Tuesday 9/18
CS61 Notes by: William Hakim, Jenny Ye, and Aaron Graham-Horowitz

I. Storage Duration

STATIC DURATION
Memory allocated for variables of static duration is allocated for as long as the program runs. The memory is allocated in data. The compiler and OS decide where to allocate this memory before execution.
Examples: globals, program code

AUTOMATIC DURATION
Memory allocated for variables of automatic duration is allocated for the duration of a function or a block (the space between curly braces). This space can be reused after the function or block finishes executing. A variable with automatic duration is recreated every time the function is called. For instance, a recursive function will have an instance of each of its local variables for each of its stack frames.
Examples: locals

DYNAMIC DURATION
Why do we need it? - Suppose we want to store something of arbitrary size (for example, a dictionary file). We do not want to reread in the dict file every time we call a function (i.e. automatic duration - this is very slow), but we also do not know how much memory to preallocate for the dict file (i.e. static duration - either very wasteful if too much room is allocated or causes an error if too little room is allocated). We need another way to store something permanently whose size depends on the user's input.

With dynamic storage duration (in C) the memory is entirely managed by the user rather than the compiler or OS. We use the function malloc to dynamically allocate memory.

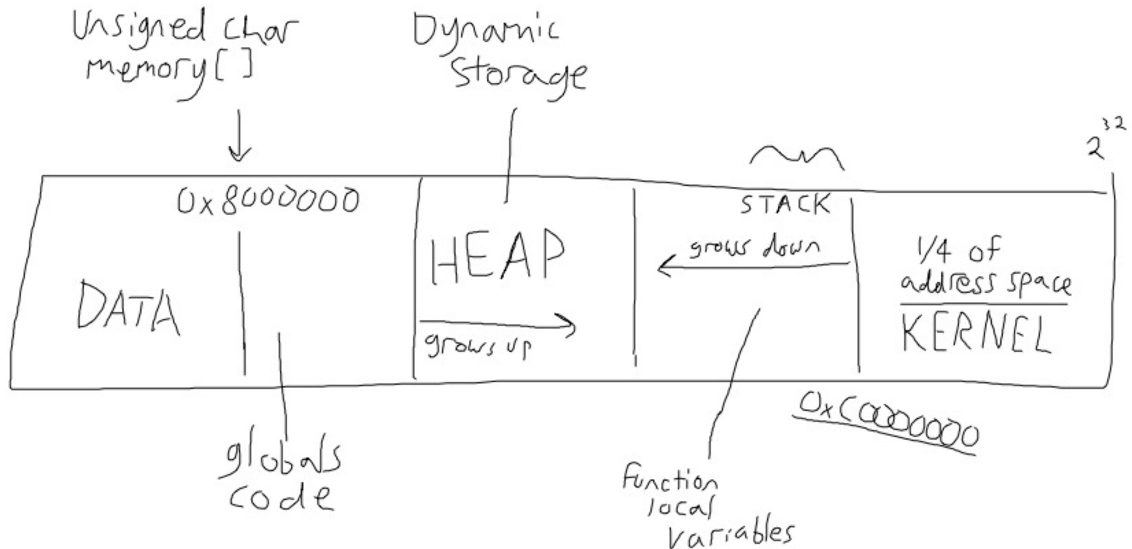
void *malloc(size_t sz) → allocates sz bytes of memory. This memory is guaranteed to not overlap with anything else.

void free(void *ptr) → frees dynamically allocated space dereferenced by ptr for future reuse.

In order to use dynamic storage, we need a pointer (stored with automatic or static duration) to point to the memory dynamically allocated. Objects of dynamic duration for which no pointer exists are called garbage. Garbage collection is the process of finding and freeing garbage. Objects of dynamic storage duration are stored on the heap.

Examples: memory allocated with malloc, calloc or realloc by the user

9/18/12

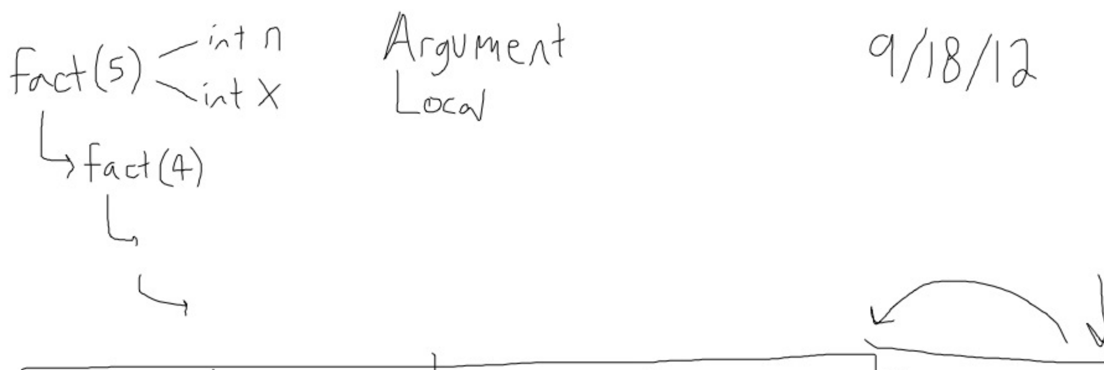


II. How is a Recursive Function Handled in Memory?

1. Create a stack allocating all of the necessary memory
2. For each iteration (call to) a function, we need to remember the arguments to and the local variables of that function. We can create a struct which holds these like this:

```
struct fn_args_locals {
        one element for each argument
        one element for each local variable
};
```

3. Within the stack, we create enough space to hold one of these structs, initiate the arguments, and then move the stack pointer to the end of the struct. We then repeat for the next iteration of the function. When we get to the base case, we initialize the locals and the arguments, and then “return” its value, i.e. return the stack pointer back to the beginning of the struct (back to the calling function), which can then initiate *its* locals and return to *its* calling function.
3. Whenever a single call to a function returns, its stack frame closes and it “restarts” the function that called it.
4. We need to be careful of stack overflows. The OS only allocates a certain amount of space for the stack to grow into. For instance, if we called a recursive factorial function on a negative number, we would be trying to generate an infinite number of stack frames which would ultimately result in this error.



III. Other Notes

- Approximately ¼ of addressable memory space is reserved for use by the OS. This space is called the kernel. The stack starts at approximately 0xC0000000 = ¾ of 2^32 (the size of memory on a 32-bit machine) and grows down by convention.
- Similarly, the heap starts closer to data and grows up.
- Strings of length n are of size n+1 bytes (each string has a "\0" null terminator)
- Virtual memory: every program runs as if it's the only program on the machine

- Unions are a way to get around C's type system. For instance, in the following union we are telling the compiler to treat the memory allocated to foo as either an array of chars or an integer:

```
union foo {
        char array[4];
        int i;
};
```

- We need to distinguish between big and little endian representations of bits in memory. The same memory can be read two different ways depending on this.