CAAM 519: Computational Science I

Module #6: C

The C programming language

The C programming language was initially developed in the 1970s by Dennis Ritchie and Brian Kernighan, and it has been one of the most popular languages ever since. One reason is its simplicity: just look how slim the C bible, “The C Programming Language,” is. Another is its versatility, as it can be used for any number of tasks from low level systems programming (e.g. writing an operating system) to scientific computing. Finally, it’s almost universally portable, meaning you can likely run your code on any machine that you might conceivably need to use.

In Julia, we could start writing code immediately: open up a REPL and start typing line-by-line. Julia is a just-in-time compiled language, which means that the compilation stage (generating the machine code that will be run) happens immediately before that code is actually run. In C, these two stages, *compilation* and *runtime*, are completely decoupled.

My first C program

First, let’s install a C compiler. In our Linux environment, you can do this with the following incantation at the terminal:

```
> sudo apt install gcc
```

On a Mac, you should already have gcc installed if you successfully installed the Xcode developer tools.

Let’s write the canonical first C program, which just prints something to STDOUT. Open your favorite text editor and save the following code to a file named `hello-world.c`:

```
#include <stdio.h>

int main(void){
    printf("Hello, world.\n");
}
```

You have written your first C *source file*! Conventionally, these files end with a `.c` extension. Let’s work through the code line-by-line.

First, we load the printing library. We do this through what is known as a preprocessor instruction. This tells the compiler to load in the `stdio` library by stating its *header file* named `stdio.h`. Conventionally, header filenames end in a `.h` extension. Header files serve to state the interface offered by the code in a source file. More on them later!

Next, we see the definition of our main function. Every C program has a main function, which is the entrypoint. That is, when you compile and run a C program, the main function is what gets run. Here we see the syntax for a function definition in C. The `int` states the return type of the function (conventionally, the main function returns 0 on successful completion), and `main` is the function name. Our main function does not take any arguments, hence the `void`, and the body of the function is delimited by curly brackets.

Our function body has but one line, where we call the `printf` function, which can be found in the `stdio` library we loaded at the top. We call the function with a single argument, which is the string we want printed to STDOUT. Note that after the function call, we have a semicolon. Every expression in C must be ended with a semicolon; you will be writing a lot of them.

Now to compile the program, we do invoke the compiler with

```
> gcc hello-world.c
```

If you now inspect your current directory, you will notice a new file, named `a.out`. This is an executable that you can now run! Simply do

```
> ./a.out
Hello, world.
```

to execute your first C program.

If you want to name the executable something different, use the compiler flag

```
> gcc hello-world.c -o hello-world
> ./hello-world
Hello, world.
```

My second C program

Now let's write a slightly more complex program, where we can observe the syntax for defining, declaring, and initializing variables.

```c
#include <stdio.h>

// Defined and declared
float x;
int main(void){
    // Defined, declared, and initialized
    int y = 1;
    printf("y = %d\n", y);
    // Declared
    extern float x;
    // Initialized
    x = 10.1;
    printf("x = %f\n", x);
}
```

```
y = 1
x = 10.100000
```

Note a couple things. First, when defining or declaring a variable, you must explicitly state the type of the variable. Second, you can declare a variable without initializing it. Third, we see the `extern` keyword, which functions similarly to the `global` keyword in Julia. Fourth, the assignment syntax is identical to in Julia. Fifth, note all those semicolons at the end of every expression.

Function prototypes and header files

Consider the following code:

```c
#include <stdio.h>

int main(void){
    int b = 2;
    printf("y = %d\n", my_func(b));
}

int my_func(int b){
    return b + 1;
}
```

If we try to compile this, we get:

```
> gcc out-of-order.c
out-of-order.c:3:12: warning: implicit declaration of function 'my_func' is
      invalid in C99 [-Wimplicit-function-declaration]
    return my_func(b);
           ^
1 warning generated.
```

The compiler is complaining because, proceeding in a straight-line fashion through the code, it encounters a function which hasn't been declared yet. Because C is a crazy language sometimes, the compiler assumes that you meant to call a function that returns an integer, implicitly adds a declaration, and proceeds with only a warning. So, we got lucky, as our later function definition can be slotted in without issue. Let's change this slightly:

```c
#include <stdio.h>

int main(void){
    int b = 2;
    printf("y = %f\n", my_func(b));
}

float my_func(int b){
    return b + 1.0;
}
```

