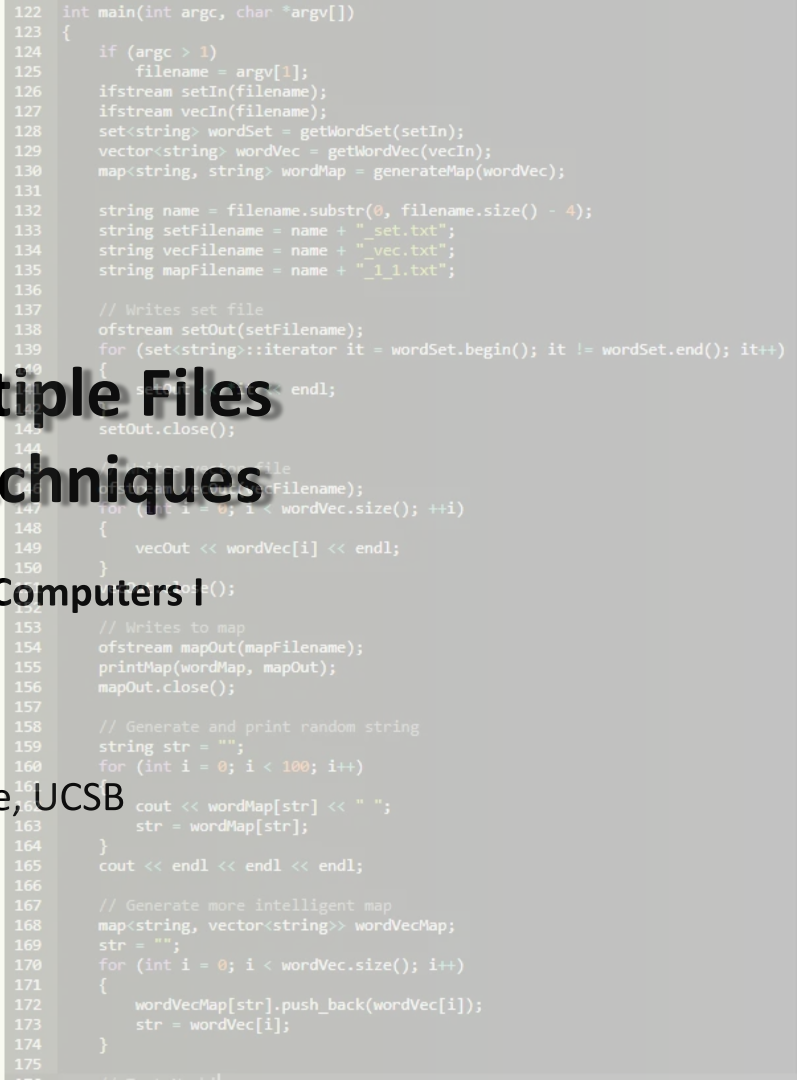
Compiling with Multiple Files and Using Debug Techniques

CS 16: Solving Problems with Computers I
Lecture #8

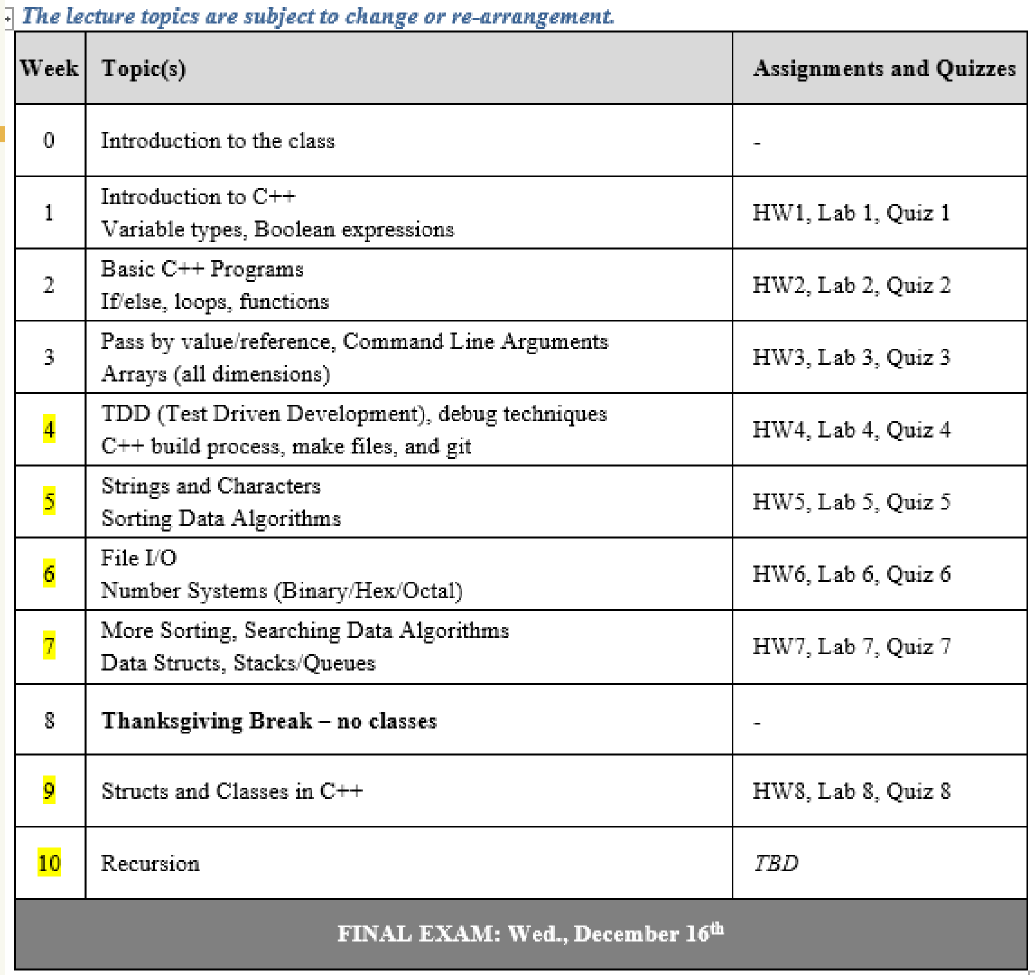
Ziad Matni
Dept. of Computer Science, UCSB



```
int main(int argc, char *argv[])
{
    if (argc > 1)
        filename = argv[1];
    ifstream setIn(filename);
    ifstream vecIn(filename);
    set<string> wordSet = getWordSet(setIn);
    vector<string> wordVec = getWordVec(vecIn);
    map<string, string> wordMap = generateMap(wordVec);

    string name = filename.substr(0, filename.size() - 4);
    string setFilename = name + "_set.txt";
    string vecFilename = name + "_vec.txt";
    string mapFilename = name + "_1_1.txt";

    // Writes set file
    ofstream setOut(setFilename);
    for (set<string>::iterator it = wordSet.begin(); it != wordSet.end(); it++)
    {
        setOut << *it << endl;
    }
    setOut.close();

    // Writes vector file
    ofstream vecOut(vecFilename);
    for (int i = 0; i < wordVec.size(); ++i)
    {
        vecOut << wordVec[i] << endl;
    }
    vecOut.close();

    // Writes to map
    ofstream mapOut(mapFilename);
    printMap(wordMap, mapOut);
    mapOut.close();

    // Generate and print random string
    string str = "";
    for (int i = 0; i < 100; i++)
    {
        cout << wordMap[str] << " ";
        str = wordMap[str];
    }
    cout << endl << endl << endl;

    // Generate more intelligent map
    map<string, vector<string>> wordVecMap;
    str = "";
    for (int i = 0; i < wordVec.size(); i++)
    {
        wordVecMap[str].push_back(wordVec[i]);
        str = wordVec[i];
    }
```

