

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

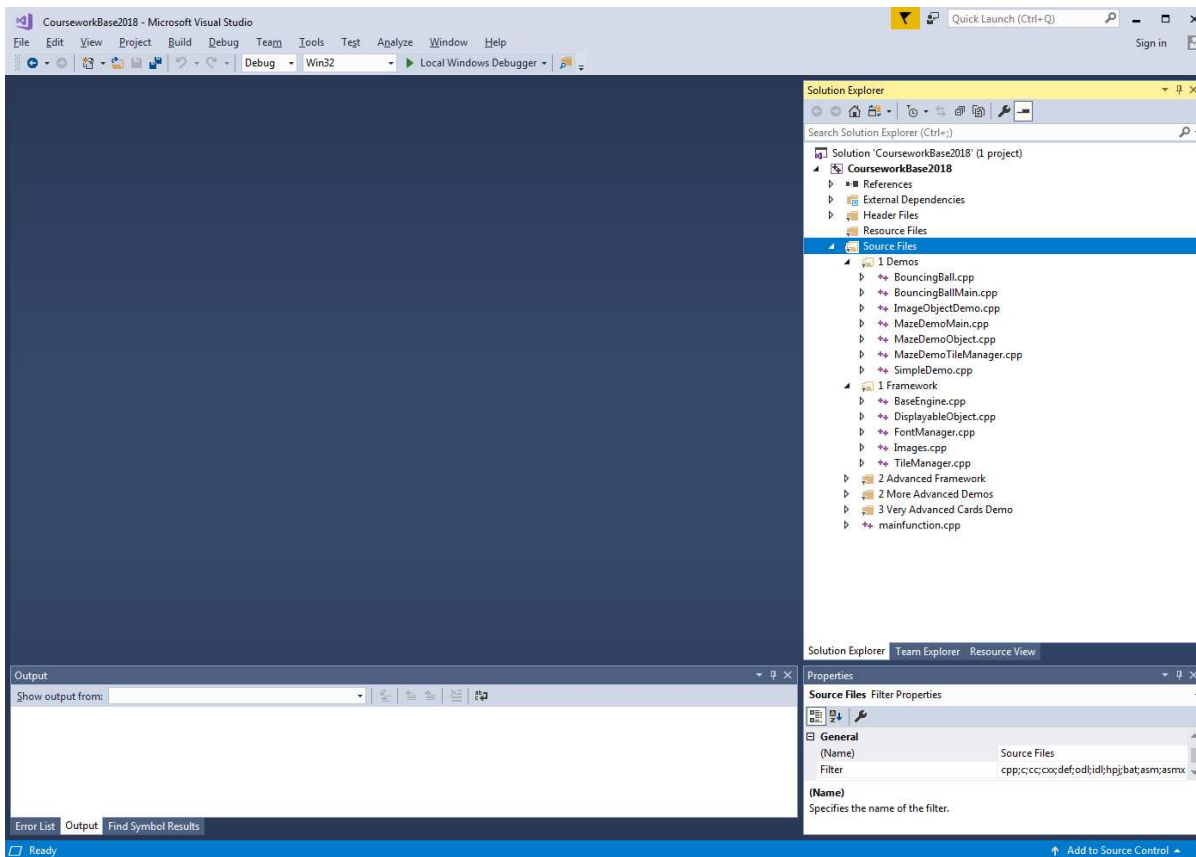
Unzip the files to a local directory. Double click on the .sln file in the root of the directory structure.

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! (Apparently Visual Studio does not like having its files on a non-Microsoft filesystem and may give you issues. If we can find a fix for this I will let you know – people are currently investigating.)

At the moment you can use Z: drive, but there is a plan to make Z: drive read-only and phase it out, so you may not be able to use this long-term. Possibly the best longer term plan is to run it from the local machine and remember to copy it back to somewhere safe each time before you log off.

Files on your local computer in the labs are NOT persisted between logins – so don't forget to copy them to somewhere safe.

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.



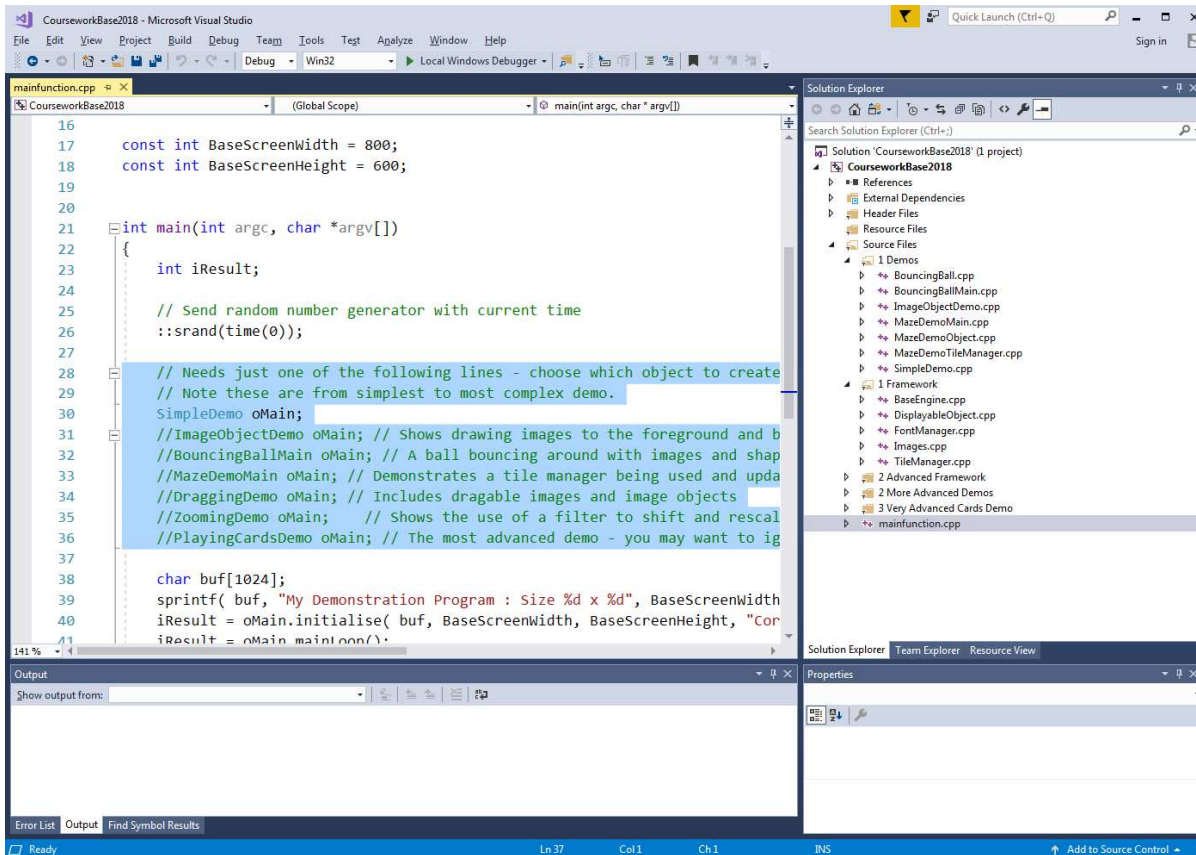
Open up Source Files and you will see some sub-folders. Go through these in numerical order – most are demos and they get more complex later on. Some people may want to just stick to the simpler ones, whereas some may want to investigate more complex versions.

