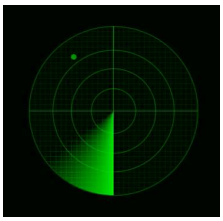


The Radar Display Project

25% of the Semester's Laboratory Assessment. Your work must be submitted via the Moodle link: Late submission without an ECF will receive the UoN mandated mark penalties.

Please note the distribution of marks for the sub tasks: They are not intended to be proportional to the "difficulty"

Overall Learning Theme of the Exercise:

Strategical Purpose: *To start to encapsulate related data into class objects and identifying the relationships between different object types is a critical design activity.*

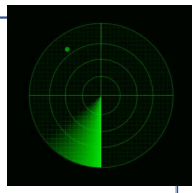
Focussed objective: *To understand the syntax of C++ for classes and realise that C++ syntax is mostly C expect for the object oriented "extras".*

Focussed objective: *To practice the use of pointers and references when passing data around.*

Why are we doing this? A major objective of this module is to understand why and how *object-oriented programming techniques* are used in modern software. This exercise is our first hands on experience of designing and implementing basic objects of our own. Furthermore, we will explore how we can build more objects from those objects etc

Focussed Learning Objectives:

- To *think* about the basic data that makes an object what it is and how to *encapsulate* that within one entity.



Graphics Pixels: We are working in a team developing a computer visualisation program for displaying air vehicles on a radar screen. Each vehicle will be displayed as a small icon made up of a number of pixels. There will exist a *rendering* function that if passed the data characterising a particular vehicle, will actually display it on the screen.

Our role is to develop the basic *pixel* and *icon* objects

Pixels: Each pixel is to be characterised by its floating-point x and y coordinates describing where it will appear on the screen and integer in the range 0-20 that gives its brightness on the screen.

The following sub task counts **10%** of the mark for this exercise

Sub Task Specification: Develop a C++ class to *encapsulate* the data associated with the concept of a graphics pixel.

Reflection Points:

Do we get the concept of *encapsulation*? Review slides "Object Oriented Design" maybe.

Design Decisions: Is it obvious what data each *instance* of the class will hold? I think so here.

Do we understand the mechanics of *declaring* to the computer that "objects called pixels exist" – "Classes in C++" , slide 3 maybe?

Do we understand the mechanics of *defining* to the computer what makes up those "objects called pixels that we know exist" –"Classes in C++" slide 5 maybe?

Testing Strategy: You were not explicitly told to write a *main* program here, but of course you will won't you – how else are you going to test if pixel object was correctly implemented? How do we test this? Well, can we actually create a pixel? Maybe check that the following compiles?

```
int main() {
        pixel myFirstEverPixel;
}
```

Gotcha – of course you have put the class definition before main in the file as well!

Gotcha – of course you did do this in a CodeBlocks C++ project containing the single file *main.cpp*? Obviously, *classes* are not a C thing so wouldn't be understood if we did it in a Codeblocks C project using a file *main.c*.

A subtlety: The brightness value lies in the range 0-20. Did you choose to store it in an *int*? Is that not wasteful of memory? Think big video games etc, rather a lot of pixels – just how much RAM can you afford?

The following sub task counts **20%** of the mark for this exercise

Sub Task Specification: Write a main program that creates an instance of a pixel call *aPixel* and then sets its x and y coordinates to 3.9 and 4.1 and its brightness to 7. Confirm this has successfully happened by printing to the screen the details of the data contained within *aPixel*. The format is up to you – decisions, decisions, what do you regard as appropriate here?

Focussed Learning Objective:

- Accessing the data within instances of our class

Reflection Points:

Are you really sure what is meant by the different *terminology*, an *instance* of a *class?* "Classes in C++" slide 7 maybe? Not sure on the syntax for accessing the data within an instance? "Classes in C++" slide 6 maybe?

Gotcha – weird compiler errors about something being inaccessible? Did you recall to include the *public: (colon not semi colon).* "Classes in C++" slide 9 maybe?

Gotcha – forgot the semi-colon at the end of the class definition ("Classes in C++" slide 9) – that will cause an error.

Much of the motivation for object-oriented approaches is the ability to pass around associated elements of data as one entity. We will now explore this.

The following sub task counts **20%** of the mark for this exercise

Sub Task Specification: Write a *function* called *showPixelDetail* that will be passed an i*nstance* of a *pixel* as it's argument. The purpose of this function is to report to screen the contents of the *pixel* it has been given. Write a suitable test main to check that this function works.

Focussed Learning Objective:

- Passing entire objects to functions

Reflection Points:

Did I really need to explain above that **"**The purpose of this function is to report to screen the contents of the *pixel…"* given that the function is called *showPixelDetail?* Well reflect on this very well. I could have named the function *func1* – debugging complicated codes is hard enough, so why not provide as much help to the poor fool debugging your code as you can by choosing helpful, ideally explanatory names! By the way, note how software people often create names from multiple words using a capitalisation strategy or else using underscores, *show_pixel_detail,* because we cannot leave blanks can we! Just a convention, but tried and proven in practice.

What is the return value of the function? Does it need to return anything given its specification? Make and take responsibility for an informed professional decision: If no return value, then what do we do?