Now if we try to compile it, we get

```
> gcc out-of-order.c

demo.c:5:24: warning: implicit declaration of function 'my_func' is invalid
in C99
      [-Wimplicit-function-declaration]
    printf("y = %f\n", my_func(b));
                       ^
demo.c:5:24: warning: format specifies type 'double' but the argument has
type 'int'
      [-Wformat]
    printf("y = %f\n", my_func(b));
                ~~     ^~~~~~~~~~
                %d
demo.c:8:7: error: conflicting types for 'my_func'
float my_func(int b){
```

```
      ^
demo.c:5:24: note: previous implicit declaration is here
    printf("y = %f\n", my_func(b));
                       ^
2 warnings and 1 error generated.
```

This is a difficult error to parse at first, but it's complaining that the implicit definition it added doesn't match the definition we later gave, is does not know how to proceed.

This example is meant to illustrate two things. First, take compiler warnings seriously! It's good practice to ask for a liberal amount of warnings with the `-Wall` gcc flag.

Second, we see that in C, we must define a function before we can use it in the definition of another function (which is not the case in Julia). This is not too difficult when you have two functions in a single file, but can become extremely difficult in larger projects, spread across many files and libraries. To help with this, C has function declaration syntax, where you can declare a function name and signature (*prototype*) to the compiler before its actual definition.

We can use this to fix the previous code like so:

```
#include <stdio.h>

float my_func(int);

int main(void){
    int b = 2;
    printf("y = %f\n", my_func(b));
}

float my_func(int b){
    return b + 1.0;
}
```

```
y = 3.000000
```

Function declarations are typically placed inside of header files. The standard pattern is to place function definitions in a source file (named, for example, `my_code.c`), and then place the function declarations (along with any other constants) in a header file with the same name (so `my_code.h`). Then the header file can be reused in multiple locations, in different pieces of code.

We can write our first header file. Inside of first-prototype.h, put

```
float my_func(int);
```

And inside of first-prototype.c, put

```
#include <stdio.h>
#include "first-prototype.h"

int main(void){
    int b = 2;
    printf("y = %f\n", my_func(b));
}

float my_func(int b){
    return b + 1.0;
}
```

Note the two differing syntaxes in the `#include` statements. The first uses arrow brackets, and directs the compiler to look in the standard library for the given header file. The second uses quotes, and tells the compiler to look for the header file in the file system, either at a relative path (like here), or an absolute path if that is what is given.

Pointers

A *pointer* is a variable that contains the address of another variable. They allow you to pass around objects to be manipulated, without necessarily making copies of the underlying data. Crucially, pointers are typed, so the compiler know exactly how to interpret the underlying data, and how far into that block of memory to read (e.g. 64 bits for a double, or 8 bits for a character).

In C, much like you can declare a variable to be an int, you can declare a variable to be a pointer to an int:

```
int my_int;  // Declare an int
int* ptr_int;  // Declare a pointer to an int

my_int = 2;  // Initialize an int
ptr_int = &my_int;  // Initialize a int pointer
```

The * character, juxtaposed with a type, denotes a pointer to that type. The & symbol returns the address to the object it is juxtaposed with. More examples:

```
int a = 2;
Int b = 3;
printf("At the beginning: a = %d, b = %d\n", a, b);
int* ptr_int;

ptr_int = &a;
b = *ptr_int;
*ptr_int = 4;
printf("At the end: a = %d, b = %d\n", a, b);
```

Here the `*ptr_int` syntax *dereferences* the pointer, returning the value pointed to in memory.

Pointers are valid function arguments, meaning you can create an object, create a pointer to it, pass it to a function, and have that function act on the object (either mutating it or otherwise using its data), all without making a copy. An example:

```
#include <stdio.h>

void add_one(int* x, double* y){
    *x = *x + 1;
    *y = *y + 1.0;
}

int main(void){
    int a = 2;
    double b = 3.0;
    printf("At the beginning: a = %d, b = %f\n", a, b);
    add_one(&a, &b);
    printf("At the end: a = %d, b = %f\n", a, b);
}
```

```
At the beginning: a = 2, b = 3.000000
At the end: a = 3, b = 4.000000
```

