Spike Summary Report 20/08/19

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.

```
// #TODO - Why are variable names not needed?
Particle getParticleWith(int, int, int);
```

- 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:*

```
// Note: uses of an initialisation list. (quicker then setting each as above!)
// #TODO - Do you know about them? If not, find out and make a note in your report.
Particle b = {.age:0, .x:0, .y:0};
```

    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.

```
b = getParticleWith(age:1, x:2, y:3);
```

  - *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".

```
b = getParticleWith(age:-1, x:2, y:3);
cout << "b with -1,2,3 ... ";
showParticle(b); // #TODO. Something odd here. Why?
// hint: debug, inspect and look at data type details ...
```

```
struct Particle
{
    unsigned int age;
    int x;
    int y;
};
```

- 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

```
setParticleWith(b, age:5, x:6, y:7);
cout << "b with 5,6,7 ... ";
showParticle(b); // #TODO: showParticle(b) doesn't show 5,6,7 ... Why?
// hint: step-into function and inspect values (and addresses)...
```

```
void setParticleWith(Particle p, int age, int x, int y){ ... }
```

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:*

```
// Note that (*p1_ptr).age gets the b.age values
if ((*p1_ptr).age == b.age) cout << "TRUE!" << endl;
// Note that (*p1_ptr).age is the same as p1_ptr->age
if ((*p1_ptr).age == p1_ptr->age) cout << "TRUE!" << endl;
// does that make sense?
```

    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:

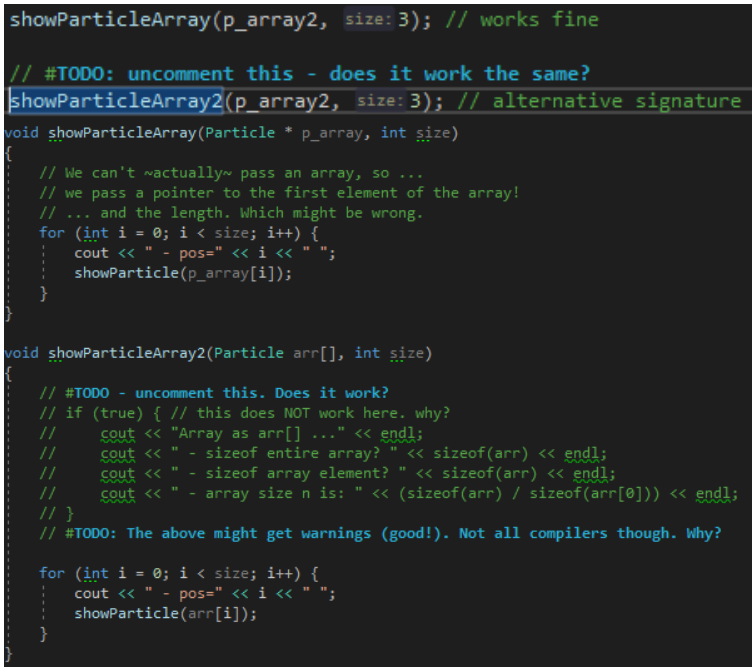
```
// Array of pointers to structs
showParticleArray(p_array1, size: 3); // #TODO: MAGIC NUMBER BAD!! What is it and why bad?
```

  - *Q:*
    A: the number passed to the size parameter is the programmer manually keeping track of the number of elements in the array p_array1, since C++'s basic arrays can't do it themselves. This is less than ideal, as one could assign a new array to p_array1 and forget to update a variable storing its size, and suddenly that value is too small, meaning the later elements in the array are forgotten about when doing for loops, or it's too big and the program reads memory that's actually allocated for something else and gets weird results.
  - *Q:*

```
// Can we work out the length? Yes, but ...
cout << "Array length?" << endl;
cout << " - sizeof entire array? " << sizeof(p_array1) << endl;
cout << " - sizeof array element? " << sizeof(p_array1[0]) << endl;
cout << " - array size n is: " << (sizeof(p_array1) / sizeof(p_array1[0])) << endl;
// #TODO: explain in your own words how the array size is worked out.
```

    A: sizeof() returns the amount of memory taken up by whatever is passed to it (in bytes?). If passed an array, it'll return how much memory is allocated for the whole array. If passed a single element, one can determine how much space a single element takes up. As the space occupied by elements of the same type is the same, when given the space of a single element and the array as a whole, one need only divide the latter by the former to deduce how many elements the array holds.

- *Q:*
  A: As originally coded, it provides the same output. The Particle parameters both effectively serve as a pointer to the first element of the array, and so give the same result when further elements are accessed. However, the first provides a pointer to the whole memory block for the array while the second points only to the memory block for the first array element?



```
showParticleArray(p_array2, size: 3); // works fine

// #TODO: uncomment this - does it work the same?
showParticleArray2(p_array2, size: 3); // alternative signature
void showParticleArray(Particle * p_array, int size)
{
    // We can't ~actually~ pass an array, so ...
    // we pass a pointer to the first element of the array!
    // ... and the length. Which might be wrong.
    for (int i = 0; i < size; i++) {
        cout << " - pos=" << i << " ";
        showParticle(p_array[i]);
    }
}

void showParticleArray2(Particle arr[], int size)
{
    // #TODO - uncomment this. Does it work?
    // if (true) { // this does NOT work here. why?
    //     cout << "Array as arr[] ..." << endl;
    //     cout << " - sizeof entire array? " << sizeof(arr) << endl;
    //     cout << " - sizeof array element? " << sizeof(arr) << endl;
    //     cout << " - array size n is: " << (sizeof(arr) / sizeof(arr[0])) << endl;
    // }
    // #TODO: The above might get warnings (good!). Not all compilers though. Why?

    for (int i = 0; i < size; i++) {
        cout << " - pos=" << i << " ";
        showParticle(arr[i]);
    }
}
```

  Consequently, the commented-out code in showParticleArray2() does not work as intended because a) arr as a whole is passed to sizeof() to ascertain both the size of the array and of a single element, returning identical values in the second and third cout statements, and b) when arr[0] is passed instead, because arr is just a reference to the memory block of the first array element, rather than to the array as a whole, and so – being an int – should have no valid first element (conceptually, at least), resulting in an inaccurate value for its size.

