CSE 332 Studio Session on C++ Expressions

These studio exercises are intended to give you experience working with expressions (here in a very basic form, for prefix addition) in C++, and to introduce basic features and techniques for debugging C++ programs within the Visual Studio C++ environment.

In this studio you will again work in small groups of 2 or 3 people, selected at your discretion. As before, students who are more familiar with the material are encouraged to help those for whom it is less familiar. Asking questions of your professor and teaching assistant (as well as of each other) during the studio sessions is highly encouraged as well.

Please record your answers as you work through the following exercises. After you have finished please email your answers to the required exercises, and to any of the enrichment exercises you completed, in an e-mail to the cse332@seas.wustl.edu course e-mail account, with the subject line "Expressions Studio". The enrichment exercises are optional but are a good way to dig into the material a little deeper, especially if you breeze through the required ones.

PART I: REQUIRED EXERCISES

1. Find your team members in the studio area, sit down at/around and log in to one of the Windows machines, and write down the names of the team members who are present (if a team member arrives late please catch them up on the work, and add their name) as the answer to this exercise.

2. Create a new Visual C++ Win32 Console Application project for this studio. Download the **`prefix_adder.cpp`** and **`prefix_adder.h`** files from the code example directory for this studio session (linked on the main course web page), add them to the project, and exclude from the project the file that has the main function that windows generated. Build the program, fix any warnings and errors, and then run it with command line arguments **`+ 8 + 9 10`** and say what the program printed out in that case, as the answer to this exercise.

3. In Visual Studio, right click next to each of the following lines of code, and set a breakpoint on each one: the first line of the **`main`** function, the first line of the **`parse_and_compute`** function, the line in the **`main`** function just after where the **`parse_and_compute`** function is called, and the first line of the catch block in the **`main`** function. On the menus at the top of your Visual Studio project window, find the keyboard shortcuts that let you step into the function calls in a statement, step over the function calls in a statement, and run to the next breakpoint. Use the Project→Properties→Configuration Properties→Debugging→Command Arguments field to specify the command line properties, and re-run the previous exercise using the debugging commands to watch what happens within your program. As the answer to this exercise, please describe how the program call stack grew and shrank, and how the values of the most important variables changed, as the program ran.

4. At the command line, try removing some of those command line arguments (for example, producing command line arguments like **`+ 9 10`** or **`+ 8 + 9`**), and as the answer to this exercise say which of the command line arguments resulted in the program computing and printing out a value, which of the command line arguments resulted in an error message, and what the error message(s) said.

5. In Visual Studio, using the breakpoints from exercise 3, but changing the settings in the Project→Properties→Configuration Properties→Debugging→Command Arguments field to specify one of the command lines that produced an error message in the previous exercise, and use the debugger to watch what happens within your program when it is executed with those command line arguments. As the answer to this exercise, please describe the sequence of steps that led the program to produce an error message (not just where the exception was caught, but where it was thrown and why).

6. In Visual Studio, introduce different kinds of programming errors into the source file (**Suggestion:** comment out the code that was there originally whenever you introduce an error so you can put it back the way it was before moving on to introducing a different error), including (1) changing the starting index given at the top of the main function, and (2) passing the current index by value rather than by reference. As the answer to this exercise, document what you changed in each case and how it affected the behavior of the program when you stepped through (and into) it in the debugger.

PART II: ENRICHMENT EXERCISES (optional, feel free to skip some and try ones that interest you).

7. Log onto shell.cec.wustl.edu (or one of the Linux lab machines) and repeat exercise 3 on the Linux platform, using the **`prefix_adder.cpp`** and **`prefix_adder.h`** and **`Makefile`** files provided on the course web page (**Hint:** in the Makefile, fill in what you want the name of the executable program file to be on the line where it says `EXECUTABLE`, put the name of the source file where it says `CMPL_SRCS`, and put the name of the header file where it says `HEADER_FILES`).

8. Repeat exercise 4 using the **`gdb`** program. To see a more complete discussions of the gdb debugging commands and environment than we covered in the lecture, please type **`man gdb`** or **`info gdb`** at the Linux command prompt on shell.cec.wustl.edu.

9. Repeat exercise 8 by opening up the header and source file in **`emacs`** and then running **`gdb`** by issuing the **`Esc x gdb`** command from within **`emacs`** (**Hint:** you can also build the program from within **`emacs`** by issuing the **`Esc x compile`** command which brings up the **`make -k`** command). For a good discussion of the emacs editor and environment, please type **`man emacs`** or **`info emacs`** at the Linux command prompt on shell.cec.wustl.edu.