PRACTICAL ASSIGNMENT 0

Visual Studio and basics C++ programming

February 9, 2012

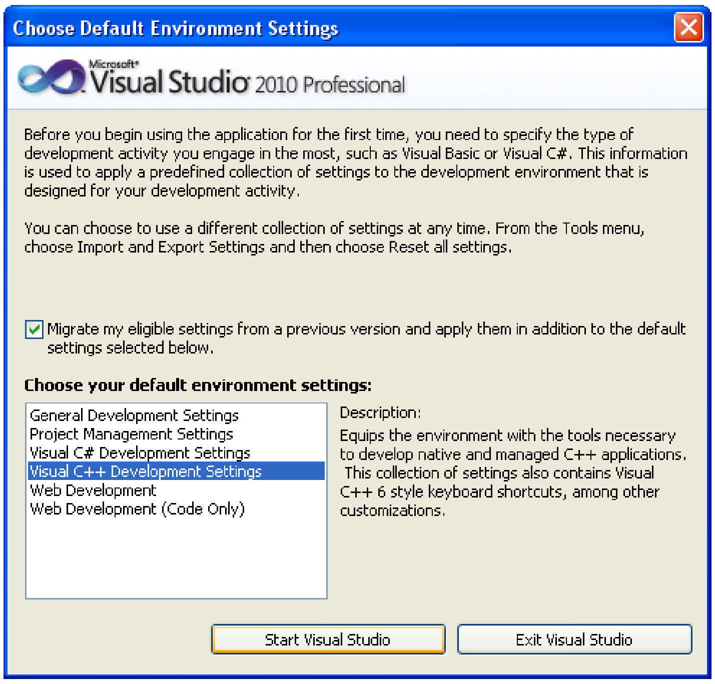
This first practical assignment starts out with a brief tutorial on the creation, compilation and execution of a project in Visual Studio 2010, followed by a tutorial on the available debugging tools. The second part of the assignment consists of several exercises to get some hands-on experience with C++ syntax and several basic functions. This assignment does not need to be handed in.
If you prefer to work on your laptop or you wish to practice at home, you can download Visual Studio from the Academic Alliance website. Please follow the instructions at:

`http://www.cs.uu.nl/intern/technical/ntdoc/msdn-aa/`

1 Starting a new project

1. Start Visual Studio from
   `start -> Standard Applications -> Informatica ->`
   `Microsoft Visual Studio 2010 Professional -> Microsoft Visual Studio 2010`.

2. The *Choose Default Environment Settings* screen might pops up, especially if this is the first time you start Visual Studio 2010. Select `Visual C++ Development Settings` from the list of options (Fig. 1). If Visual Studio is already set to another default environment, you can change it by selecting `Tools -> Import and Export Settings... -> Reset all settings`.


Choose Default Environment Settings
Microsoft Visual Studio 2010 Professional

Before you begin using the application for the first time, you need to specify the type of development activity you engage in the most, such as Visual Basic or Visual C#. This information is used to apply a predefined collection of settings to the development environment that is designed for your development activity.

You can choose to use a different collection of settings at any time. From the Tools menu, choose Import and Export Settings and then choose Reset all settings.

☑ Migrate my eligible settings from a previous version and apply them in addition to the default settings selected below.

Choose your default environment settings:

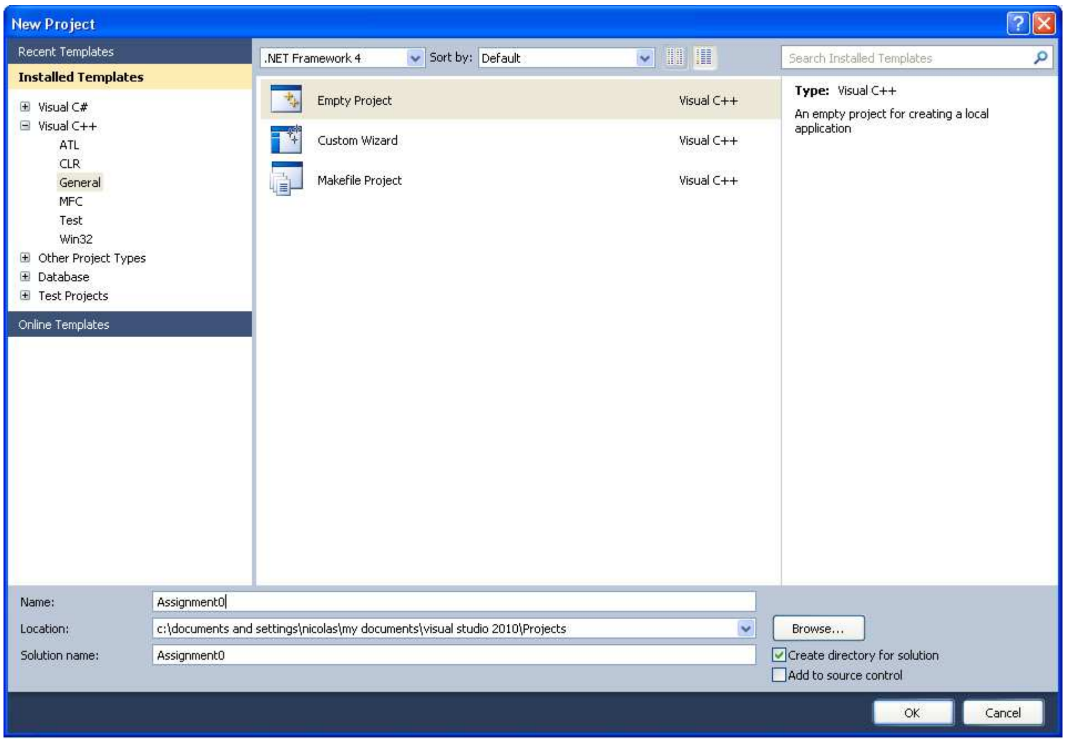
General Development Settings
Project Management Settings
Visual C# Development Settings
Visual C++ Development Settings
Web Development
Web Development (Code Only)

Description:
Equips the environment with the tools necessary to develop native and managed C++ applications. This collection of settings also contains Visual C++ 6 style keyboard shortcuts, among other customizations.

Figure 1: The Choose Default Environment Settings screen.

3. A project is a set of files that belong together (header files, implementation files, etc.) and in most cases form one program. Nevertheless multiple projects and programs can be ordered in a single solution. You can now close the start page, and construct a new project:

   - Select `File -> New -> Project...`
   - You can see that many types of project can be created, containing code and settings for specific applications, such as graphical user interfaces, web applications or applications for smart devices. The most useful ones are the Empty Project (to code from scratch), the Win32 Console Application (to have the DOS console as user interaction tool and precompiled headers), the Win32 Project (to create a window based project) and the MFC Application (to have window and GUI widgets). Choose `Visual C++ -> General -> Empty Project`. Then you specify the name of your project under *Name*, and select the storage location for this project. Pressing *OK* finishes the project creation (Fig. 2). Note that Visual Studio will automatically create a folder for your project named after the project name.



Type: Visual C++

An empty project for creating a local application

Name: Assignment0

Location: c:\documents and settings\nicolas\my documents\visual studio 2010\Projects

Solution name: Assignment0

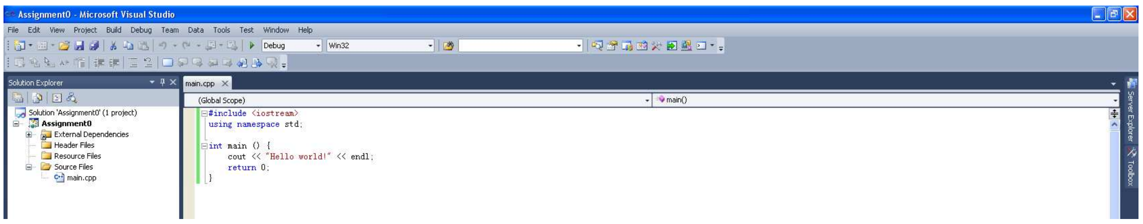
Figure 2: Creation of a new project.

