Title: Simple debugging

Author: Parth Madhani, 101901858

Tasks undertaken:

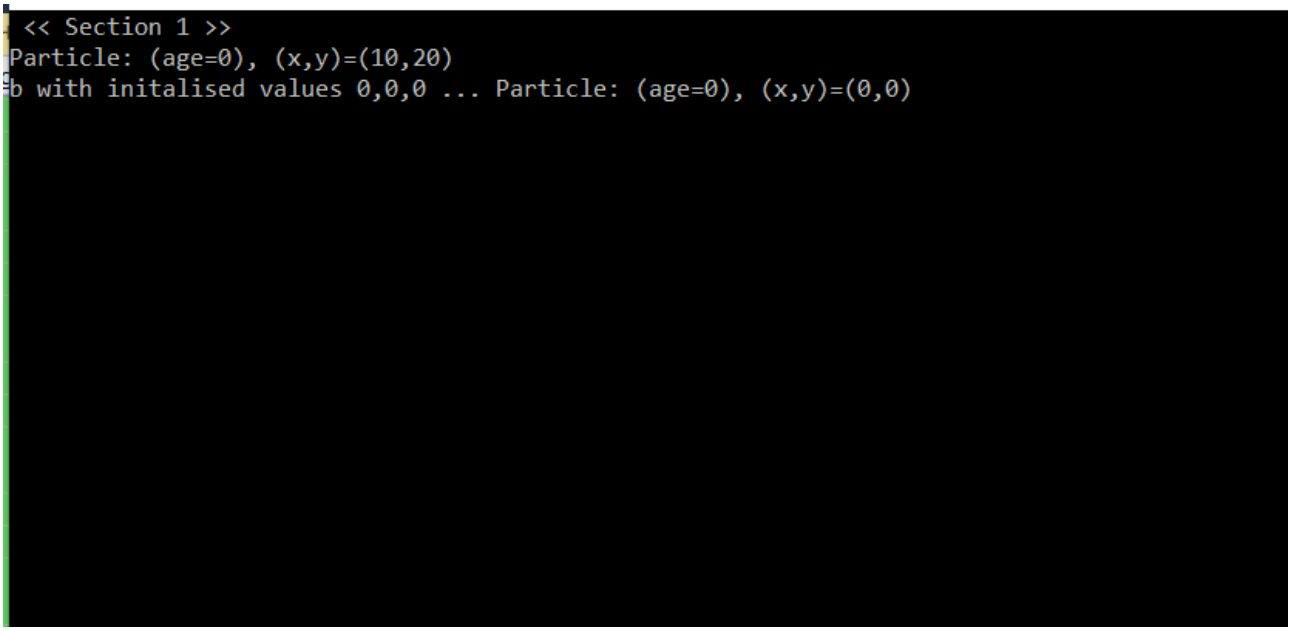
- Applying the skills learnt from previous spike, to fix/analyse the code using the debugger.

What we found out:

I found out how to fix issues in the given source code by using the advanced functionality of the debugger to step through the source code and inspect the data and flow of the program.

Notes :

- First thing found out the difference between struct and class which in c++ are almost similar with main difference being that members of a class are private by default while members of a struct are public by default.
- Variable names are not needed when declaring functions.
- Next , it was about uninitialized list which would give you an error in studio and won't run the program. Another thing I found was about methods to initialize.



- Next coming to section 2 when we run it the details for the first time shows correctly while for the second set of values we pass it shows different because we pass -1 as age given which the data type we use for age is unsigned int which signifies that no negative values are allowed.

```
 << Section 2 >>
b with 1,2,3 ... Particle: (age=1), (x,y)=(2,3)
b with -1,2,3 ... Particle: (age=4294967295), (x,y)=(2,3)
```

- Next coming to section 3 we learn about pass by value and pass by reference as when we pass the 'b' to the show particle it would show last value of b which was not 5,6,7 as we set it up before using function due to it being passed by value and not by reference as for the compiler at that address there is still same value as before.

```
 << Section 2 >>
b with 1,2,3 ... Particle: (age=1), (x,y)=(2,3)
b with -1,2,3 ... Particle: (age=4294967295), (x,y)=(2,3)
 << Section 3 >>
b with 5,6,7 ... Particle: (age=4294967295), (x,y)=(2,3)
```

