week4

Tommy MacWilliam

tmacwilliam@cs50.net

October 3, 2011

Announcements

- pset2: returned!
- pset3: Friday
- Quiz 0: 10/12
- `https://cs50.net/lectures`
- `http://cs50.net/ohs`

Today

- pset2
- stack
- pointers
- heap
- arrays, fo’ realz
- recursion
- merge sort

Hacker Tip of the Week

- learn Vim!
  - http://blog.interlinked.org/tutorials/vim_tutorial.html
  - http://yannesposito.com/Scratch/en/blog/Learn-Vim-Progressively/
  - http://stackoverflow.com/questions/1218390/what-is-your-most-productive-shortcut-with-vim/1220118

pset2 Correctness

- make sure your code compiles!
  - make everything before you submit

pset2 Design

- you don't need to
  - `(char)((int)c + 5)`
- C handles casting between chars and ints for you!

pset2 Design

- all of these are legal
  - `int a = ’a’;`
  - `char b = 67`
  - `int c = ’N’ - ’a’ % 26`
  - `printf(“%d\n”, ’a’);`
- using `’A’` instead of 65 makes your code much easier to read!

Memory

- code and data for your program are stored in random-access memory (RAM)
- memory is essentially a huge array of 1 byte (8 bits) blocks
- each block has a numerical address

Stack

- big pile of data
- need to store something? put it on top of the pile
- don’t need something anymore? take it off the top of the pile
    - don’t take anything off the bottom of the pile :(

Variables

- local variables will be stored on the stack
  - `int x = 5` occupies 4 bytes of memory and stores a variable called `x` on the stack
- as more variables are created, the stack grows to accommodate them

Functions

- each function call gets its own block on the stack called a stack frame
  - all variables created by the function stored in that stack frame
- recursion: each call gets a new stack frame
  - too many calls? stack overflow!

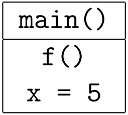
Functions

```
int main() {
    f();
    g();
}
void f() {
    int x = 5;
    g();
}
void g() {
    int y = 3;
}
```

Functions

`main()`

Functions

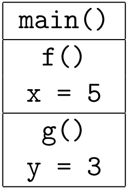


main()

f()

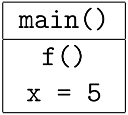
x = 5

Functions



```
main()
f()
x = 5
g()
y = 3
```

Functions



main()

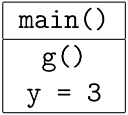
f()

x = 5

Functions

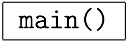
```
main()
```

Functions



```
main()
g()
y = 3
```

Functions



`main()`

Functions

- when a function returns, its stack frame becomes inaccessible
  - will probably get overwritten when another function gets called
  - stack is a pile: if something gets taken off, when we put something back on, it will occupy that space

Pointers

- rather than storing the value of a variable, we can store its address instead
  - each block of memory has a numerical address
  - each address is 32 bits (4 bytes)
- pointer to any type is the same size: 4 bytes (the size of the address)

Pointer Notation

- `int*` declares a pointer to an `int`, `char*` a pointer to a `char`, etc.
- `&` gets the address of a variable
- to store the address of the variable `x` in `y`:

```
int x = 5;
int* y = &x;
```

Pointers

- example time!
  - `address.c`

Pointer Notation

- `&`: get address
- `*`: go to address
- set the value stored at the address of `x` to `10`:

```
int* y = &x;
*y = 10;
```

More Pointers

- example time!
  - `increment.c, pointers.c`

Passing Arguments

- pass by value: pass a non-pointer (aka value) to a function
  - value CANNOT be modified
- pass by reference: pass a pointer (aka address) to a function
  - value CAN be modified

Pointers and the Stack

- example time!
  - `local.c`

Pointers and the Stack

- wtf?
- `f()` created a new stack frame and returns the **address** of `a`
  - `a` must be in the stack frame of `f()`
- `g()` then **overwrites** that stack frame, changing the value of `x`!
  - `b` occupies the same address that `a` did, and `x` points to that address

The Heap

- what if we don't want that?
    - and we probably don't...
- we need variables to stay in memory even after function returns

malloc/free

- can also store variables on the heap
  - separate from the stack, so data won't be randomly overwritten
- YOU have total control over when memory in the heap gets created/destroyed
- `malloc`: get memory from the heap
- `free`: give memory back to the heap

malloc/free

- `malloc` takes one argument: the number of bytes to get from the heap
- `sizeof` tells you the number of bytes a type occupies
    - `int* a = malloc(sizeof(int))`
- make sure you `free()` **EVERYTHING** you `malloc()`

Dynamic Memory Allocation

- using `malloc` is called dynamic memory allocation
  - memory is requested on the fly, as your program needs it
- on the stack, the compiler knows exactly how much memory you need before your program even runs

Using the heap

- to fix `local.c`, store variables on the heap instead of the stack
    - now, they won't be overwritten by other function calls

The Heap

- example time!
  - `heap.c`

Pointers and Arrays

- remember, a `string` is just a `char*` is just an array of `char`s
- in memory, the blocks of an array are stored next to each other
  - contiguous
- so, an array is actually just a pointer to the first element in the array

Creating Arrays

- create an array of 4 `int`s on the stack
    - `int a[4];`
- on the heap:
    - `int* a = malloc(4 * sizeof(int))`

Accessing Arrays

- accessing the elements of `a`
  - `a[0] == *a`
  - `a[1] == *(a + 1)`
- 2nd element is one more than the address of the first element
  - by “one”, C knows you mean `1 * sizeof(int)`

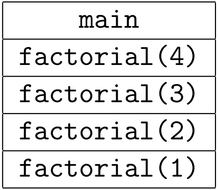
Pointers and Arrays

- example time!
  - `array.c`

Recursion

- basic idea: function calls itself repeatedly
- base case: when function should stop calling itself
- recursive case: how the function should call itself
  - probably with different arguments!

Recursion and the Stack



```
main
factorial(4)
factorial(3)
factorial(2)
factorial(1)
```

Merge Sort

- implementation: divide list into two smaller lists
  - repeat until list cannot be divided any further
  - sort each smaller list
  - merge the results
- runtime: $O(n \log n)$, $\Omega(n \log n)$

Merge Sort

- naturally leads itself to recursion
  - recursively break up list until it can't be broken up any more
  - then, iteratively merge the two lists
- code is super elegant
  - wait, this really works?

Merge Sort

5 0 1 6 4

Merge Sort

5 0 1 6 4

Merge Sort

5 0 1 6 4

Merge Sort

0 5 1 4 6

Merge Sort

0 1 4 5 6

Merge Sort

- example time!
  - `mergesort.rb`
  - new language, Ruby!
    - great for final projects!
    - syntax looks like psuedocode, easy to read

Merge Sort

- `http://www.smbc-comics.com/index.php?db=comics&id=1989`

Practice Problems

- using the heap, write a function that returns an array of size *n*
  - now, try just using the stack
- iterate through an array without using `[]`

Feedback

- how was section today?
  - `http://tommymacwilliam.com/cs50/feedback`