CS 222: Pointers and Manual Memory Management

Chris Kauffman

Week 4-1

Logistics

Reading

- Ch 8 (pointers)
- Review 6-7 as well

Exam 1 Back Today

Get it in class or during office hours later

HW 3 due tonight

Any questions?

HW 4 up tomorrow,

- Due next week
- More advanced struct, arrays
- Multidimensional arrays

Goals

- Exam 1 Feedback
- String Practice
- Pointers

Exam 1 Stats

Range	Count
90 - 100	12
80 - 89	16
70 - 79	3
60 - 69	3
50 - 59	4
40 - 49	1

Stat	Val	R2014
Count	39	33
Min	45.00	43.64
Max	98.33	100.00
Range	53.33	56.34
Average	81.84	80.67
Median	86.67	83.64
StdDev	13.48	16.33

Strings as Input

`scanf` can read strings with `%s`

- Must have a character array of sufficient size
- Don't use `&`

```
{
  char buffer[1024];
  scanf("%s",buf);
}
```

Q: How would determine if the string read is the string `Millenium`?

Oh, the bits you'll smash

`scantest.c`: Let's make some trouble

http://stackoverflow.com/questions/1345670/stack-smashing-detected

Practice Program

`wordguess.c`

- A mystery word called `answer`
- Repeated prompting to user for guess word
- Check if guess word is correct
- End game is guess is correct
- Otherwise, reveal progressive characters of `answer`

Write this program for me

Required Elements

- Read a string using `scanf()`
- `strcmp()` is very useful

```
if(strcmp(answer, guess) == 0) /* is it the right word?
```

- Loop, tracks number guesses
- Print single characters with `printf()`

Other Cool Functions in `string.h`

See `stringlib.c`

- Length : `strlen()`
  - myint $\leftarrow$ length(str)
  - `int l = strlen(str);`
- Copy : `strcpy()`
  - str1 $\leftarrow$ str2
  - `strcpy(str1, str2);`
- Concatenation : `strcat()`
  - str1 $\leftarrow$ str1 str2
  - `strcat(str1, str2);`

A few Character Functions

In `ctype.h`: can be useful for checking conditions

```
int isupper(char c);
int islower(char c);
int isspace(char c);
...

int toupper(int c);
int tolower(int c);
...
```

Not really needed for HW: just check specifically for characters with ==.

Relation of `*a` and `a[]`

What is `a` versus what is `c`?

```
int a[10];
char c[5];
```

- A memory address
- Access `a[4]` means `a + 4*sizeof(int)`
- Access `c[4]` means `c + 4*sizeof(char)`
- Second half explicitly deal with memory locations
  - `int *ap;` a pointer to memory which contains `int`s
  - `char *cp;` a pointer to memory which contains `char`s

One common error: Passing array args

Some function on arrays

```
double arrfunc(int [] arr, int length);
double otherfunc(int arr[], int length);
```

Call function with an array

```
int ia[5] = {2, 2, 0, 3, 0};
double ans = arrfunc(ia, 5);
```

Call with *bare name only*

- No square braces

  `WRONG: arrfunc(ia[], 5);`

- No size indication in square braces

  `WRONG: arrfunc(ia[5], 5);`

Arrays are a Fixed Memory Address

What *are* `a` and `c`?

```
int a[10];
char c[5];
```

- A memory address
- Access `a[4]` means `a + 4*sizeof(int)`
- Access `c[4]` means `c + 4*sizeof(char)`
- More on `sizeof` after the break

Pointers

- A memory address
- A fundamental type in C, like `int`,`char`,`double`
- Point at a data type like `int`,`char`,`double`
- Can also be a void pointer - generic pointer
- *Very unfortunate homonyms*
  - `void fun(void);` takes no args, returns nothing
  - `void` pointer *can* actually point at stuff
  - `NULL` pointer points at nothing
  - Null character `\0` ends strings

Question: Pointers are a data type. What should we discuss next?

Pointer Basics

Define Done for you

Declare Use a * in front of another type

```
int *intptr;   // I point at ints
double *doubs; // I point at doubles
char *chars;   // I point at chars
ze_struct *zs; // Well, you get the point
```

Access `x = intptr;` *What type should x have?*

Assign `intptr = x;`

Wait a minute. . . . what aren't you telling me?

