**CS253 Laboratory session 6**

**Part 1: Looking at the Microsoft .NET frame work and CIL using a simple C# program**

In this handout we are going to investigate a new programming language called C# (pronounced see-sharp). In a while we will use it to build an event driven applications (this should be straightforward) but for now we will look more closely at its inner workings (we are Software Engineers after all). The aim of this section is to provide a fuller explanation of code found in modern executable files.

C# is a Microsoft language similar to Java from the point of view of the programmer (Object Orientated and similar syntax). Java compiles to an intermediate Class File that consists of byte-code (similar to machine code) that runs on the Java Virtual Machine. An approach called JIT (Just In Time) compiling is done at run time that converts the byte code into processor specific machine code. The byte-code is portable between processors the machine code is not. JIT is an approach to running byte-code that works by compiling and running byte-code in separate smaller blocks at run time. The JIT compiler (which is a component of the Virtual Machine) can "manage" the code as it is run, which guarantees that many run time errors can be caught prior to executing them. If you did compile an entire program to an x86 assembly language prior to running it then errors such as (array bound errors, index out of range, memory leaks) cannot be detected. C++/C can compile directly to x86 assembly language and this is sometimes referred to as "unmanaged code". Unmanaged code does give the programmer closer control of the hardware and memory and so has the potential to run faster but at the cost not detecting certain errors at run time.

There are three approaches to running source code,

- Full compilation to native machine code, ahead-of-time (AOT) (e.g. MASM, C++, C),
- Line by line interpretation runs on a Virtual Machine (e.g. Python, Basic, MatLab),
- Just In Time running on a Virtual Machine with integrated JIT compiler (e.g.Java, C#).

**A comment:** As software engineers you should be able to get to a stage where you can program in any language (or convert to a new one quickly) but also be aware of how you would choose the most suitable language for a specific problem. Consider the analogy with driving, you would probably answer the question to "Can you drive?" with the response "Yes", it would be unlikely that you would say "Only a Nissan Micra". Furthermore you would be able to comment on which vehicle you would use for a specific task, e.g. Ford SMax for motorway driving, Toyota Prius for low emissions or Bugatti Veyron Super Sport for speed. The same applies to operating

systems (OS); I use both Linux and Windows for research on assessing driver behavior, being closed to one or other OS would impede development.

The C# compiler converts your code to CIL Common Intermediate Language which is an object oriented byte code (similar to machine code) associated with the .NET framework. This step is used by Microsoft so as to provide interoperability between languages (e.g. Visual Basic, C#, Mono, C++/CLI, F#). The CIL is just in time compiled by the CLR (Common Language Runtime compiler). The CIL code is embedded in the executable file.

Create a new C# console application called CIL,

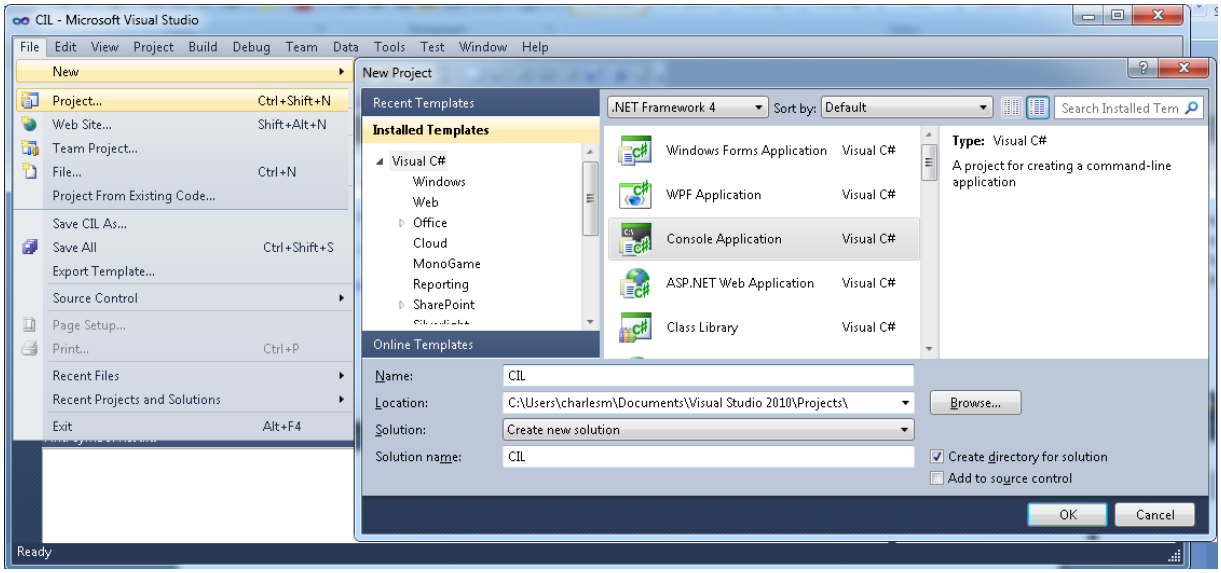


**Figure 1** Creating a C# Console Application called CIL

Enter the following short program that adds two numbers,

```
namespace CIL
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 10, b = 10, c;
            c = a + b;
            Console.WriteLine(c.ToString());
            Console.ReadKey();
        }
    }
}
```

compile and run the program in Debug mode.

Find the ildasm (intermediate language disassembler) and start it by double clicking on the Icon in Windows Explorer. Use this program to open the CIL.exe (executable) file you created. You can launch the Windows explorer by holding down the Windows Key and ***E***.

*Ildasm.exe* is located at the following path on my laptop,

C:\Program Files\Microsoft SDKs\Windows\v7.0A\bin\NETFX 4.0 Tools

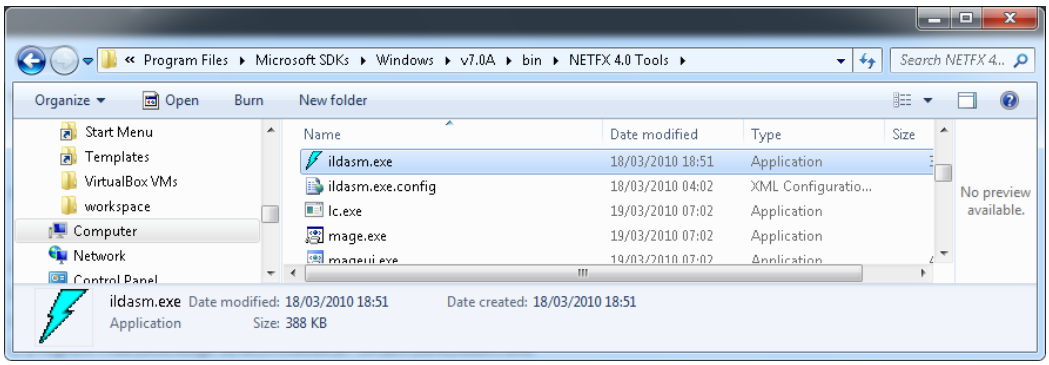


**Figure 2** Find and run the intermediate language disassembler, *ildasm.exe*

Use ildasm.exe to open your CIL.exe file.

