G52CPP Coursework 3 and 4

Demo Tutorial A

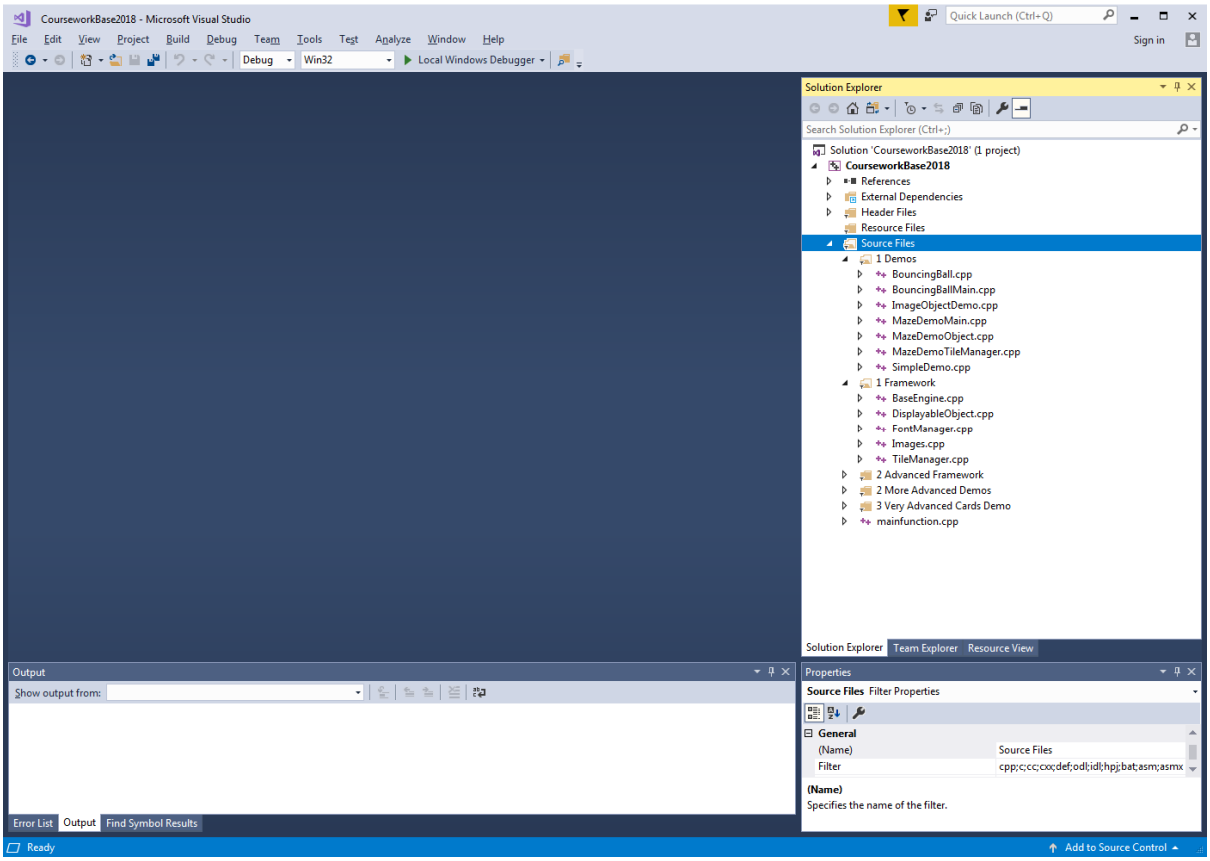
Completing these two demo tutorials should enable you to learn enough of the basics of using the framework to be able to understand the various demos that have been supplied.

Open the Visual Studio project

Unzip the files to a local directory. Double click on the .sln file in the root of the directory structure.

Unzip the project to a directory with write access: Ideally a local directory but it worked from me from my university (windows) home directory. DO NOT USE THE COMPUTER SCIENCE/LINUX HOME DIRECTORY! (Visual Studio does not like having its files on a non-Microsoft filesystem and may give you issues.)

Open the coursework project in visual studio 2017. Double click on the .sln file in the root of the directory structure to open the project.

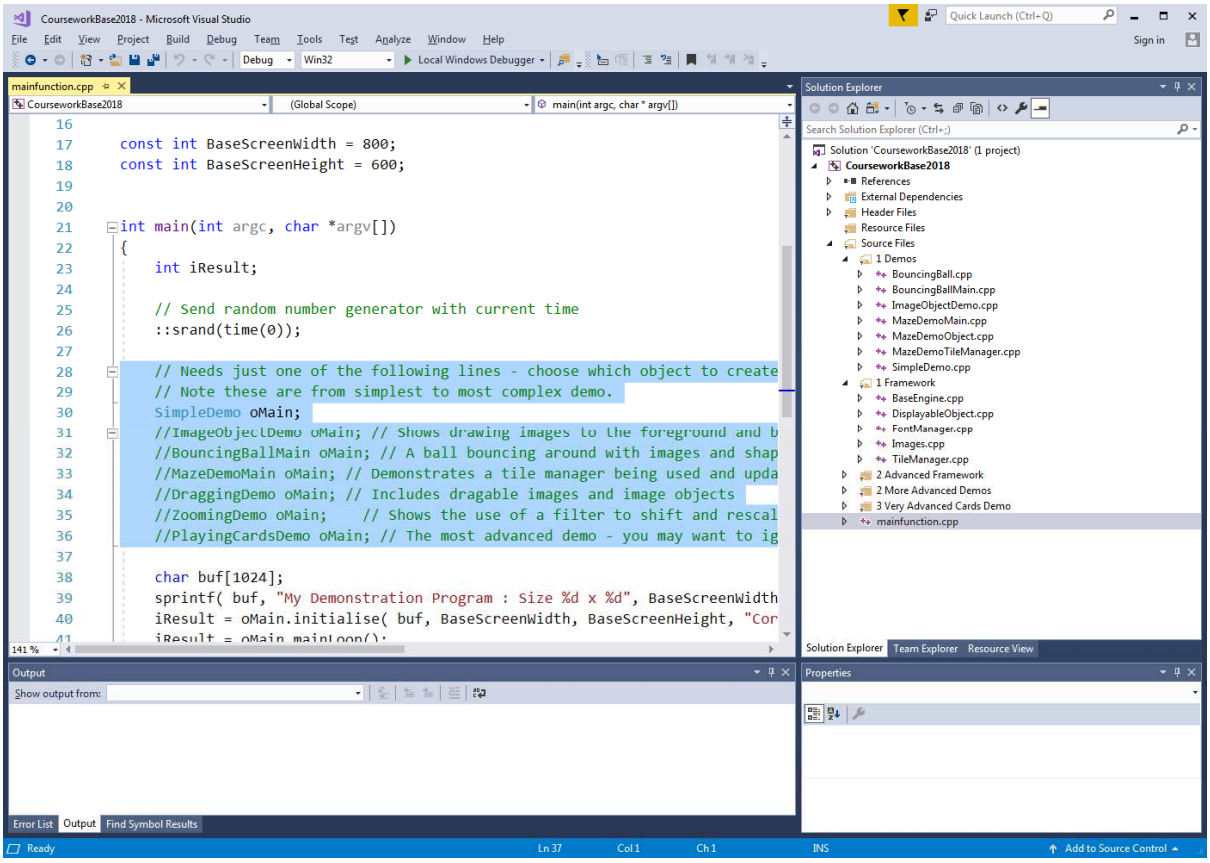


Open up Source Files and you will see some sub-folders. Go through these in numerical order – most are demos and they get more complex later on. Some people may want to just stick to the simpler ones, whereas some may want to investigate more complex versions.

Choose a demo, compile and execute

Double click mainfunction.cpp to open it.

Scroll down to where it creates a top-level object (e.g. SimpleDemo) on the stack:

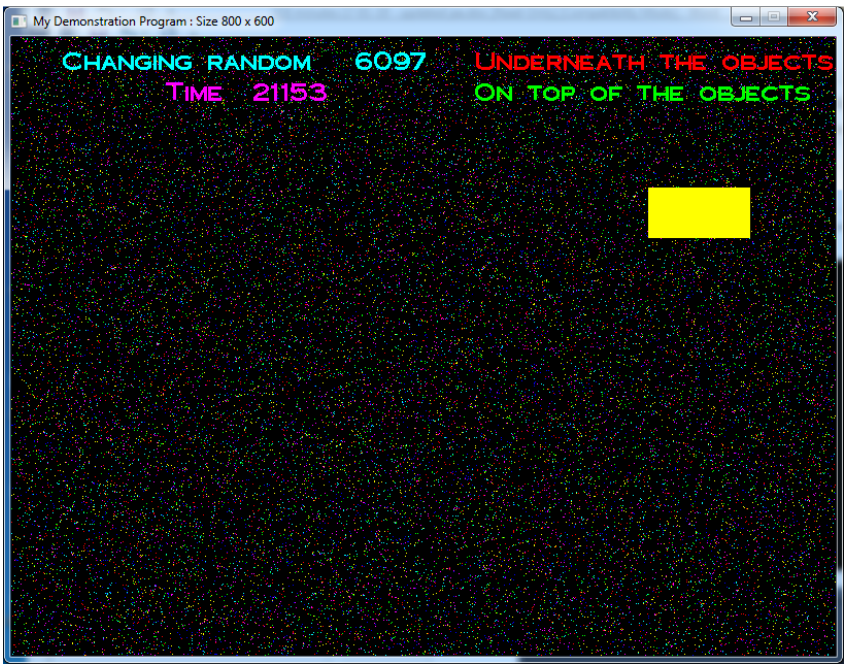


To change demo that is running just change which object is created.

