C++ Coursework, Demo Tutorial Lab B – Moving objects

Completing the two demo tutorials should enable you to learn enough of the basics of using the framework to be able to understand the other various demos that have been supplied and to do the coursework. These demos also teach/emphasize a few standard C++ features.

Contents

Start from Coursework Lab A 1

Create your new class for your moving object 2

- Notice the problem on compilation 3
- Make the object appear 4

Create one of these new objects in the `BaseGame` sub-class 5

- Implement the virtInitialiseObjects() function 6

Make the object draw itself - Add a virt`Draw()` function to it 9

Drawing objects – important information 11

Making the object move 13

- Implementing the virtDoUpdate() function 14
- Prevent the object being moved off the edge of the screen 14

What to do next 16

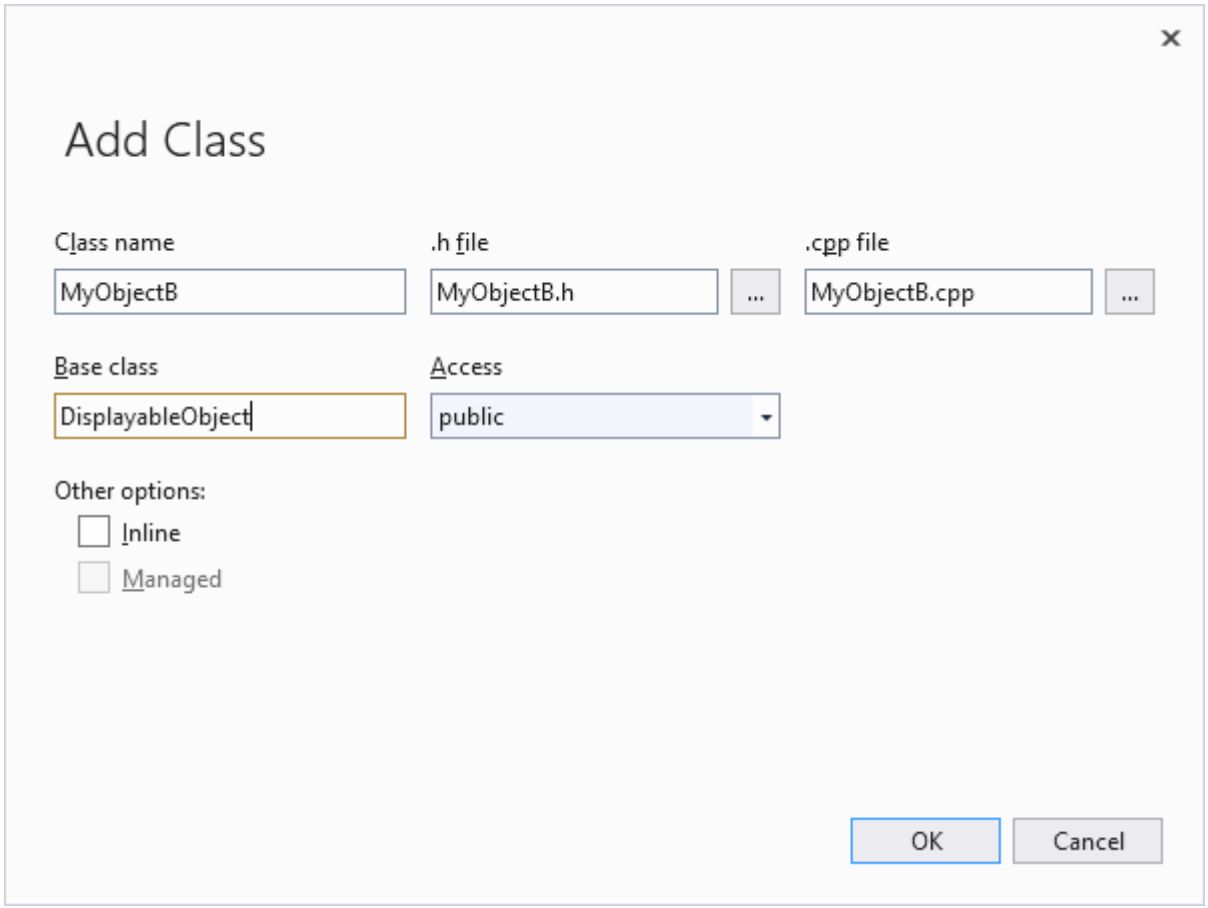
Start from Coursework Lab A

It is extremely important that you finish demo tutorial A first, otherwise this lab session will probably not make sense to you. Lab A gives you a lot of the background and basics. The aim of the first of these two lab demos is for you to learn the basics of how the coursework framework works. In the first session you see how to draw backgrounds, using tile managers and handling user input. In this second session you will learn about moving objects – drawing them and controlling them.

Load your files from the end of Lab A, since these will be your starting point.