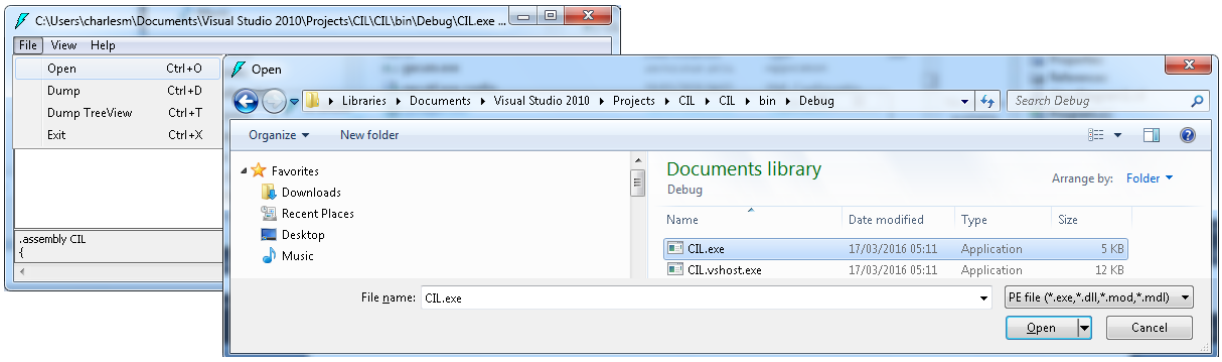


**Figure 3** Open the executable file CIL.exe

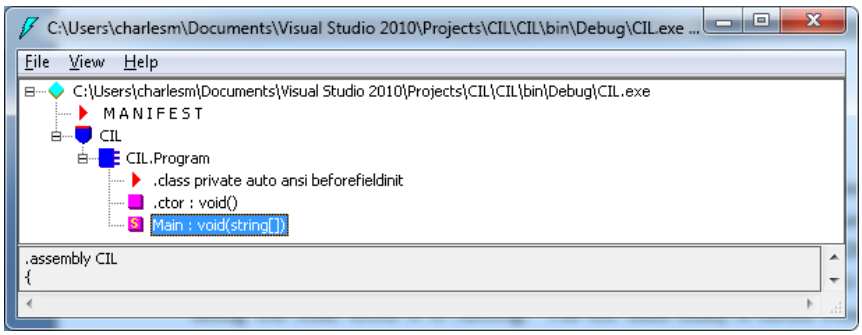


**Figure 4** Double click on the Main:void(string[])

Once you have opened *main()* within CL.EXE you should now be able to see the CLR byte code. It may come as a surprise to see a windows executable that does not contain machine code; it contains CLR byte-code called CIL.

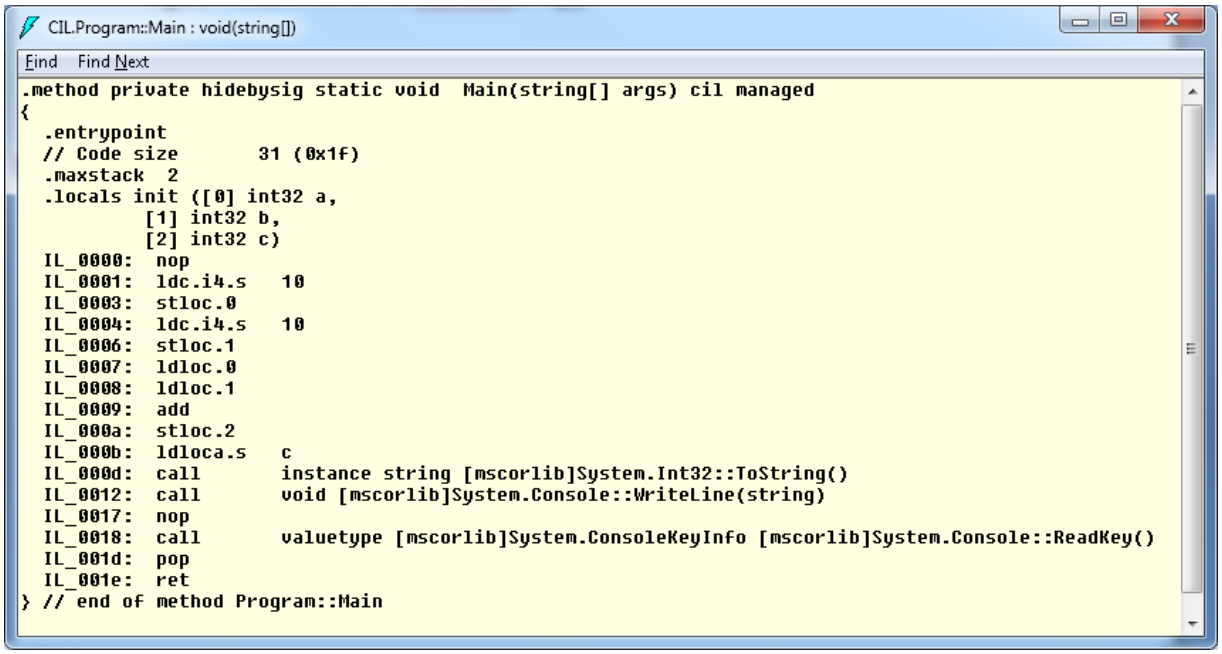


**Figure 5** CIL listing for program evaluating c=a+b.

The code for the addition can be seen within the following fuller listing, you can see that CIL is Object Orientated (. dot operator) and stack based. Again our experience with Assembly Language has given us an ability to understand this very modern byte-code, line numbers are equivalent to instruction addresses, Opcodes are equivalent to Machine Code and Instructions are equivalent to Assembly Language mnemonics.

A full listing of CIL instructions is available at the following location…

https://en.wikipedia.org/wiki/List_of_CIL_instructions

You could if you wish use *ildsam* to view code statistics and to turn on viewing of additional information such as op-code hex values. The folder containing *ildasm* is part of the VS2010 installation contains many other diagnostic programs.

