The Adventures of Malloc and New

Lecture 2: The Logistics of Pointers and Memory Management

Eunsuk Kang and **Jean Yang**

MIT CSAIL

January 20, 2010

Lecture plan

1. Recap from last lecture.
2. Introduction to pointers.
3. Memory management on the stack and heap.
4. Data structures: arrays and structs.
5. Linked list example.
6. Tools and tips.

Recall from last time. . .

- C is an **imperative** language that is typically **compiled** and requires **manual memory management**.
- We can think of **stack** and **heap** memory as an array.
- We access this memory using **pointers**.
- We use `malloc` and `free` to help us manage memory.

Questions I'll answer in the next two lectures

Today

- What memory abstractions does C provide?
- What exactly is the distinction between stack and heap?
- How do I use pointers to access memory locations?
- How do I allocate and free memory on the heap?
- How should I use GDB and Valgrind?

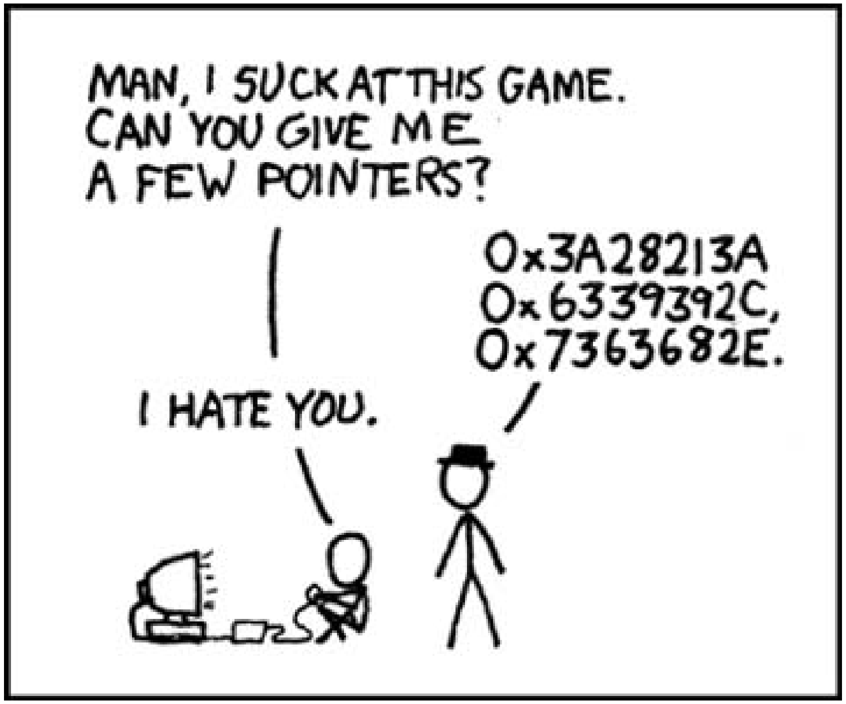
Tomorrow

- How does the compiler actually work?
- What was that thing you did with == and =?

Questions I'll answer right now

- What is the difference between a compile-time (static) error and a (dynamic) run-time error?
- Why are we not using an IDE?
- What are some good ways to edit my C files?
- How do I use Valgrind if I run Windows?
- Other questions?

Accessing memory in C: pointers



Courtesy of xkcd.com. Comic is available here: http://xkcd.com/318/

Pointers are **memory addresses**.

Every variable has a memory address.

It's all about tracking which addresses are still in use.

Pointer syntax

Symbol	Pronunciation	Example use
&	"Take the address of"	&x
$*$	"Take the value of"	$*$p

Somewhat confusing! $*$ is also used to denote a pointer type (e.g., int$*$ x, which is pronunced "int pointer").

Practicing pronunciation

```
int* xp, yp;
int z;
```

Declare int pointers xp and yp and int z.

```
xp = &z;
```

Set xp equal to the address of z.

```
yp = xp;
```

Set yp equal to xp. Now yp and xp “point to” the same value.

```
*xp = *yp;
```

Set the value at address xp equal to the value at address xp. Do xp and yp “point to” the same value?

