Part 1:

Question 1:

2. Your Sunday submission will be marked with no penalty.
Submissions up to 72 hours after the deadline are accepted with no penalty.

Question 2:

4.

Question 3:

2. Typed up PDF.

Part 2:

Question 4:

(size_t): the macro returns a value of type size_t.
(type *) 0: A null pointer parsed as the type of the input parameter, in our case either of the ‘byname’ or ‘bysize’ structs.
(-> field): Access the field of the struct with the name given by the ‘field’ parameter.
&: Return the address of that field.

```
( size_t )( & ( ( (type *) 0 ) -> field ) )
```

The macro returns the address of a field in a struct. Using a null pointer makes these addresses start at zero, so we can see the relative start addresses of different fields clearly.

Question 5:

The sizes of the two structures are different because of padding. With C structures there are two rules to the padding of variables in memory:

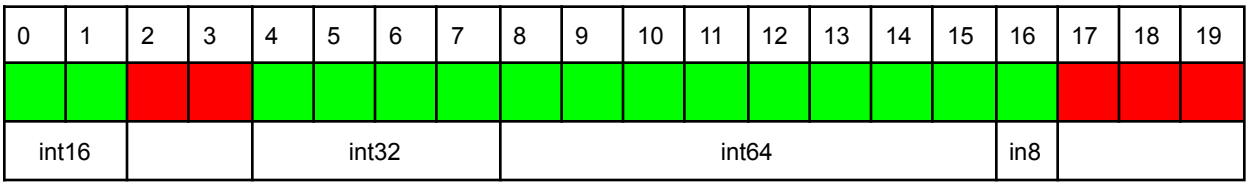
1. A variable of a given type must be stored on an address that is a multiple of its type’s size. Eg. int32 is 4 bytes long so it must be stored on a memory address that is a multiple of 4.
2. A structure’s size must be a multiple of its largest data type. Eg. a struct contains an int8 and an intptr, this is a total of only 9 bytes but the struct will be padded to the next multiple of 8 which is 16 bytes.

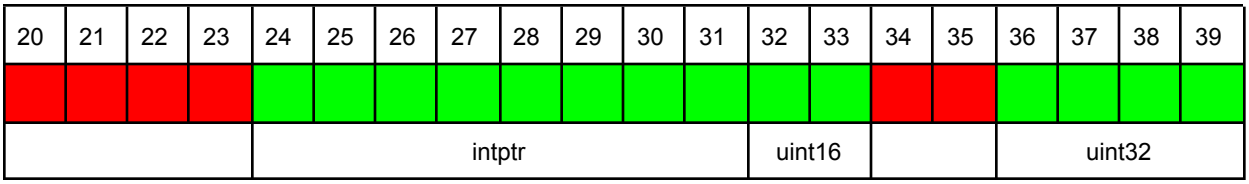
Depending on the order fields are defined in, the size of a structure can vary despite storing the same variables. Diagrams of the two structures in memory are shown below.

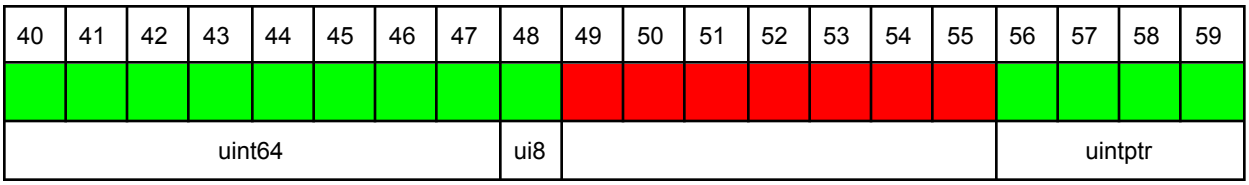
By Name:

Type	int16	int32	int64	int8	intptr	uint16	uint32	uint64	uint8	uintptr
Size	2	4	8	1	8	2	4	8	1	8

This order of the fields results in the following padding:



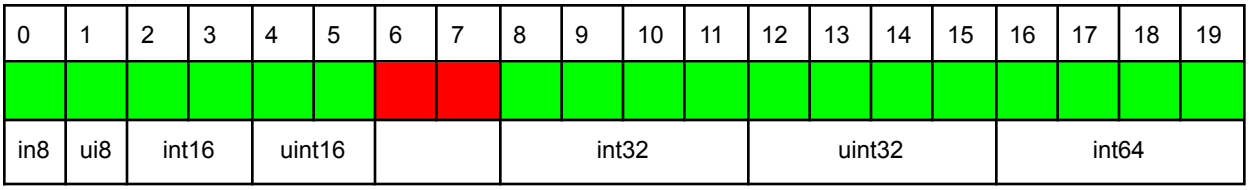


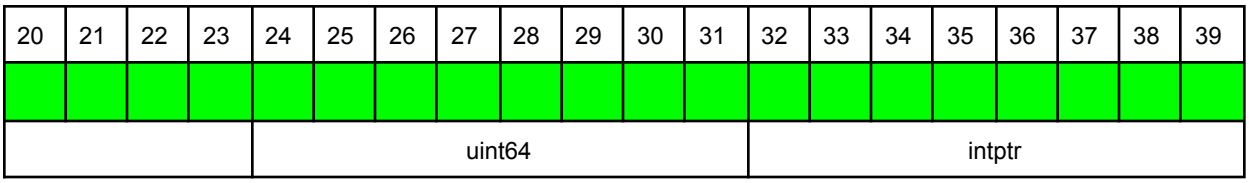


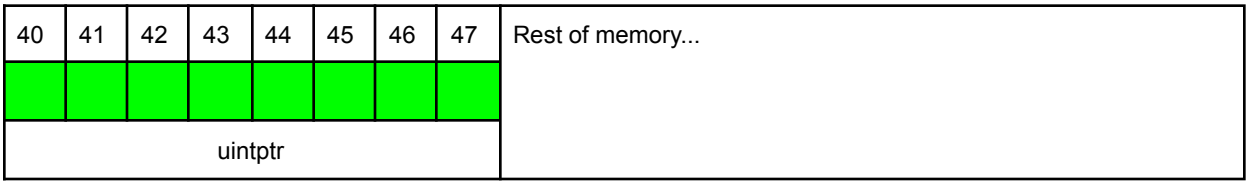
By Size:

Type	int8	uint8	int16	uint16	int32	uint32	int64	uint64	intptr	uintptr
Size	1	1	2	2	4	4	8	8	8	8

This order of the fields is much more efficient, resulting in far less padding:







Question 6:

To get the smallest struct possible we want to make all the fields fill up memory until a multiple of the next field. This can be done by pairing up similar sized fields, such as:

Type	int8	uint8	int16	int32	uint32	int64	uint64	intptr	uintptr	uint16
Size	1	1	2	4	4	8	8	8	8	2

This is the smallest possible struct size of 48 bytes, only wasting 2 bytes of memory due to padding.

To get the largest possible struct we want to alternate small and large types so that memory is lost to padding. This can be done with the following structure:

Type	int8	int64	uint8	uint64	int16	intptr	uint16	uintptr	int32	uint32
Size	1	8	1	8	2	8	2	8	4	4

This ordering is the least memory efficient with a size of 72 bytes. This struct wastes 24 bytes of memory per instance.

Question 7:

Bysize wastes 2 bytes for each instance. This is because int32 has a size of 4 bytes so it can only start on memory addresses that are a multiple of 4. The int8, uint8, int16 and uint16 before this type have a total memory of 6 bytes, so 2 bytes must be padded before int32 starts.