

Malloc Recitation

Ben Spinelli

Recitation 11: November 9, 2015

Agenda

- **Macros / Inline functions**
- **Quick pointer review**
- **Malloc**

Macros / Inline Functions

Macros

- **Pre-compile time**

- **Define constants:**

- `#define NUM_ENTRIES 100`
- OK

- **Define simple operations:**

- `#define twice(x) 2*x`
 - Not OK
 - `twice(x+1)` becomes `2*x+1`
- `#define twice(x) (2*(x))`
 - OK
- Always wrap in parentheses; it's a naive search-and-replace!

Macros

■ Why macros?

- “Faster” than function calls
 - Why?
- For malloc
 - Quick access to header information (payload size, valid)

■ Drawbacks

- Less expressive than functions
- Arguments are *not* typechecked, local variables
 - This can easily lead to errors that are more difficult to find

Inline Functions

- **What's the keyword `inline` do?**
 - At **compile-time** replaces “function calls” with code
- **More efficient than a normal function call**
 - Less overhead – no need to set up stack/function call
 - Useful for functions that are
 - Called frequently
 - Small, e.g., `int add(int x, int y);`

Differences

- **Macros done at pre-compile time**
- **Inline functions done at compile time**
 - Stronger type checking / Argument consistency
- **Macros cannot return anything (why not?)**
- **Macros can have unintended side effects**
 - `#define xsquared(x) (x*x)`
 - What happens when `xsquared(x++)` is called?
- **Hard to debug macros – errors generated on expanded code, not code that you typed**

Macros / Inline Functions

- **You will likely use both in malloc lab**
- **Macros are good for small tasks**
 - Saves work in retyping tedious calculations
 - Can make code easier to understand
 - `HEADER(ptr)` versus doing the pointer arithmetic
- **Some things are hard to code in macros, so this is where inline functions come into play**
 - More efficient than normal function call
 - More expressive than macros

Pointers: casting, arithmetic, and dereferencing

Pointer casting

■ Cast from

- `<type_a>*` to `<type_b>*`
 - Gives back the same value
 - Changes the behavior that will happen when dereferenced
- `<type_a>*` to integer/ unsigned int
 - Pointers are really just 8-byte numbers
 - Taking advantage of this is an important part of malloc lab
 - Be careful, though, as this can easily lead to errors
- integer/ unsigned int to `<type_a>*`

Pointer arithmetic

- The expression `ptr + a` doesn't mean the same thing as it would if `ptr` were an integer.

- Example:

```
type_a* pointer = ...;  
(void *) pointer2 = (void *) (pointer + a);
```

- This is really computing:

- `pointer2 = pointer + (a * sizeof(type_a))`
- `lea (pointer, a, sizeof(type_a)), pointer2;`

- **Pointer arithmetic on `void*` is undefined**

Pointer arithmetic

- `int * ptr = (int *)0x12341230;`
`int * ptr2 = ptr + 1;`
- `char * ptr = (char *)0x12341230;`
`char * ptr2 = ptr + 1;`
- `int * ptr = (int *)0x12341230;`
`int * ptr2 = ((int *) ((char *) ptr) + 1));`
- `char * ptr = (char *)0x12341230;`
`void * ptr2 = ptr + 1;`
- `char * ptr = (int *)0x12341230;`
`void * ptr2 = ptr + 1;`

Pointer arithmetic

- `int * ptr = (int *)0x12341230;`
`int * ptr2 = ptr + 1; //ptr2 is 0x12341234`
- `char * ptr = (char *)0x12341230;`
`char * ptr2 = ptr + 1; //ptr2 is 0x12341231`
- `int * ptr = (int *)0x12341230;`
`int * ptr2 = ((int *) ((char *) ptr) + 1);`
`//ptr2 is 0x12341231`
- `char * ptr = (char *)0x12341230;`
`void * ptr2 = ptr + 1; //ptr2 is 0x12341231`
- `char * ptr = (int *)0x12341230;`
`void * ptr2 = ptr + 1; //ptr2 is still 0x12341231`

More pointer arithmetic

- `int ** ptr = (int **)0x12341230;`
`int * ptr2 = (int *) (ptr + 1);`
- `char ** ptr = (char **)0x12341230;`
`short * ptr2 = (short *) (ptr + 1);`
- `int * ptr = (int *)0x12341230;`
`void * ptr2 = &ptr + 1;`
- `int * ptr = (int *)0x12341230;`
`void * ptr2 = ((void *) (*ptr + 1));`
- **This is on a 64-bit machine!**

