# Task 6 – 102564760

Ryan Chessum

**Q.1 [line 102] There are inside array_demo_1 - answer them there.**

**Q.1.1 [line 164] What do the < and > mean or indicate?**

*<> is a template and is used for initialising std arrays, vectors, etc. They specify what datatype to use. Custom datatypes/classes can also use a template if you want them to be able to use different datatypes with them.*

**Q.1.2 [line 165] Why don't we need to write std:array here? (Is this good?)**

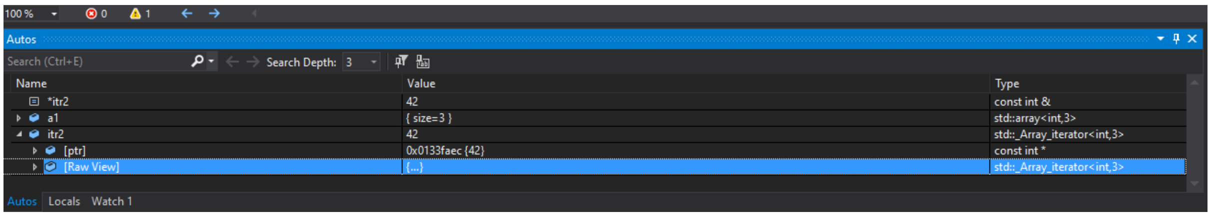
*Because we're using namespace std at the top we can use things from the standard library without writing std:: before using them. This is great as we don't need to write std:: before everything from the standard library so our code looks nicer,  is a bit easier to read and saves us a bit of time.*

**Q.1.3 [line 166] Explain what the int and 3 indicate in this case?**

*In this case int is the data type for the items in the array and 3 is the number of elements in the array.*

**Q.1.4 [line 204] In the code above, what is the type of itr2?**

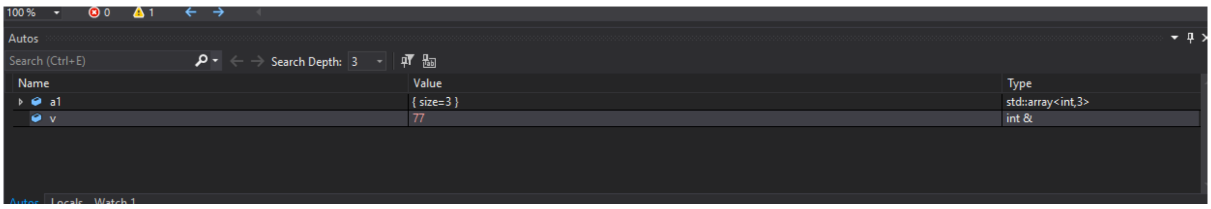
*Auto is a keyword which lets the compiler work out the type for us. If we use a break point we can see that Itr2 is set to an Array iterator.*



```
100 %    ⊗ 0   ⚠ 1
Autos
Search (Ctrl+E)          Search Depth: 3
Name                 Value                 Type
   *itr2             42                    const int &
 ▸ a1                { size=3 }            std::array<int,3>
 ◢ itr2              42                    std::_Array_iterator<int,3>
   ▸ [ptr]           0x0133faec {42}       const int *
   ▸ [Raw View]      {...}                 std::_Array_iterator<int,3>
Autos  Locals  Watch 1
```

**Q.1.5 [line 211] In the code above, what is the type of v?**

*V is a constant reference of an int.*



```
100 %    ⊗ 0   ⚠ 1
Autos
Search (Ctrl+E)          Search Depth: 3
Name                 Value                 Type
 ▸ a1                { size=3 }            std::array<int,3>
   v                 77                    int &
Autos  Locals  Watch 1
```

**Q.1.6 [line 212] In the code above, what does the & mean in (auto &v : a1)**

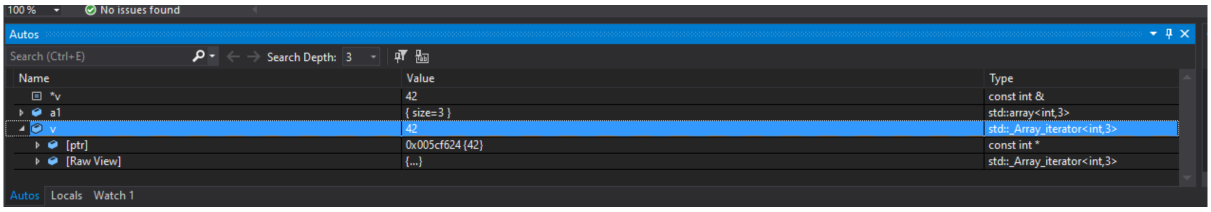
*& is a constant reference to an object or variable. So, if we manipulate it within a function, it wont manipulate a copy, the variable will be changed even outside of the function. It's similar to passing a pointer as essentially what is happening is we are manipulating data at the address of the variable.*

