Introduction to Software Development – CS 6010
Lecture 6 – Vectors and Linking

Master of Software Development (MSD) Program

Varun Shankar

Fall 2022

Week 1

- Any questions after week 1?
- At this point it would be ideal if you feel comfortable with:
  - Creating and assigning variables, including understanding the different datatypes.
  - If statements (Boolean Logic - &&, ||, !)
  - Basic Loops (For and While)
- It is okay to
  - Still need help from TAs
  - Not have solutions immediately occur to you
- If you feel you are falling behind, speak to me after class.

Lecture 6 – Vectors and Linking

- Topics
  - Testing / Debugger
  - Multi-file projects (Linking)
  - Lab – Multi-File Project
  - Lab – Vectors
  - Homework – Vectors

Debugger

- Breakpoints
- Stepping
  - Over
  - Into (a function)
- Call Stack
  - Variable values

Testing – Making sure the code works!

- Computer will help with
  - Code Compiles – Only tells you that the syntax is correct
  - Runs without crashing
- Now it’s up to us
  - Verify that “Known” outputs are produced based on known inputs
  - assert - #include <cassert>
  - assert(condition)
    - If true, nothing happens
    - else, program crashes

Testing (cont)

- assert(isVowel(‘a’)) // If this works, what % of confidence that code is correct?
- What, and how many, inputs do we need to use in our testing?
- Want tests to be fast, automatic, and deterministic

```
void runTests()  // One way to automate our testing would be to create a runTests() function.
{
        // bunch of asserts… or calls to individual test functions
}
int main() {
        runTests(); // comment this in/out as we develop more code
}
```

Testing (cont)

```
void testIsVowel()
{
        assert( isVowel( 'a' ) );  // What else to test?
        assert( isVowel( 'e' ) && isVowel( 'i' ) ); // Why is this not ideal?
        assert( isVowel( 'o' ) );
        assert( isVowel( 'u' ) );
        assert( isVowel( 'y' ) );  // Tested a,e,I,o,u,y, what's missing?
        assert( isVowel( 'A' ) );  // Other capital vowels.  What else?
        assert( !isVowel( 'b') );
        assert( !isVowel( 'z' ) ); // Edge case consonants.  What else?
        assert( !isVowel( '!' ) ); // Punctuation marks, etc.
        // While not a exhaustive set of tests, these tests would give us a
        // fairly high confidence level that isVowel() works correctly.
}
```

```
void runTests()
{
        testIsVowel();
        testIsNumWords();
}
```

Projects Using Multiple (Source Code) Files

- Header Files
  - So far have contained library code, but we can also use them to help organize our own project.
  - Named <something>.h (or .hpp, .hxx)
    - Standard library files do not use the *.h* but are also header files.
  - Contains function *signatures*.
    - Just the declaration of the function, not the implementation.
  - To use a function, you don't care about how it is implemented, all you need to know is what parameters it takes, and what it returns – ie. the *signature*.
- Implementation Files – CPP Files (.cpp, .cc, .CC)
  - Contain the full code of functions
  - Usually named <something>.cpp – ie: much of the time, the .h and .cpp file come in pairs

Function Declaration vs Definition

```
int isOdd( int x ); // Function Declared – although this should actually be in a .h file…
int main() {
        cout << isOdd( 5 );
        …
}
```

- Where do we *define* isOdd()? // define == implement
  - Can create it in myUtilFunctions.cpp; or even in
  - isOdd.cpp

Compilation and Linking

- Each separate .cpp file will be compiled into a .o (object) file.
  - The .o file contains the *object code* (binary code) that can be directly run on the computer.
  - If your project has multiple .cpp files, then it will have multiple .o files.
  - All the .o files will then be *linked* together to create a single executable.

Compilation / Linking (Cont)

- Given a project with these files:
  - main.cpp – main does not have a corresponding .h file.
  - mathFunctions.cpp / mathFunctions.h (Remember: .ccp, .h files usually come in pairs)
  - ioFunctions.cpp / ioFunctions. h
- To compile them on the command line, we would run:
  - clang++ -c main.cpp // -c means compile, don't link
    - Assuming no errors, this would create a main.o file.
  - clang++ -c mathFunctions.cpp // Note, the .h file isn't specified in the compile command.
    - Produces mathFunctions.o
  - clang++ -c ioFunctions.cpp // Compiler creates ioFunctions.o
  - clang++ -o myProgram main.o mathFunctions.o ioFunctions.o
    - Links all the .o files into one output (-o) file named myProgram
- To run myProgram:
  - > ./myProgram

Compilation / Linking

main.cpp

```
#include< myFuncs.h>

 int main() {
            …
            isOdd()
            …
 }
```

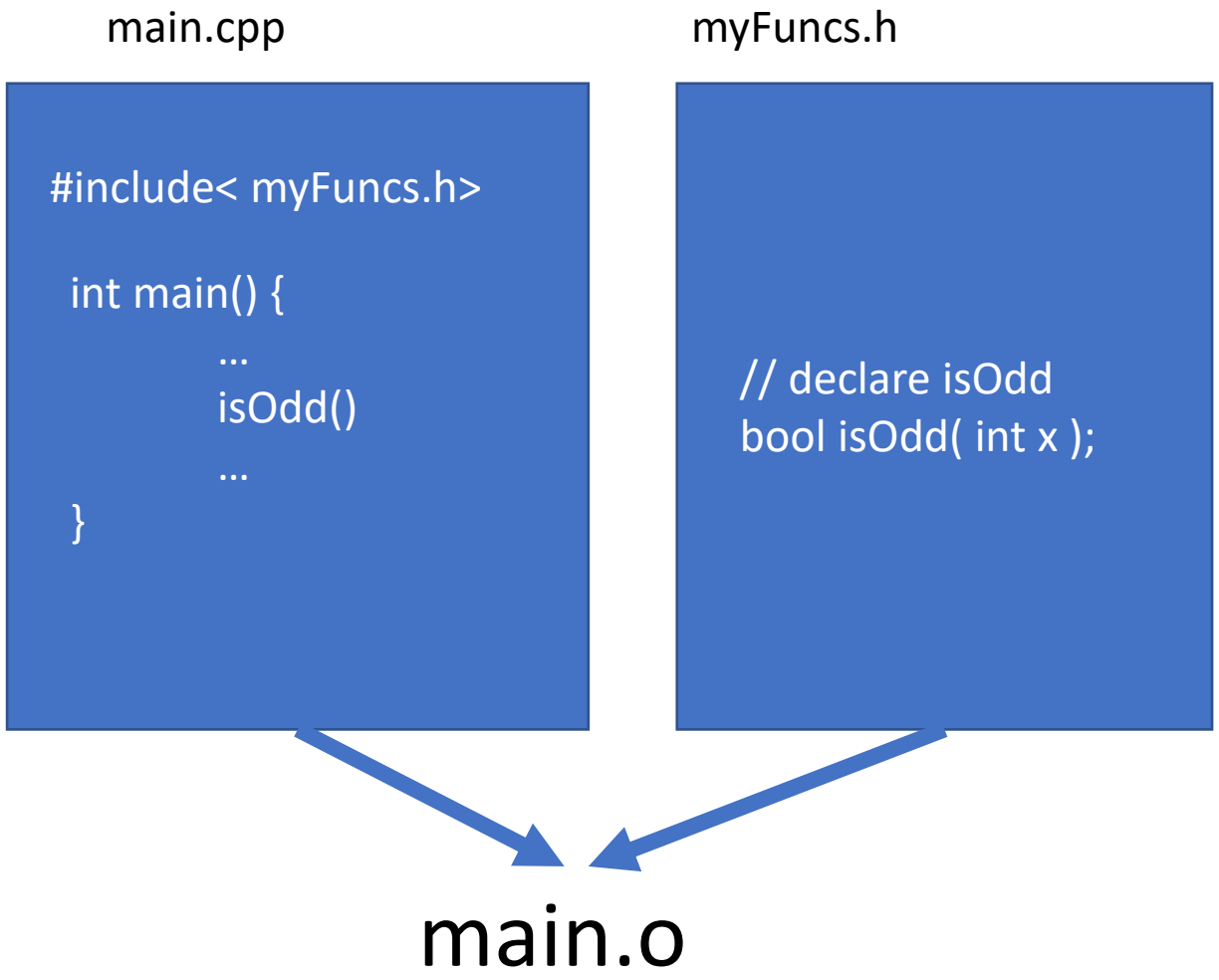
myFuncs.h

```
// declare isOdd
bool isOdd( int x );
```

myFuncs.cpp

```
// implement isOdd
bool isOdd( int x ) {
    return x % 2 == 1;
}
```

