CS 61C Fall 2024

C

Discussion 2

1 Pre-Check: C

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 include an explanation:

1.1 True or False: C is a pass-by-value language.

True. If you want to pass a reference to anything, you should use a pointer.

1.2 In compiled languages, the compile time is generally pretty fast, however the run-time is significantly slower than interpreted languages.

False. Reasonable compilation time, excellent run-time performance. It optimizes for a given processor type and operating system.

1.3 What is a pointer? What does it have in common to an array variable?

As we like to say, "everything is just bits." A pointer is just a sequence of bits, interpreted as a memory address. An array acts like a pointer to the first element in the allocated memory for that array. However, an array name is not a variable, that is, &arr = arr whereas &ptr != ptr unless some magic happens (what does that mean?).

1.4 If you try to dereference a variable that is not a pointer, what will happen? What about when you free one?

It will treat that variable's underlying bits as if they were a pointer and attempt to access the data there. C will allow you to do almost anything you want, though if you attempt to access an "illegal" memory address, it will segfault for reasons we will learn later in the course. It's why C is not considered "memory safe": you can shoot yourself in the foot if you're not careful. If you free a variable that either has been freed before or was not malloced/calloced/realloced, bad things happen. The behavior is undefined and terminates execution, resulting in an "invalid free" error.

1.5 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.)

2 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.

2.1 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:

```
#define y 5

int plus_y(int x) {
    x = x + y;
    return x;
}
```

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.

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

(a) An array `arr` of k integers

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

(b) A string `str` containing p characters

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

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

```
mat = calloc(n * m, sizeof(int));
```
Alternative solution. This might be needed if you wanted to efficiently permute the rows of the matrix.

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

(d) Unallocating all but the first 5 values in an integer array `arr`. (Assume `arr` has more than 5 values)

```
arr = realloc(arr, 5 * sizeof(int));
```

2.3 Compare the following two implementations of a function which duplicates a string. Is either one correct?

```
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.

3 C Generics

3.1 **True or False:** In C, it is possible to directly dereference a **void** * pointer, e.g.

```
... = *ptr;
```

False. To dereference a pointer, we must know the number of bytes to access from memory at compile time. Generics employ generic pointers and therefore cannot use the dereference operator!

3.2 Generic functions (i.e., generics) in C use void * pointers to operate on memory without the restriction of types. Such generics pointers do not support dereferencing, as the number of bytes to access from memory is not known at compile-time. They instead use byte handling functions such as `memcpy` and `memmove`.

Implement `rotate`, which will prompt the following program to generate the provided output.

```
int main(int argc, char *argv[]) {
  int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  print_int_array(array, 10);
  rotate(array, array + 5, array + 10);
  print_int_array(array, 10);
  rotate(array, array + 1, array + 10);
  print_int_array(array, 10);
  rotate(array + 4, array + 5, array + 6);
  print_int_array(array, 10);
  return 0;
}
```

Output:

```
$ ./rotate
Array: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Array: 6, 7, 8, 9, 10, 1, 2, 3, 4, 5
Array: 7, 8, 9, 10, 1, 2, 3, 4, 5, 6
Array: 7, 8, 9, 10, 2, 1, 3, 4, 5, 6
```

Your Solution:

```
void rotate(void *front, void *separator, void *end) {
  size_t width = (char *) end - (char *) front;
  size_t prefix_width = (char *) separator - (char *) front;
  size_t suffix_width = width - prefix_width;
  char temp[prefix_width];
  memcpy(temp, front, prefix_width);
  memmove(front, separator, suffix_width);
  memcpy((char *) end - prefix_width, temp, prefix_width);
}
```

4 Pass-by-who?

4.1 Consider the following blocks of C code:

```
1  void printall(int *x) {
2      // Suppose x points to 0xABDE2464
3      const int NUM_ELEMS = 3;
4      for(int i = 0; i < NUM_ELEMS; i += 1) {
5          printf("Address: %x \n", x);
6          x++;
7      }
8  }
```

(a) What three memory addresses are printed by this program?

0xABDE2464, 0xABDE2468, 0xABDE246C.

```
1  void printall(char *x) {
2      // Suppose x points to 0xABDE2464
3      const int NUM_ELEMS = 3;
4      for(int i = 0; i < NUM_ELEMS; i += 1) {
5          printf("Address: %x \n", x);
6          x++;
7      }
8  }
```

(b) What three memory addresses are printed by this program?

0xABDE2464, 0xABDE2465, 0xABDE2466.

4.2 The following functions may contain logic or syntax errors. Find and correct them.

(a) Returns the sum of all the elements in `summands`.

It is necessary to pass a size alongside the pointer.

```
1  int sum(int* summands, size_t n) {
2      int sum = 0;
3      for (int i = 0; i < n; i++)
4          sum += *(summands + i);
5      return sum;
6  }
```

(b) Increments all of the letters in the `string` which is stored at the front of an array of arbitrary length, `n <= strlen(string)`. Does not modify any other parts of the array's memory.

The ends of strings are denoted by the null terminator rather than n. Simply having space for n characters in the array does not mean the string stored inside is also of length n.

