CS 61C — Spring 2020

C Basics

Discussion 2: February 2, 2020

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 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.

Address	Box-and-pointer
0xFFFFFFFF	
	...
0xF93209B0	x=0x61C
0xF93209AC	0x2A
	...
0xF9320904	p
0xF9320900	pp
	...
0x00000000	

Address	Memory
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`.

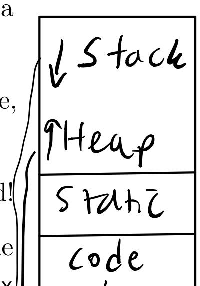
   Let's say we have an **int** `**pp` that is located at `0xF9320900`.

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

Strings end with a null terminator ('\0') This is equivalent to zero.

Array size is not kept so you must keep it yourself.

sizeof gets size of the type passed in and not the length of the array.



2 parts of static: writable + read only

used to store global variables

instructions which your CPU executes

dynamically allocated memory which persists beyond a function call.

Memory allocated in functions, passes args to functions. Contains return values & return address.

This is the data from pp dereferenced

&pp = 0xF9320900
pp = 0xF9320904
*pp = 0xF93209AC

dereference the address 0xF93209AC which is 0x2A.

pp evaluates to 0xF9320904. *pp evaluates to 0xF93209AC. **pp evaluates to 0x2A.

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;
}
```

This is to "pop" off first elm & set next item in array

Tail case returns 0.

gets sum of the rest of the elm

gets first elm in arr if arr has elm

Returns the sum of the first N elements in arr. ← this is equivalent to:

```
if (n){
  return arr[0]+foo(arr+1, n-1)
} else {
  return 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.

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

← add 1 to sum if item in arr is 0.

invert & add one. This is twos complement inversion!

Returns -1 times the number of zeroes in the first N elements of arr.

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

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

$y = x' \wedge y$

$y = x \wedge \underbrace{y \wedge y}_{0}$

$y = x$

$x'' = x' \wedge y'$

$x'' = x' \wedge x' \wedge y$

$x'' = y$ so $y = x$ & $x = y$

Ultimately does not change the value of either x or y.
this is because x & y were changed only in the function & not globally.

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

```
x == y
```

Truth Table

A	B	xor	xnor
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

So A must be the same as B.

Excersize: How would you make it so it affected them globally? Answer: make x & y pointers & edit the dereferenced items.

2 Programming with Pointers

2.1 Implement the following functions so that they work as described.

(a) Swap the value of two ints. *Remain swapped after returning from this function.*

```
void swap(int *x, int *y) {
    int temp = *x;
```

← need to store a temp int so that when we write to *x, we still have its value.

Note: temp only has to be an int since x is an int pointer & *x dereferences the pointer so it returns an int.

```
    *x = *y;
    *y = 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;
}
```

This is equivalent to *(str ++)

There is a table online with operator precedence.

note

x++ (postincrement): temp = X; x += 1; return temp.

++X (preincrement): X += 1; return X

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`.

Excersize: What is another method we could use to determine the length/end of an array? (Hint: there is a downside). Hint: Think about strings.

It is necessary to pass a size alongside the pointer.

Answer: Add some null byte to signify end. Drawback is you lose one integer you could have used.

To fix this issue, we need to pass in the size of the array.

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

was not there

was sizeof(summands)

sizeof() returns the size of the type, since summands is an int pointer, on a standard 32 bit system this would be 4 B. (Aka sizeof(int*)==4). There is an edge case where sizeof can get the length in bytes of an array: when the compiler defined the array size, It can optimize the correct value in the n.

(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.

The ends of strings are denoted by the null terminator rather than n. Simply having space for n characters in the array does not mean the string stored inside is also of length n.

was i < n

This is because the null terminator '\0' == 0.

```
void increment(char* string) {
    for (i = 0; string[i] != 0; i++)
        string[i]++; // or (*(string + i))++;
}
```

was:

does same thing

0xFF = 1111 1111
+ 0000 0001
→ 1 0000 0000 (gets dropped)
so 0000 0000
this => 0x00 == '\0'

Another common bug to watch out for is the corner case that occurs when incrementing the character with the value 0xFF. Adding 1 to 0xFF will overflow back to 0, producing a null terminator and unintentionally shortening the string.

This means you need to check for null before incrementing.

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

```
void copy(char* src, char* dst) {
```

```
    while (*dst++ = *src++);
}
```

remember
*dst++ means
temp = dest
dst += 1
return *temp

So this copies each elm to next arr. Common errors are students confusing this with *++dst = *++src which would skip first elm & go out of bounds by 1.