The Interesting Operations

`&` : Address Operator

- Applicable to any variable
- Produces memory address of the variable
- Results in *a pointer*

```
int i = 2;
int *ptr = &i; // Point at i
int j = 3;
ptr = &j;      // Point at j
```

- LHS must be a pointer
- RHS should be some variable

`*` : Dereference Operator

- Applicable to pointers
- Produce contents of pointer
- Results in *pointed at type*
- `*ptr` on RHS: access value pointed at variable `i`

```
int i = 2;
int *ptr = &i;
int j = *ptr;
```

- `*ptr` on LHS: allows assignment to pointed at variable `i`

```
*ptr = 10;
if(i == 10){ printf("wow!"); }
```

See `simplepointers.c`

* and & are Inverse Operations

- $f()$ and $g()$ are inverse operations if

$$x = f(g(x)) = g(f(x))$$

- $f(x) = x + 1$ and $g(x) = x - 1$ are inverse functions
- Derivative and Integral are inverse operations

$$f \approx Deriv(Integ(f)) \approx Integ(Deriv(f))$$

- `*var` and `&var` are inverse operations

```
var == *(&var)
var == &(*var)
```

- For this to work, what type must `var` have?

The Pointer Play

Hot seat thespians act out the drama of C
Script in `pointerplay.c`

First Real Uses

Pointers as function arguments are interesting

- `nonlocal_set.c`

Remember how we can't return more than one thing from a function?

- Now you can: `multiplereturns.c`

Exercise

Write a function that swaps two integers

- `swap_ints` takes two integer pointers
- What does its prototype look like?
- How is the swap accomplished?

Relation of *ap and a[]

What *are* a and c?

```
int a[10];
char c[5];
```

- A memory address
- Square brace syntax is offset from that location
  - Access `a[4]` means `a + 4*sizeof(int)`
  - Access `c[4]` means `c + 4*sizeof(char)`

How about for pointers?

```
int *ap;
char *cp;
```

- Also a memory address
- Pointers also allow [] syntax
  - Access `ap[4]` means `a + 4*sizeof(int)`
  - Access `cp[4]` means `c + 4*sizeof(char)`

See `arrayVptr.c`

Differences of `*ap` and `a[]`

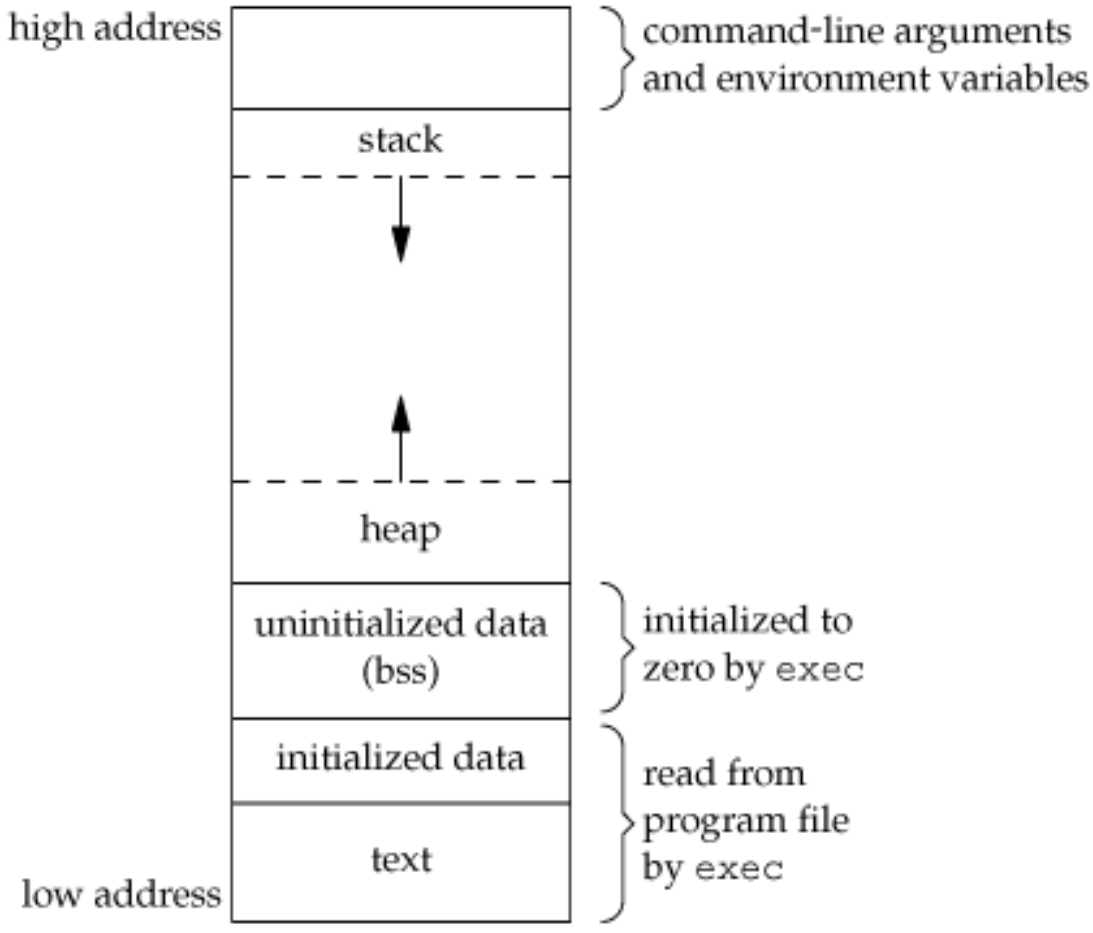
Array `a[]` A fixed memory location (stack or global memory)

