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		Dynamic Allocations During runtime.

Why? Heap memory enables

- A program to access more memory than what's allocated to it by the compiler.
- Blocks of memory to be allocated and freed in an arbitrary order during runtime.

How?

IN stdlib.h

`malloc(size_in_bytes)`: Heap mem allocator. return a generic pointer. that is safely assigned to any pointer type.

`free(pointer)`: Frees the heap mem block that pointer points to

operator

`sizeof(operand)`:
Returns the sizes in bytes of its operand.

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

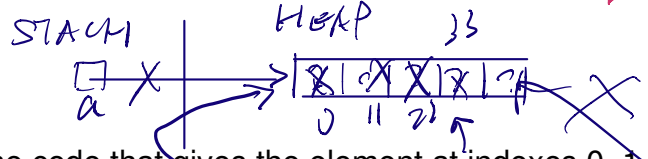
8 1 4

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

```
void someFunction(){
    int *a;
    a = malloc(sizeof(int)*5);
```

→ Draw a memory diagram showing array `a`.

DANGER: elements are uninitialized by default.



→ 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.

```
*a = 0
a[1] = 11
*(a+2) = 22.
```

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

```
int *p = a
p[3] = 33
```

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

```
free(a);
a = null;
```

still points to heap but now is a dangling point.

Pointer Caveats

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

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

Very bad. Intermittent error

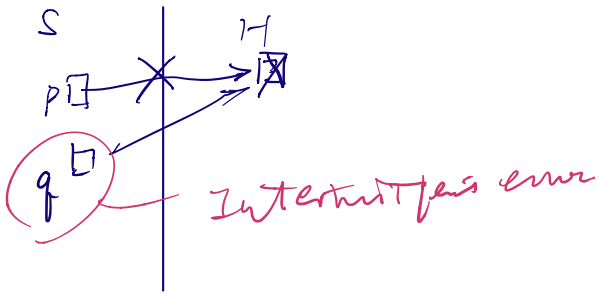
```
int *q = NULL;
*q = 11;
```

SEC Fault.

- *Don't dereference freed pointers!*

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

P=Null

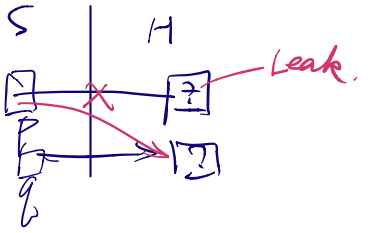


dangling pointer: PTR var with an Addr to mem that has been freed.

- *Watch out for heap memory leaks!*

memory leak: Heap Mem that is useable since it hasn't been properly freed.

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



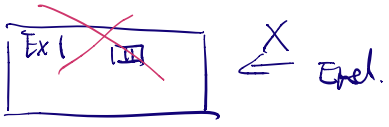
- *Be careful with testing for equality!*

assume p and q are pointers

p = q	compares nothing because it's assignment
p == q	compares values in pointers
*p == *q	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;
}
```

DANGER. both examples result in dangling pointers, when the callee ends execution its stack frame (box) is freed.

Meet C Strings

'10' Diagram

What? A *string* is

- A sequence of characters terminated with a null char

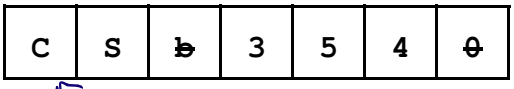
A *string literal* is

- Represented as a 1D array of chars with a minimum size equal to string length +1

How?

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

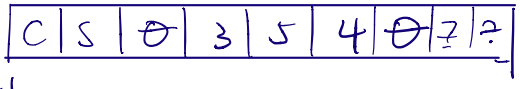
This initialization copies the characters of literal string into str1 char array.



→ During execution, where are string literals allocated? code segment is read only.

→ During execution, where is `str1` allocated? stack.

→ Draw the memory diagram of `str1`.



str1

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

```
char *sptr1 = "CS 354";
```

This initialization copy the addr of the litteral string into sptr1

sptr1

→ 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";`

   DANGER: Compiler Error
   This Assignment attends to assign a char * to an identifier of type char[9]

2. `sptr1 = "mumpsimus";`

   Compile, run
   this assignment assign a char* to a variable of type char*.

❋ *Both char [ ] and char * variables*

Can be initialize with a string literal but only a char* variable can be assigned to a string literal.

Meet `string.h`

What? *`string.h`* is a collection of functions to manipulate C strings

```
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.

destination must be large enough for the result.

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

otherwise, string overflow

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`.

<0 if str1 comes before str2
0 if str1 is the same as str2.
>0 if str1 comes after str2.

❋ *Use strcpy (or strncpy) to* copy a string from on character to another

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");` 9 char string overflow
   Compile, runs
4. `strcpy(sptr1, "vomitory");`
   Compile, run crashes with segmentation fault.
   since sptr1 is pointing the code seg. which is read only.