4. Next, we add a file to the empty project. Select your project by clicking on the project name in the *Solution Explorer* (on the left side). The solution explorer gives an overview of your projects and the files that these projects contain. The *class view* pane specifies your projects in terms of classes, functions and variables. Then right click on `Source files` and choose `Add -> New Item...`. In the *Add New Item* window, select `Visual C++ -> Code -> C++ file (.cpp)` to add a source file to your project, and name it as you want. You now see the new file inside the source folder of the solution explorer pane, and your empty source file opened. Note that, although we do not need it right now, you can also add header files or other files to your project that way.

2 Simple program

At this point, we have a project containing a file that is ready to be used. Add the following, typical, but illustrative first program to the source file (Fig. 3):

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello world!" << endl;
    return 0;
}
```



```
#include <iostream>
using namespace std;

int main () {
    cout << "Hello world!" << endl;
    return 0;
}
```

Figure 3: Creation of a simple program.

Compiling the project. By selecting `Build -> Build Solution...` in the menu, all projects in the open solution are compiled. Alternatively, you can choose to compile individual projects through this same menu or by right clicking on your project name in the solution explorer and selecting `Build`. The `Build Solution` option compiles only those project files and components that have been changed since the last build. To clean and rebuild the entire solution (or project), select `Build -> Rebuild Solution...` (or by the right click menu). This is necessary when a compilation of a project breaks down after a *Build Solution*, which might happen with large projects. Lastly, `Clean` simply removes all files generated when building a project. Build your project, and check the successful compilation in the output pane (bottom). The location of the compiled executable file is given.

Errors and warnings. Visual Studio reports compilation errors and warnings in that output pane. Modify your code to introduce an error (misspell cout or return for example) and try to build again. By double clicking on the error or warning line in the output pane or by selecting *Go to Next Message*, the editor jumps to the line of code where the error or warning occurs. If an error or warning message is too cryptic, you can find an explanation by doing a query with the error or warning code (such as `C2065`) in the on-line or off-line help (press F1). Alternatively, you can use the less cluttered `Error list` pane to browse through errors and warnings. To activate this pane, select `View -> Error list`.

Debug and Release Build. A debugger is a computer program that lets you step through your program, line by line, and examine the values of variables or look at values passed into functions. Using the debugger gives a better insight in the program flow and helps you figuring out why it is not running the way you expected. As the compiler is currently configured, it produces debug information when you compile the program. This information is used by the debugger such that you can view the source code while debugging, instead of the machine code. The added debugging information becomes redundant once the application is finished. A release build of your software is the compiled version of your software without debugging information. In general, a release build is faster and smaller. To switch between the release and debug build, use the rolling list just below the menu.

Program Execution. Fix the introduced error, and build again. At this point, the program can be executed by selecting `Debug -> Start Debugging` (or pressing F5 or clicking the green arrow left to the rolling list). Selecting `Debug -> Start Without Debugging` (or pressing Ctrl+F5) results in a program execution that does not stop at debugging breakpoints (on which we will elaborate in the next section). If the software executes as expected, then the job is finished. However, if your code crashes during execution or it displays unexpected behavior, the debugger can help to track down the problem(s). To stop the program execution during a debug session, select `Debug -> Stop Debugging` (or press Shift+F5). Start your project, you will briefly see a console window with the printed text.

As the program terminates, the console window closes, it is the normal termination. To verify that everything goes well and the printed message is correct, you can start your program from the command line (where the console is always active and does not close). To do that, open a command prompt: `start -> Run...` then write `cmd` and click `OK`. Move to the folder where your executable file is using the `cd` command and start the program by typing its name (like `Assignment0.exe`). You should see the printed text.

3 Debugger

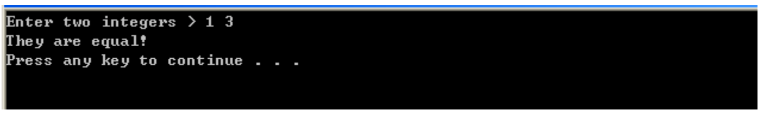
The debugger helps you to find and fix problems in your programs. Any debugger (whether in Visual Studio or another software developing environment) will provide two basic functions to help you fix your code:

- Trace Execution: You can execute one line of your program at a time, so you will be able to see exactly where (and when) a problem occurs.
- Watch Variables: As you step through your program, you can monitor your variables as they get changed (so you can identify the source of bad data right away).

A variety of tools are available in Visual Studio to help you accomplish these two things (and perhaps a few more that are more subtle). We will start by looking at the most essential tools.

3.1 A buggy program

Let us start by looking at a simple program that has a simple bug. The program is supposed to determine whether two numbers provided by the user are equal. This program compiles clean (no errors, no warnings) but it does not behave as expected. The Figure 4 shows an example of what it does:



```
Enter two integers > 1 3
They are equal!
Press any key to continue . . .
```

Figure 4: Sample run of the buggy program.

Clearly there is something wrong with this program, even though it compiled without any trouble. The program is reporting that 1 and 3 are equal! The code is below. You can replace the code of the earlier example project with this new code, or start a new project and add this code in a new file. Try running the program yourself. Even if you can identify the bug just by looking at the code, do not change anything yet. This is a deliberately simple example, but if this small piece of code was part of a larger program the errors would not be as easy to find.

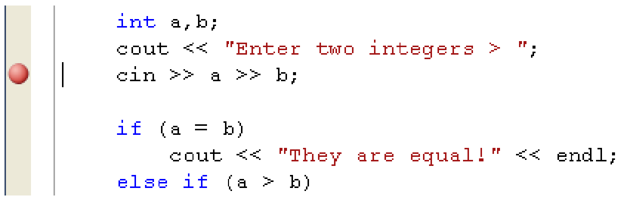
```
#include <iostream>
using namespace std;

int main() {
    int a, b;
    cout << "Enter two integers > ";
    cin >> a >> b;

    if (a = b)
        cout << "They are equal!" << endl;
    else if (a > b)
        cout << "The first one is bigger!" << endl;
    else
```

```
        cout << "The second one is bigger!" << endl;

    system("pause");
    return 0;
}
```

3.2 Breakpoints

The first thing we want to do in our general debugging strategy is to identify which part of the program is responsible for the problem. When you have absolutely no idea, you can just start at the beginning of `main()`. Usually you have some idea of what part of the code may be responsible.

Once you know where you want to start debugging, put a breakpoint there. You can do this by clicking in the gray margin next to a line of code, or pressing F9 while on that line. (Note that there are many ways to activate most debugger functions, but only one or two will be presented here for each command). In this case, let us start just before the user enters the two numbers (Fig. 5).



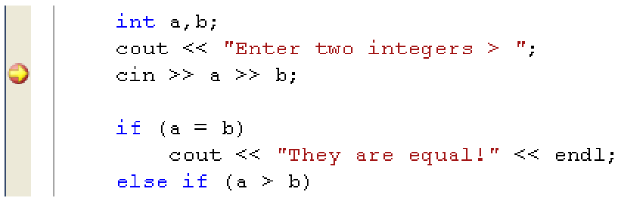
```
int a,b;
cout << "Enter two integers > ";
cin >> a >> b;

if (a = b)
    cout << "They are equal!" << endl;
else if (a > b)
```

Figure 5: A breakpoint.

The visual effect is obvious: a red dot appears in the margin. The meaning is that whenever the program reaches this line of code in the debugger, everything will stop and you will be able to look at the details. It is a simple concept, but it is what makes this such a powerful problem-solving tool.

3.3 Starting the debugger

Visual Studio lets you run your program by selecting Start Without Debugging, but it treats Debug mode as the default. This may cause some frustration as you start setting up breakpoints so you may want to get in the habit of using Start Without Debugging (Ctrl+F5) unless you specifically want to use the debugger. Obviously in this case we do want to use the debugger, so press F5 (or click the green arrow 'Start' button). The output console will appear normally while the debugger starts. Then Visual Studio will be brought back to the foreground, now with a yellow arrow in the margin on top of the red breakpoint dot (Fig. 6).