Since pointers are themselves just ordinary variables, you can also take pointers to pointers! This is actually fairly common in C. For example, if you pass arguments to a main function, you will see a double pointer appear:

```c
#include <stdio.h>

int main(int argc, char** argv){
    printf("You have %d values available.\n", argc);
    printf("The invocation of the program is: %s\n", *argv);
    printf("The first argument is: %s\n", *(argv+1));
}
```

```
> gcc main-args.c
> ./a.out 1.2
You have 2 values available.
The first is: ./a.out
The second is: 1.2
```

A couple things of note here. First, argv contains the name of the program invoked, as well as any arguments passed at the command line. Each piece of data is stored as a pointer to `char`. In other words, it points to a location of `char`s in memory; you can read this string of `char`s and interpret it as, well, a string. Since the number of fields (which is available in the program via the `argc` value) is dynamic and dependent on how we call the program, we store each pointer to char sequentially in memory, and provide the program with a pointer to those `char*`'s; hence the double `**`. Second, we see our first example of pointer arithmetic. You can think of this as a (relatively) safe way to traverse through the block of memory nearby the address stored in the pointer. Addition or subtraction is relative to the address specified by the pointer, and increments in units of (# bits used to store the type pointed to). So adding one to the pointer, as we see above, gives the address to the next `char*` in memory.

This leads us to our last observation: what happens if we call the executable with no arguments? Let's try:

```
> ./a.out
You passed 1 arguments.
The first is: ./a.out
The second is: (null)
```

Here we see what is known as a null pointer. Null pointers have special meaning; they are understood to not point to anything, and attempting to read from them will lead to a runtime error. are powerful as they allow you to create pointers before you instantiate the object you would like to point to:

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

```c
int main(void){
    int* ptr;
    printf("Initially: ptr = %p, value = %d\n", ptr, *ptr);
    int i = 12;
    ptr = &i;
    printf("Next: ptr = %p, value = %d\n", ptr, *ptr);
    ptr = NULL;
    printf("Finally: ptr = %p, value = (cannot access)\n", ptr);
}
```

```
> gcc null-ptr.c
> ./a.out
Initially: ptr = 0x10893d036, value = -125990072
Next: ptr = 0x7ffeedd0a994, value = 12
Finally: ptr = 0x0, value = (cannot access)
```

Note something here: a declared, but uninitialized pointer points to garbage, and can be dereferenced to look at garbage. However, a null pointer cannot be dereferenced without producing an error.

Null pointers are powerful, but can lead to catastrophic runtime failures if you're not careful. Famously, the "inventor" of the null pointer (in ALGOL W) says that:[1]

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Arrays

Basic array operations should look pretty familiar:

```c
int int_array[10];
int int_array_literal[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

[1] https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/

```
int_array[0] = 7;
int_array_literal[6] = int_array[0];
Int two_d_array[10][5];
```

Indeed, arrays and pointers are nearly the same thing in C. However, pointers have a bit more freedom, as you can change the address that a pointer points to; this is not allowed when working with arrays.

Control flow

The control flow constructs in C are fairly similar to what we have seen in Julia. You can create in/else conditional statement:

```
#include <stdio.h>

int main(int argc, char** argv){
    if (argc == 1){
        printf("No arguments.\n");
    } else if (argc == 2){
        printf("One argument.\n");
    } else {
        printf("More than one argument.\n");
    }
}
```

```
> gcc if.c
> ./a.out
No arguments.

> ./a.out 1
One argument.

> ./a.out 1 2 3
More than one argument.
```

C does not have "true" boolean types, so comparisons return 0 or 1. If statements will proceed if the condition expression evaluates to any nonzero value.

For loops are slightly more verbose. The statement has three parts, delineated with semicolons: `(initialization; termination; increment)`.

```c
#include <stdio.h>

int main(void){
    for (int i = 0; i < 6; ++i){
        printf("Loop iteration %d\n", i);
    }
}
```

```
Loop iteration 0
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
Loop iteration 5
```

You can also use while loops:

```c
#include <stdio.h>

int main(void){
    int i = 0;
    while (i < 6){
        printf("Loop iteration %d\n", i);
        i += 1;
    }
}
```

```
Loop iteration 0
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
Loop iteration 5
```

There's also the more esoteric do while loop, which checks the condition after the loop body is executed:

```c
#include <stdio.h>

```

```
int main(void){
    int i = 0;
    do {
        printf("Loop iteration %d\n", i);
        i += 1;
    } while (i < 0);
}
```

```
Loop iteration 0
```

There are two other control flow keywords that are useful inside of while or for loops: `continue` and `break`. Both immediately stop and exit the current iteration. The `continue` keyword proceeds on to the next iteration, while the `break` keyword terminates the loop completely.

