Task 05 – Debugging

Ryan Chessum – 102564760

Q.1 [line 55] What is the difference between a struct and a class?

In C++ the difference between a struct and a class is the default accessibility of variables and methods. If not specified a class has its member variables and functions set to private while a struct has it's variable and functions set to public. Typically structs are to be used for storing and organising data only while classes are typically used for creating objects with methods and functionality as well as storing data.

Q.2 [line 63] What are function declarations?

Function declarations are necessary in C++ as they essentially tell the compiler about a function and how to call it. The code executed by calling the function can be defined separate of the declaration. A function can be defined with the declaration, but the declaration must be included in every cpp file the function is called in (this is why header files exist). So it's better to have the definition and declaration separate so that the compiler isn't reading the definition for every file that function in used in.

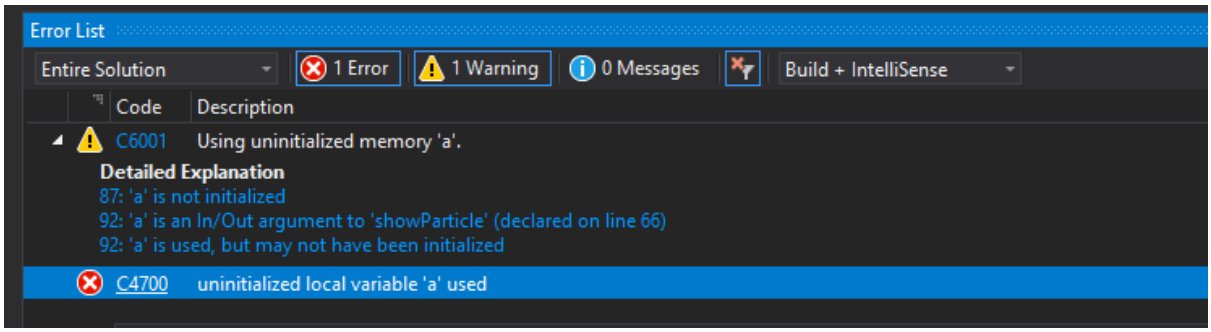
Q.3 [line 67] Why are variable names not needed here?

Parameter variable names are not necessary in function declarations, only in the definition where they need to be used. However, including the variable names can be helpful for programmers to remember/understand what they are used for.

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

No, it does not. It will tell me if there is no definition for the declaration with a warning and green squiggly line under the function, but it does not say whether or not the function is called.

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



When the variable is uninitialized, the program wont compile. In C++ uninitialized variables are not assigned a default value automatically. So, if we tried to use we would get whatever is left there in the assigned memory.

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

Yes, each value is assigned correctly.

```
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?

Yes, initialisation lists are a more efficient way of initialising object variables. They assign each value in order for us. They can be used instead of initialising separate variables just to pass in first.

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

Not Sure if the question here has a typo or if the code is done incorrectly. The question printed to screen also asks for different values. Going by the commented question, it does not, it shows age = 1, x = 2, y = 3 because when you use get particle you are passing 1, 2 and 3. Going by the one printed to screen, yes it does.

```
Q.8: p1 with 1,2,3 ? ... Particle: (age=1), (x,y)=(2,3)
```

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

The age of the particle is being shown as 4294967295 instead of -1 or 1. This is because the datatype of age for the particle is an unsigned int. Setting the value of an unsigned int to a negative number will not set it to be the positive of that value but instead to an extremely high number. This is due to the way both datatypes are stored internally.