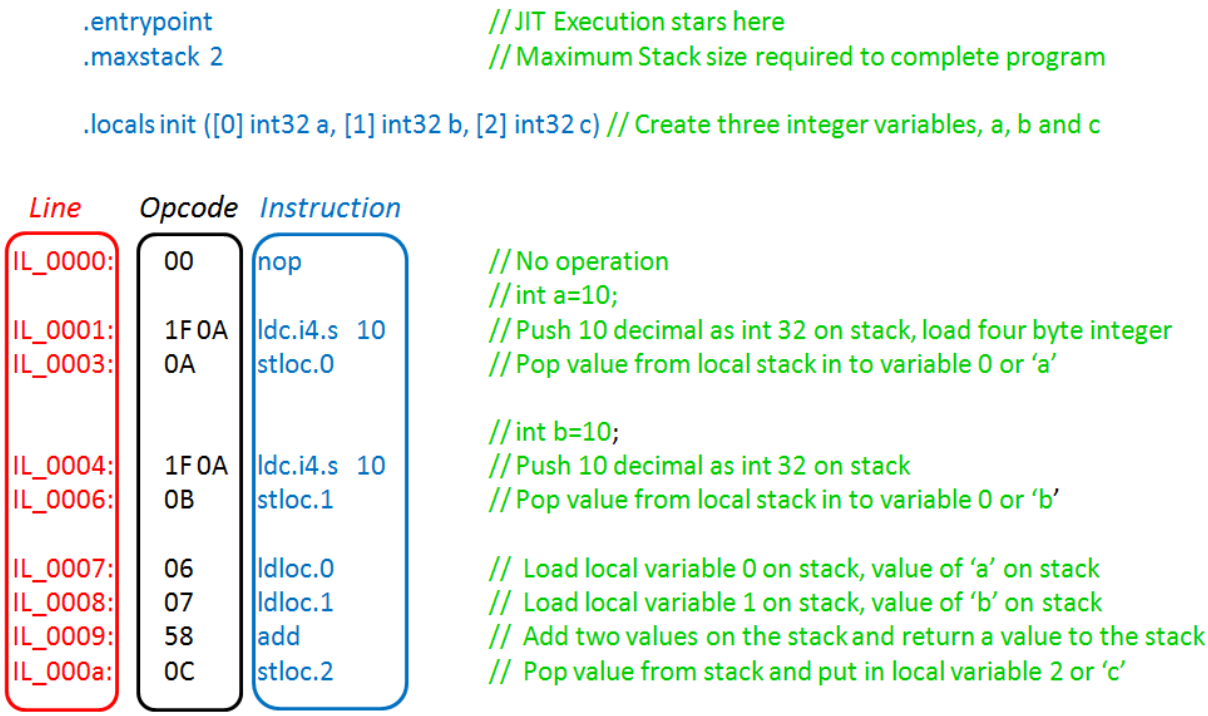


**Figure 6** CIL listing for a program evaluating c=a+b created using *ildasm.exe*.

**Aside:** I thought as you have used Java that I would repeat the above process for a Java program that has the same functionality.

The following Java Program is essentially a copy of the C# program used earlier,

```
public class AddEm
{
  public static void main(String[] args)
  {
    int a = 10, b = 10, c;
    c = a + b;
    System.out.print(c);
  }
}
```

When you compile this source code (.java file) it produces a byte code (.class file) that can be disassembled so as to list the byte code as instructions (mnemonics) rather than op-codes (numbers). Again you can see values for A and B being pushed on a stack and then added.

```
javap -c AddEm.class
```

| Line | Opcode | Operation | |
|---|---|---|---|
| 0: | 10 0A | bipush 10 | // Push operand value (10) onto the stack |
| 2: | 3C | istore_1 | // Store integer in local variable array, pos [1] |
| 3: | 10 0A | bipush 10 | // Push operand value (10) onto the stack |
| 5: | 3D | istore_2 | // Store integer in local variable array, pos [2] |
| 6: | 1B | iload_1 | // Local variable at [1] pushed on operand stack |
| 7: | 1C | iload_2 | // Local variable at [2] pushed on operand stack |
| 8: | 60 | iadd | // Two vales on operand stack are added |
| 9: | 3E | istore_3 | // Store integer in local variable array, pos [3] |
| 10: | B2 00 10 | getstatic | #16//Field java/lang/System.out:Ljava/io/PrintStream; |
| 13: | 1D | iload_3 | // Local variable at [3] pushed on operand stack |
| 14: | B6 00 16 | invokevirtual | #22//Method java/io/PrintStream.print:(I)V |
| 17: | B1 | return | |

**Figure 7** Use of javap to convert op-codes in a *.class* file to a list of operations in a text file.

Opening the .class file in the Notepad++ HEX editor allows direct access to the byte code.

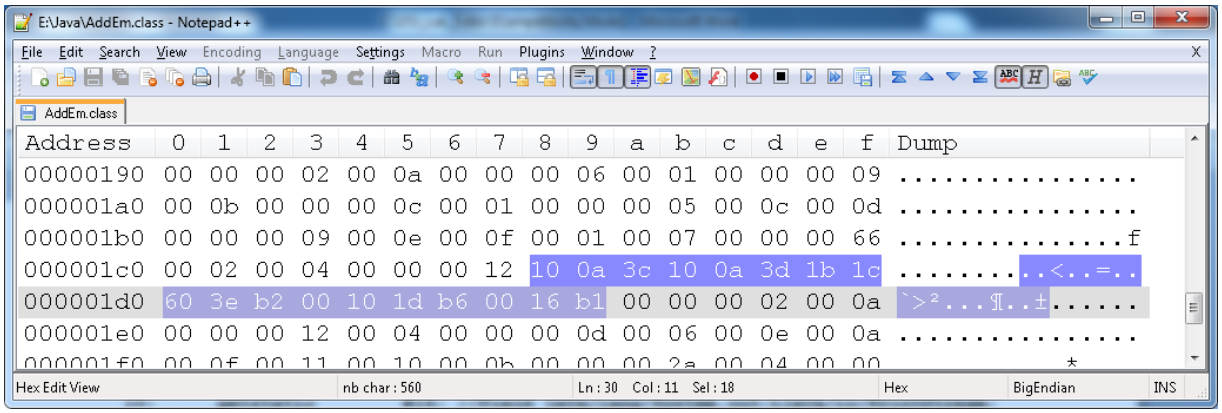


**Figure 8** AddEm.class viewed in Notepad++ hex editor, byte-code located.

The JVM instruction set can be found at the following link,

https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html

**End of aside**

**Structure of a modern .EXE file:** The first two bytes of all executable files are the characters "MZ", these are the initials of Mark Zbikowski one of the developers of MS_DOS. You can see this if you open the file in Notepad++.

Windows/.NET executables are “PE format” files, there is still a MZ start to the file that contains standard assembly code to print “This program cannot be run in DOS mode”, to the screen when run in DOS mode.  If run on a Windows machine the PE format executable will start the CLR machine and then run the CIL code contained within the executable file.

