EE6411 Lab Sheet #3

Debugging C++ Programs & Develop Functions

Reiner Dojen[1]

[1]reiner.dojen@ul.ie

1 Objectives

- Learn how to use the debugger to find problems in a program.
- Develop functions to solve typical engineering problems.
- Declare, define and call functions.
- Using (static) arrays.

2 Preparation

If you haven't worked through "E-Activity 1-6: The Debugger", please do so now (the E-Activity can be found on the module's Brightspace page under Week 5→E-Activity 1-6: The Debugger).

3 Exercises

Ex3.1 The Debugger

Download the archive "Ex3-1-TheDebugger.zip" from the module's Brightspace page (you'll find it on the "Laboratory 3" page) and copy it to a suitable location. Extract the contained VSC project and open it in VSC. A file called "f2c.cpp" is available in this project - please open it in VSC: this source code contains several errors - please do not "fix" these!!! In this execise we are using the debugger to reveal the mistakes.

Now its time to set a breakpoint (a location at which execution of your program in debug mode will be interupted) - click with the left mouse button in the area at the very left edge of the screen - a red circle will appear to indicate the breakpoint (alternatively, right-click in the source code and select "Breakpoint→Insert Breakpoint").

To execute your program in debug mode, select "Debug→Start Debugging" (or click in the menu bar on "Local Windows Debugger") - execution will stop at the breakpoint. The current location (next statement to be executed) is highlighted with a yellow arrow.

You now can use the control buttons - Continue, Step Into, Step Over and Step out - to control the flow of the program.

Proceed as follows:

- Start the program in debug mode.
- Click repeatedly on “Step Over” and observe execution of the program.
- You (hopefully) will recognize that the same line “300.00 0.00” is printed continously.
- Restart your program by clicking on “Debug→Restart” (or click on the restart icon in the menu bar).
- When debugging stops at the breakpoint, right-click on the variable “`fahrenheit`” and select “Add Watch” and make sure that at the bottom of the VSC window the watch is displayed. The variable will appear in the watch area (bottom of screen), where it’s value is shown. At this point in time, it probably has some unpredictable apparently random value. Click on “Step Over” until you reach the `for` loop. Click on “Next” again and observe the change in the value of the variable: As you can see, its value changes to 300 and not the expected value 0. Inspection of the `for` loop reveals that the programmer was careless and used an assignment (`=`) instead of a less equal comparison (`<=`). Stop the debugger, fix this error in your source code and start the debugger again.
- The modified program is opened, the breakpoint should still be indicated. Make sure that variables `fahrenheit` and `celsius` are added to the watch. Delete the breakpoint (either left-click on the breakpoint or right-click on the source code line and select “Breakpoint→Delete Breakpoint”). Set a new breakpoint in the line “`celsius = 5/9 ...`” (the first line in the `for` loop and step through the `for` loop several times. You will recognize that now the fahrenheit value progresses as expected, but the celsius value remains at 0. Let’s inspect the expression for celsius: Check the fahrenheit value (it should not be non-zero). Highlight the expression “`fahrenheit - 32.0`”, right-click on the highlighted area and select “Quickwatch”: A new window pops up displaying the value (and type) of the expression. Highlight the whole expression “`5/9 * (fahrenheit - 32.0)`” and Quickwatch its value (it should appear as 0). Hightlight the expression “`5/9`” and Quickwatch its value - again we get 0. This is of course the case, as whole numbers in our code by default are integers!!!. To fix this problem, stop the debugger, change the source code from “`5/9`” to “`5.0/9.0`” and restart the debugger.

- Make sure the same breakpoint is still activated and run your program. Evaluate the value of "`5.0/9.0`" by Quickwatching it (you should get a value like 0.55555555558). Step through the program and observe now the correct values.
- Make sure that the breakpoint at the `for` loop is activated and that the `fahrenheit` variable is displayed in the watch. Restart your program. Execute your program until `fahrenheit` has a value of 60 and execution stops at the `for` loop. Double-click on the value of `fahrenheit` and change the value 60 to 400 (or any other value that is greater than 300) and hit enter. Click on "Step Over" and observer how now the `for` loop terminates (as value of the variable is now larger than 300).

Ex3.2 Print Numbers in Binary Format

Write a program that prints out integer numbers in binary form (print out 8 binary digits, ie. decimal numbers from 0-255). The following algorithm presents a possible solution (others do exist):

```
printBinary(value)
   digit = 0x80 (leftmost bit in 8-bit number)
   as long as digit is not equal 0
      if ( (value AND digit) is 0) print 0
      else print 1
      shift digit one bit to right
```

Your program should use a function `void printBinary(int value);` that will print the passed number in binary format. Please make sure to declare your function at the beginning of your source file. Make sure to include a pseudo-code representation for this function in the comments.

The user will pass the integer number to the main function (using its arguments `argc` and `argv` - don't forget to convert the arguments from string to int using the library function `int atoi(const char *value)`. The main function will then call `printBinary(...)` to display the nubmer in binary format.