```
#include <stdio.h>

int main(void) {
    int i = 0;
    while (1) {
        i++;
        if (i % 2) {
            continue;
        }
        printf("Iteration %d\n", i);
        if (i == 10) {
            break;
        }
    }
}
```

```
Iteration 2
Iteration 4
Iteration 6
Iteration 8
Iteration 10
```

Memory management

Everything we have seen up to this point--even arrays--has involved data stored on the *stack*. This is a portion of RAM where objects allocated inside functions are placed, pushed onto the stack in a FIFO manner, and then popped off and freed once the calling function exits.

Pros of the stack:

- Read/write access is very quick.
- All the memory management is handled automatically by the compiler and CPU.

Cons:

- Limited memory available on the stack.
- Variables cannot be dynamically resized.
- No global state, only preserved in lifetime of allocating function.

In contrast, the *heap* is a portion of memory where memory management is handled by the user.

Pros of the heap:

- Global access to memory.
- Much larger amount of memory available for use.
- Data can be resized dynamically.

Cons: does memory on the heap persist after program ends?

- Slower read/write access.
- Must be manually managed and freed.
- Possible memory fragmentation for long-running programs.

As mentioned, everything we have done up to this point has been with variables stored on the stack. To work with the heap, we need to use functions like `malloc` (or friends) to allocate chunks of memory, and then `free` to free that memory when done.

Here's an example of code that only uses stack allocations:

```
#include <stdio.h>

double multiplyByTwo (double input) {
  return input * 2.0;
}

int main (void) {
  int age = 30;
  double salary = 12345.67;
  double myList[3] = {1.2, 2.3, 3.4};

  printf("double your salary is %.3f\n", multiplyByTwo(salary));
}
```

```
double your salary is 24691.340
```

And a version that only uses heap allocations:

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

double *multiplyByTwo (double *input) {
  double *twice = malloc(sizeof(double));
  *twice = *input * 2.0;
  return twice;
}

int main (void) {
  int *age = malloc(sizeof(int));
  *age = 30;
  double *salary = malloc(sizeof(double));
  *salary = 12345.67;
  double *myList = malloc(3 * sizeof(double));
  myList[0] = 1.2;
  myList[1] = 2.3;
  myList[2] = 3.4;

  double *twiceSalary = multiplyByTwo(salary);

  printf("double your salary is %.3f\n", *twiceSalary);

  free(age);
  free(salary);
  free(myList);
  free(twiceSalary);
}
```

```
double your salary is 24691.340
```

The code is much more cumbersome this way (many more pointers), and as you can imagine it is sometimes easy to forget to free memory when you are done with it.

The functions to allocate or deallocate memory are available in the stdlib library. As we see, `malloc` allocates a block of memory of size equal to its sole argument (in bytes):

```
int* ptr;
ptr = malloc(4 * sizeof(int));
```

It returns a pointer to that allocated memory. The data is uninitialized, so it is full of garbage values.

In fact, the data is uninitialized and untyped. The `malloc` function returns a `void*`, which is just a raw pointer to a string of bytes that the compiler has no idea how to interpret. Of course, we declared ptr to be of type `int*`, so above we see a type cast to convert the type of an object (`void*`) to another type (`int*`). This occurs implicitly above, but we can also make this explicit:

```
ptr = (int*) malloc(4 * sizeof(int));
```

If you would instead like a chunk of memory initialized to zero, use the `calloc` function, which has a slightly different signature:

```
int* ptr;
ptr = calloc(4, sizeof(int));
```

To resize a chunk of memory on the heap, use the `realloc` function:

```
int* ptr;
ptr = malloc(4 * sizeof(int));
ptr = realloc(ptr, 8 * sizeof(int));
```

Finally, once you are done with the data, free it with `free`. Be careful that you are truly done with the data, and do not leave pointers *dangling*! For example,

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

int main(void) {
    int* ptr = calloc(4, sizeof(int));
    int* second_element = ptr + 1;   does second_element need to be freed?
    // A bunch of computations ensue...
    free(ptr);
    int* big_alloc = malloc(100000 * sizeof(int));
    printf("First element pointed to is %d.\n", *ptr);
}
```

Even more insidious is

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

int main(void) {
    int* ptr = calloc(4, sizeof(int));
    int* second_element = ptr + 1;
    // A bunch of computations ensue...
    free(ptr);
    int* big_alloc = malloc(100000 * sizeof(int));
    printf("The second element from before was %d.\n", *second_element);
}
```

This won't necessarily give you a segmentation fault! It might work fine if the memory you initially allocated wasn't overwritten, or it might silently give wrong answers or corrupt data. The silent and unpredictable nature is why this is so dangerous.