More pointer arithmetic

- `int ** ptr = (int **)0x12341230;`
`int * ptr2 = (int *) (ptr + 1);` `//ptr2 = 0x12341238`
- `char ** ptr = (char **)0x12341230;`
`short * ptr2 = (short *) (ptr + 1);`
`//ptr2 = 0x12341238`
- `int * ptr = (int *)0x12341230;`
`void * ptr2 = &ptr + 1;` `//ptr2 = ??`
`//ptr2 is actually 8 bytes higher than the address of`
`the variable ptr, which is somewhere on the stack`
- `int * ptr = (int *)0x12341230;`
`void * ptr2 = ((void *) (*ptr + 1));` `//ptr2 = ??`
`//ptr2 is one plus the value at 0x12341230`
 - (so undefined, but it probably segfaults)

Pointer dereferencing

■ Basics

- It must be a POINTER type (or cast to one) at the time of dereference
- Cannot dereference expressions with type `void*`
- Dereferencing a t^* evaluates to a value with type t

Pointer dereferencing

- What gets “returned?”

```
int * ptr1 = malloc(sizeof(int));  
*ptr1 = 0xdeadbeef;
```

```
int val1 = *ptr1;  
int val2 = (int) *((char *) ptr1);
```

What are val1 and val2?

Pointer dereferencing

■ What gets “returned?”

```
int * ptr1 = malloc(sizeof(int));  
*ptr1 = 0xdeadbeef;
```

```
int val1 = *ptr1;  
int val2 = (int) *((char *) ptr1);
```

```
//    val1 = 0xdeadbeef;  
//    val2 = 0xffffffffef;
```

Little endian, and sign extension.

What happened??

Malloc

Malloc basics

■ What is dynamic memory allocation?

■ Terms you will need to know

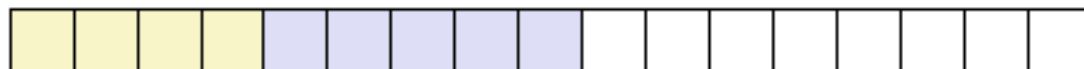
- malloc/ calloc / realloc
- free
- sbrk
- payload
- fragmentation (internal vs. external)
- coalescing
 - Bi-directional
 - Immediate vs. Deferred

Allocation Example

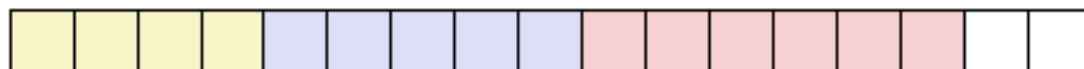
```
p1 = malloc(4)
```



```
p2 = malloc(5)
```



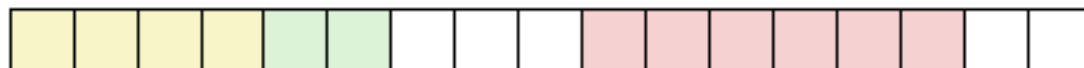
```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(2)
```



Fragmentation

■ Internal fragmentation

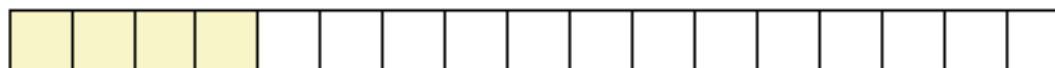
- Result of payload being smaller than block size.
- `void * m1 = malloc(3); void * m1 = malloc(3);`
- `m1, m2` both have to be aligned to 8 bytes...

■ External fragmentation

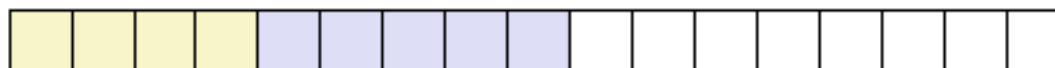
External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