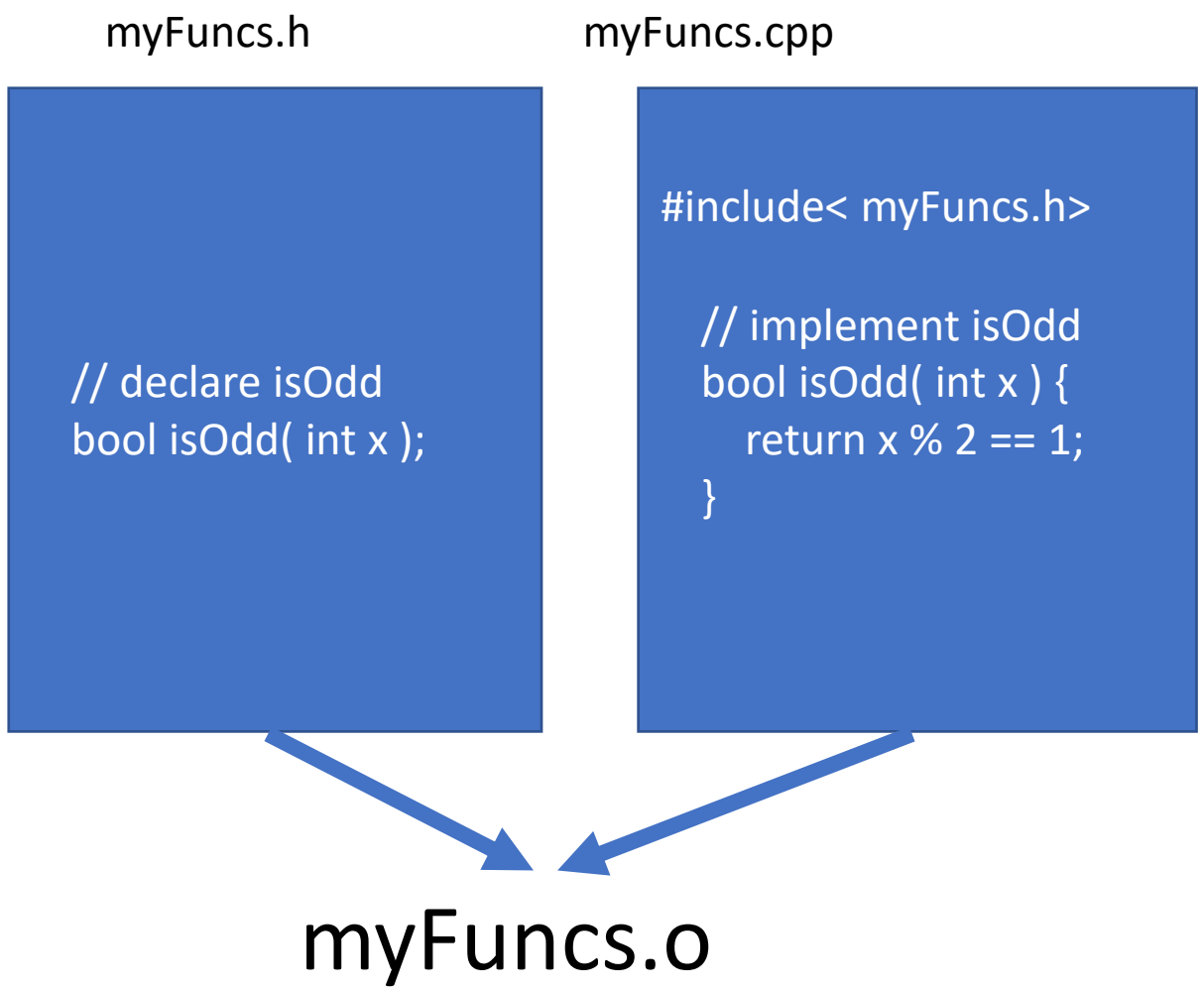
Compile main.cpp into main.o



- What is missing from main.o (in order to actually run the program)?
  - Where is the actual code for isOdd()?
- What does #include actually do?