Additionally, as mentioned above, memory allocated on the stack won't survive outside the scope of the function. So, the following code is unsafe, and could potentially crash:

```
#include <stdio.h>

int* ptr_to_int(void) {
    int x = 12;
    return &x;
}

int main(void) {
    int* x_ptr = ptr_to_int();
    printf("x_ptr =  %p, x = %d\n", x_ptr, *x_ptr);
}
```

Passing pointers to functions

One of the main advantages of pointers is that they allow you to pass objects as arguments to functions without making a copy. Imagine the following setting: you allocate a huge block of memory to store results from a simulation. After computing everything, you want to run some analysis. Instead of copying the humongous amount of data, amount of data, you could just pass a pointer to its location in memory.

This raises a question, though: how do you ensure that, once you have passed the pointer to another function, the data is not being altered if you don't want it to be? To help with this, C has a `const` annotation that you can use to declare a variable constant, meaning that the compiler will ensure that the associated value will not change.

```
int main(void) {
    const int j = 20;
    j = 30;
}
```

Trying to compile this leads to:

```
main.c:3:7: error: cannot assign to variable 'j' with
      const-qualified type 'const int'
    j = 30;
    ~ ^
main.c:2:15: note: variable 'j' declared const here
    const int j = 20;
    ~~~~~~~~~~^~~~~~
1 error generated.
```

This is simple enough in this case. What about pointers? In this case, we have to things we can label const: the pointer itself, and the underlying value. Here an illustration:

```
int main(void) {
    int i = 1;
    int j = 2;
    // Pointer to constant
    const int* ptr_1 = &i;
    // The following won't work:
    // *ptr_1 = 100;
    // But this will:
```

```
    ptr_1 = &j;

    // Constant pointer to variable
    int* const ptr_2 = &i;
    // This will work:
    *ptr_2 = 100;
    // But this won't:
    // ptr_2 = &j;

    // Constant pointer to constant
    const int* const ptr_3 = &i;
    // Neither of the following work:
    // *ptr_3 = 100;
    // ptr_3 = &j;
}
```

Dynamic memory arrays

We saw above the syntax for declaring arrays on the stack (e.g. `int my_array[5]`). While the syntax is clean, because they are stored on the stack these arrays must have fixed size. However, we saw that once allocated, they can be treated similarly to pointers to raw data:

```
int main(void) {
    int M[4][3];
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 3; ++j) {
            int k = 3 * i + j;
            // six equivalent ways to do the same thing:
            M[i][j] = k;
           *(M[i] + j) = k;
           *(M[0] + k) = k;
           M[0][k] = k;
           (*M)[k] = k;
           *((*M)+k) = k;
        }
    }
}
```

To create dynamically managed arrays stored on the heap, we can return to `malloc`. First, to create vectors, we can do:

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

int main(void) {
    int* M;
    int length = 4;
    M = (int*) malloc(length*sizeof(int));
    M[1] = 7;
    printf("M[1] = %d\n", *(M+1));
}
```

For dynamic multidimensional arrays, we have three options.

Option 1: A single pointer

```c
#include <stdlib.h>

int main(void) {
    int rows = 3;
    int columns = 4;
    int* ptr = (int*) malloc(rows * columns * sizeof(int));
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < columns; j++) {
            *(ptr + i * columns + j) = i * columns + j;
        }
    }
    free(ptr);
}
```

Option 2: A pointer to pointers, separately allocated

```c
#include <stdlib.h>

int main(void) {
    int rows = 3;
    int columns = 4;
    int** ptr = (int**) malloc(rows * sizeof(int*));
    for (int i = 0; i < rows; ++i) {
        ptr[i] = (int*) malloc(columns * sizeof(int));
        for (int j = 0; j < columns; j++) {
            ptr[i][j] = i * columns + j;
        }
    }
    for (int i = 0; i < rows; ++i) {
        free(ptr[i]);
    }
    free(ptr);
}
```

Note we have to free each pointer separately, so `rows + 1` calls to `free`.

Option 3: A pointer to pointers, with contiguous memory