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`.

length of (61C_is_awesome!\0) = 16

```
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++;
    }
}
```

in static memory

`char *srcptr, replaceptr` initializes a `char` pointer, and a `char`—not two `char` pointers.

The correct initialization should be, `char *srcptr, *replaceptr`.

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**.

Ex Fn:
int x=0 ← global variable
void foo(){
  int y=x; ← y is a local variable.
  x++;
  char *static = "Hello!";
  char stack[] = "CS61C";
this is a pointer to read only static data
this is a pointer to a part of the stack.

(a) Static variables

Static

(b) Local variables — Function variables

Stack

(c) Global variables — program variables

Static

(d) Constants

Code, 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.

→ ex. add a0 a0 1
a0 = x which is a variable
constant in Assembly
Note a0 == register in CPU

```c
#define y 5
```
← pre-processor macro

```c
int plus_y(int x) {
    x = x + y;
    return x;
}
```
x is local variable (stack)

y is just 5 which is changed at compile. It is NOT available once compiled.

Constants can also be found in the stack or static storage depending on if it's declared in a function or not.

read only

```c
const int x = 1;
```
(Same as int const x=1;)

```c
int sum(int* arr) {
    int total = 0;
    ...
}
```

In this example, `x` is a variable whose value will be stored in the static storage, while `total` is a local variable whose value will be stored on the stack. Variables declared **`const`** are not allowed to change, but the usage of **`const`** can get more tricky when combined with pointers.

aka where it points to is the same but the data there can change depending on where it is stored/what parameters it was stored with.

(e) Machine Instructions

Code (text)

(f) Result of `malloc`

Heap

other things which allocate heap = calloc, realloc, malloc. Free can free any of these.

(g) String Literals

Note: they ALL return a pointer to the location on the heap where the data is stored. If it returns NULL, then it could not allocate any more memory. DON'T FORGET NULL CHECK for any alloc!

Static or stack.

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.

Also realloc may or may not use the same location in memory!

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

(a) An array `arr` of k integers

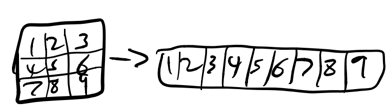
`arr = (int *) malloc(sizeof(int) * k);`

To make it computable w/ all systems. If you put just 4, it would be only computable with systems where sizeof(int) == 4 which is not generally true

(b) A string `str` containing p characters

`str = (char *) malloc(sizeof(char) * (p + 1));` Don't forget the null terminator!

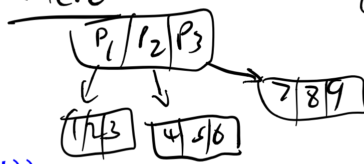
(c) An $n \times m$ matrix `mat` of integers initialized to zero.



`mat = (int *) calloc(n * m, sizeof(int));` ← linear array where

Alternative solution. This might be needed if you wanted to efficiently permute the rows of the matrix.

where:



Could do same but store rows. Different methods useful in different types of accesses.

```
mat = (int **) calloc(n, sizeof(int *));
for (int i = 0; i < n; i++)
    mat[i] = (int *) calloc(m, sizeof(int));
```

3.3 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";
    return important_stuff;
}
```

what happens if we have more than 63 characters? we may override the stack!

If the user input contains more than 63 characters, then the input will override other parts of the memory! (You will learn more about this and how it can be used to maliciously exploit programs in CS 161.)

Note that it's perfectly acceptable in C to create an array on the stack. It's often discouraged (mostly because people often forget the array was initialized on the stack and accidentally return a pointer to it), but there's it's not an issue in and of itself.

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;
}
```

3.4 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* item = (struct ll_node*) malloc(sizeof(struct ll_node));
    item->first = value;
    item->rest = *lst;
```

makes new struct ll_node in the heap

puts value to newly created structure.

sets rest to current start.

```
    *lst = item; ← Sets start to newly created & new setup structure
}
```

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

```
void free_ll(struct ll_node** lst) {
    if (*lst) { ← Checks to see if has actual node & not null.
        free_ll(&((*lst)->rest)); ← recursively frees the rest structure.
        free(*lst); ← frees current structure
    }
    *lst = NULL; // Make writes to **lst fail instead of writing to unusable memory.
}
```

Remember since this is a recursive call it will free ALL structs in the linked list.