lab[12]

CS 2110 Lab 12:

Storage, Pointers, Arrays, and Strings

Wednesday, July 6, 2022

Lab Assignment: C Continuation

- Go to Quizzes on Canvas
- Select **Lab 12**, password: **gdb\0**
- Get 100% to get attendance!
  - Unlimited attempts
  - Collaboration is **allowed**!
  - Ask your TAs for help :)

Homework 6

- Covers Assembly Subroutines and Calling Convention
- Released!
- **Due Thursday July 7th at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)

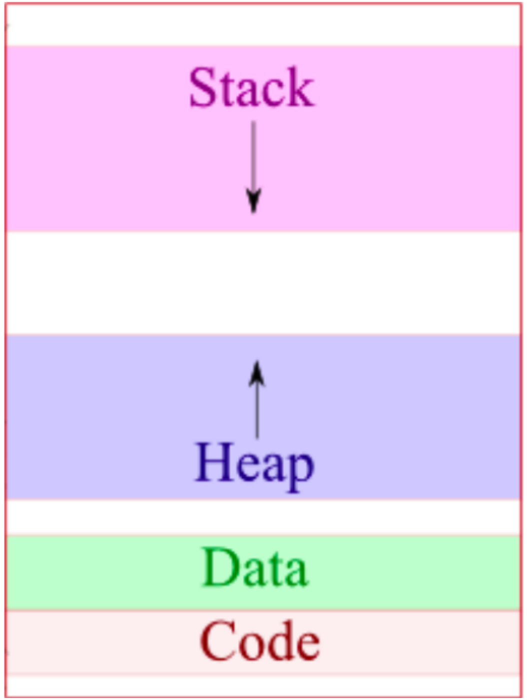
Homework 7 — Intro to C

- Covers introductory C concepts
- Released Friday, July 1nd
- **Due Monday, July 11th at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)

Memory Layout & Storage

Memory Regions in C

- Just like in the LC-3, we have distinct memory regions
- Stack – local variables and arguments are stored here
- Heap – dynamically allocated memory
  - Related to allocating a “new” object in Java
  - We will learn more about this later
- Data – global and static variables are stored here
- Code – our executable code



Global Variables

x is a global variable stored in the "data" section of our memory
y is a local variable stored on the stack (specifically, on the stack frame for `main`)

```
int x = 4;
int main(void) {
    int y = 3;
    return 0;
}
```

Type Qualifiers: `static`

- static defined *functions* are not visible outside of its C file (like private in java)
- static defined *global variables* are not visible outside of its C file (like private in java)
- static defined *local variables* do not lose values between function calls, they are stored in Data section of memory
- This distinction will be on a quiz.

```
foo(); /*x=1*/
foo(); /*x=2*/
foo(); /*x=3*/
foo(); /*x=4*/
```

```
void foo(void) {
    static int x = 0;
    x++;
    printf("x=%d\n", x);
}
```

Type Qualifiers: `const` and `extern`

- `const` defines a variable as constant
- Ex: `const int x = 5;`
- `extern` tells the compiler that the variable has been defined in another file

other.c

```
int x = 5; /*global var*/
```

main.c

```
extern int x;
int main(void) {
     printf("%d", x) /*prints 5*/
}
```

Pick all statements that are true

- static defined global variables are visible outside of their C file
- static defined functions are not visible outside of their C file
- extern tells the compiler that variable has been defined in another file
- The heap contains statically allocated memory

Pointers, Arrays, Strings & Pointer Arithmetic

MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

POINTERS

- Pointers are variables that contain a ***memory address***
- They also have a ***type***
  - This refers to the type of the data AT that memory address.
  - There is also a special case: void pointers which point to a memory address but there is no type for the data at the address
- Denoted by an asterisk symbol following the type

The type of a pointer refers to:

- The type of the value inside the pointer variable
- The type of the value stored at the memory address inside the pointer variable
- Pointers do not have a type as they are memory addresses, not values
- The font of the code in which the pointer is declared

POINTER EXAMPLES

- Some pointer types:

```
char *x;     // declares that x is a pointer to a char.
char **y;    // declares that y is a pointer to a pointer to a char.
void *z;     // declares that z is a pointer to an unspecified type
```

- An & symbol can be used to find the address of a code element such as a variable or function, and then assigned to a pointer:

```
char i = 97;       // i stores the value 97 or 'a'
char *x = &i;      // x stores the address of the variable i
char **y = &x;     // y stores the address of the variable x
void *z = &i;      // z also stores the address of the variable i
```

DEREFERENCING POINTERS

- We say that a pointer ***points at*** or ***refers to*** the memory value at the address contained in the pointer
- We can use the operator * to ***dereference*** a pointer; in other words, to get the value that a pointer is pointing to.

```
char i = 97;    // i stores the value 97 or 'a'
char *x = &i;   // x stores the address of the variable i

x == 97;        // FALSE: the address stored in x is not 97
*x == 97;       // TRUE: the value at address x is 97

*x = 0;         // the value at address x is set to 0 or NULL (this changes i)
x = 0;          // x is now a NULL pointer (segfault if dereferenced)
```

When this block of code finishes executing, the value of the variable *i* will be:

```
int i = 5;
int *x = &i;
*x = 12;
x = 0;
```

ARRAYS

- Arrays in C are much like arrays in other languages.
- They declare a **fixed-size** sequence of same-typed elements. They are laid out consecutively in memory, like in LC-3 assembly.
  - `int arr[5];` // declares arr as an array of 5 ints
  - `char arr[] = {'A',66,'C'};` // arr is an array of 3 chars
- You can also make arrays of pointers:
  - `int *arr[3];` // arr is an array of 3 *int pointers*, or 3 memory addresses.
  - Each memory address holds the type specified (in this example, `int`)
- Unlike in Java, you cannot count on uninitialized data to be NULL or 0; it is simply *un-initialized*.

