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

1.2 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.

It stores the address It stores the head address of an array

1.3 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

It may point to some place it shouldn't

It may free some incorrect place

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

If you want the variable to survive during the processing of the program. Yes.

If you need to keep access to data over several function calls, use the heap If you're dealing with a large piece of data, passing around a pointer to something on the heap is more efficient and a better practice than passing around the data itself

Heaps grow up and stacks grow down, meeting when working memory is full

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.

0xFFFFFFFF	
	...
0xF93209B0	x=0x61C
0xF93209AC	0x2A
	...
0xF9320904	p
0xF9320900	pp
	...
0x00000000	

0xFFFFFFFF	
	...
0xF93209B0	0x61C
0xF93209AC	0x2A
	...
0xF9320904	0xF93209AC
0xF9320900	0xF9320904
	...
0x00000000	

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?

0xF9320904 0xF93209AC 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.

maybe is not the whole array

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

it returns the sum of ~~arr's~~ elements.

Returns the sum of first N elements in array

(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.

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

it returns ~~the additive~~ invers of sum of ~~an~~ array.

Returns -1 times the number of zeros in the first N elements

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

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

swap the values of x and y within the function

ultimately does not change the value of either x or y

```
    y = x ^ y;
    x = x ^ y;
}
```

~~~(x^y)~~ !(x^y) 或 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( int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

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

```
int mystrlen( char * string ){
    int count = 0;
    while ( string[0] ) {
        string++;
        count++;
    }
    return count;
}
```

simplify →

```
while ( *string++) {
    count ++;
}
```

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 n must pass the size of the array!

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

n

sizeof(~~summands~~) / sizeof(int)

is just a pointer, whose size is not the size of array

(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++)
```

strlen(string)
~~~

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

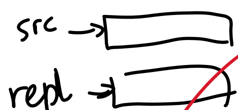
(c) Copies the string src to dst.

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

no errors. the value of the expression is the value of *src.

```
while ( *scr){
    *dst ++ =  * scr++;
}
*dst = '\0',
```

(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.



char *srcptr, replaceptr initializes a char pointer, and a char — not 2 char pointers

```
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

$\boxed{4.1}$ 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 ~~heap~~ static

(d) Constants code/ static / stack →

(e) Machine Instructions code

(f) Result of malloc heap

(g) String Literals 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
Constants can also be found in the stack or static storage depending on if it's declared in a function or not

When declared in a function, string literals can be stored in different places. char *s = "string" is stored in the static memory segment while char[7] s = "string" will be stored in the stack

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 ( k * sizeof (int) )
```

(b) A string `str` containing p characters
```
char * str = (char *) malloc( (p+1) * sizeof (char))
```

(c) An $n \times m$ matrix `mat` of integers initialized to zero.
```
int ** mat = (int **) malloc ( n * sizeof (int *))
for ( int i = 0; i < n; i++) {
    mat [i] = (int *) malloc ( m * sizeof(int) )
    for (int j = 0; j < m; j++ )  mat[i][j] = 0;
}
```

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.)

```
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";  -> '\0'
    return important_stuff;
}
```

If the user input contains more than 63 characters, then the input will override other parts of memory !

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) {
    struct ll_node *newNode = (struct ll_node *)malloc( sizeof (struct ll_node)),
    newNode->first = value;
    newNode->rest = (*lst),
    *lst = newNode;
}
```

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

```
void free_ll(struct ll_node** lst)
    if (*lst) {
        struct ll_node * p = (*lst)->rest,
        while ( p ) {
            free( * lst),
            * lst = p,
            p = p->rest;
        }
```

```
        free(*lst);
    }
```