```c
#include <stdlib.h>

int main(void) {
    int rows = 3;
    int columns = 4;
    int** ptr = (int**) malloc(rows * sizeof(int*));
    ptr[0] = (int*) malloc(rows * columns * sizeof(int));
    for (int i = 1; i < rows; ++i) {
        ptr[i] = ptr[0] + columns * i;
    }
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < columns; j++) {
            ptr[i][j] = i * columns + j;
        }
    }
    free(ptr[0]);
    free(ptr);
}
```

Exercise: Which approach is preferable...from a performance perspective? From a usability perspective?

Compiling projects across multiple files

Any substantial coding project will naturally be spread across multiple source files; trying to pack everything into a single file is a recipe for disorganized clutter. We will now see how you can compile multiple source files together to create a single executable. As a simple example, consider one source file, named `greeting_func.c`:

```c
#include <stdio.h>

void my_greeting(char* name) {
    printf("Hello, %s!\n", name);
}
```

along with its header file, `greeting_func.h`:

```c
void my_greeting(char* name);
```

And another source file named my`_greeting.c`:

```c
#include "greeting_func.h"

int main(void) {
    my_greeting("world");
}
```

To compile everything, we can run

```
> gcc my_greeting.c greeting_func.c -o hello_world
```

A few things to observe here. First, since we only use the `printf` function inside the `my_greeting` function, we only need to load the `stdio` header in `greeting_func.c`. Second, to load in the declaration for `my_greeting`, we include the appropriate header (`greeting_func.h`) in the source file containing our `main` function. We see now why header functions can be extremely handy!

Structures

C has structures that the user can define to collect and organize data, much as we saw in Julia. Even the definition and basic operations are roughly the same:

```c
#include <stdio.h>

struct my_struct {
    int n;
    char* name;
};
```

```c
int main(void) {
    // Declare, then initialize field-by-field
    struct my_struct instance_1;
    instance_1.n = 10;
    instance_1.name = "Joey";

    // Declare and initialize
    struct my_struct instance_2 = {11, "Jerry"};

    // Change
    typedef struct my_struct renamed_struct;

    // Declare and initialize by named fields
    renamed_struct instance_3 = {.name = "Johnny", .n = 12};
}
```

In fact, we can typedef the name right at definition so we don't keep having to write `struct` everywhere in our code:

```c
#include <stdio.h>

typedef struct {
    int n;
    char* name;
} your_struct;

int main(void) {
    your_struct instance = {10, "Joey"};
    printf("n = %d, name = %s\n", instance.n, instance.name);
}
```

Where should you place struct definitions? Like we saw with functions, if you define it in a source file, it is only accessible in functions defined after it in the same source file. To make it more broadly accessible, you should place it in a header file, which can be shared among potentially many source files that want to use the same data structure.

However, you must be a bit careful not to define the same structure multiple times in the same project. To illustrate this, consider the following example:

my_struct.h:

```c
typedef struct {
    int n;
} my_struct;
```

unpack.h

```c
#include "my_struct.h"

int unpack(my_struct instance);
```

unpack.c

```c
#include "unpack.h"

int unpack(my_struct instance) {
    return instance.n;
}
```

main.c

```c
#include <stdio.h>
#include "my_struct.h"
#include "unpack.h"

int main(void) {
    my_struct instance = {12};
    printf("n = %d\n", unpack(instance));
}
```

When I try to compile this, I get:

```
> gcc main.c unpack.c -o unpack
In file included from main.c:3:
./my_struct.h:3:3: error: typedef redefinition with different types
('struct my_struct' vs 'struct my_struct')
} my_struct;
  ^
./unpack.h:1:10: note: './my_struct.h' included multiple times, additional
```

```
include site here
#include "my_struct.h"
         ^
main.c:3:10: note: './my_struct.h' included multiple times, additional
include site here
#include "my_struct.h"
         ^
./my_struct.h:3:3: note: unguarded header; consider using #ifdef guards or
#pragma once
} my_struct;
  ^
1 error generated.
```

The problem is that, by including the same header file with a type definition multiple times, the compiler thinks we are trying to define (and redefine) it repeatedly.

How do we get around this? In C, the convention is to use *header guards*. You can think of these roughly as if statements for the compiler. To fix this, rewrite `my_struct.h` to be:

```
#ifndef _MY_STRUCT_H
#define _MY_STRUCT_H

typedef struct {
    int n;
} my_struct;