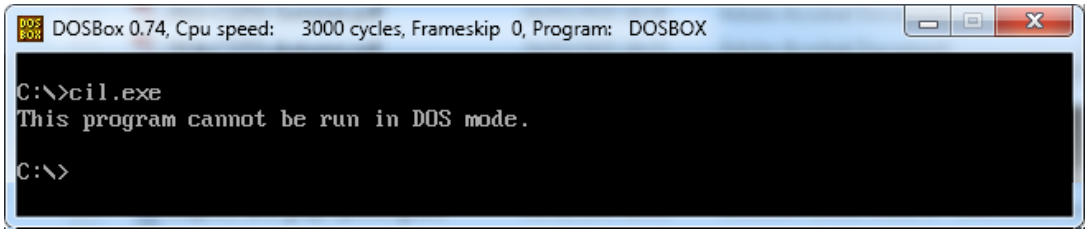


**Figure 9** A Windows PE executable run on a DOS emulator, CIL.EXE.

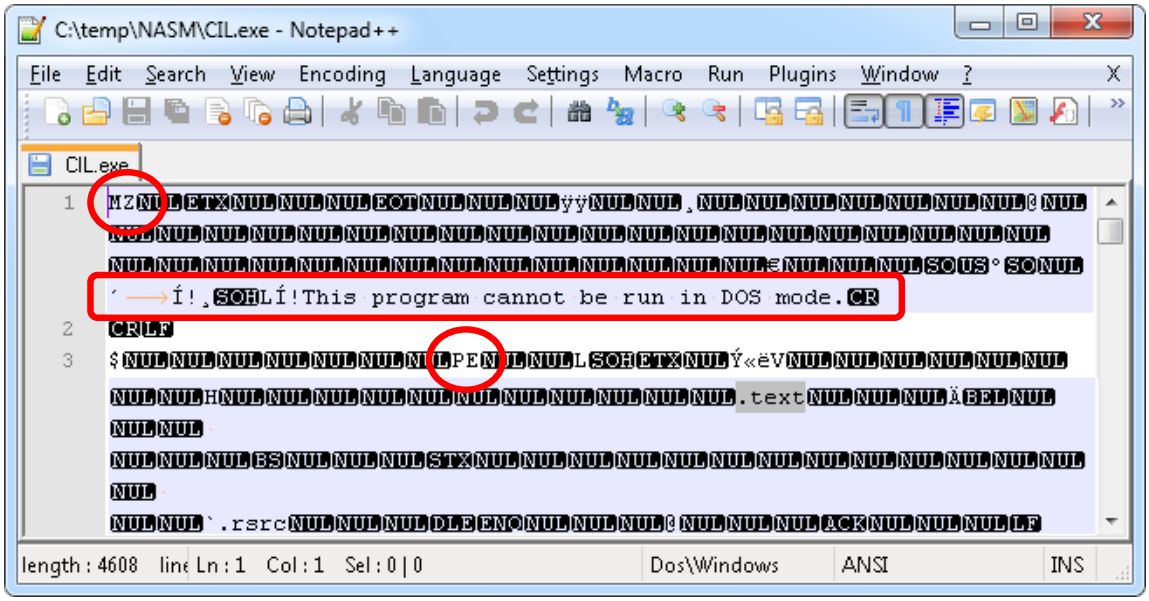


**Figure 10** Open the Windows executable CIL.exe (*a=c+d*) using Notepad++, identified are the location of, “MZ”, DOS message and “PE”.

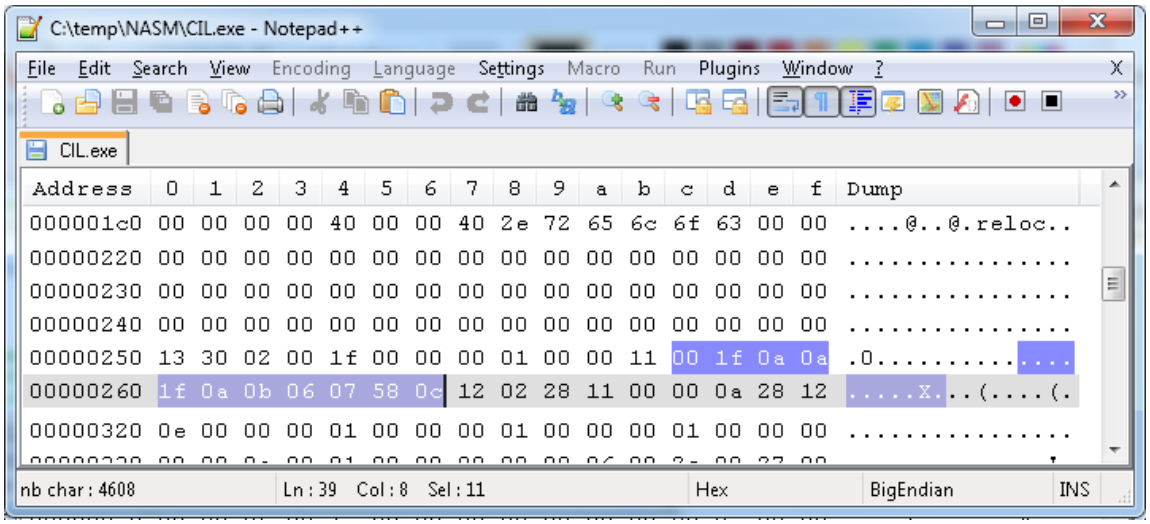


**Figure 11** Open a Windows executable (CIL.exe) in Notepad++ with Hex Editor, location of the CIL code has been identified proving CIL and not assembly language is the code of the executable. This may not be possible on lab machines as the HEX plug-in is not include with lab version of Notepad++.

**A few final observations:** It is possible to run a .NET/CIL binary (executable) on a Linux machine using "Mono", if you want to do C# on Linux/Mac or using OpenSource software then have a look at MonoGame and Mono. Microsoft have made Visual Studio available at their DreamSpark website and also in express edition format.

http://www.mono-project.com/docs/gui/winforms/

**To do:** work your way through this section and then answer the relevant quiz questions.

**A final thought:** Preparing this section has reminded me that you need to keep playing with code and data to learn. Keep asking questions and doing experiments and writing little bits of code to confirm your ideas. In the quiet moments try some coding that is new, slightly bigger and of interest to just you (perhaps Unity, Blender, Maya, ROS, Python, Drupal, MIT App Inventor, Android App programming using Processing, OpenGL, OpenCV) and become an expert in something the rest of us haven't studied as much. Talk about what you are doing and listen to others. Find a problem you would like to code or a platform you would like to know more about and have a go. Keep a hardback laboratory note book and use it to write notes as you go. After a few months you will be surprised by how much you have done. Working on scraps of paper for each separate project does not work, laboratory notebooks are essential.

