C Review/Crash Course

- We're assuming you already know C++.
- C is a subset of C++, so they're more similar than different.
- In this class, we'll use the `gcc` compiler with the C99 standard:

```
gcc --std=c99 -o program_name program_name.c
```

Basics

- The basic C program template (basically just like C++):

```
int main(int argc, char** argv) {
    return 0;
}
```

- What if we want to do something more? Let's print the arguments:

```
#include <stdio.h>

int main(int argc, char** argv) {
    int i;
    for (i = 0; i < argc; i++) {
        printf("argv[%d]: %s\n", i, argv[i]);
    }
    return 0;
}
```

Printing to stdout: `printf()`

- `printf()` is the C way of printing stuff to `stdout`; there are no streams in C.
- `printf()` uses a format string as a template for the output.
  - Format specifiers, designated by `%`, act as placeholders into which additional `printf()` arguments are inserted.
    - The i'th `printf()` argument goes to the i'th format specifier.
  - In general, the character after the `%` indicates the type of the argument to be printed
    - `%d` – `int`, as a signed decimal number
    - `%f` – `double`, in fixed-point notation (e.g. `3.1415`…)
      - `float` arguments are cast as `double`
    - `%c` – `char`, as a character
    - `%s` – a null-terminated string
    - These are just a few, there's lots more you can do with the format specifier: https://en.wikipedia.org/wiki/Printf_format_string#Format_placeholder_specification

Functions

- Functions work mostly like they do in C++ (except they're all at file scope: no class methods):

```
#include <stdio.h>

void foo(int x) {
    printf("foo got %d\n", x);
}
```

```
int main(int argc, char** argv) {
    foo(2);
}
```

- Just like C++, you can prototype a function first (e.g. in a header file) and define it later:

```
#include <stdio.h>

/* This could be in a separate .h file too */
void foo(int);

int main(int argc, char** argv) {
    foo(2);
}

/* This could be in a separate library .c file */
void foo(int x) {
    printf("foo got %d\n", x);
}
```

- C has no reference types, unlike C++! That means functions are all pass-by-value.
    - In other words, a copy of each argument is passed to the function as a parameter.
    - Passing big structures can be a problem with small stack sizes.
    - More under “pointers” below.

- C is procedural, not object oriented like C++.
    - Operate on data by passing it as arguments to functions.
    - No classes or class methods.
    - Structured data represented with `struct`.

- For example, we might do something like this in C++:

  ```
  Student s = new Student("Luke Skywalker");
  s.print();
  ```

- In C, we'd do something like this:

  ```
  struct student s = {.name = "Luke Skywalker"};
  print_student(s);
  ```

Structures

- Here's what the definition of our student `struct` might look like:

```
struct student {
    char* name;
    int standing;
    float gpa;
};
```

- We can initialize a new struct with a *designated initializer* :

```
struct student s = {.name = "Luke Skywalker",
    .gpa = 4.0};
```

  - Note that we don't need to initialize all fields when using a designated initializer. Uninitialized fields will be zeroed.

- We can access or update a field in a `struct` using the `.` operator:

```
s.standing = 2;
printf("%d\n", s.standing);
```

Pointers

- A pointer is a variable whose value is a memory address.
    - For example:

```
int i = 20;
int* i_ptr = &i;

printf("%d\n", i);  // Prints 20.
printf("%p\n", i_ptr);  // Prints 0xffffcc04
printf("%d\n", *i_ptr);  // Prints 20.

*i_ptr = 8;
printf("%d\n", i);  // Prints 8.
printf("%p\n", i_ptr);  // Prints 0xffffcc04
 printf("%d\n", *i_ptr);  // Prints 8.
```

    - A few important things here:
        - `*` acts as both a pointer variable designator (`int*`) and as the dereference operator (`*i_ptr`).
            - The dereference operator gives you the *value* at an address.
        - `&` is the “address-of” operator.

- C, unlike C++, has no pass-by-reference, but we can achieve the same thing with pointers:

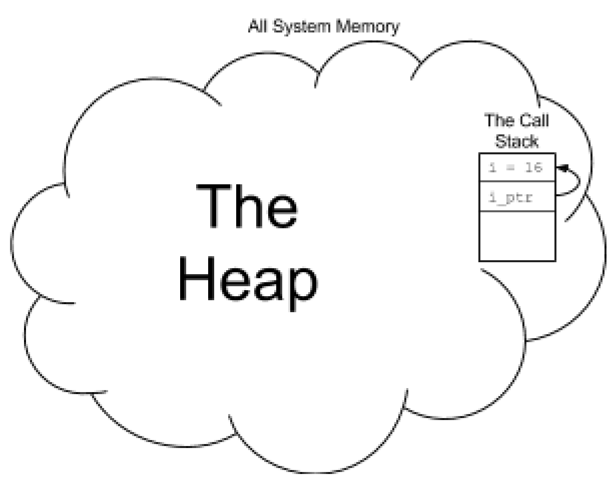
```
/* Takes an *address* of an int */
void make_it_32(int* a) {
    *a = 32;
}
```

```
int main() {
    int i = 6;
    make_it_32(&i);  // Pass the *address* of i.
    printf("%d\n", i);  // Prints 32.
}
```

