**CS241 Lecture 4 August 31, 2015 - Angrave and Karrels - Structures and I/O**

How is an integer stored in memory?

little endian: bytes[0] + 256\*bytes[1] + 2562\*bytes[2] + 2563\*bytes[3]

big endian: 2563\*bytes[0] + 2562\*bytes[1] + 256\*bytes[2] + bytes[3]

What's the difference between a and b? Give an example of something you can do with a but not b

char a[] = "Hello";

char\* b = "Hello";

sizeof() returns the number of bytes. So using above code, what is sizeof(a) and sizeof(b) ?

What is sizeof(sizeof(a))?

|  |  |  |
| --- | --- | --- |
|  | Lifetime | Storage |
| Local variables |  |  |
| Global variables |  |  |
| Malloc()’d memory |  |  |
| Static local variables |  |  |

Which of the following code is incorrect?

|  |  |
| --- | --- |
| int\* f1(int \*p) {  \*p = 42;  return p;  } | char\* f2() {  char p[] = "Hello";  return p;  } |
| char\* f3() {  char\* p = "Hello";  return p;  } | char\* f4() {  static char p[] = "Hello";  return p;  } |

How do you allocate memory on the heap? How to you deallocate it?

Fix & Complete this function!

char\* mystrdup(char\*source) {

char\* p = \_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ ( sizeof(source) );

strcpy(source, p);

return p;

}

What is double free? How can you avoid?

What is a dangling pointer? How do you avoid?

How do you declare a stuct in C?

*(trivia)* How is a C struct different from a C++ class?

What is 'typedef' and how do you use it?

With C (FILE\*) I/O, how do you:

* Open a file for reading? For writing?
* Write formatted text?
* Read formatted text?
* Write byte arrays?
* Read byte arrays?
* Close a file?

With POSIX (file-descriptor) I/O, how do you:

* Open a file for reading? For writing?
* Write formatted text?
* Read formatted text?
* Write byte arrays?
* Read byte arrays?
* Close a file?

Which set of I/O calls are built on top of which?

Which set of I/O calls are buffered by default?

Why buffer?

**Demo 1 - How big is the stack?**

#include <stdio.h>

void useStack(int depth) {

char array[1024];

array[0] = 'x';

fprintf(stderr, "%ld bytes\n", depth \* sizeof(array));

useStack(depth+1);

}

int main() {

useStack(1);

return 0;

}

**Demo 2 - Where in memory are variables stored?**

#include <stdio.h>

#include <stdlib.h>

int global\_int;

void myfn2() {

int local;

printf("myfn2 stack: %p\n", &local);

}

void myfn1() {

int local;

printf("myfn1 stack: %p\n", &local);

myfn2();

}

int \*myfn0() {

static int static\_local;

int local;

printf("myfn0 stack: %p\n", &local);

myfn1();

return &static\_local;

}

int main() {

int \*static\_local = myfn0();

int \*heap = (int\*) malloc(sizeof(int));

printf("static local: %p\n", static\_local);

printf("global: %p\n", &global\_int);

printf("string literal: %p\n", "this is a string");

printf("heap: %p\n", heap);

free(heap);

return 0;

}

**Demo 3 - How far we can write before crashing?**

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*p = malloc(sizeof(int) \* 4);

int i=0;

while (1) {

printf("write to p[%d]\n", i);

// fprintf(stderr, "write to p[%d]\n", i);

// To redirect stdout to a file:

// ./program > outfile

// To redirect stderr to a file (with the bash shell):

// ./program &> outfile

// If the output is sent to a file, why does stderr produce

// more output than stdout?

p[i] = 0;

i++;

}

free(p);

printf("No crash??\n");

return 0;

}

**Demo 4 – Try to corrupt malloc’s data structures**

#include <stdio.h>

#include <stdlib.h>

// try to corrupt malloc's data structures

int main() {

int \*p = malloc(sizeof(int) \* 4);

// p[4] = 0;

// p[4] = 0xffffffff;

// p[-1] = 0;

p[-1] = 1;

// p[-1] = 0xffffff;

free(p);

printf("No crash!\n");

return 0;

}