For the moment, consider only the folders “1 Demos” and “1 Framework”, which are the simplest files to understand. The others are really just for more advanced features and demos.

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:

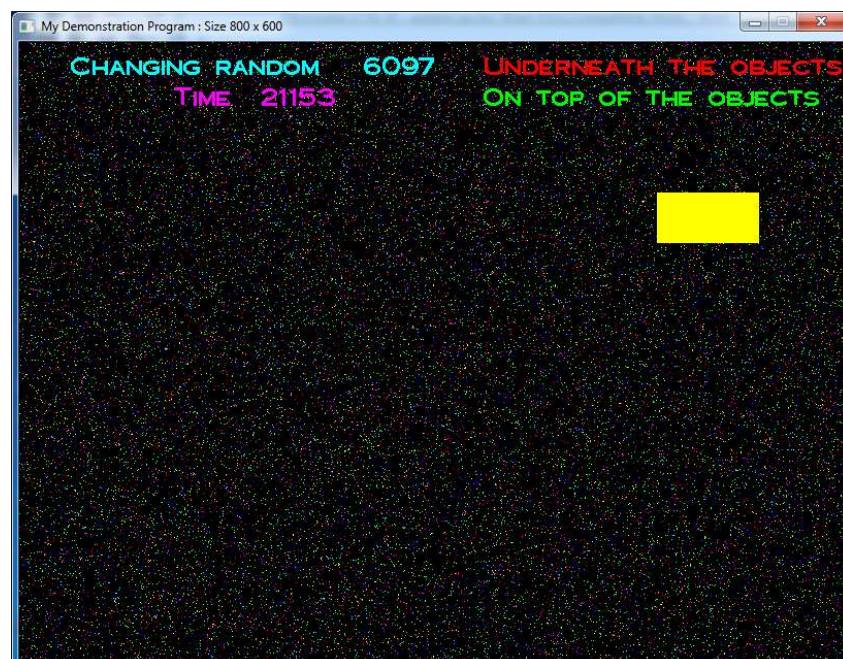


To change demo that is running just change which object is created.

(I will choose SimpleDemo for the moment.)

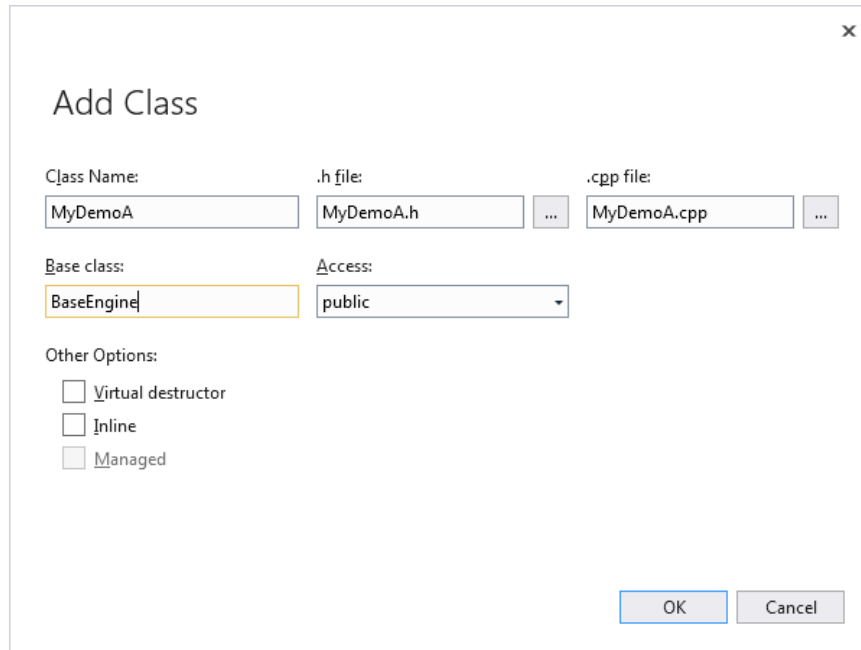
Compile the program (choose 'Rebuild Solution') from the Build menu.