Administrative

- Class schedule topics have been shifted around a bit
  - New version of the syllabus available



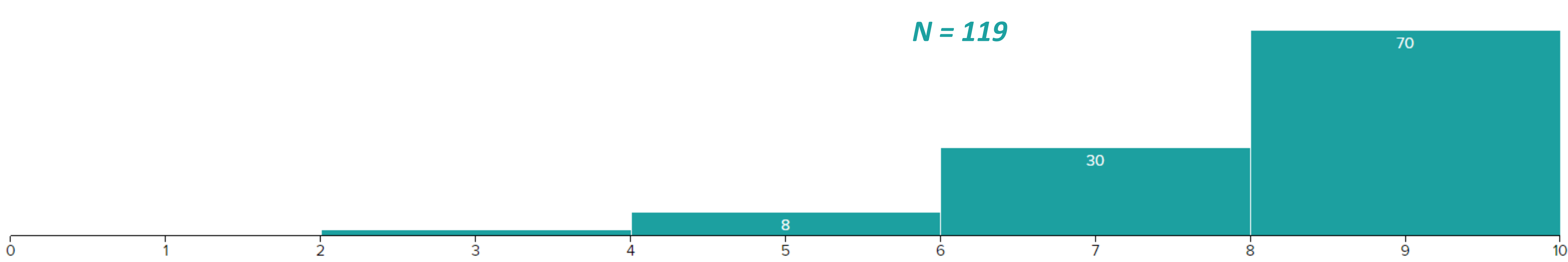
The lecture topics are subject to change or re-arrangement.

Week	Topic(s)	Assignments and Quizzes
0	Introduction to the class	-
1	Introduction to C++ Variable types, Boolean expressions	HW1, Lab 1, Quiz 1
2	Basic C++ Programs If/else, loops, functions	HW2, Lab 2, Quiz 2
3	Pass by value/reference, Command Line Arguments Arrays (all dimensions)	HW3, Lab 3, Quiz 3
4	TDD (Test Driven Development), debug techniques C++ build process, make files, and git	HW4, Lab 4, Quiz 4
5	Strings and Characters Sorting Data Algorithms	HW5, Lab 5, Quiz 5
6	File I/O Number Systems (Binary/Hex/Octal)	HW6, Lab 6, Quiz 6
7	More Sorting, Searching Data Algorithms Data Structs, Stacks/Queues	HW7, Lab 7, Quiz 7
8	**Thanksgiving Break – no classes**	-
9	Structs and Classes in C++	HW8, Lab 8, Quiz 8
10	Recursion	*TBD*
FINAL EXAM: Wed., December 16th		

Administrative

- Lab #3 and beyond
  - We will be scrutinizing copying/plagiarism VERY stringently
- New lab (#4) and new homework (#4) are out!
- Homework 3 and Lab 3 were due yesterday
- Quiz 4 is on Friday

Quiz 3

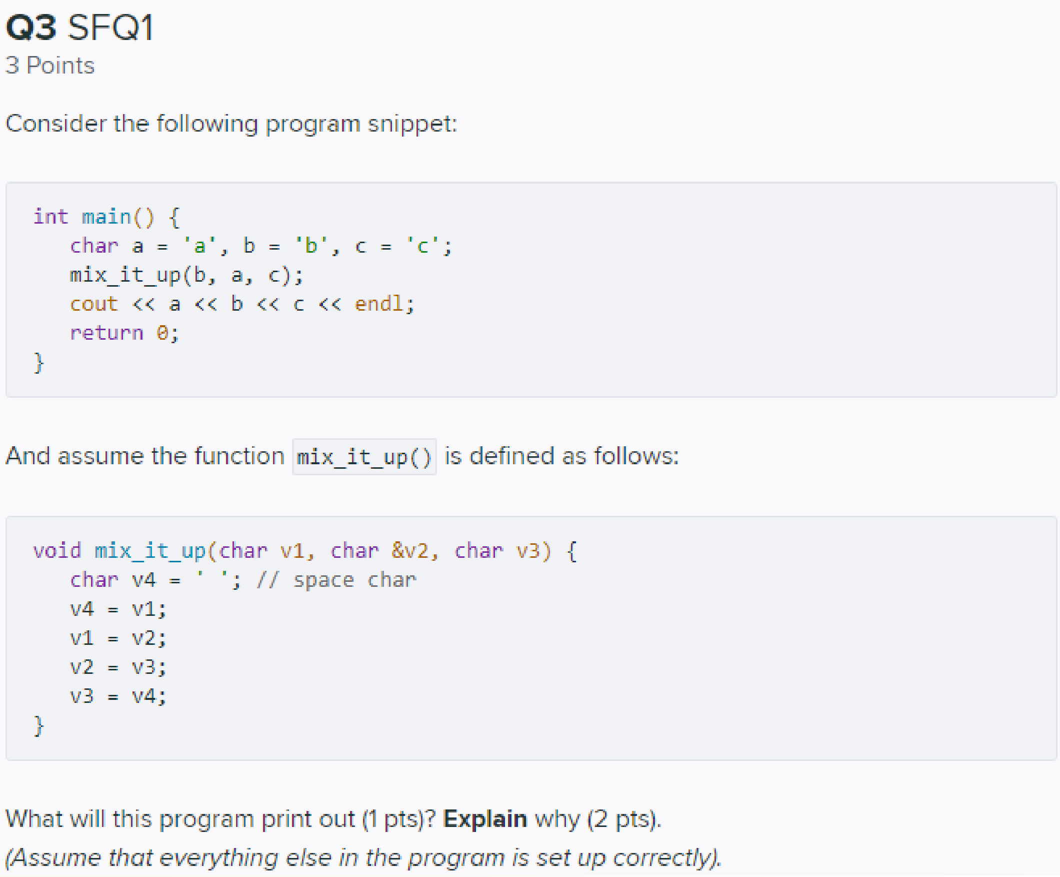


- Mean: **8.25/10**
- Median: **9/10**

- Note that ONLY v2 is passed by reference into the funct.
  - What does that mean??

