University of Central Lancashire
UCLan

CO1404 : Introduction to Programming

Lab 5 : Variable Scope, Nesting Loops, Debugging and Recap

Lab Summary

The lab worksheets are designed to help you write your first programs. Students who have experience programming should find it straightforward at first. However, there will be advanced questions later on. There's a lot of reading in this worksheet as we introduce the tools required. Later worksheets will tend to be shorter, with more work for you to do!

Remember:

- If you have any questions or require assistance, please get your tutors attention. We are all here to help!
- Refer to previous weeks materials for guidance.

Objectives

This Lab aims to achieve the following objective(s) below:

- **PO.1: Implement Loops and User validation techniques**
- **PO.2: Implement Loops and User validation techniques**
- **PO.3: Implement Loops and User validation techniques**

Lab Activity

This weeks lab activity begins with precursor tasks. These will help you understand what was discussed in this weeks lecture. Unlike prevoius weeks this week Lab only consists of one stage, Stage 1. As usually you are expected to complete this stage by the end of the week. This week is really important as it recaps previous weeks in preparation for your coursework.

Set your goals sensibly - don't just aim for the minimum or you may struggle to pass the module.

> **Disclaimer:**
>
> The "Advanced Challenges" are intended as fun exercises designed to push your knowledge to the limits. I encourage you all to give them ago, ask questions if you get stuck but most importantly don't stress about them.

Nested `for` Loops

1. Create a new Visual Studio C# Console project called `NestedShapes`.

2. Write a 'for' loop that displays 10 X's in a row.

   > Note: The code to do this is shown in the lecture notes.

3. Now ***nest*** this `for` loop inside another `for` loop to display a 10 x 10 box of X's as shown below:

   ```
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   xxxxxxxxxx
   ```

   > Note:
   >
   > - Again, this is shown in the lecture notes.
   > - You must use nested for loops in all these exercises. **Do not try to solve these problems with many repeated Console.WriteLine statements!**

4. Now change the size of the box so it is 10 high and 5 wide:

```
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
```

5. Now we will try to draw different shapes with 'for' loops. First update your code to show an **L** shape as shown below.

```
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx
xxxxxxxxxxxxxxx
```

This is easy when you see the shape is actually made of two different sized boxes. So you need two copies of the code you have already with different dimensions.

6. Now change your program to draw a **T** shape.

```
  xxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxx
  xxxxxxxxxxxxxxx
       xxxxx
       xxxxx
       xxxxx
       xxxxx
       xxxxx
       xxxxx
       xxxxx
```

```
        XXXXX
        XXXXX
        XXXXX
```

This has the same two boxes swapped around. Also the tall thin box is pushed to the right. Do this by displaying extra spaces before each line for this box. Think carefully where to put that code, but also experiment - you will learn by trying out different ideas.

Nested Number Grid

1. Create a *new* Visual Studio C# Console project called `NestedNumberGrid`. The goal of this task is to dispaly a grid of numbers as shown below:

```
01 02 03 04 05 06 07
08 09 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31 32 33 34 35
```

2. Write with the code to draw a box of X's, 7 wide and 5 high

> Note: Copy one of your nested loops from the previous exercise.

Next, we want to update the code to display a grid of numbers as shown above. There should aso be a space between each number and a blank line between each row. To draw the number grid we need to replace each "X" with a number. Also the numbers must have two digits, small numbers like 7 need to be displayed 07.

3. Start by replacing the **X** in your `Write` statement with your *inner* loop variable. So, for example, if your inner loop uses a variable called `j`, then display `j` instead of X (remember you don't use quotation marks to display variables)

```
Console.Write( j );
```

4. Add an extra `WriteLine` between each row and you should see a result like this:

```
1234567
1234567
1234567
1234567
...
```

5. Now we can make sure that each number has two digits with some special C# formatting. You may have seen this if you reached the later stages of last week's lab - the `:00` part makes sure numbers are displayed with at least 2 digits. Replace the line you just wrote with this, ***but make sure you use the correct variable name for your program***:

   ```
   Console.Write("{0:00}  ", j);
   ```

   You should see a result like this:

   ```
   01 02 03 04 05 06 07
   01 02 03 04 05 06 07
   ...
   ```

   This is close, but not quite what we want. The number should not reset back to 01 on each row, but that is what the loop variable is doing. The loops are working OK, so we don't want to change their variables. So we need another variable.