Ex3.3 Array with Random Numbers

Write a program that creates an (static) array of 20 integers and initialises it with random values between 0 and 99 (see Exercise 2.2 Random Numbers on lab sheet #2 for details on how to create random numbers). Implement the following functions (make sure to declare all functions - except `main()` - at the beginning of your source file):

`int main()`: A simple main function that creates the array, and fills it with random numbers. Afterwards, it

- prints all the values of the array to the screen
- calls all other functions and prints their return value (please use meaningful messages, such as "max value = xx" rather than just printing the values by themselves).

`int minValue(int field[], unsigned int size)`: Finds the smallest value in the array field and returns it.

`int maxValue(int field[], unsigned int size)`: Finds the largest value in the array field and returns it.

`double averageValue(int field[], unsigned int size)`: Calculates the average value of all elements in the array field. Please make sure that you get the correct floating point (double) value and not the integer value of the average.

Ex3.4 Salespeople

Use a single subscripted array to solve the following problem: A company pays its salespeople on a commission basis. The salespeople receive €200 per week plus 9 percent of their gross sales for that week. For example, a salesperson who grosses €3000 in sales in a week receives €200 plus 9 percent of €3000, or a total of €470.

Write a C++ program that creates an array of 200 random gross salary values in the range 200 to 10,000 and then determines how many of the salespeople earned salaries in each of the following ranges (assume that each saleperson's salary is truncted to an integer amount):

1. €200 - €299	2. €300 - €399	3. €400 - €499
4. €500 - €599	5. €600 - €699	6. €700 - €799
7. €800 - €899	8. €900 - €999	9. €1000 and over

Ex3.5 Duplicate Elimination

Use a single-subscripted array to solve the following problem: Prompt the user to enter a number in range 10-100 (both limits inclusive) 20 time. As each number is read, print and keep the number only if it is not a duplicate of a number already read. Make sure that the user is prompted exactly 20 times - i.e. if the user enters 20 times the same number, only one number is stored. At the end, print a message displaying the total number of elements stored and list all stored numbers.

To test you program, you can use a textfile that contains a list of twenty numbers. Assuming that the file is called "numbers.txt" and your program is called "duplicate", you can run your application with command:

```
duplicate < numbers.txt
```

to read the entered numbers from your file (rather than from the keyboard). In essence, you now connect the standard input stream (`cin`) to the file rather than to the keyboard. Thus, `cin` will read from the file ☺.

Note: this will not work in Powershell - please start the command prompt by searching for "cmd".

Ex3.6 Duplicate Elimination II

Modify the previous exercise: This time make sure that each number is within the valid range - if not, keep prompting the user for a valid number. Also ensure that each number is unique - if a duplicate is entered, keep prompting the user for a unique number. Thus, this time, you will end up with 20 unique numbers. At the end, print a message displaying a list of all stored numbers.

Ex3.7 Turtle Graphics - Advanced

The Logo language made the concept of *turtle graphics* famous. Imagine a mechanical turtle that walks around the room under the control of a C++ program. The turtle holds a pen in one of two positions, up or down. While the pen is down, the turtle traces out shapes as it moves; while the pen is up, the turtle moves around freely without writing anything. In this problem, you'll simulate the opertion of the turtle.

Use a 30-by-30 array `floor` which is initialized to zeros. Read command from an array that contains numbers to indicate the commands (see Table 1 below). Keep track of the turtle position and direction at all times and whether the pen is currently up or down. Assume that the turtle always starts at position 0,0 (top left corner) of the floor with its pen up and facing to the right.

Command	**Meaning**
1	Pen up
2	Pen down
3	Turn right
4	Turn left
5 x	move x spaces forward (two number command), where the next number indicates the number of space to move
6	Print the 30-by-30 array
9	End of data

Table 1: Turtle Commands

Suppose that the turtle is somewhere near the center of the floor. Then the following "program" would draw and print a 12-by-12 square:

```
2
5,12
3
5,11
3
5,11
3
5,11
1
6
9
```

As the turtle moves with the pen down, set the appropriate elements of array `floor` to 1s. When the command 6 (print) is given, wherever there's a 1 in the array, display an asterisk (*), or some other character you choose. Wherever there's a zero, display a blank.

Write a function `void turtle(int commands[], int commandSize)` that implements the turtle graphics capabilities here.

Add a `main()` function to the program. Add at least three arrays with commands to create different turtle graphics. Also, add a text menu with the following options:

- “Execute” your sample turtle graphic 1 (it should finish with a “draw” command (6) and end marker (9)).
- “Execute” your sample turtle graphic 2 (it should finish with a “draw” command (6) and end marker (9)).
- “Execute” your sample turtle graphic 3 (it should finish with a “draw” command (6) and end marker (9)).
- Add additional options if you created more samples.
- “Record” a turtle graphic - prompt the user for up to 1000 commands and store these in an array. Use the end marker (9) to terminate command input.
- Display the last recorded turtle graphic.
- Exit the program.

You can download a sample executeable from the module’s Brightspace page to give you an idea of what is expected.