

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

- The default access for members of a struct are public by default
- The default access for members of a class are private by default
- When deriving a struct, the default access specifier is public
- When deriving a class, the default access specifier is private

Q.2 [line 63] What are function declarations?

It tells the compiler about the function's name, return type and parameters. Basically how to call that function.

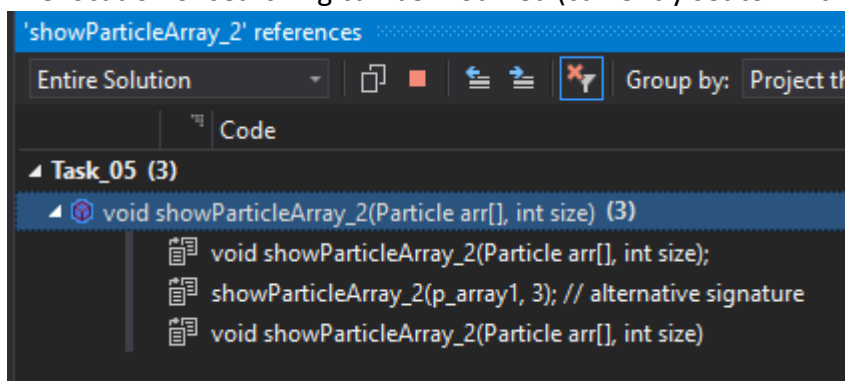
This is necessary when you **define** a function in a source file, then you call that function in another file.

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

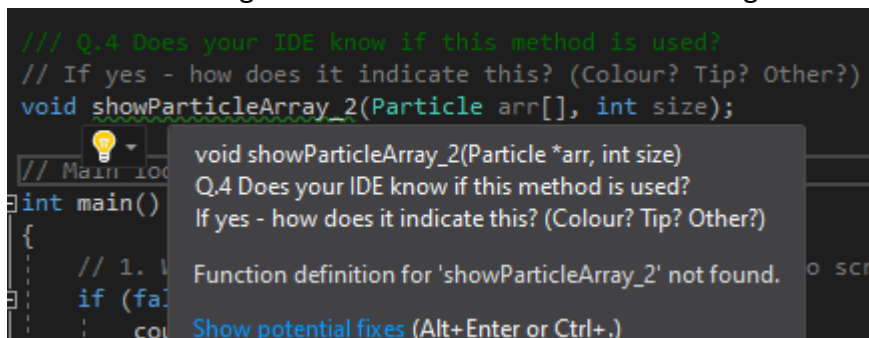
- A parameter name is required when a compiler needs to know what variable to read/write into.
- A function declaration alone is just a "blueprint" of how a function should work, it's not a working function, so the compiler doesn't need to know parameter names since it just need to know how to create the function when it needs to call it in a different file, not how to work it.

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

If you press Shift + F12, it'll show where this method is being used in the current document. The location of searching can be modified (currently set to "Entire Solution").



There would be a green underline under it if it's not being used.