**Part 2: Using the debugging tools available in the IDE**

Using a debugger is an essential skill when using an IDE. Use your existing CIL Windows Console Application and modify the program so that it adds the integers from 1 to 20.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CIL
{
    class Program
    {
        static void Main(string[] args)
        {
            int total = 0;
            for (int i = 1; i <= 20; i++)
            {
                total += i;
            }
            Console.WriteLine(total.ToString());
            Console.ReadKey();
        }
    }
}
```

Double click on the margin to the left of the line *total+=i*, to add a break point. Run the code and note that the code stops running when you reach the break point. Hover, the mouse pointer, over each variable to see its value. Look at the "locals" window to see more information about each variable. Press F11 to step through the code line by line, or Press F5 to continue until the next break point.

You can right click on the red dot locating the break point and set its properties. For example you could break when it has been hit 5 times or when the condition i==6 has been reached.

Quite often you will add a *printf* or *cout* to code to watch something happening. It is better practice to use the debugger as you are not modifying the code in way that needs undoing before it is considered finished.

Double clicking on a break point will delete it.

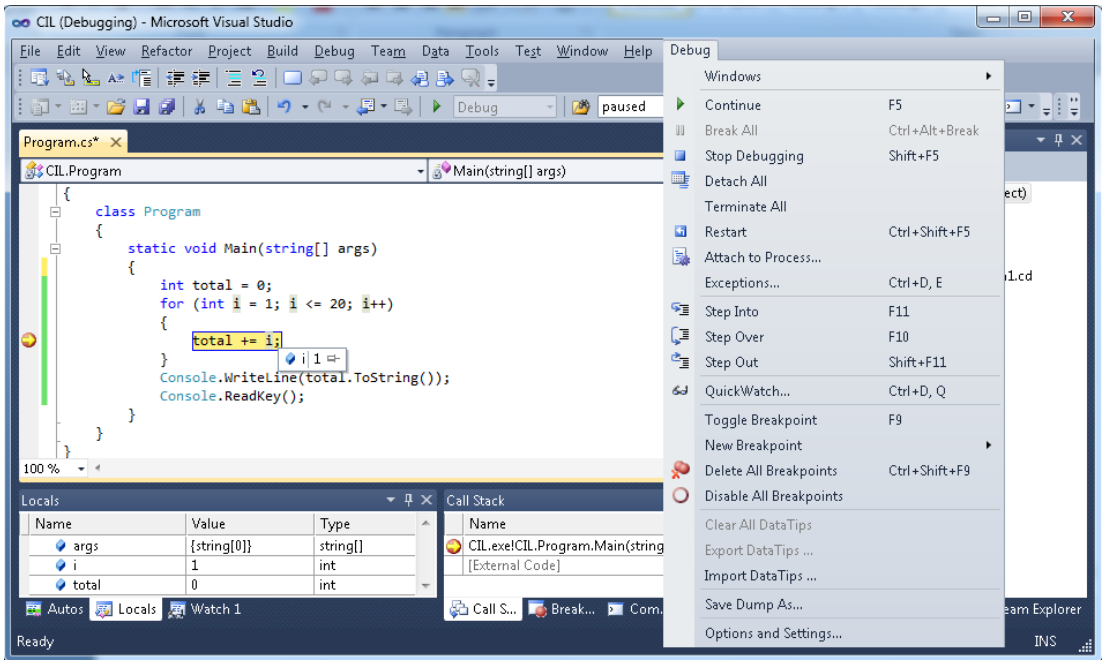


**Figure 12** Debugging: Setting a break point, watching variables and stepping through a program.

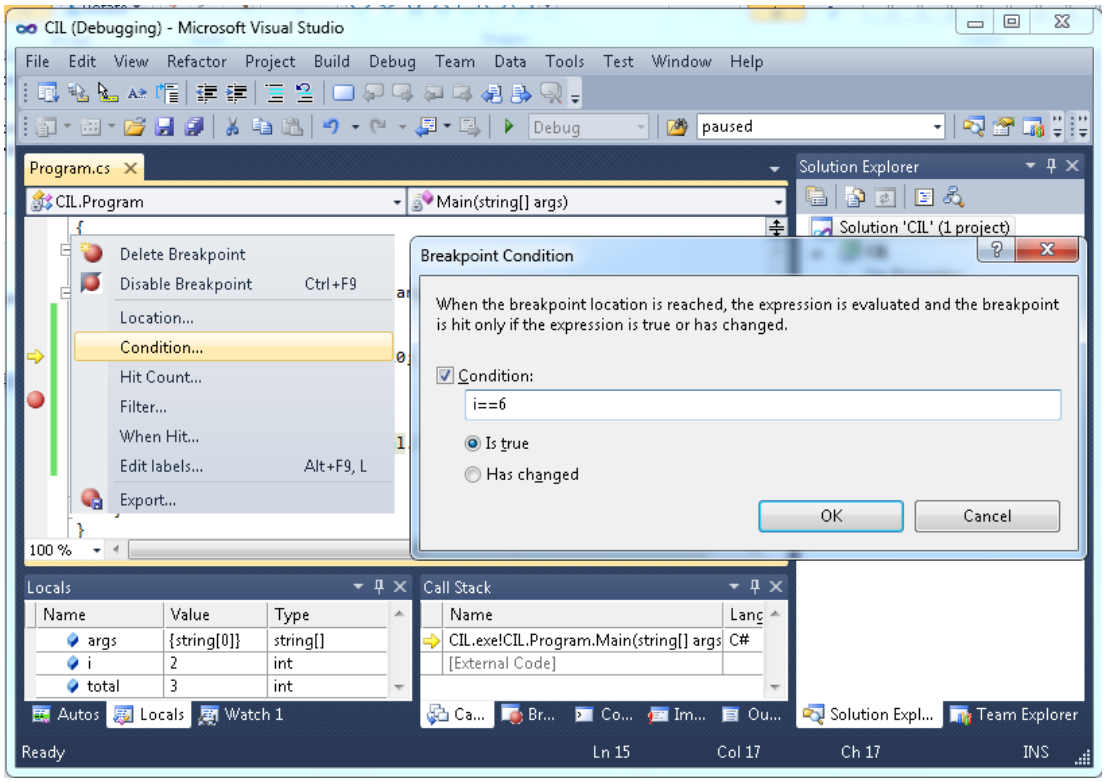


**Figure 13** Debugging: Using conditional break points.

**Part 3: Recreating a classic computer game, Pong as a C# Windows Form Application**

To introduce C# we will create a classic computer game called Pong. First we will need to create a new C# windows forms application called "Pong". Again we will use VS2010 more recent versions should also work.

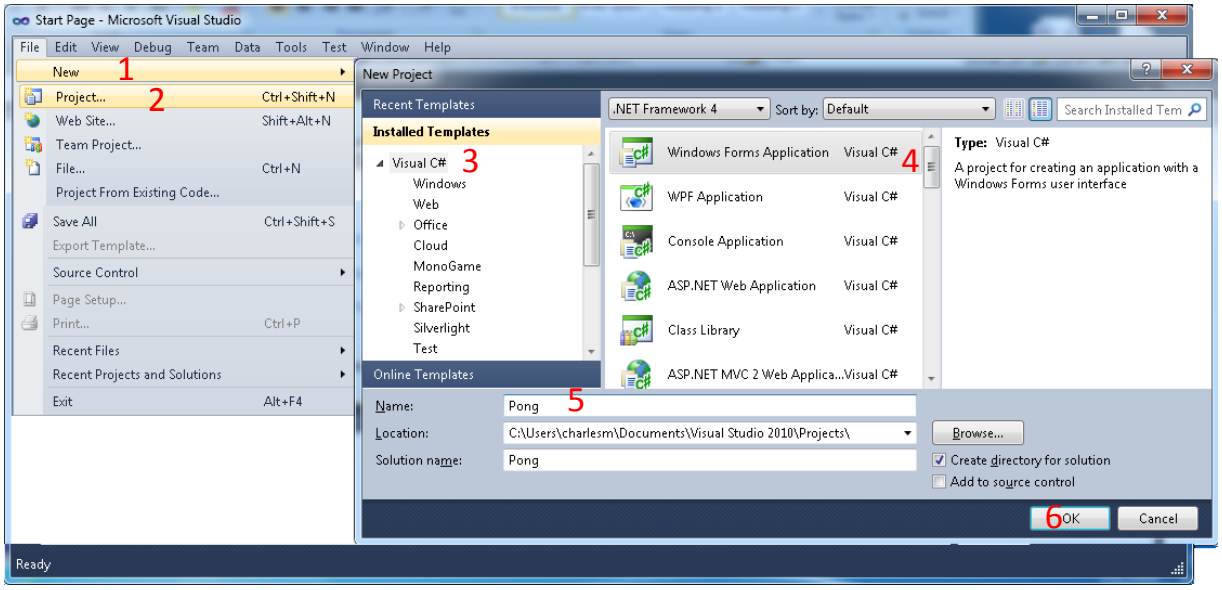


**Figure 14** Create a new C# project of type Windows Form Application called Pong.

From the solution explorer, open the Form1 design window, right click on the form and use the properties dialogue to change its size to about 740x340 pixels. Change the text field to "Pong". Use the toolbox to add two labels to the form, label 1 on the left and label 2 on the right. Change the label properties to use an Arial Narrow 48 pt font and change the text to read "0". Add a picture box located at (90, 20) and of size 512x256 with a border style of "Fixed Single". In each case right click on each item and then select properties to change them.

Double click on timer1 on the form designer so as to add a "tick" event manager. This method will be called every 10 milliseconds (about 100 times a second). This method forms our game loop in which we can advance the game and check the players input.

A potential pitfall: Remember if the Toolbox is not present then the program is probably still running so Debug then Stop debugging, or if this doesn't work try Windows then Reset Windows Layout.

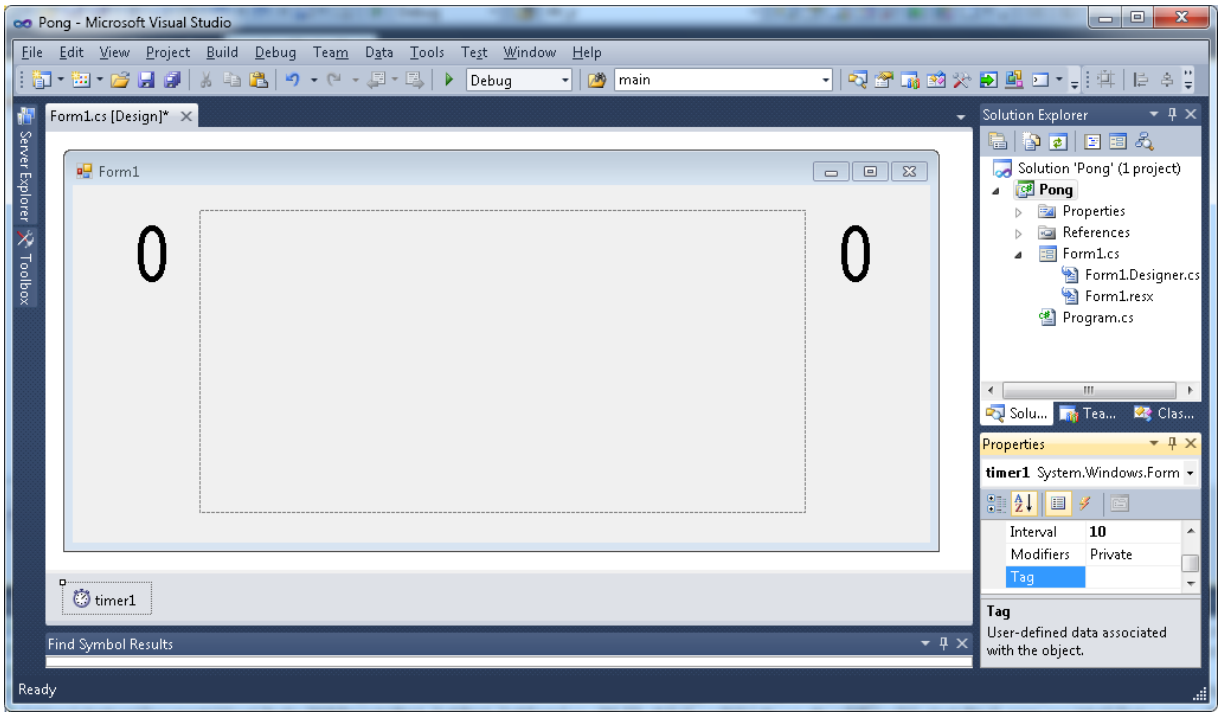


**Figure 15** Use the form designer to create two labels (Arial 48pt "0"), a picture-box and timer (Enabled with an interval of 10).

We will create variable to store the ball position of type "Point" which contains two integers X and Y. Add the following code (not the grey highlighted code), which declares the Ball, creates an instance of it and then draws a red circle at this point on the picture box.

