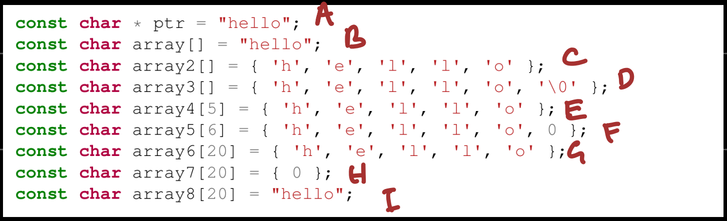
COMP2017 - Tutorials

week 1

Question 5



```
const char * ptr = "hello";
const char array[] = "hello";
const char array2[] = { 'h', 'e', 'l', 'l', 'o' };
const char array3[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
const char array4[5] = { 'h', 'e', 'l', 'l', 'o' };
const char array5[6] = { 'h', 'e', 'l', 'l', 'o', 0 };
const char array6[20] = { 'h', 'e', 'l', 'l', 'o' };
const char array7[20] = { 0 };
const char array8[20] = "hello";
```

A: ptr is a pointer to a constant character string. In this case "Hello". Whatever it points to cannot be modified but pointer can be reassigned.

B: A constant character Array initialised w/ string "Hello". It will add a null terminator at end making size = 6.

C: constant character Array Initialized with h,e,l,l,o. no explicit null terminator so not valid C string for string manipulation function.

D: Constant character Array w/ null terminator at the end.

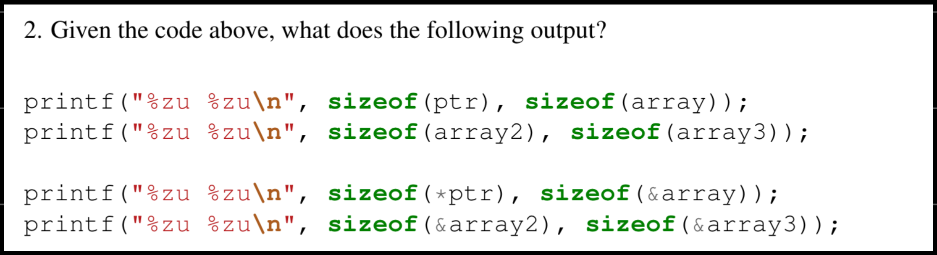
E A char string Array with 5 spaces however no space for null terminator so invalid

F. A char string Array with 6 spaces & includes null terminator

G. 20 character initialised. 5 being hello & 15 are NULL.

H. Specified size 20: all null {0}

I. "hello" include with null terminator +14 nulls.



2. Given the code above, what does the following output?

```
printf("%zu %zu\n", sizeof(ptr), sizeof(array));
printf("%zu %zu\n", sizeof(array2), sizeof(array3));

printf("%zu %zu\n", sizeof(*ptr), sizeof(&array));
printf("%zu %zu\n", sizeof(&array2), sizeof(&array3));
```

ptr = 8 or 4 array = 6

array2 = 5 array3 = 6

*ptr = 1 &array = 8 or 4

&array2 = 8 or 4 &array3 = 8 or 4

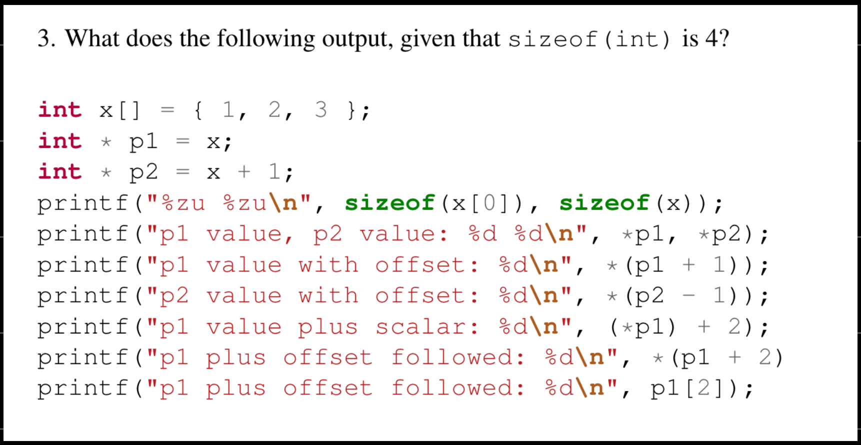
· *ptr ← dereferences to first character of ptr.

