Jon 112 Dis 2

Tuesday, September 4, 2018 10:22 AM

CS 61C
Fall 2018

C Basics
Discussion 2: September 3, 2018

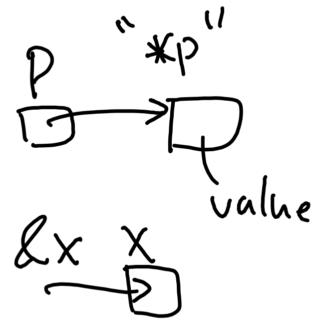
1 C

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 allocated the way you're accustomed to*: data is garbage immediately after the *function in which it was defined* returns.
   - Heap memory, or *things allocated with* `malloc`, `calloc`, *or* `realloc` *commands*: data is freed only when the programmer explicitly frees it!
   - In any case, allocated memory always holds garbage until it is initialized!
3. C uses pointers explicitly. `*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.



Mem

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

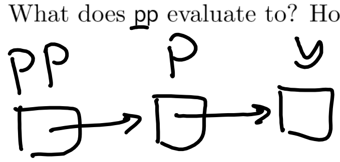
p points to y

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

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

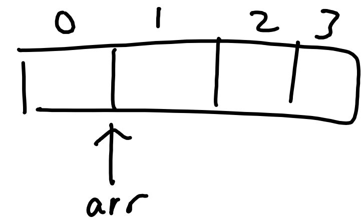
Let's say we have an **int** `**pp` that is located at `0xF9320900`.

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



pp = 0xF9320904
*pp = p = 0xF93209AC
**pp = y = 0x2A

arr[0] + arr[1] + ... + arr[n]



1.2 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.

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

4

//sum first n vals 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;
}
```

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

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

swap x & y locally, but no persisting change

$n \wedge n = 0$

$n \wedge 0 = n$

$x = x \wedge y$

$y = x \wedge y \wedge y = x \wedge 0 = x$

$x = (x \wedge y) \wedge x = 0 \wedge y = y$

2 Programming with Pointers

2.1 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 *x, int *y) {
    *x = *x ^ *y
    *y = *x ^ *y
    *x = *x ^ *y
}
```

Calling swap:

```
int z;
int q;
swap(&z, &q)
```

```
int temp = *x;
*x = *y
*y = temp
```

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

```
int mystrlen(
```

2.2 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;
}
```

sizeof ?
returns Bytes

→ int * ___ } same
int* ___

(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`.

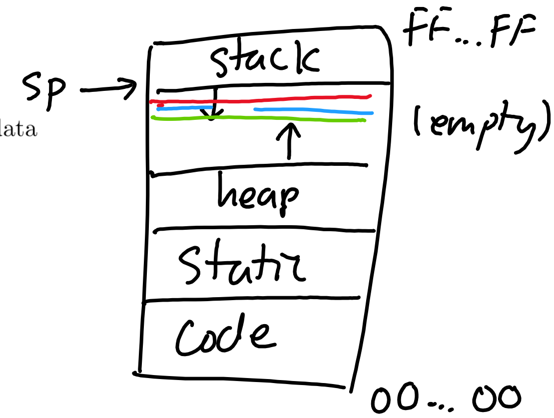
```
void copy(char* src, char* dst) {
    while (*dst++ = *src++);
}
```

(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++;
    }
}
```

char *src ; char *rep ;
char* src, rep

3 Memory Management



3.1 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, stack, #Define → code

(e) Machine Instructions code

(f) Result of `malloc` heap

(g) String Literals static, (stack) char[] = "string"

3.2 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

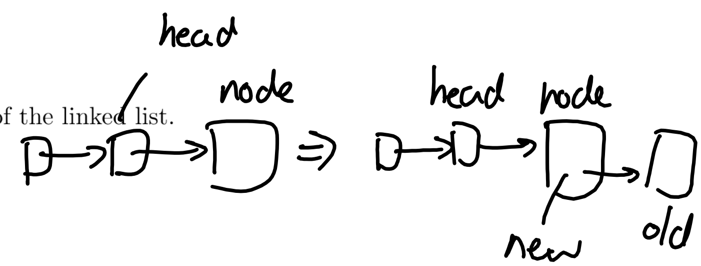
(c) An $n \times m$ matrix `mat` of integers initialized to zero.

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;
}
```

3.3 Implement `prepend`, which adds one new `value` to the front of the linked list.

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



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

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