Program memory: the call stack vs. the heap

- A running C program (or a program in any language, for that matter) has two separate areas of memory in which it can store data, the call stack and the heap.
  - **The call stack** is a small, limited-size chunk of memory from the larger blob of system memory. Among other things, the values of variable declared in a program's functions are stored on the call stack.
    - The call stack is small: usually at most 8kb.
  - **The heap** comprises essentially all the rest of system memory.
    - A program must specifically request to allocate memory from out of the heap.
    - The heap is huge compared to the call stack.
  - Here's what this might look like for a simple program:

```
int main() {
    int i = 16;
    int* i_ptr = &i;
}
```



Allocating memory on the heap: `malloc()`

- The call stack is small (e.g. only 2048 4-byte integers fit on an 8kb stack). We need to be able to work with more memory.

- We'll need to start allocating from the heap.

- You use `malloc()` to allocate memory from the heap in C:

  ```
  void* mem_block = malloc(NUMBER_OF_BYTES);
  ```

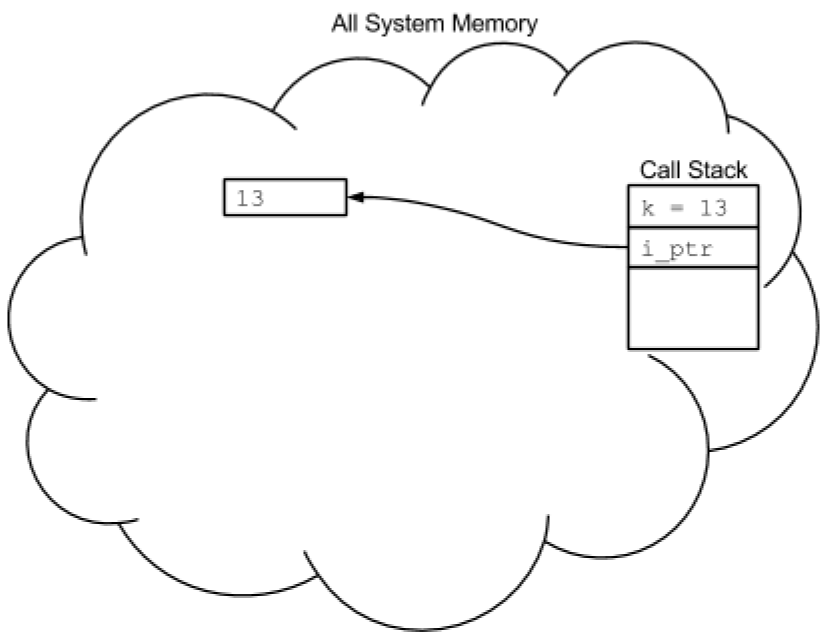
- A few things to note:
  - Must `#include <stdlib.h>`
  - `malloc()` allocates a *contiguous* block of memory.
    - This is useful for allocating arrays.
  - `malloc()` returns a pointer of type `void*`.
    - This can be cast to any type.
  - We need to know how many bytes we need to allocate.

- We use `sizeof()` to help figure out how many bytes to allocate, e.g.:

  ```
  printf("sizeof(int): %d\n", sizeof(int));  //
  4
  ```

