CS 61C Fall 2020

C Basics

Discussion 2: September 7, 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. If we want to pass the reference, we need to use a pointer.

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

Pointer is a variable which stacks the address of data. An array variable is actually a special pointer.

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

C will treat this variable as a pointer, and try to access the data at the address this "pointer" is pointing to.

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

Heap should be used to store large data, with a pointer in the stack which points to the data.

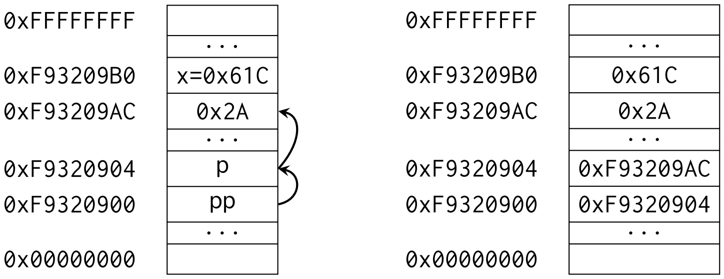
2 C

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 0xF9320904. (pp == &p)
*pp evaluates to 0xF93209AC. (*pp == p)
pp evaluates to 0x2A. (pp == *p)

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.

The foo function will take in an array and an integer n, and return the sum of the first n items.

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

(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.

Count how many ~~non-zero~~ zeros numbers are in the last n items of the arr, and return the negative number of it.

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

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

Exchange the value of x and y.

Note: y = x_org ^ y_org ^ y_org, where y_org ^ y_org evaluates to all zeros,and x_or

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

void swap(int* x, int* y) { int

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

```
int mystrlen(
```

redundant

int mystrlen(char* s) {   c

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`.

The size of array is needed.

This won't work for getting the size of an array.

```
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++)
```

```
        if (*(string + i)) {
            *(string + i)++; } else { break;}
}
```

This function is bad. Shouldn't pass an integer n in. We should just iterate over the string, and stops when there's no more elements in the string.

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

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

Tested in C. There seems to be no errors.

(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++;
    }
}
```

Didn't find something wrong.

Wrong initialization.

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 static

(b) Local variables stack

(c) Global variables static

(d) Constants ~~static~~

(e) Machine Instructions code

(f) Result of `malloc` heap

(g) String Literals ~~heap~~

static or stack. char* s = "string" will be stored in STATIC,while cha

Example: char* s = "string

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

(a) An array `arr` of k integers int* arr = (int*) malloc*(sizeof(int) * k);

(b) A string `str` containing p characters char* str = (char*) malloc(sizeof(char) * p);

(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.)

Get the first 10 characters of the user input.

Well...
The issue is that the user input may be more than 63 chars.
In this case, the input will override other parts of the memory!

```c
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;
}
```

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.

```c
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$*lst$?

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

The reason why a pointer to the pointer is passed in is because we need to change the address of the header.

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

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

```c
void free_ll(struct ll_node** lst) {
    if ((*lst)->rest) {
        free_ll(&((*lst)->rest));
    }
    free(*lst);
    *lst = NULL;
}
```

Should consider the situation where lst is empty.

→

```c
void free_ll(struct ll_node** lst) {
    if (*lst) {
        free_ll(&((*lst)->rest));
        free(*lst);
    }
    *lst = NULL;
}
```