- We call the function with:
  - **b** for v1 (by value)
  - **a** for v2 (by reference)
    - *a is the only var that changes OUTSIDE of the function!!*
    - *Gets swapped out for v3 = 'c'*
  - **c** for v3 (by value)

- We print a << b << c
  - Result is "**c**bc"



Q3 SFQ1

3 Points

Consider the following program snippet:

```
int main() {
   char a = 'a', b = 'b', c = 'c';
   mix_it_up(b, a, c);
   cout << a << b << c << endl;
   return 0;
}
```

And assume the function `mix_it_up()` is defined as follows:

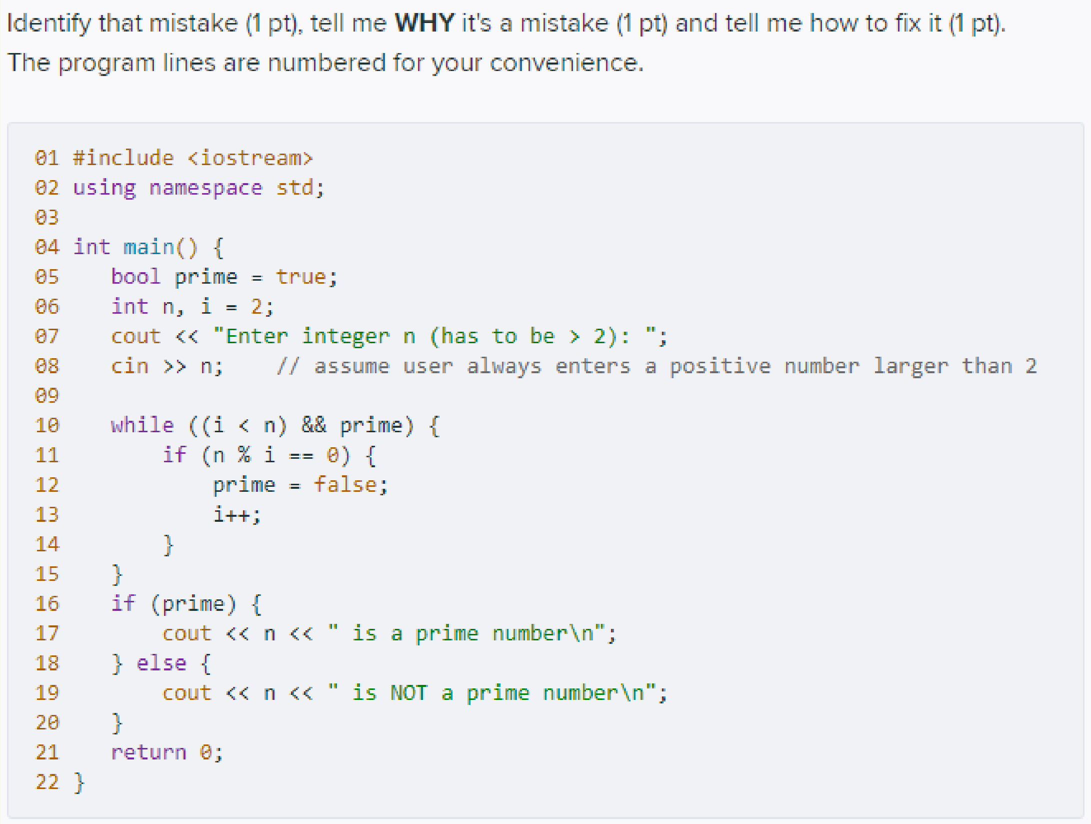
```
void mix_it_up(char v1, char &v2, char v3) {
   char v4 = ' '; // space char
   v4 = v1;
   v1 = v2;
   v2 = v3;
   v3 = v4;
}
```

What will this program print out (1 pts)? **Explain** why (2 pts).

(Assume that everything else in the program is set up correctly).

- The way it is written, line 13 causes variable **i** to iterate indefinitely
  - Regardless of what starting **n** value is…
  - So we get an **infinite loop**

- Simply **moving line 13 to outside the if-block, but still within the while-block works**
  - i.e. swap lines 13 and 14



Identify that mistake (1 pt), tell me **WHY** it's a mistake (1 pt) and tell me how to fix it (1 pt).

The program lines are numbered for your convenience.

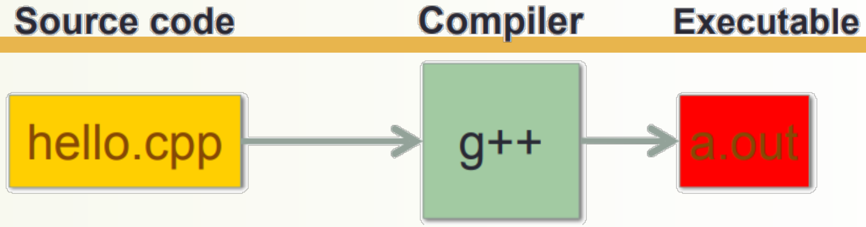
```
01 #include <iostream>
02 using namespace std;
03
04 int main() {
05    bool prime = true;
06    int n, i = 2;
07    cout << "Enter integer n (has to be > 2): ";
08    cin >> n;    // assume user always enters a positive number larger than 2
09
10    while ((i < n) && prime) {
11        if (n % i == 0) {
12            prime = false;
13            i++;
14        }
15    }
16    if (prime) {
17        cout << n << " is a prime number\n";
18    } else {
19        cout << n << " is NOT a prime number\n";
20    }
21    return 0;
22 }
```