```
namespace Pong
{
    public partial class Form1 : Form
    {
        Point Ball;                     // Instance of the point object called Ball, contains position

        public Form1()
        {
            InitializeComponent();
            Ball = new Point(25, 13);   // Set starting position of ball
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            Bitmap image = new Bitmap(pictureBox1.Width, pictureBox1.Height);
            Graphics g = Graphics.FromImage(image);
            g.FillEllipse(new SolidBrush(Color.Red), Ball.X - 10, Ball.Y - 10, 20, 20);
            g.Dispose();
            pictureBox1.Image = image;
        }
    }
}
```

A potential pitfall: If you did not create the event manager for the timer then you won't see the timer1_Tick() method in which to paste your code. Alternatively you copied the entire code over and you have a timer1_Tick() method but no event manager setup in Form1.Designer.cs to call it. Another fault is to create a timer event manager after you have copied the entire code across, this will produce an event manager called timer1_Tick_1(), to correct this you will need to open Form1.Designer.cs and change the event manager line to read as follows

```
this.timer1.Tick += new System.EventHandler(this.timer1_Tick);
```

Compile and run the code and verify that it creates a red ball on screen.

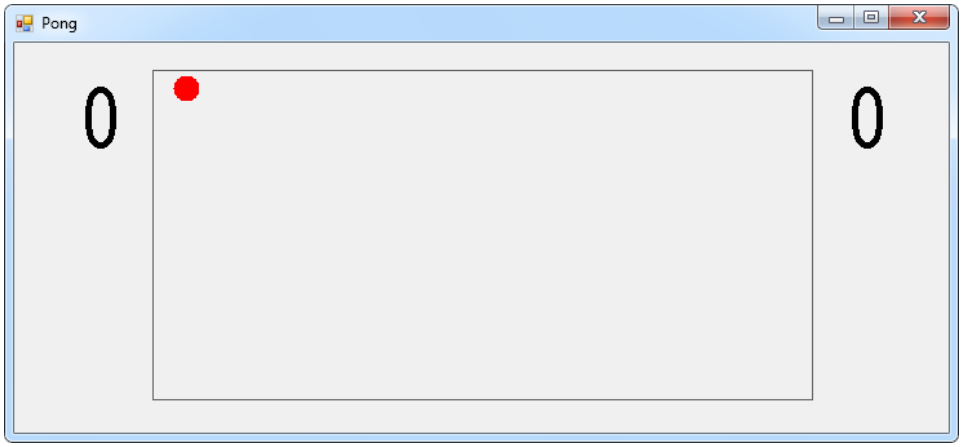


**Figure 16** Run time display of ball on form.

To make the ball move with a diagonal motion we need to simultaneously make it move in the X and Y direction. The direction variable contains the size of advance that the ball makes each time step, in this case 3 pixels in both the X and Y direction. When the ball reaches the side walls we need to reverse the direction of motion so as to cause a bounce effect.

Add the following un-grayed lines of code to your program and re run it. You should see the ball bouncing around the screen. Adjust the direction values and timer duration to obtain a fast but smooth movement of the ball.