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

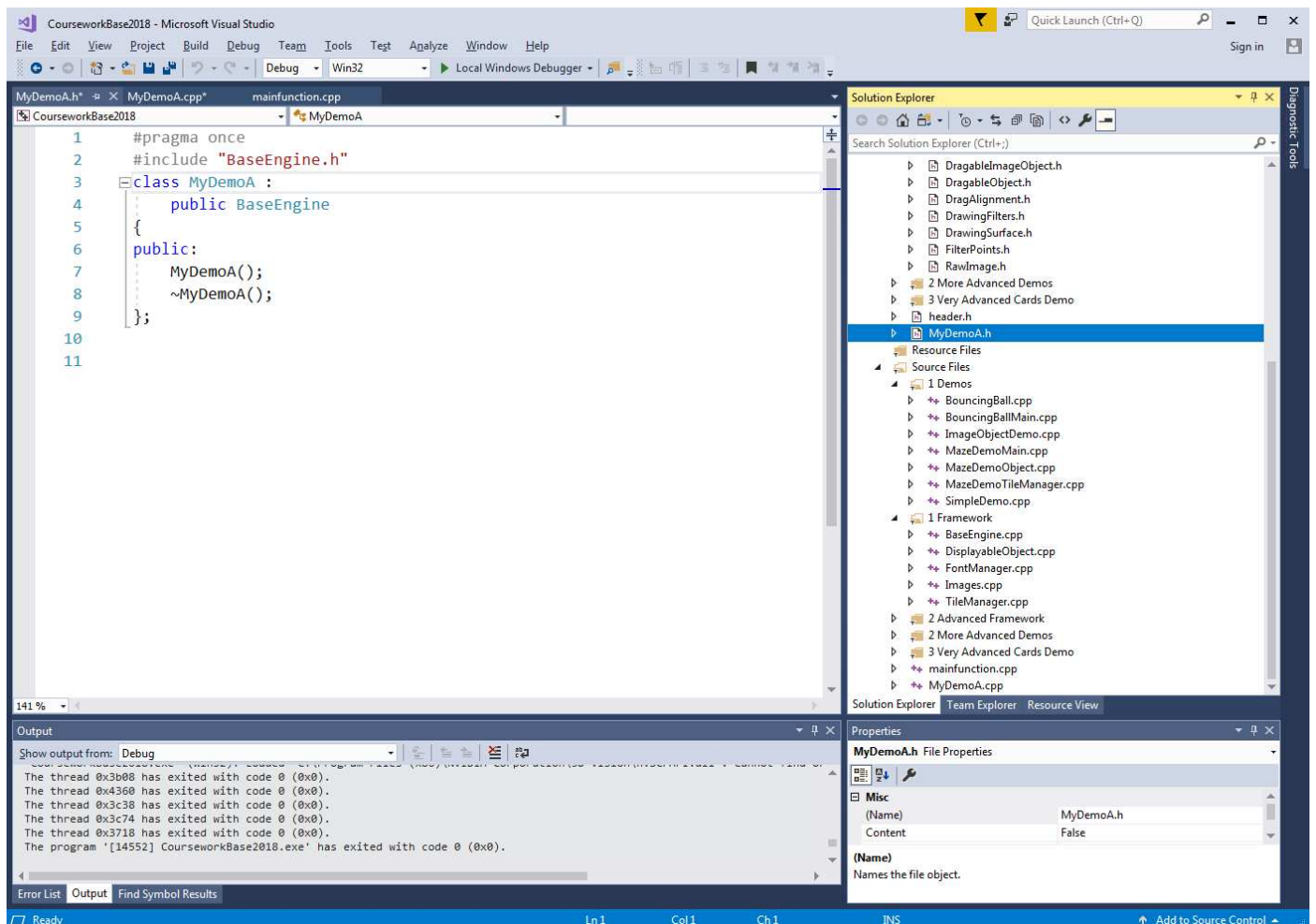
Right click on Source Files, and choose Add Class. Choose a class name of MyDemoA (you can call it what you wish but the following instructions assume this name) and a base class of BaseEngine (this has to be BaseEngine because you are subclassing an existing class).



The 'Add Class' dialog box in Visual Studio. It has a title bar with a close button. The main area contains the following fields and options:

- Class Name:** A text box containing 'MyDemoA'.
- .h file:** A text box containing 'MyDemoA.h'.
- .cpp file:** A text box containing 'MyDemoA.cpp'.
- Base class:** A dropdown menu with 'BaseEngine' selected.
- Access:** A dropdown menu with 'public' selected.
- Other Options:** Three checkboxes: 'Virtual destructor' (unchecked), 'Inline' (unchecked), and 'Managed' (unchecked).
- Buttons:** 'OK' and 'Cancel' buttons at the bottom right.

This creates a MyDemoA.h and a MyDemoA.cpp file and puts them in the header files and source files folders 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 different 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 after you finish these tasks to investigate these files.

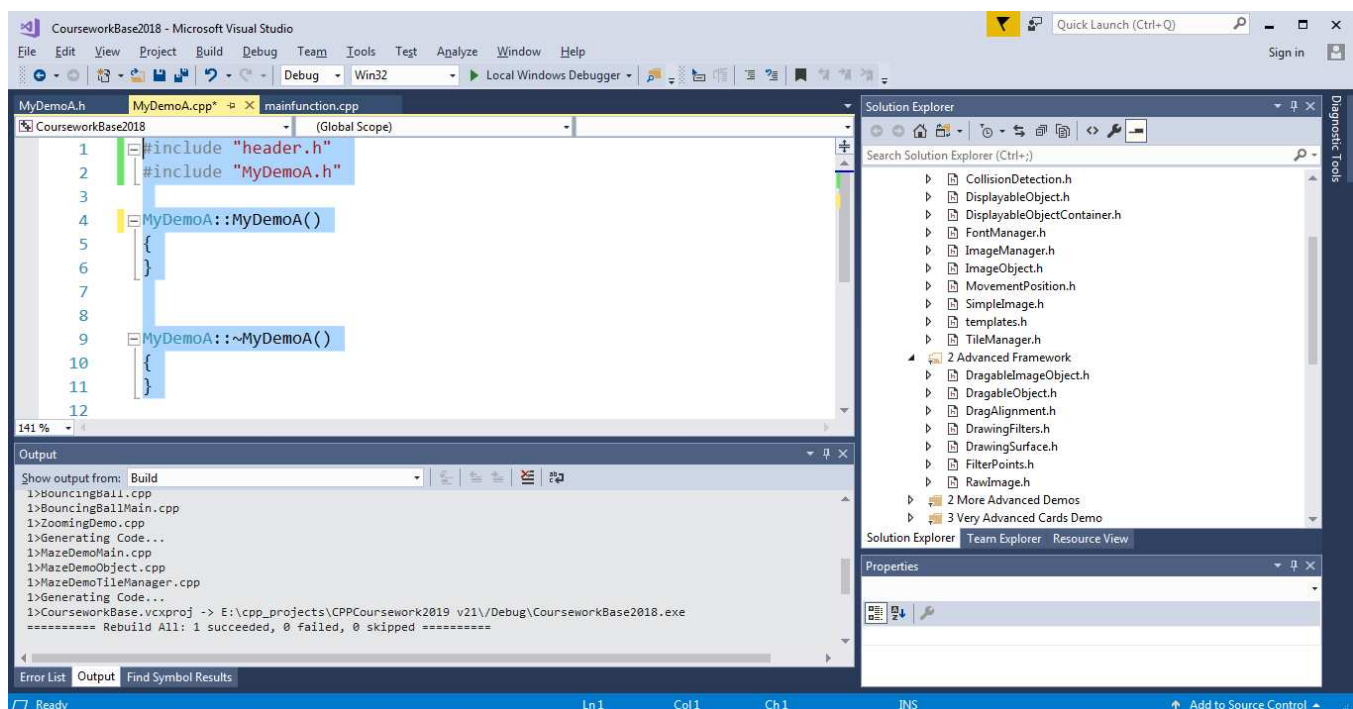
When you use the wizard to create a new class, it will usually put the .h file under Header Files and the .cpp file under Source Files (which is probably where you want them). Where the files are categorised is irrelevant: these ‘folders’ have no effect upon the program, they just help you to organise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