- In the 4th section we learn about pass by reference and usage of pointers. For instance in our case we create a pointer that points to address at variable b making the pointer same as b even if we change values of b the values of pointer changes to as address of b always remains same regardless. Additionally just to ensure that we check the values and usage of -> for pointers. We then pass the deferenced pointer (which basically means to retrieve the value the pointer is pointing at in the same data type as to which it points to) to the function which shows the value of b we just set before. After that to ensure again we change the value of b and check the pointer value now which again is now set to value of b which we just changed to.

```
 << Section 4 >>
new values of b ... Particle: (age=5), (x,y)=(5,5)
TRUE!
TRUE!
b via dereferenced pointer ... Particle: (age=5), (x,y)=(5,5)
new values of b ... Particle: (age=7), (x,y)=(7,7)
show particle pointer (same still?) ... Particle: (age=7), (x,y)=(7,7)
```

- Moving to the 5th section which is mainly about making array of structs. This basically creates array with size of 3 and then we try and access its elements and displaying array. Then we work out the length of array using sizeof function which basically is to first calculate the amount of byte for the whole array then calculating the byte for just a single block of it and then dividing it which would give u the size of array which is 3 in our case. Next it uses different initialization approach for the array. Lastly we overflow the array by passing size 4 instead of 3 which overflows the array which basically means that when we access something that is out of bounds, it won't throw a segmentation fault unless its completely out of your stack memory.

```
 << Section 5 >>
p_array[2] with 4,5,6 ... Particle: (age=4), (x,y)=(5,6)
 - pos=0 Particle: (age=1), (x,y)=(2,3)
 - pos=1 Particle: (age=4), (x,y)=(5,6)
 - pos=2 Particle: (age=7), (x,y)=(8,9)
Array length?
 - sizeof entire array? 36
 - sizeof array element? 12
 - array size n is: 3
easy (~nested) initialiation ...
 - pos=0 Particle: (age=1), (x,y)=(1,1)
 - pos=1 Particle: (age=2), (x,y)=(2,2)
 - pos=2 Particle: (age=3), (x,y)=(3,3)
Array as arr[] ...
 - sizeof entire array? 4
 - sizeof array element? 4
 - array size n is: 0
 - pos=0 Particle: (age=1), (x,y)=(1,1)
 - pos=1 Particle: (age=2), (x,y)=(2,2)
 - pos=2 Particle: (age=3), (x,y)=(3,3)
array position overrun ...
 - pos=0 Particle: (age=1), (x,y)=(1,1)
 - pos=1 Particle: (age=2), (x,y)=(2,2)
 - pos=2 Particle: (age=3), (x,y)=(3,3)
 - pos=3 Particle: (age=3435973836), (x,y)=(-858993460,1)
```

- Moving on to section 6 which was to me the most interesting bit of code where I learnt / revised a lot about pointers. So first we create a pointer that points to nothing and test if we can see any address but as it points to nothing it will give an error. When we use new keyword instead for creating pointer we are making the pointer point to the space that we created in heap using new keyword. When we copy the pointer to our struct it will then point to the address of that struct which is stored in stack memory. Next we create pointer using new keyword and this time when we print it it does show a address as explained above. When we cleanup / delete the pointer we created using new keyword then try to see if it still points to anything it will show nothing as even though when we delete it will still have some address but that address is not pointing to anything. After that we change the pointer to null ptr and as expected it will now point to 0 address.

```
 << Section 6 >>
pointer address 006FF7C0
pointer address 006FF7C0
pointer address 0079E5E0
Particle: (age=0), (x,y)=(0,0)
show via dereferenced pointer ... Particle: (age=0), (x,y)=(0,0)
set a value via pointer
Particle: (age=63), (x,y)=(0,0)
pointer address 00008123
Can we still show value at pointer address? (It was deleted, so ...) pointer address 00000000
pointer == 0
```

- Moving to last section, another interesting one where we create a array of pointers to structs. Here we create an array of pointers of 5 value. When not initialized and printed by default visual studio just shows CCCCCC. After that we use new keyword to assign it some address. At the end we have to cleanup memory we created as its heap memory because otherwise we will get a memory leak, and some allocated memory space will never be returned for other programs to use.