(I will choose SimpleDemo for the moment.)

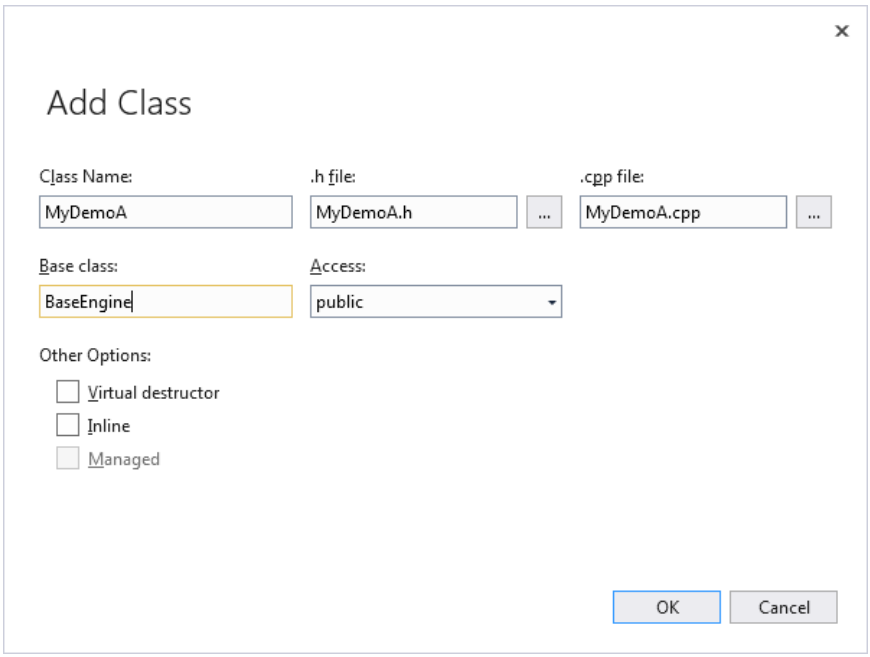
Compile the program (choose ‘Rebuild Solution’) from the Build menu.

Choose to “Start Debugging” under the “Debug” menu, and you will get a program like this, with a moving object:

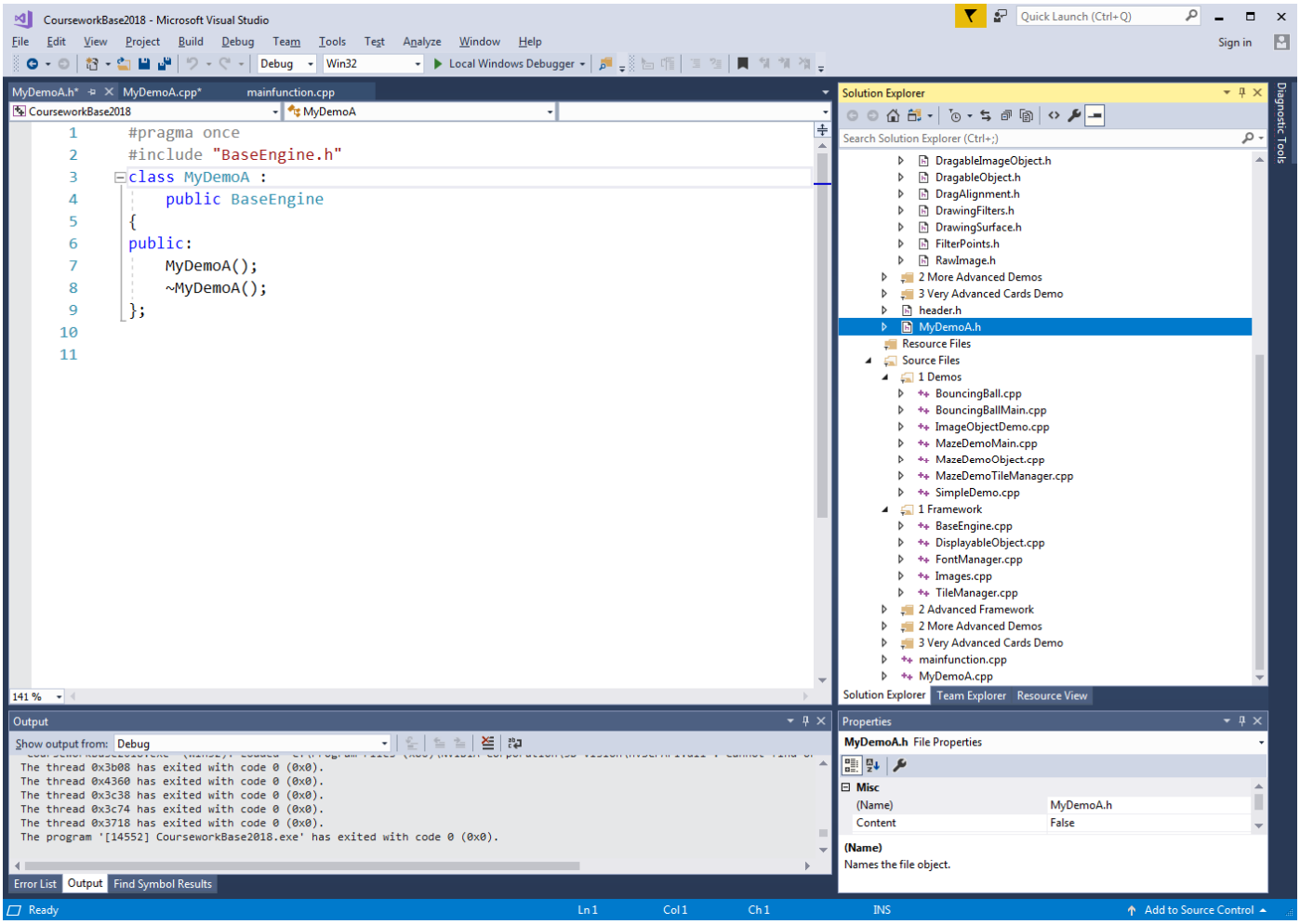


Create your own demo A

Right click on Source Files, and choose Add Class. Choose a class name of MyDemoA (you can call it what you wish but the following instructions assume this name) and a base class of BaseEngine (has to be BaseEngine because you are subclassing an existing class).



This creates a MyDemoA.h and a MyDemoA.cpp file and puts them in the header files and source files filders in your solution explorer:



Further information if you want it:

Your header files will be organised under the “Header files” group and your source files under the “Source Files” group. Within each group I have divided the files into three categories: the uncategorised files (e.g. `header.h`) the Demo files (open these to see the various demos) and the shared files, which are basically the coursework framework. These shared files include the base classes that you will inherit your new classes from. Please feel free to take some time to investigate these files.

