This is Week 4

Jason Hirschhorn
Fall, 2011
Agenda

• Resources + Announcements
• Review
  • Problem Set 2
• Memory, Part 1
  • Hexadecimal numbers
  • Stack
• Pointers
• Memory, Part 2
  • Heap
  • Dynamic memory allocation
  • Arrays
• Merge Sort
Resources + Announcements

• Problem Set 4 Walkthrough (Sun, 7pm, NW B103) – https://www.cs50.net/psets/
• Office Hours – https://www.cs50.net/ohs/
• Lecture videos, slides, source code, scribe notes – https://www.cs50.net/lectures/
• Problem Set feedback and scores
  • pset1, pset2 – all ready sent out!
  • pset3 – Thursday
• Quiz 0 (Wed, 10/12) – https://www.cs50.net/quizzes/
Review
pset2 - Correctness

- Make sure your code compiles
  - Run a fresh make of each program before you submit
- Make sure your code works properly
  - Compare its results to the staff solution’s results
  - Check corner cases
pset2 - Style

- Block comments at the beginning of each file

/*
 * caesar.c
 *
 * Computer Science 50
 * Jason Hirschhorn
 *
 * Encrypts a phrase using a Caesar cipher.
 */
int main(int argc, char *argv[]) {
    // validates user input
    if(argc != 2)
        return 1;

    // creates variables to store name and length
    char *name = argv[1];
    int length = strlen(name);

    // ensures each character is a valid letter
    for(int i = 0; i < length; i++)
    {
        if(!isalpha(name[i])
            return 1;
    }
}
Memory, Part 1
Memory

- Code and data for your program are stored in random-access memory (RAM)
- Memory is a huge array of 1 byte (8 bits) blocks
- Each block has a numerical address
- We use hexadecimal numbers to concisely represent the memory addresses of these blocks
Hexadecimal Numbers

- Hexadecimal = numbers are in base 16
  - 0 to 9 then A to F

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>
Hexadecimal Numbers

• Every set of 4 bits (a “nibble,” or half a “byte”) can be represented by 1 hex digit
• To signal that we’re using hexadecimal, we start with “0x”
Stack

• A part of memory
  • Need to store something? Put it on top
  • Done with something? Take it off

• Each function that’s called gets its own block
  • “Frame”
  • It puts the variables it creates in its frame

• When a function returns, its frame becomes inaccessible

```c
function1 creates
int num = 5;
string name = “cs50”;
char x = ‘a’;
```

```
function2()

function1()

name
num

main()
```
Pointers
Pointers

- Data stored in memory has both a *value* and an *address*
- A pointer is a special type of variable
  - Its value *is* an address
- How big is a pointer?
  - Every memory address is 4 bytes (32 bits)
  - So a pointer is also 4 bytes
  - No matter what type it is

Quick Quiz

- What are the following hexadecimal numbers?

<table>
<thead>
<tr>
<th>8s</th>
<th>4s</th>
<th>2s</th>
<th>1s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Using Pointers

- `<type>* <variable name>` declares a pointer
  - It will hold an address, not a value
- `&<variable name>` gets the address of a variable
- `*<variable name>` goes to the address stored in the variable and gets its value

```c
int x = 5;
int* y = &x;
int copy = *y;
```
Examples

```c
char m = 'A';
char* n = &m;
```

- What is m? What is n?
- If I want char p to be 'A', what two things can I set it equal to?

```c
int x = 13;
int* y = &x;
*y = (*y) * 2;
```

- What is x?
- What is y?

```c
int a = 3, b = 4, c = 5;
int *pa = &a, *pb = &b, *pc = &c;
```

- What happens after each statement?

```c
a = b * c;
a *= c;
b = *pa;
pc = pa;
*pb = b * c;
c = (*pa) * (*pc);
```
Practice Problems

• address.c
  • Concepts to practice – pointers
• pointers.c
  • Concepts to practice – pointers (these take a ton of practice)
Man, I suck at this game. Can you give me a few pointers?

I hate you.

0x3A28213A
0x6339392C,
0x7363682E.
Memory, Part 2
Heap

- Sometimes we want our variables to stay in memory after a function returns
  - Remember, local variables are stored in the stack
  - Stack frames go away after a function returns
- So, we can store these variables on the heap
  - Another part of memory
  - Separate from the stack
- Data on the heap won’t get overwritten
  - We get to choose when it gets created and destroyed
- However, we have to explicitly reserve this memory
Dynamic Memory Allocation

- Requesting memory on the fly
- `void* malloc(int <number of bytes>)`
  - Reserves a block of memory on the heap
  - Returns the address of this block
- `sizeof(<data type>)`
  - Returns the number of bytes a given type occupies
- `void free(void* <name of pointer>)`
  - Frees up the reserved memory

Quick Quiz
- How many bytes is an int? A char? An int*? A char*?
Dynamic Memory Allocation

// reserves enough space to store 10 chars
char* x = malloc(sizeof(char) * 10);

// frees this reserved block
free(x);
Stack vs. Heap

- Contains global variables
- Dynamically allocated memory

- Contains local variables
- Function calls create new frames
Heap.c

• Concepts to practice – pointers, dynamic memory allocation, the heap
• When using malloc, always remember to...
  • Check whether it returns null
  • Free the allocated memory exactly once
• Compare to stack.c
**Pointer Arithmetic**

- Adding an integer n to a pointer shifts the pointer over
- Shift = n * sizeof(<type of pointer>) bytes
- The address of x is 0x04
  ```
  int x;
  int* y = &x; // y has the value 0x04
  y += 1; // y has the value 0x08
  ```
- Why?

**Quick Quiz**

- The address of x is 0xAA
  ```
  char x;
  char* y = &x; // what is the value of y?
  y += 1; // what about now?
  ```
Arrays

- Arrays are pointers!

char array[4];

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>O</th>
<th>P</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>*array</td>
<td>*(array + 1)</td>
<td>*(array + 2)</td>
<td>*(array + 3)</td>
<td></td>
</tr>
</tbody>
</table>
Array.c

- Concepts to practice – command line arguments, dynamic memory allocation, pointers, pointer arithmetic

```c
// counts up to a given number
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
    // ensure user enters an integer greater than 0
    // use malloc to initialize a new array of size = argv[1]
    /* use pointer arithmetic to store integers
       starting at 0 */
    // use index notation to add 1 to each element
    // print each element of final array
}
```
Merge Sort
Merge Sort

• Yet another sorting algorithm
  • Like bubble sort and selection sort, but way better!

Method

• If the list is length 0 or 1, it is already sorted
• Else divide the unsorted list into two halves
  • Sort each half
  • Merge the two halves into one sorted list
    • Compare first element of each half
    • Put lowest overall in front of new sorted list
    • Keep moving down each half until one runs out
Merge Sort

Unsorted list

Base cases

Sorted list
Running Time

Big O
- $O(n \log n)$
- Breaking a list in half and rebuilding it = $O(\log n)$
- Sorting each half = $O(n)$
That was Week 4

http://xkcd.com/179/