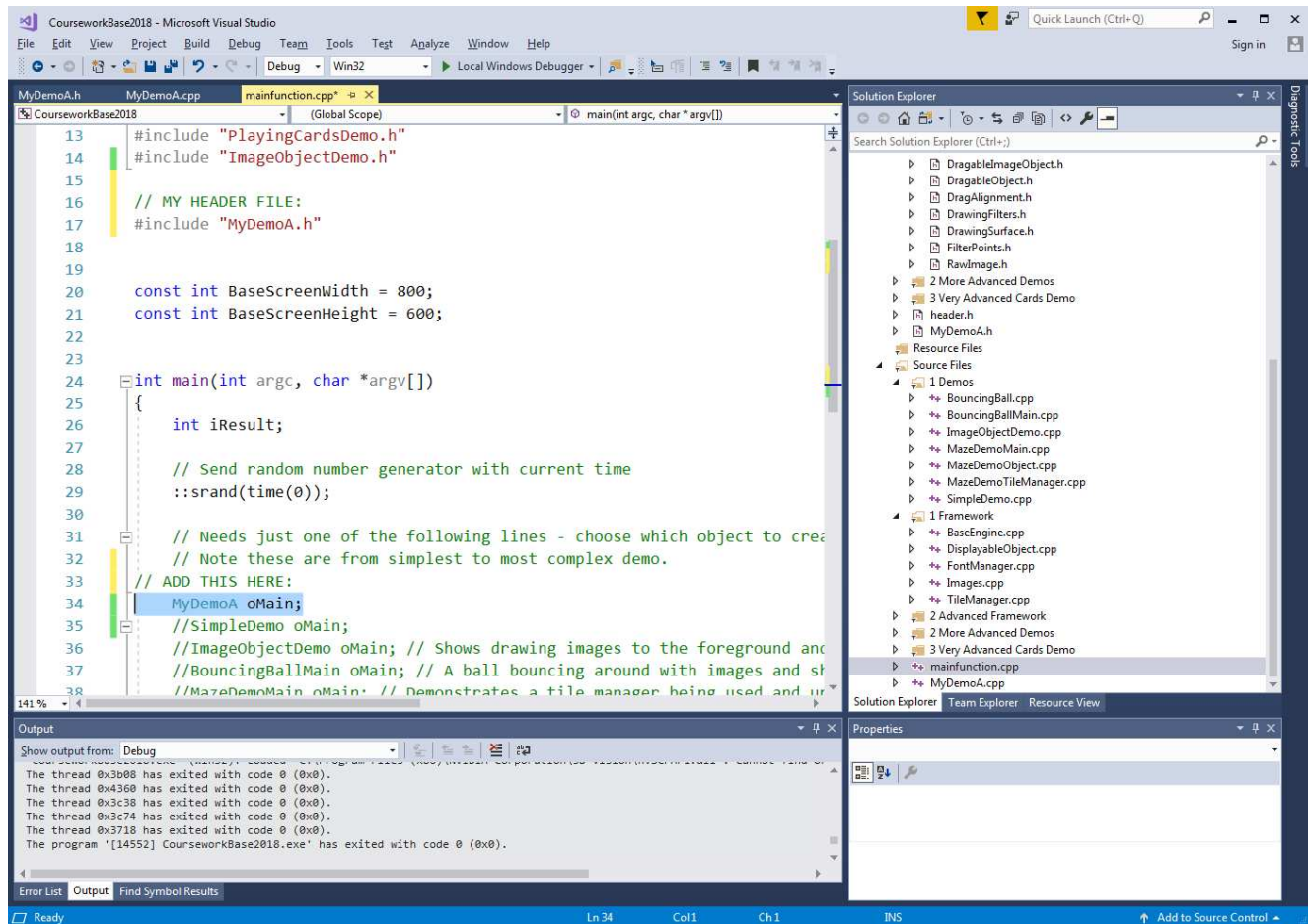
SDL.h adds various #defines to the code to change the names of various functions so that SDL, the graphics library we will use, will work. These need to apply to everything so you need to add this to the very top of every .cpp file you create. If you forget one, you will get a mix of functions with the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



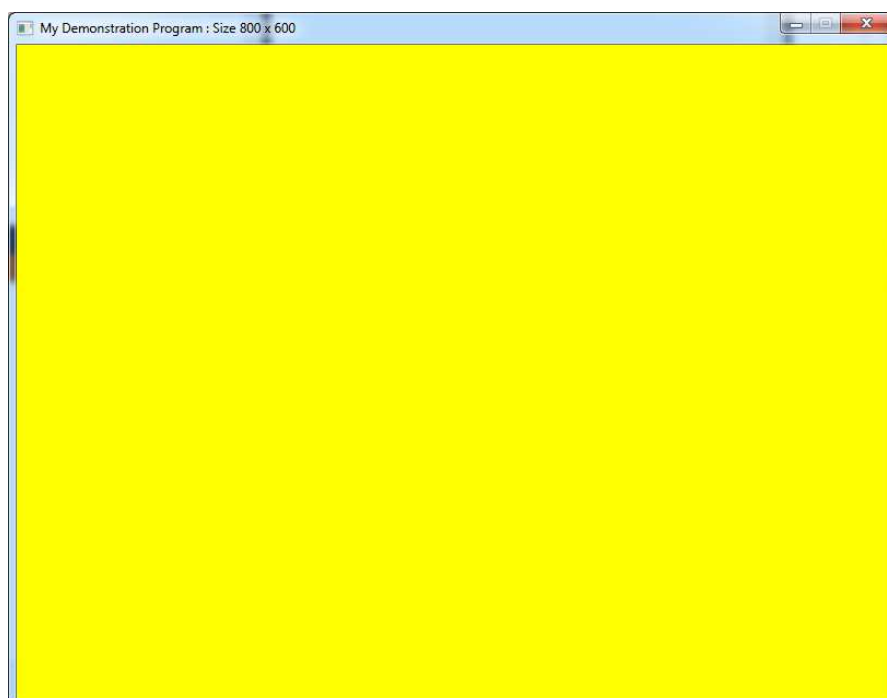
Save everything (shift-ctrl-S is a shortcut for this).