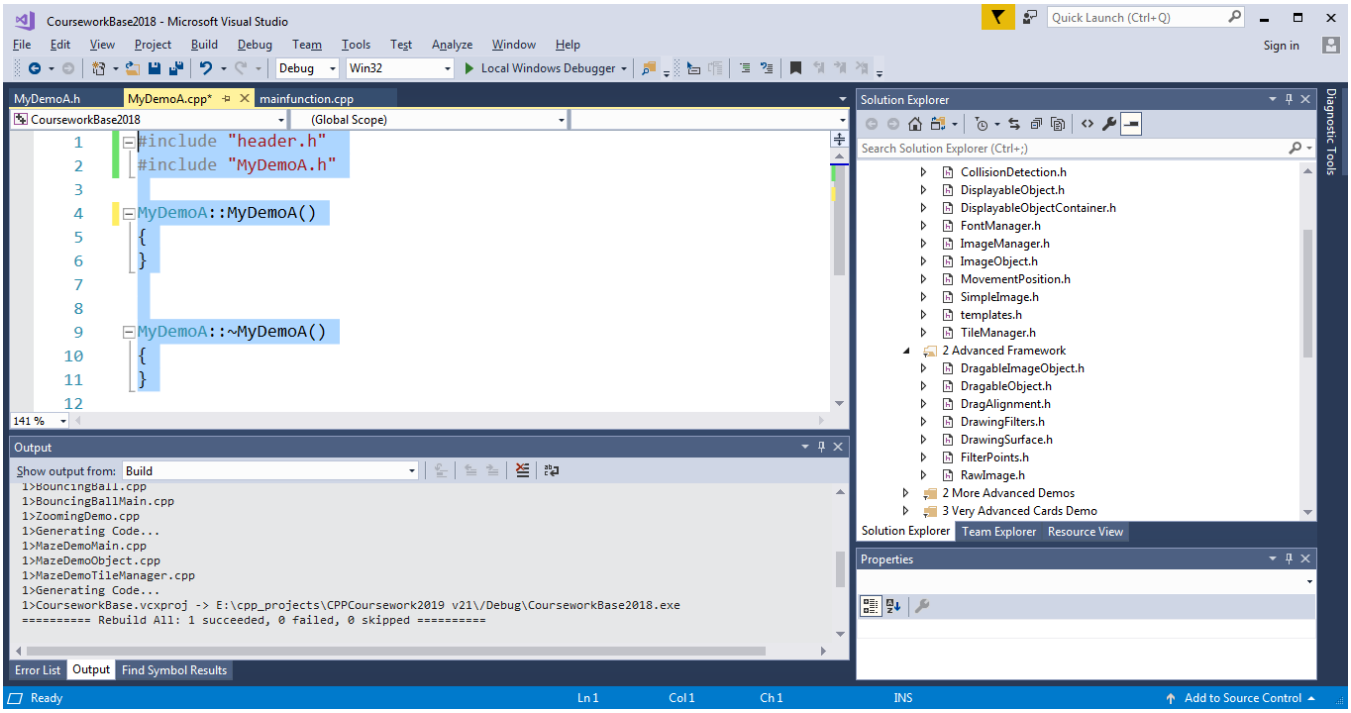
When you use the wizard to create a new class, it will usually put the .h file under Header Files and the `.cpp` file under Source Files (which is probably where you want them). Where the files are categorised is irrelevant: these ‘folders’ have no effect upon the program, they just help you to organise you files within the development tool.

IMPORTANT: Go to the top of your MyDemoA.cpp (only the source file, not the header file) and add a #include for “header.h”. This MUST be the first line in any of your new .cpp files that you create.

SDL header files

Header.h #includes SDL.h.

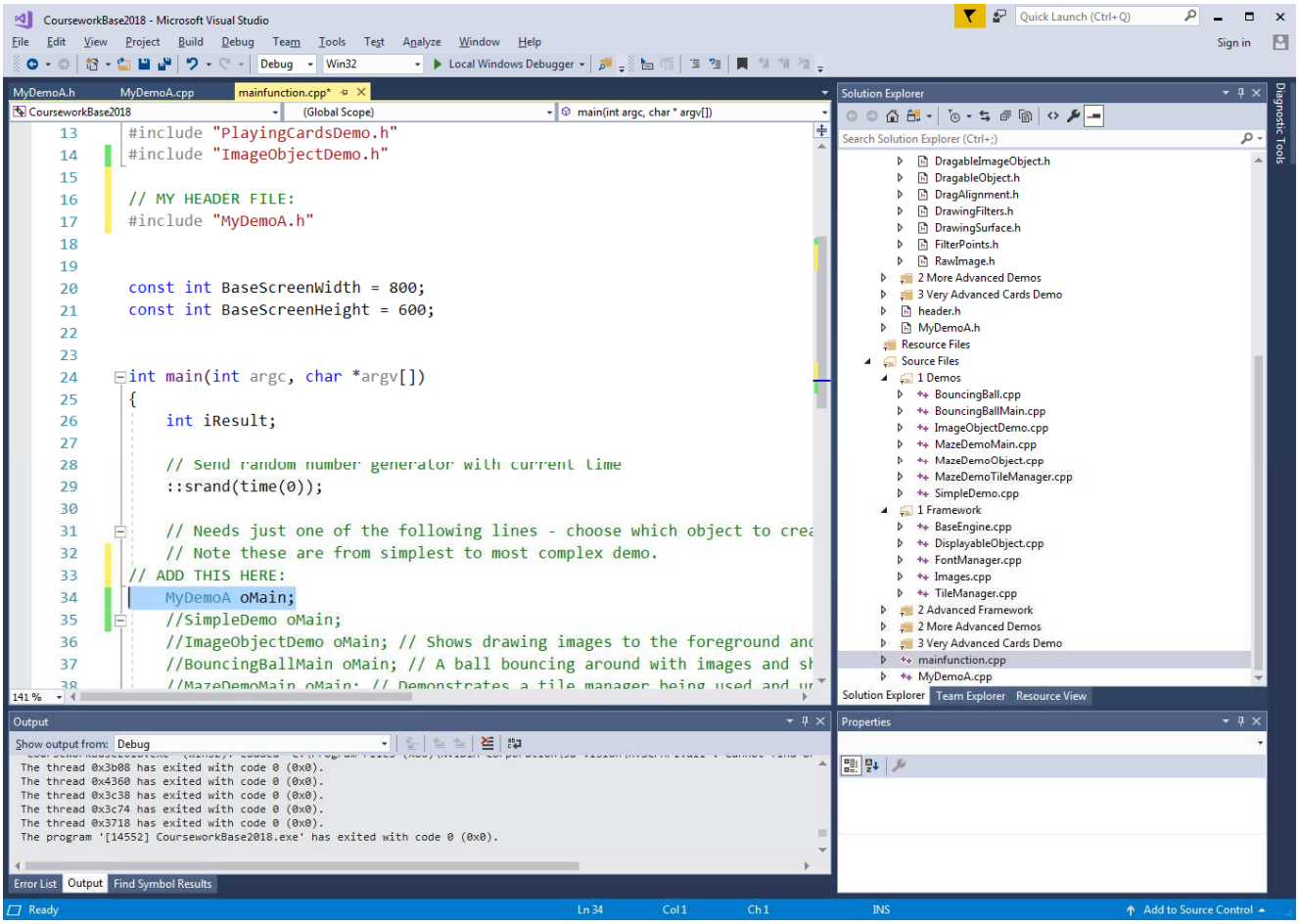
SDL.h adds various #defines to the code to change the names of various functions so that SDL, the graphics library we will use, will work. These need to apply to everything so you need to add this to the very top of every .cpp file you create. If you forget one, you will get a mix of functions with the changes and functions without and probably get a LOT of compilation errors.



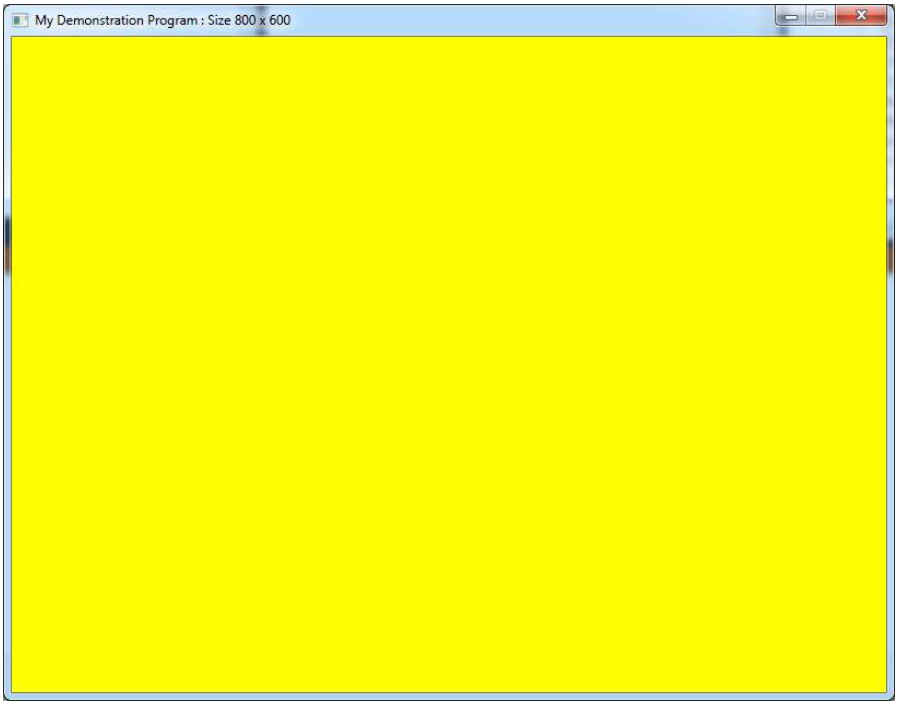
Save everything (shift-ctrl-S is a shortcut for this).