Create your new class for your moving object

Create a new class call MyObjectB (for the object in Demo B), which is a subclass of DisplayableObject:



Here is the basic code which it should create:

```
#pragma once
#include "DisplayableObject.h"
class MyObjectB :
    public DisplayableObject
{
};
```

It also created an *almost* empty .cpp file.

Before you forget, add the include to the top of that file (the very first line!!!):

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

Notice the problem on compilation

There is a problem which may not be obvious unless you actually try to create an object. To see it, create a constructor for the objects:

```
#pragma once
#include "DisplayableObject.h"
class MyObjectB :
    public DisplayableObject
{
public:
    MyObjectB() {}
};
```

Then try to compile the code. It will not compile because the constructor is not valid. But why?

Here is the output when I try to compile, which tells me the problem:

```
Build started...
1>------ Build started: Project: CourseworkBase2020, Configuration: Debug Win32 ------
1>MyObjectB.cpp
1>G:\TEACHING\CPP_CW\CPPCoursework2021-v101\src\MyObjectB.h(7,14): error C2512:
'DisplayableObject': no appropriate default constructor available
1>G:\TEACHING\CPP_CW\CPPCoursework2021-v101\src\DisplayableObject.h(24,7): message :
see declaration of 'DisplayableObject'
1>Done building project "CourseworkBase.vcxproj" -- FAILED.
========== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped ==========
```

The key point is: **`'DisplayableObject': no appropriate default constructor available`**

Take a look at DisplayableObject and you will see that its constructors need parameters to be passed. In C++ we use the initialisation list for this. The base class constructor needs a pointer to the main program class (BaseEngine subclass). To handle this problem we need to accept this pointer in our own constructor and pass it on to the base class constructor manually. We will fix this now by changing the header file as follows:

```
#pragma once
#include "DisplayableObject.h"

class MyObjectB :
    public DisplayableObject
{
public:
    MyObjectB(BaseEngine* pEngine)
        : DisplayableObject(pEngine)
    {}
};
```

Note that I left this code in the .h file because it doesn't really do anything so is safe to inline. You could split it between the .cpp and .h file if you wish, and it may be a useful exercise for you to try.

It should now compile correctly, but the object will not appear if you run the program (you should still see the dotted background from the previous lab though).

Make the object appear

We need to tell your MyDemoA class to display the objects, but first we should implement the constructor to set the initial position and the size of the object. These are all members of the base class which are accessible to sub-class objects (protected), so you can assign to them in the constructor. Add the following to the body of the constructor, between the { }

```
	m_iCurrentScreenX = 100; // Starting position on the screen
	m_iCurrentScreenY = 200;
	m_iDrawWidth = 100;	// Width of drawing area
	m_iDrawHeight = 200;	// Height of drawing area
}
```

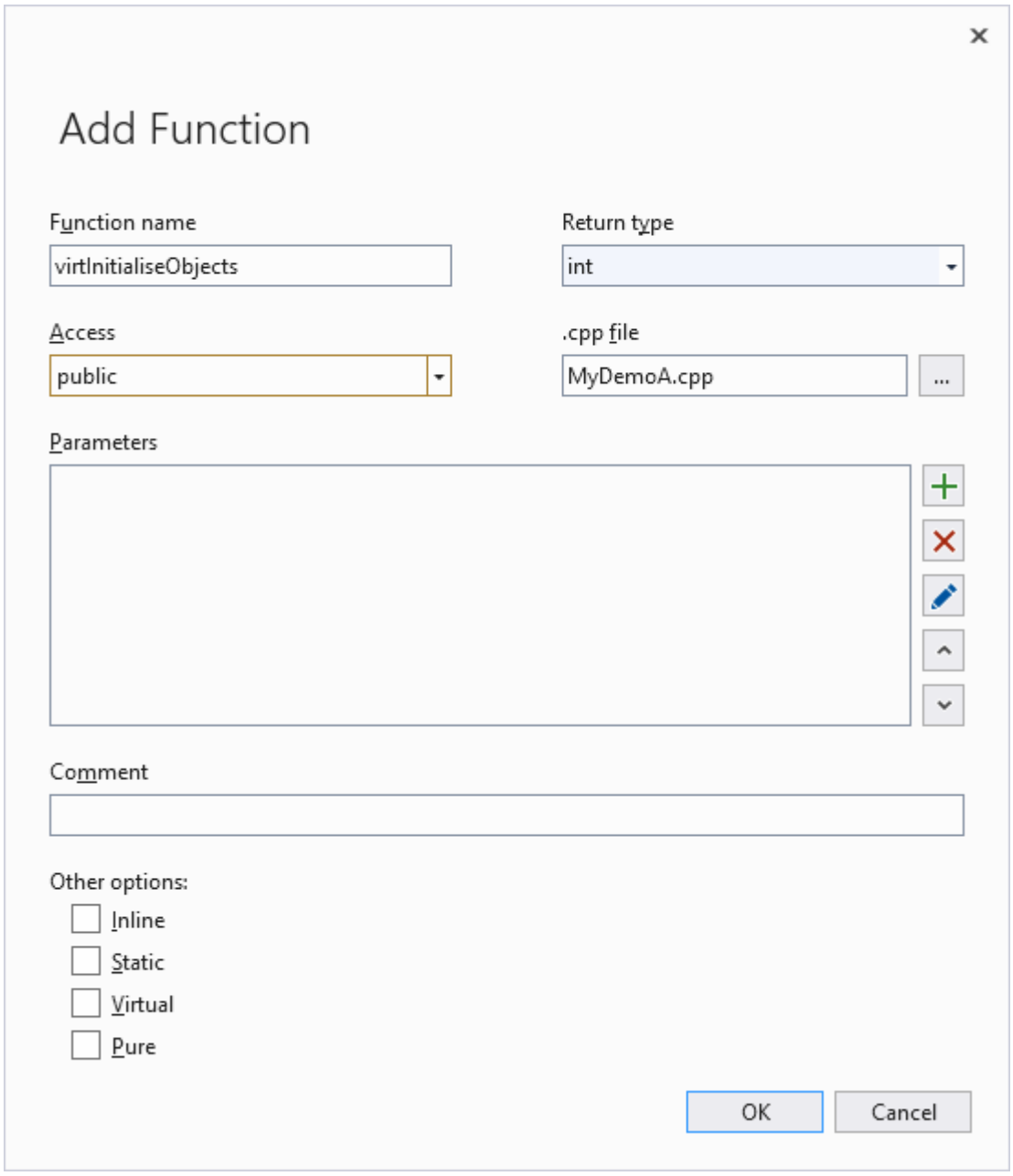
OR you can pass these values to the alternative form of the base class constructor to avoid having to set the attributes yourself:

```
MyObjectB::MyObjectB(BaseEngine* pEngine)
	: DisplayableObject(100, 200, pEngine, 100, 200, true )
{
}
```