```
1  void increment(char* string) {
2      for (i = 0; string[i] != 0; i++)
3          string[i]++; // or (*(string + i))++;
```

```
}
```

Another common bug to watch out for is the corner case that occurs when incrementing the character with the value 0xFF. Adding 1 to 0xFF will overflow back to 0, producing a null terminator and unintentionally shortening the string.

4.3 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;
}
```

5 Endianness

- Machines are byte-addressable. Memory is like a large array of cells. Each storage cell stores 8 bits, and these byte cells are ordered with an address.
- A 32b architecture has 32 bit memory addresses, addresses 0x00000000 - 0xFFFFFFFF

Typed variables

- Examples: int, long, char
- sizeof(dataType) indicates the number of bytes in memory required to store a particular data type

Pointers

- a variable whose value is an address of another variable
- Declaration: dataType* name;
- Dereference operator: Based on the pointer declaration statement, the compiler fetches the corresponding amount of bytes. For example, if p is a pointer to a 4 byte integer variable x, then *p involves fetching 4 bytes starting from the address of x, which is the value of p. Therefore, the value of x and value of *p are equal

Endianness

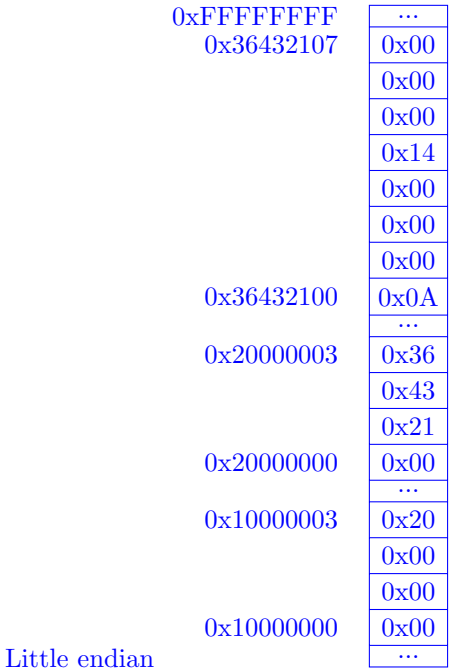
- Recall different data types are stored in x amount of contiguous byte cells in memory
- Big endian: the most significant byte of the value of a variable is stored in memory at the lowest address of the chunk of byte cells allocated for that variable
- Little endian: the least significant byte of the value of a variable is stored in memory at the lowest address of the chunk of byte cells allocated for the variable

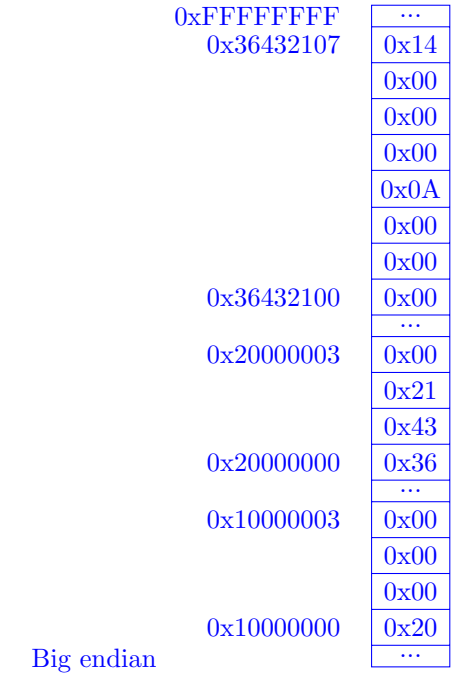
5.1 Based on the following code and a 32b architecture, fill in the values located in memory at the byte cells for both a big endian and little endian system.

Suppose:

- the array nums starts at address 0x36432100
- p's address is 0x10000000

```
1 uint32_t nums[2] = {10, 20};
2 uint32_t* q = (uint32_t*) nums;
3 uint32_t** p = &q;
```





5.2 Provide two answers for the following questions: big endian system and little endian system

Suppose `uint64_t*` y = (`uint64_t*`) nums is executed after the code

1. What does *y evaluate to?

   Because y is a pointer to a `uint64_t*` variable, dereferencing results in evaluating 8 contiguous bytes starting from the value of y (an address in memory = 0x36432100) in big endian or little endian.

Little-endian: 0x00000014 0000000A

Big-endian: 0x0000000A 00000014

2. What does `&q` evaluate to? What does `&nums` evaluate to?

   `&q` evaluates to 0x20000000 in both big endian and little endian. This is the value of variable p (p is located at 0x10000000). `&nums` evaluates to 0x36432100.

   Both q and nums act as pointers to the first element of the nums array. However, nums is not like a variable. The values of `nums` and `&nums` are equal, while the address of variable q is not equal to the address of the data it is pointing to.

3. What does *(q+1) evaluate to?

   *(q+1) = nums[1] = *(nums+1) = 20 (decimal). q and nums have the same value. q is a pointer to a 32 bit integer. Therefore, *(q+1) means the 4 bytes stored starting at address = q plus 1*sizeof(`uint32_t`) = 0x36432100 + 0x4 = 0x36432104 evaluated in big endian or little endian