Hint: Life is too short to keep writing everything from scratch, so if I were you, I think my test code would be a modification of the *main* of the previous task. Of course, we'd keep of unmodified copy of it as well.

The following sub task counts **10%** of the mark for this exercise

.

Sub Task Specification: Please design and implement a *class* that encapsulates the concept of an *icon* in the context described above. Each *icon* is to contain up to 16 pixels and an integer that holds the *id* number of the vehicle it represents. Ensure the functionality described below is implemented and thoroughly tested. It goes without saying that suitable levels of in code comments are required throughout.

Focussed Learning Objectives:

- To appreciate how objects that we introduce can themselves contain instances of other objects.
- To start to appreciate how the *design* of objects can take more time than their final coding!

Reflection Points:

Obviously, an icon instance must hold its *id*, which we know is an integer. What is not so obvious is how *icons* can contain *pixels.* Well, if we were just required to put one *pixel* in each icon, we could do this as

```
class icon {
public:
        pixel thePixel;
};
```

There is no reason why our own previously defined objects such as *pixels* cannot then be used to build up other, probably more complicated, objects, such as *icons.* In fact, that is a key capability we really like – we can progressively construct very complex codes and objects form a sequence of less complicated objects.

But an *icon,* must actually contain 16 pixels. Well an array then?

```
class icon {
public:
        pixel thePixels[16];
};
```

Design decisions. How do we get the data into the 16 pixels within an *instance* of an *icon?*

The following sub task counts **10%** of the mark for this exercise

Sub Task Specification: Write a main program that creates an *instance* of an *icon* called myIcon. Create an *instance* of a *pixel* called pixelOne in main and give it some suitable data values for the purpose of testing (?). Set the data of the first pixel of myIcon to have the same values as that of pixelOne.

Reflection Points:The last line requires some thinking about! pixelOne is an actual pixel we created in *main*. The 16 pixels within myIcon are 16 different pixels that exist. We are not asking that one of the latter *is* pixelOne, rather it is to have the same data as pixelOne. Think about the difference!

I propose we take an educated guess here! If I have two *pixels* in a main code such as

```
int main()
        pixel pixelOne;   // Create a pixel
        …give pixelOne some data values
        pixel pixelTwo; // Create another pixel
        pixelTwo=pixelOne;
};
```

```
int main()
        float floatOne;   // Create a float
        …give floatOne a data value
        float floatTwo; // Create another float
        floatTwo=floatOne;
};
```

By comparison with the more familiar example of floats, we can hope that the compiler understands the "=" sign as make the data values of *pixelTwo* the same as those of *pixelOne.*

Try it - does it work[1]?

Lets us illustrate how we build up codes by gluing together smaller parts.

The following sub task counts **10%** of the mark for this exercise

Sub Task Specification: Create an *icon* instance with *id* number 7 and initialise its 16 pixels to have x, y and brightness values of 1,1,15 2,2,15 3,3,15 etc. This would be a bright diagonal line on the screen? Pass the whole *icon* instance to a function called *showIconDetail* which uses the *showPixelDetail* function above to list the details of the icon's pixels.

However, for reasons of code speed, it is specified that we must not generate a temporary copy of the icon instance argument when we call *showIconDetail.*

Reflection Points:

How can we *choose* (a professional decision) to pass data to functions? By value, pointer or reference. What is requested here? "Passing Arguments to Functions " slide 12 maybe?

One of guiding design principles ought to be to *hide detail* so that we can "see the wood for the trees"

The following sub task counts **10%** of the mark for this exercise

Sub Task Specification: I would prefer that for the task just done that the details of creating the pixels were not in the main code, rather hidden away inside another function called *initialiseAsDefaultDiagonalLine*. Implement this please.

```
int main() {
        icon myIcon;
        initialiseAsDefaultDiagonalLine(myIcon);
        showIconDetail(myIcon).
}
```

My intention here is that the line *initialiseAsDefaultDiagonalLine(myIcon);*

calls a function within which we place the detailed code that used to be in *main.* Why bother? Clarity; the main is now a simple 3 liner – obvious big picture: Make an icon, give it some default data values and show it.

Reflection Points:

We must be very careful with how we pass an icon as an argument to *initialiseAsDefaultDiagonalLine*.

[1] It will, but we shall shortly discuss in lectures why this is *risky!*

Previously with *showIconDetail* we made a decision because it was *preferable* for speed reasons not to make a copy of the argument. Showing the contents of a copy of the argument actually prints the same on the screen as would showing the contents of the argument. However here, do we want to change the pixel data of a copy of the argument icon, or of the actual argument icon? Think carefully and see "Passing Arguments to Functions " slides 10 & 12.

We shall now conclude today by completing the overall design. Moreover,

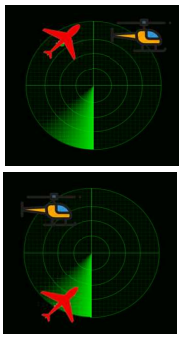
Focussed Learning Objective:

- To appreciate how it is rare to work on a code on your own. Here you and I are a *team* and I have already started a solution and you are required to continue it. Oh dear, I didn't document it that well; now you know what it's like to have to work out someone else's code with inadequate comments!