**Q.1.7 [line 220] Try this. Why does a1[3] work but at(3) does not?**

*My IDE does not let me do either but in theory a1[3] would let us look outside the array otherwise because it does not have any bounds checking at() performs bound checking for us.*

**Q.1.8 [line 233] auto is awesome. What is the actual type of v that it works out for us?**

*int array iterator.*



100 % No issues found

Autos

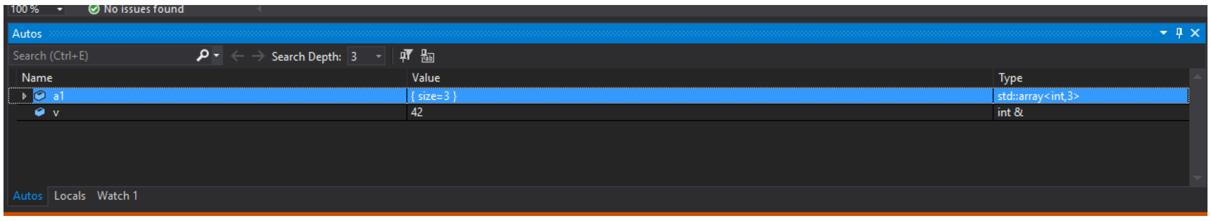
Search (Ctrl+E) Search Depth: 3

| Name | Value | Type |
|---|---|---|
| *v | 42 | const int & |
| a1 | { size=3 } | std::array<int,3> |
| v | 42 | std::_Array_iterator<int,3> |
| [ptr] | 0x005cf624 {42} | const int * |
| [Raw View] | {...} | std::_Array_iterator<int,3> |

Autos Locals Watch 1

### Q.1.9 [line 240] auto is still awesome. What is the actual type of v here?

*Constant reference to an int.*



100 % No issues found

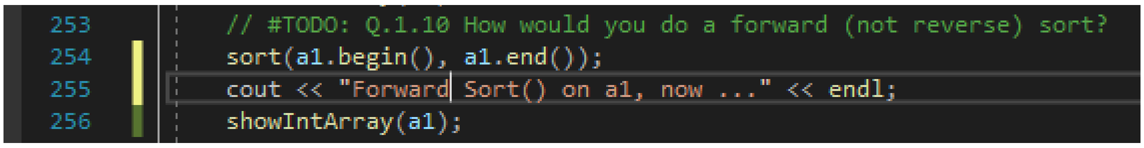
Autos

Search (Ctrl+E) Search Depth: 3

| Name | Value | Type |
|---|---|---|
| a1 | { size=3 } | std::array<int,3> |
| v | 42 | int & |

Autos Locals Watch 1

### Q.1.10 [line 250] How would you do a forward (not reverse) sort?

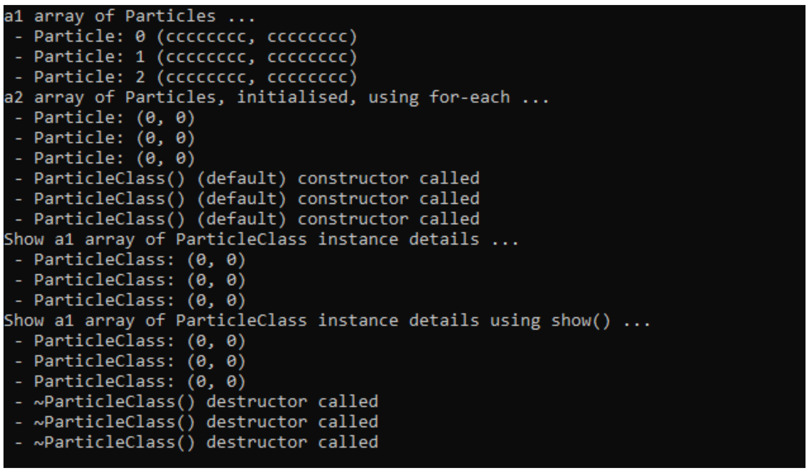


```
// #TODO: Q.1.10 How would you do a forward (not reverse) sort?
sort(a1.begin(), a1.end());
cout << "Forward Sort() on a1, now ..." << endl;
showIntArray(a1);
```

### Q.2 [line 105] In array_demo_2, explain what a4(a1) does

*This calls a copy constructor and makes a copy of a1.*

### Q.3 [line 108] No questions for array_demo_3, it's just a demo of Struct/Class use with array.



```
a1 array of Particles ...
 - Particle: 0 (cccccccc, cccccccc)
 - Particle: 1 (cccccccc, cccccccc)
 - Particle: 2 (cccccccc, cccccccc)
a2 array of Particles, initialised, using for-each ...
 - Particle: (0, 0)
 - Particle: (0, 0)
 - Particle: (0, 0)
 - ParticleClass() (default) constructor called
 - ParticleClass() (default) constructor called
 - ParticleClass() (default) constructor called
Show a1 array of ParticleClass instance details ...
 - ParticleClass: (0, 0)
 - ParticleClass: (0, 0)
 - ParticleClass: (0, 0)
Show a1 array of ParticleClass instance details using show() ...
 - ParticleClass: (0, 0)
 - ParticleClass: (0, 0)
 - ParticleClass: (0, 0)
 - ~ParticleClass() destructor called
 - ~ParticleClass() destructor called
 - ~ParticleClass() destructor called
```