```
    public partial class Form1 : Form
    {
        Point Ball;                     // Ball position
        Point Ball_Direction;           // Direction of motion both X and Y

        public Form1()
        {
            InitializeComponent();
            Ball = new Point(25, 13);         // Starting position of ball
            Ball_Direction = new Point(3, 3); // move diagonal top right to bottom left,
        }

        private void timer1_Tick(object sender, EventArgs e)
        {
            Ball.X += Ball_Direction.X;
            if ((Ball.X > 511)&&(Ball_Direction.X>0))   // Ball hit back wall travelling left
            {
               Ball_Direction.X = -Ball_Direction.X;    // Reverse horizontal motion
            }

            if ((Ball.X < 1) && (Ball_Direction.X < 0)) // Ball hit back wall travelling right
            {
               Ball_Direction.X = -Ball_Direction.X;    // Reverse horizontal motion
            }

            Ball.Y += Ball_Direction.Y;
            if ((Ball.Y > 255) && (Ball_Direction.Y > 0)) // Ball hit bottom wall travelling down
            {
               Ball_Direction.Y = -Ball_Direction.Y;      // Reverse horizontal motion
            }

            if ((Ball.Y < 1) && (Ball_Direction.Y < 0))   // Ball hit top wall travelling down
            {
              Ball_Direction.Y = -Ball_Direction.Y;       // Reverse horizontal motion
            }

            Bitmap image = new Bitmap(pictureBox1.Width, pictureBox1.Height);
            Graphics g = Graphics.FromImage(image);
            g.FillEllipse(new SolidBrush(Color.Red), Ball.X - 10, Ball.Y - 10, 20, 20);
            g.Dispose();
            pictureBox1.Image = image;
        }
    }
}
```

Now that we have a ball bouncing around the screen we need to create some paddles. The vertical positions of the left and right paddles require storing in separate variables. Add the following lines of code and re-run the program to check everything is progressing well.

```
        Point Ball_Direction;           // Direction of motion both X and Y
        int   Paddle_Left_Y = 128;
        int   Paddle_Right_Y = 128;
        public Form1()
        …
            Graphics g = Graphics.FromImage(image);
            g.FillEllipse(new SolidBrush(Color.Red), Ball.X - 10, Ball.Y - 10, 20, 20);
            g.FillRectangle(new SolidBrush(Color.Blue), 10, Paddle_Left_Y - 50, 10, 100);
            g.FillRectangle(new SolidBrush(Color.Blue), 490, Paddle_Right_Y - 50, 10, 100);
            g.Dispose();
            pictureBox1.Image = image;
```

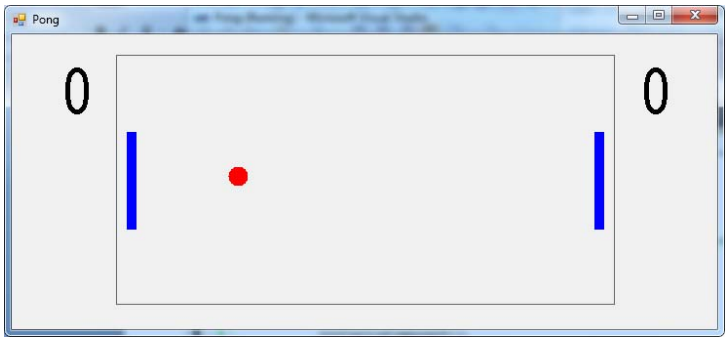


**Figure 17** Run time display of ball and two paddles on the form.

Moving the paddles requires keyboard input. You might consider using an event handler such as KeyDown or KeyPress associated with form1. However this is not very effective as the there is a delay between the first key being detected and the second and even after this the key repeat rate limits the speed of the paddle. The solution is to use a direct call to the Windows API to obtain the keyboard state. This will allows you to detect which keys are pressed at any moment in time. To create this new *IsKeyPushedDown()* method add the following lines of code to the start of your program.

```
using System.Windows.Forms;
using System.Runtime.InteropServices;  // For keyboard

namespace Pong
{
    public partial class Form1 : Form
    {
        [DllImport("user32.dll")]
        static extern ushort GetAsyncKeyState(int vKey);
        public static bool IsKeyPushedDown(Keys keyData)  // For fast keyboard input
        {
            return 0 != (GetAsyncKeyState((int)keyData) & 0x8000);
        }

        Point Ball;
```

We will use keys W and X to move the left paddle and P and M to move the right paddle. The top left hand corner of the screen is (0,0) so increasing the paddle position value moves it down. The paddle positions are clamped so they don't disappear from the screen.

Add the following lines of code to your program,

```
private void timer1_Tick(object sender, EventArgs e)
        {
            if (IsKeyPushedDown(Keys.W)) Paddle_Left_Y -= 10;
            if (IsKeyPushedDown(Keys.X)) Paddle_Left_Y += 10;
            if (IsKeyPushedDown(Keys.P)) Paddle_Right_Y -= 10;
            if (IsKeyPushedDown(Keys.M)) Paddle_Right_Y += 10;

            if (Paddle_Left_Y < 5) Paddle_Left_Y = 0;  // Clamp paddle positions
            if (Paddle_Left_Y > 251) Paddle_Left_Y = 251;
            if (Paddle_Right_Y < 5) Paddle_Right_Y = 0;
            if (Paddle_Right_Y > 251) Paddle_Right_Y = 251;

            if (IsKeyPushedDown(Keys.Escape)) Application.Exit();
```

Run the code again and check that you can move both paddles simultaneously; pressing <escape> will cause the program to terminate.

We now need to detect a collision between the ball and paddle which will result in the ball bouncing. If the ball hits an end wall then the score should be increased and the game paused until the next serve. The following code will cause the ball to bounce off the paddles,

```
            if (IsKeyPushedDown(Keys.Escape)) Application.Exit();

            // Ball/Paddle collision detection on left paddle
            if ((Ball.X < 30) && (Math.Abs(Ball.Y - Paddle_Left_Y) < 50) && (Ball_Direction.X < 0))
            {
                Ball_Direction.X = -Ball_Direction.X;
            }

            if ((Ball.X > 482) && (Math.Abs(Ball.Y - Paddle_Right_Y) < 50) && (Ball_Direction.X > 0))
            {
                Ball_Direction.X = -Ball_Direction.X;
            }
            Ball.X += Ball_Direction.X;
```

We introduce a new Boolean variable that is used to control the advance of the ball. The ball advances following the pressing of <space bar> which is used to serve. The ball pauses when it hits a back wall. Rather than bounce of the back ball the winning player should be allowed to re-serve.

