Jon Dis 2 Notes

Monday, September 3, 2018 2:44 AM

CS 61C Fall 2018

C Basics

Discussion 2: September 3, 2018

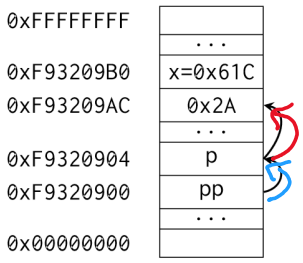
1 C

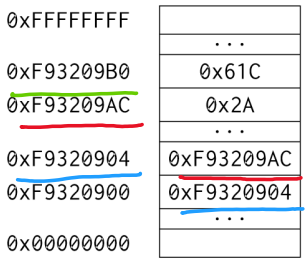
C is syntactically similar to Java, but there are a few key differences:

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.



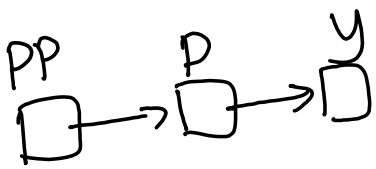


Let's assume that `int* p` is located at `0xF9320904` and `int x` is located at `0xF93209B0`. As we can observe:

- `*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 = 0x F9320904

*pp = p = 0xF93209AC

**pp = *p = y = 0x2A

what is size_t? - represents sizes & counts

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.

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

recursive call
sum first n elems of array
if n nonzero
base case (if n=0)
~ n times

(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.

```
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]; // sum of # of 0's
5     return ~sum + 1; // invert (two's comp)
6 }
```

returns -1 * (# of 0's in first n elems)

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

```
1 void baz(int x, int y) {
2     x = x ^ y;
3     y = x ^ y;
4     x = x ^ y;
5 }
```

both local vars → does nothing
if x replaced with *x and y with *y → changes would persist
What does it do?
2. x = x^y
3. y = (x^y)^y = x
4. x = (x^y)^x = y
- swaps x & 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;
}
```

or
soln on website creates a temp var & swaps them using that temp

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

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

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`.

```
1  int sum(int* summands) size_t n)
2      int sum = 0;
3      for (int i = 0; i < sizeof(summands); i++)
4          sum += *(summands + i);
5      return sum;
6  }
```

n ← gives pointer size, not # elems

(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.

```
1  void increment(char* string, int n) {
2      for (int i = 0; i < n; i++)
3          (*(string + i))++;
4  }
```

← need double parenthesis

or string[i]++;

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

No errors

```
1  void copy(char* src, char* dst) {
2      while (*dst++ = *src++);
3  }
```

(sets *dst = *src, while uses this result, increments both

(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`.

```
1  void cs61c(char* src, size_t length) {
2      char *srcptr, *replaceptr;
3      char replacement[16] = "61C is awesome!";
4      srcptr = src;
5      replaceptr = replacement;
6      if (length >= 16) {
7          for (int i = 0; i < 16; i++)
8              *srcptr++ = *replaceptr++;
9      }
10 }
```

← both need to be pointers!

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
(e) Machine Instructions code
(f) Result of `malloc` heap
(g) String Literals static

pretty much same

code #Define macro
static declared outside
stack declared in func

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

(a) An array `arr` of k integers `arr = (int *) malloc(sizeof(int) * k)`
(b) A string `str` containing p characters `arr = (char *) malloc(sizeof(char) * (p+1))` always cast!
(c) An $n \times m$ matrix `mat` of integers initialized to zero. `arr = (int *) calloc(m, sizeof(int))`

+1 for null term.

note different notation

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;
}
```

no typedef → always refer to this with "struct ll_node"

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

```
void prepend(struct ll_node** lst, int value)
    struct ll_node *new = (struct ll_node *) malloc(sizeof(struct ll_node));
    new -> first = value;
    new -> rest = *lst;
    *lst = new
}
```

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

```
void free_ll(struct ll_node** lst)
    if (*lst)
        free_ll(&((*lst)->rest)); // recursive call before free
        free(*lst);
    }
    *lst = NULL; // always safer to get rid of old pointers
}
```