Open the mainfunction.cpp file and:

- #include the MyDemoA.h file at the top (i.e the file you just created).
- 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:

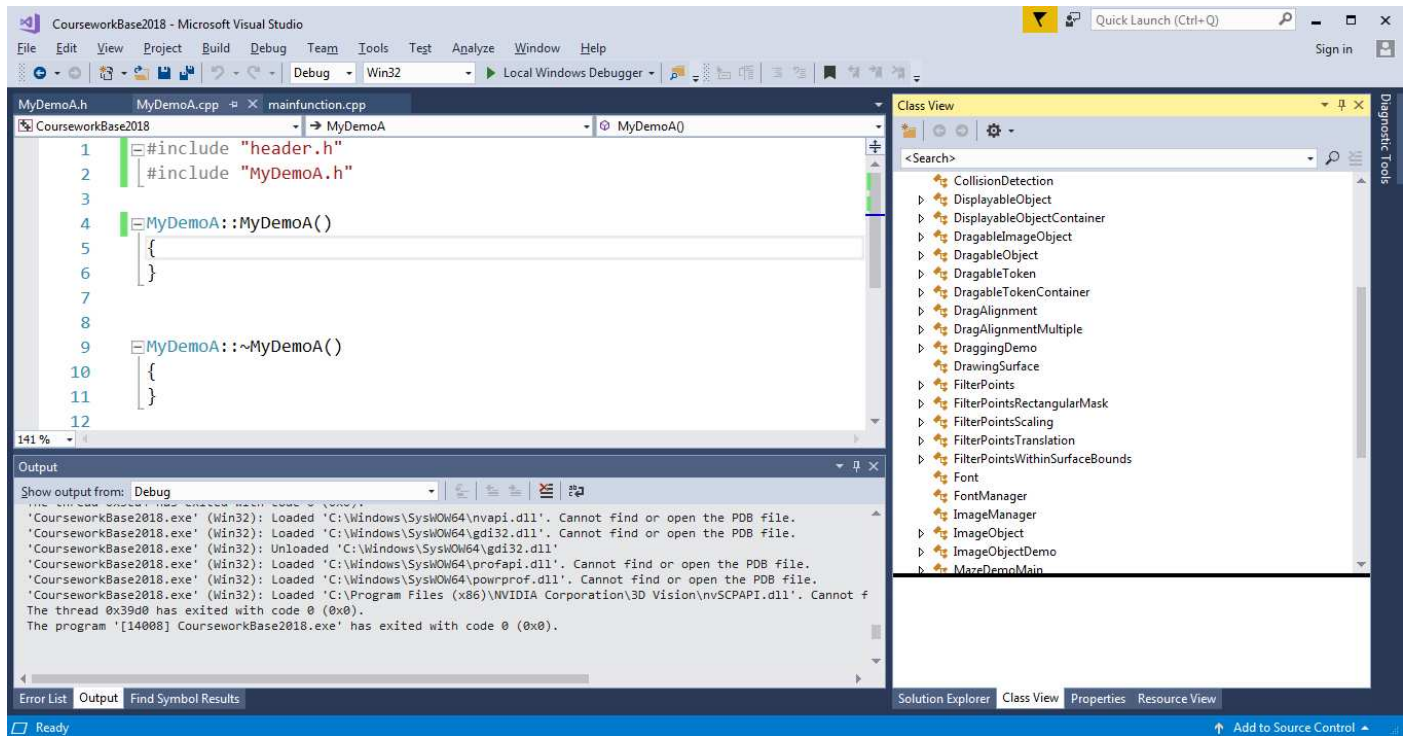


Drawing your own background

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 pane 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.

The dialog box on the next page will be shown.

Choose the function name of "virtSetupBackgroundBuffer" and a return type of void (see next page).

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);
```

Add Function

Function Name:

Return Type:

Access:

.cpp file:
...

Comment:

Other Options:

☐ Inline

☐ Static

☐ Virtual

☐ Pure

OK

Cancel

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).

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.

The virtual functions in `BaseEngine` that I expect you may want to override all have names starting 'virt' (for virtual), 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.

If you want to ensure that you got this right, you can label the function 'override' in the class declaration (in the .h file), e.g. "`void virtSetupBackgroundBuffer() override;`". (A bit like Java's `@Override`.)

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` to the `MyDemoA` class 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.

Add Function

Function Name:

virtMouseDown(int iButton, int iX, int iY)

Return Type:

void

Access:

public

.cpp file:

MyDemoA.cpp

...

Comment:

Other Options:

☐ Inline

☐ Static

☐ Virtual

☐ Pure

OK

Cancel

When you choose Finish the following function will be created:

```
void MyDemoA::virtMouseDown(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)
    {
        lockBackgroundForDrawing();
        drawBackgroundRectangle(iX - 10, iY - 10, iX + 10, iY + 10, 0xff0000);
        unlockBackgroundForDrawing();
        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. (Look for the console window and you should see some text appearing.)

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).

Now there is a bit of a pain for SDL here, the graphics library which we are using. We are doing some drawing outside of the normal place where drawing occurs (we will see this in demo b). Because of this we need to tell the system that we want to draw to the surface by first locking it, then unlocking it afterwards. You don't need to do this normally if you just implement the various draw functions that I gave you, but do need to do it if you want to draw in unusual places, like here. If you draw to the background, first lock it, then draw to it, then unlock it. (In fact, I don't think that this will matter as long as you run this on windows, but you may have issues on Mac or Linux if you don't do this. SDL says to do it, so we will do it and follow the rules. In the normal drawing position I do it for you behind the scenes. In general, forget about this 'locking' once this demo is over. If you don't do this lock you should see a load of errors appear in the console telling you to lock the surface before drawing to it, so you will know if something went wrong and you needed to lock it.)

The redrawDisplay() 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 called BaseEngine (take a look at it if you wish). 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 virtMouseDown() 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 the virt 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)
    {
        lockBackgroundForDrawing();
        drawBackgroundRectangle(iX - 10, iY - 10, iX + 10, iY + 10, 0xff0000);
        unlockBackgroundForDrawing();
        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.

I note that mainLoop() also calls a number of other functions which you can override to implement different behaviour at different stages of the loop, without having to change the mainLoop function itself. These are there in case you need them, but most people will not.

