CS 61C Summer 2020

C Basics

Discussion 2: June 24, 2020

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 include an explanation:

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

True

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

A pointer is a variable that stores a memory address. It is very similar to an array variable, because arr[0] is a pointer to element 0, and you can access first element using arr[1] or *(arr+1)

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

Derefencing a pointer means "get value at pointer address". If you dereference a variable that is not a pointer, it would treat that value as if it was a memory address. For instance, int x = 7, x* would reference the memory address "7". As bits are bits, it always works.

1.4 When should you use the heap over the stack? Do they grow?

Free takes some section of memory we have reserved, and there's a pointer p* to that memory. If I call free(p), that memory is no longer reserved. Freeing a value that is not a pointer will error

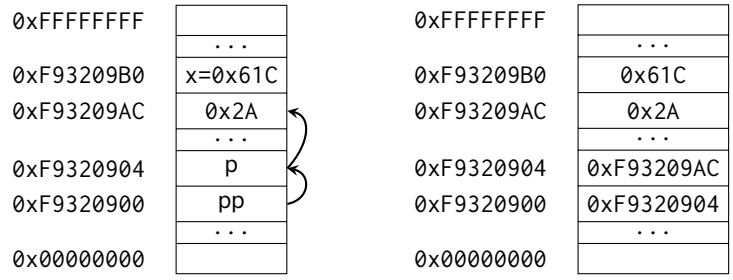
2 C

C is syntactically similar to Java, but there are a few key differences:

1. C is function-oriented, not object-oriented; there are no objects.
2. C does not automatically handle memory for you.
   - Stack memory, or *things that are not manually allocated*: data is garbage immediately after the *function in which it was defined* returns.
   - Heap memory, or *things allocated with* `malloc`, `calloc`, *or* `realloc`: data is freed only when the programmer explicitly frees it!
   - There are two other sections of memory that we learn about in this course, *static* and *code*, but we'll get to those later.
   - In any case, allocated memory always holds garbage until it is initialized!
3. C uses pointers explicitly. If `p` is a pointer, then `*p` tells us to use the value that `p` points to, rather than the value of `p`, and `&x` gives the address of `x` rather than the value of `x`.

   On the left is the memory represented as a box-and-pointer diagram.

   On the right, we see how the memory is really represented in the computer.



Let's assume that `int* p` is located at `0xF9320904` and `int x` is located at `0xF93209B0`. As we can observe:

- `*p` evaluates to `0x2A` (42_{10}).
- `p` evaluates to `0xF93209AC`.
- `x` evaluates to `0x61C`.
- `&x` evaluates to `0xF93209B0`.

Let's say we have an `int **pp` that is located at `0xF9320900`.

2.1 What does `pp` evaluate to? How about `*pp`? What about `**pp`?

pp evaluates to 0xF…904 (what pp is pointing at)
*pp gets value that pp, which is p, which is 0xf9309Ac
**pp is derefence twice, which is *(*(pp)) = *(p) = *p = 0x2A

2.2 The following functions are syntactically-correct C, but written in an incomprehensible style. Describe the behavior of each function in plain English.

(a) Recall that the ternary operator evaluates the condition before the `?` and returns the value before the colon (`:`) if true, or the value after it if false.

sums the first n elements of our array

```
1  int foo(int *arr, size_t n) {
2      return n ? arr[0] + foo(arr + 1, n - 1) : 0;
3  }
```

(b) Recall that the negation operator, `!`, returns 0 if the value is non-zero, and 1 if the value is 0. The `~` operator performs a *bitwise not* (NOT) operation.

returns negative number of zeros in the array

```
1  int bar(int *arr, size_t n) {
2      int sum = 0, i;
3      for (i = n; i > 0; i--)
4          sum += !arr[i - 1];
5      return ~sum + 1;
6  }
```

(c) Recall that `^` is the *bitwise exclusive-or* (XOR) operator.

XOR, if one of x or y is truthy, return 1 if both or none, return 0

Does nothing to x or y

```
1  void baz(int x, int y) {
2      x = x ^ y;
```

```
    y = x ^ y;
    x = x ^ y;      x = x ^ ((x ^ y) ^ y)
}
```

(d) (Bonus: How do you write the *bitwise exclusive-nor* (XNOR) operator in C?)

3 Programming with Pointers

3.1 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.*

```
void swap(
```

(b) Return the number of bytes in a string. *Do not use* `strlen`.

```
int mystrlen(
```

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

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

```
int sum(int* summands) {
    int sum = 0;
    for (int i = 0; i < sizeof(summands); i++)
        sum += *(summands + i);
    return sum;
}
```

(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.

```
void increment(char* string, int n) {
    for (int i = 0; i < n; i++)
```

```
        *(string + i)++;
}
```

(c) Copies the string `src` to `dst`.

```
void copy(char* src, char* dst) {
    while (*dst++ = *src++);
}
```

(d) 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`.

```
void cs61c(char* src, size_t length) {
    char *srcptr, replaceptr;
    char replacement[16] = "61C is awesome!";
    srcptr = src;
    replaceptr = replacement;
    if (length >= 16) {
        for (int i = 0; i < 16; i++)
            *srcptr++ = *replaceptr++;
    }
}
```

4 Memory Management

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

(b) Local variables

(c) Global variables

(d) Constants

(e) Machine Instructions

(f) Result of `malloc`

(g) String Literals

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

(a) An array `arr` of k integers

(b) A string `str` containing p characters

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

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

```
1  char* foo() {
2      char* buffer[64];
3      gets(buffer);
4
5      char* important_stuff = (char*) malloc(11 * sizeof(char));
6
7      int i;
8      for (i = 0; i < 10; i++) important_stuff[i] = buffer[i];
9      important_stuff[i] = "\0";
10     return important_stuff;
11 }
```

Suppose we've defined a linked list **struct** as follows. Assume `*lst` points to the first element of the list, or is `NULL` if the list is empty.

```
struct ll_node {
    int first;
    struct ll_node* rest;
}
```

4.4 Implement `prepend`, which adds one new `value` to the front of the linked list. Hint: why use ll_node $**lst$ instead of ll_node$*lst$?

```
void prepend(struct ll_node** lst, int value)
```

4.5 Implement `free_ll`, which frees all the memory consumed by the linked list.

```
void free_ll(struct ll_node** lst)
```