Open the mainfunction.cpp file and:

- #include the MyDemoA.h file at the top.
- Create an object of the new class type (see below), e.g. “MyDemoA oMain”.
- Comment out the object creation of the old type, e.g. “SimpleMain oMain;”



Run the program and a yellow background will be shown for the new program:

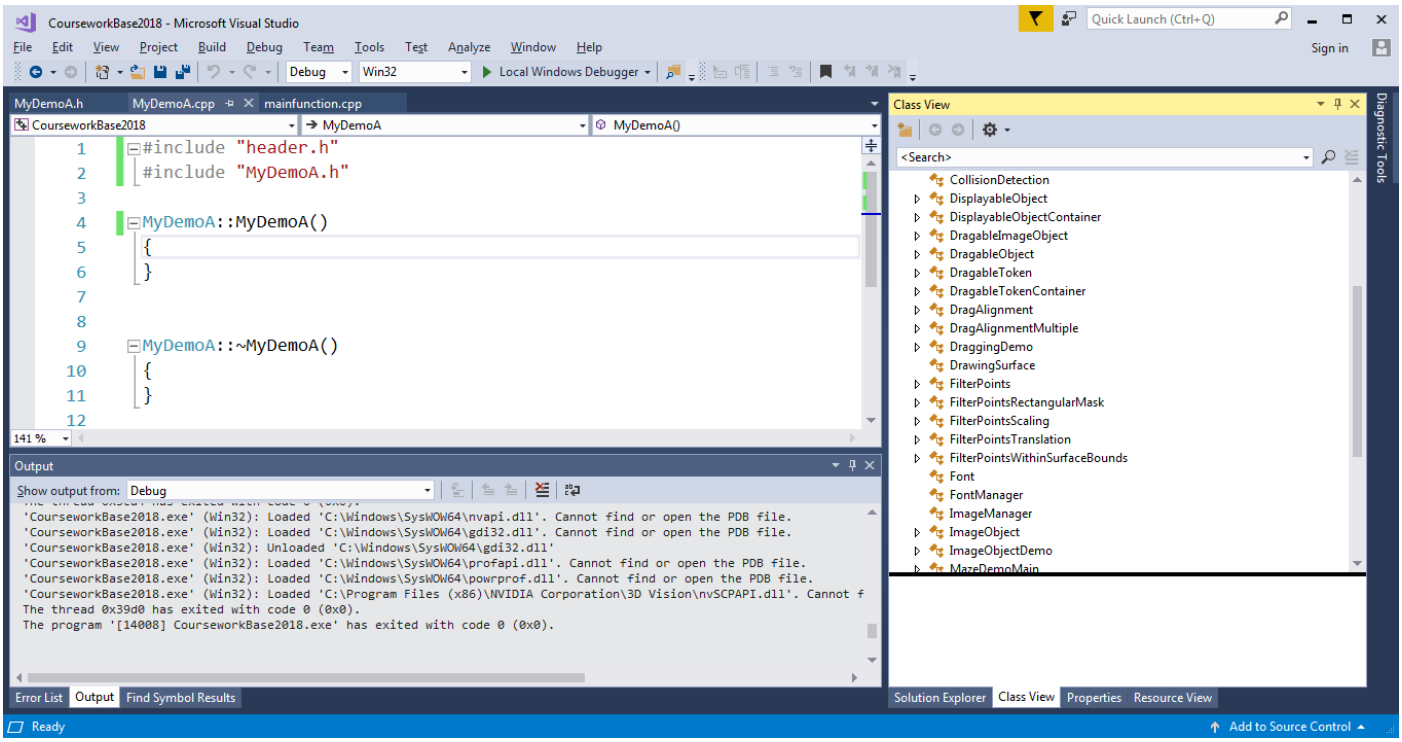


Drawing your own background

The framework keeps a background buffer for you. This is something that you can draw to and then copy onto the screen to overwrite anything drawn there already.

Switch to the class view - it's an option at the bottom of the solution explorer. If it is not there then choose 'Class View' from the View menu. Please note the tabs along the bottom of the pane, which let you choose which view to show. You can drag tabs around to reorder them. You can also drag panes. E.g. here I dragged the properties ane on top of the solution explorer so it just becomes another tab.

Class View will show classes instead of files. Expand the coursework base project and you will see something like this:



Right click on MyDemoA and choose "Add" > "Add Function…". A dialog box will be displayed asking what function you wish to add (see the next page). It will then the code to both the .h and .cpp file. This will add the function to the .h and .cpp files.

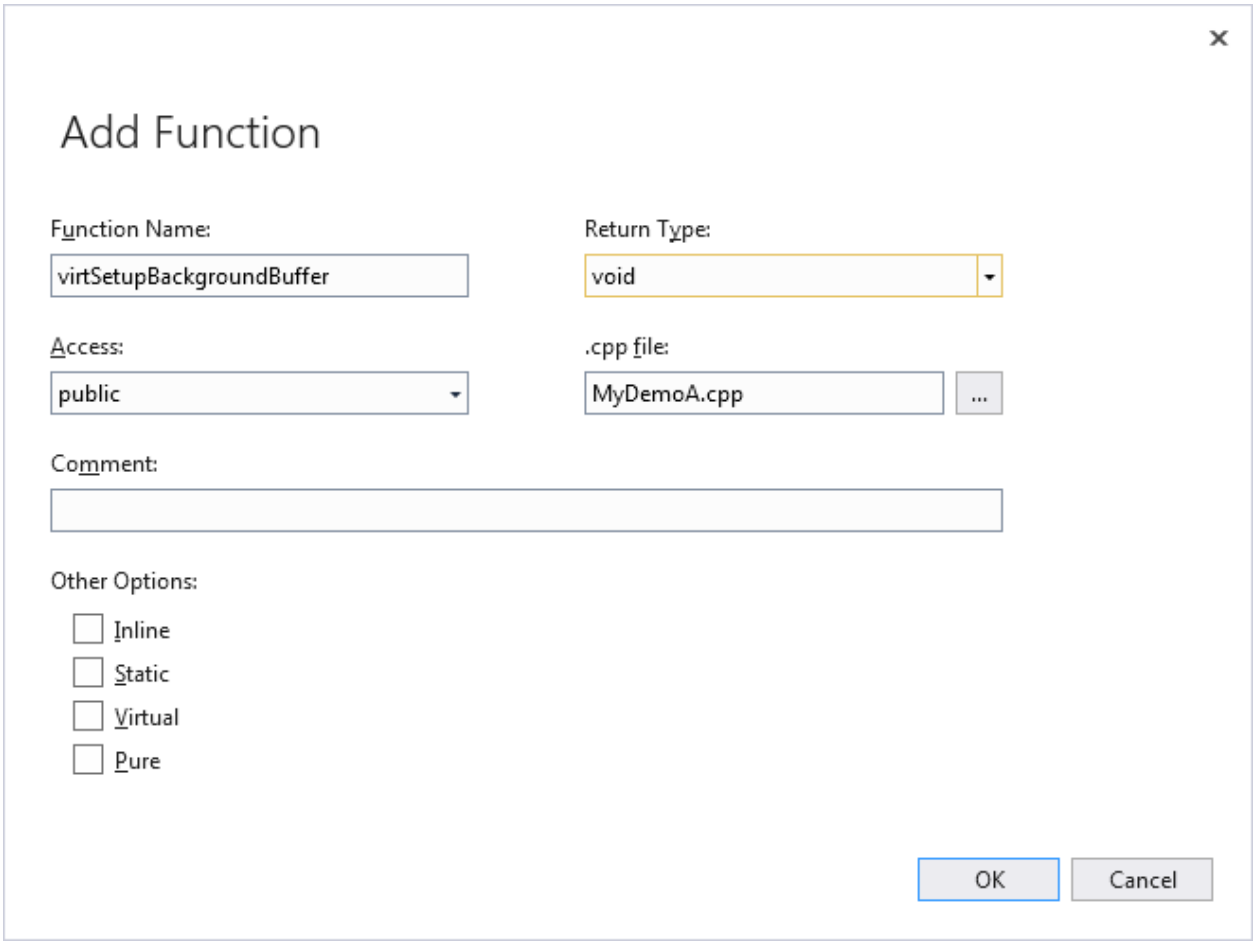
Choose the function name of "virtSetupBackgroundBuffer" and a return type of void.

Overriding Virtual Functions

This method of adding functions and seeing features appear works because the functions such as virtSetupBackgroundBuffer(), virtMouseDown(), virtKeyDown() etc are all implemented in the base class called BaseEngine, of which your class is a subclass.

These functions are all virtual functions and by implementing a version in your class, your version will be called instead of the base class version. You can look at BaseEngine.h to see that the framework provides many different functions, including a number of different drawing functions.

Most of the virtual functions in BaseEngine that you will need to override have names starting 'virt', to help you to find them. You must get the parameter types, names and return type exactly the same as in the base class when you override them.