Rendering to the screen

Usually your drawing code will be done within one of many virtual functions which are there for this purpose.

virtRenderScreen() should do all of the drawing. It first locks the surfaces, hence why you don't need to do this yourself normally. At the end it will unlock them again. It then calls the following methods, in this order, and you can put drawing code in any you wish:

```
virtPreDraw
virtDrawStringsUnderneath();
virtDraw() on each displayable object (see demo b), via a call to drawAllObjects()
virtDrawStringsOnTop();
virtPostDraw();
```

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

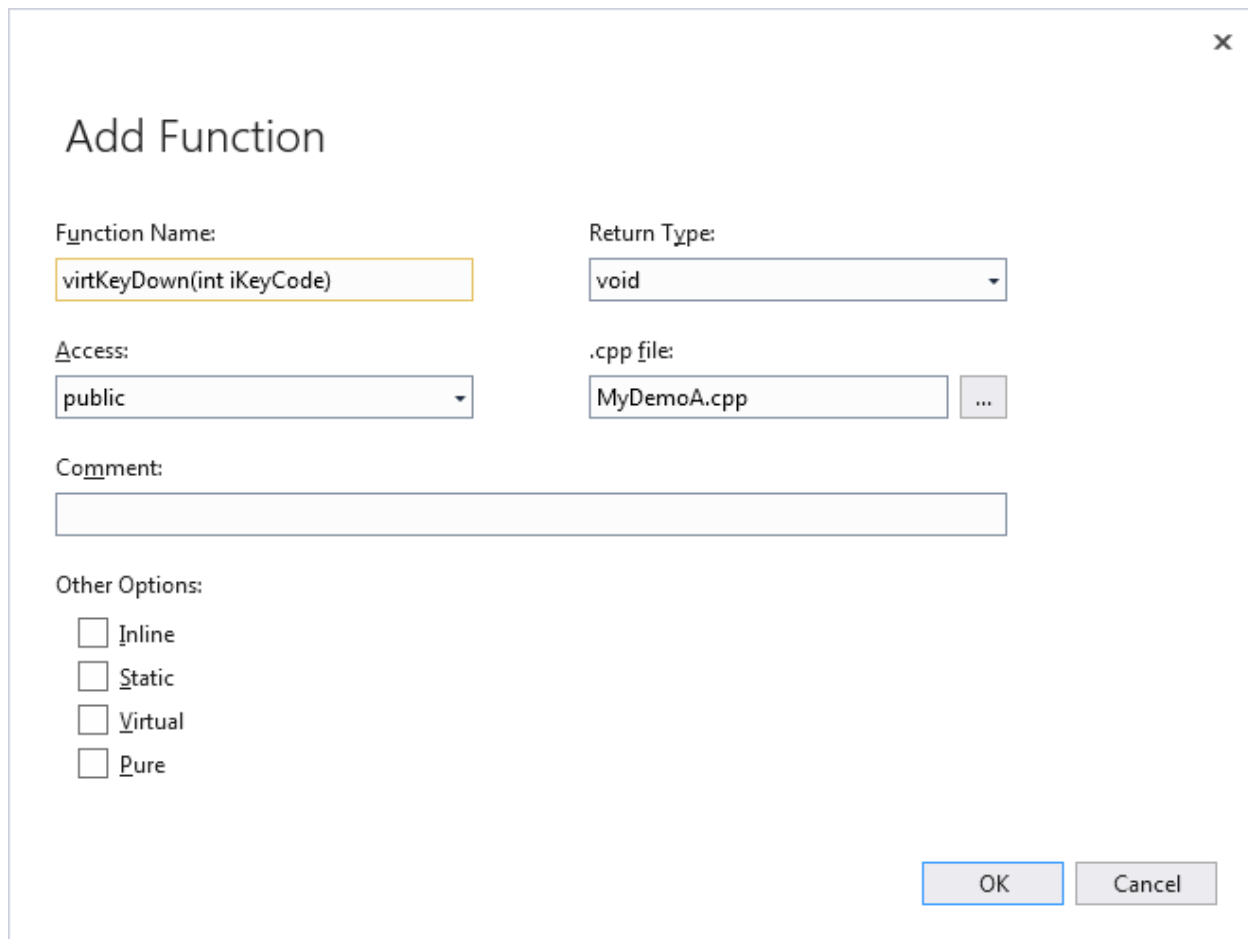
Rules:

- 1) If you are drawing a stationary/unchanging item then 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):



The image shows a 'Add Function' dialog box with the following fields and options:

- Function Name:** `virtKeyDown(int iKeyCode)`
- Return Type:** `void`
- Access:** `public`
- .cpp file:** `MyDemoA.cpp`
- Comment:** (empty text box)
- Other Options:**
 - ☐ `Inline`
 - ☐ `Static`
 - ☐ `Virtual`
 - ☐ `Pure`

Buttons: `OK` and `Cancel`

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 ' ':
            lockBackgroundForDrawing();
            virtSetupBackgroundBuffer();
            unlockBackgroundForDrawing();
            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.

Again we are drawing to the background in a slightly unusual place (i.e. not the usual virtual drawing functions) so first lock the background before you write to it, then unlock it afterwards. Otherwise you will again get thousands of warning messages in the consoles and it may not work properly on non-windows systems.

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.