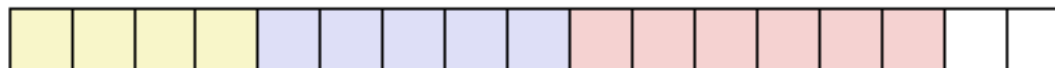
```
p1 = malloc(4)
```



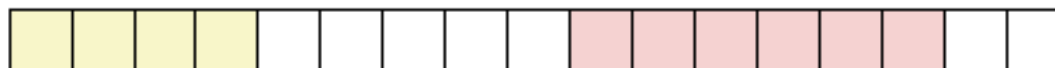
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(6)
```

Oops! (what would happen now?)

- Depends on the pattern of future requests
 - Thus, difficult to measure

Implementation Hurdles

- How do we know where the blocks are?
- How do we know how big the blocks are?
- How do we know which blocks are free?
- Remember: can't buffer calls to malloc and free... must deal with them real-time.
- Remember: calls to `free` only takes a pointer, not a pointer and a size.
- Solution: Need a data structure to store information on the "blocks"
 - Where do I keep this data structure?
 - We can't allocate a space for it, that's what we are writing!

The data structure

■ Requirements:

- The data structure needs to tell us where the blocks are, how big they are, and whether they're free
- We need to be able to CHANGE the data structure during calls to malloc and free
- We need to be able to find the **next free block** that is “a good fit for” a given payload
- We need to be able to quickly mark a block as free/allocated
- We need to be able to detect when we're out of blocks.
 - What do we do when we're out of blocks?

The data structure

- **It would be convenient if it worked like:**

```
malloc_struct malloc_data_structure;  
...  
ptr = malloc(100, &malloc_data_structure);  
...  
free(ptr, &malloc_data_structure);  
...
```

- **Instead all we have is the memory we're giving out.**

- All of it doesn't have to be payload! We can use some of that for our data structure.

The data structure

- The data structure IS your memory!
- A start:
 - <h1> <pl1> <h2> <pl2> <h3> <pl3>
 - What goes in the header?
 - That's your job!
 - Lets say somebody calls free(p2), how can I coalesce?
 - Maybe you need a **footer**? Maybe not?

The data structure

■ Common types

- Implicit List
 - Root -> block1 -> block2 -> block3 -> ...
- Explicit List
 - Root -> free block 1 -> free block 2 -> free block 3 -> ...
- Segregated List
 - Small-malloc root -> free small block 1 -> free small block 2 -> ...
 - Medium-malloc root -> free medium block 1 -> ...
 - Large-malloc root -> free block chunk1 -> ...

Implicit List

- From the root, can traverse across blocks using headers
- Can find a free block this way
- Can take a while to find a free block
 - How would you know when you have to call sbrk?

Explicit List

- **Improvement over implicit list**
- **From a root, keep track of all free blocks in a (doubly) linked list**
 - Remember a doubly linked list has pointers to next and previous
- **When malloc is called, can now find a free block quickly**
 - What happens if the list is a bunch of small free blocks but we want a really big one?
 - How can we speed this up?

Segregated List

- **An optimization for explicit lists**
- **Can be thought of as multiple explicit lists**
 - What should we group by?
- **Grouped by size – let's us quickly find a block of the size we want**
- **What size/number of buckets should we use?**
 - This is up to you to decide

Design Considerations

- I found a chunk that fits the necessary payload... should I look for a better fit or not? (First fit vs. Best fit)
- Splitting a free block:

```
void* ptr = malloc(200);  
free(ptr);  
ptr = malloc(50); //use same space, then "mark" remaining  
bytes           as free
```

```
void* ptr = malloc(200);  
free(ptr);  
ptr = malloc(192); //use same space, then "mark" remaining  
bytes           as free??
```


Design Considerations

- **Free blocks: address-ordered or LIFO**
 - What's the difference?
 - Pros and cons?
- **Coalescing**
 - When do you coalesce?
- **You will need to be using an explicit list at minimum score points**
 - But don't try to go straight to your final design, build it up iteratively.

Heap Checker

■ Part of the assignment is writing a heap checker

- This is here to help you.
- Write the heap checker as you go, don't think of it as something to do at the end
- A good heap checker will make debugging much, much easier

■ Heap checker tips

- Heap checker should run silently until it finds an error
 - Otherwise you will get more output than is useful
 - You might find it useful to add a “verbose” flag, however
- Consider using a macro to turn the heap checker on and off
 - This way you don't have to edit all of the places you call it
- There is a built-in macro called `__LINE__` that gets replaced with the line number it's on
 - You can use this to make the heap checker tell you where it failed

Demo

- Running Traces
- Heap checker
- Using `gprof` to profile