U N I V E R S I T Y

CS 240: Programming in C

Lecture 12: Midterm 1 Review
malloc() and free()

Prof. Jeff Turkstra

Announcements

- Midterm Exam 1 next Monday
- No lecture on Wednesday!
- Sample exams and questions available on course website

Ed Discussion Etiquette Reminder

- Ed Discussion…
  - is NOT a place to complain about something you may dislike or disagree with (talk to us instead - We will listen and try to understand the point you're making. You may even change our mind!)
  - is NOT a place to discuss grades or grading in any way. Homework grading issues should be sent via email to jeff@cs.purdue.edu. Quizzes/code standard grading issues should be addressed via Gradescope, and then email if still unresolved.
  - is NOT a place to belittle, mock, or in any other way be disrespectful toward other students or course staff

Ed Discussion Code

- Do not post screen shots. Paste code only.
- Posts including code must be private.
- Do not paste entire functions or programs - only relevant excerpts. The smaller, the better.
- List what you have tried so far in terms of debugging. You must have tried something. Preferably multiple "somethings."
- Ask specific questions. "How do I make this work?" is not a good question.
- We reserve the right to direct you to office hours instead.

C complex numbers

- ISO C99 has support for complex numbers
  - #include <complex.h>
- Three types:
  - float complex
  - double complex
  - long double complex
- Many operations
  - creal(), cimag()
- https://www.gnu.org/software/libc/manual/html_node/Complex-Numbers.html

Practice

- Can you rewrite (most of) your Homework 4 solution using C99 complex.h?

Basic debugger (*nix)

- gdb is the root of all UNIX debuggers
- Very useful in determining where the segmentation fault occurred
  - Not necessarily what caused it
- How to use? Easiest is a 5 step procedure:

```
$ gcc -g file.c -o file # -g flag important!
$ gdb file
(gdb) run  (if problem, will stop at error line)
(gdb) bt (backtrace problem, can provide more info)
(gdb) quit
```

A note about precedence

- It's a little verbose to have to say `(*p).x`
- If the parentheses are omitted, the natural precedence is `*(p.x)` which means something really different
- Wouldn't it be nice if we had an operator that could be used to refer to a field x within a structure pointed to by p???
- It exists! `p->x`

Example

```
#include <stdio.h>

struct coord { int x; int y; };

int main() {
  struct coord c = { 12, 14 };
  struct coord *p = 0;
  p = &c;
  p->x = 4;
  printf("c.x = %d\n", c.x);
  printf("c.y = %d\n", p->y);
  return 0;
}
```

End Exam 1 Material

Midterm 1

- This coming Monday 3/3
  - 8pm – 10pm
  - Sample questions and exam on website
  - Seating chart available soon
    - Please look at it in advance!
  - Short answer and coding

Review

- What follows is a broad overview of topics
- Questions on the exam may cover anything covered during lecture
- You are encouraged to:
  - Review lecture notes and videos
  - Hand write code
    - Quizzes
    - Lecture examples
    - Parts of homeworks
  - Practice writing quickly but clearly

Review

- Compiling and linking:
  - gcc options and usage
  - Object files and executables
- File operations:
  - fopen() / fclose()
  - fprintf() / fscanf()
  - fseek() / ftell()
  - fread() / fwrite()
  - Error checking and error handling

Review (contd)

- Typedef
  - Syntax, usage
- Structures
  - Properties
  - Declaration
  - Definition (what's a definition?)
  - Initialization (what are the properties?)
  - Nested structure declarations
  - Arrays of
  - Passing to and returning from functions
  - Assignment

Review (contd)

- assert()
  - When should you use it?
- Basic string operations:
  - strncpy()
  - strncmp()
  - What do they rely on for correctness?
- Variables
  - Are they global or local? Why?
  - Memory layout
    - Alignment and padding

Review (contd)

- Variables
  - sizeof()
  - Arrays and their initialization
  - Endianness
- bitfields
- unions
- enums
- Bitwise operators

Review (contd)

- Debugging (gdb)
- Basic pointers
  - &, *
  - Pointer arithmetic
  - Pass-by-reference vs. pass-by-value
- Pointers as arrays, arrays as pointers

Be prepared!

- Review lecture notes and videos
- Hand write code
  - Quizzes
  - Lecture examples
  - Parts of homeworks
- Practice writing quickly but clearly

Purdue Trivia

- Orville Redenbacher graduated from Purdue in 1928 with a degree in Agronomy
  - Marched Tuba in the AAMB
  - Also on the track team
  - Worked for the exponent
  - Honorary doctorate in 1988
- Mural in PMU basement includes him

Structures containing pointers

- We mentioned several weeks ago that a structure can contain any definition (except a function)
- A pointer definition can be placed in a structure declaration
- In fact, we can define a pointer to the type of struct that we're presently declaring!

Example of internal pointer…

```
#include <stdio.h>

struct node {
  int val;
  struct node *next;
};

struct node g_node = { 12, NULL };
```


What's the point?

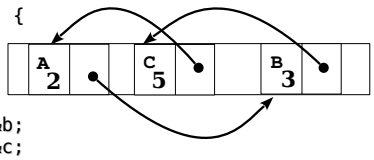
- There's not a lot of use creating a structure that contains a pointer to itself, other than for demonstration
- What if we had several structures?
- What if we set them up to point to each other?
- Better yet, what if we organized them into a list?

Nodes in a ring…

```
struct node a;
struct node b;
struct node c;

void setup() {
  a.val = 2;
  b.val = 3;
  c.val = 5;

  a.next = &b;
  b.next = &c;
  c.next = &a;
}
```

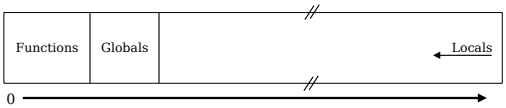


What's the point?

- Still not much use for this except in setting up "state machines"
- We still have the same number of node structures
- What if we could create new node structures dynamically?

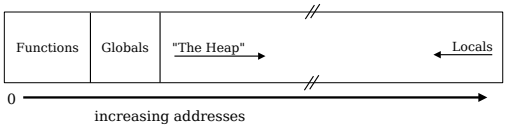
Memory layout *again*

- Here's a macroscopic view of memory for your application. It contains your functions, your global variables and local (inside the function) variables
- Seem's there's a lot of unused memory…



Let's use that memory

- What if we just put something in that memory?
- How would we decide what addresses to use?
- How would we remember what's in use and what's not?
- We want something that will do this for us



malloc() and free()

- The malloc() function is used to allocate a chunk of "The Heap"
  `address malloc(int size);`
- The free() function tells the system that we're done with that chunk:
  `void free(address);`
- How do we tell malloc() what size of "thing" we want to reserve?
  - …use sizeof()!

Example of malloc()

```
#include <stdio.h>
#include <malloc.h>

void get_some_memory() {
  int *int_arr = 0;

  int_arr = malloc(40 * sizeof(int));
  for (int i = 0; i < 40; i++)
    int_arr[i] = 15;
  free(int_arr);
  int_arr = NULL;
}
```


Allocating a struct

```
#include <stdio.h>
#include <malloc.h>

struct node { int val;
              struct node *next };

void alloc_a_struct() {
  struct node *node_ptr = 0;

  node_ptr = malloc(sizeof(struct node));
  node_ptr->val = 42;
  node_ptr->next = 0;
  free(node_ptr);
  node_ptr = 0;
}
```


Things to remember

- When using malloc() always double check that you specify the proper size.
  - Otherwise, chaos will ensue
- Always check the return value from malloc()
- After free(ptr); ptr still points to the same chunk of memory
  - But we no longer have it reserved
    - A subsequent malloc() may reuse it!
- Always say ptr = NULL; after a free(ptr); call
  - That way we do not try to use that memory again

malloc(), calloc()

- malloc(int size) reserves a chunk of memory
  - So… what does that chunk contain?
- calloc(int n, int s) reserves n chunks of memory of size s
  - …and sets all of the bytes to zero
- free(void *ptr) will cancel the reservation for memory from either source
  - What happens to the contents of that memory?
- The CS 240 memory allocator is not so nice

What's wrong with this?

```
#include <stdio.h>

struct node { int val;
              struct node *next };

struct node *alloc_a_struct() {
  struct node my_node = 0;

  my_node.val = 42;
  my_node.next = 0;
  return &my_node;
}
```

What's wrong with this?

```
#include <stdio.h>

struct node { int val;
              struct node *next };

struct node *alloc_a_struct() {
  struct node my_node = 0;

  my_node.val = 42;
  my_node.next = 0;
  return &my_node;
}
```

Never return a pointer to something that is stack-allocated

For next lecture

- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Boiler Up!