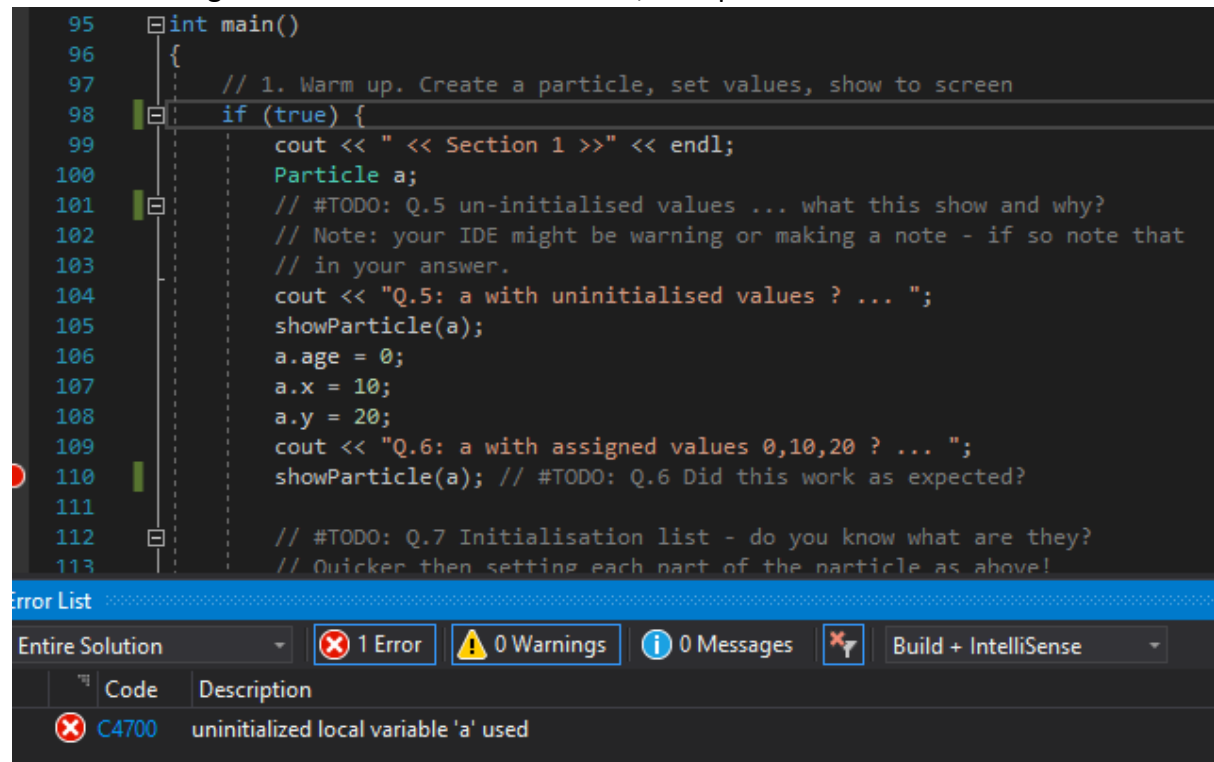
Q.5 [line 86] un-initialised values ... what this show and why?

It doesn't show anything . . . It's probably the settings of the IDE that suppressed this, since it happens for me in C# in the same IDE.

```
Particle a;
// #TODO: Q.5 un-initialised values ... what this show and why?
// Note: your IDE might be warning or making a note - if so note that
// in your answer.
```

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

Yes. We didn't give "a" a value. So it didn't work, as expected.



The screenshot shows a Visual Studio IDE with a C# file. The code is as follows:

```
95 int main()
96 {
97     // 1. Warm up. Create a particle, set values, show to screen
98     if (true) {
99         cout << " << Section 1 >>" << endl;
100         Particle a;
101         // #TODO: Q.5 un-initialised values ... what this show and why?
102         // Note: your IDE might be warning or making a note - if so note that
103         // in your answer.
104         cout << "Q.5: a with uninitialised values ? ... ";
105         showParticle(a);
106         a.age = 0;
107         a.x = 10;
108         a.y = 20;
109         cout << "Q.6: a with assigned values 0,10,20 ? ... ";
110         showParticle(a); // #TODO: Q.6 Did this work as expected?
111
112         // #TODO: Q.7 Initialisation list - do you know what are they?
113         // Quicker then setting each part of the particle as above!
```

Below the code, the Error List is visible, showing one error:

Code	Description
C4700	uninitialized local variable 'a' used

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

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

It does.

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

unsigned int means it can't be negative, yet p1 has -1 as age

```
p1 = getParticleWith(-1,2,3);
cout << "Q.9: p1 with -1,2,3 ? ... ";
```

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

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

You're not taking the references of p (by using ->), but you're taking p's values (by using .)
So if you pass p1 into it, it would look like 1 = 5, 1 = 6 and 1 = 7, not p1's age = 5, etc.

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

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

Eg. ptr -> a

You'll be using the memory address (ptr) to pass the "properties" of that variable (a), not passing its "properties' values" like Q10

(This helped me find words to use for explanation) <https://www.tutorialspoint.com/What-is-arrow-operator-in-Cplusplus>

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

It's like when you do 5 * (1 + 2) and you get 15. It's for indicating precedence. Without it, you're essentially saying "get the memory address of this value", which doesn't make sense ... like you can't get a memory address of number 6.

```
if (*p1_ptr.age == p1_ptr->age) cout << " - TRUE!"; else cout << " - False!";
cout << " ";
// Extra: What does the arrow operator mean? If so, replace the two if lines above.
// #TODO: What does the arrow operator mean? (in words)?
// #TODO: What does the arrow operator mean? (in words)?
// Tip: State what it means, or what it would mean if we didn't write it.
```