In this alternative form, the first two parameters are the starting x and y coordinates, then you get the pointer to the main class (which you added previously), then the width and height. The final parameter specifies whether the object is drawn relative to the centre of the object or the top left. If you specify true then the coordinates you specified are the top left of the object. If not then the coordinates are the centre of the object – i.e. the object is drawn at an offset from those coordinates. Some demo examples use one form and some the other. For example, if you want to check for collision between two circles then it is sensible to maintain the centre of the circles rather than the top left of the square around them. Experiment to see how these work.

Create one of these new objects in the `BaseGame` sub-class

Implement the virtual function virtInitialiseObjects() in the MyDemoA.h class (NOT the MyObjectB class!). This has an `int` return value and no parameters:



Go to the DemoAMain.cpp file and find the new function:

```
int MyDemoA::virtInitialiseObjects()
{
	// TODO: Add your implementation code here.
	return 0;
}
```

This is a really important function. You need to create all of the objects which will be moving, and store pointers to them in the array. You will now add some code to it to create an object of type `MyDemoA`.

First, go to the top of the .cpp file **for MyDemoA**, and add a `#include` for the header file, so that the top of the file looks like this:

```
#include "header.h"
#include "MyDemoA.h"
#include "ImageManager.h"
#include "MyObjectB.h"  // This is the new line!
```

You need to add the `include` so that the compiler knows what a `MyObjectB` is when you use it.

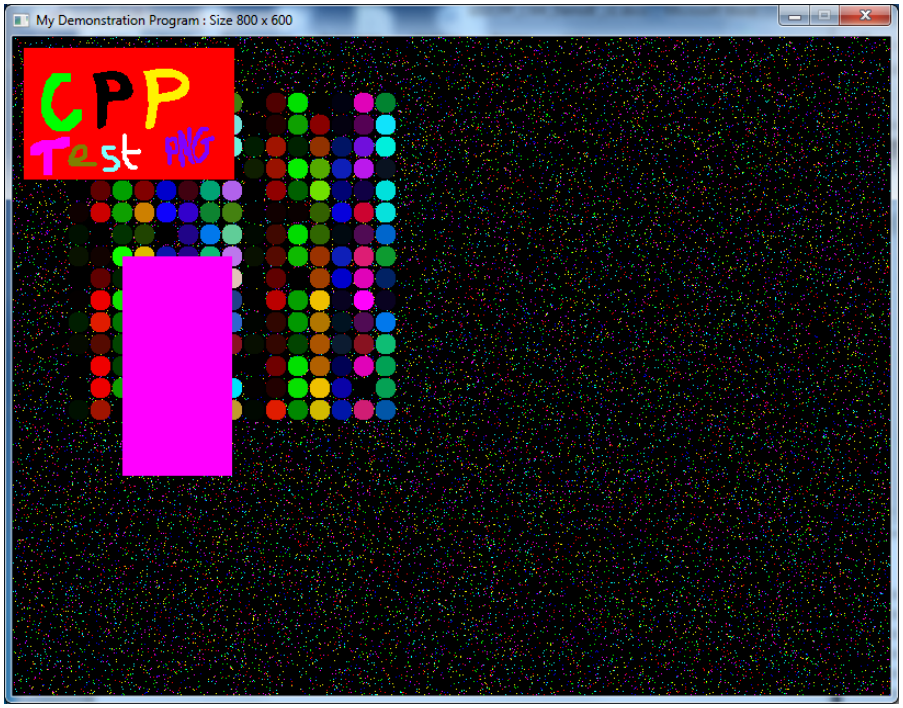
Implement the virtInitialiseObjects() function