### Q.4 [line 111] How do we (what methods) add and remove items to a stack?

*Push() adds an item to the top of a stack and pop() removes the item at the top of a stack.*

### Q.5 [line 112] A stack has no no [] or at() method - why?

*Stacks aren't meant to be used like arrays and so haven't been designed with their functionality. A stack is designed to be more limited and data stored in it can only be accessed by it's rules. What you put in goes on the top and what you take out is from the top. If you need to use a stack this way you should use a different data structure like an array or vector.*

**Q.6 [line 115] What is the difference between a stack.pop() and a queue.pop() ?**

*Stack.pop() will give you the element that was last inserted into it while queue.pop() will give you the first element inserted into it.*

**Q.7 [line 118] Can we access a list value using and int index? Explain.**

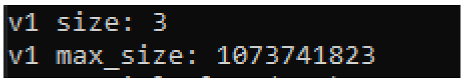
*No, this is because lists are doubly linked meaning they don't store each element in order in memory like arrays do. This means we need to use an iterator to cycle through the list to get an object at a certain position. The advantage of this is that we can insert and remove items as we please as their location in memory is not important.*

**Q.8 [line 119] Is there a reason to use a list instead of a vector?**

*Yes, if order is important and you're going to be doing a lot of insertion, deletions and rearranging then a list is better as you are less restricted in that regard. It is also necessary if the objects you are storing are too large and need more memory allocated to them.*

**Q.9 [line 122] Was max_size and size the same? (Can they be different?)**

*No, so they can be different.*



```
v1 size: 3
v1 max_size: 1073741823
```

**Q.10 [line 123] Which ParticleClass constructor was called?**

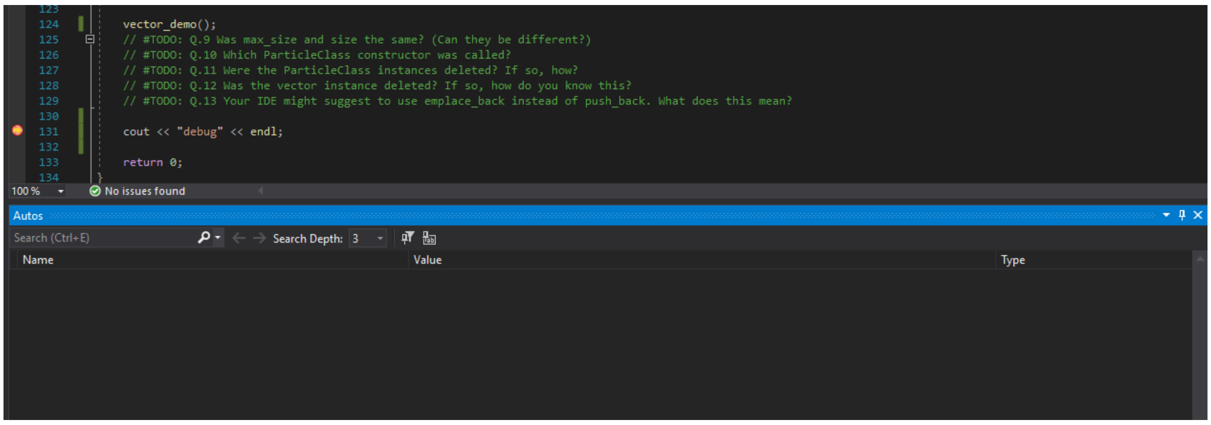
*The regular (not default) constructor.*

**Q.11 [line 124] Were the ParticleClass instances deleted? If so, how?**

*Yes, they were stored on the stack so they were deleted at the end of the scope.*

**Q.12 [line 125] Was the vector instance deleted? If so, how do you know this?**

*Yes, it was also on the stack so it also automatically got deleted. We can check using a breakpoint and see that it isn't there.*



```
    vector_demo();
    // #TODO: Q.9 Was max_size and size the same? (Can they be different?)
    // #TODO: Q.10 Which ParticleClass constructor was called?
    // #TODO: Q.11 Were the ParticleClass instances deleted? If so, how?
    // #TODO: Q.12 Was the vector instance deleted? If so, how do you know this?
    // #TODO: Q.13 Your IDE might suggest to use emplace_back instead of push_back. What does this mean?

    cout << "debug" << endl;

    return 0;
}
```

**Q.13 [line 126] Your IDE might suggest to use emplace_back instead of push_back. What does this mean?**

*Both put the object into the vector but work slightly differently. push_back will construct the object and then copy it into the container while emplace_back will construct the object within the container.*