An example **without** pointers

What will this print?

```
void process_var(int x) {
  x = 5;
}

void fun() {

  process_var(x);
  printf("%d\n", x);
}
```

An example **with** pointers

What will this print?

```
/* Passing a pointer as an argument. */
void process_var(int* xp) {
  *xp = 5;    /* The value of xp... */
}

void fun() {

  process_var(&x);  /* The address of x... */
  printf("%d\n", x);
}
```

Revisiting C memory: stack vs. heap

- The C compiler lays out memory corresponding to functions (arguments, variables) on the *stack*.
- C allows the programmer to allocate additional memory on the *heap*.

	Stack	**Heap**
Memory is allocated	Upon entering function	With `malloc`
Memory is deallocated	Upon function return	With `free`
Addresses are assigned	Statically	Dynamically

Managing memory

Conceptually

Keep track of what memory belongs to your program, making sure

- all addresses you give to other functions are valid for those functions and
- you deallocate memory you are not using while you still know about it.

While programming

- Use `malloc` to allocate memory on the heap if you will need it after the current function returns.
- Use `free` to free pointers *before* you reassign them and lose the pointer.

Dynamic allocation and deallocation

Allocation

`malloc` is a C standard library function that finds a chunk of free memory of the desired size and returns a pointer to it.

```
int* p = malloc(sizeof(int));
```

Deallocation

`free` marks the memory associated with a specific address as no longer in use. (It keeps track of how much memory was associated with that address!)

```
free(p);
```

Asking for memory: arrays

Statically allocated arrays

```
int arr[5];
arr[1] = 6;
```

Dynamically allocated arrays

```
int* arr;
arr = malloc(sizeof(int) * 5);
arr[1] = 5;
```

Asking for memory: structs

Defining a struct

```
struct pair {

  int second;
};
```

Statically allocated structs

```
struct pair p1;
p1.first = 0;    /* Note the "." syntax. */
```

Dynamically allocated structs

```
struct pair* pp = malloc(sizeof(struct pair));
(*pp).first = 2;
pp->second = 3;       /* Note the -> syntactic sugar. */
```

Some other things

typedef

```
typedef struct pair pair_t;
pair_t p;
```

```
typedef struct pair {
  int first;
  int second;
} pair_t;
```

enum

```
enum ANSWER { yes, no, maybe };
```

```
typedef enum BOOL { true = 0, false = 1 } bool_t;
bool_t b = true;
```

Review: singly linked list

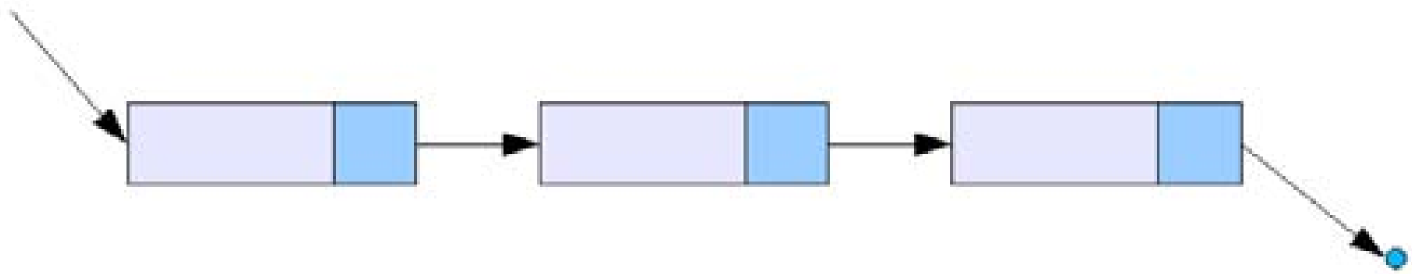


Figure: Schematic singly linked list.

