LAB 05 - Debugging

Hayden Whiteford
104001272
COS30031

Wednesday 9th August

Release Notes:

Answers to each Question

- **Q.1 [line 55] What is the difference between a struct and a class?**
  While both can contain both methods and variables, structs have public access by default.
  Structs also only allow public inheritance, meaning that if you use a struct as a base class for something else, those inherited members will also have public access.
- **Q.2 [line 63] What are function declarations?**

By declaring our functions before we define them we can inform the compiler of the name, return type, and parameters allowing us to call the function before it's defined. This works well for using header files for classes, organisation and readability, and perhaps situations where functions call each other.

- **Q.3 [line 67] Why are variable names not needed here?**

Since this is a function declaration and not function definition, the compiler only needs to know the parameter types at this stage. Later in line 346 we find the real definition that includes names.

- **Q.4 [line 75] Does your IDE know if this method is used?**

In Xcode, if there is code that is never run, the relevant line/s are highlighted yellow and specify such. There is no warning on line 75, so we can assume that it is run. Searching with ctrl + F confirms that it gets called in line 212.

- **Q.5 [line 86] un-initialised values ... what this show and why?**

Taking a look back in the particle struct we can see that there is no constructor that takes parameters, meaning no need to specify this when we declare a particle. Instead this gets done later on in line 92. Anytime we try to show the contents of the particle before we define them will show nothing.

- **Q.6 [line 95] Did this work as expected?**

Yes! The console verifies this with the following:
Q.6: a with assigned values 0,10,20 ? ... Particle: (age=0), (x,y)=(10,20)

- **Q.7 [line 97] Initialisation list - do you know what are they?**

Previously no! But now I understand them to be a more efficient and concise way of initialising variables instead of initialising them inside the constructor's body.

- **Q.8 [line 113] Should show age=1, x=1, y=2. Does it?**

No. P1 was initialised using the getparticlewith() function which initialises the particle variables with whatever parameters were given, in this case 1,2,3. If we wanted x and y as 1 and 2 we would need to call this function with (1,1,2).

- **Q.9 [line 117] Something odd here. What and why?**

-1 is being treated as an unsigned int when being passed into the getparticlewith() function. This is because inside the struct for particles, age is type unsigned int. -1 essentially gets treated as the maximum 32 bit value plus one, 4294967295

- **Q.10 [line 128] showParticle(p1) doesn't show 5,6,7 ... Why?**

When Particle p gets passed into the function, it only gets passed by value, meaning that a copy of the particle is made, and not the original. To fix this you would need to pass the object in by reference, so Particle& p.

- **Q.11 [line 153] So what does -> mean (in words)?**

-> is used to access members of an object using its pointer.

- **Q.12 [line 154] Do we need to put ( ) around *p1_ptr?**

Yes, without using -> , we need to first dereference the pointer. Since Age is a member of the object and not the pointer, we will need to put brackets over this operation.

- **Q.13 [line 160] What is the dereferenced pointer (from the example above)?**

Particle p1 (5,5,5).

- **Q.14 [line 165] Is p1 stored on the heap or stack?**

Since it is a local variable to this block of code, it is only in the stack

- **Q.15 [line 166] What is p1_ptr pointing to now? (Has it changed?)**

Its still pointing to p1. Its just that the values in memory have changed to (7,7,7).

- **Q.16 [line 172] Is the current value of p1_ptr good or bad? Explain**

Good! It still returns the current p1 even after being changed.

- **Q.17 [line 175] Is p1 still available? Explain.**

Yes. Since p1 is a local variable p1 is still available to use for something else outside of the block.

- **Q.18 [line 180] <deleted - ignore> :)**
- **Q.19 [line 189] Uncomment the next code line - will it compile?**

Yes, still compiles, but no 4th index shows.

- **Q.20 [line 192] Does your IDE tell you of any issues? If so, how?**

Yes, Xcode says that the index is past the end of the array, using a yellow highlight.

- **Q.21 [line 200] MAGIC NUMBER?! What is it? Is it bad? Explain!**

Show particle array simply prints out a given number of objects in an array, in this case we specified 3 (only 3 in the array).

- **Q.22 [line 207] Explain in your own words how the array size is calculated.**

It counts the number of integers used in the array, and returns this in bytes, as an integer is 4 bytes, and there are 3 numbers given each for 3 indexes, the size should be 3 x 3 x 4 = 36. The following line takes this number and divides it by the length of the first index, being 12, so that we're no longer talking in terms of bytes, and now just indexes in the array.

- **Q.23 [line 375] What is the difference between this function signature and**

The first showparticlearray() uses a pointer to the first element in the array to gain access to the array elements. In this version we now pass in the full array of particles.

- **Q.24 [line 380] Uncomment the following. It gives different values to those we saw before**

It won't work as a way to determine the array size, as arr is a pointer to the array and no the actual value itself. Xcode says as much by giving yellow warnings on these lines.

- **Q.25 [line 219] Change the size argument to 10 (or similar). What happens?**

Once the loop inside showparticlearray reached the end of the array, its started looking at adjacent memory values, which is where it got the other values from.

- **Q.26 [line 237] What is "hex" and what does it do? (url in your notes)**

It changes the base of the number reprisentation to hexadecimal.

- **Q.27 [line 242] What is new and what did it do?**

It allocates memory for the object type, in this case particle, in the heap and returns a pointer to the object.

- **Q.28 [line 252] What is delete and what did it do?**

It released the allocated memory in the heap for the object.

- **Q.29 [line 256] What happens when we try this? Explain.**

Its undefined behaviour. That memory has been released, so accessing it leads to random memory addresses.

- **Q.30 [line 265] So, what is the difference between NULL and nullptr and 0?**

NULL is an older way of representing a null pointer constant. Nullptr is a a more modern way of representing this and addresses a lot of the ambiguities that NULL has ie. nullptr doesn't implicitly convert to integers. 0 is not strictly a null pointer but is equivalent to NULL.

- **Q.31 [line 267] What happens if you try this? (A zero address now, so ...)**

"Thread 1: EXC_BAD_ACCESS (code=1, address=0x0)" is what gets returned by the IDE.

- **Q.32 [line 302] Are default pointer values in an array safe? Explain.**

Probably not, as accessing them can lead to undefined behaviour. Better practice would be to use nuullptr.

- **Q.33 [line 317] We should always have "delete" to match each "new".**

This is best practice for memory management. When new memory is allocated, it is important to release that memory when no longer in use to prevent memory leaks.

- **Q.34 [line 325] Should we set pointers to nullptr? Why?**

By doing this we are controlling what happens in the case that these pointers are accessed before pointing to a proper value. Without nullptr, our code enters the unknown, and lead to undefined behaviour.

- **Q.35 [line 330] How do you create an array with new and set the size?**

int* array = new int[size];