```
Q.9: p1 with -1,2,3 ? ... Particle: (age=4294967295), (x,y)=(2,3)
```

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

It still shows 1, 1, 1 because the SetParticleWith function uses a regular parameter for the particle rather than a constant reference parameter. So what happens is a copy of p1 is used and a new particle is returned but not used. To make it set the particle with the new values you could either write p1 = SetParticleWith(5,6,7); or change the parameter of the function to be a constant reference using an & symbol after the data type.

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

-> is used to access a member function or variable of an object through a pointer.

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

Yes and no. You do need to if you want to use . to access the member variables as a pointer is a memory address and does not have the member you would be trying to access. If you don't want to use brackets you can but you would have to use -> instead of . to access the member you want.

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

A dereferenced pointer is used to get the data stored where the pointer is pointing. It can also be used to manipulate that data. P1_ptr is pointing to p1 so dereferencing it gives us p1, which has the values of (5,5,5).

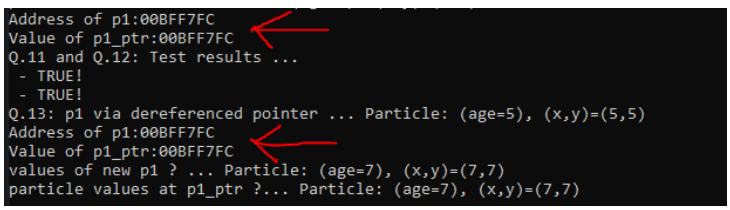
```
Q.13: p1 via dereferenced pointer ... Particle: (age=5), (x,y)=(5,5)
```

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

P1 is a local variable declared without the new keyword so it would be stored on the stack.

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

p1_ptr is still pointing at p1, only the data of p1 has changed, not the address the data was stored in.



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

Good. The pointer still points to the variable it was originally set to, so we don't need to set it to point at p1 again.

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

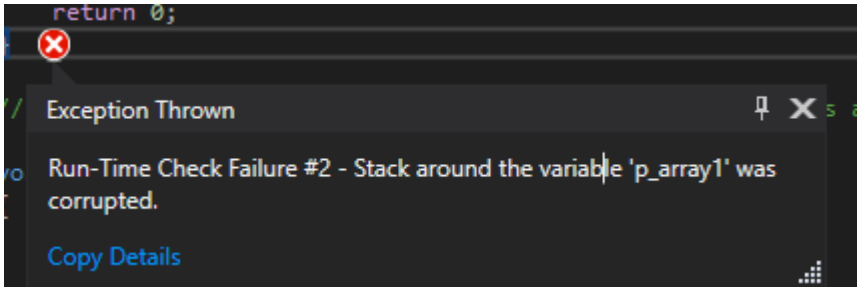
No, P1 was stored on the stack. So, after the scope it was declared in was over, it was deallocated.

Q.18 [line 180] <deleted - ignore> :)