(Answer found here) <http://www.cplusplus.com/forum/beginner/138279/>

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

p1

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

```
Particle p1 = getParticleWith(5,5,5);

Particle getParticleWith(int age, int x, int y)
{
    Particle result;
    result.age = age;
    result.x = x;
    result.y = y;
    return result;
}
```

Because of the way p1 was create, it's stored on the stack

If it was `Particle* p1 = new Particle;` then it would be on the heap

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

No it hasn't changed.

Before (memory address is affc18)

```
174 | showParticle(*p1_ptr);
175 | // #TODO: Q.13 What is p1_ptr pointing to now? (Has it changed?)
    | p1_ptr 0x00affc18 {age=5 x=5 y=5}
```

After (same memory address, but with different values)

```
186 | showParticle((*p1_ptr));  
187 | // #TODO: Q.16 Is the current value of p1_ptr good or bad? Explain  
    | p1_ptr 0x00affc18 {age=7 x=7 y=7}
```

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

It's good. The pointer shouldn't change because p1 is still p1, it just has a different value now because a new particle struct was assigned to it after being created.

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

Since the question exists on line 190 (outside of the if statement), I'm gonna assume you're asking if p1 still exists outside of the if statement. No. Because p1 was created inside of it, and after every scope (that is every loop, if statement, or even empty { }), everything stored in the stack would get deleted.

Research: https://www.youtube.com/watch?v=wJ1L2nSIV1s&ab_channel=TheCherno

```
148 | if (true) {  
149 |     cout << " << Section 4 >>" << endl;  
150 |     Particle *p1_ptr;  
151 |     // set b to be something sensible  
152 |     Particle p1 = getParticleWith(5,5,5);  
153 |     cout << "p1 with 5,5,5 ? ... ";  
154 |     showParticle(p1);  
155 |     // get address of b, keep it ...  
156 |     p1_ptr = &p1;  
157 |     cout << "Address of p1:" << &p1 << endl;  
158 |     cout << "Value of p1_ptr:" << p1_ptr << endl;  
159 |  
160 |     // Note that (*p1_ptr).age gets the p1.age value, so ...  
161 |     cout << "Q.11 and Q.12: Test results ..." << endl;  
162 |     if ((*p1_ptr).age == p1.age) cout << " - TRUE!"; else cout << " - False";  
163 |     cout << endl;  
164 |     // Note that (*p1_ptr).age is the same as p1_ptr->age  
165 |     if ((*p1_ptr).age == p1_ptr->age) cout << " - TRUE!"; else cout << " - False!";  
166 |     cout << endl;  
167 |     // Extra: Does C++ have a ternary operator? If so, replace the two if lines above.  
168 |     // #TODO: Q.11 So what does -> mean (in words)?  
169 |     // #TODO: Q.12 Do we need to put ( ) around *p1_ptr?  
170 |     // Tip: State what it means, or what it would mean if we didn't write it.  
171 |  
172 |     // pass the dereferenced pointer as argument  
173 |     cout << "Q.13: p1 via dereferenced pointer ... ";  
174 |     showParticle(*p1_ptr);  
175 |     // #TODO: Q.13 What is the dereferenced pointer (from the example above)?  
176 |  
177 |     // update p1, ...  
178 |     p1 = getParticleWith(7,7,7);  
179 |     // Note: p1 is now a new particle struct with new values. So, ...  
180 |     // #TODO: Q.14 Is p1 stored on the heap or stack?  
181 |     // #TODO: Q.15 What is p1_ptr pointing to now? (Has it changed?)  
182 |     // Tip: Use your IDE inspector to check the "address" of p1 and value of p1_ptr  
183 |     cout << "values of new p1 ? ... ";  
184 |     showParticle(p1);  
185 |     cout << "particle values at p1_ptr ?... ";  
186 |     showParticle(*p1_ptr);  
187 |     // #TODO: Q.16 Is the current value of p1_ptr good or bad? Explain  
188 |  
189 | }  
190 | // #TODO: Q.17 Is p1 still available? Explain.
```