- Made up of nodes.
- Each node points to the next node.
- We have a pointer to the head.
- The last node points to nothing.

Node structure

```
struct node {
  int val;
  struct node* next;
};
typedef struct node node_t;
```

Doing this is the same as:

```
typedef struct node {
  int val;
  struct node* next;
} node_t;
```

Creating nodes on the heap

```c
/* Returns a new node with the given value. */
node_t* make_node(int val) {
  /* Using malloc. */
  node_t* new_node = malloc(sizeof(node_t));
  new_node->val = val;
  new_node->next = NULL;
  return new_node;
}
```

Inserting at the head of a linked list

```
node_t* insert_val(int val, node_t* cur_head) {
  node_t* new_node = make_node(val);
  new_node->next = cur_head;
  return new_node;
}
```

To think about. How would we insert in order?

Deleting a value

```
node_t* delete_val(int val, node_t* cur_head, int*
    succeeded) {
  *succeeded = 0;

  if (cur_head == NULL) {
    return NULL;
  } else if (cur_head->val == val) {
    node_t* new_head = cur_head->next;
    free(cur_head);
    *succeeded = 1;
    return new_head;
  } else {

  }
}
```

To think about. How could we have written this without recursion? Why might we want to do it that way?

Memory errors

How you can produce memory errors

- Program accesses memory it shouldn't (not yet allocated, not yet freed, past end of heap block, inaccessible parts of the stack).
- Dangerous use of unitialized values.
- Memory leaks.
- Bad frees.

Manifestations of memory errors

- "Junk" values.
- Segmentation fault–program crashes.

Tools for programming with memory

GDB: The GNU Project Debugger

Can do four main things:

- Start your program, specifying things that might affect behavior.
- Make your program stop on breakpoints.
- Examine variables etc. at program points.
- Change things in your program while debugging.

Useful for debugging crash dumps.

Valgrind: a memory profiling tool

- A GPL system for **debugging** and **profiling** Linux programs.
- Tool suite includes **memcheck**, cachegrind, callgrind, massif, and helgrind.

Valgrind: our bug-free program

Running Memcheck on our singly list program with `valgrind ./sll`, where `sll` is our binary.

```
==24756== Memcheck, a memory error detector.
==24756== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==24756== Using LibVEX rev 1804, a library for dynamic binary translation.
==24756== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==24756== Using valgrind-3.3.0-Debian, a dynamic binary instrumentation framework.
==24756== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==24756== For more details, rerun with: -v
==24756==
0
1 0
1 1 0
1 0
0
2 1 0
==24756==
==24756== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 8 from 1)
==24756== malloc/free: in use at exit: 0 bytes in 0 blocks.
==24756== malloc/free: 5 allocs, 5 frees, 80 bytes allocated.
==24756== For counts of detected errors, rerun with: -v
==24756== All heap blocks were freed -- no leaks are possible.
```

Valgrind: a buggy program that doesn't free memory

If we run Memcheck on the same program without freeing the memory, the tool informs us that we leak memory.

```
==25520==
==25520== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 8 from 1)
==25520== malloc/free: in use at exit: 48 bytes in 3 blocks.
==25520== malloc/free: 5 allocs, 2 frees, 80 bytes allocated.
==25520== For counts of detected errors, rerun with: -v
==25520== searching for pointers to 3 not-freed blocks.
==25520== checked 78,552 bytes.
==25520==
==25520== LEAK SUMMARY:
==25520==    definitely lost: 48 bytes in 3 blocks.
==25520==      possibly lost: 0 bytes in 0 blocks.
==25520==    still reachable: 0 bytes in 0 blocks.
==25520==         suppressed: 0 bytes in 0 blocks.
==25520== Rerun with --leak-check=full to see details of leaked memory.
```

Some things to remember for the homework

- C functions require variables to be declared/assigned at the top of functions.
- You will need to have `malloc` allocate functions that are used beyond the current function.
- You will need to call `free` on a pointer you are done using before you reassign the pointer.
- Accessing memory at the `NULL` location will result in a segmentation fault.
- Accessing memory at an invalid location *may or may not* result in a segmentation fault.