This function is called whenever the program needs to draw the background. By changing its implementation, you make it use your version rather than the original (which just drew a yellow background).

Implement the function to fill the background with a solid colour, using the fillBackground() method:

fillBackground(0xff0000);

Build and run the program and it will have a red background. The 0xff0000 is a hexadecimal colour code. The first two digits are the amount of red (00 to ff), the next two the amount of green and the last two the amount of blue. The colours are expressed in the form 0xrrggbb where rr is a 2 digit hexadecimal amount of red, gg the amount of green and bb the amount of blue. E.g. 0x0000ff is blue (no red, no green, full blue, and 0x000080 is a darker blue).

Try some other colours before you continue.

Now type in the following code instead:

```
void MyDemoA::virtSetupBackgroundBuffer()
{
	fillBackground(0x000000);

	for (int iX = 0; iX < getWindowWidth(); iX++)
		for (int iY = 0; iY < this->getWindowHeight(); iY++)
			switch (rand() % 100)
			{
			case 0: setBackgroundPixel(iX, iY, 0xFF0000); break;
			case 1: setBackgroundPixel(iX, iY, 0x00FF00); break;
			case 2: setBackgroundPixel(iX, iY, 0x0000FF); break;
			case 3: setBackgroundPixel(iX, iY, 0xFFFF00); break;
```

```
                    case 4: setBackgroundPixel(iX, iY, 0x00FFFF); break;
                    case 5: setBackgroundPixel(iX, iY, 0xFF00FF); break;
                    }
}
```

Try to work out what this does before running it or continuing. Then run it to see what happens.

Make sure that you try to understand what it is doing before reading the explanation below.

Explanation of what the code does:

This code has two loops, forcing it to consider every pixel in the background in turn. i.e. it iterates every column of the screen, and for each column it iterates through each pixel in that column.

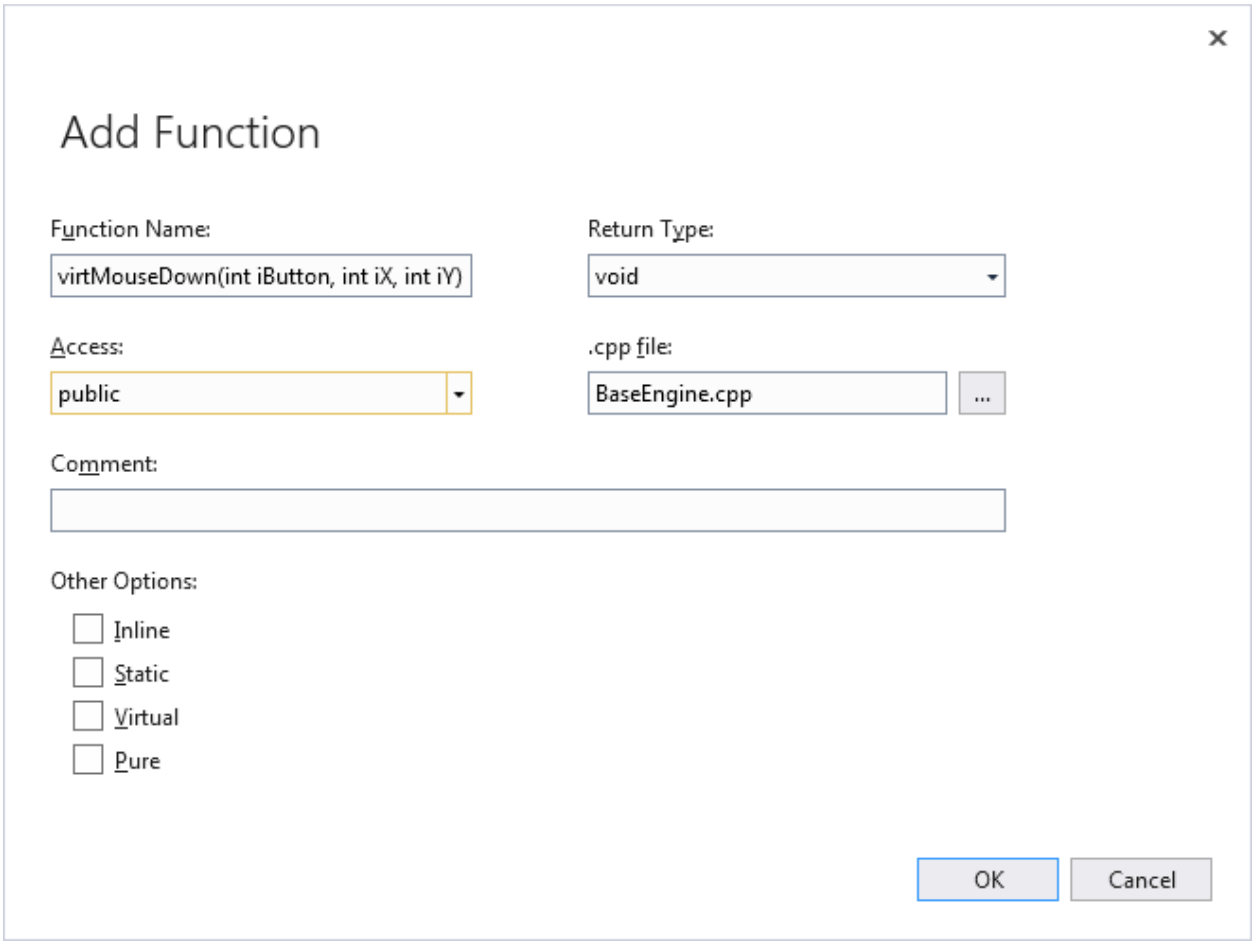
For each pixel it chooses a random number. It then takes the remainder when that number is divided by 100, giving it a random number from 0 to 99, with each value equally likely.

If the number is a 0 it will set the pixel to Red (0xff0000). If it is 1 then it will be green. If 2 then it is blue. If 3 then it is yellow (red+green). If 4 then it is cyan (green+blue). If 5 then it is magenta (red+blue). Any other value will leave the pixel unchanged. So the majority will be unchanged, but some will be coloured.

Handling mouse input

Now we will add some interaction:

Add a new function called virtMouseDown as follows, this will handle mouse presses. Note: Return type void, name "virtMouseDown" with three integer parameters. For the name of the function, you will need to specify both the name and the parameters.



When you choose Finish the following function will be created:

```
void MyDemoA::mouseDown(int iButton, int iX, int iY)
{
	// TODO: Add your implementation code here.
}
```

Modify the function by typing the following code to draw a rectangle on the BACKGROUND when the left button is pressed:

```
void MyDemoA::virtMouseDown(int iButton, int iX, int iY)
{
	printf("Mouse clicked at %d %d\n", iX, iY);

	if (iButton == SDL_BUTTON_LEFT)
	{
		drawBackgroundRectangle(iX - 10, iY - 10, iX + 10, iY + 10, 0xff0000);
		redrawDisplay(); // Force background to be redrawn to foreground
	}
}
```

The first line will cause some debug info to go to the output, so you can see what the x and y coordinates were for the click.

Then the if statement verifies that it was the left mouse button that was clicked.

The `drawBackgroundRectangle` function will draw a rectangle on the background, from X-10 to X+10 and Y-10 to Y+10 (i.e. height and width of 21).

The Redraw function is important because it is the way that you tell SDL that something changed. There is a discussion of the meaning of the parameter later in this document. The important thing at the moment is to know that you need to call `redrawDisplay` () to tell SDL to redraw your changes: If you draw to the background it will not appear if you do not call `redrawDisplay` () from somewhere. Calling `redrawDisplay` () multiple times is no problem, but not calling it at all will stop your change appearing.

> **Behind the scenes:**
>
> Functions like `drawBackgroundRectangle()`, `drawBackgroundOval()` and `redrawDisplay` () are implemented in the base class. Your class inherits these functions so you can call them to 'draw a rectangle' or the tell SDL that the screen has changed and it needs to be redrawn.
>
> The virt`MouseDown()` function is another of these virtual functions in the base class. You can look for it in BaseEngine.cpp if you wish. You will see that the base class version does nothing.
>
> The `BaseEngine` code will check for events (like a mouse click) constantly, and every time it gets a mouse down event it will call thevirt `MouseDown()` function. In the base class this will do nothing, but as soon as you add your own implementation, it will start to do something.

You can now add some code for the right mouse button as well:

```
void MyDemoA::virtMouseDown(int iButton, int iX, int iY)
{
	printf("Mouse clicked at %d %d\n", iX, iY);

	if (iButton == SDL_BUTTON_LEFT)
	{
		drawBackgroundRectangle(iX - 10, iY - 10, iX + 10, iY + 10, 0xff0000);
		redrawDisplay(); // Force background to be redrawn to foreground
	}
	else if (iButton == SDL_BUTTON_RIGHT)
	{
		drawBackgroundOval(iX - 10, iY - 10, iX + 10, iY + 10, 0x0000ff);
		redrawDisplay(); // Force background to be redrawn to foreground
	}
}
```

Compile and execute this to try it.