Compile myFuncs.cpp into myFuncs.o

- Here is the actual implementation of isOdd().
- Notice, again, that the .h and .cpp files usually come in pairs.



Linking

- Source files (.h and .cpp) do not play a part in linking
- Only .o (object) files
- clang++ -o myProgram main.o myFuncs.o

- > ./myProgram

- Tools to compile and link everything on the command line.
  - Makefile

Command Line Arguments, PATHs

- ls –l
  - The –l is a *command line argument* provided to the *ls* program.
- ./myProgram
  - The *./* means, look in the current directory.
- PATH
  - Your command line shell knows to look for *executables* (programs) in a few places.
  - Those places are in your PATH variable.
    - On the command line, type *echo $PATH*
  - Notice that the current directory is not in your PATH.

Arrays

- Why do we use arrays?
  - To store a set of closely *related* enumerated (positioned one after another) information
  - Additionally, to store this type of information when we don't know how much there is (until runtime)
  - Useful when a function needs to process a bunch of (the same/related) data
    - Eg: float average(int score1, int score2, int score3, …)
    - Better to use: float average(int[] scores)

- C++ actually supports two types of arrays.
  - C-style arrays, which look like this: int[] scores
  - and std::array, which we'll talk about later
  - Both are fixed-size, i.e., their sizes don't vary during program execution.

C++ Vectors

- C++ also has an object called std::vector (found in <vector>)
- This is a variable-size array (size can change during program execution).
  - Arrays were created first (as part of the C) language, and thus are very primitive, requiring the programmer to do a lot of work to manage the memory they use.
  - Vectors (found in the standard library) were created in C++ (relatively recently in terms of C++ lifespan) to make arrays easier to use.
  - While understanding the concept of array data storage using built-in *arrays* is reasonable, most C++ code will use *vectors* because they are easier (and safer) to use.

Vector Basics

- #include <vector>
- To create (define/assign) a vector: // Same as any var: <type> <name> = <value>
  - But because a vector is a *list* of some *type* of data… we have to specify that type when we declare the variable.
- std::vector<int> grades; // Creates an empty *array* that will hold integers
- vector<string> names(5); // A vector of 5 strings (with default values of "")
- vector<double> numbers = { 1.0, 2.2, 9.4, -3.7 }; // Creates a vector of those 4 #s
- Accessing values in a vector:
  - The same as accessing the data in a string
  - double n2 = numbers[2]; // The position ('2' here) is known as the "index"
- Adding a new value to the *end* of a vector uses *.push_back(value)*
  - numbers.push_back(17.7);
- Remove the last element from the vector: .pop_back()
- Use a variable name that is plural (ie, make sure the variable name ends with an 's')

Vector Functions (Methods)

- .size() // Returns the number of elements currently in the vector
- .push_back()
- .pop_back()
- [ ] // Index into the vector, i.e., access element in vector
- .front()
- .back()

Vector Example

- Create a vector and place the numbers 12, 13, 14, 15, 16, 17 in it.

```
vector<int> myNumbers;
int baseNum = 12;
for( int index = 0; index < 6; index++ ) {
        myNumbers.push_back( baseNum + index   );
}
```

For Each Loops - Iterating Over A Vector

- Introducing a new form of the *for* loop – the *for each* loop.
  - Note: In C++ this loop is not explicitly named *for each*, but in some other languages it is.
  - for(int x : numbers) { cout << x << "\n"; } // read, for each integer (x) in numbers
  - for(char c : aString) { // Note, foreach should be indented just like everything else.

```
if( isPunctuation(c) ) {
    count++;
}
```

  - }
  - for(double d : numbers) { total += d; }
- What is the difference between a *for* and a *for each* loop?
- When can't we use for each?
  - When we do not wish to look at every element (must go first to last)
  - When we need to know the index we are looking at

Today’s Assignment(s)

- Code Reviews (Roman Numerals)
- Lab – Multi-File Projects
- Lab – Vectors
- Homework – Vectors