6. Declare a new integer at the start of your program and initialise it to 1. Inside the *inner* loop, increment this variable. Display this variable in your write statement instead of the loop variable. You should get the correct result:

   ```
   01 02 03 04 05 06 07
   08 09 10 11 12 13 14
   15 16 17 18 19 20 21
   ...
   ```

When writing loops you can use the loop variables themselves or you can declare new variables that work independently. Exactly what you need depends on the problem you're trying to solve.

> *Virtually every loop you write will need to use variables, but always think carefully about the correct variable to use, or if you need to create a new variable.*

The Visual Studio Debugger

1. Create a new Visual Studio C# Console project called `DebuggerTest`

2. Copy this code into the correct place in the C# shell code. Note that this isn't a very useful or well-written program because it's just a test for the debugger:

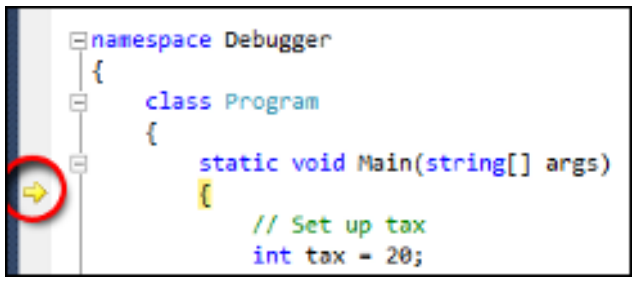
```
// Set up tax
int tax = 20;
Console.WriteLine("Tax is {0}", tax);

// Read cost from user
int cost;
Console.Write("Enter cost between 1 and 100: ");
cost = int.Parse(Console.ReadLine());

// Check the user's input **Not good validation! For testing only**
if (cost > 1 || cost < 100)
{
    Console.WriteLine("Correct, your cost is between 1 and 100");
}
else
{
    Console.WriteLine("You're not taking me seriously. I quit");
    Console.ReadLine();
    return; // This line quits the Main method, which exits the program
}

// Add tax to cost
cost = cost = tax;
Console.WriteLine("Total cost is {0}", cost);
```

   This program contains two bugs. However, even if you can see them already don't fix them yet. We are going to see how the bugs show up using the debugger.

3. Run the program. Ignore the request to type a value in the range `1-100` and instead type `123`. The program still thinks your input was in the correct range. And then the tax calculation is incorrect - it is supposed to add 20 to your cost and the answer should be 143, but the program gives the answer 20. Press return to exit the console program if you haven't already.

   We will **single step** through the program, just run a line at a time and see what's happening inside to find out why it is going wrong.

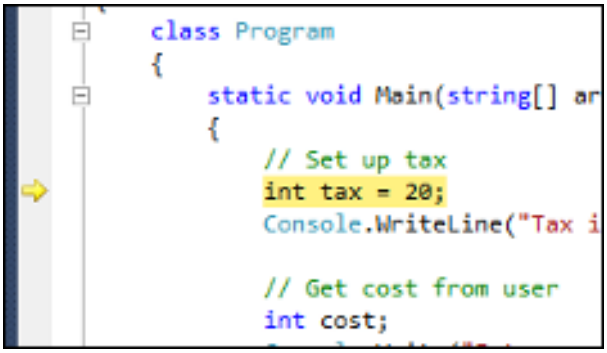
4. To single step press F10, or go to the menus: Debug -> Step Over. Visual Studio will rearrange it's windows a little and show you the debugger view. The first thing to notice is that the program has paused on the first line. The yellow arrow shows the next line that will be executed, the opening brace of the program:



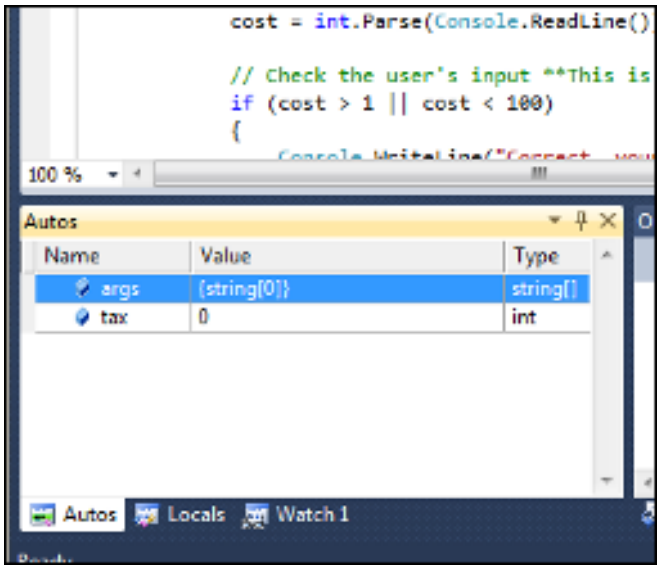
5. Let's single step again, you can press F10 again or alternatively press the "Step Over" button that has appeared now that you are in the debugger:

6. The yellow arrow will step to the next code line. Notice that it skipped over the green comment, because comments are not part of the program.

> Note :
>
> It is now on the line that sets tax to 20, but this line hasn't executed yet. The yellow arrow shows *the next line to be executed*:



7. Look at the right side Visual Studio window (or sometimes at the bottom). You will see the "Autos" pane and two other tabs to the "Locals" and "Watch" pane:



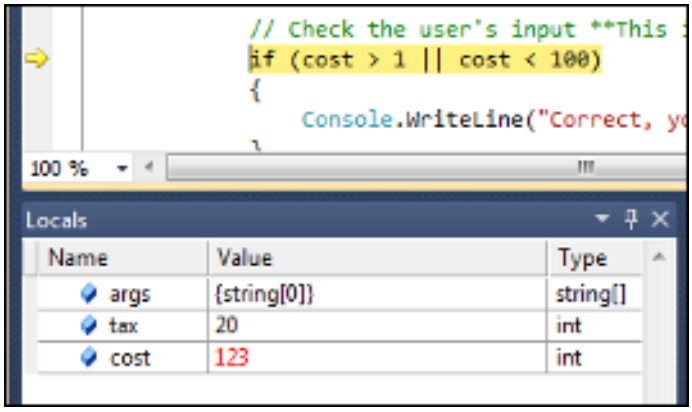
These panes show the variables in your program:

- The "Autos" pane shows the variables that are used near the current line. You can see the `tax` variable, and also an `args` variable that comes from the line above that says "Main". We are not concerned with this `args variable here.
- Click on the "Locals" tab. It shows all the variables used in the current block (i.e. within the current curly brackets). That includes the 'tax' variable and the 'cost' variable that is declared a few lines down.
- Click on the "Watch" tab. This shows variables you are interested in. At first it is empty. Click in the "Watch" pane and type `tax` . The `tax` variable will be shown. You can also type expressions, try typing `tax + 20`.
- All variables and expressions are updated as you step through the program.

8. Click on the "Locals" tab. Notice that `tax` and `cost` are `0`. That is because they have not been initialised or assigned yet. The line we are currently on will initialise `tax` to `20`. So single step again (press F10 or the Step Over button), and see how the `tax` variable in the "Locals" pane is now 20. It is also shown red, which means that this value has just changed.

9. So far we have seen no bugs. So single step three more times to pass over the text display and user input. When you step over the user input line, the console window will display. The debugger cannot step over that line until you enter something. So type `123` again and press return. You may then need to bring the Visual Studio window to the front again. User input in the debugger always happens like this.

   Now you should have reached the if statement:



Everything looks OK, The variables `tax` and `cost` contain the values we expect. As we have typed a cost that is out of range, we are hoping that this `if` statement is `false` and the code inside is not executed. We want the yellow arrow to skip over the block so the text "Correct, your cost is between 1 and 100" is *not* displayed.

- Single step two more times. There will be a problem. The arrow goes into the if block and will display the message. That is a bug. Since everything was correct before the `if` statement, we know that the `if` statement contains the bug.
- We want to stop the program and look more deeply. Press the stop button (near the play button at the top). This will quit out of the debugger and stop the program.
- Single step through the program again (type in `123`) until you reach the `if` statement again - but do not execute that line!