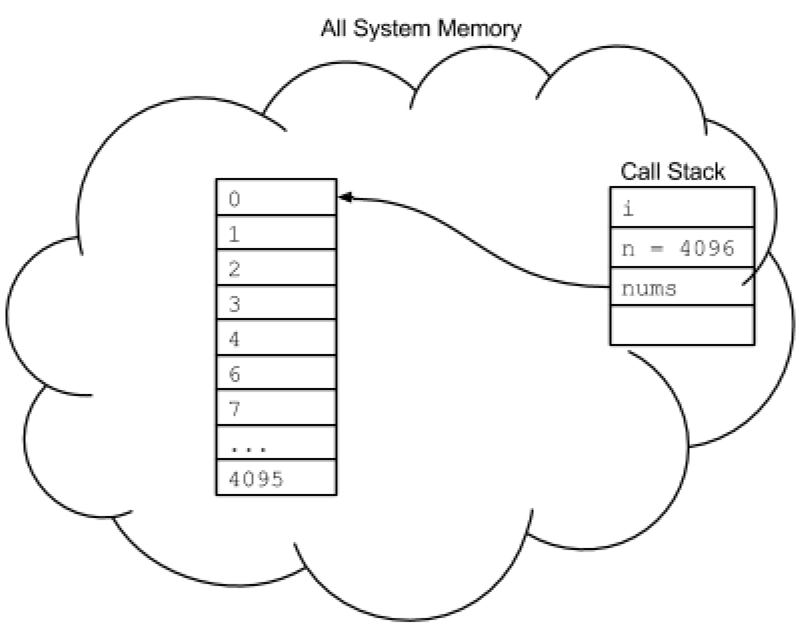
- We generally use `sizeof()` in conjunction with `malloc()`:

  ```
  int k = 13;
  int* i_ptr = (int*)malloc(sizeof(int));
  *i_ptr = k;
  printf("*i_ptr: %d", *i_ptr);  // Prints 13.
  ```



- We can use `malloc()` and `sizeof()` to allocate arrays:

```
int i, n = 4096;
int* nums = (int*)malloc(n * sizeof(int));
for (i = 0; i < n; i++) {
    nums[i] = i;
}
```



- We can iterate through arrays using pointers too!

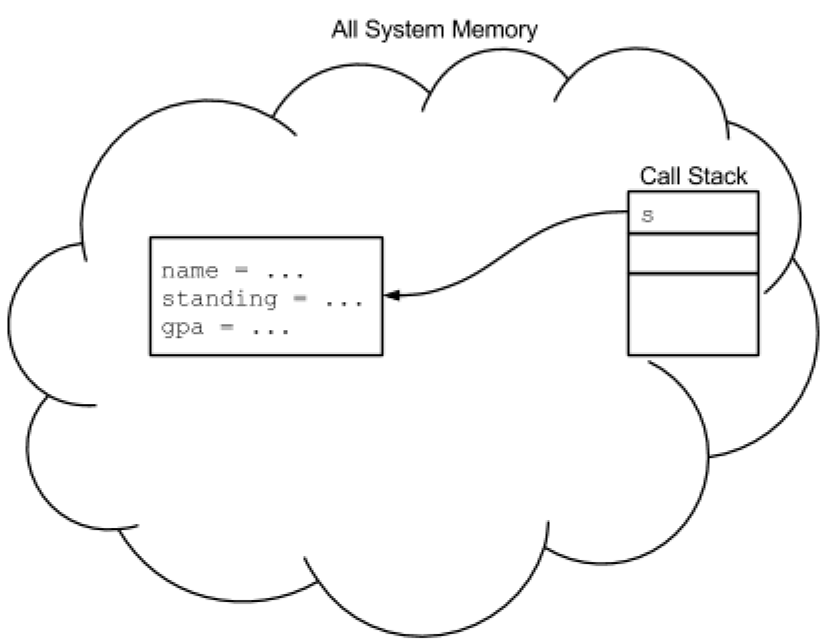
```
int i, n = 4096;
int* nums = (int*)malloc(n * sizeof(int));
int* cur;
for (i = 0, cur = nums; i < n; i++, cur++) {
    *cur = i;
}
```

  - The same would work if `nums` was a (smaller) array on the call stack.

`malloc()` and `struct`

- We can use `malloc()` with any `struct` just like we do for C's standard types to get a pointer to a `struct`:

```
struct student s* = malloc(sizeof(struct
student));
```



- When we have a pointer to a `struct`, we need to dereference the pointer to access the `struct`'s fields:

```
(*s).name = "Luke Skywalker";
(*s).gpa = 4.0;
```

- That's so much typing just to get to one field! Luckily C has the `->` operator that both dereferences a `struct` pointer *and* accesses one of its fields:

```
s->name = "Luke Skywalker";
s->gpa = 4.0;
```

- We allocate an array of structs just like an array of any other type:

```
int i, n = 96;
struct student* students =
    malloc(n * sizeof(struct student));
for (i = 0; i < n; i++) {
    students[i].name = ...;
    students[i].standing = ...;
    students[i].gpa = ...;
}
```

- Something to notice: `students[i]` is a dereferenced memory address (a pointer).
    - That's why we use the `.` operator to access the fields of `students[i]`.

- You can also use *memory arithmetic* to access the elements of an array:

```
(students + 13)->name = "Luke Skywalker";
```

- We see here that `(students + 13)` is the same thing as `&students[13]`.

