CS61C Spring 2025

# C Memory Management

Discussion 2

## 1 Memory Management

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

a) Local variables

Stack

b) Global variables

Static

c) 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.

```
1 int x = 1;
2
3 int sum(int* arr) {
4     int total = 0;
5     ...
6 }
```

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.

d) Functions (i.e. Machine Instructions)

Code

e) Results of Dynamic Memory Allocation (`malloc` or `calloc`)

Heap

f) 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 initialized from the literal and be mutable.

1.2 Write the code necessary to allocate memory **on the heap** in the following scenarios:

(a) An array of `arr` of `k` integers

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

(b) A string `str` of length `p`. Note that a string's length is defined by `strlen`

```
char *str = malloc(sizeof(char) * (p + 1));
```

A string's length is defined by `strlen`, and all strings must contain a null terminator. We must allocate space for `p` characters plus one null terminator.

(c) An `n` × `m` matrix `mat` of integers initialized to zero.

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

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

```
1 int **mat = calloc(n, sizeof(int *));
2 for (int i = 0; i < n; i++) {
3   mat[i] = calloc(m, sizeof(int));
4 }
```

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

```
int *arr = ... ;
```

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

This will resize `arr` by deallocating the memory segments containing the remainder of the array.

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

```
1 char* strdup1(char* s) {
2   int n = strlen(s);
3   char* new_str = malloc((n + 1) * sizeof(char));
4   for (int i = 0; i < n; i++) new_str[i] = s[i];
5   return new_str;
6 }
```

```
1 char* strdup2(char* s) {
2     int n = strlen(s);
3     char* new_str = calloc(n + 1, sizeof(char));
4     for (int i = 0; i < n; i++) new_str[i] = s[i];
5     return new_str;
6 }
```

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.

# 2 Pass-by-Who?

2.1 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))++;
4 }
```

Additionally, the operator precedence is incorrect for the expression `*(string + i)++`. The `++` increment will occur *before* the dereference operator occurs. Thus, we need to wrap in parentheses with `(*(string + i))++` or use array subscripting–which has equal precedence as dereferencing–like `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.

(c) Overwrites an input string **`src`** with "61C is awesome!" if there's room. Does nothing if there is not. Assume that **`length`** correctly represents the length of **`src`**.

`char *srcptr, replaceptr` initializes a `char` pointer, and a `char`—not two `char` pointers.

The correct initialization should be, `char *srcptr, *replaceptr`.

2.2 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.

```
1 void swap(int *x, int *y) {
2     int temp = *x;
3     *x = *y;
4     *y = temp;
5 }
```

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

```
1 int mystrlen(char* str) {
2     int count = 0;
3     while (*str != 0) {
4         str++;
5         count++;
6     }
7     return count;
8 }
```

# 3 Endianness

3.1 Suppose we run the following code on a 32b architecture:

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

Find the values located in memory at the byte cells for both a Big Endian and a Little Endian system given that:

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

**Little Endian**

| | |
|---|---|
| 0xFFFFFFFF | … |
| 0x36432107 | 0x00 |
| | 0x00 |
| | 0x00 |
| | 0x14 |
| | 0x00 |
| | 0x00 |
| | 0x00 |
| 0x36432100 | 0x0A |
| | … |
| 0x20000003 | 0x36 |
| | 0x43 |
| | 0x21 |
| 0x20000000 | 0x00 |
| | … |
| 0x10000003 | 0x20 |
| | 0x00 |
| | 0x00 |
| 0x10000000 | 0x00 |
| | … |

**Big Endian**

| | |
|---|---|
| 0xFFFFFFFF | … |
| 0x36432107 | 0x14 |
| | 0x00 |
| | 0x00 |
| | 0x00 |
| | 0x0A |
| | 0x00 |
| | 0x00 |
| 0x36432100 | 0x00 |
| | … |
| 0x20000003 | 0x00 |
| | 0x21 |
| | 0x43 |
| 0x20000000 | 0x36 |
| | … |
| 0x10000003 | 0x00 |
| | 0x00 |
| | 0x00 |
| 0x10000000 | 0x20 |
| | … |

- A 32b architecture has 32-bit memory addresses from `0x00000000` - `0xFFFFFFFF`
- 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 that variable.

3.2 Suppose we add an an additional instruction (line #4) to the end of the previous code block:

```
1  uint32_t nums[2] = {10, 20};
2  uint32_t  *q = (uint32_t *) nums;
3  uint32_t **p = &q;
4  uint64_t  *y = (uint64_t *) nums;
```

Provide answers for the following questions for both a Big Endian system and Little Endian system:

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?

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

3) What does `&nums` evaluate to?

   `&nums` evaluates to `0x36432100` in both big endian and little endian.

   Both `q` and `nums` act as pointers to the first element of the nums array. However, `nums` is different than a normal 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.

4) 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 32b integer. `(q+1)` is equivalent to the address = `q + 1 * sizeof(uint32_t) = 0x36432100 + 0x4 = 0x36432104`. Dereferencing this address gives us the value of `0x14 = 20` (decimal). This is the same in a big endian or little endian system.

# 4 C Generics

4.1 **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. A `void *` pointer contains the address for an arbitrary region of memory without a known size, so they cannot be dereferenced – they must be typecast beforehand (e.g. `... = *((int *) ptr)`)

4.2 Generic functions (i.e., generics) in C use `void *` pointers to operate on memory without the restriction of types. Generic 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.

```
1  int main(int argc, char *argv[]) {
2    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
3    print_int_array(array, 10);
4    rotate(array, array + 5, array + 10);
5    print_int_array(array, 10);
6    rotate(array, array + 1, array + 10);
7    print_int_array(array, 10);
8    rotate(array + 4, array + 5, array + 6);
9    print_int_array(array, 10);
10   return 0;
11 }
```

Output:

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

```
1  void rotate(void *front, void *separator, void *end) {
2    size_t width = (char *) end - (char *) front;
3    size_t prefix_width = (char *) separator - (char *) front;
4    size_t suffix_width = width - prefix_width;
5    char temp[prefix_width];
6    memcpy(temp, front, prefix_width);
7    memmove(front, separator, suffix_width);
8    memcpy((char *) end - prefix_width, temp, prefix_width);
9  }
```

See slides provided under "Discussion Resources" for a visual walkthrough of the solution.