ptr ⟶ "hello"

↗

1

so sizeof(*ptr) = 1


3. What does the following output, given that `sizeof(int)` is 4?

```
int x[] = { 1, 2, 3 };
int * p1 = x;
int * p2 = x + 1;
printf("%zu %zu\n", sizeof(x[0]), sizeof(x));
printf("p1 value, p2 value: %d %d\n", *p1, *p2);
printf("p1 value with offset: %d\n", *(p1 + 1));
printf("p2 value with offset: %d\n", *(p2 - 1));
printf("p1 value plus scalar: %d\n", (*p1) + 2);
printf("p1 plus offset followed: %d\n", *(p1 + 2)
printf("p1 plus offset followed: %d\n", p1[2]);
```

4 ✓ 12 ✓

p1 value, p2 value: 1 ✓, 2 ✓

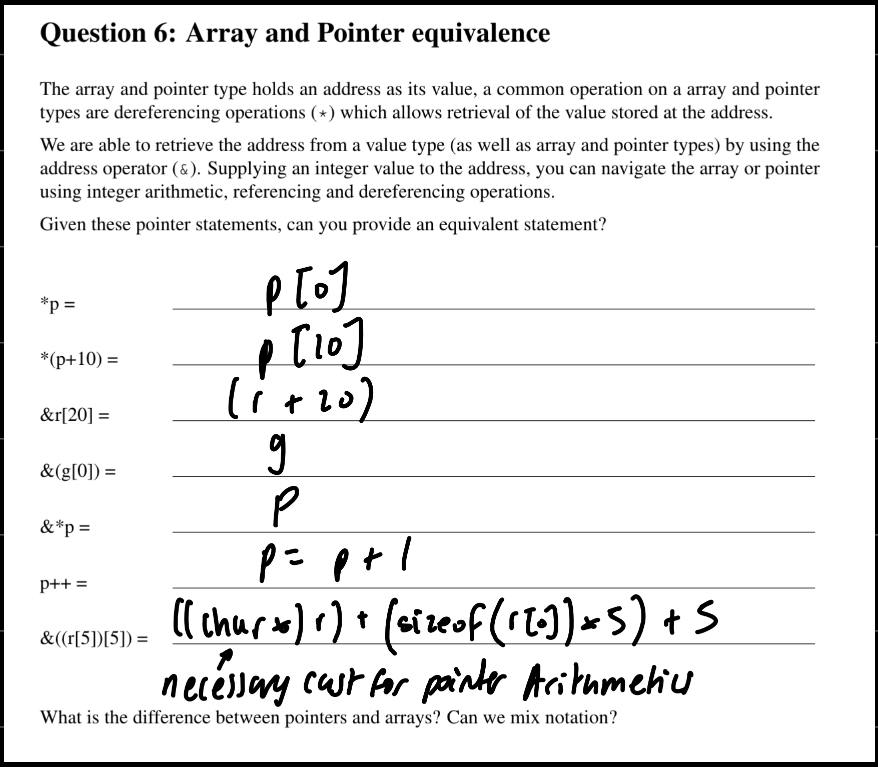
p1 value with an offset: 2 ✓

p2 ~~value~~ value with an offset: 1 ✓

p1 value plus scaler: 3 ✓

p1 plus offset followed 3 ✓

p1 plus offset followed: 3 ✓


Question 6: Array and Pointer equivalence

The array and pointer type holds an address as its value, a common operation on a array and pointer types are dereferencing operations (*) which allows retrieval of the value stored at the address.

We are able to retrieve the address from a value type (as well as array and pointer types) by using the address operator (&). Supplying an integer value to the address, you can navigate the array or pointer using integer arithmetic, referencing and dereferencing operations.

Given these pointer statements, can you provide an equivalent statement?

*p = p[0]

*(p+10) = p[10]

&r[20] = (r + 20)

&(g[0]) = g

&*p = p

p++ = p = p + 1

&((r[5])[5]) = ((char *) r) + (sizeof(r[0]) * 5) + 5

necessary cast for pointer Arithmetics

What is the difference between pointers and arrays? Can we mix notation?

pointers are variables that can store memory addresses
such as

const char *ptr = "Hello"

will point to the memory address of "H" which is the
first element of "Hello". This can be closely refered to
as Arrays which holds elements stored in contigous
memory locations.