CS 61C Spring 2023

C and Memory

Discussion 2

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and if false, correct the statement to make it true:

1.1 Memory sectors are defined by the hardware, and cannot be altered.

False. The four major memory sectors, stack, heap, static/data, and text/code for any given process (application) are defined by the operating system and may differ depending on what kind of memory is needed for it to run.

What's an example of a process that might need significant stack space, but very little text, static, and heap space? (Almost any basic deep recursive scheme, since you're making many new function calls on top of each other without closing the previous ones, and thus, stack frames.)

What's an example of a text and static heavy process? (Perhaps a process that is incredibly complicated but has efficient stack usage and does not dynamically allocate memory.)

What's an example of a heap-heavy process? (Maybe if you're using a lot of dynamic memory that the user attempts to access.)

1.2 For large recursive functions, you should store your data on the heap over the stack.

False. Generally speaking, if you need to keep access to data over several separate function calls, use the heap. However, recursive functions call themselves, creating multiple stack frames and using each of their return values. If you store data on the heap in a recursive scheme, your `malloc` calls may lead to you rapidly running out of memory, or can lead to memory leaks as you lose where you allocate memory as each stack frame collapses.

2 Pass-by-who?

2.1 Implement the following functions so that they work as described.

(a) Swap the value of two **int**s. *Remain swapped after returning from this function.*
Hint: Our answer is around three lines long.

```
void swap(int *x, int *y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

(b) Return the number of bytes in a string. *Do not use* `strlen`.
Hint: Our answer is around 5 lines long.

```
int mystrlen(char* str) {
    int count = 0;
    while (*str != 0) {
        str++;
        count++;
    }
    return count;
}
```

3 Bit-wise Operations

3.1 In C, we have a few bit-wise operators at our disposal:

- `AND` (`&`)
- `NOT` (∼)
- `OR` (`|`)
- `XOR` (∧)
- `SHIFT LEFT` (`<<`)
  - Example: `0b0001 << 2 = 0b0100`
- `SHIFT RIGHT` (`>>`)
  - Example: `0b0100 >> 2 = 0b0001`

a	**b**	**a & b**	**a | b**	**a ∧ b**	**∼a**
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

For your convenience, truth tables for the logical operators are provided above. With the binary numbers `a`, `b`, and `c` below, perform the following bit-wise operations:
`a = 0b1000 1011`

```
b = 0b0011 0101
c = 0b1111 0000
```

(a) `a & b`

`0b0000 0001`

(b) `b` $\wedge \sim$ `c`

`0b0011 1010`

(c) `a | 0`

`0b1000 1011`
Anything `| 0` always evaluates to the original value, so this just returns the value of `a`.

(d) `a | (c >> 5)`

`0b1000 1111`
`c >> 5` evaluates to 0b0000 0111.

3.2 Bitwise operators are frequently useful for implementing functions on specific bits of a value. For the following operations, describe what they do to the input `x` (an arbitrary 8-bit integer) :

(a) `x | 0b10000001`

Turns on the first and last bit of `x`.

(b) `x & 0b00000001`

Turns off every bit except the last bit of `x`. This can also be interpreted as `x % 2`.

(c) `x` $\wedge$ `0b11110000`

Flips the first four bits of `x`.

4 Memory Management

C does not automatically handle memory for you. In each program, an address space is set aside, separated in 2 dynamically changing regions and 2 'static' regions.

- The Stack: local variables inside of functions, where data is garbage immediately after the *function in which it was defined* returns. Each function call creates a stack frame with its own arguments and local variables. The stack dynamically changes, growing downwards as multiple functions are called within each other (LIFO structure), and collapsing upwards as functions finish execution and return.
- The Heap: memory manually allocated by the programmer with `malloc`, `calloc`, or `realloc`. Used for data we want to persist beyond function calls, growing upwards to 'meet' the stack. Careful heap management is necessary

to avoid Heisenbugs! Memory is freed only when the programmer explicitly frees it!

- Static data: global variables declared outside of functions, does not grow or shrink through function execution.
- Code (or Text): loaded at the start of the program and does not change after, contains executable instructions and any pre-processor macros.

There are a number of functions in C that can be used to dynamically allocate memory on the heap. The following are the ones we use in this class:

- `malloc(size_t size)` allocates a block of `size` bytes and returns the start of the block. The time it takes to search for a block is generally not dependent on `size`.
- `calloc(size_t count, size_t size)` allocates a block of `count * size` bytes, **sets every value in the block to zero**, then returns the start of the block.
- `realloc(void *ptr, size_t size)` "resizes" a previously-allocated block of memory to `size` bytes, returning the start of the resized block.
- `free(void *ptr)` deallocates a block of memory which starts at `ptr` that was previously allocated by the three previous functions.

4.1 For each part, choose one or more of the following memory segments where the data could be located: **code, static, heap, stack**.

(a) Static variables

Static

(b) Local variables

Stack

(c) Global variables

Static

(d) Constants (constant variables or values)

Code, static, or stack

Constants can be compiled directly into the code. `x = x + 1` can compile with the number 1 stored directly in the machine instruction in the code. That instruction will always increment the value of the variable `x` by 1, so it can be stored directly in the machine instruction without reference to other memory. This can also occur with pre-processor macros:

```
1  #define y 5
2
3  int plus_y(int x) {
4      x = x + y;
5      return x;
6  }
```

Constants can also be found in the stack or static storage depending on if it's declared in a function or not.