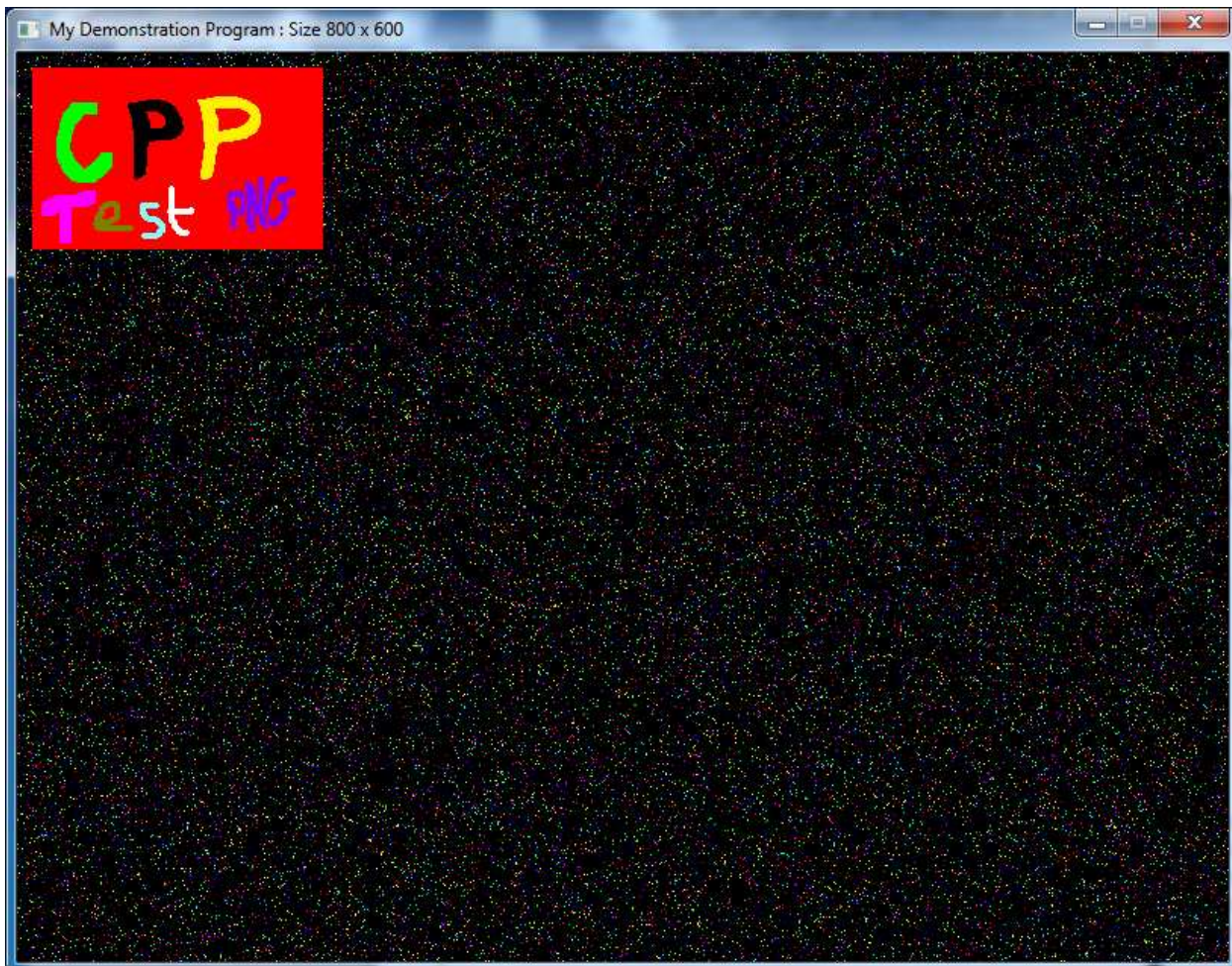
First load the image. A boolean parameter is supplied, which says whether to keep it permanently loaded. If you specify `false` for this then the image will only stay loaded while you have a pointer to it, then it will unload again. If you use `true` then the next time you call this method it will just 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 (see later lecture), 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. Note: if you take a pointer to the `SimpleImage` object and pass that around, it may break the system, so please use the object itself, not a pointer to it.

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

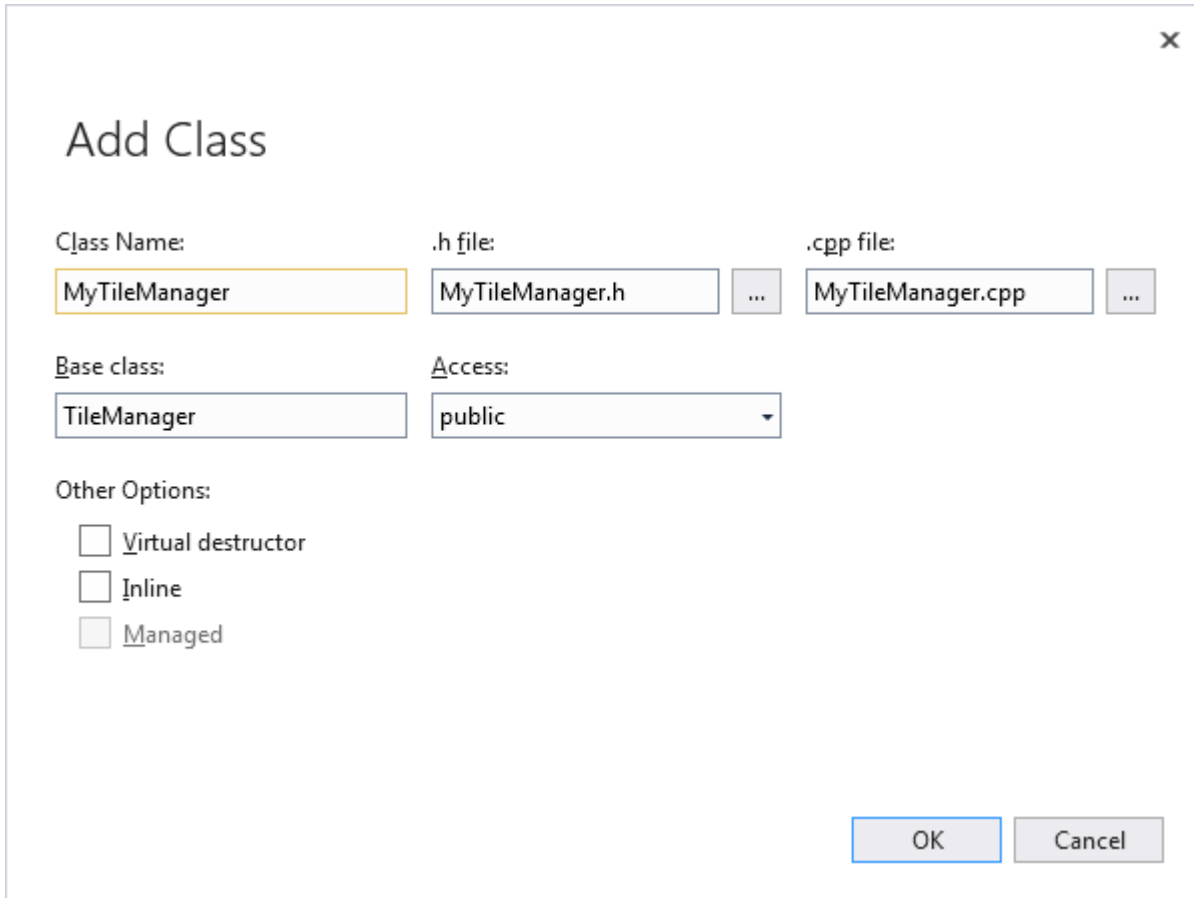
```
#include "ImageManager.h"
```



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, first thing to do with a new .cpp file, add to the top of the .cpp file:

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

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)`. Then when you tell it to draw the tiles, it looks at each tile in turn as asks your code how to draw that tile.