```
int   Paddle_Right_Y = 128;
bool  Game_paused = true;  // Game paused at start waiting for serve

public Form1()
...

if (IsKeyPushedDown(Keys.M)) Paddle_Right_Y += 10;

if (IsKeyPushedDown(Keys.Space)) Game_paused = false; // Start ball motion following a <space>

if (IsKeyPushedDown(Keys.Escape)) Application.Exit();
...
if (!Game_paused)                                  // Do not move ball if game paused
{
    Ball.X += Ball_Direction.X;
    if ((Ball.X > 511)&&(Ball_Direction.X>0))     // Ball hit back wall travelling right
    {
        // Ball_Direction.X = -Ball_Direction.X; // Remove or "comment out" this line
        Ball.X = 30;                              // Restart ball in front of paddle
        Ball.Y = Paddle_Left_Y;
        Game_paused = true;
    }

    if ((Ball.X < 1) && (Ball_Direction.X < 0))  // Ball hit back wall travelling left
    {
        // Ball_Direction.X = -Ball_Direction.X; // Remove or "comment out" this line
        Ball.X = 480;                             // Restart ball in front of paddle
        Ball.Y = Paddle_Right_Y;
        Game_paused = true;
    }
    ...
} // end of condition
```

Run the code and press the space bar to release the ball, check that the ball bounces off the paddles but that it is repositioned correctly when it hits a back wall.

Now we need to introduce a scoring mechanism, the left players score should increase each time the right player fails to return the ball (and vise versa). These scores can be displayed by changing the text associated with the two labels we added earlier. The following code shows you how this can be implemented.

```
                if ((Ball.X > 511)&&(Ball_Direction.X>0))   // Ball hit back wall traveling right
                {
                    // Ball_Direction.X = -Ball_Direction.X;
                    Ball.X = 30;                            // Restart ball infront of paddle
                    Ball.Y = Paddle_Left_Y;
                    Game_paused = true;
                    Left_score++;
                }

                if ((Ball.X < 1) && (Ball_Direction.X < 0)) // Ball hit back wall traveling left
                {
                    // Ball_Direction.X = -Ball_Direction.X;
                    Ball.X = 480;                           // Restart ball infront of padd
                    Ball.Y = Paddle_Right_Y;
                    Game_paused = true;
                    Right_score++;
                }
                ...

            g.Dispose();
            label1.Text = Left_score.ToString("0.#");
            label2.Text = Right_score.ToString("0.#");
            pictureBox1.Image = image;
        }
```

Finally we can add some sound to the game. Add the two sound files supplied to the Debug folder of your project, the one containing Pong.exe. If you don't have headphones then you could skip this bit.

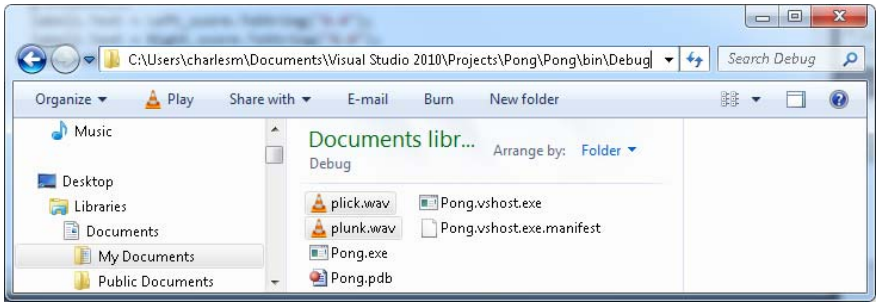


**Figure 18** Adding two files that play "plick" and "plunk" sounds.

Then add the following code to your program.

```
using System.Windows.Forms;
using System.Media;                     // For Sounds
using System.Runtime.InteropServices;  // For keyboard
…
    public partial class Form1 : Form
    {
        SoundPlayer plick = new SoundPlayer(@"plick.wav");
        SoundPlayer plunk = new SoundPlayer(@"plunk.wav");

        [DllImport("user32.dll")]
…
            if ((Ball.X < 30) && (Math.Abs(Ball.Y - Paddle_Left_Y) < 50) && (Ball_Direction.X < 0))
            {
                Ball_Direction.X = -Ball_Direction.X;
                plick.Play();
            }

            if ((Ball.X > 482) && (Math.Abs(Ball.Y - Paddle_Right_Y) < 50) && (Ball_Direction.X > 0))
            {
                Ball_Direction.X = -Ball_Direction.X;
                plick.Play();
            }

            if (!Game_paused) // Do not move ball if game paused
            {
                Ball.X += Ball_Direction.X;
                if ((Ball.X > 511)&&(Ball_Direction.X>0))   // Ball hit back wall traveling right
                {
                    // Ball_Direction.X = -Ball_Direction.X;
                    Ball.X = 30;                            // Restart ball infront of paddle
                    Ball.Y = Paddle_Left_Y;
                    Game_paused = true;
                    Left_score++;
                    plunk.Play();
                }

                if ((Ball.X < 1) && (Ball_Direction.X < 0)) // Ball hit back wall traveling left
                {
                    // Ball_Direction.X = -Ball_Direction.X;
                    Ball.X = 480;                           // Restart ball infront of padd
                    Ball.Y = Paddle_Right_Y;
                    Game_paused = true;
                    Right_score++;
                    plunk.Play();
                }

                Ball.Y += Ball_Direction.Y;
                if ((Ball.Y > 255) && (Ball_Direction.Y > 0)) // Ball hit bottom wall traveling down
                {
                     Ball_Direction.Y = -Ball_Direction.Y;
                     plunk.Play();
                }

                if ((Ball.Y < 1) && (Ball_Direction.Y < 0))  // Ball hit top wall traveling down
                {
                    Ball_Direction.Y = -Ball_Direction.Y;
                    plunk.Play();
                }
            }
```

Finally, we can create a message box declaring the winner as the first person to get to three.

```
            label1.Text = Left_score.ToString("0.#");
            label2.Text = Right_score.ToString("0.#");
            if ((Left_score > 2)||(Right_score > 2))  // Identify a winner
            {
                timer1.Stop();  // Stop game loop
                String text="Right";
                if (Left_score > Right_score) text = "Left";
                DialogResult reply = MessageBox.Show(text+" player wins\rDo you wish to play again?",
"Winner", MessageBoxButtons.YesNo);
                if (reply == DialogResult.Yes)
                {
                    Left_score = 0;     // Reset game values
                    Right_score = 0;
                    Paddle_Left_Y = 128;
                    Paddle_Right_Y = 128;
                    Game_paused = true;
                    Ball.X = 25; Ball.Y = 13;
                    Ball_Direction.X = 3; Ball_Direction.Y = 3;
                    timer1.Start();  // Restart game loop
                }
                else
                {
                    Application.Exit(); // Quit game
                }

            }
            pictureBox1.Image = image;
```


**Figure 19** Finished Pong game.

**A final word:** Complete the quiz questions and submit the Form1.cs file from your Pong project. If you liked using C# for event driven programming then the additional handout prepared for a Summer Camp may be of interest to you.