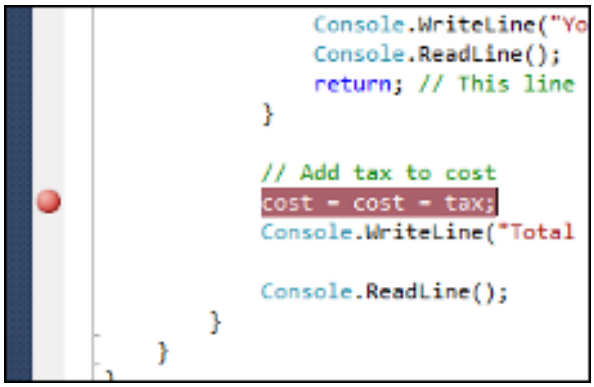
10. Now you can hover your mouse over variables and expressions to see their values. You need to be a little careful, sometimes you have to try a couple of times:
    - Hover over the `>` operator in `cost > 1`. It shows that `cost > 1` is evaluated to be **true**. That's OK
    - Hover over the `<` operator in `cost < 100`. It shows `cost < 100` is evaluated to be **false**. That's OK
    - So hover over the `||` operator (or) in the middle of the condition. It shows the entire condition is evaluated to be **true**, which means that the program will go into the `if` statement...

11. There's the bug! - the `if` statement effectively says: display "Correct" if `cost > 1` **or** `cost < 100`. If we type a value over 100 it will be `> 1` even if it isn't `< 100`, so the message is displayed. The `if` statement should use 'and', the `&&` symbol.

12. Stop debugging and fix this bug. Run the program and confirm it correctly identifies that 123 is out of range.

13. However, there is still a second bug. The tax is not added on correctly. Let's get to that point in the program quickly with a **breakpoint**:

    - You should have stopped debugging at this point. Now select the line that tries to add the tax to the cost (yes, the bug is obvious, but don't fix it yet!). Press F9 or in the menus Debug -> Toggle Breakpoint. The line will go red and a red dot appear at the left. This is a breakpoint:



        - You can quickly switch on/off breakpoints by clicking where the red dot is. Try it now, switch it off, then on again.
        - Run the program (not single step) with F5 or the Play button. The program is execute as normal - enter 60 for the cost. Then it will stop and enter the debugger soon as the program reaches the breakpoint (you may have to bring the Visual Studio window to the front again).

14. Breakpoints are a quick way to run your program normally but then stop it at an area of interest. It can be much faster than single stepping through the entire code. Now we are in the debugger we can use all same debugging features as before.

    - Notice in the "Locals" pane that the variables `cost` and `tax` are correct (60 and 20). Single step so the calculation occurs... and you will see that the `cost` variable is changed to 20 instead of the 80 we would expect. So we know there's a bug on that line we just executed.
    - Fix the obvious mistake and confirm that the program works. Remember to switch off breakpoints after you've finished with the, or you program will keep going into the debugger.

That completes a quick overview of the debugger. There are many more features, which you may discover for yourselves or be introduced in other modules. Do use the debugger when your program does not do what you expect. Bugs are an everyday part of programming, even with years of experience a professional programmer will spend considerable time using the debugger. So you should be professional too and know when to use this powerful tool.

STAGE 1

1. Open your shape drawing project from earlier **NestedShapes**
2. Repeatedly single step through the program and watch how the yellow arrow jumps around the loops within loops. Watching the debugger step through loops can help you understand better how they work.

You now have a choice of tasks:

- There are some **Practice** questions below, which reinforce the key programming skills you should have gained so far. Practice is vital to learn programming so this section is recommended. Solutions are provided so you can check your progress.
- However, if you feel very comfortable with your skills so far, then you may skip the practice questions and try the **Advanced** tasks later in the worksheet.

1.1 Practice

The solutions to these tasks will be provided on blackboard early next week! So you can check your work. **Attempt the exercises before looking at the answers or you will gain no skill**.

