C Introduction, Memory

CS 2130: Computer Systems and Organization 1

Xinyao Yi Ph.D.
Assistant Professor

UNIVERSITY of VIRGINIA | ENGINEERING

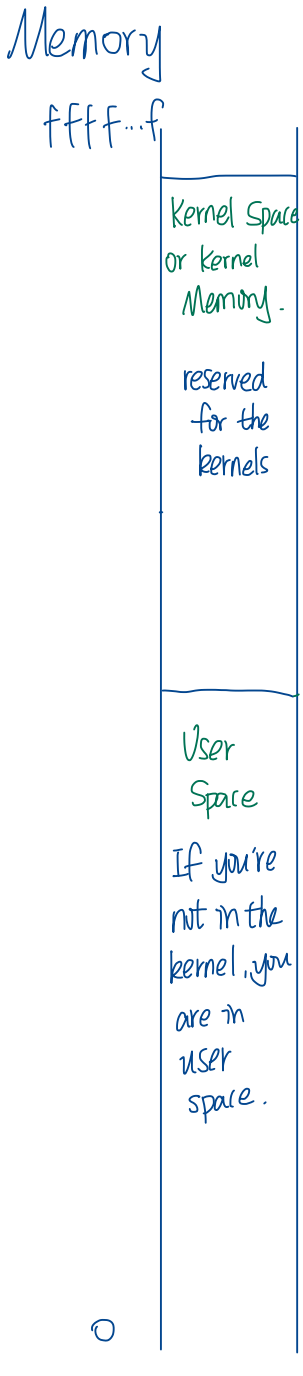
Announcements

- Homework 8 due Monday on Gradescope

Left over from the last class....

- The header file example
- Variadic functions
- man page

Memory



From assembly:

① Generate a memory address and go to memory.

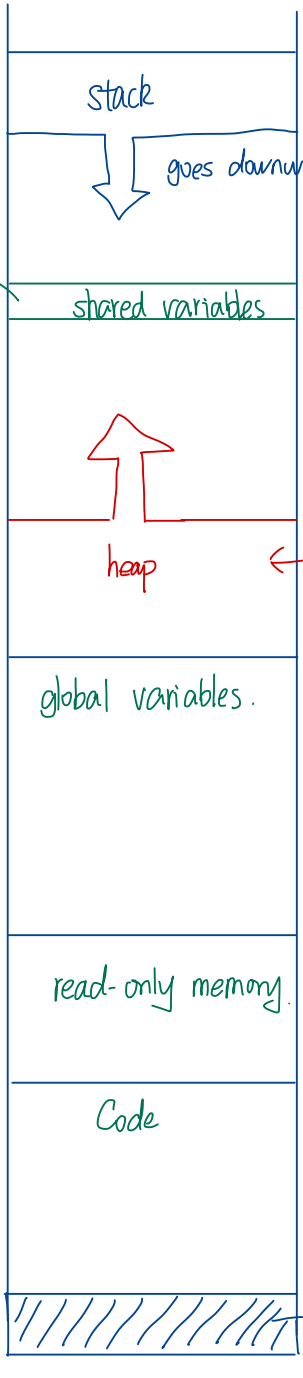
② hardware is going to check if we're actually allowed to read/write from that place in memory. (rely on a piece of software).

2 names for that: kernel or OS (operating system).

↳ help the hardware decide whether these addresses are good or not.

It divides memory up into a couple of different segments. a chunk of memory

It's going to have different properties based on what segment of memory I'm in.



stack frame:

Anytime I call a function, there are a couple of things that get put on the stack: ① parameters
② return address.
③ local variables.

When I return, RSP moves away, memory does not get erased.

One thing missing: pointers, variables that we want to leave when our functions return. (pass to another function?)

In C, I can define variables outside of functions
For local variables in functions: put values in stack.
But for global variables, we put them here. (The size of the variables are set at compile time, and the size cannot be changed.)

Example: string literals
Things are not code, but that were in my code that are really helpful for my program.

My code is going to be stuck in memory somewhere near the bottom.

We put the code near the bottom, but not at the bottom. (Explanation next page).

At the very bottom of memory — near address 0 — there's a special protected region.
The operating system doesn't let us read or write anything there.
Why? To protect us from ourselves.