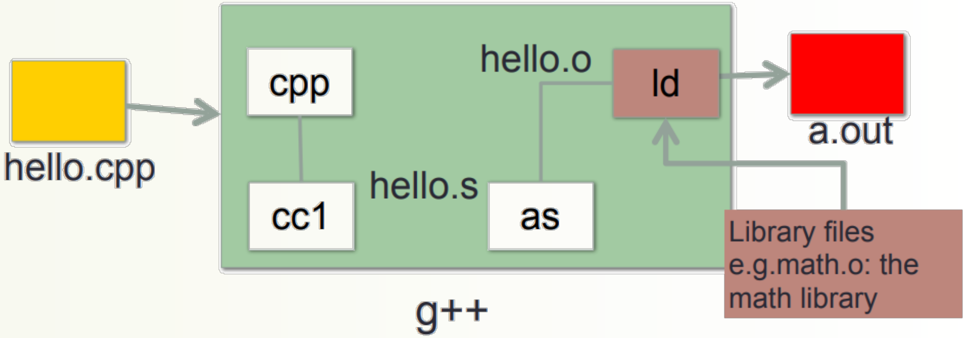
Lecture Outline

- The Magic of Makefiles!
- Programming in Multiple Files
- Design and Debug Tips
  - Designing and Debugging Loops
  - The Mighty TRACE
  - Designing and Debugging Functions

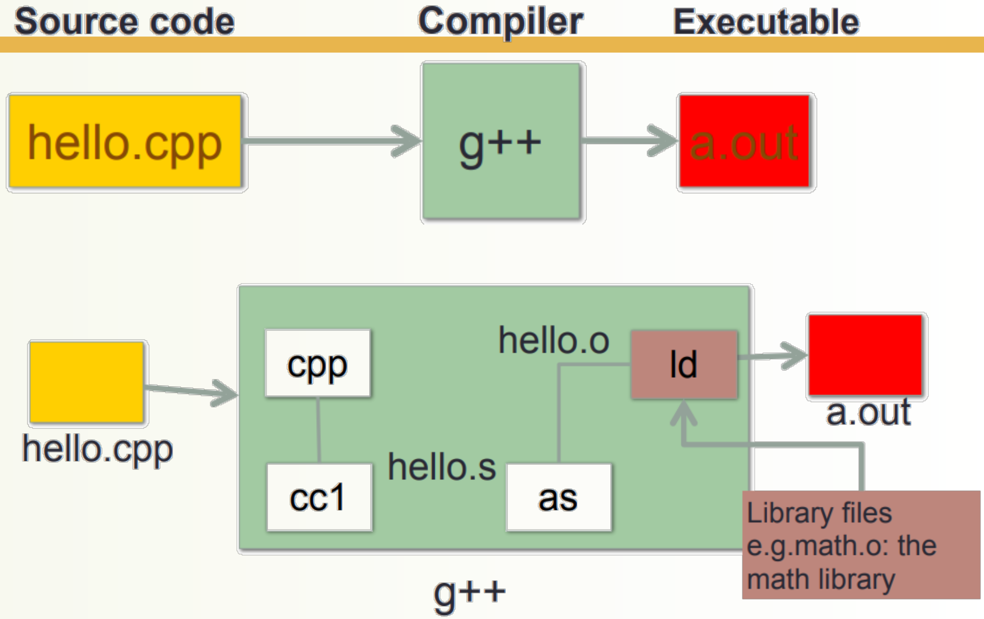
The Compilation Process (g++)

- **g++** is composed of a number of smaller programs
- Code written by others (libraries) can be included
- ld (linkage editor) merges one or more object files with the relevant libraries to produce a **single executable**





The Compilation Process (g++)



You can actually view all these "in-between" files/pieces:

- **g++ –S hello.cpp**
  - Produces hello.s (assembly code)
- **$ g++ –c hello.cpp**
  - Produces hello.o (object code)
- **$ g++ –o hello hello.cpp**
  - Produces hello (exec object code)

Make

- "Make" is a *build automation tool*
  - Automatically builds executable programs and libraries from source code
  - The instructions for **make** is in a file called ***Makefile***.

- Makefile is code written in OS-friendly code
  - Linux OS, to be precise…

Makefile

- The file must be called “makefile” (or “Makefile”)
- Put all the instructions you’re going to use in there
  - There is a syntax to follow for makefiles
  - Just type “**`make`**” at the prompt, instead of all the g++ commands
- Makefiles can easily be used to do other OS-related stuff
  - Like “clean up” your area, for example

Syntax of a Make

```
all: Exercise1 Exercise2

# This one forces the compile to use version 11 rules
# Also shows me all warnings (not just errors)
Exercise1: ex1.cpp
        g++ ex1.cpp –o ex1 -std=c++11 –Wall

# This next one’s a doozy
Exercise2: ex2.cpp
        g++ ex2.cpp –o ex2
clean:
        rm *.o ex2 ex1
```

Note: These are TAB characters used for the indents!
Don’t use spaces!!

Syntax of a Make

Target “all” programs in this project

Dependencies (macros) that are declared below

```
all: Exercise1 Exercise2

# This one forces the compile to use version 11 rules
# Also shows me all warnings (not just errors)
Exercise1: ex1.cpp
	g++ ex1.cpp –o ex1 -std=c++11 –Wall

# This next one’s a doozy
Exercise2: ex2.cpp
	g++ ex2.cpp –o ex2
clean:
	rm *.o ex2 ex1
```

Dependency files

# is for commenting

Make doesn’t have to have <u>compiling</u> instructions only!

What Should YOU do with Makefiles???

- Learn how to use them; their syntax
  - You can be quizzed about them

- Get into the habit of using them to compile your projects
  - I will always tell you how to do that in lab descriptions