Behind the scenes:

The framework will sit in a loop, constantly repeating the following steps: (look for `MainLoop` in BaseEngine.cpp to see the code)

1) Handle any events, e.g. key presses, mouse presses etc, and call the relevant functions.
2) Call `updateAllObjects()`. This will eventually call the `virtDoUpdate()` functions on the moving objects which you will learn about in the next lab. This function is responsible for changing anything, e.g. moving any objects around or handling some user input.
3) Call `virtRenderScreen()`. This function is responsible for re-drawing the screen if necessary.

Foreground and Background

It is important to understand the difference between the foreground and the background.

The background is usually unchanging, and objects will move across in front of the background.

Think of there being two different copies of the screen – a background and a foreground. When you draw moving objects (see next lab) you will draw to the foreground. This means that an object can be moved by redrawing the background over the old position of the object (removing the object from its old position) then drawing it again in the new position. You can only do this because you have a copy of the background as well, which will be used to draw over the old positions of the objects to remove them.

Usually, the entire background is drawn every frame: the background will be copied onto the foreground, then other functions will be called to draw text and moving objects on top.

If anything changes (e.g. an object moves), and the screen needs to be redrawn then you need to tell the program to redraw the screen by calling the function `redrawDisplay();`

If you redraw anything to the background then you will need to redraw the screen to make it appear. If you do not then it will not appear on the foreground and not be shown.

virtSetupBackgroundBuffer() is called when the program starts up, so any changes you make will appear at the start of the program. You can call it again later manually, if you need to.

The background and foreground

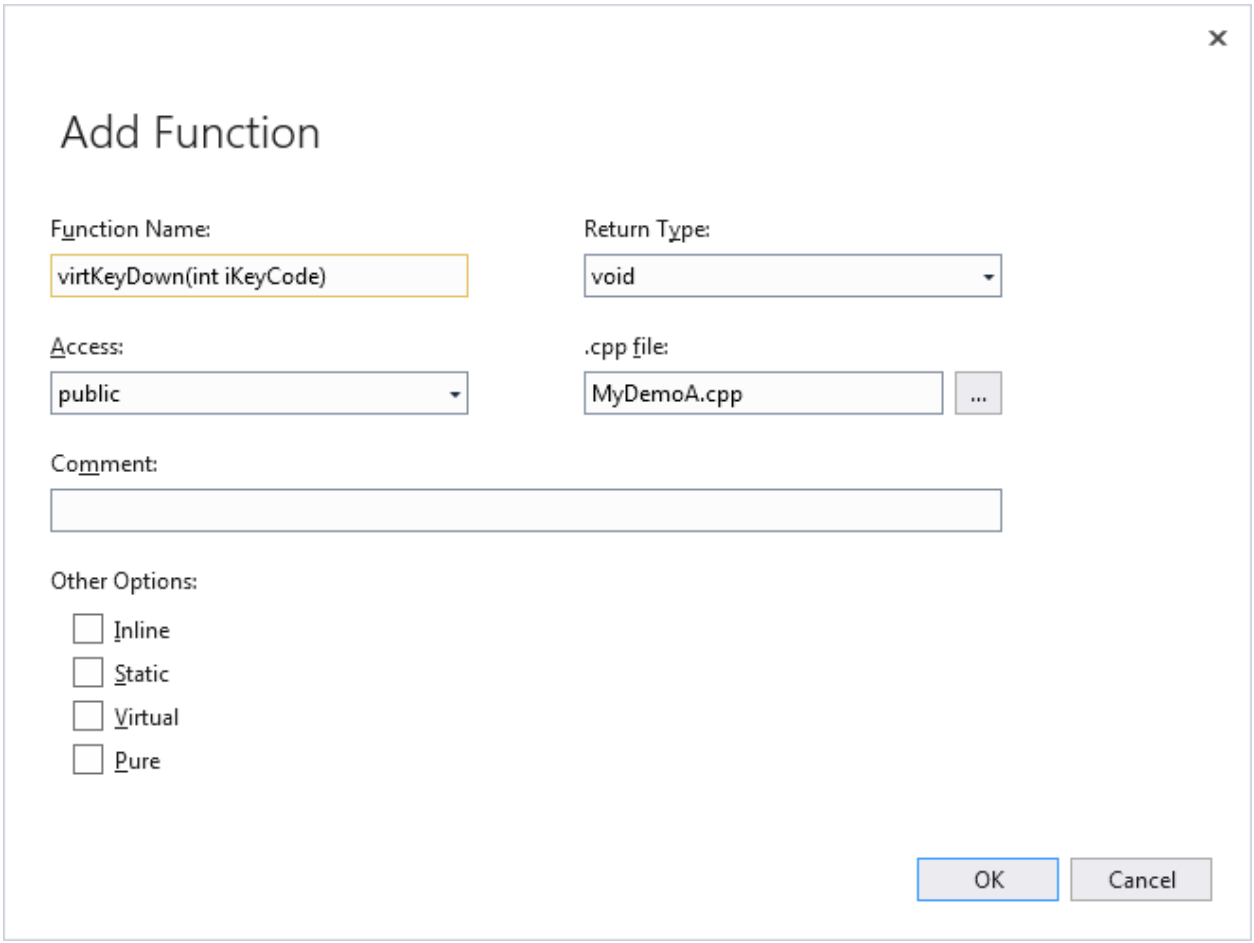
Rules:

1) If you are drawing a stationary/unchanging item them draw it to the background, otherwise it will be eliminated when you redraw the screen.
2) If you are drawing a moving object then draw it to the foreground – you WANT it to be removed when you redraw the screen, to remove it from the old position on the screen. You need to draw it every frame – e.g. in the virtDoUpdate function of a displayable object (see demo B).

Handling key presses

You can handle key presses as the key is pressed down using the following method:.

First add the virtKeyDown function as specified below (note the single integer parameter which tells you which key was pressed):



This will create you the following function in your cpp file (and a declaration in the .h file):

```
void MyDemoA::virtKeyDown(int iKeyCode)
{
	// TODO: Add your implementation code here.
}
```

Then type in the following implementation:

```
void MyDemoA::virtKeyDown(int iKeyCode)
{
	switch (iKeyCode)
	{
	case ' ':
		virtSetupBackgroundBuffer();
		redrawDisplay();
		break;
	}
}
```

This will redraw the entire background again when you press SPACE, calling the function which we specified earlier, drawing over any existing images/shapes. You can handle other keys in a similar way.

If you need to handle other characters, you can use the SDL keycode, usually named SDLK_ then the name of the key. See http://wiki.libsdl.org/moin.cgi/SDLKeycodeLookup for a list, or look in SDL_keysym.h. e.g.:

```
void MyDemoA::virtKeyDown(int iKeyCode)
{
	switch (iKeyCode)
	{
	case SDLK_SPACE:
		virtSetupBackgroundBuffer();
		redrawDisplay();
		break;
	}
}
```

It just happens that most are set to the same as the printable ASCII character, so you can just use ‘a’ for example, rather than SDLK_a. (Well it’s actually deliberate, but still very useful.)

Displaying images from image files

You can also draw images as well as text to the background. First you need to load the image into memory, then you can draw the image. The ImageManager class has a lot of code in it to handle images without you having to do much of the work yourself.

