Part 2

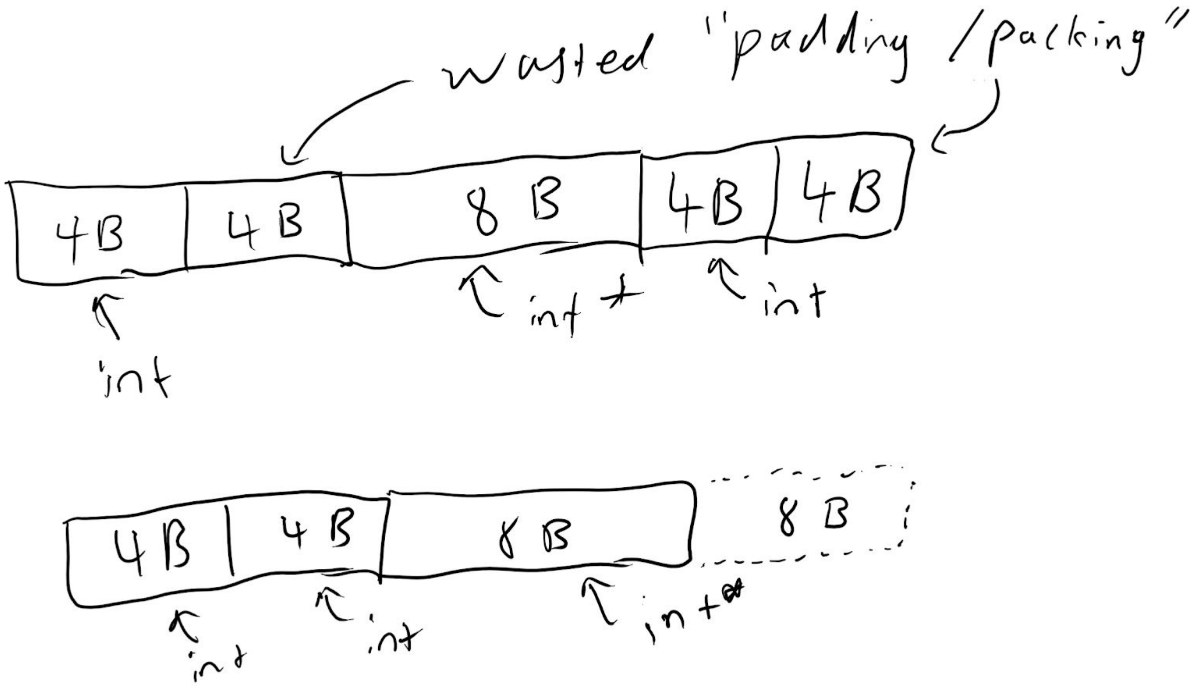
Question 6)

Casts the O(type, field) to (size_t)(&(((type *)0)->field)). This means that whenever the compiler receives the O data structure, it converts the input into a size_t (unsigned int), where the type pointer's "field" field will be dereferenced, and the result will be converted into an unsigned int.

Question 7)

One of either the OS or compiler packs the data up on a read first basis.

In the example below, the first operation is allocating an int, followed by an int*, then another int. For dereferencing reasons, the data is stored in blocks of modulo (sizeof (type)). In the case of the int*, it can't occupy the 4 bytes space after the int, because int* are 8 bytes long and the end of the stack is currently 4 away from mod 8. This means that the int* has to leave a 4 byte gap. The second version represents a more optimal approach of storing the data.



Question 8)

Assuming this is c, the min size this can have is 48 bytes, as there has to be padding to align the tail of the object, so technically, bysize is one way to align the data structure, such that the padding is the smallest.

Here's another way to do it:

```
uint64_t uint64;
int64_t int64;
intptr_t intptr;
uintptr_t uintptr;
int32_t int32;
uint32_t uint32;
int16_t int16;
uint16_t uint16;
int8_t int8;
uint8_t uint8;
```

Max: (trying to create as much padding as possible)

```
struct max {
uint64_t uint64;
uint8_t uint8;
int64_t int64;
int8_t int8;
intptr_t intptr;
uint16_t uint16;
uintptr_t uintptr;
int16_t int16;
int32_t int32;
uint32_t uint32;
```

By trial and error this produced a 72 byte object, which is likely the largest this data structure can get.

Question 9)
0 bytes are wasted when using a data structure with these sizes. When adding the sizes up, the result is 46 bytes. Most processors need this to be a multiple of 8, so the smallest the size can be is 48.