Task 5.P - 100595153

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

The key difference between a struct and class ignoring the naming, is the default accessibility of each member variable and function. Classes are set to private, structs are set to public.

2. [line 63] What are function declarations?

Functions declarations are forward declarations of functions that are yet to be explicitly defined. These are generally used in header files, though as is clearly with the implementation here, can be done within cpp files too. The purpose of a forward declaration is to let the compiler know that the function exists so that declaration issues can be avoided.

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

As these are forward declarations, the actual name of the parameters does not need to be included yet, nothing has referenced the parameters yet. This can be included during the function definition.

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

Yes. Generally unused methods and variables are duller than those that are used. Additionally, the IDE sometimes shows a warning underline.

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

Currently, the variable has not been initialised. It has a type, but currently is null. There is no assigned value.

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

Without fixing the error from the previous question, particle is not output to console and instead, an error is thrown. After fixing the error, the particle is output and runs as expected.

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

I'm not the most familiar with them, but I believe I have used them at least once prior. They are essentially lists of values to be assigned to the class's member variables on initialisation. As noted, having a list of variables is a tad easier than assigning one by one. Although, using the new keyword and initializing via constructor would probably be fine too, from what I understand.

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

No… It should not… I assume this question contains typos.

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

The particle class uses an unsigned int type, therefore it cannot accurately represent a negative integer value. The data type would need to be changed to a signed int to represent -1.

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

The parameters are passed by value, rather than by reference. The particle and integers within the function setParticleWith are clones of the original variables, rather than references to them.

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

The arrow operator is used to point to members of a class or structure in order to access said member. In this case, p1's age is being referenced by the operator.

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

The brackets are there to contain the specific pointer being dereferenced. Removing them causes an error in which the datatype of the age member variable is unable to be determined.

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

This is showing the values in memory from the p1 Particle, declared at the start of this section.

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

p1 is stored on the stack, as it is created locally, within a function. It is then assigned to memory allocated for p1 again, on the stack. Were it to use the keyword new, that would add it to the heap instead, however the class does not currently contain a constructor.

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

The pointer is still pointing to the same memory address. The data stored at that memory address has however, changed.

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

Neutral? It depends on its purpose. Currently, it is pointing to p1 and when used to output, shows the same results as p1. If it wasn't supposed to be doing that, it would be bad.

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

Nope, we've gone outside the scope of it (by exiting the if statement) and therefore it has now been removed by auto garbage collection.

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

Sure.

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

It does not. It is trying to access an element that does not exist and goes outside of the bounds of memory allocated for this variable.

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

It underlines it with a green squiggle, as well as not compiling.

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

It's definitely not ideal. It's there so that the loop within showParticleArray knows when to stop looping. It's not great practice, as there are other ways to do that. Different

collections with an accessible size function or member variable could be used, etc. This way means there's an unlabelled number lying around and will need to be changed if the array size is ever changed.

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

It's the byte size of the array, divided by the byte size of the first element. You are then left with the number of elements in the array.

23. [line 375] What is the difference between this function signature and the function signature for showParticleArray?

While they are semantically different, they still essentially produce the same result. The first one is a pointer to an object, the second one is a pointer to an array.

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

As arr is a pointer, the function is getting the size of the pointer, rather than the size of the dereferenced underlying object which would actually give the correct result.

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

Memory leak. We're accessing things we really shouldn't be.

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

Hex is an output modifier. It sets cout to output as hexadecimal.

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

New allocated memory for the particle on the heap. This will now also require manual garbage collection.

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

This is garbage collection. It frees up the memory set aside previously and the pointer now points to who knows what.

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

An exception is thrown. We're committing a read access violation. Basically we're trying to access nothing, but since we haven't set it as a nullptr yet, it could be pointing to somewhere important that we shouldn't be touching.

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

All of them are able to represent a null pointer, though nullptr is a safer implementation as it has a specific type.

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

Read access violation. Same as before.

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

Nope, there are memory leaks happening here. Setting them to nullptr is again, much safer. That way, we know it points to nothing.

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

Yes.

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

To avoid errors and memory leak. Setting a pointer to nullptr after we're done with it is good practice.

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

A declaration would look like this:

type *arr = new type[size];

This would essentially end up as a pointer to an array.