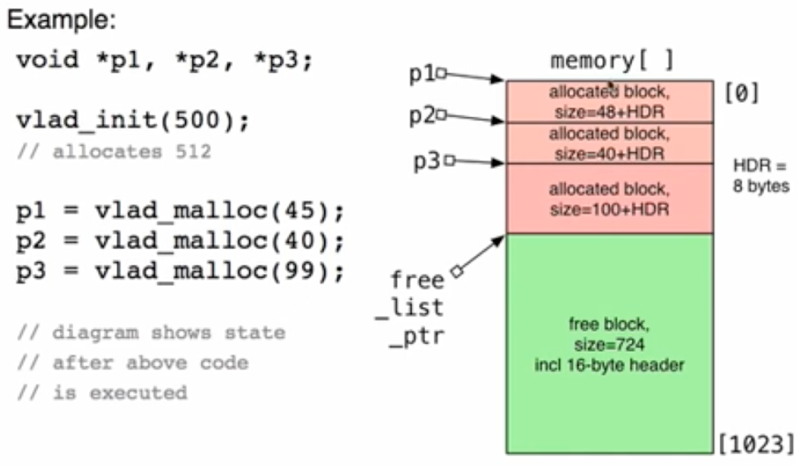
**Assignment 1**

* Implement a memory allocator
* Dynamic data structure
  + You don’t know how big to start with, so you dynamically add data
* Aims of assignment:
  + Learn about how memory is managed
  + Doubly linked lists
  + More practice with C and data structures
* DUE **Monday 29th August at 9:00am**
* What to do?

1. vlad\_init(vsize\_t MAX)

* Initialises allocator
* (size = power of 2, min = 1024)
* If it is not, then round up to power of 2

1. vlad\_malloc(vsite\_t N)

* Reserves block N bytes inside initial allocated  
  chunk, returns address as void \*
* Returns N bytes + Header HDR
* (size = multiple of 4)
* If it is not, then round up to multiple of 4

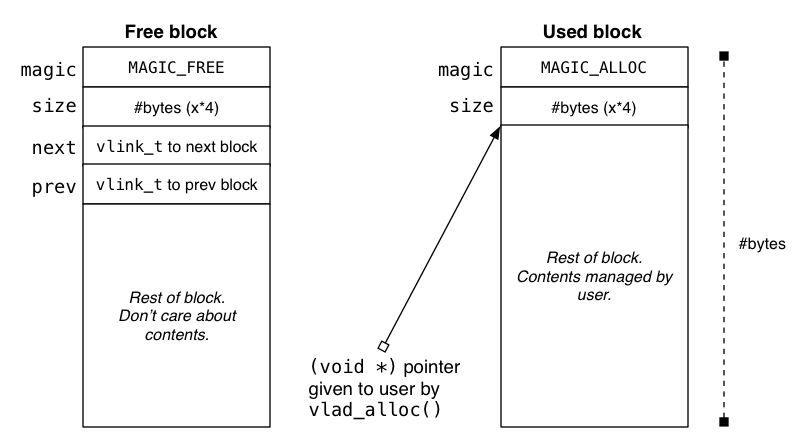
where vsize\_t = unsigned int type

The header tells you that:

* It is an allocated block
* How big the block is

The blocks aren’t AREN’T allocated are in a FREE LIST.

* 1st block in the free list = always the one with the LOWEST MEMORY ADDRESS
  + It is pointer to by the FREE LIST POINTER
* There can be multiple free blocks

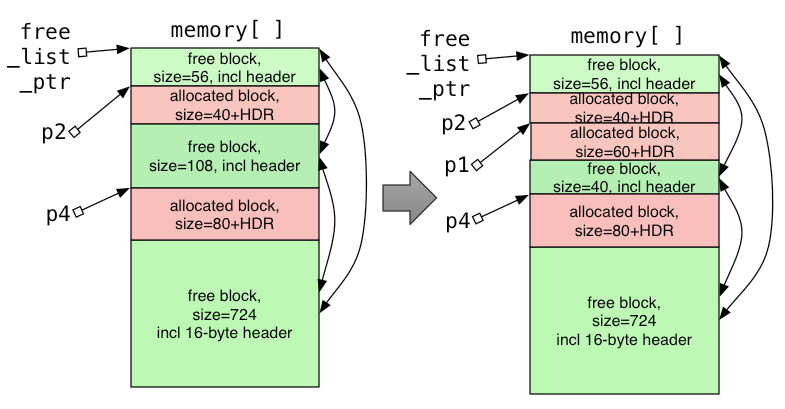


**Vlad\_free function**

* User wants to free up specific allocated memory block, pointed to by pointer P3
* Function will:
  + know where P3 is pointing
  + know how large the block is
  + be able to make the block “look like a free block” and link in to the FREE LIST
  + Because it is newly allocated to the free block, it becomes the new start of the free list (the 1st free block)
  + I.e. New free block will point to old free block, then old free block points back to the new free block.  
    (therefore connecting the free blocks together into a “single block”)
* If user requests for memory size:
  + SOMEWHAT LESS or EQUAL than the available free block, then it can use that specific free block (even if its not exact size).
  + SIGNIFICANTLY LESS, then it will only use partial free block, then link the rest back to the old free block
  + LARGER than the available block, it will use up the older, larger free block.

**According to the project spec, when allocated memory (vlad\_alloc), we should choose the free block that is LARGER and CLOSEST in size** **to use up**.

Diagram: Vlad\_malloc(60) function, before allocation + after allocation



The allocated block size = **user memory alloc request + 8byte header**

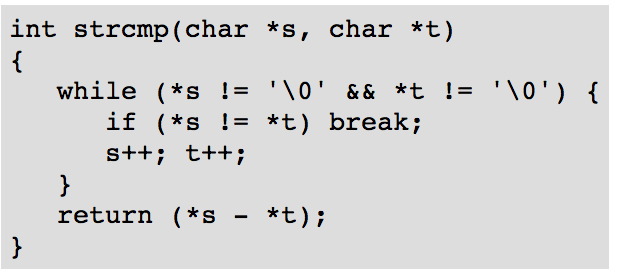
**Vlad\_merge** function

* Instead of having many small fragments of free blocks which would be unusable, the free block should be merged into a larger block of memory ready for use.

Different approaches to allocating:

* You can allocate using the largest size block
* You can allocate using the most fit size block
* Any way you want (read the specs though)

Hints:

* **Void \*** … real memory address (C pointer)
* **Byte \*** … real memory address (C pointer)
* **Vaddr\_t** … offset within **memory[]** chunk
  + Just references memory addresses
* **Vlink\_t** … offset within **memory[]** chunk
  + Inside headers with the prev / next nodes are all done with **vlink\_t**
* From outside the ADT, the user only sees **void \*** pointer values
* **Void\***, **vaddr\_t**, **vlink\_t** all refer to locations in **memory[]**
* Need ways to map between **void \*** and **vlink\_t** values.
* In C, we can do arithmetic on pointer values
  + Consider: **Type \*p; … p = p + n;**
  + New value of **p** is calculated as **p + n\*sizeof(Type)**  
    (numeric input x size of type that you’re pointing at)
  + Applies to ints, chars, etc.
    - E.g. Int = 4 bytes / char = 1 byte
  + A use for pointer arithmetic: **string comparison**
    - 
* Another use for pointer arthimetic: **fast array access**
  + In theory, accessing memory via. Pointers than Arrays are faster because:  
    POINTERS: Directly access memory locations  
    ARRAYS: Indices are processed, then memory locations calculated / accessed

**Sortlab Program**

A program that runs sorting algorithms and analyses them.