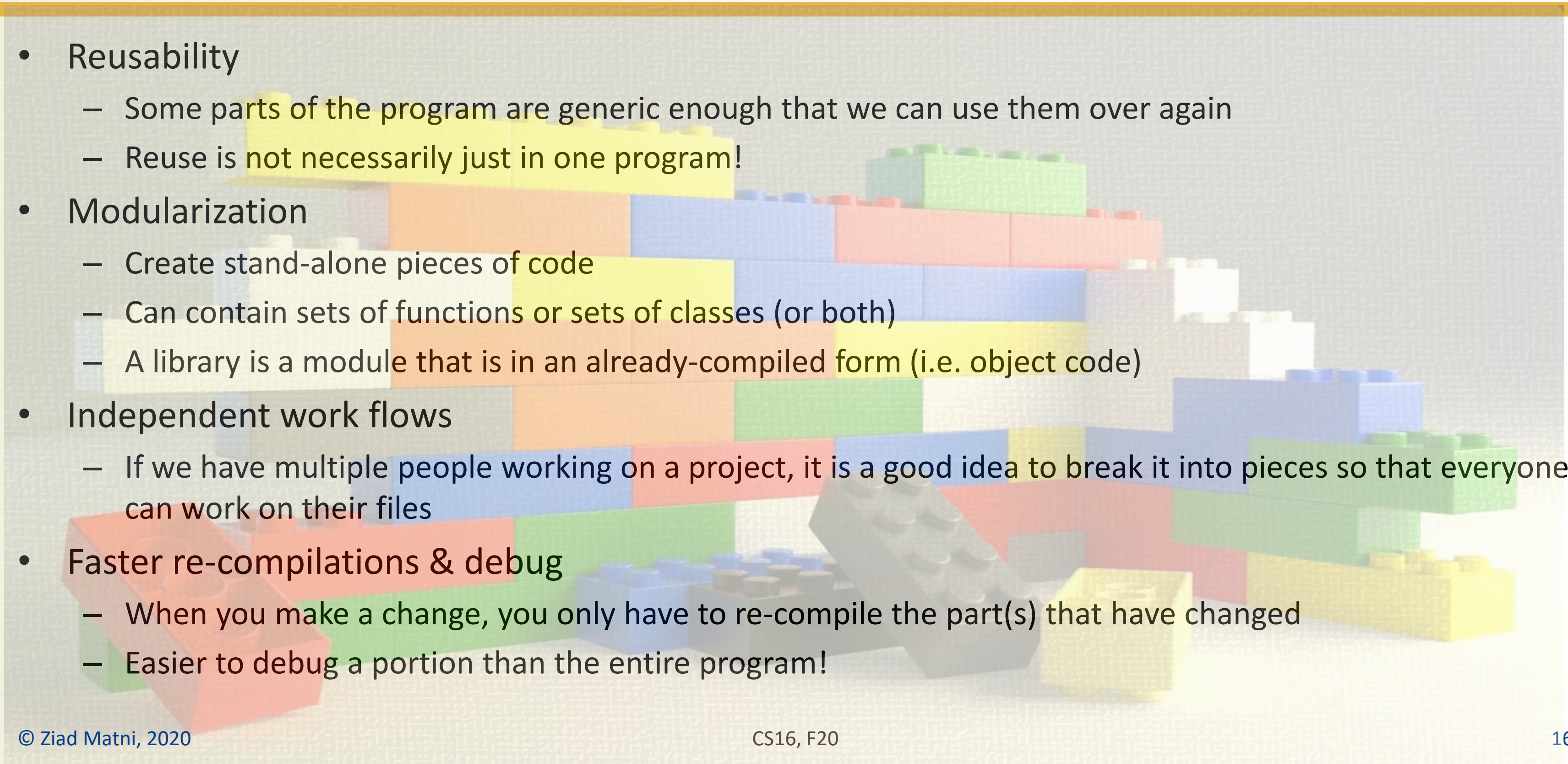
- You don’t have to turn them in with labs, **unless told otherwise**

C++ Programming in Multiple Files

- Novice C++ Programming:
  - All in one .cpp source code file
  - All the function definitions, plus the main( ) program

- Actual C++ Programming separates parts
  - There are usually one or more ***header files*** with file names ending in **.h** that typically contain function prototypes
  - There are one or more files that contain function definitions, some with **main( )** functions, and others that don't contain a **main( )** function

Why?



- Reusability
  - Some parts of the program are generic enough that we can use them over again
  - Reuse is not necessarily just in one program!
- Modularization
  - Create stand-alone pieces of code
  - Can contain sets of functions or sets of classes (or both)
  - A library is a module that is in an already-compiled form (i.e. object code)
- Independent work flows
  - If we have multiple people working on a project, it is a good idea to break it into pieces so that everyone can work on their files
- Faster re-compilations & debug
  - When you make a change, you only have to re-compile the part(s) that have changed
  - Easier to debug a portion than the entire program!

```
#include <iostream>
#include <etc…>

float linearScale(...);
float quadraticScale(...);
void printBellCurve(...);

int main()
{
    ...
}

float linearScale(...){ ... }
float quadraticScale(...) { ... }
float printBellCurve(...) { ... }
```

```
// File: MyFunctions.h
#include <iostream>
#include <etc…>
float linearScale(...);
float quadraticScale(...);
float printBellCurve(...);
```

```
// File: MyFunctions.cpp
#include “MyFunctions.h”
float linearScale(...){ ... }
float quadraticScale(...) { ... }
float printBellCurve(...) { ... }
```

```
// File: main.cpp
#include “MyFunctions.cpp”

int main()
{
    ...
}
```

Compiling Everything…

In this Example, in 3 steps…

```
g++ -c MyFunctions.cpp –o Myfunctions.o
```

(creates MyFunctions.o)

```
g++ -c main.cpp –o main.o
```

(creates main.o)

*The –c option creates object code – this is machine language code, but it's not the entire program yet… The target object file here is always generated as a **.o** type*

```
g++ -o ProgX main.o MyFunctions.o
```

(creates ProgX)

The –o option creates object code – in this case, it's object code created from other object code. The result is the entire program in executable form. The object file here is always generated with the name specified after the –o option.

What Do You End Up With?

MyFunctions.h	Header file w/ function prototypes
MyFunctions.cpp	C++ file w/ function definitions
MyFunctions.o	Object file of MyFunctions.cpp
main.cpp	C++ file w/ main function
main.o	Object file of main.cpp
ProgX	“Final” executable file

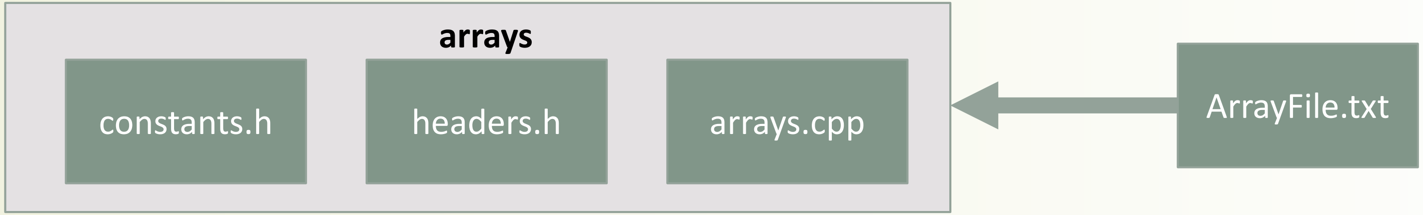
…and this is a simple example!!…

Wouldn’t it be nice to have code that generates/controls this?

Lab 04

You are given multiple files:

• constants.h	GLOBAL VARS
• headers.h	HEADERS
• skeleton_arrays.cpp → arrays.cpp	MAIN FILE
• ArrayFile.txt	EXTERNAL DATA FILE

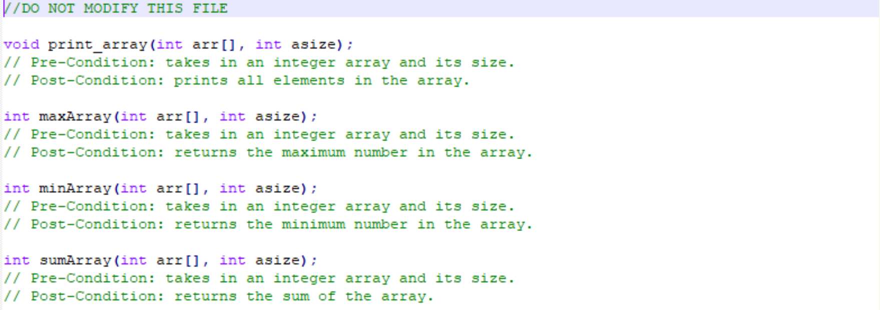

arrays

constants.h

headers.h

arrays.cpp

ArrayFile.txt



```
//DO NOT MODIFY THIS FILE

void print_array(int arr[], int asize);
// Pre-Condition: takes in an integer array and its size.
// Post-Condition: prints all elements in the array.

int maxArray(int arr[], int asize);
// Pre-Condition: takes in an integer array and its size.
// Post-Condition: returns the maximum number in the array.

int minArray(int arr[], int asize);
// Pre-Condition: takes in an integer array and its size.
// Post-Condition: returns the minimum number in the array.

int sumArray(int arr[], int asize);
// Pre-Condition: takes in an integer array and its size.
// Post-Condition: returns the sum of the array.
```

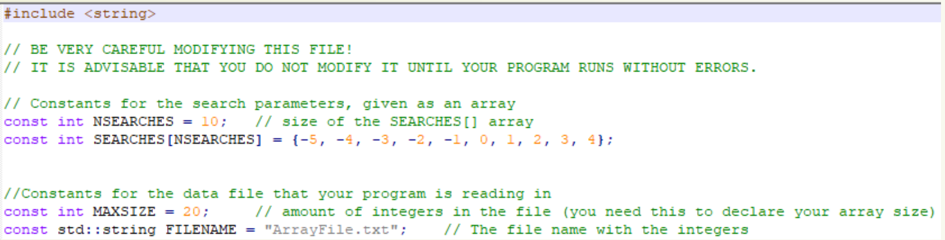
headers.h

Note the syntax difference between:

#include <xxx> // for C std. libs

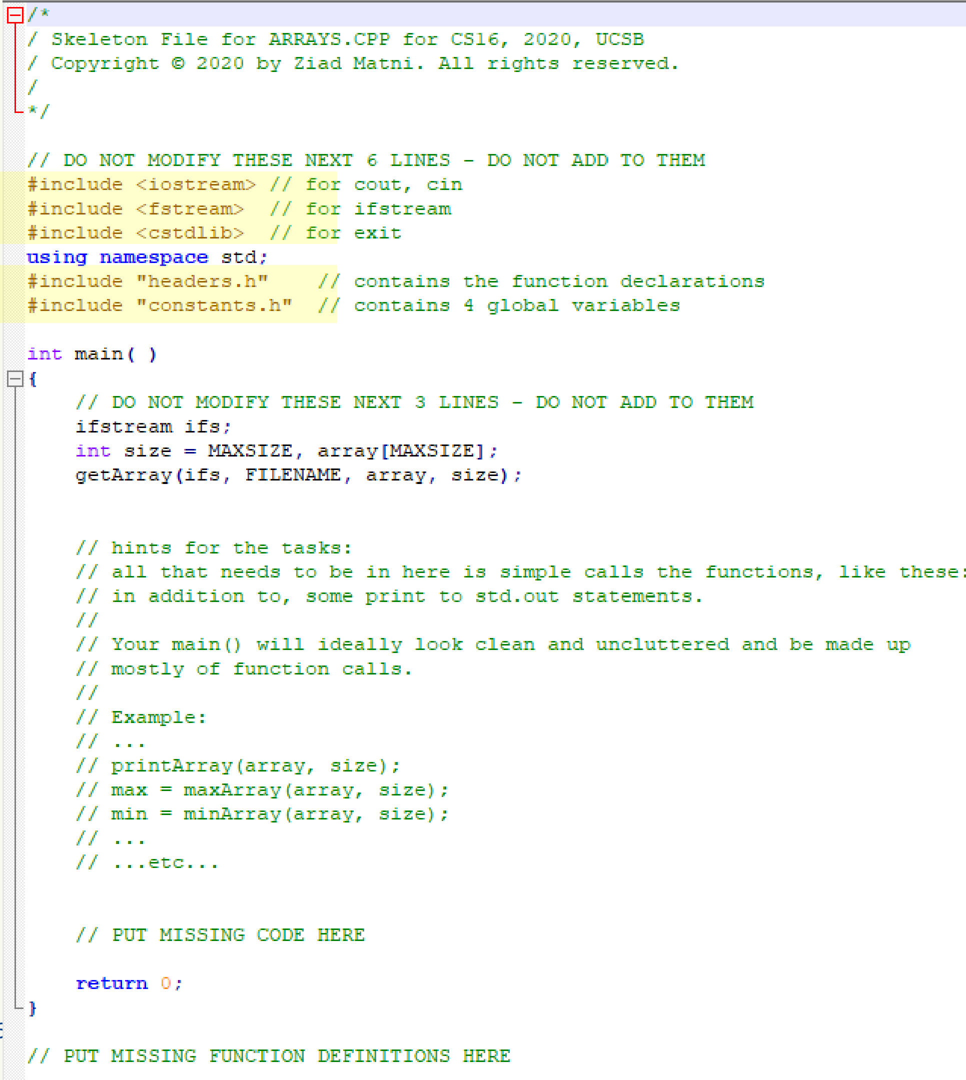
#include “xxx” // for personal libs

When we use **#include “headers.h”**, all of the declarations/definitions in the **headers.h** file are included in the main file. This allows us to call these functions in our file.



```
#include <string>

// BE VERY CAREFUL MODIFYING THIS FILE!
// IT IS ADVISABLE THAT YOU DO NOT MODIFY IT UNTIL YOUR PROGRAM RUNS WITHOUT ERRORS.

// Constants for the search parameters, given as an array
const int NSEARCHES = 10;   // size of the SEARCHES[] array
const int SEARCHES[NSEARCHES] = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4};


//Constants for the data file that your program is reading in
const int MAXSIZE = 20;     // amount of integers in the file (you need this to declare your array size)
const std::string FILENAME = "ArrayFile.txt";    // The file name with the integers
```