```
int a,b;
cout << "Enter two integers > ";
cin >> a >> b;

if (a = b)
    cout << "They are equal!" << endl;
else if (a > b)
```

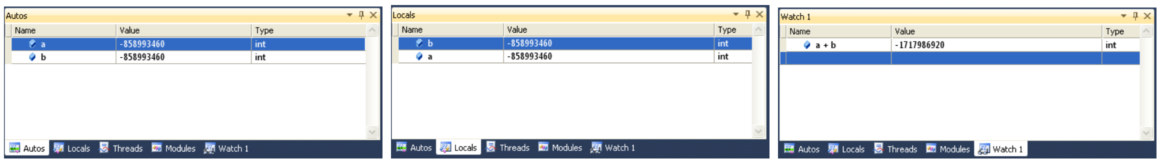
Figure 6: The breakpoint is hit.

This yellow arrow indicates which statement will execute next. Commit that to memory, since it is easy to forget when you are looking at a larger piece of code or complex statements.

3.4 Watching variables

Before we execute that code, we want to make use of the debugger's other primary tool: the ability to look at the contents of variables. The *Autos* window should appear automatically when you start the debugger, but if it does not you can access it from the following menu: `Debug -> Windows -> Autos`.

The *Locals* and *Watch* windows are also available from this menu. The Autos window shows a list of all the variables you are currently using. Locals will show all the variables that are local to the current function (a and b in this case). The Watch windows will let you type in whatever variables or expressions are really interesting to you. They will not contain anything initially, but you can add entries just by typing an expression. The Figure 7 shows an example with the variable a and b, and the watch value a + b.



Autos

Name	Value	Type
a	-858993460	int
b	-858993460	int

Locals

Name	Value	Type
b	-858993460	int
a	-858993460	int

Watch 1

Name	Value	Type
a + b	-1717986920	int

Figure 7: The Autos, Locals and Watch variable windows.

3.5 Stepping

Now we would like to execute the statement that is hit and see what effect it has. The *Step Over* command will do this, and the easiest way to perform a Step Over is by pressing the F10 key. Even if you do not normally use keyboard shortcuts, you will probably find it helpful to be able to concentrate on your code while you are using the debugger, so this particular keyboard shortcut is worth knowing. You can also use the Step Over icon.

Do one step over request. The console will be brought to the foreground so you can enter data. Enter the same values as before in case the bug depends on those particular values. Once you have entered the two numbers, the debugger will come back to the foreground with the yellow arrow advanced on the next code line. There is another change though, that is far more important: the variables in the Autos window have changed. Notice that the values are in red, indicating that they changed when you executed that line. If you are watching many things at once, this helps you figure out which information is new. Since the result of the statement was what we expected, it is unlikely that there is anything wrong. That is the key to this whole process. Since that statement did not seem to cause any problems, execute the next by performing another Step Over (the if condition).

Now b is black again (check in the Local window), meaning it did not change when the last statement executed but a is still red. It changed a second time, and clearly the two numbers are equal now. We found the problem! The key is that the change happened after that this particular line of code was executed, so that line is likely to cause the problem. Of course, when you examine it closely you can see that it should have been == (mathematical equal) instead of = (assignment instruction) and that certainly explains the incorrect results. Stop the debugger by clicking the stop button (or press Shift+F5), then make the change. Run the program normally to see if your correction fixed the bug. Do not remove any breakpoints or Watch variables until you are sure you have solved the problem, otherwise you will just have to do it all over again. This example also shows that the C++ compiler is less picky then, for instance, a Java compiler. In Java, that buggy program would not compile.

3.6 Step Into, Step Out

We have already discussed what the Step Over button does, but there are two more icons on the debugger tool bar that look quite similar. Their functions are related, so let us discuss them now. When you *Step Over* a statement, the entire statement is executed including any function calls. What if you want to step into a function and see what happens inside though? That is where you would use the *Step Into* command (F11 key).

Just like the Step Over icon, which shows the arrow leading completely OVER the code, the Step Into icon shows an arrow leading INTO the code (relate that logic to what the commands do and it will be easy to remember which icon you want).

The *Step Out* command (Shift+F11) tells the debugger to run the current function in its entirety and then stop. Commit that to memory, since you will undoubtedly need it in time: when you step into a function by mistake, use the Step Out command to effectively undo that mistake, or when you are done debugging in called function.

3.7 Conditional Breakpoints

Whenever you have a piece of code that runs more than once, it can create a problem for debugging. Suppose, for example, that you have a loop that works for 100 iterations, but then fails for some reason (this could easily happen if you were processing elements in a large array, for example, or when using recursion). Setting an ordinary breakpoint would not help much, since the breakpoint would be hit every single time through the loop. Visual Studio lets you set breakpoints conditionally so the program will only stop if a particular condition is met. To give a breakpoint a condition, set the breakpoint normally, then right-click and select:

- **Hit Count**. You can tell the breakpoint to trigger only after it has been hit n times. This is the *Hit Count* method. Note that this is a count of how many times the breakpoint itself is encountered, so if the breakpoint is inside an *if* statement, the Hit Count may or may not match the number of iterations of the loop. You can also tell the program to stop when the hit count is anything above n, or even if it is a multiple of n (so you can check on the progress of a loop every 500 iterations for example).
- **Condition**. You can tell the breakpoint to stop the program only if some (arbitrary) condition is met. You might do this if there is a particular scenario you are interested in studying (for example, you might want to observe all situations in a loop where some variable is negative). In the *Breakpoint Condition* window, there are two choices marked *is true* and *has changed*. We use the first when we want to stop the program only under certain circumstances. The second can be used to monitor a variable. If you enter x in the *Condition* field, and check *hasChanged* then the breakpoint would only stop on iterations where the value of x is different from when it was last seen.

3.8 Summary

Ultimately the debugger just gives you the ability to follow the execution of your code closely, and the ability to watch the contents of variables. Although this example was extremely simplified, exactly the same technique applies to programs of any size:

1. Identify a general area of the program that may be at fault;
2. Set a breakpoint at the beginning of that area;
3. Step through your code one line at a time, and watch any interesting variables for changes;
4. Whenever something unexpected happens, scrutinize that line very closely as there is something wrong!

3.9 Practice

The only way you will really learn the debugger is to use it. Introduce errors in your own programs deliberately so you can observe the behavior of the debugger for problems you already understand. Then use the debugger whenever you encounter an error (even a simple error you could probably solve on your own) for the sake of practicing.

4 Exercises

The following series of short exercises serve to get some hands-on experience with programming in C++ with Visual Studio. Although these exercises do not need to be handed in, you should get used to placing comments in the code. Even small programs can be hard to understand when you have not seen them for 2 years.

1. Write a program that asks the user to type the coordinate of two points (in the plane), A and B, and then writes the distance between A and B.

   Tips: Use cin and cout of the `<iostream>` library to read/write date and use the `sqrt` function of the `<cmath>` library for the distance calculation.

2. Write a program that asks the user to type an integer and writes 'You Win' if the value is between 56 and 78 (56 and 78 included). In the other case it writes 'You Lose'.

3. Write a program that creates a table of all characters with ASCII values from 62 ('>') to 125 ('}'), and prints the list of characters.

4. Write a program that asks the user to type a positive integer i and compute u(i) defined as follows: $u(0) = 3$ and $u(n) = 3 \times u(n-1) + 4$.

5. Write a function that takes integers n and k as input and outputs the following binomial coefficient: $f(n,k) = \frac{n!}{(n-k)!k!}$

   Write a main program that asks the user for two integers n and k with n $\geq$ k $\geq$ 0, calls the function and prints the binomial coefficient to the screen. Try to make the function work for as big n as possible.

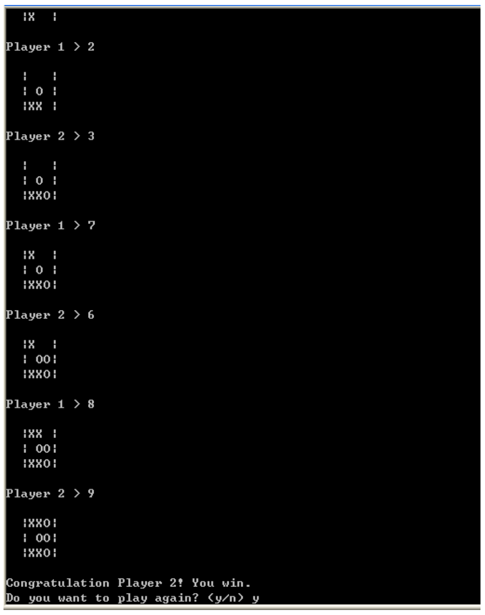
6. Write a program that reads an integer i, and then iterates on the reading of a value j. When j is larger or smaller than i, print the corresponding text and continue to read a new j. When j is equal to i, print a winning text and terminate the program.

5 Tic-Tac-Toe

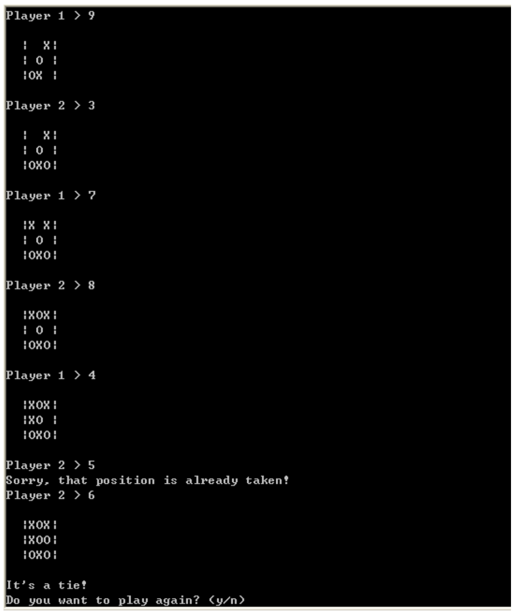
Create a new project in your solution called TicTacToe: right click on your solution, `Add -> New Project...`, select an `Empty Project`. Add a new `main.cpp` file in your `Sources Files` that will contain your code. Make this project the default program to run by selecting `Set as StartUp Project` in the right click menu.

This program will implement the well known Tic-Tac-Toe game. Therefore it should display a `3x3` board, reads the inputs and declares the winner when appropriate (Fig. 8). The program reads the two players' inputs alternatively. The goal is to create a line of three (horizontally, vertically or diagonally). At the end of the game (if one player wins or a tie), the program asks if the players want to play again. If so the game starts again, and the first to play is changed, else the program terminates.

- The board should be represented as a 3x3 (multidimensional) array of `char`.
- Write a function displaying the board.
- Write a function reading player inputs, where the numeric pad (1 to 9) represents the board. Give references to the 2D coordinates as parameters to that function.
- Write a function updating the board according to a player's request and the current board state.
- Write a function checking if a player wins (8 possible configurations).
- Write a reset function that put back the board at its initial state.