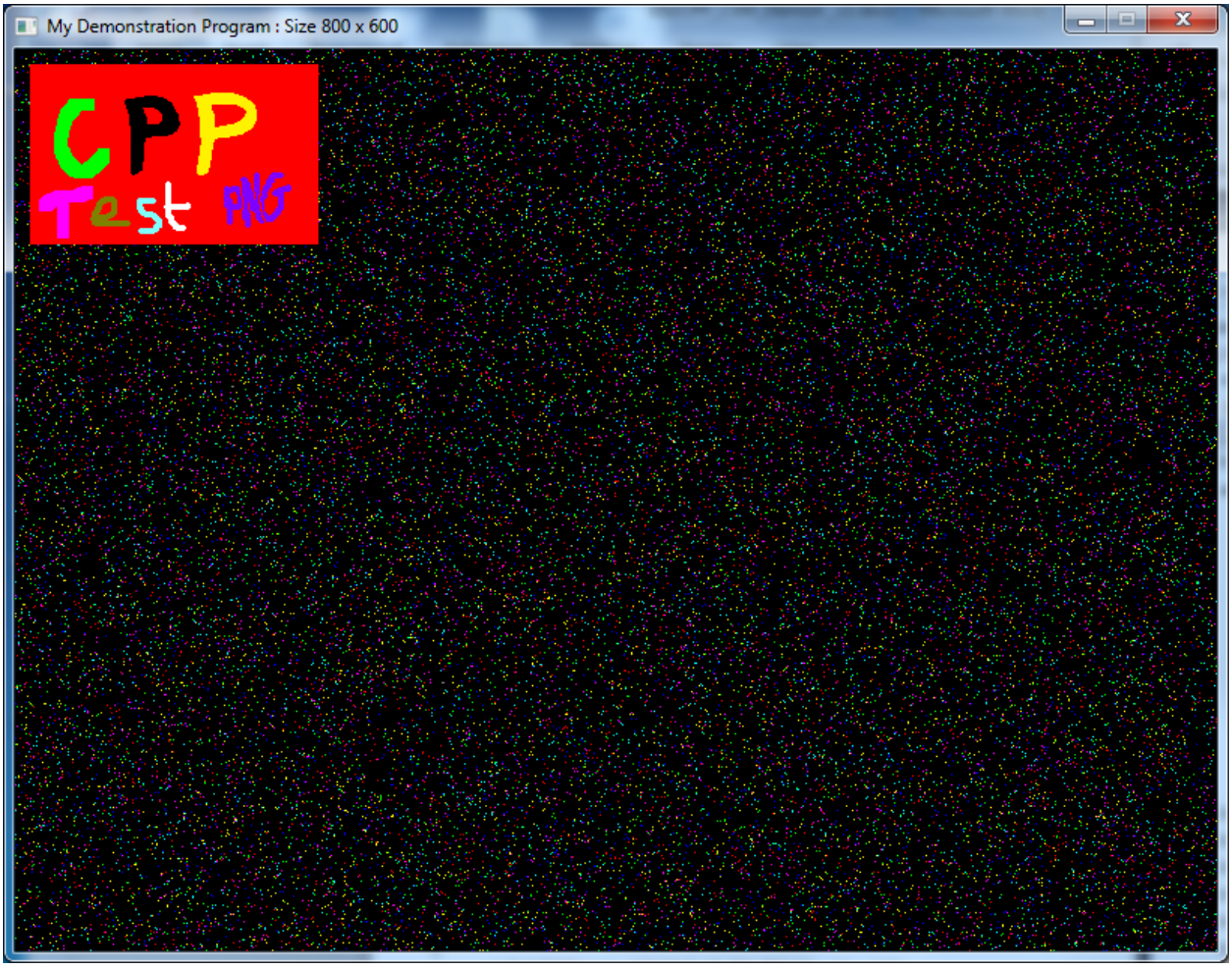
First load the image. A boolean parameter is supplied , which says whether to keep it permanently loaded. If you specify false for this then the image will only stay loaded while you have a pointer to it, then it will unload again. If you use true then the next time you call this method it will just give you an object referring to the previously loaded image, avoiding reloading. Add the following code to your virtSetupBackgroundBuffer() function:

```
SimpleImage image = ImageManager::loadImage("demo.png", true);
image.renderImage(getBackgroundSurface(), 0, 0, 10, 10,
            image.getWidth(), image.getHeight());
```

Note: the SimpleImage class wraps up a smart pointer, which is an object which acts like a pointer. You can use the SimpleImage object like a pointer to the image. Use the object itself, and pass the objects around into functions, etc, rather than passing pointers to the SimpleImage objects.

And don't forget to #include the header file for the image manager near the top of the file (AFTER header.h, not before it):

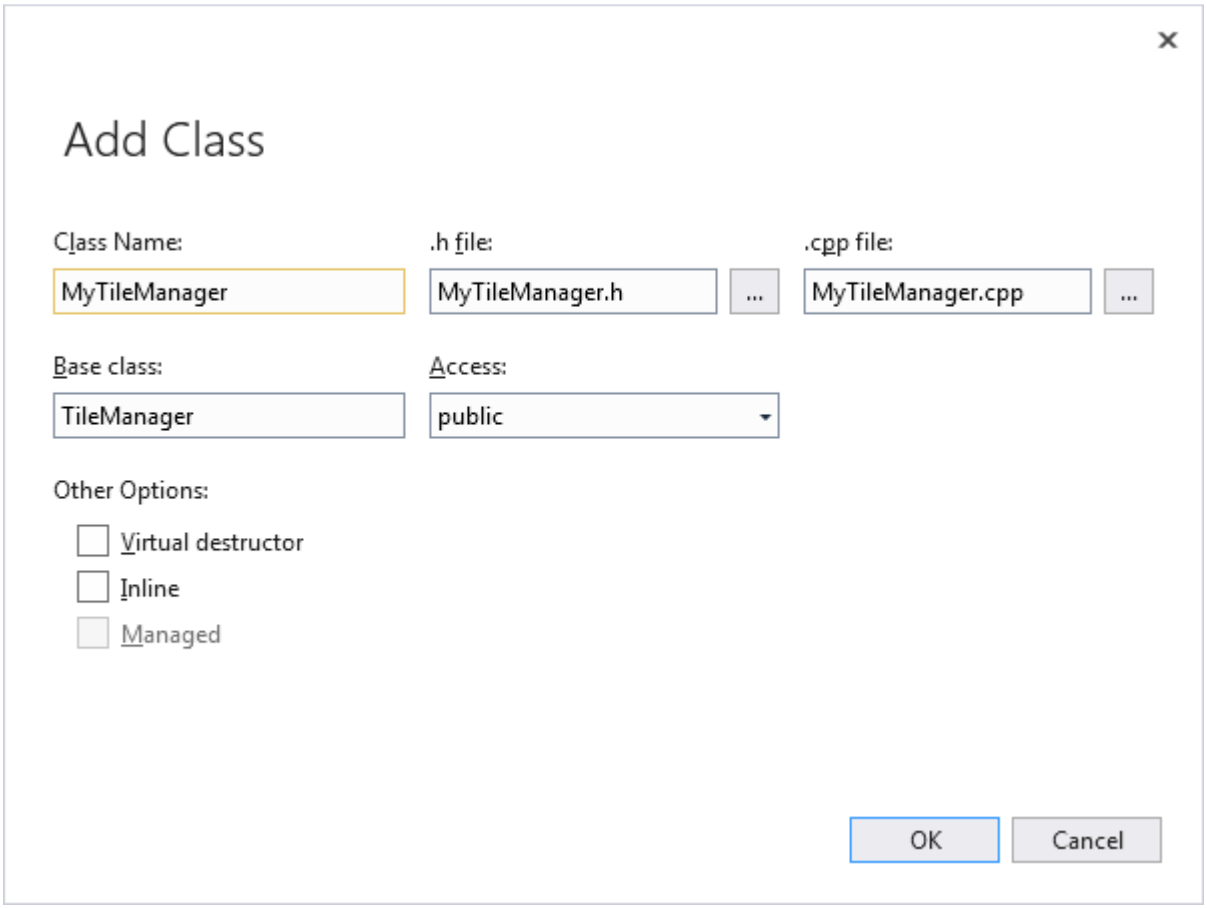
```
#include "ImageManager.h"
```



Creating a basic Tile Manager

Tile managers are useful for drawing a rectangular grid background (for example for maze games).

Create a new TileManager subclass:



Don't forget to add to the top of the .cpp file:

```
#include "header.h"
```

The key method for the class is the function:

```
void TileManager::virtDrawTileAt(
      BaseEngine* pEngine,
      DrawingSurface* pSurface,
      int iMapX, int iMapY,
      int iStartPositionScreenX, int iStartPositionScreenY) const
```

(See TileManager.cpp for a basic implementation.)

At this point, you could use the add function dialog again, but personally, when there are this many parameters, I would copy-paste the text from the .h and .cpp files in the subclass into the new class, then replace the text "TileManager" by the new class name (MyTileManager in this case).

E.g. for the .h file I get:

```
	virtual void virtDrawTileAt(
		BaseEngine* pEngine,
		DrawingSurface* pSurface,
		int iMapX, int iMapY,
		int iStartPositionScreenX, int iStartPositionScreenY ) const;
```

And for the .cpp file I get:

```
void TileManager::virtDrawTileAt(
	BaseEngine* pEngine, // We don't need this but maybe a student will so it is
here to use if needed
	DrawingSurface* pSurface,
	int iMapX, int iMapY,
	int iStartPositionScreenX, int iStartPositionScreenY) const
{
	int iMapValue = getMapValue(iMapX, iMapY);
	unsigned int iColour = 0x101010 * ((iMapX + iMapY + iMapValue ) % 16);
	pSurface->drawRectangle(
		iStartPositionScreenX,  // Left
		iStartPositionScreenY,  // Top
		iStartPositionScreenX + getTileWidth() - 1,           // Right
		iStartPositionScreenY + getTileHeight() - 1,    // Bottom
		iColour);   // Pixel colour
}
```

The tile manager maintains a two dimensional array of map tile values. You can set a value using int setMapValue(int iMapX, int iMapY, int iValue) and get a value using int getMapValue(int iMapX, int iMapY).

