# Studio Session on C++ Strings and Arrays

These studio exercises are intended to introduce basic features of C++ strings and arrays, and to give you experience using them with basic input/output features within the Visual C++ environment.

As before, students who are more familiar with the material are encouraged to help those who are less familiar with it and asking questions of your instructors and teaching assistants during the studio sessions is highly encouraged as well.

Please record your answers you work through the following exercises. After you have finished, please submit your answers to the required exercises and to any of the enrichment exercises you completed to Blackboard or our OJ system (if appliable).

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. Declare a (3-dimensional) 2 by 3 by 5 array of unsigned integers and initialize all its elements to zero when you declare it. Iterate through the cells in the array, and set the value of each cell to be the product of its coordinates (the cell at 0, 0, 0 would have value 0, the cell at 1, 2, 4 would have value 8, etc.) and print (to **cout**) the coordinates and value of each cell. Build and run your program and show its output (with the coordinates and values of the array's cells) as the answer to this exercise.

2. Use **argc** and **argv** to iterate through all of the program's command line arguments from position 0 to position **argc-1** and **push_back** each command line argument into a vector of C++ style strings (of type **vector<string>**). After that, your program should iterate through the vector (hint: you can use iterators or use indexing from position 0 to one less than the value given by the size method) and print out the string at each position in the vector. Run the program with some random command line arguments, and as the answer to this exercise (1) show the output from the program, and (2) say whether or not all of the command line arguments given to the program appeared in that output.

3. Change your program so that instead of pushing the command line arguments into a vector, it concatenates them into a single C++ style string with spaces in between them. Wrap the completed string in an input string stream (of type **istringstream**) and then use its extraction operator (the >> operator) to pull off one argument at a time from the string stream into a C++ style string and then print each one on its own line to **cout**. Again, run the program with random arguments after the program name on the command line, and as the answer to this exercise (1) show the output from the program, and (2) say whether or not all of the command line arguments given to the program appeared in that output.

4. Modify your program so that it accepts exactly two command line arguments (in addition the program name which always appears in **argv[0]**). If exactly two additional command line arguments are given, the program should print out each of the additional command line arguments (directly to **cout** instead of using a string or vector as intermediate storage) and then return 0 to indicate success.

Otherwise, if more or fewer than 2 command line arguments are given to the program, the program should print out an appropriate usage message that gives the program's name (use **argv[0]** to do that, so that if someone recompiles the code to produce a different executable the error message is still correct) and asks the user to run the program with two additional arguments, and then return a non-zero value to indicate failure. Build the program and run it three times: with 1, 2, and 3 additional command line arguments respectively. Immediately after each run of the program, run the **echo %errorlevel%** command, to see what value the program returned. As the answer to this exercise, show the output and return value that was produced by each run of the program (and the subsequent **echo %errorlevel%** command).

5. Modify your code from the previous exercise so that if exactly two additional command line arguments were given, instead of printing them out it uses them as the names of input and output files from which to read and write respectively. The program should open the first file in an **ifstream** and test whether or not it was opened successfully (please see today's slide on file streams for how to do that test), and if not it should (1) simply output an error message giving the file's name and indicating it could not open it for reading, and (2) return a different non-zero value than it would return if the wrong number of command line arguments was given.

Otherwise the program should open the second file in an **ofstream** and test whether or not it was opened successfully, and if not it should output an error message giving the file's name and indicating it could not open it for writing, and return a different non-zero value than it would return if the wrong number of command line arguments was given or if a file could not be opened for reading.

Otherwise, the program should repeatedly (until it reaches the end of the input file) extract a string (using the >> extraction operator) from the input file stream and write it (using the << insertion operator) on its own line (i.e., ending the line after each string it writes) to the output file stream, and then return 0. Build your program and fix all errors and warnings you may see. Use an editor program of your choice to open up a text file (which you will use for input to your program) in the directory where your executable file was placed by the compiler and put a number of different space-separated strings (some on the same line and some on different line) into the file. Save and close the file, and then run your program three times: (1) once with only the name of the input file you created, (2) once with two file names, neither of which exists in the directory, and (3) once with the name of the input file you created and another name of a file that does not exist in the directory (which the program should then create with its output). As the answer to this exercise, for each of those runs show the program's output and its return value, and if it created an output file show the contents of both the input and output files for that run.

**PART II: ENRICHMENT EXERCISES** (optional, do the ones that interest you).

6. Modify your code from exercise 1 so that in addition to printing out the coordinates and contents of the cell, it also prints out the memory address of the cell. Please show the program's output, and based on it describe how you think the array is laid out in memory relative to the position indexes of the cells (for example as a formula involving coordinates i, j, and k), as the answer to this exercise.

7. Repeat exercise 3 but instead of treating all the data as strings, give different integer values to the program and read those values from the string into int variables. String (and file) input streams will do such conversions for you automatically in their extraction operators, which can really ease some programming tasks.

8. Run your program from exercise 5 with a valid input file name but the name of an output file that is read-only (you can make a file read only from the graphical interface by opening its properties, or from the command line you can use the **attrib** command). As the answer to this exercise please show the output of the program and the value it returned when you did that.