Ok.

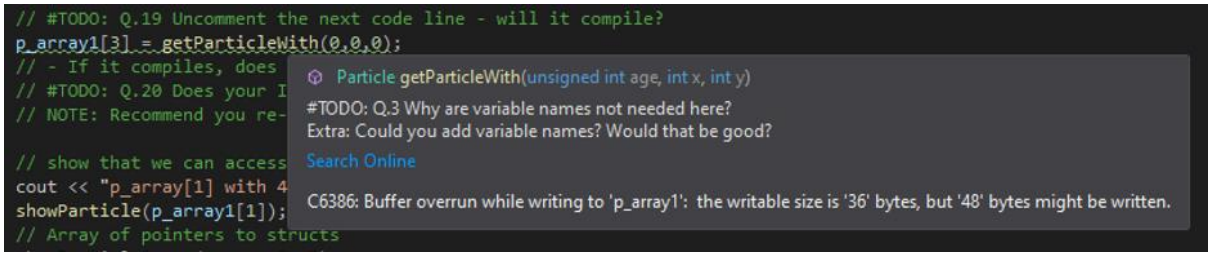
Q.19 [line 189] Uncomment the next code line - will it compile?

It does compile but when the program runs it throws an exception.



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

Yes, it has a green squiggly line underneath the line. Holding the mouse over it says this. It seems like there is not enough memory allocated to the array so when you try to add a 4th value to the array you get an error.



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

A magic number is a hard coded raw number value. It's better to use variables as it helps improve the readability of the code (what you're using the numbers for, etc.), the numbers can be reused more effectively and is generally easier to work with.

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

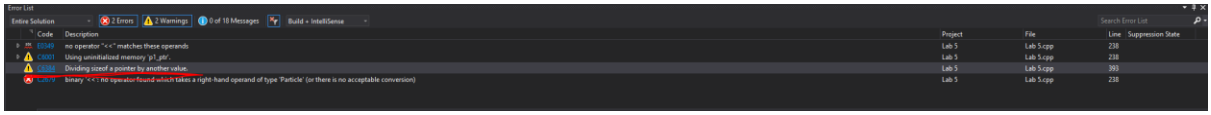
The array size is calculated by getting the size of the array (in bytes) using the sizeof function and then dividing that by the size of a single element in the array. This will give us the number of elements in the array.

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

This function signature takes a Particle array as a parameter while the other one takes a pointer to a Particle as a parameter.

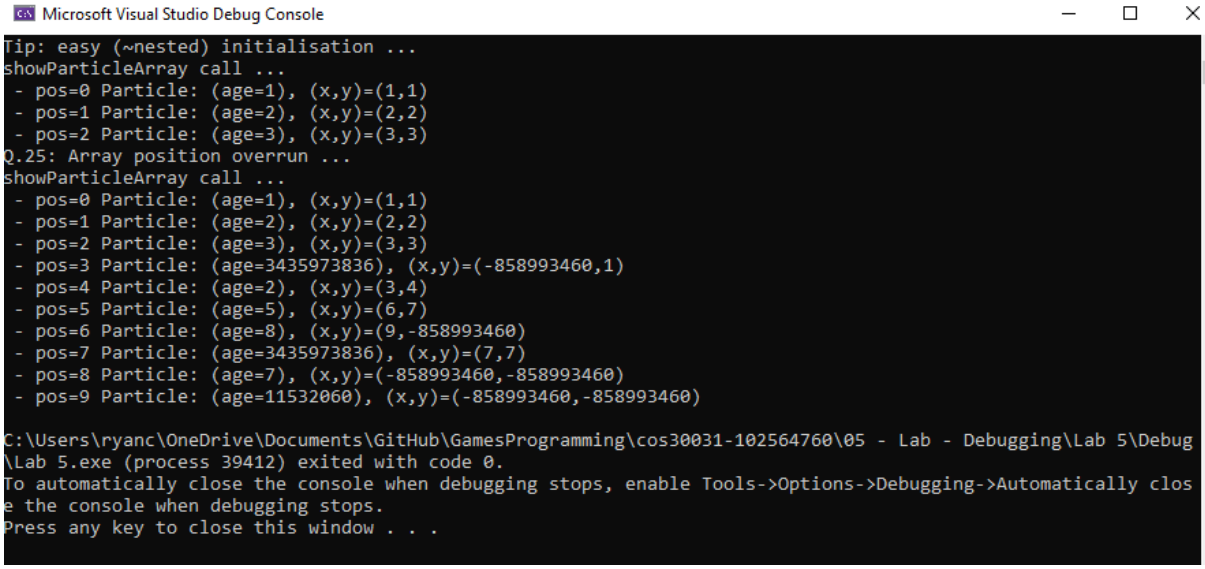
Q.24 [line 380] Uncomment the following. It gives different values to those we saw before

C++ cannot copy arrays so when you pass it into the function it gets passed as a pointer. So, you are just diving the size of a pointer by the size of a value in the array.



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

My guess would be that the function is looking at data in different memory addresses assuming that the data there is a part of the array.



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

https://www.cplusplus.com/reference/ios/hex/

In short, hex makes it, so integer values are displayed in hexadecimal format.

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