If we accidentally take an integer, cast it to a pointer, and try to access that address — for example, address 5 — the hardware will stop us.
That's why we get a segmentation fault: we tried to read or write from a memory segment we're not allowed to touch.

In C, the constant NULL is not a keyword but is usually defined as a pointer to address 0.
This means that dereferencing a null pointer — trying to access the memory at address 0 — will also trigger a segmentation fault.
And that's intentional.
We want the system to complain immediately instead of silently reading some random value from the bottom of memory.

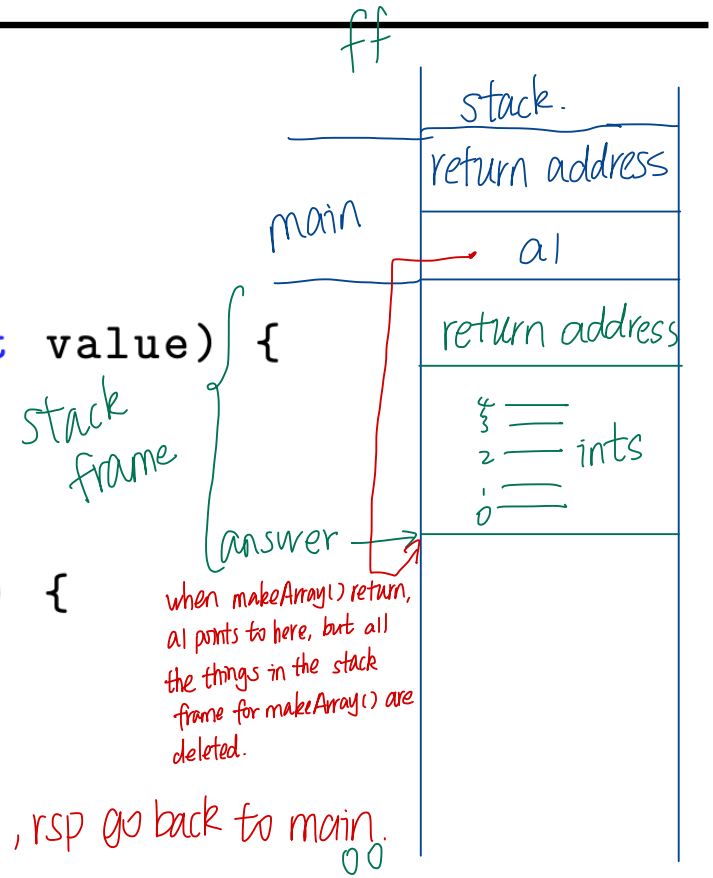
So, the "bottom of memory" is deliberately left empty to catch common pointer mistakes —
for example, dereferencing NULL or casting an integer like 5 to a pointer and using it as an address.
When that happens, the operating system stops the program with a segmentation fault rather than letting it continue with undefined behavior.

In short:

The low-address region exists as a safety barrier — it's the system's way of saying,
"You're trying to access memory that doesn't belong to you."

An Interesting Stack Example

```
int *makeArray() {
    int answer[5];
    return answer;
}

void setTo(int *array, int length, int value) {
    for(int i=0; i<length; i+=1)
        array[i] = value;
}

int main(int argc, const char *argv[]) {
    int *a1 = makeArray();
    setTo(a1, 5, -2);
    return 0;
}
```



once makeArray returns, remove everything, rsp go back to main.

