C Basics

- In this course, we'll use the C programming language.
    - Specifically, we'll use the C99 standard of the C language.

- We'll assume you already know C++. Since C is a subset of C++, this means making the jump from C++ to C shouldn't be too hard. Still, there are some key differences between the way things are done in C++ and the way they're done in C, so we'll start the course by covering some basics of the C language.
    - If you don't already know C++, you should use a resource like the C tutorial on cprogramming.com to provide a more in-depth look at some of the C features we'll gloss over here (like loops, functions, if/else statements, etc.).

- Importantly, for this course, we'll use the GCC C compiler, which is installed on all of the ENGR machines. The GCC C compiler (`gcc`) works very much like the GCC C++ compiler (`g++`), which you've likely used. For example, here's how to use the GCC C compiler to compile (under the C99 standard) a single C file called `main.c` to produce an executable file named `main`:

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

- Let's start by reviewing some essentials.

C Program structure

- The basic C program template is just like in C++. Specifically, every program needs a function called `main()`, which serves as the entrypoint into the program. This file typically returns an integer and accepts arguments describing the number of arguments passed on the command line (`argc`) and providing the string values of those command line arguments (`argv`; we'll talk more later about how strings are represented in C):

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

- C programs will typically also contain one or more include statements at the top of the file. Like in C++, these incorporate external code into the current file. Unlike in C++, we need to specify the complete name of the file being included (including the file extension) in all include statements.
    - The standard file extension for header files in C is `.h`.

- For example to include functionality for writing text to and reading it from the console (or a file), we would include the (built-in) standard I/O header:

```
#include <stdio.h>
```

Printing text to stdout with `printf()`

- One of the simplest things we'll want to be able to do in a C program is to print text to the terminal (i.e. to the standard output stream, or stdout). This is done differently in C than in C++. Specifically, in C, we use the function `printf()` to print text to stdout:

```
printf("This is a string I'm printing to stdout.\n");
```

- In the example above, we're simply printing a constant string to stdout. Importantly, note that we explicitly include the newline (`\n`) character in this string to include a line break in the text printed to stdout. In general, we can print as many line breaks as we want by including multiple newline characters in the string passed to `printf()`.

- What if we want to do more than print a constant string? For example, what if we want to print the values held in some of our program's variables?

- We can do this by passing a ***format string*** and accompanying arguments to `printf()`.

- You can think of a format string as a template for the text to be printed. In particular, a format string will contain placeholders (known as ***format specifiers***) into which specific values will later be inserted.

- Each format specifier is designated by a `%` character, followed by at least one other character describing the data that will eventually be plugged in.

- For example, if we had an integer variable (i.e. an `int`) called `x` whose value we wanted to print, we could include a `%d` specifier (i.e. placeholder) in our format string and then, importantly, pass `x` as an additional argument to `printf()`:

```
int x = 8;
printf("This is the value of x: %d\n", x);
```

- There are lots of different kinds of format specifiers we can use:
  - `%d` – indicates an `int`, to be printed as a signed decimal number
  - `%f` – indicates a `double`, to be printed in fixed-point notation (e.g. `3.1415…`)
    - `float` arguments are cast as `double`
  - `%c` – indicates a `char`, to be printed as a readable character
  - `%s` – indicates a null-terminated string
  - Lots more…

- We can also print multiple values by inserting multiple format specifiers. Then, the i'th argument to `printf()` (after the format string) is plugged into the i'th format specifier, e.g.:

```
char* name = "Luke Skywalker";
double gpa = 3.75;
printf("%s's GPA is %f\n", name, gpa);
```

If statements

- For program control, C has several features that work basically the same way as they do in C++. If statements are one of these:

```
if (a == 0) {
  /* Do something. */
} else if (b != 0) {
  /* Do something different. */
} else {
  /* Do a third thing altogether. */
}
```

Loops

- Loops also work pretty much the same way in C as they do in C++. C has for loops:

```
int i;
for (i = 0; i < 32; i++) {
  /* Do something 32 times. */
}
```

- C also has while loops:

```
while (i != 16) {
  /* Do something repeatedly until i is 16. */
}
```

- Like C++, C also has do/while loops, though these are less commonly used.

Functions

- C also has functions that 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 was passed this argument: %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. This can be a problem if we're passing big structures around. We'll talk more about this later.

