursday at 8 am!




```
ricegf@pluto:~                        $ make clean
rm -f *.o main test_student
ricegf@pluto:~/dev/cpp/201708/P2/fc$ make
g++ -std=c++11 -o main main.cpp
./main
Enter student's name: Fred
Enter next grade: 100
Enter next grade: 87
Enter next grade: 98
Enter next grade: 93
Enter next grade: 97
Enter next grade: -1
Fred has a 95 average.
ricegf@pluto:~/dev/cpp/201708/P2/fc$ ./main
Enter student's name: Ruth
Enter next grade: -1
Ruth has a 100 average.
ricegf@pluto:~/dev/cpp/201708/P2/fc$
```

“Now is the time for all good persons to come to the aid of their final average.”
– Patrick Henry*

Do the bonus and extreme bonus!

*OK, maybe it was Charles E. Weller. And he didn’t say “final average”. Or “persons”. Or possibly anything like this at all.