Now add the following implementation to the `InitialiseObjects` function:

```
int MyDemoA::virtInitialiseObjects()
{
	// Record the fact that we are about to change the array
	// so it doesn't get used elsewhere without reloading it
	drawableObjectsChanged();

	// Destroy any existing objects
	destroyOldObjects(true);

	// Creates an array big enough for the number of objects that you want.
	createObjectArray(1);

	// You MUST set the array entry after the last one that you create to NULL,
	// so that the system knows when to stop.
	storeObjectInArray(0, new MyObjectB(this));

	// NOTE: We also need to destroy the objects, but the method at the
	// top of this function will destroy all objects pointed at by the
	// array elements so we can ignore that here.

	setAllObjectsVisible(true);
	return 0;
}
```

The key parts here are the createObjectArray, which creates a (static sized) array of objects, and the storeObjectInArray. The system will notify all objects in the array every few milliseconds so that they have a chance to move themselves or perform other actions, and then will draw each object if it needs redrawing. **If you do not put objects in the array, they will not be asked to update themselves and will not be drawn!**

Run the program and you should see the default purple colour for object drawing:



`virtInitialiseObjects` does the following:

- Firstly it records that you have changed the drawable objects array. You MUST do this at the start of any implementation of this function, in case this function is being called while the object array is being iterated. The iteration methods check whether this value gets set and if so abort and restart the iteration, avoiding the issue of using a non-existent object.
- Next it looks at any objects which are stored in the array already and deletes them by calling the `destroyOldObjects()` function. This means that you never need to worry about destroying the objects, just call this function and it will do it for you.
  Aside: usually this is correct behaviour, but if you need the objects to NOT be destroyed for you (e.g. you don't want to keep re-creating them) then you should remove them from the array before you call this function, OR you can set the deleteOnRemoval method to return false by overriding the function in your object class:

  ```
  bool deleteOnRemoval() { return false; }
  ```

- Now it creates a new array of `DisplayableObject` pointers by calling the createObjectArray() function. You need to tell it how big to make this array, so make it big enough for all of the objects you will store in it.
- Finally, it sets the elements of the array to point to new objects of the correct type. In this case one `MyObjectB`. You should call the `storeObjectInArray`() function to store the object pointer for objects that you create.
- Finally, the very last thing this code does is set all objects to be visible. The objects could instead set themselves as visible in their constructors (or elsewhere) if you wish, and then you could not need to do this.