First create a tile manager. The easiest way is to create it as a member variable.

```
#include "BaseEngine.h"
#include "MyTilemanager.h"

class MyDemoA :
	public BaseEngine
{
public:
	MyDemoA();
	~MyDemoA();
	void virtSetupBackgroundBuffer();
	void virtMouseDown(int iButton, int iX, int iY);
	void virtKeyDown(int iKeyCode);

protected:
	MyTileManager tm;
};
```

If you do this then you probably also want to initialise the tile manager member. You can do this from the constructor for the subclass:

```
MyTileManager::MyTileManager()
```

```
      : TileManager( 20, 20, 15, 15 )
{
}
```

This tells the tile manager to initialise for a 20x20 pixel tile size and a 15 tile by 15 tile map size. You can set this to whatever you wish. Ensure that you choose something different when implementing your coursework.

Now go to your virtSetupBackgroundBuffer() function in your MyDemoA class. If we want the tiles to display then we need to actually draw them to the screen. We will do this at the end of this function by adding the following lines:

```
      for (int i = 0; i < 15; i++)
            for (int j = 0; j < 15; j++)
                  tm.setMapValue(i, j, rand());
      tm.setTopLeftPositionOnScreen(50, 50);
      tm.drawAllTiles( this, getBackgroundSurface() );
```

This does three things – first it sets the tiles to random values (you probably want to do something better than this) then it tells the tile manager where the top left corner of the tiles is (so it knows where to draw them). i.e. you can provide a border before getting to the tiles.

Next you just tell it to draw all of the tiles. The tile manager will draw all of the tiles, using the current map values, and you just need to tell it the BaseEngine class to use (i.e. your subclass object, this) and the surface to draw to (the background in this case).

Run your code at this point and test it.

Now the only thing you need to change is the function that actually draws the tiles.

Try changing the code to the following:

```
      int iMapValue = getMapValue(iMapX, iMapY);
      unsigned int iColour = (0x100000 * (iMapX & 1)
            + 0x001000 * (iMapX & 2)
            + 0x000010 * (iMapX & 4)
            + 0x010000 * (iMapY & 1)
            + 0x000100 * (iMapY & 2)
            + 0x000001 * (iMapX & 4)) * (iMapValue % 16);
      pSurface->drawOval (
            iStartPositionScreenX,  // Left
            iStartPositionScreenY,  // Top
            iStartPositionScreenX + getTileWidth() - 1,           // Right
            iStartPositionScreenY + getTileHeight() - 1,    // Bottom
            iColour);
```

This changes the code to both draw ovals and to change the colours depending upon the map value of the tile.

Run your code at this point and test it.

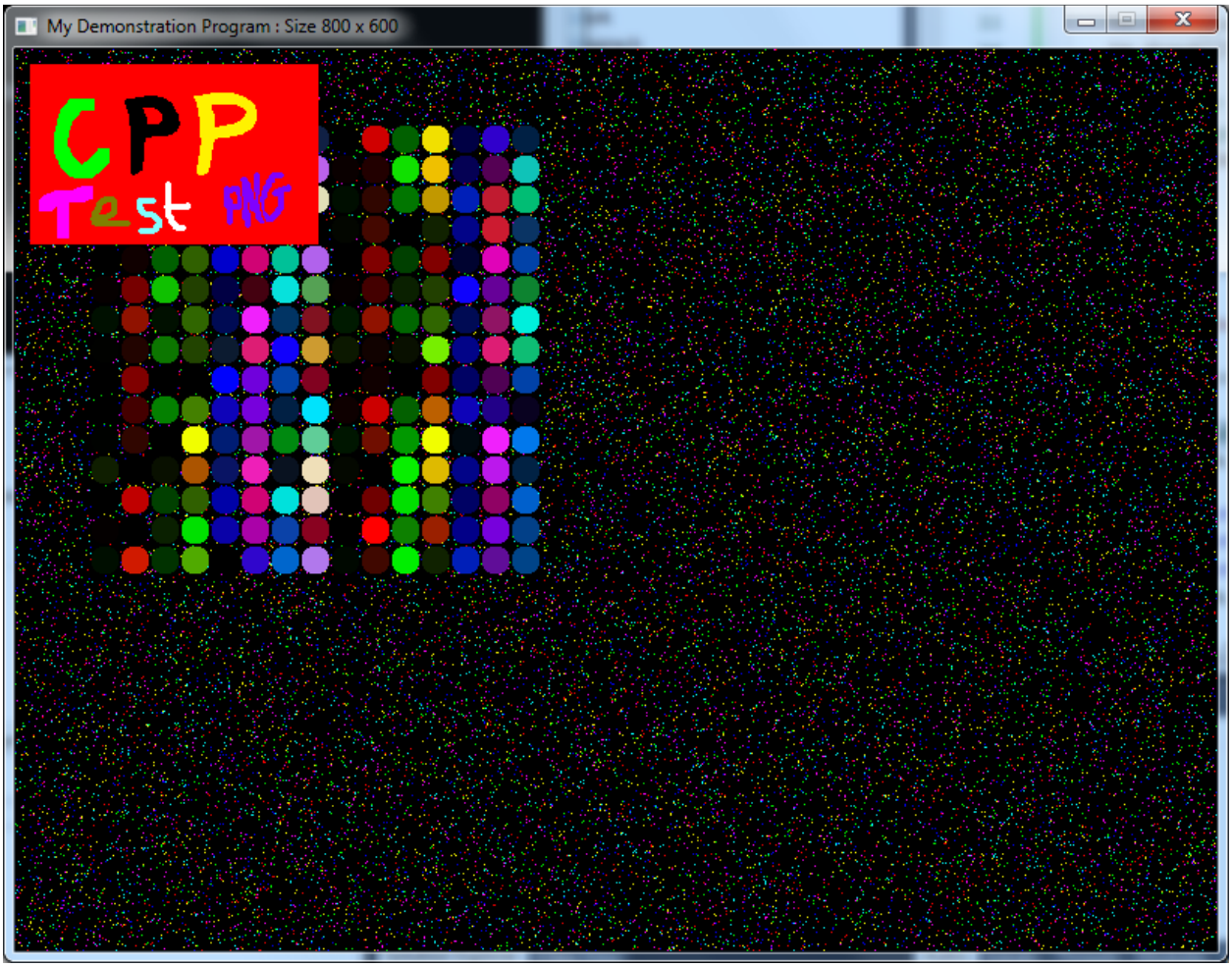
Ensure that you understand what how the tile manager values work.

Finally, you can also implement a facility to change tile values when you click on this. Go to your code for the click handling and change it to work out which tile is clicked on, and change the value. The important code is this:

```
if (tm.isValidTilePosition(iX, iY)) // Clicked within tiles?
{
```

```
        int mapX = tm.getMapXForScreenX(iX); // Which column?
        int mapY = tm.getMapYForScreenY(iY); // Which row?
        int value = tm.getMapValue(mapX, mapY); // Current value?
        tm.setAndRedrawMapValueAt(mapX, mapY, value + 1, this, getBackgroundSurface() );
}
```

Try this code yourself and try clicking on a tile in the display.



The full method is included below in case you need it for reference:

```
void MyDemoA::virtMouseDown(int iButton, int iX, int iY)
{
        printf("Mouse clicked at %d %d\n", iX, iY);

        if (iButton == SDL_BUTTON_LEFT)
        {
                //drawBackgroundRectangle(iX - 10, iY - 10, iX + 10, iY + 10, 0xff0000);
                if (tm.isValidTilePosition(iX, iY)) // Clicked within tiles?
                {
                        int mapX = tm.getMapXForScreenX(iX); // Which column?
                        int mapY = tm.getMapYForScreenY(iY); // Which row?
                        int value = tm.getMapValue(mapX, mapY); // Current value?
                        tm.setAndRedrawMapValueAt(mapX, mapY, value + 1, this, getBackgroundSurface()
);
                }
                redrawDisplay(); // Force background to be redrawn to foreground
        }
        else if (iButton == SDL_BUTTON_RIGHT)
        {
                drawBackgroundOval(iX - 10, iY - 10, iX + 10, iY + 10, 0x0000ff);
                redrawDisplay(); // Force background to be redrawn to foreground
        }
}
```

What now?

Consider what you have done so far. Your program already has quite a lot of functionality, and you don't even have any moving objects yet. We will look at adding these in demo B.

Next time we will consider how to handle moving items, but this should be enough to get you started with knowing how to draw a background and how to handle both mouse presses and key presses.

Please take some time to experiment, and to look at what functions are available in the BaseEngine.h file. You will find that there are quite a few useful drawing functions.

Please also look at the various demos which you have been given. The bouncing ball in particular draws quite a lot of things to the background, and may give you some ideas, although some bits will be difficult to understand until you have done lab B.