constants.h



```
/*
/ Skeleton File for ARRAYS.CPP for CS16, 2020, UCSB
/ Copyright © 2020 by Ziad Matni. All rights reserved.
/
*/

// DO NOT MODIFY THESE NEXT 6 LINES - DO NOT ADD TO THEM
#include <iostream> // for cout, cin
#include <fstream>  // for ifstream
#include <cstdlib>  // for exit
using namespace std;
#include "headers.h"    // contains the function declarations
#include "constants.h"  // contains 4 global variables

int main( )
{
    // DO NOT MODIFY THESE NEXT 3 LINES - DO NOT ADD TO THEM
    ifstream ifs;
    int size = MAXSIZE, array[MAXSIZE];
    getArray(ifs, FILENAME, array, size);


    // hints for the tasks:
    // all that needs to be in here is simple calls the functions, like these:
    // in addition to, some print to std.out statements.
    //
    // Your main() will ideally look clean and uncluttered and be made up
    // mostly of function calls.
    //
    // Example:
    // ...
    // printArray(array, size);
    // max = maxArray(array, size);
    // min = minArray(array, size);
    // ...
    // ...etc...


    // PUT MISSING CODE HERE

    return 0;
}

// PUT MISSING FUNCTION DEFINITIONS HERE
```

There Are Several Ways To Do This Piece-wise Approach

- See “example1” and “example2” in the demo code folder

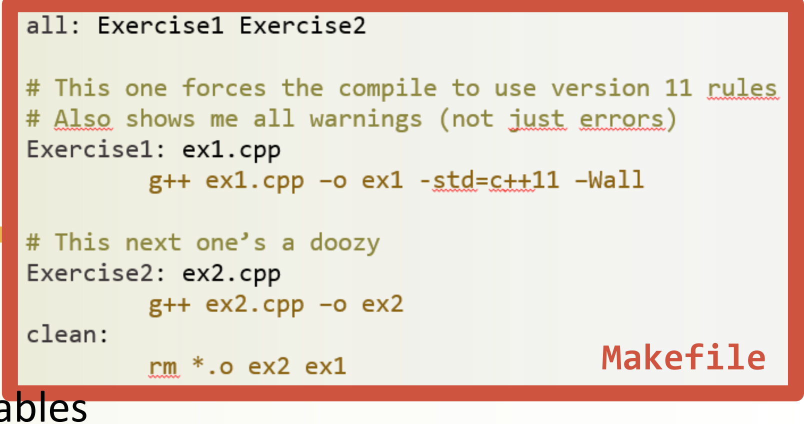
- **example1**: similar to the 1st one I went through (a few slides ago)

- **example2**: by re-arranging headers, we can make one compile command (simpler, but also more limiting)

Syntax of a Make

```
all: Exercise1 Exercise2

# This one forces the compile to use version 11 rules
# Also shows me all warnings (not just errors)
Exercise1: ex1.cpp
        g++ ex1.cpp –o ex1 -std=c++11 –Wall

# This next one’s a doozy
Exercise2: ex2.cpp
        g++ ex2.cpp –o ex2
clean:
        rm *.o ex2 ex1
```

Using **make** in the Linux OS Environment



```
all: Exercise1 Exercise2

# This one forces the compile to use version 11 rules
# Also shows me all warnings (not just errors)
Exercise1: ex1.cpp
        g++ ex1.cpp –o ex1 -std=c++11 –Wall

# This next one’s a doozy
Exercise2: ex2.cpp
        g++ ex2.cpp –o ex2
clean:
        rm *.o ex2 ex1
```

Makefile

- Now that you have a **makefile**, you can execute a compiling process simply by issuing:

`$ make` ← This is will create **<u>ALL</u>** the output executables

- Or you can execute make for one dependency (i.e. program) in particular, like this:

`$ make Exercise1` ← This is will create the output executable for Exercise1 (i.e. the file ***ex1*** in our example)

- In our example, we even provided a way to “clean up” after we’re done by deleting all the executables that we created (in case we wanted to run the compiling again, let’s say)

`$ make clean` ← This will delete all the executables that we created (like ***ex1*** and ***ex2***)

How Many Makefiles Should I Have???

- The usual convention is to have 1 makefile per project
  - For example, 1 per lab
- There should (hard-rule) be only 1 makefile per directory
  - Otherwise, Linux will be confused when you issue “**make**” command
- Remember: you can call it **`makefile`** or **`Makefile`**
  - All the same to Linux

Re: git

- Most (all?) of you have created accounts on GitHub for use with this class' organization
  - Keep your accounts! You will use them in other CS classes!

- You have learned how to operate git from the GitHub website

- You should also know how to operate git from a Linux prompt
  - Please read: https://ucsb-cs56-pconrad.github.io/topics/git_basic_workflow/
  - Link is on our webpage (this week's module)

Debugging Loops

Common errors involving loops include:

- ***Off-by-one*** errors in which the loop executes one too many or one too few times
- ***Infinite loops*** usually result from a mistake in the Boolean expression that controls the loop

Fixing Off-By-One Errors

- Check your comparison: **should it be < or <= ?**
  - Saw a few mistakes like this on the exam ☹

- Check that the var. initialization uses the correct value

Fixing Infinite Loops

- Common mistake: check the direction of inequalities:

should I use < or > ?

- Lean towards using < or > in your loop conditions
- Avoid equality (==) or inequality (!=)

More Loop Debugging Tips: **Tracing**

- Be sure that the mistake is *really in the loop*

- **Trace** the variable to observe *how* it changes
  - Tracing a variable is watching its value change *during* execution.
  - Best way to do this is to insert **cout** statements and have the program *show you* the variable at every iteration of the loop.

Debugging Example

- The following code is supposed to conclude with the variable "**`product`**" equal to the product of the numbers 2 through 5
  - i.e. 2 x 3 x 4 x 5, which, of course, is 120.
- What could go wrong?! ☺

Where might **you** put a trace?

```
int next = 2, product = 1;
while (next < 5)
{
    next++;
    product = product * next;
}
```

DEMO!

