Headers all live in memory

* Say you want to allocate 40 bytes to memory
* The header could be before the 40 bytes allocated.
* The freelist header could sit after the 40 bytes allocated, indicating that it is an unallocated block

Pointer arithmetic

Void \* vlad\_alloc (size\_t n Bytes);

Void vlad\_free(void \* p);

What is a void pointer?

* It points to void, which in C means: “there isn’t anything in here”

We can take this void pointer in our code and do something like:

* Struct game \*g = vlad\_alloc(sizeof (struct game));

All pointers have the same size

* How big is the size of a pointer? 🡪 depends
* The length of a pointer on a system is a function of that system
* E.g. 32bit architecture 🡪 32bit sized pointer

If you add 1 to a void pointer, what happens?

What other property does a pointer have?

* It is a number
* The fact that it is a number, we can do number things to it like ADD / SUBTRACT
* What happens if I add 1 to this pointer?
  + The pointer moves and points to something else
  + It moves 1 sizeof across memory (4 bytes if it is an int)
  + E.g
  + Int n = 1

Int \* p = &n

P++  
  
Where am I pointing now?

I am now pointing to the address of P

If you dereference this, you will get the value of P as a number

How about P--? Where am I pointing now?

I am pointing back to n

Why do pointers work in this way?

ISO/IEC 9899-1999 OR N1256

* A standard for C

Vlad\_free takes in a pointer and gives you back void

Size of VOID = zero?

**Typedef / Types**

Typedef struct game\* Game;

Type Typedefs you will never come across

* Void, int, long etc.
* Most of these things are given / implicit

Int \*p = &n;

(struct game\*)p; 🡪 This is a “typecast”

Int \*p = &n;

((void\*)p+1)  
🡪 It will move everything 1 byte across  
🡪 This will end up possibly pointing to the 2nd byte of the int.

**[ ] [ ] [ ] [ ] [ ]**

[ ] [ ] [ ] [ ] 🡪 It creates a byte to the next variable

For the assignment, use “byte”

If I have an array of bytes:

e.g. byte memory [1024];

- What happens when I say memory[4] 🡪 \*(memory + 4); / deference memory + 4

char \*s = “Hello, world!”;

* s[3]; 🡪 L
* 3[s]; 🡪 L / 3 somehow becomes something we can index on. This decomposes to the same reference as s + 3

When you test vlad, should you do it when you have finished your initialising or allocating function

* Once you have done the initialising function, you can start the program
  + However, it won’t work when you haven’t finished the allocation functions

Traverse your free blocks

* Blocks are nodes
* If I do: start of free block -> next
* I have to make a function that converts it from integer indices to a pointer
* You also want a function that converts pointers to integer indices

What does static do?

* It changes a parameter of the function to make it no longer visible
* Inline: It says to the compiler, when anyone calls this function, replace this function
  + It makes things go faster when you call it
* Static inline blahblahblah

Does #define a function make it faster than doing static inline?

* Yes, but it is a little hacky

**TUTORIAL**

Bitwise operations

* X = 0xCC in dec? 🡪 204  
  in binary? 🡪 11001100
* Y = 0x55 in dec? 🡪 85  
  in binary? 🡪 010101
* Z = 0

^ exclusive or  
~ not (bit flip)

<< bitwise shift  
>> bitwise shift

**Exlusive Or ( ^ )**

A or B but not both.  
(x^y)^x = y

Exercise: write a function showBits(w) that displays the individual bits in Word w on the standard output. Display the bits left-to-right from bit 31 down, to bit 0.