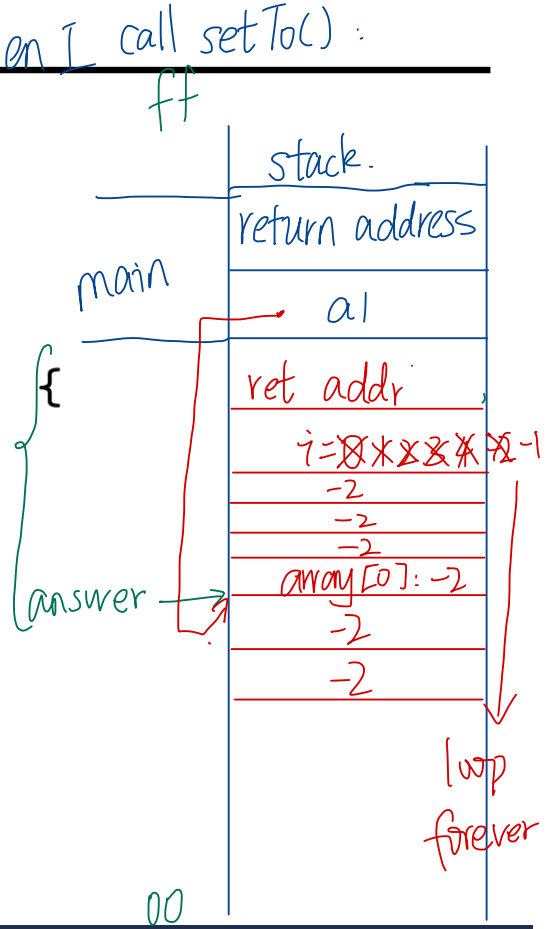
An Interesting Stack Example

```
int *makeArray() {
    int answer[5];
    return answer;
}

void setTo(int *array, int length, int value) {
    for(int i=0; i<length; i+=1)
        array[i] = value;
}

int main(int argc, const char *argv[]) {
    int *a1 = makeArray();
    setTo(a1, 5, -2);
    return 0;
}
```

escaping pointer

in registers

stack frame



if I compile it using clang xxx.c, then it will loop forever.

But if I compile it with "-O1". same warning, but it optimize the code maybe put "i" in register, maybe do unrolling?

The Heap

The heap is a section of memory that's different from the stack.
Anything you put on the heap will persist past function calls and returns — it stays there until you explicitly free it.
That means you can allocate space in one function, pass a pointer to another function, and still use that same memory later.
It's a part of memory that your program is allowed to use for dynamic data that needs to live longer than a single function call.

The heap: unorganized memory for our data

- Most code we write will use the heap
- *Not a heap data structure…*

Early computer scientists decided "heap" was a great name for this memory space —
but later someone also thought "heap" would be a great name for a data structure.
So remember: the heap in memory is not a heap data structure.
They just happen to share the same name.
You'll probably meet the other heap again in your Data Structures course.

The Heap: Requesting Memory

The way we ask for memory on heap is with a function called malloc.

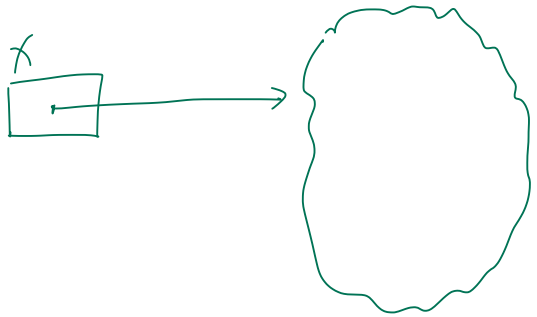
```
void *malloc(size_t size);
```

- Ask for `size` bytes of memory
- Returns a `(void *)` pointer to the first byte
- It does not know what we will use the space for!
- Does not erase (or zero) the memory it returns

Java

What is the closest thing to malloc in Java?

```
MyC x = new MyC();

int[]y = new int[100];
```



malloc **man page**

man malloc

calloc and realloc

malloc **Example**

```
typedef struct student_s {
    const char *name;
    int credits;
} student;

student *enroll(const char *name, int transfer_credits) {
    student *ans = (student *)malloc(sizeof(student));
    ans->name = name;
    ans->credits = transfer_credits;
    return ans;
}
```

(*ans).name

if I don't free → run out the memory, things get really slow.

The Heap: Freeing Memory

Freeing memory: `free`

```
void free(void *ptr);
```

We will use the pointer to free the space. If I change the pointer, I will lose the information of where to free.

- Accepts a pointer returned by `malloc`
- Marks that memory as no longer in use, available to use later
- You should `free()` memory to avoid *memory leaks*

When my program ends, the operating system will come in and free the memory for me.

An Interesting Stack Example

```
int *makeArray() {
    int answer[5];
    return answer;
}

void setTo(int *array, int length, int value) {
    for(int i=0; i<length; i+=1)
        array[i] = value;
}

int main(int argc, const char *argv[]) {
    int *a1 = makeArray();
    setTo(a1, 5, -2);
    return 0;
}
```

int * answer = malloc(5 * sizeof(int));

int * answer = (int *) malloc(5 * sizeof(int));
↳ Better! But if you don't have it, it's fine.
because malloc returns a void*. it will automatically cast to int*.

free(a1);