CS 354 - Machine Organization & Programming
Tuesday, September 17, 2019

Project p1 (3%): DUE at 10 pm on Monday, September 23rd

Project p2A will be assigned tomorrow.

Homework hw1 will be assigned tomorrow.

Last Time

Practice Pointers
Recall 1D Arrays
1D Arrays and Pointers
Passing Addresses

Today

Passing Addresses (from last time)
1D Arrays on the Heap
Pointer Caveats
Meet C Strings
Meet `string.h`

Next Time

2D Arrays and Pointers

Read:

K&R Ch. 5.7: Multi-dimensional Arrays
K&R Ch. 5.8: Initialization of Pointer Arrays
K&R Ch. 5.9: Pointers vs. Multi-dimensional Arrays
K&R Ch. 5.10: Command-line Arguments

See: Piazza post for web alternatives to K&R readings

1D Arrays on the Heap

What? Memory segments used by a program include

STACK	vs.	*HEAP*
static allocations during compile time		

Why? Heap memory enables

-
-

How?

`malloc(size_in_bytes)`:

`free(pointer)`:

`sizeof(operand)`:

→ For IA-32, what value is returned by `sizeof(double)`? `sizeof(char)`? `sizeof(int)`?

→ Write the code to dynamically allocate an integer array named `a` having 5 elements.

```
void someFunction(){
```

→ Draw a memory diagram showing array `a`.

→ Write the code that gives the element at indexes 0, 1 and 2 a values of 0, 11 and 22 by using pointer dereferencing, indexing, and address arithmetic respectively.

→ Write the code that uses a pointer named `p` to give the element at index 3 a value of 33.

→ Write the code that frees array `a`'s heap memory.

Pointer Caveats

- *Don't dereference uninitialized or NULL pointers!*

```
int *p;
*p = 11;
```

```
int *q = NULL;
*q = 11;
```

- *Don't dereference freed pointers!*

```
int *p = malloc(sizeof(int));
int *q = p;
. . .
free(p);
. . .
*q = 11;
```

dangling pointer:

- *Watch out for heap memory leaks!*

memory leak:

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

- *Be careful with testing for equality!*

assume p and q are pointers

compares nothing because it's assignment
compares values in pointers
compares values in pointees

- *Don't return addresses of local variables!*

```
int *example1_badcode() {
   int i = 11;
   return &i;
}

int *example2_badMakeIntArray(int size) {
   int a[size];
   return a;
}
```

Meet C Strings

What? A *string* is

-

A *string literal* is

-

How?

```
void someFunction(){
   char str1[9] = "CS 354";
```

C	S	\b	3	5	4	\0

→ During execution, where are string literals allocated?

→ During execution, where is `str1` allocated?

→ Draw the memory diagram of `str1`.

→ Declare a character pointer, named `sptr1`, and initialize it with the literal “CS 354”.

→ Draw the memory diagram of `sptr1`.

STRING CAVEATS - Assignment!

→ Assume `str1` and `sptr1` have been declared in `somefunction` above, what happens when the code below is attempted to be compiled and run?

1. `str1 = "folderol";`

2. `sptr1 = "mumpsimus";`

❋ *Both char [ ] and char * variables*

Meet `string.h`

What? *`string.h`* is

`int strlen(const char *str)`

Returns the length of string `str` up to but *not* including the null character.

`char *strcpy(char *dest, const char *src)`

Copies the string pointed to by `src` to the memory pointed to by `dest` and terminates with the null character.

`char *strcat(char *dest, const char *src)`

Appends the string pointed to by `src` to the end of the string pointed to by `dest` and terminates with the null character.

`int strcmp(const char *str1, const char *str2)`

Compares the string pointed to by `str1` to the string pointed to by `str2`.

❋ *Use strcpy (or strncpy) to*

STRING CAVEATS - `strcpy`!

→ Assume `str1` and `sptr1` have been declared in `somefunction` on the prior page, what happens when the code below is attempted to be compiled and run?

3. `strcpy(str1, "formication");`

4. `strcpy(sptr1, "vomitory");`