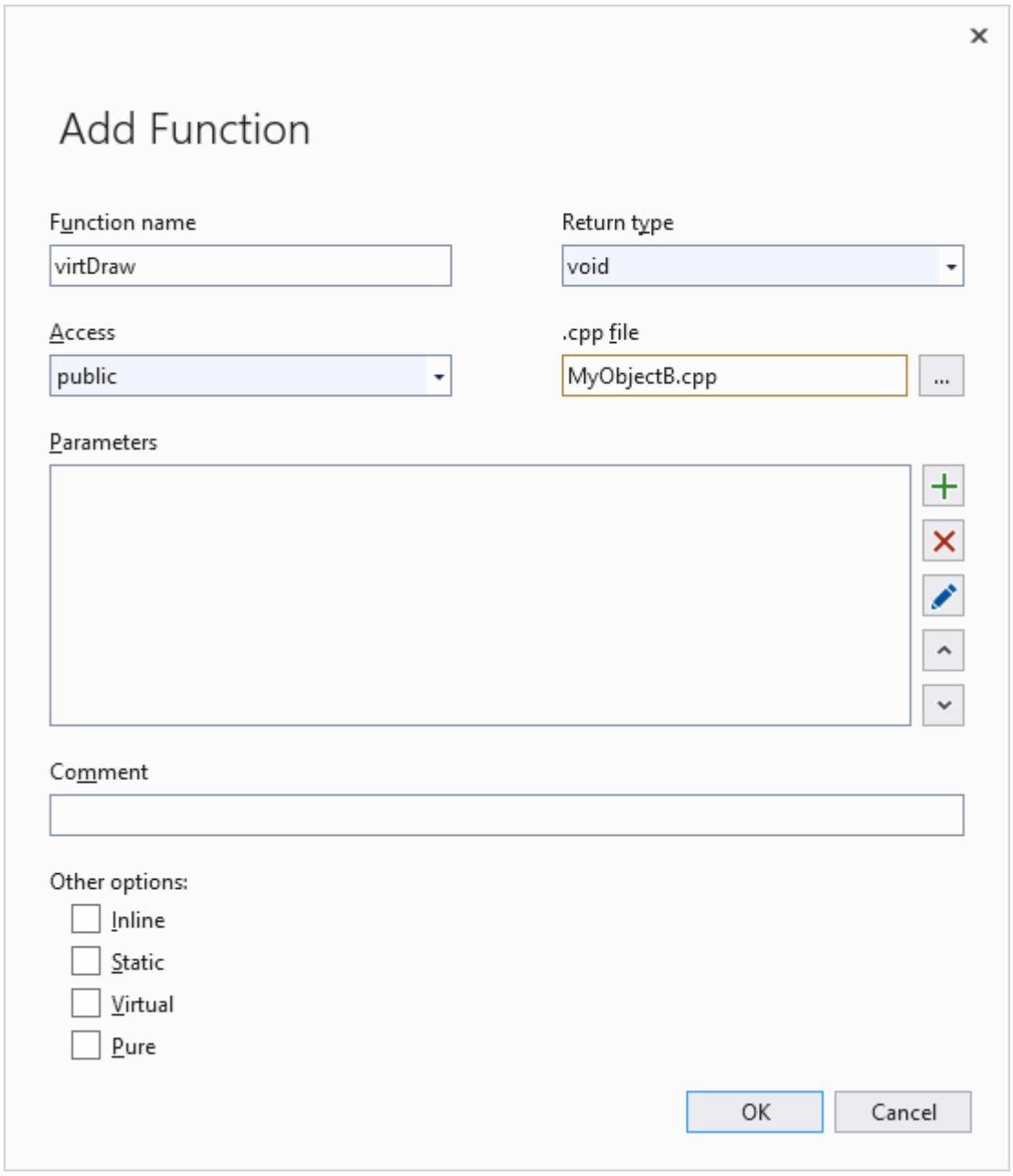
Example: If you wanted to create 3 objects you would just create an array of size 3 and set the three pointers to point to the new objects. Note that if we created multiple objects at the moment they would be in the same place, so it would be pointless (because they are set to a fixed place by the MyObjectB constructor).

Note that you MUST use `new` (not malloc()) to create these objects since `destroyOldObjects()` will use **delete** on them for you. DO NOT USE MALLOC()!!!

Build and test your program and it should look something like this, showing the new object:

Make the object draw itself - Add a virtDraw() function to it

Add a function called virtDraw() with no parameters and void return type to the MyObjectB class. (Right click on the class in Class View and choose “Add” and “Function”.)



Note: Remember to change the .h file to be a .cpp file, to use the MyObject.cpp file.

Go to the .cpp file and add the #include “header.h” to the top if you have not already done so.

We want to use some methods of BaseEngine so add an include for that header file at the top of your .cpp file, below the include for header.h:

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

You should now have something like this in your .cpp file:

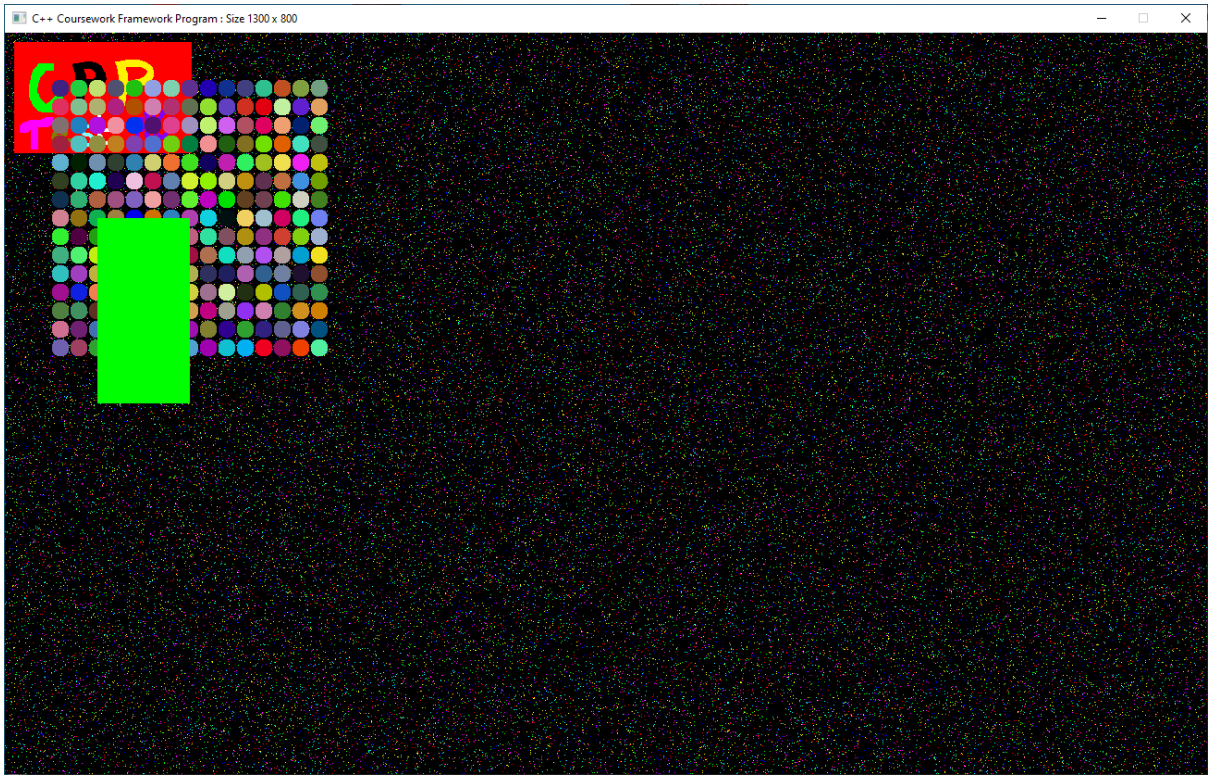
```
#include "header.h"

#include "BaseEngine.h"

#include "MyObjectB.h"

void MyObjectB::virtDraw()
{
	// TODO: Add your implementation code here.
}
```

Your program should build but the object will no longer appear, because the function to draw itself now does nothing.

Now add an implementation of the `virtDraw()` function as follows:

```
void MyObjectB::virtDraw()
{
    getEngine()->drawForegroundRectangle(
        m_iCurrentScreenX, m_iCurrentScreenY,
        m_iCurrentScreenX + m_iDrawWidth - 1,
        m_iCurrentScreenY + m_iDrawHeight - 1,
        0x00ff00);
}
```

Run the program now and test it – you should see that the colour of the object has changed.



Drawing objects – important information

It is important to ensure that you only draw within the region that you specified by various variables – which will have been set based on what you did in your constructor. The drawing region is specified by:

The top left corner has the coordinates:

```
X:   m_iCurrentScreenX + m_iStartDrawPosX
Y:   m_iCurrentScreenY + m_iStartDrawPosY
```

The bottom right corner has the coordinates:

```
X:   m_iCurrentScreenX + m_iStartDrawPosX + m_iDrawWidth
Y:   m_iCurrentScreenY + m_iStartDrawPosY + m_iDrawHeight
```

The only difference between making the points the top left or centre of the object is what values `m_iStartDrawPosX` and `m_iStartDrawPosY` are set to. In this case, we set the start draw position to `0,0` so the `CurrentScreen` values specify the top left corner. If you want to make the `CurrentScreen` values specify the middle of the object then instead you could set `m_iStartDrawPosX` to `-m_iDrawWidth/2`, and similarly for `m_iStartDrawPosY`. In other words, the `StartDrawPos` allows you to change the drawing of the object relative to the logical position of the object on the screen. If you don't need to do this then you can always set the `StartDrawPos` values to `0`, but will have to remember that the position of the object is for the top left corner of the object.

Knowing this, the following code draws a rectangle filling the whole drawing area:

```
getEngine()->drawForegroundRectangle(
        m_iCurrentScreenX, m_iCurrentScreenY,
        m_iCurrentScreenX + m_iDrawWidth - 1,
        m_iCurrentScreenY + m_iDrawHeight - 1,
        0x00ff00);
```

- The `-1` offset values are needed, because, for example, a rectangle of width `100`, with the left side at position `0`, would fill values from `0` to `99` (i.e. 100 pixels).
- The `0xff00` is a green colour, as for the background colours in demo tutorial A.
- `drawForegroundRectangle()` means to draw the rectangle to the foreground. There is also a Background version of the function. Please review Coursework Lab A if you cannot remember the difference between the foreground and the background. Moving objects should be drawn to the foreground so that they can be 'undrawn' from their old positions when they move.
- `getEngine()` retrieves a pointer to the `BaseEngine` object. In the constructor you took a pointer of this type and passed it to the base class constructor. The base class constructor stored that pointer for you. When you call `getEngine()` you are retrieving the pointer which was stored. In this way you can call a function which is on the `BaseEngine` class (i.e. `drawForegroundRectangle`) even when you are in the `DisplayableObject` subclass.

