CSE 425 Studio: Control Abstraction I

These studio exercises are intended to extend your familiarity with ideas and techniques for more abstract (procedure and function based) program control flow, again in a multi-paradigm programming language (C++). In this studio you will again work in self-selected groups of ~3 people, and will report the results of your work on the studio exercises in an e-mail to the course account. Students who are more familiar with the material are encouraged to partner with and help those who are less familiar with it, and asking questions of your classmates and professor during the studio sessions is highly encouraged as well. There are also links on the main course web page to helpful web sites for these exercises, which you also are encouraged to use as resources. Please record your answers as you work through the following exercises. After you have finished please e-mail your answers to the required exercises, and to any of the enrichment exercises you completed, to the cse425@seas.wustl.edu account, with the subject line "Control Abstraction Studio I". The enrichment exercise is a chance to dig deeper into the material, especially if you breeze through the required ones.

PART I: REQUIRED EXERCISES

1. Form a team of ~3 people and write down your names as the answer to this exercise.

2. Open up Microsoft Visual Studio 2013 and create a new project for this studio (for example, named **`ControlAbstractionStudioI`** or something similar that identifies which studio this is for). Modify the main function signature so that it matches the standard portable (e.g., between Windows and Linux) main function signature (and variable names) for a C++ program: **`int main (int argc, char * argv[])`**.

Outside your main function, declare and define a function that takes a single integer by value and returns (again by value) the result of squaring that value. Also declare and define another "function" (really a procedure as we have defined them) that has a **`void`** return type, takes an integer by reference, and squares the integer that was passed to it. Also declare and define another function that has a **`void`** return type, takes a pointer to integer by value, and squares the integer pointed to by the pointer that was passed to it. In your main function, declare a local integer variable and initialize it to some value greater than 1 (e.g., 3), and print out the value of that variable before and after each of the following steps: (1) pass that variable into a call the first function you wrote (also print out the value the function returned), (2) pass the variable into a call to the second function you wrote, and then (3) pass the address of the variable into a call to the third function you wrote. Build and run your program, and as the answer to this exercise describe what you observed, and what those observations say about how side effects are introduced or avoided when writing functions and procedures in C++.

3. Repeat the previous exercise, but inside each of the functions (including the main function) print out the address of the function itself, the address of each of its parameters and local variables, and for any that were pointer variables also print out their values (which are the addresses of the locations to which they point). Build and run your program, and as the answer to this exercise (based on what you observed) please describe (1) which of the addresses you printed out were the same (if any), and (2) how far apart the different addresses were relative to each other (e.g., within the call stack).

4. Repeat the previous exercise, but in your main function dynamically allocate the integer variable instead of declaring it as a local (stack) variable: use **`new`** to allocate it, and then store its address in a local pointer variable (which you should dereference in your calls to the first and second functions, and pass directly to your call to the third function). Build and run your program and as the answer to this exercise (based on what you observe) please describe (1) which of the addresses you printed out were the same (if any), and (2) how far apart the different addresses were relative to each other (with some variables being allocated on the call stack and others on the heap).

5. Declare and define (or use ones from a previous set of studio exercises) a base class (or struct) and another class (or struct) derived from it. In your main function comment out the code from the previous exercises, declare a local variable of the derived type, and also declare a pointer to the base type and a pointer to the derived type, both initialized with the address of the local variable. Print out the address of the local variable, and also print out the values of the pointers. Build and run your program and as the answer to this exercise please (1) show your program's output, and (2) based on the output your program produced describe how the base vs. derived portions of a class are positioned in the stack frame for a local variable in C++.

6. Convert your derived class (or struct) from the previous exercise into a template with a single type parameter **`T`**, and give the derived class (or struct) a member variable of type **`T`**. Replace the local variable declaration for the derived type in your main function with two local variables, one that instantiates the template with type **`int`**, and another that instantiates the template with type **`long`**. Replace the pointer to the derived type with two different pointers, one to each instantiation of the template. Similarly, replace the pointer to the base type with two different pointers to the base type, one pointing to each instantiated object.

Print out the addresses of the local variables, print out the addresses of their member variables, and also print out the values of the pointers. Build and run your program and as the answer to this exercise please (1) show your program's output, and (2) based on the output your program produced describe how the base vs. derived vs. member variable portions of each instantiation are positioned in the stack frame.

PART II: ENRICHMENT EXERCISE (Optional, feel free to skip or try variations that interest you)

7. Repeat exercises 2 and/or 3 but convert the functions into function templates (which the calls to them will instantiate implicitly with the appropriate argument types). Build and run your code, and as the answer to this exercise please (1) show your code and the output your program produced, and then (2) based on the output and anything else you observed in writing or building your code, comment on the extent to which function templates are transparently integrated within C++.