**CS 162 Worksheet 4**

1. Work with multiple files – dos and don’ts

* NEVER compile .h files in a g++ command
  + Simply compile the .cpp files and the #include'd header files will be compiled as part of those.
* DO #include header files in each source (.cpp) file that uses those functions (or classes).
* DO recompile any .cpp files that #include the header file WHEN the header file changes.

1. Concept of Structs:

Structs (originated in the C) are the predecessors of classes (C++)

* a user-defined data type of grouping related data together to model some 'object'.

Ex. Create a struct that represents a vehicle. Explain why you listed your member variables the way you did.

**Struct Vehicle {**

**string make; // make of the car**

**string model; // model of the car**

**int year; // year the car was first produced**

**string doors; // 4-door, 2-door, etc**

**string drive; // AWD, RWD, FWD, etc**

**};**

Now, suppose you are given the following struct:

struct Garage {

int num\_of\_vehicles;

Vehicle\* v;

};

Explain the relationship between the Vehicle and the Garage struct.

**The garage struct contains an integer representing the total number of vehicles, and contains a Vehicle struct *v*.**

Once a Garage object g is defined, how would you populate the array of vehicles within g?

**You would populate the array of vehicles within *g* by selecting an array index of g.[index].**

How would you use these structs in a program that allows users to search for a vehicle of a specific model, rent a vehicle, and return a vehicle?

**You would use these structs in a program by allowing users to search for specific members of the *garage* struct that contain vehicles that match the make and model of their choosing. You can also use this for renting and returning vehicles by keeping track of who rented the vehicle, when it should be returned by, and which vehicle was rented.**

1. Describe what each line of the makefile does and answer the following questions.

CC = g++ **// sets the compiler equal to g++**

exe\_file = mult\_div **// the executable file to write to equals mult\_div**

$(exe\_file): mult\_div.o prog.o **// specifies the object files needed for the variable exe\_file (mult\_div)**

$(CC) mult\_div.o prog.o –o $(exe\_file) **// compiles the object files to the variable exe\_file**

mult\_div.o: mult\_div.cpp **// specifies the required file for mult\_div.o**

$(CC) –c mult\_div.cpp **// compiles mult\_div.cpp to mult\_div.o**

prog.o: prog.cpp **// specifies the required .cpp file needed for prog.o**

$(CC) –c prog.cpp **// compiles prog.cpp to an object file**

clean: **// after make is run, cleans the remaining files**

rm –f \*.out \*.o $(exe\_file) **// removes all object and output files**

(1). What happens if you put the clean target at the top above the first target to make the executable?

**The clean target will remove any object and output files present in the current working directory.**

(2). What happens if you define your .o targets before your executable target?

***Make* will throw an error.**

(3). What happens if you do not use tabs to indent?

***Make* will throw an error.**

1. Understand Program Errors
   1. Compiling Errors - The compiler GIVES you a line number and error message.

E.g., syntax, missing a semicolon, use a variable before declaring it, etc.

What is your debug strategy for compiling errors?

**Read the error message, go to the line number, and try to troubleshoot the error, looking up documentation if needed.**

* 1. Runtime Errors - produce the wrong output or crash, leaving no clues as to why.

E.g., Segmentation Fault

What is your debug strategy for runtime errors?

**For runtime errors, run the code through Valgrind to make sure that there are no memory issues, and use try/catch statements to narrow down the point of error.**

1. Pointer with structs:

**1 struct Pokemon {**

**2 string name;**

**3 int size\_move;**

**4 string \*moves;**

**5 };**

**6**

**7 struct Pokedex {**

**8 int num;**

**9 Pokemon\* p\_arr;**

**10 };**

**11**

**12 void test( \_\_\_①\_\_\_ var){**

**13 // code**

**14 }**

**15**

**16 int main () {**

**17 Pokedex p;**

**18 p.num = 5;**

**19 p.p\_arr = new \_\_\_\_④\_\_\_\_ [p.num];**

**20 for (int i = 0; i < p.num; i++) {**

**21 ...**

**22 p.p\_arr[i]\_③\_moves = new \_\_\_⑤\_\_\_ [p.p\_arr[i].size\_move];**

**23 }**

**24 test (\_\_\_②\_\_\_);**

**25 return 0;**

**26 }**

1. From line 17, is **p** a data type or an object?

**From line 17, p is a data type.**

1. What should be filled in **③** at line 22, a . or ->? What’s the difference? Can we replace it with:

**p.p\_arr->moves = ...**

Why or why not?

**In line 22, 3 should be a . (period). This is because *p* is a struct, and *p.p\_arr[i].moves* is a member of the *p* struct.**

1. What should be filled in **④** at line 19?

**In line 19, 4 should be a *Pokemon* struct.**

1. What should be filled in **⑤** at line 22?

**In line 22, 5 should be a *Pokemon* struct.**

1. If **②** is **p**, what should be filled in **①**?

**If 2 is p, 1 should be a *Pokedex struct*.**

1. If **②** is **p.p\_arr**, what should be filled in **①**?

**If 2 is p.parr, 1 should be a *Pokemon* struct array.**

1. If **②** is **p.p\_arr[0]**, what should be filled in **①**?

**If 2 is p.p\_arr[0], 1 should be a *Pokemon* struct.**

1. If **②** is **p.p\_arr[0].moves**, what should be filled in **①**?

**If 2 is p,p\_arr[0].moves, 1 should be *Pokemon.moves*.**

1. If **②** is **p.p\_arr[0].moves[1]**, what should be filled in **①**?

**If 2 is p.p\_arr[0].moves[1], 1 should be *Pokemon.moves[1]*.**