The area to draw within is defined using the following equations, so the values of `m_iCurrentScreenX` and `m_iCurrentScreenY` are stored and this is the area of the screen which will be redrawn when the object moves, removing it from its old location:

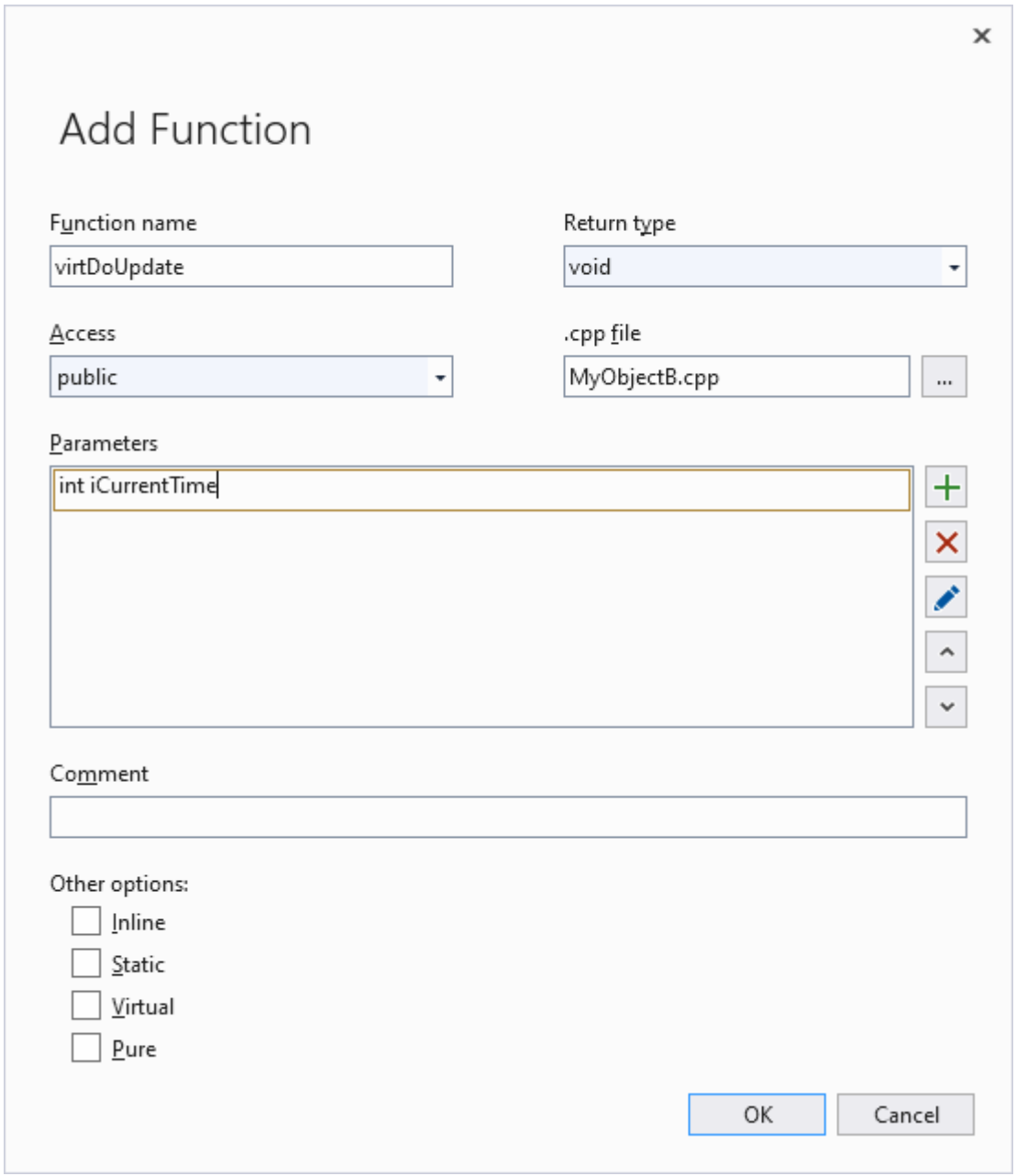
```
X from        m_iCurrentScreenX + m_iStartDrawPosX
To            m_iCurrentScreenX + m_iStartDrawPosX + m_iDrawWidth
Y from        m_iCurrentScreenY + m_iStartDrawPosY
To            m_iCurrentScreenY + m_iStartDrawPosY + m_iDrawHeight
```

These variables tell the framework where your object is and how big it is. You can set these variables to whatever you want, but you must not draw outside of this area.

Making the object move

To move objects you need to implement their virtDoUpdate() methods.

Add a new function to MyObjectB called virtDoUpdate(int iCurrentTime), with return type `void` and one parameter, of type `int`, with the name `iCurrentTime`. This parameter tells the object what the time is, so that it can alter how far it moves depending upon how long it was since it was last asked.