- Can't move the array around
- Can change elements of the array: `a[1] = x;`
- Usually points at more than one thing

Pointer `*ap` Only points at something else

- Can change where ap points: `ap = &x;`
- change data at location: `*ap = x;`
- change data at offset: `ap[1] = x;`
- May point at 1 thing or a whole array of things
- Must use context to tell...

Remember Memory Layout



Stuff we've had so far is on the stack

What about that other part?

Stack and Heap

Stack

- Grows when functions get called, shrinks when functions finish
- Compiler knows how much to shrink and grow stack
  - For this function I need 2 `ints` and an array of 10 `doubles`
  - 2*4 + 10*8 = 88 bytes
- Stack space is there for you automatically

Heap

- For memory with size not known at compile time
- Used for *run-time allocation*
  - Read `n` from the user
  - Allocate space for `n` integers
- Programmer (you) must manually manage heap space
  - With help from libraries

`malloc` and `free`

`malloc(n)` Allocate `n` bytes somewhere on the heap

- used as `void *p = malloc(n);`
- `p` now points at memory on heap which can be used
- Allocation may *fail* - not enough memory

`free(p)` Deallocate memory pointed to by `p`

- Memory available for further calls from `malloc`
- Gives errors if `p` doesn't point to `malloc`'d memory

Allocating Useful Stuff

Prototype: `void *malloc(size_t size);`

- `size_t` is an integer-like value (probably `long`, 64-bit integer)
- Usually want `int *`, `double *`, `planet_t *`, not `void *`
- Need to figure out how many bytes required
- Use two C features for this: sizeof and casting

Casting

Force conversion of one type to another

Numerical
```
int i = (int) 45.3 * 0.4432;
```

Pointer
```
char *str = (char *) malloc(100);
```

Pointer
```
planet_t *p = (planet_t *) malloc(100);
```

Gross
```
double d = (double) 'H';
```

Old School
```
int ip = (int) &i;
```

Bad Bad
```
double q = (double) str;
```

Compiler, I'm removing the safety net because it's in the way.

`sizeof()`

Like a function that returns number of bytes for a type

- `sizeof(int)` is # bytes an integer uses
- `sizeof(planet_t)` is # bytes an `planet_t` uses
- `sizeof.c`

malloc useful stuff

See `malloc.c`

- `chars/string`

```
char *str = (char *) malloc(sizeof(char)*100);
```

- `doubles`

```
double *arr = (double *) malloc(sizeof(double)*100);
```

- `planet_t`

```
planet_t *p = (planet_t *) malloc(sizeof(planet_t)*100);
```

Fun things to try

See how much memory you can get: `malloc_madness.c`

`free`

Keep using `malloc` and eventually it wil fail: no memory left

- Use `free` to deallocate
- Important for long-running programs
- Memory leak: `malloc`, lose pointer, can't free, program gets bloated