The following sub task (and its continuation below) counts **10%** of the mark for this exercise

Sub Task Specification: Design and implement the top-level object in our software solution which is to be an instance of a *radarDisplay* class. A *radarDisplay* instance "*refers to*" a number of *icon* instances. We don't know how many *icon* instances it will actually need to hold so must allow this to be variable; but we are prepared to limit this to 100 in the code. A further complication is that different people in the airport control tower will be looking at different instances of *radarDisplays,* that show different regions of the airspace and if a particular air vehicle appears on multiple displays, they must *share* the same *icon* instance that represents it.

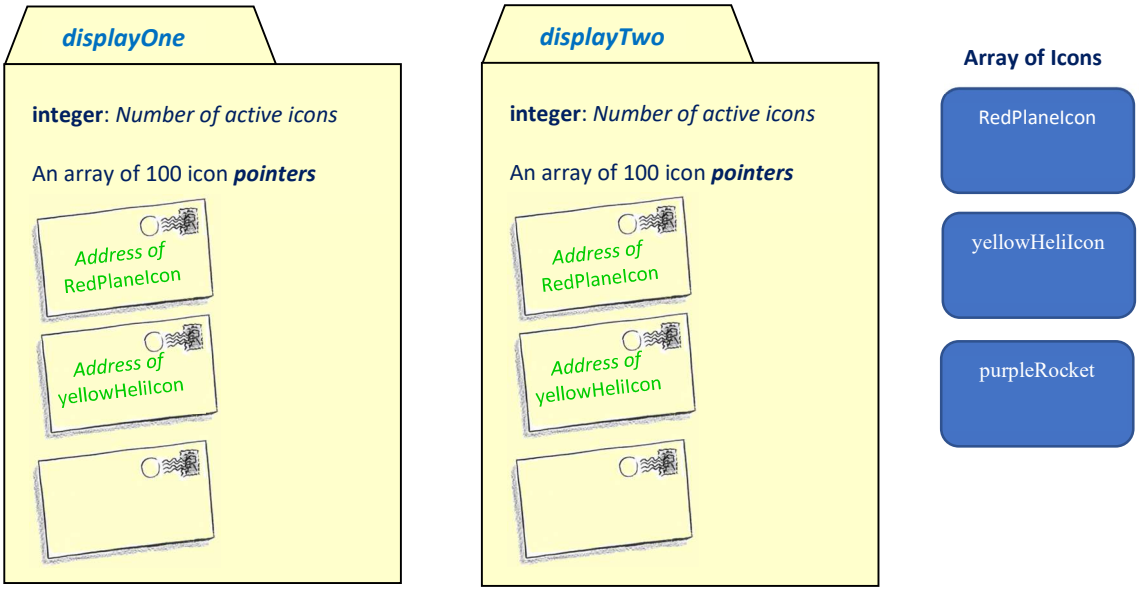
Reflection Points:



Major suggestion – think and design as a human being, don't fixate on *how to code it*!

radarDisplays must know which *icons* they are to display. Does that mean they have to contain *icons* embedded within them? Maybe they could just *point to* the *icons* they are to display? Does this matter here? Yes, the two *radarDisplays* shown in the figure are required to "use" the *same* instance of the red plane icon rather each having its own duplicate. The reasoning behind this is that if we were to *move* the single icon we want both displays to respond consistently.

Personally, I draw a lot of sketches to develop my design ideas. How about?



I might suggest a skeleton object structure and a main code as

```
class displayObject {

public:
        int noActiveIcons;

        icon* iconsToDisplay[100];
};

initialiseRedPlaneIcon(icon& anIcon);          // These functions exist and will be defined later

initialiseYellowHeliIcon (icon& anIcon);

int main() {

        icon allTheIcons[500];

        initialiseRedPlaneIcon(allTheIcons[0]);

        initialiseYellowHeliIcon(allTheIcons[1]);

        displayObject displayOne;

        displayObject displayTwo;


        displayOne.iconsToDisplay[0]=&allTheIcons[0]);

        displayOne.noActiveIcons++;


        displayTwo. iconsToDisplay[0]=&allTheIcons[0]);

        displayTwo.noActiveIcons++;
}
```

Reflection Points:

Do you notice in the design phase how we are prepared to introduce the declaration and interface for a pair of functions without worrying about their detail? The key objective is to get the *big picture* sorted first.

Reflect on the line

```
icon* iconsToDisplay[100];
```

What do we have here, 100 of what?

Similarly, reflect on the line

```
icon allTheIcons[500];
```

What do we have here, 500 of what?

Sub Task Specification continued: What you must specifically do here is to add suitable implementations for the two functions that have just been declared and anything else required to make the above code compile and run. Your decision on suitable test values.

We will not pursue this further today as we have yet to cover the material required to do it *correctly.*

Task Review: Observe how we approached the design and implementation of what sounds quite like a complex object, *radarDisplays*, from its basic constituent parts. We very much undertook this in terms of human being understanding. *radarDipslays* refer to *icons* which themselves are built from collections of basic *pixel* objects.