  Visual Studio did not output any warnings for the uncommented code; it must have thought that the calls made were technically correct or possible, even the call for sizeof(arr[0]).

- *Q:*

```
// read (access) more then we should ... size 4 (but it's only 3)
cout << "array position overrun ... " << endl;
showParticleArray(p_array2, size: 4); // #TODO: HMM! What is the issue here??
// #TODO: would you expect that same output all the time?
```

  A: the issue here is that the array was originally only defined to have 3 elements, but showParticleArray() is being told to show 4 elements. So it reads what might have been the next array element in its block of memory if it had 4 elements rather than 3, interprets whatever is stored there as a value of the array's data type, and outputs completely junk data. You wouldn't expect the same output all the time, as the computer is treating it as memory for something else – and so updates it as that other thing is updated – rather than as memory for that array and that shouldn't be allocated to other variables.

- Section 6:
  - *Q: in "cout << "pointer address " << hex << &c << endl;", what is hex and what does it do? Ref/url?*
    A: From https://en.cppreference.com/w/cpp/io/manip/hex: hex tells the compiler to output integers following it in a cout statement as hexadecimal values rather than decimal values.

    Note: I tested removing it for the pointer memory address output, and it still gave the hexadecimal memory address. When tested on raw integers (alongside dec, to render output as decimal, and oct, to render output as octal (base 8)), however, the statements worked just fine. That suggests the pointer address either can only be output as hexadecimal, or isn't output as an integer to begin with.
  - *Q: Now let's create a Particle that we only access via a pointer . . . what is new and what did it do?*

A: When calling “variable = new ClassName()”, new returns a pointer to the object created, rather than returning the object itself.

- *Q: in “delete p2_ptr;”, what is delete and what did it do? . . . What happens if you try [then accessing what was stored at the memory address p2_ptr pointed to]? Explain*
  A: the statement “delete” deletes the value stored at the memory address pointed to by a pointer. (https://www.geeksforgeeks.org/delete-in-c/: destroys arrays OR non-array (pointer) objects created by the new expression.) Trying to then access it throws an error, as the computer thinks there’s nothing stored there that a variable should know about, and thinks you’re trying to read data that you’re not supposed to (which is ironic, considering how arrays work).
- *Q: what is the difference between NULL and nullptr and 0?*
  A: From https://www.quora.com/Whats-the-difference-between-NULL-and-nullptr-in-C++: NULL is an integer that can be assigned to a pointer because of an implicit conversion; nullptr is “a keyword representing a value of self-defined type, that can convert to a pointer, but not into integers”. So if you tried to assign NULL to an int or a pointer, it would work fine, but nullptr would only work with pointers, and not ints.
  0, I assume, also serves as an improvised “null” value for pointers. For integers, it’d just be a value of 0, which is different from null in that context.
- *Q:*
  A: no, it throws a read-access violation, as it thinks you’re trying to read memory where nothing is stored / that the pointer cannot access.

```
// TODO: what happens if you try this? (Zero address now, so ...)
if (true) {
    showParticle((*p2_ptr));
}
```

- Section 7:
  - *Q:*
    A: As written, this outputs an error, complaining that n isn’t a constant. Hardcoding the value of 5, however, results in it outputting an address of CCCCCCCC for ptr_array[0], which differs from the nullptr address of 00000000 observed in Section 6, so it’s not technically a nullptr? However, it looks like a special memory address reserved for particular scenarios, and google isn’t turning up any results for what that would be, so I don’t know that I’d trust it. I’d assume the values of each array element are undefined.

```
int n = 5;
Particle *ptr_array[n]; // contains pointers to nowhere so far!
// #TODO: what is the pointer value? (Is it empty?/nullptr?) Can you trust it?
```

  - *Q:*
    A: including the brackets and the asterisk dereferences the pointer, telling the compiler that you don’t want the memory address pointed to, but the value stored at that memory address.

```
cout << "Show each particle pointed to in the pointer array ..." << endl;
for (int i = 0; i < n; i++) {
    showParticle((*ptr_array[i]));
    // #TODO: why is (*ptr_array[i]) needed?
}
```

  - *Q:*
    A: This seems like a good habit to have as it ensures that all array elements are destroyed and the memory they were taking up is freed for use and not later readable by something else that shouldn’t be reading it, as it is set to nullptr.

```
// cleanup! Can you see what happens if you DON'T do this?
if (true) {
    for (int i = 0; i < n; i++) {
        delete ptr_array[i];
        ptr_array[i] = nullptr; // Not always needed but good habit! #TODO: agree? why?
    }
}
// Note: if we dynamically created the array (with new), we should clean that up too.
```