- The main difference between C and C++ is that C is procedural, not object oriented like C++.
    - This means that C has no classes or class methods. Instead, all operations on a given data entity are performed by passing that entity to a function.
    - As we'll discuss later, structured data is represented in C 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

- If we want to represent structured data in C, we typically use a struct type. For C++ programmers, a struct is much like a class with *only* public data members and *no* class methods.

- For example, here's what a definition of a struct for representing individual students might look like:

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

- We can initialize a new struct with what's known as a ***designated initializer***:

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

    - Note that, like in the example here, we don't need to initialize all fields of a struct with a designated initializer. Uninitialized fields will be zeroed.

- We can access or update a field in a struct using the `.` operator (assuming we have a struct *value*, like `s` here, and not a struct pointer; more on this in a bit):

```
s.id = 933111111;
printf("%d\n", s.id);
```

Pointers

- A pointer is a variable whose value is a memory address. Every pointer points to data of a specific type. For example, we can initialize a pointer to a single integer value like this:

```
int n = 20;
int* n_ptr = &n;
```

- In the example above, `n` is a normal integer value, and `n_ptr` is a pointer whose value is the memory address where `n` is stored (i.e. `&n`). In other words, `n_ptr` points to `n`.

- This becomes even more clear if we print out the values associated with these two variables:

```
printf("n: %d\n", n);
printf("n_ptr: %p\n", n_ptr);
```

    - Note here that we print the pointer value using a `%p` format specifier. This is used specifically for printing pointer values.

- The output of these two lines will look something like this:

```
n: 20
n_ptr: 0x7ffee5032958
```

- We can see here more explicitly that `n` represents the value of `n`, and `n_ptr` represents the *memory address* where `n` is stored.

- If we want to obtain the underlying value that's stored at a particular memory address, we must ***dereference*** the pointer using the `*` operator:

```
printf("*n_ptr: %d\n", *n_ptr);  /* Prints 20. */
```

- We can also dereference a pointer to *update* the value stored at the corresponding memory address, e.g.:

```
*n_ptr = 8;
printf("n: %d\n", n);
printf("n_ptr: %p\n", n_ptr);
printf("*n_ptr: %d\n", *n_ptr);
```

- This will print something like this:

```
n: 8
n_ptr: 0x7ffee5032958
*n_ptr: 8
```

- Note here that the underlying value has changed, but that value is stored at the same memory address (i.e. `0x7ffee5032958`).

- Importantly, 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. */
}
```

Void pointers (`void*`)

- In C, you'll often run into something called a ***void pointer***. This is a pointer represented by the type `void*`.

- A void pointer is a generic pointer. In other words, it can point to data of any data type. For example, we could make a void pointer that pointed to an integer:

```
int n = 32;
void* v_ptr = &n;
```

- If we tried to use any different pointer type, e.g. `float*`, to point to an integer value, the compiler would give us a warning. However, using a `void*` works just fine.

- We can use a void pointer to point to values of any other type, too, though:

```
float pi = 3.1415;
struct student s = { .name = "Luke Skywalker" };
v_ptr = &pi;  /* Totally valid. */
v_ptr = &s;   /* Totally valid. */
```

- Importantly, because there is no type information built into a void pointer, it can't be dereferenced directly. In other words, we *can't* do this:

```
struct student s = { .name = "Luke Skywalker" };
void* v_ptr = &s;
printf("%s\n", v_ptr->name);  /* Compile-time error: can't
                                 dereference void pointer
*/
```

- Instead, we must either assign the void pointer back to a pointer variable of the correct type or else cast it to the correct type and then dereference it.

- For example, here's how we could assign the void pointer `v_ptr` we created just above back to a pointer variable of the correct type (i.e. `struct student*`) in order to dereference it:

```
struct student* s_ptr = v_ptr;
printf("%s\n", s_ptr->name);
```

- And here's how we could just cast back to the correct pointer type first to dereference the void pointer:

```
printf("%s\n", ((struct student*)v_ptr)->name);
```

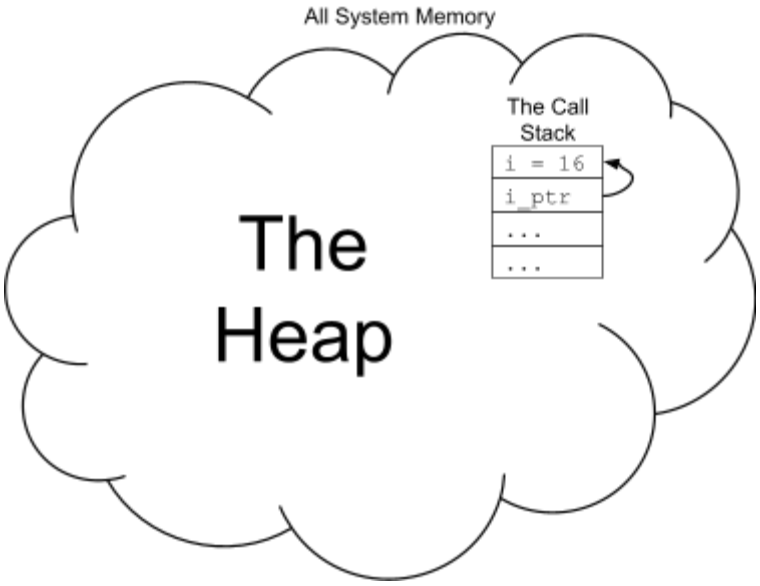
- You'll often see void pointers used in data structures implemented in C, since they allow the data structures to contain data of any type while remaining type agnostic.

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 variables 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 the heap. We'll see how to do this in C in just a second.

    - 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()`

- Again, the call stack is quite small (e.g. only 2048 4-byte integers fit on an 8kb stack), so we'll need some way to work with bigger chunks of memory. The heap allows us to do this.

- When we want to use memory from the heap, though, we need to explicitly ask to have it allocated. In C++, we did this using the `new` operator. Unfortunately, `new` is not available in C. Instead, we'll use the `malloc()` function.

- The `malloc()` function requires us to `#include <stdlib.h>`. It takes as its argument the *number of bytes* to be allocated from the heap and returns the allocated bytes as a void pointer, which can be cast as a pointer of any type:

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

- Importantly, `malloc()` allocates a *contiguous* block of memory. This means that the memory is allocated as a single, continuous chunk, rather than potentially in multiple pieces. This, for example, makes it easy to use `malloc()` to allocate an array.

- Again, memory allocated using `malloc()` lives on the heap. This is known as ***dynamically allocated memory***. It is allocated at *runtime*.
    - Memory allocated on the call stack, on the other hand, is known as ***statically allocated memory***. It is allocated at *compile time*.

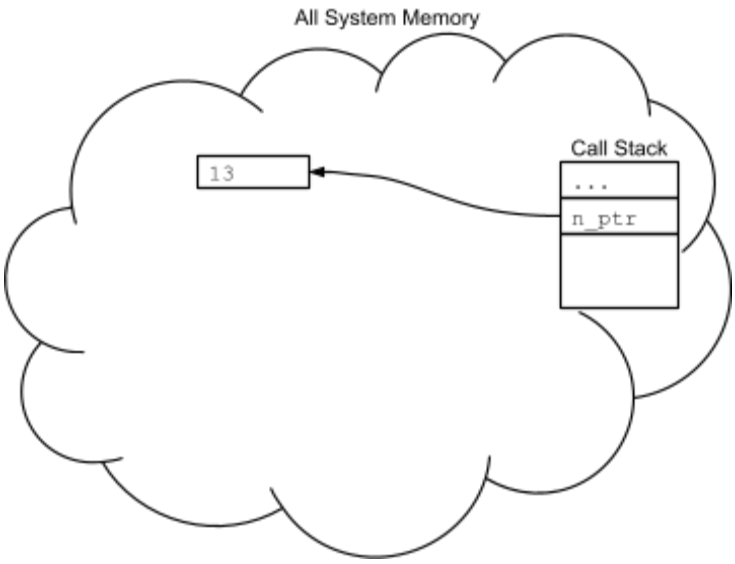
- You may be wondering how one might figure out how many bytes to allocate when using `malloc()`. The `sizeof()` operator is very useful for this. `sizeof()` returns the size, in bytes, of a given variable or data type. For example, if we wanted to know the number of bytes required for a single integer, we could use `sizeof()` like so:

  ```
  sizeof(int)
  ```

- Thus, to let `malloc()` know how many bytes to allocate from the heap, we can make use of `sizeof()`, e.g.:

  ```
  int* n_ptr = malloc(sizeof(int));
  *n_ptr = 13;
  ```

- This would look something like this within system memory:



- To allocate an array, we can simply multiply the value returned by `sizeof()` by the desired size of the array. For example, here's how we could allocate an array of 1000 integers on the heap:

```
int* array = malloc(1000 * sizeof(int));
```

- Once an array is allocated on the heap, it is used just like any other array:

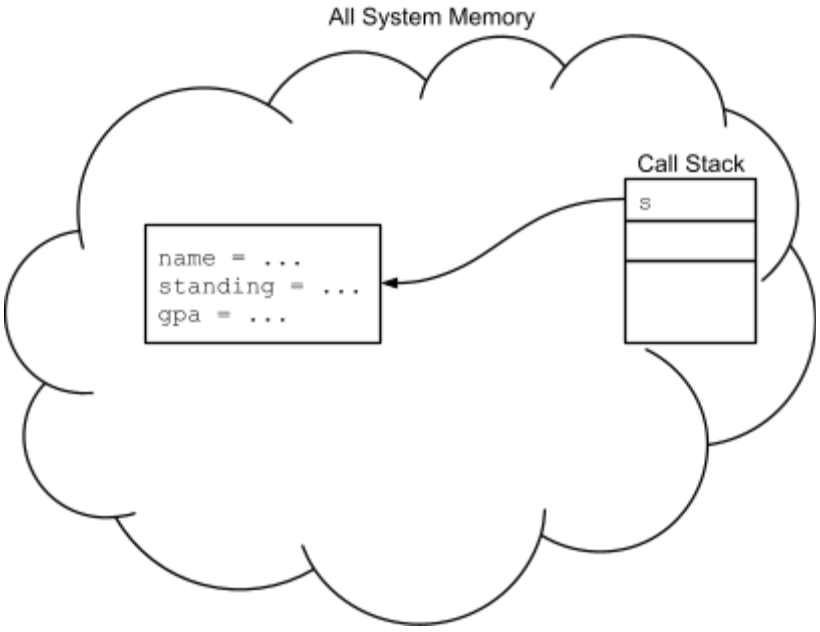
```
int i;
for (i = 0; i < 1000; i++) {
  array[i] = i;
}
```

`malloc()` and `struct`

- We can use `malloc()` with any `struct` just like we do for C's standard types to allocate a `struct` on the heap, e.g.:

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

- This would look something like this in system memory:



- 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, e.g.:

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

Freeing `malloc()`'ed memory

- It's incredibly important that we **ALWAYS** free all of the memory allocated using `malloc()`. If we don't our program will have what's called a ***memory leak***. Memory leaks can cause unpredictable, catastrophic failure of a program, so it's best to avoid them.

- Memory allocated with `malloc()` is always freed using `free()`:

```
int* array = malloc(1000 * sizeof(int));
...
free(array);
```

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

- `free()` works the same regardless of what kind of memory is being freed. In other words, whether a chunk of allocated memory holds a single value, an array, a struct, or whatever, we simply pass a pointer to that memory to `free()`, and it will be released.

Using valgrind to debug memory issues

- Memory issues, like leaks, are sometimes the most insidious ones to diagnose and debug. Fortunately, there are some useful tools available to us to help debug memory issues. valgrind is one of these tools.

- valgrind works both by inserting extra instructions into a program being executed and by replacing the standard system memory allocator with its own implementation. The combination of these modifications to the program being executed allows valgrind to closely monitor that program's memory usage and to detect several kinds of memory error.

- To make valgrind most useful, it is best to include debug flags in the program to be debugged. We can do this by passing the `-g` option to `gcc`:

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

- Once the program is compiled into an executable, we can run that program through `valgrind` on the command line by simply adding the `valgrind` command before our normal command-line invocation of our program:

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

- Valgrind will then execute your program, and, among other things, it will detect memory leaks. If you have any leaks, it will let you know that some memory was "lost".

- To dig deeper into where memory was lost, pass the `--leak-check=full` option to the `valgrind` command:

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

- If debug flags are compiled into the program being executed, the `--leak-check=full` option will tell valgrind to print a report with the line numbers of the `malloc()` calls for the memory that was lost. This is extremely helpful for tracking down allocated memory you may have forgotten to free.

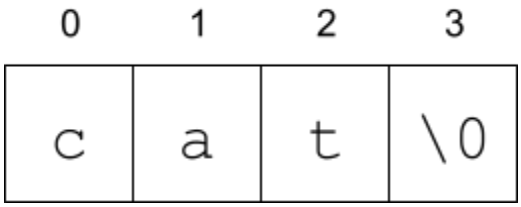
Strings in C

- In C, there is no special type for representing character strings, unlike in C++, where we can use the `std::string` class.

- Instead, character strings are represented in C as arrays of `char` values, typically as `char*`.

- Importantly, all strings in C use a special character value to indicate the end of the string. This value is known as the ***null character*** and is represented as the character `'\0'` (i.e. a backslash-escaped zero character).

- Thus, in addition to the normal characters within a string, every valid C string contains this extra null character. For example in memory, the string “cat” would look like this in C:



- In other words, in C, the string “cat” would be represented as an array of *4* characters: `'c'`, `'a'`, `'t'`, and `'\0'`.

- The null character is extremely important in a C string. This is how C knows where the string ends. Functions that use strings—including `printf()` but also including several other functions, some of which we’ll see in a second—rely on the presence of the null character to know when to stop processing the string.
    - For example, `strlen()`, which simply returns the number of characters in a string, essentially counts characters by starting at the beginning of the string and incrementing until it finds a null character.

- Thus, if we are manually building a string, we *must* make sure to include a null character at the end to terminate it.

- Similarly, if we are allocating memory in which to store a string, we need to make sure to allocate enough space to hold all of the normal characters in the string *plus* the terminating null character.

- For example, the following line allocates enough space for a string with 63 normal characters and a terminating null character:

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

- C allows us to initialize constant strings as you might expect, e.g.:

```
char* name = "Leia Organa";
```

- However, when we create a constant string this way, it is important to remember that the string itself is *read-only*. In other words, a string created this way cannot be modified. For example, this would be an illegal operation on the above string:

```
name[0] = 'l';
```

- Unfortunately, most C compilers will not issue an error or warning if you try to modify a read-only string like this. Instead, the program will simply crash at runtime (often with a very cryptic error message).

- Thus, it is typically best to explicitly mark constant strings with the `const` keyword. By doing this, we ensure that the compiler will throw an error if we try to modify the string. For example, this will result in a clear compiler error:

```
const char* name = "Leia Organa";
name[0] = 'l';  /* This will be a compile-time error. */
```

- Finally, there are a lot of utility functions available in C to help us work with strings. Many of these will require you to use the following include statement:

```
#include <string.h>
```

- One very useful string utility function is `strlen()`, mentioned above. This simply returns the number of characters in a string, *not including the terminating null character*. For example, `strlen("cat")` will return 3 not 4.

- Another useful string utility function from string.h is `strncpy()`, which will copy up to a specified number of characters from one string to another. Here is a typical use that combines both `strncpy()` and `strlen()`:

```
char* name = "Leia Organa";
int n = strlen(name);
char* copy = malloc((n + 1) * sizeof(char));
strncpy(copy, name, n + 1);
```

- The third argument to `strncpy()`, which above is `n + 1`, limits the number of characters that will be copied into the destination string (i.e. `copy` above). This argument must provide enough space for the terminating null character to be copied, which is why we use `n + 1` above instead of `n`.
    - Our use of `malloc()` above also accounts for the trailing null character by allocating space for `n + 1` characters.

- Another useful string utility function is `snprintf()`, which is included via `stdio.h`. `snprintf()` provides `printf()`-like functionality for "printing" content into a string, up to a specified number of characters.

- For example, here, we print a representation of a complete student struct into a single string using `snprintf()`:

```
struct student s = {.name="Rey", .id=933222222, .gpa=3.8};
int maxlen = 128;
char* str = malloc(maxlen * sizeof(char));
snprintf(str, maxlen, "%s %d %f", s.name, s.id, s.gpa);
```

- The string stored in `str` here will look like this: `"Rey 933222222 3.8"`.

- Importantly, the `maxlen` argument we pass here to `snprintf()` limits the number of characters that can be "printed" into the string `str`. In particular, this will result in at most `maxlen - 1` normal characters being "printed" to leave room for the terminating null character, which `snprintf()` automatically inserts.

- Note that there are versions of `strncpy()` and `snprintf()` (specifically `strcpy()` and `sprintf()`) that *do not* require an argument to be passed to specify the maximum number of characters to store in the destination string. You are **STRONGLY DISCOURAGED** from using these functions, since they are highly vulnerable to a particularly insidious form of attack.

- We've only explored a very small subset of the string utility functions available in C. Check out what's in string.h to see more.

Function pointers (advanced C)

- ***Function pointers*** are an advanced feature of the C language that allows us to store the memory address of a *function* in a variable and to use that memory address to *call* the function being pointed to.

- To understand why function pointers are useful, it's helpful to know that the primary way function pointers are used is to allow us to pass functions as arguments to other functions.
    - If you're a JavaScript programmer, you probably just felt a mild, warm, fuzzy feeling thinking about passing functions as arguments to other functions.

- Why would we want to pass a function as an argument to another function? To see why, let's explore a common situation in which this is used.

- Imagine you're writing a C function to sort an array. You'd like this function to be useful in a wide variety of situations. Specifically, instead of limiting this function to sorting arrays of only one type (e.g. only `int` arrays, only `double` arrays, etc.), you'd like to implement it so that it can potentially sort arrays of any type.

- Your first step here might be to specify the function so that the array to be sorted is passed as a void pointer. For example, your sorting function's prototype might look like this:

```
void sort(void* arr, int n);
```

- So far, so good. A function prototype like this will allow us to pass an array of any kind of data into our `sort()` function (ignoring for now the fact that we don't know how big each element of the array is, i.e. how many bytes it occupies).

- One major problem remains, though, which is, without knowing the type of data it's sorting, how will our `sort()` function be able to *compare* the values in the array? In other words, how will it figure out how to *order* the data, so it can move smaller values to the front of the array and larger ones to the end?

- This is where function pointers come into play. Again, remember that a key feature of function pointers is that they allow us to pass functions as arguments to other functions.

- In particular, while our `sort()` function doesn't know the type of the data it's sorting and thus doesn't know how to compare it, the function *calling* our `sort()` function and passing data into it probably does know these things.

- Thus, our `sort()` function could require its calling function to provide, in addition to the array to be sorted, a *function* to compare elements of that array to each other to determine which ones are bigger and which ones are smaller.

- This can be accomplished by adding a third argument to our `sort()` function's prototype. This argument will be a function pointer argument, and it will be used to require the calling function to provide a function for comparing elements in the array to be sorted.

- To use a function pointer, we must know the *exact prototype* of the function we want to point to. In the context of our `sort()` function, we'll want the calling function to provide a function that compares two values from the array to be sorted and returns a value indicating which is bigger.

- In this specific situation, where we're sorting data in an array represented as a void pointer, the comparison function could take two void pointers as arguments, where the two void pointers point to different elements in the array to be compared with each other), and it could return an integer value indicating which one is bigger than the other (e.g. it could return 0 if one is bigger or 1 if the other is). In other words, the prototype of the comparison function could look like this:

```
int cmp(void* a, void* b);
```

- Let's include into our `sort()` function's prototype an additional function pointer argument matching the comparison function prototype just above. This is what it'd look like:

```
void sort(void* arr, int n, int (*cmp)(void* a, void* b));
```

- You should be able to see here how the function pointer specification corresponds to the comparison function prototype above. Specifically it has an `int` return type and takes two `void*` arguments, `a` and `b`. The only difference is that now the function name is represented using pointer syntax, i.e. as `(*cmp)`.

- How can we use this? Let's first take a look at how our `sort()` function will be called. We'll explore a simple case, where the calling function wants to sort an array of integers.

- First, the calling function will need access to a function for *comparing* integers. This function will have to match the prototype specified by the function pointer argument to our `sort()` function. This means it will have to accept the *pointers* to the integers to be compared, specifically represented as void pointers. To perform the comparison, it can simply cast the void pointers back to integer pointers and then dereference:

```
int compare_ints(void* a, void* b) {
  int* ai = a, * bi = b;  /* Cast void* back to int*. */
  if (*ai < *bi) {
    return 0;
  } else {
    return 1;
  }
}
```

- Here, we assume that our `sort()` function expects the comparison function to return 0 if `a < b` and 1 otherwise.

- Now, when the calling function invokes our `sort()` function, it can pass a pointer to `compare_ints` as the third argument:

```
sort(array_of_ints, number_of_ints, compare_ints);
```

- How do we *use* the comparison function then, within our `sort()` function? Whenever we need to compare two values from the array being sorted, we can just call `cmp()` (which is the name given to the function pointer argument), e.g.:

```
if (cmp(arr[i], arr[j]) == 0) {
  /* Put arr[i] before arr[j] in the sorted array. */
} else {
  /* Put arr[i] after arr[j] in the sorted array. */
}
```

- In this way, the calling function can simply provide a comparison function for whatever data it wants to sort, and our `sort()` function can remain agnostic about what kind of data it's actually operating on. This kind of approach will be extremely useful in allowing us to implement completely general-purpose data structures in C.

- Indeed, this kind of approach is common practice for widely-used C utilities, such as the `qsort()` function, which does almost exactly what our `sort()` function here does using a similar approach.