ARRAYS

- Arrays in C are also similar to *pointers:*
  - `int arr[5];`
  - `arr[0] = 1;` // arr at index 0 is assigned value 1
  - `*arr = 1;` // arr at index 0 is assigned value 1
- In the second assignment, ***arr*** is treated as a ***pointer*** to the first value in the array (in other words, the address of this value)
- "Arrays decay to pointers:" arrays can implicitly be converted to pointers to their first element; however, unlike pointers, they cannot be reassigned
  - In other words, you can write `arr` instead of `&arr[0]`
- Pointers may also be accessed using array notation

SIZES OF C TYPES

- C has a special **`sizeof`**`(type_t)` operator that returns the size of a type.
  - What is the size of a type?
- Size of a type is expressed **in bytes:** i.e. a 32-bit integer has a size of 4 bytes.
- Remember, the size of a type is often ***architecture dependent*.**
  - **Always use `sizeof()`** to express/use the size of a type.
- Examples:
  - `sizeof(int)` = *depends (often 32 bits, **4 bytes**)*
  - `sizeof(char)` = **1 byte**
  - `sizeof(long)` = *depends (often 64 bits, **8 bytes**, 32 bits on Windows)*
- To avoid these variable sizes, there are explicitly-sized types in `<inttypes.h>`
  - `uint8_t, uint16_t, uint32_t` …

POINTER ARITHMETIC

- ***Pointer arithmetic*** is the special way that C treats adding and subtracting to/from pointers; it adds ***offset*** times ***the size of the pointer type.***
- So, using what we just learned, given:
  - `int *y;`
  - `y + 2` evaluates to `y + 2*sizeof(int)`
- This also applies to arrays:
  - `int arr[4];`
  - `arr[3] = 12;`
  - `*(arr + 3) = 12;`
- The above evaluates to **`arr + 3*sizeof(int)`** to get the correct physical address of the fourth element.

What value is printed out by this block of code?

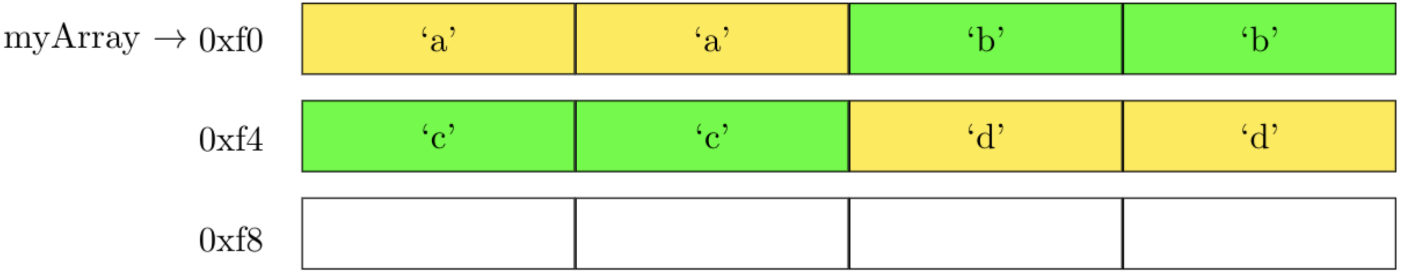
```
// sizeof(int) = 4
int *x = 0xf000;
x += 2;
printf(“%p\n”, x);
```

What value is printed out by this block of code?

```
int i = 94;
int *x = &i;
x = 0;
*x = 42;
printf(“%d\n”, i);
```

ARRAYS

```
short myArray[]     =   {0x6161, 0x6262, 0x6363, 0x6464};
sizeof(short)       ==  ?          (answer given the layout below)
*(myArray + 2)      ==  ?
```



myArray → 0xf0	‘a’	‘a’	‘b’	‘b’
0xf4	‘c’	‘c’	‘d’	‘d’
0xf8				

STRINGS

- Strings in C make use of all the concepts you just learned
- Strings in C are accessed through a pointer to the first character
  - `char *a;   //denotes a pointer to a character`
- C-strings are ***null terminated*** (just like in assembly)
- You can think of this as an array of characters in memory.
- A string can be declared using either string literal or array notation:

```
char *a = "Hello";  // null terminator is added implicitly
char b[] = {'W', 'o', 'r', 'l', 'd', '\0'};
   // you must add the null terminator
```

- Why is it important to have the null terminator?

Which of these is a valid type for a string in C?

- `char`
- `char*`
- `char[]`
- `String`