1. Create a new Visual Studio C# Console project called **Practice**.

   > Note: You don't need to create a new project for every answer, you can put multiple answers in the same project. Create a new project if you feel your current project is getting too full though. Separate each answer with these lines (same as a previous worksheet);

   ```
   Console.ReadLine();
   Console.Clear();
   ```

2. Follow these simple instructions exactly and when you are finished your code should match the solution ***precisely, line for line***. This tests whether you understand the programming terms used:
   - Declare an integer called `fileCount` and initialise it to `0`.
   - On the next line declare an integer called `fileSize` but do not initialise it.
   - On the next line increment `fileCount`
   - On the next line assign `1000` to `fileSize`

- On the final line display a message in this format (using the `fileCount` and `fileSize` variables to display the numbers in green). **Your output should be *identical* except for the colours:**

```
The number of files is 1, the file size is [1000].
```

3. Ask the user for their age. Display a different message depending on whether they are a child, adult or pensioner. You can decide the age ranges.

4. Write a number guessing game program. You can use the word guessing game from the week 3 and 4 lectures as a basis for your code:
   - The program selects a number - you can just choose a number and type it in the program (e.g. 231), or better you can get a random number using the method shown in the week 3 lab sheet.
   - The user guesses this number. If they are wrong the program say so and tells them whether the correct number is higher or lower than their guess.
   - The guessing game continues until the user guesses correctly - a winning message is displayed.

5. Use a for loop to display the 12 "times-table" up to 12 x 12:

```
12 x 1 = 12
12 x 2 = 24
12 x 3 = 36
...
```

> *A little tricky*: Update your program to show all the "times-tables" from 1 to 12. You will need to nest your loop.

6. Display a menu with 5 choices (you decide what the choices are...). Ask the user for their choice that must be in the range 1-5. Fully validate their answer:
   - If they type a choice outside the valid range, or if they type something that isn't a number (like words), then an error message is displayed and they are prompted to enter their choice again.

1.2 Advanced Challenges

Tasks for those who want to stretch themselves. This material is not necessary to pass the module. But I recommend having a go as even thinking about these tasks will help you develop.

1.2.1 Printing Triangles

1. Use nested for loops to draw these shapes. Keep the code for each shape in one project so that your final program draws all three shapes.

```
x
xx
xxx
xxxx
xxxxx
xxxxxx
xxxxxxx
xxxxxxxx
xxxxxxxxx
xxxxxxxxxx
```

```
         x
        xxx
       xxxxx
      xxxxxxx
     xxxxxxxxx
    xxxxxxxxxxx
   xxxxxxxxxxxxx
  xxxxxxxxxxxxxxx
```

```
xxxx         xxxx
 xxxx       xxxx
  xxxx     xxxx
   xxxx   xxxx
    xxxxxxxx
```

2. At the start of your program allow the user to input the height of the shapes as an integer. Then update your code so each shape is drawn as the desired size. So for example, the first triangle above is currently height 10, the second triangle is height 8. After this code change, all the shapes will be the same height - the loops extended to match the user's input.

3. Change each shape drawing to show the outline only, for example:

```
x
xx
x x
x  x
x   x
x    x
x     x
x      x
x       x
xxxxxxxxxx
```

1.2.2 Drawing Circles (even more advanced!)

1. Draw a **circle** of X's using nested for loops, with the user specifying the radius. The example below is illustrative, you will need a formula:

$$2\sqrt{R^2 - L^2}$$

If the user's choice for the radius of the circle is R (rows), and you are drawing the row that is L lines above or below the centre-most row, then the number of X's in that row is (from Pythagoras): Using this formula adds a little difficulty because L = 0 on the centre line, but we have to start drawing at the top line. For that reason you might want to start your loop counter at a negative value.

The final full circle should look somewhat like this. Larger circles will look better.

```
      xxxx
    xxxxxxxx
  xxxxxxxxxxxx
 xxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
 xxxxxxxxxxxxxx
 xxxxxxxxxxxxxx
  xxxxxxxxxxxx
    xxxxxxxx
      xxxx
```

2. However, once you have your program printing out the circle to the console, you will see that the circles are stretched vertically due to the fact that the X character is taller than it is wide. Can you overcome this effect?