```
  ¦X  ¦

Player 1 > 2

  ¦   ¦
  ¦ O ¦
  ¦XX ¦

Player 2 > 3

  ¦   ¦
  ¦ O ¦
  ¦XXO¦

Player 1 > 7

  ¦X  ¦
  ¦ O ¦
  ¦XXO¦

Player 2 > 6

  ¦X  ¦
  ¦ OO¦
  ¦XXO¦

Player 1 > 8

  ¦XX ¦
  ¦ OO¦
  ¦XXO¦

Player 2 > 9

  ¦XXO¦
  ¦ OO¦
  ¦XXO¦

Congratulation Player 2! You win.
Do you want to play again? (y/n) y
```



```
Player 1 > 9

  ¦  X¦
  ¦ O ¦
  ¦OX ¦

Player 2 > 3

  ¦  X¦
  ¦ O ¦
  ¦OXO¦

Player 1 > 7

  ¦X X¦
  ¦ O ¦
  ¦OXO¦

Player 2 > 8

  ¦XOX¦
  ¦ O ¦
  ¦OXO¦

Player 1 > 4

  ¦XOX¦
  ¦XO ¦
  ¦OXO¦

Player 2 > 5
Sorry, that position is already taken!
Player 2 > 6

  ¦XOX¦
  ¦XOO¦
  ¦OXO¦

It's a tie!
Do you want to play again? (y/n)
```

Figure 8: Two executions of the Tic-Tac-Toe board game.

- The pseudo-code for that program is:

```
Display the game welcome message and the instructions
Create an empty Tic-Tac-Toe board
While players want to play
    While no winner and not a tie
        Display the board
        Determine player
        Read player's input
        While invalid input or location already used, read again
        Update the board
    Display the board
    Congratulate the winner or declare tie
Terminates the program
```