You should then have this code generated:

```
void MyObjectB::virtDoUpdate(int iCurrentTime)
{
    // TODO: Add your implementation code here.
}
```

The purpose of this function is to update the `CurrentScreenX` and `CurrentScreenY` variables, to change where the object will be drawn.

Implementing the virtDoUpdate() function

Add the following implementation for the function:

```
void MyObjectB::virtDoUpdate(int iCurrentTime)
{
	// Change position if player presses a key
	if (getEngine()->isKeyPressed(SDLK_UP))
		m_iCurrentScreenY -= 2;
	if (getEngine()->isKeyPressed(SDLK_DOWN))
		m_iCurrentScreenY += 2;
	if (getEngine()->isKeyPressed(SDLK_LEFT))
		m_iCurrentScreenX -= 2;
	if (getEngine()->isKeyPressed(SDLK_RIGHT))
		m_iCurrentScreenX += 2;

	// Ensure that the objects get redrawn on the display
	redrawDisplay();
}
```

Importantly, if you change any values you must call `redrawDisplay()` at the end of this function to tell the system that you changed something that means it needs to redraw the screen. This will ensure that the `virtDraw()` function is called to draw the object in its new position. It will also ensure that the object is 'undrawn' from its old position.

Compile and test the program, trying to move the object using the cursor keys.

This example illustrates how you can check whether a key is currently pressed. You saw in the previous lab that you can implement a function which is called when a key is pressed. Instead this function will check whether a key is already pressed down. For example, if a key is pressed twice then the virtOnKeyDown function would get called twice. On the other hand the `isKeyPressed()` function lets you check at any time whether the key is currently pressed down or not, not how many times it has been pressed.

The implementation here checks whether the cursor keys are pressed. If a key is pressed then it changes the position of the object by 2 pixels.

Prevent the object being moved off the edge of the screen

At the moment you can move the object off the edge of the screen (momentarily; it will probably crash). You can add code to stop this quite easily, as show below:

```
void MyObjectB::virtDoUpdate(int iCurrentTime)
{
	// Change position if player presses a key
	if (getEngine()->isKeyPressed(SDLK_UP))
		m_iCurrentScreenY -= 2;
	if (getEngine()->isKeyPressed(SDLK_DOWN))
```

```
            m_iCurrentScreenY += 2;
      if (getEngine()->isKeyPressed(SDLK_LEFT))
            m_iCurrentScreenX -= 2;
      if (getEngine()->isKeyPressed(SDLK_RIGHT))
            m_iCurrentScreenX += 2;

      if (m_iCurrentScreenX < 0)
            m_iCurrentScreenX = 0;
      if (m_iCurrentScreenX >= getEngine()->getWindowWidth() -
m_iDrawWidth)
            m_iCurrentScreenX = getEngine()->getWindowWidth() -
m_iDrawWidth;
      if (m_iCurrentScreenY < 0)
            m_iCurrentScreenY = 0;
      if (m_iCurrentScreenY >= getEngine()->getWindowHeight() -
m_iDrawHeight)
            m_iCurrentScreenY = getEngine()->getWindowHeight() -
m_iDrawHeight;

      // Ensure that the objects get redrawn on the display
      this->redrawDisplay();
}
```

Again compile and execute this to test it.

That completes the tutorial on moving objects. You should now be able to do a lot of the requirements of the coursework part 3. Experiment with the samples to learn more.

What to do next

Please try the various demos. You can do this by commenting in/out the relevant lines in `main()` in `mainfunction.cpp`. Ensure that only one is active at a time though, otherwise you will have two objects with the same name.

I suggest that you start with the SimpleDemo then the `BouncingBall` sample, then move on to MazeDemo when you understand the other demos. These demos give progressively more code to consider and more complex examples, building up to relatively complex programs. Using what you have learned you should be able to understand these.

Be aware that there are various subclasses of DisplayableObject that I already created for you to simplify matters – see ExampleObjects.h

Be aware that I provided a few classes to avoid you having to do simple maths – look at UtilCollisionDetection (collision detect bounding rectangles or circles for objects) and UtilMovementPosition (work out where to draw an object using linear interpolation, based on a start and end position and time.

Each of the 'advanced demos' shows you how to do specific things which may be useful for CW part 2, but not for part 1. The Dragging and Zooming demos show you how to do simple dragging and zooming/scrolling and work by mapping coordinates from a virtual drawing position to a real drawing position.

You can do various things with images by providing some kind of point mapping when you draw the image – e.g. to rotate the image. Basically you give it a CoordinateMapping object which will map from a starting coordinate to a position in the image.

StarfieldDemo and FlashingDemo show you how you could switch out multiple backgrounds.

ImageMappingDemo shows you how you can rotate or otherwise manipulate images.

I suggest to read the requirements for both parts 1 and 2 and to skim the FAQ document because it has a lot of useful information about various parts of the coursework. Knowing what is coming up later and how to find the information may save you a lot of time in the long run.