Malloc Recitation

By sseshadr
Agenda

• Macros in C
• Pointer declarations
• Casting and Pointer Arithmetic
• Malloc
Macros
 Macros

• Runtime, compile-time, or pre-compile time?
• Constant:
  – #define NUM ENTRIES 100
  – OK
• Macro
  – #define twice(x) 2*x
    • Not OK
    • twice(x+1) becomes 2*x+1
  – #define twice(x) (2*(x))
    • OK
  – Use lots of parenthesis, it’s a naïve search-and-replace!
Macros

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

• What’s the keyword **inline** do?
  – At **compile-time** replaces “function calls” with code
Pointer declarations
C operators (K&R p. 53)

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()  []  -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>! ~ ++ -- + - * &amp; (type) sizeof</td>
<td>right to left</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>left to right</td>
</tr>
<tr>
<td>= += -= *= /= %= &amp;= ^= != &lt;&lt;= &gt;&gt;=</td>
<td>right to left</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
</tr>
</tbody>
</table>

Note: Unary +, −, and * have higher precedence than binary forms
Review of C Pointer Declarations (K&R section 5.12)

- `int *p`  
  p is a pointer to `int`  

- `int *p[13]`  
  p is an array[13] of pointer to `int`  

- `int *(p[13])`  
  p is an array[13] of pointer to `int`  

- `int **p`  
  p is a pointer to a pointer to an `int`  

- `int (*p)[13]`  
  p is a pointer to an array[13] of `int`  

- `int *f()`  
  f is a function returning a pointer to `int`  

- `int (*f)()`  
  f is a pointer to a function returning `int`  

- `int (**f())[13]()`  
  f is a function returning ptr to an array[13] of pointers to functions returning `int`  

- `int (**x[3])() [5]`  
  x is an array[3] of pointers to functions returning pointers to array[5] of `ints`
Pointer casting, arithmetic, and dereferencing
Pointer casting

• Separate from non-pointer casting
  – float to int, int to float
  – \texttt{<struct\_a>} to \texttt{<struct\_b>}
    • No! gcc error.

• Cast from
  – \texttt{<type\_a>} * to \texttt{<type\_b>} *
  – \texttt{<type\_a>} * to integer/ unsigned int
  – integer/ unsigned int to \texttt{<type\_a>} *
Pointer casting

• What actually happens in a pointer cast?
  – Nothing! It’s just an assignment. Remember all pointers are the same size.
  – The magic happens in dereferencing and arithmetic


• The expression $ptr + a$ doesn’t always evaluate into the arithmetic sum of the two

• Consider:
  
  `<type_a> * pointer = ...;
  (void *) pointer2 = (void *) (pointer + a);

• Think about it as
  
  - leal (pointer, a, sizeof(type_a)), pointer2;
Pointer arithmetic

• int * ptr = (int *)0x12341234;
  int * ptr2 = ptr + 1;

• char * ptr = (char *)0x12341234;
  char * ptr2 = ptr + 1;

• int * ptr = (int *)0x12341234;
  int * ptr2 = ((int *) (((char *) ptr) + 1));

• void * ptr = (char *)0x12341234;
  void * ptr2 = ptr + 1;

• void * ptr = (int *)0x12341234;
  void * ptr2 = ptr + 1;
Pointer arithmetic

- \texttt{int * ptr = (int *)0x12341234;}
  \texttt{int * ptr2 = ptr + 1; \textcolor{red}{//ptr2 is 0x12341238}}

- \texttt{char * ptr = (char *)0x12341234;}
  \texttt{char * ptr2 = ptr + 1; \textcolor{red}{//ptr2 is 0x12341235}}

- \texttt{int * ptr = (int *)0x12341234;}
  \texttt{int * ptr2 = ((int *) (((char *) ptr) + 1));}
  \texttt{\textcolor{red}{//ptr2 is 0x12341235}}

- \texttt{void * ptr = (char *)0x12341234;}
  \texttt{void * ptr2 = ptr + 1; \textcolor{red}{//ptr2 is 0x12341235}}

- \texttt{void * ptr = (int *)0x12341234;}
  \texttt{void * ptr2 = ptr + 1; \textcolor{red}{//ptr2 is still 0x12341235}}
More pointer arithmetic

- int ** ptr = (int **)0x12341234;
  int * ptr2 = (int *) (ptr + 1);

- char ** ptr = (char **)0x12341234;
  short * ptr2 = (short *) (ptr + 1);

- int * ptr = (int *)0x12341234;
  void * ptr2 = &ptr + 1;

- int * ptr = (int *)0x12341234;
  void * ptr2 = ((void *) (*ptr + 1));

- This is on a 64-bit machine!
More pointer arithmetic

- `int ** ptr = (int **)0x12341234;
  int * ptr2 = (int *) (ptr + 1); //ptr2 = 0x1234123c`

- `char ** ptr = (char **)0x12341234;
  short * ptr2 = (short *) (ptr + 1);
  //ptr2 = 0x1234123c`

- `int * ptr = (int *)0x12341234;
  void * ptr2 = &ptr + 1; //ptr2 = ??
  //ptr2 is actually 8 bytes higher than the address of the variable ptr`

- `int * ptr = (int *)0x12341234;
  void * ptr2 = ((void *) (*ptr + 1)); //ptr2 = ??
  //ptr2 is just one higher than the value at 0x12341234 (so probably segfault)`
Pointer dereferencing

• Basics
  – It must be a POINTER type (or cast to one) at the time of dereference
  – Cannot dereference (void *)
  – The result must get assigned into the right datatype (or cast into it)
What gets “returned?”

```c
int * ptr1 = malloc(100);
*ptr1 = 0xdeadbeef;

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

What are val1 and val2?
Pointer dereferencing

• What gets “returned?”

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

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

// val1 = 0xdeadbeef;
// val2 = 0xfffffffffef;

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

\[ p_1 = \text{malloc}(4) \]

\[ p_2 = \text{malloc}(5) \]

\[ p_3 = \text{malloc}(6) \]

\[ \text{free}(p_2) \]

\[ p_4 = \text{malloc}(2) \]
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

\[
\begin{align*}
  p_1 & = \text{malloc}(4) \\
  p_2 & = \text{malloc}(5) \\
  p_3 & = \text{malloc}(6) \\
  \text{free}(p_2) \\
  p_4 & = \text{malloc}(6)
\end{align*}
\]

Oops! (what would happen now?)

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

- How do we know where the chunks are?
- How do we know how big the chunks are?
- How do we know which chunks 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 “chunks”**
  - Where do I keep this data structure?
The data structure

• Requirements:
  – The data structure needs to tell us where the chunks 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 chunk that is “a good fit for” a given payload
  – We need to be able to quickly mark a chunk as free/allocated
  – We need to be able to detect when we’re out of chunks.
    • What do we do when we’re out of chunks?
The data structure

• It would be convenient if it worked like:

```c
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 -> chunk1 -> chunk2 -> chunk3 -> ...
  – Explicit List
    • Root -> free chunk 1 -> free chunk 2 -> free chunk 3 -> ...
  – Segregated List
    • Small-malloc root -> free small chunk 1 -> free small chunk 2 -> ...
    • Medium-malloc root -> free medium chunk 1 -> ...
    • Large-malloc root -> free large chunk1 -> ...
Design considerations

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

```c
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?

• Implicit / Explicit / or Seg?
  – Implicit won’t get you very far... too slow.
  – Explicit is a good place to start, and can be turned into a seg-list.
  – Seg-list: what are the thresholds?