```
const int x = 1;

int sum(int* arr) {
    int total = 0;
    ...
}
```

In this example, `x` is a variable whose value will be stored in the static storage, while `total` is a local variable whose value will be stored on the stack. Variables declared **`const`** are not allowed to change, but the usage of **`const`** can get more tricky when combined with pointers.

(e) Functions (i.e. Machine Instructions)

Code

(f) Result of Dynamic Memory Allocation(`malloc` or `calloc`)

Heap

(g) String Literals

Static.

When declared in a function, string literals can only be stored in static memory. String literals are declared when a character pointer is assigned to a string declared within quotation marks, i.e. **`char`**`* s = "string"`. You'll often see a near identical alternative to declaring a string: **`char`** `s[7] = "string"`. This string array will be stored in the stack (when declared inside a function) and is mutable, though they cannot change in size. Note that the compiler will arrange for the char array to be initialised from the literal and be mutable.

4.2 Write the code necessary to allocate memory on the heap in the following scenarios

(a) An array `arr` of k integers

```
arr = (int *) malloc(sizeof(int) * k);
```

(b) A string `str` containing p characters

`str = (char *) malloc(sizeof(char) * (p + 1));` Don't forget the null terminator!

(c) An $n \times m$ matrix `mat` of integers initialized to zero.

```
mat = (int *) calloc(n * m, sizeof(int));
```

Alternative solution. This might be needed if you wanted to efficiently permute the rows of the matrix.

```
mat = (int **) calloc(n, sizeof(int *));
for (int i = 0; i < n; i++)
    mat[i] = (int *) calloc(m, sizeof(int));
```

4.3 Compare the following two implementations of a function which duplicates a string. Is either one correct? Which one runs faster?

```
char* strdup1(char* s) {
    int n = strlen(s);
    char* new_str = malloc((n + 1) * sizeof(char));
    for (int i = 0; i < n; i++) new_str[i] = s[i];
    return new_str;
}
char* strdup2(char* s) {
    int n = strlen(s);
    char* new_str = calloc(n + 1, sizeof(char));
    for (int i = 0; i < n; i++) new_str[i] = s[i];
    return new_str;
}
```

The first implementation is incorrect because malloc doesn't initialize the allocated memory to any given value, so the new string may not be null-terminated. This is easily fixed, however, just by setting the last character in new_str to the null terminator. The second implementation is correct since calloc will set each character to zero, so the string is always null-terminated.

Between the two implementations, the first will run slightly faster since malloc doesn't need to set the memory values. calloc does set each memory location, so it runs in O(n) time in the worst case. Effectively, we do "extra" work in the second implementation setting every character to zero, and then overwrite them with the copied values afterwards.

4.4 What's the main issue with the code snippet seen here? (`gets()` is a function that reads in user input and stores it in the array given in the argument.)

```
char* foo() {
    char buffer[64];
    gets(buffer);

    char* important_stuff = (char*) malloc(11 * sizeof(char));

    int i;
    for (i = 0; i < 10; i++) important_stuff[i] = buffer[i];
    important_stuff[i] = '\0';
    return important_stuff;
}
```

If the user input contains more than 63 characters, then the input will override other parts of the memory! (You will learn more about this and how it can be used to maliciously exploit programs in CS 161.)

Note that it's perfectly acceptable in C to create an array on the stack. It's often discouraged (mostly because people often forget the array was initialized on the stack and accidentally return a pointer to it), but it's not an issue itself, especially here since we don't return a pointer to the array on the stack.

5 Endianness

Compare these different ways of storing 'DEADBEEF'. Assume that each program is run on the same machine and architecture.

```
char[] arr = 'DEADBEEF'
```

```
int arr[2];
arr[0] = 0xDEADBEEF;
arr[1] = 0x00000000;    //null terminator in hex is 0x00
```

5.1 Do these two C programs store 'DEADBEEF' in memory the same way?

No, the first program stores 'DEADBEEF' as a null-terminated array of chars, each element stored as their ASCII value (e.g D = 0x44), while the latter encodes the hexadecimal DEADBEEF as a 32b integer.

You take a look at the ASCII table and translate the string 'DEADBEEF' into bytes.

```
int arr[2];
//storing 'DEAD' in ascending order in arr[0]
arr[0] = 0x44454144

//storing 'BEEF' in ascending order in arr[1]
arr[1] = 0x42454546
```

5.2 Does this C program store 'DEADBEEF' in memory the same way as storing it as a string?

It depends. A string is stored with the leftmost character in the lowest address, up to the null terminator, regardless of the endianness. Storing the string as an integer changes depending on the system architecture. In a big-endian system, the integers would be stored in the same order byte-by-byte as the string 'DEADBEEF'. However, in a little-endian system, the Least Significant Byte is stored in the lowest address, so arr[0] in memory would actually be stored, in ascending order, as `0x44 0x41 0x45 0x44`, and arr[1] as `0x46 0x45 0x45 0x42`.