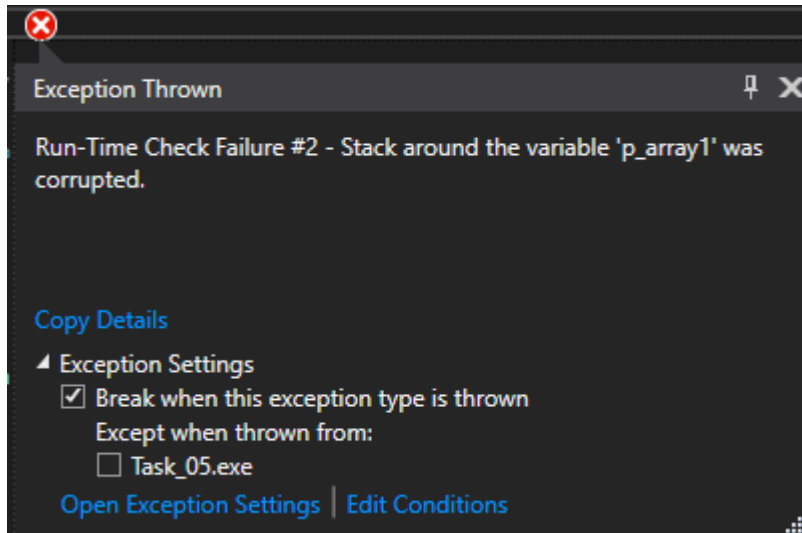
Q.18 [line 180] <deleted - ignore> :)

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

Yes.

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

You're assigning something outside the range of the array.



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

The for loop only goes to just before 3, which is 2, and p_array[2] exists

```
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.
    cout << "showParticleArray call ..." << endl;
    for (int i = 0; i < size; i++) {
        cout << " - pos=" << i << " ";
        showParticle(p_array[i]);
    }
}
```

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

```
array size n is: " << (sizeof(p_array1) / sizeof(p_array1[0]))
```

Step 1. Get the byte size of the entire array

Step 2. Get the byte size of an element in the array (or the type of the array that works too)

Step 3. Divide the result Step 1 by Step 2

Answer found here <https://en.cppreference.com/w/cpp/language/sizeof>

Q.23 [line 375] What is the difference between this function signature and
Function Signature

- A function's **signature** includes the function's name and the number, order and type of its formal parameters.

The first version takes a pointer to the array, the second version takes a value.

Q.24 [line 380] Uncomment the following. It gives different values to those we saw before
Yes, because we passed the entire array into the function. And by doing this, it would only pass the address of the first element into the function, resulting in sizeof finding the size of a hex number, resulting in 4 being displayed.

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

Values from p_array1 are shown. This makes sense because these exist on the stack, and the way the stack works is whatever comes first comes out last, which is what we see here. The reason for the gibberish in between is the program setting some space between the data for safety. Realistically when you release the code, these would be right next to each other.

```
Q.25: Array position overrun ...
showParticleArray call ...
- 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, -858993460)
- pos=4 Particle: (age=3435973836), (x,y)=(-858993460, -858993460)
- pos=5 Particle: (age=3435973836), (x,y)=(-858993460, -858993460)
- pos=6 Particle: (age=3435973836), (x,y)=(-858993460, -858993460)
- pos=7 Particle: (age=3435973836), (x,y)=(-858993460,1)
- pos=8 Particle: (age=2), (x,y)=(3,4)
- pos=9 Particle: (age=5), (x,y)=(6,7)
- pos=10 Particle: (age=8), (x,y)=(9, -858993460)
```

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

Hex is the memory address of a variable used by pointers.

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

It assigned a new memory address to the pointer. This exists on the heap (refer to Q14)

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

It deleted the value from the heap.

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

It can't return a value, cuz there's nothing there to return. You can still return the address, but what was at the address is no longer there.

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

They're the same

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

Crashed. You can't read a memory address to nowhere...

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

No because it's valid, meaning user can access it, finds nothing and crash the program.

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

True. If we don't, this will cause memory leaks and greatly hinder performance.

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

If you don't set your pointer to NULL when initializing it, it could be pointing to an inaccessible piece of memory and crash the program, or an accessible piece of memory that you didn't want the user to see in the first place.

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

```
int *myArray = new int[SIZE];
```