Using variable tracing

Loop Testing Guidelines

- **Every time a program is changed, it should be re-tested**
  - Changing one part may require a change to another

- Every loop should at least be tested using input to cause:
  - Zero iterations of the loop body
  - One iteration of the loop body
  - One less than the maximum number of iterations
  - The maximum number of iterations

If all of these are ok, you likely have a very robust loop

Starting Over

- Sometimes it is more efficient to throw out a buggy program and start over!
  - The new program will be easier to read
  - The new program is less likely to be as buggy
  - You may develop a working program faster than if you work to repair the bad code
    - The lessons learned in the buggy code will help you design a better program faster

Testing and Debugging **Functions**

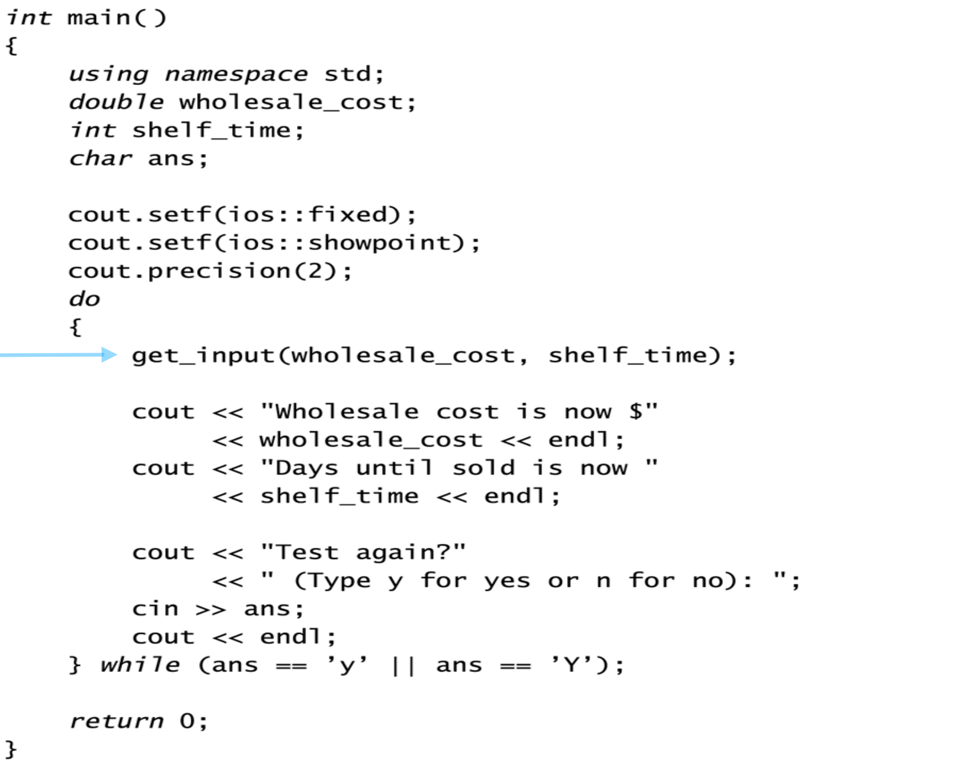
- Each function should be tested as a separate unit

- Test functions by themselves: it make finding mistakes easier!

- “Driver” or “Test” Programs can help
  - Yes: create *another* program to test your original program…

- Once a function is tested, it can be used in the driver program to test other functions

Example of a Driver Test Program



```
int main()
{
    using namespace std;
    double wholesale_cost;
    int shelf_time;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        get_input(wholesale_cost, shelf_time);

        cout << "Wholesale cost is now $"
             << wholesale_cost << endl;
        cout << "Days until sold is now "
             << shelf_time << endl;

        cout << "Test again?"
             << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == ’y’ || ans == ’Y’);

    return 0;
}
```

Stubs

- When a function being tested
  calls other functions that are not yet tested, use a **stub**

- A **stub** is a *simplified version of a function*
  - **A placeholder for the real thing…**
  - i.e. **they're fake functions**

- **Stubs should be so simple**
  that you have full confidence they will perform correctly

Stub Example

```
#include <iostream>
#include <cmath>
use namespace std;
double WeirdCalc(double x, double y);

int main( ) {
    double n, m, w;
    cout << “Enter the 2 values for weird calculation: ”;
    cin >> n >> m;
    w = WeirdCalc(n, m) / (37 – pow(n/m, m/n) );
    cout << “The answer is: ” << w << endl;
    return 0;
}

```

Stub Example

```
#include <iostream>
#include <cmath>
use namespace std;
double WeirdCalc(double x, double y);

int main( ) {
    double n, m, w;
    cout << “Enter the 2 values for weird calculation: ”;
    cin >> n >> m;
    w = WeirdCalc(n, m) / (37 – pow(n/m, m/n) );
    cout << “The answer is: ” << w << endl;
    return 0;
}

double WeirdCalc(double x, double y) // Make WeirdCalc a stub – just for testing!!
{
    //return ( (sqrt(pow(3*x, y%(max(x,y))) – sqrt(5*y/(x-6)) + 0.5*pow((x+y), -0.3);
    return ( 7 );
}
```

Debugging Your Code: The Rules

- Keep an open mind
  - Don’t assume the bug is in a particular location

- **Don’t randomly change code** without understanding what you are doing until the program works
  - This strategy may work for the first few small programs you write ***but it is doomed to failure*** for any programs of moderate complexity

- Show the program to someone else

General Debugging Techniques

- **Check for common errors**, for example:
  - Local vs. Reference Parameters
  - = instead of ==
  - Did you use **&&** when you meant **||**?
  - These are typically errors that might not get flagged by a compiler!!

- **Localize the error**
  - Narrow down bugs by using **tracing and stub techniques**
  - Once you reveal the bug and fix it, remove the extra **cout** statements

- Your textbook has great debugging examples

Pre- and Post-Conditions

Concepts of pre-condition and post-condition in functions

We recommend you use these concepts when making comments

Pre-condition: What must "be" before you call a function

- States what is assumed to be true when the function is called
- Function should not be used unless the precondition holds

Post-condition: What the function will do once it is called

- Describes the effect of the function call
- Tells what will be true after the function is executed (when the precondition holds)
- If the function returns a value, that value is described
- Changes to call-by-reference parameters are described

Why use Pre- and Post-conditions?

- Pre-conditions and post-conditions should be the first step in designing a function

- Specify what a function should do BEFORE designing it
  - This minimizes design errors and time wasted writing code that doesn’t match the task at hand

- **This approach is very popular in industry and is called Test-Driven Development (TDD)**
  - More about this in the next lecture (pre-rec.)

YOUR TO-DOs

❑ Start **Lab4** today

❑ Do **Homework4**

❑ Do **Quiz4** this week (Fri.)

</LECTURE>