Sneak preview of tomorrow

- A closer look at how the compiler works (with war stories).
- Fancier memory examples.
- Pitfalls and style guidelines.
- Requests? E-mail us!

Review for homework: binary search trees

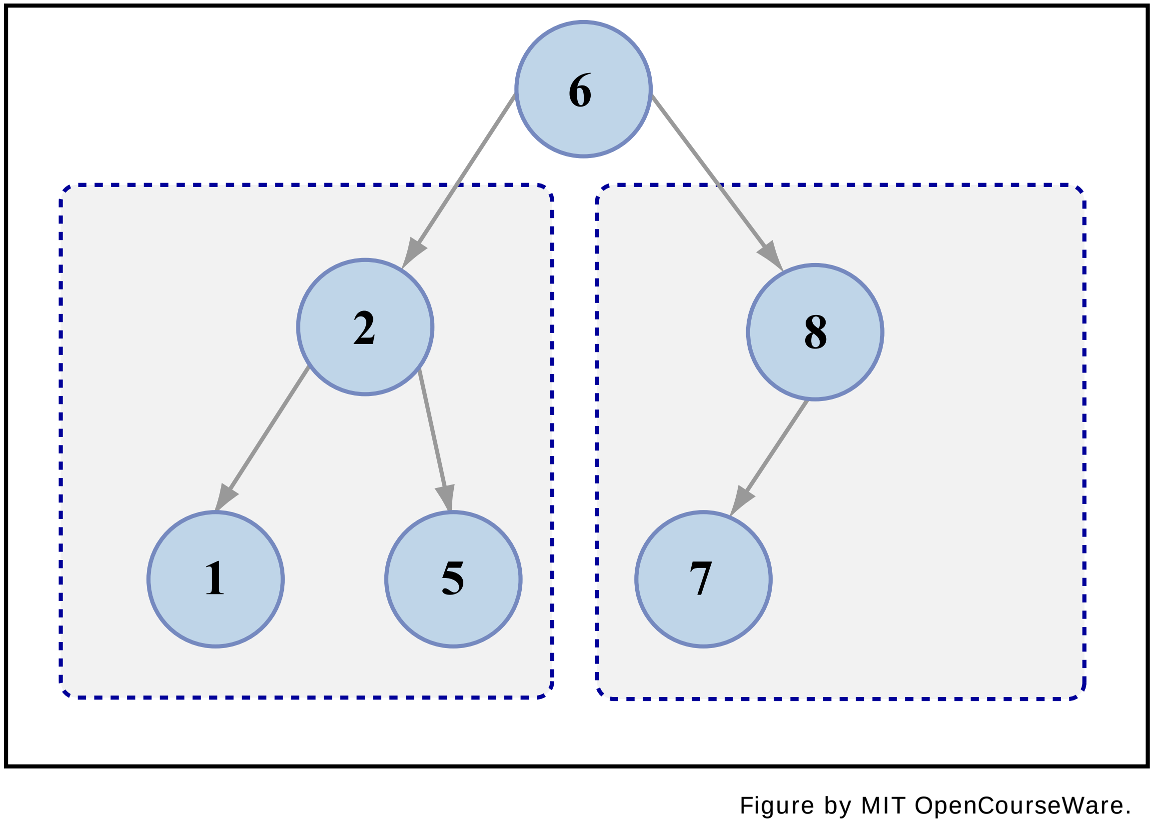


Figure: Example binary search tree.

- Each node has a pointer to the left and right child.
- Smaller values go to the left; larger values go to the right.

Until tomorrow. . .

Homework (due tomorrow)

- Implement `insert`, `find_val`, and `delete_tree` based on the partial binary search tree code we provide.
- More details and code on the course website.

Questions?

- The course staff will be available after class.

6.088 Introduction to C Memory Management and C++ Object-Oriented Programming
January IAP 2010