The new operator creates a new variable that is allocated memory on the heap and returns the memory address to that variable. So here we created a new particle stored on the heap and p1_ptr now points to that particle. It also automatically initialises the values as 0 for us as it calls the default constructor.

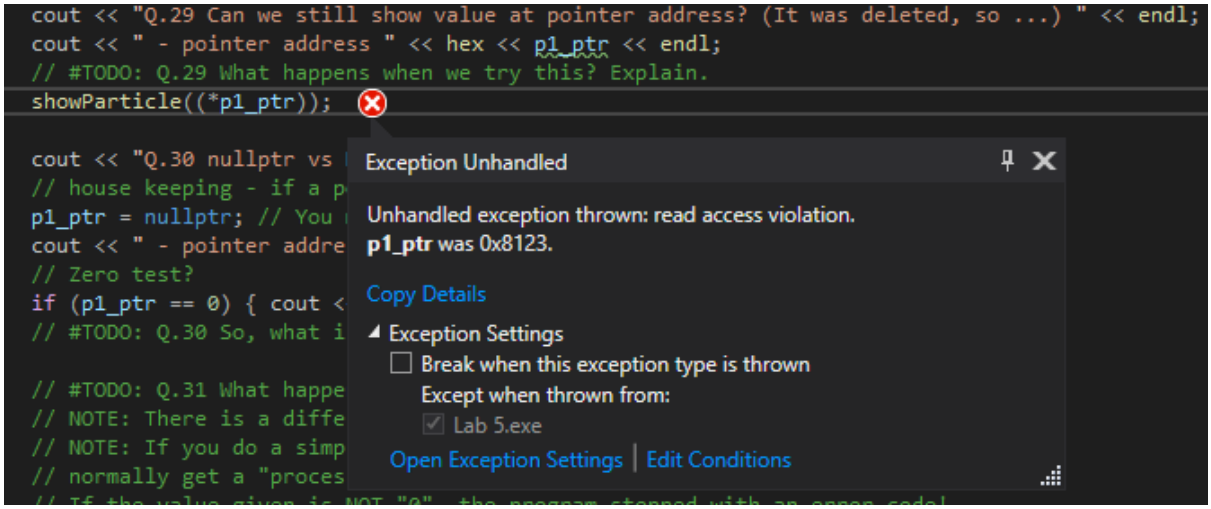
Also note that memory on the heap is not freed automatically after we are done with it unlike stack memory. So we need to make sure we delete it at some point.

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

Delete deallocates memory that we allocated on the stack. So the memory we allocated for p1_ptr has been deallocated.

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

We get a read access violation error. This is known as a memory access violation, it's a form of protection to stop us from messing with data at addresses that have being reserved for important tasks.

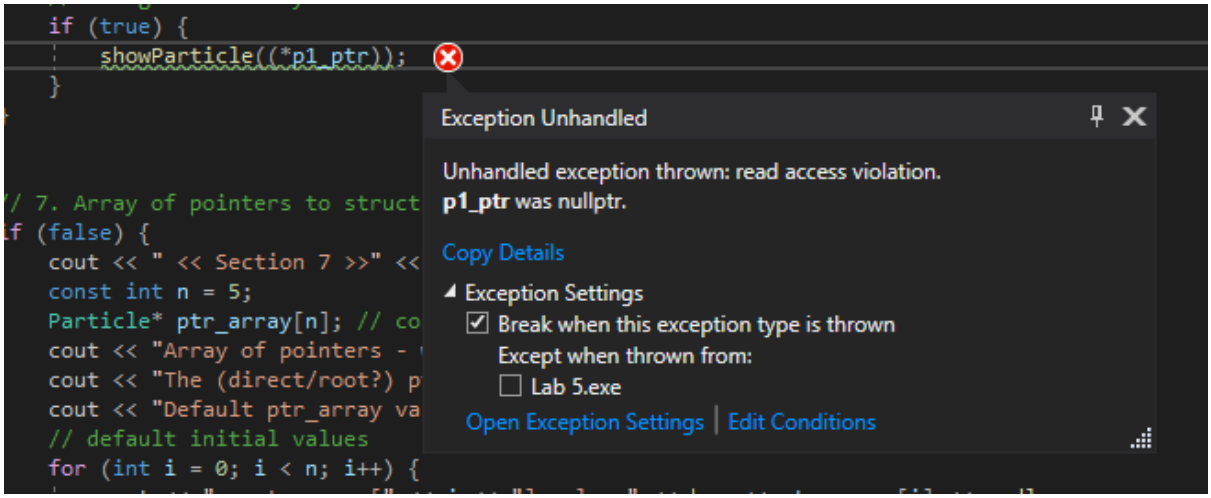


Q.30 [line 265] So, what is the difference between NULL and nullptr and 0?

All can be used but in C++ 11, nullptr is generally the better one to use as it's clearer on what you're trying to do and helps with the readability of the code.

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

It does not let you read it either. 0 address is not meant to store data.



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

No, it's better to set them to be a null pointer when it's not pointing to anything we asked it to in memory. Whatever is stored there is junk and could be anything stored there previously. This makes testing much safer.

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

Yes, otherwise we can cause memory leaks. Memory leaks are where the allocated memory is never released so the memory is unusable.

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

Our pointers are no longer pointing to anything we want them to, so we should change them back to null pointers.

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

```
// Note: if we dynamically created the array (with new), we should clean that up too.
// #TODO: Q.35 How do you create an array with new and set the size?

Particle* my_array;
my_array = new Particle[n];

delete[] my_array;
```