**Lab:** Task 6

**Title:** Debugging

**Author:** Sam Huffer, 101633177

* Misc:
  + *Q: What is the difference between a struct and a class?*

A: a struct’s member fields and methods are by default public, while a class’s member fields and methods are private by default.

* + A: C++ prototype methods can be declared without specifying variable names for parameters; they need only specify the data types.
* Section 1:
  + *Q: un-initialise values? What did it show and why?*

A: The compiler threw an error message “uninitialized local variable “a” used”. Presumably C++ can’t process variables whose values are uninitialized/unknown.

Creating a new Particle and assigning it to a resulted in showParticle() outputting values of 0 for the particle’s age, x and y fields, 0 being the default value of unassigned int fields.

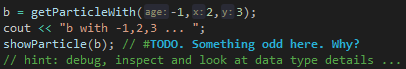
* + *Q: worked as expected?*

A: Yes, the values assigned to Particle a’s fields after its instantiation and original printing were printed as expected.

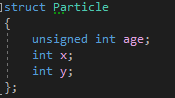
* + *Q:*

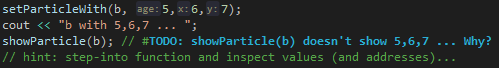
A: I knew you could initialise arrays like this, but I was not aware that structs could be initialised like this. Looking it up, it seems to be an initialisation method for structs and classes that have no private fields nor a user-declared or inherited constructor method.

* Section 2:
  + *Q: showParticle(b) should show age=1, x=1, y=2. Does it?*

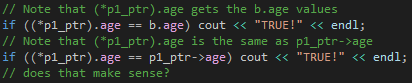
A: typo, as b is initialised with age=1, x=2, y=3. In any case, showParticle(b) does output the correct values.

* + *Q:*

A: after the reassignment, showParticle(b) outputs age=4294967295, not -1. This is because age is defined as an unsigned int, i.e. the first bit does not indicate whether it is positive or negative, and therefore it is assumed to be positive. When -1 was assigned, presumably the program took the binary for signed int -1 and assigned it to age, and when age was read from memory, it was treated as an unsigned int rather than a signed int and the bit at the front of the binary designation was read not as “this number is negative” but “this number is big”.

* Section 3:
  + *Q:*

In the definition of setParticleWith(), the Particle parameter is a pass-by-value parameter; the original Particle struct passed to the procedure is copied, and that copy is given the age, x and y values passed. The values are never assigned to the original, so when the original is passed to showParticle() again, its values are the same as the last time it was shown. For setParticleWith() to affect the original, it needs to take a pointer to the Particle the values are supposed to be assigned to, rather than just taking a copy of the Particle that will vanish once the procedure finishes.

* Section 4:
  + *Q:*

A: yes that makes sense, but the ugly a->b formatting is gonna annoy me; a.b is so much cleaner and quicker to type.

* + *Q: what is a dereferenced pointer (from the example)? Example: “showParticle((\*p1\_ptr));”*

A: I had to look up this piece of terminology, but to dereference a pointer is the technical jargon for saying “okay, at this spot where I’m accessing pointer X, I don’t want to access the memory address of the thing it’s pointing to; I want to access the value stored at that memory address”.

* + *Q: update [Particle] b, pointer [\*p1\_ptr] still what we expect (or something new)? If they are the same, what does this imply? If they are different, what does this imply?*

A: okay, so b was reassigned to using getParticleWith(), which creates a new Particle and assigns it to b, rather than updating the values of b’s fields. When p1\_ptr was accessed to read the values of the Particle it was pointing to, they were the values of the Particle created in getParticleWith(). This suggests that once Particle b is declared, it stays at the same memory location, even if the value stored there is overwritten by an entirely new object. (This would also suggest that when a new Particle is created and then assigned to b, that new particle is moved from wherever it was being stored in memory at its creation to the memory space originally designated for b, rather than the variable and the value being matched up but stored wherever in memory.) Consequently, any new Particle objects assigned to b will be accessible from p1\_ptr as b’s memory address remains that pointed to by p1\_ptr.

* Section 5:
  + *Q:*

A:

* Section 6:
  + *Q:*

A:

* Section 7:
  + *Q:*

A: