CS 240: Programming in C

Lecture 12: More Pointers, malloc() and free()

Announcements

- No class on Wednesday

Pointer example from last time

```
int main() {
  int ctr = 0;
  int *ptr = 0;
  int int4 = 18;
  int int3 = 11;
  int int2 = 10;
  int int1 = 7;

  ptr = &int1;

  for (ctr = 0; ctr < 7; ctr++) {
    printf("Value at address %p: 0x%x (%d)\n",
           ptr, *ptr, *ptr);
    ptr++;
  }

  return 0;
}
```

Pointer example from last time

```
Value at address 0x7ffe41adeb4c: 0x7 (7)
Value at address 0x7ffe41adeb50: 0x1 (1)
Value at address 0x7ffe41adeb54: 0x12 (18)
Value at address 0x7ffe41adeb58: 0xb (11)
Value at address 0x7ffe41adeb5c: 0xa (10)
Value at address 0x7ffe41adeb60: 0x41adeb60 (1101917024)
Value at address 0x7ffe41adeb64: 0x7ffe (32766)
```

- What is happening here?

Pointers are dangerous

- One of the characteristics of a useful computer language is that it should protect the programmer from potential disaster
- C is not like that
- What happens when you have a pointer problem?

When good pointers go bad

```
#include <stdio.h>

int main() {
  int *ptr = 0;
  int array[] = { 5, 6, 7, 8, 9 };

  printf("Before: %d\n", *ptr);
  ptr = &array[2];
  printf("After: %d\n", *ptr);
  return 0;
}
```

When good pointers go bad

```
#include <stdio.h>

int main() {
  int *ptr = 0;
  int array[] = { 5, 6, 7, 8, 9 };

  printf("Before: %d\n", *ptr);
  ptr = &array[2];
  printf("After: %d\n", *ptr);
  return 0;
}
```

```
Segmentation fault (core dumped)
```

How to find the problem?

- You can design your code right in the first place
- You can carefully examine every statement in your program until you understand what happened
- You can insert print statements in your code until you narrow down where the problem is
    - …and probably `fflush(NULL)` a lot
- Or you can *admit defeat* and use a debugger

Basic debugger

- 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
```

More on gdb

- GDB HOWTO on course website
- GDB Tutorial
- [Beej's Quick Guide to GDB]

PURDUE UNIVERSITY

Address-of structures

- You can get the address of anything that stores a value, including a structure

```
struct coord c = { 5, 12 };
struct coord *p = 0;

p = &c;
(*p).x = 1;
(*p).y = (*p).x;
```

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?

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?
- We do! `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;
}
```

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!

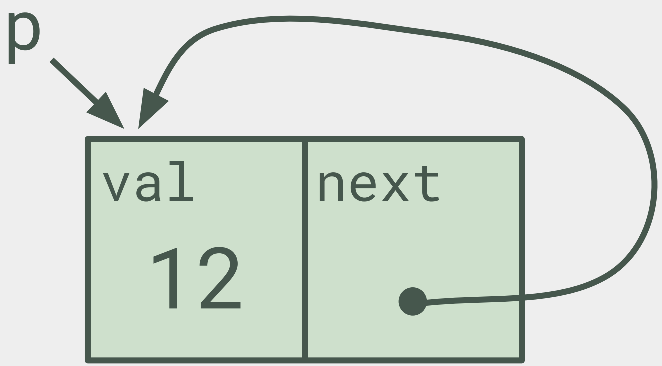
Example of internal pointer

```c
#include <stdio.h>

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

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

Example continued...

```
void subroutine() {
  struct node *p = 0;
  p = &g_node;

  p->next = p;
  printf("%d\n", p->val);
  printf("%d\n", p->next->val);
  printf("%d\n", p->next->next->val);
}
```

Example visualized

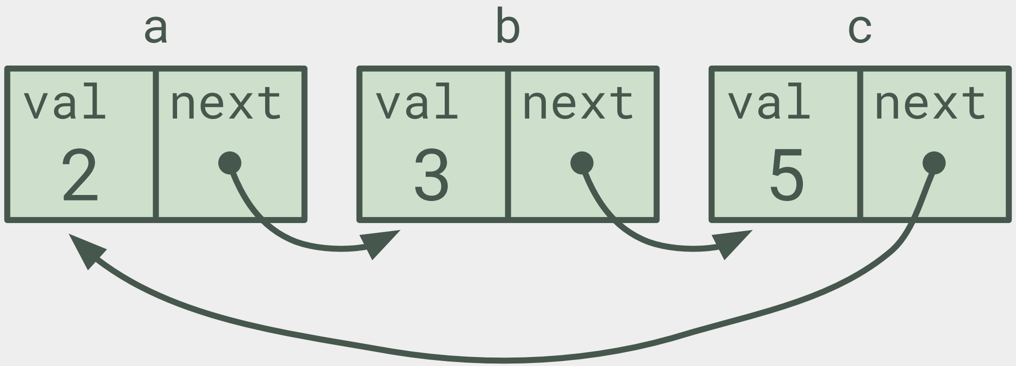


What’s the point?

- 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;
}
```

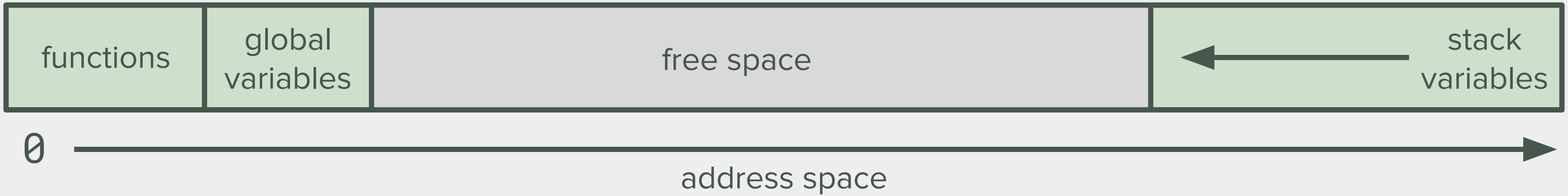


What's the point?

- Still not much use for this...
- We still have the same number of node structures
- What if we don't know the number of nodes we need ahead of time?
- We can create new node structures dynamically

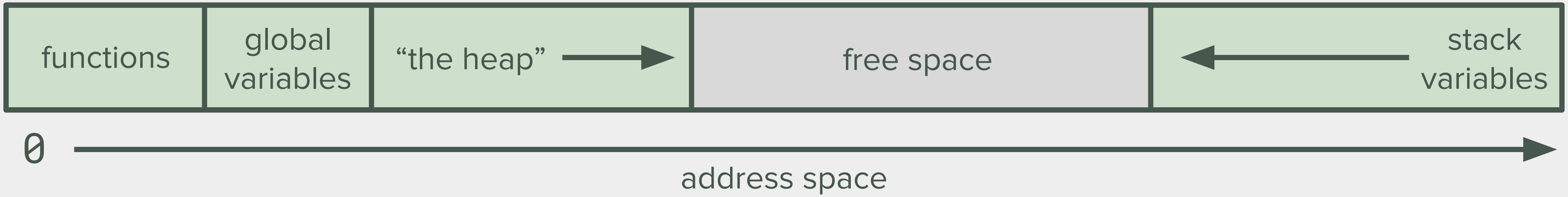
Memory layout revisited

- Here's a macroscopic view of memory for your application.
- Local variables (defined inside functions) appear on the stack
- The stack starts at the highest address and grows downwards
- What about all of that unused memory?



Let’s use that memory

- We can allocate memory in what we call “the heap”
  - Not related to the binary heap data structure
  - A more apt name would be “the pool”
- Unlike the stack, the heap grows upwards
- Use the standard library to manage allocation for us



Stack vs. Heap

- Why not just use the stack?
- The stack is used for function calls
- Variables within a function have a specific lifetime
  - They are destroyed when their function returns
  - i.e., the “stack frame” is “popped” from the stack
- Sometimes we want a variable to live longer than that

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);
```

Example of malloc()

```c
#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
    - 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?

What's wrong with this?

```
#include <stdio.h>

struct node { int val;
              struct node *next; };
struct node *alloc_a_struct() {
  struct node my_node;

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

PURDUE UNIVERSITY

What's wrong with this?

```
#include <stdio.h>

struct node { int val;
              struct node *next; };
struct node *alloc_a_struct() {
  struct node my_node;

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

Never return a pointer to something that is stack-allocated

For next lecture

- Read K&R Chapter 8.7, Beej Chapter 11
- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Slides

- Slides are heavily based on Prof. Turkstra's material from previous semesters.