- This can allow you to factor your code nicely, something like this:

```
void init_students(struct student* s, char* name,
    float gpa) {
    s->name = name;
    s->standing = 1;
    s->gpa = gpa;
}

...
for (i = 0; i < n; i++) {
    init_student(students + i, ...);
}
```

Freeing `malloc()`'ed memory

- We ***MUST*** free all of the memory we allocate using `malloc()`, otherwise our program will have memory leaks.

- To free memory, we use `free()`.

- A good rule of thumb is this: For every call to `malloc()` you should have a call to `free()`.

- To use `free()`, just pass the pointer that was returned by `malloc()`, no matter whether that pointer represents a single item, an array, a struct, or whatever, e.g.:

```
int* i = malloc(sizeof(int));
...
free(i);

int* nums = malloc(1000000 * sizeof(int));
...
free(nums);

struct student* s = malloc(sizeof(struct
student));
...
free(s);

struct student* students =
    malloc(1000 * sizeof(struct student));
...
free(students);
```

- `valgrind` is a great tool for helping debug memory issues.

- To run a program with `valgrind`, you need to compile it with debug flags using the `-g` option:

```
gcc --std=c99 -g prog.c -o prog
```

- The just pass your program to `valgrind`:

```
valgrind ./prog [args to prog]
```

- `valgrind` will run your program and detect memory leaks. If you have any leaks, it will let you know that some memory was “lost”.

- To dig deeper into where the memory was lost, use `valgrind --leak-check=full`:

  ```
  valgrind --leak-check=full ./prog [args]
  ```

- This will give you a report with the line numbers of the `malloc()` calls for the memory that was lost.

C strings

- In C, strings are just arrays of characters, but there are some things to be aware of.

- This doesn't do what you might expect:

  ```
  char* str = "foo";
  ```

- That actually allocates static, read-only memory, so that you couldn't later do this:

  ```
  str[2] = 'x';  // You can't do this.
  ```

- If you initialize a string that way, you really should use `const`:

  ```
  const char* str = "foo";
  ```

- Otherwise, you can allocate a string just like any other array:

  ```
  int n = 64;
  char* str = malloc(n * sizeof(char));
  ```

- Then, you have lots of functions available to help you get data into that string, e.g.:

```
#include <string.h>
strncpy(str, src_str, n);  // Copy up to n chars.

#include <stdio.h>
snprintf(str, n, "%s %d", some_str, some_int);
    // Use a format string to initialize.
```

- Check out string.h for more string functions.

Segmentation faults and debugging with `gdb`

- A segmentation fault (segfault) is a memory access violation.
  - i.e. when a program tries to access a memory location in a way that's not allowed.

- Working with pointers, arrays, and memory allocation, you *will* have segfaults.
  - Common causes are trying to dereference a `NULL` pointer or an uninitialized pointer, e.g.

    ```
    struct student* s;
    s->name = "Luke Skywalker";   // Woops!  Didn't
                                  // allocate s.
    ```

- `gdb` is a great tool for helping debug segfaults.

- To make `gdb` most useful, you should use the `-g` flag to compile with debugging symbols:

```
gcc --std=c99 -g prog.c -o prog
```

- Start `gdb` like you were running the program, adding `gdb` to the front:

```
gdb ./prog
```

- Here's an example `gdb` session (the prompts start with `(gdb)`):

```
scratch — @e36bcefa4dc9:/scratch — docker run -it --rm -v ~/courses/cs261/scratch:/scratch --security-opt seccomp:unconfined centos /bin/bash — 80×24
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /scratch/prog...done.
(gdb) run
Starting program: /scratch/./prog

Program received signal SIGSEGV, Segmentation fault.
0x00000000004004ff in main (argc=1, argv=0x7fffffffe7d8) at scratch.c:13
13        s->name = "Luke Skywalker";
Missing separate debuginfos, use: debuginfo-install glibc-2.17-106.el7_2.8.x86_6
4
(gdb) break 13
Breakpoint 1 at 0x4004fb: file scratch.c, line 13.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /scratch/./prog

Breakpoint 1, main (argc=1, argv=0x7fffffffe7d8) at scratch.c:13
13        s->name = "Luke Skywalker";
(gdb) print s
$1 = (struct student *) 0x0
(gdb) 
```

- Here's what we see:
  - We run the program.
  - We see that it fails with a segfault at line 13, where we're assigning to `s->name`.
  - We set a breakpoint at line 13.
  - We run the program again.
  - The program stops at our breakpoint at line 13 before executing line 13 (i.e. before the segfault occurs).
  - We print the value of `s` and see that it is `0x0`, or `NULL`.
  - We realize we were trying to dereference a `NULL` pointer.