The base class (TileManager) needs a tile height and width to be provided to the constructor. If we create a subclass we need to provide that information ourselves. Let's set the width and height of a tile to 20 pixels each for the moment, and pass these to the base class constructor using the initialisation list. We can also (optionally, using a second constructor) specify how many columns and rows of tiles there are, so let's do that too. Find the constructor code in the .cpp file and alter it as follows:

```
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.

Your code should not compile – but nothing will have changed because you are not using the tile manager yet.

The key method for the TileManager 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 superclass (TileManager) into the new subclass, then replace the text "TileManager" by the new class name (MyTileManager in this case). Aside: I added override to the declaration in the header file, so that it tells me if I accidentally change something in future.

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

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

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  
}
```


Next you need to actually use the class, so you need an instance of this class. The easiest way is to create it as a member variable, so you get one automatically every time an instance of your BaseEngine subclass is created:

```
#include "BaseEngine.h"
#include "MyTilemanager.h" // Insert this line (to define the class)!!!

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

protected:
    MyTileManager tm; // Insert this line (for composition)!!!
};
```

Now go to your **virtSetupBackgroundBuffer()** function, which is where you draw the background, **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. You should see a grid of grey squares – your tiles.

Now the only thing you need to change is the function that actually draws the tiles to make it draw something more interesting.

Try changing the code in your virtDrawTileAt() method in your tile manager class to the following:

```
int iMapValue = getMapValue(iMapX, iMapY); // which was set to result of rand()
unsigned int iColour = (unsigned int)((iMapValue & 0xf00) << 12) // red
    + (unsigned int)((iMapValue & 0xf0) << 8) // green
    + (unsigned int)((iMapValue & 0xf) << 4); // blue
pSurface->drawOval (
    iStartPositionScreenX, // Left
    iStartPositionScreenY, // Top
    iStartPositionScreenX + getTileWidth() - 1, // Right
    iStartPositionScreenY + getTileHeight() - 1, // Bottom
    iColour);
```

This changes the code so that it draw ovals instead of squares, and changes the colours depending upon the map value of the tile – which you set in your virtSetupBackgroundBuffer() method so changes every time you press space.

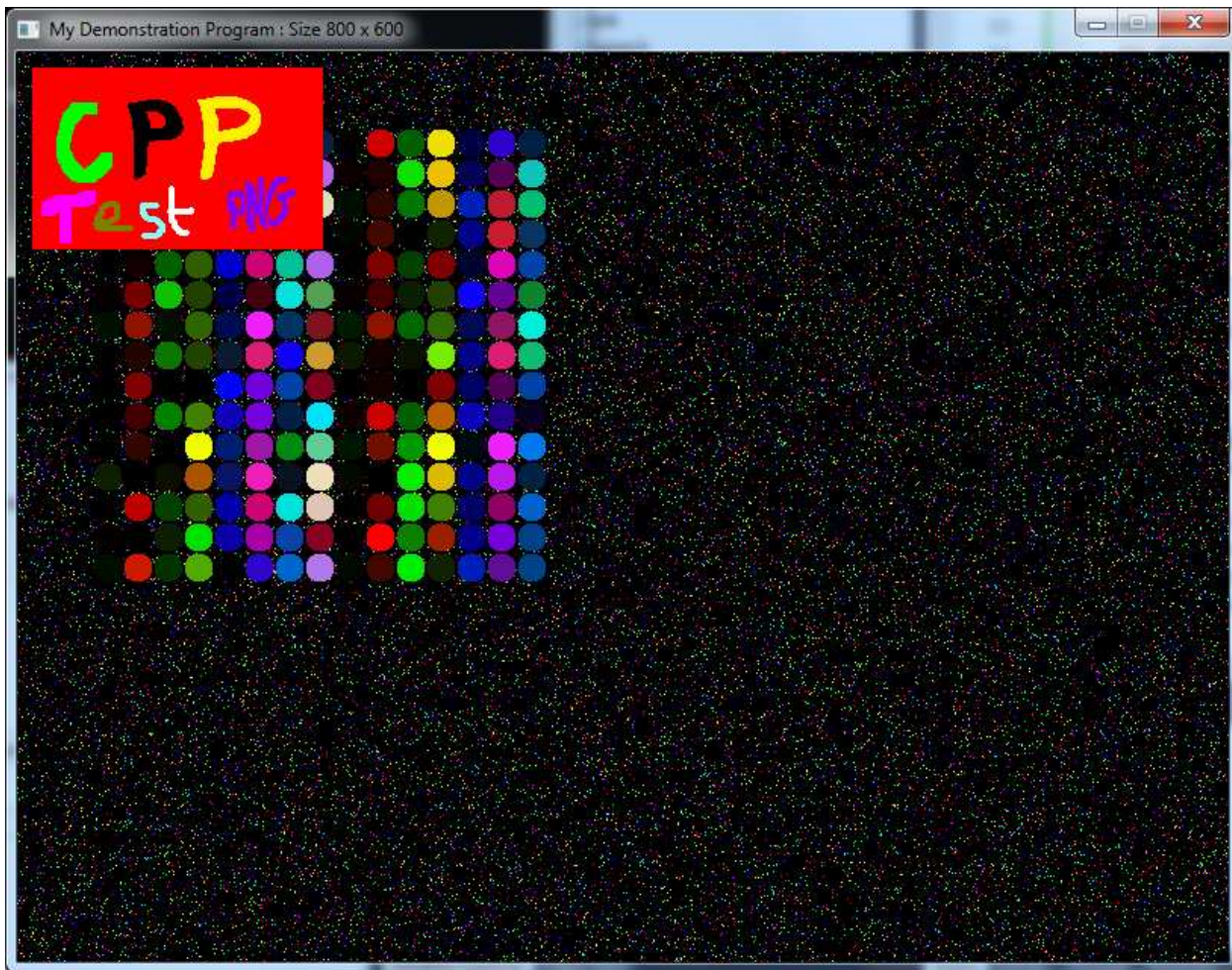
Run your code at this point and test it.

Ensure that you understand how the tile manager values work and what they are doing. Don't worry about how I get colour codes from the lowest 12 bits of the random number in the map tile value, as long as you understand what they are doing.

Finally, you can also implement a facility to change tile values when you click on them. Go to your code for the click handling (if you can't remember how you did this, look at the methods in your main BaseEngine sub-class) 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+rand(), this, getBackgroundSurface() );
    redrawDisplay(); // Force background to be redrawn to foreground
}
```

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



The full method that I made 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+rand(), 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, and at the main files, not the files for objects. 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.

From this demo you know a lot about how to use the framework, and will need to know:

- How to draw a background

- How to use a tile manager

- How to draw shapes and images to the background

The next demo shows you have to have moving objects. These move over the background so will be drawn to the foreground so that the background can be redrawn after they have moved.