#endif
```

Essentially, we are defining a dummy macro the first time we define the struct, and then checking each time to see if that macro is already defined to avoid a redefinition error. It's good practice to name the macro to match the header file name in some way to avoid name clashes that break this logic and lead you to miss out on needed definitions.

Back to structs…

We saw above that you can pass and return structs to and from functions in the familiar manner. We can also work with pointers to these objects. In fact, we have new syntax as well:

```
#include <stdio.h>

typedef struct {
```

```c
    int age;
    char* name;
} Player;

Player young_player(Player player) {
    Player player_out;
    player_out.age = 22;
    player_out.name = player.name;
    return player_out;
}

int main(void) {
    Player bill = {30, "Bill"};
    Player young_bill = young_player(bill);
    Player* ptr_player;
    ptr_player = &bill;
    // Two equivalent ways to do the same thing:
    (*ptr_player).age = 23;
    ptr_player->age = 23;
    printf("Players name: %s\n", ptr_player[0].name);
    printf("Players age: %d\n",(*ptr_player).age);
}
```

For a final example, we show how to allocate user defined structs on the heap:

```c
#include <stdlib.h>

typedef struct {
    int age;
    char* name;
} Player;

int main(void) {
     int n = 10;
     Player* ptr_players;
     ptr_players = (Player*) malloc(10 * sizeof(Player));
     if (ptr_players != NULL) {
          ptr_players->age = 30;
          (ptr_players+4)->age = 40;
          (*(ptr_players+5)).age = 25;
          ptr_players[6].age = 22;
    }
}
```

Compilation and Linking

Up to this point, we have been using our compiler, gcc, to directly translate source code to executables. However, this glosses over how gcc actually works. In particular, there are really two separate stages: compilation and linking. Let's break them down separately.

Compilation

If a source file is code that a human can understand, an object file contains code that your computer can understand. Object files typically have a `.o` extension in Linux. They contain two things: code (corresponding to the functions you defined in your source files) and data (corresponding to global variables in your source files).

Let's look at some object files. Write the following source code file `source.c`:

```
int x_global_uninit;

int x_global_init = 1;

extern int x_global;

int fn_a(int x, int y);

int fn_b(int x_local) {
    return fn_a(x_local, x_global_init) + x_global;
}
```

We can compile this down to an object file with

```
> gcc -c source.c
```

Looking in our file system, there was an object file named `source.o` created. We can inspect the symbols available in this object file with

```
> nm source.o
```

```
                 U _fn_a
0000000000000000 T _fn_b
                 U _x_global
0000000000000028 D _x_global_init
0000000000000004 C _x_global_uninit
```

Here's how to read this output. The first column is the value associated with the symbol, if any. The second column is the class of the symbol:

- U: Undefined reference
- T: Defined code
- D: Initialized global variable
- C: Uninitialized global variable

So we compiled our code without error, even though we never initialized some things that we compiled. That's fine, though, because the next step allows us to link together the symbols across multiple object files.

Linking

A declaration of a function or variable is a promise in code that such a method or variable will be defined elsewhere. The linker is the program which resolves this promise. Let's write a second C source file, `source_2.c`, and compile it:

```c
#include <stdio.h>

int x_global = 12;

extern int x_global_init;

int fn_b(int x);

int fn_a(int x, int y) {
    return x + y;
}

int main(void) {
    printf("Hello, world!\n");
    return fn_b(1);
}
```

Then compile and inspect the object file:

```
> gcc -c source_2.c
> nm source_2.o
0000000000000000 T _fn_a
                 U _fn_b
0000000000000020 T _main
                 U _printf
0000000000000050 D _x_global
```

If we match up the two symbol tables, we see that each symbol is defined exactly once between the two (except for `printf`, which will come from a separate system object file). Great! Now let's link the two together into an executable, and run it:

```
> gcc -o linked source.o source_2.o
> ./linked
Hello, world!
```

Here, gcc is taking a bunch of computer-readable code and data, resolving the names between them into a single table, and producing an executable. This executable is merely another file that instructs your OS to load the code and data in those object files into memory, and then begin execution of the `main` code.

Libraries

Above I said that each symbol has to be defined exactly once, but then quickly turned around and said that `printf` was defined elsewhere, not in our object files, and was somehow magically brought in during linking. In fact, `printf` comes from a library, which is a collection of object files kept together for reuse among many programs.

More specifically, a *static library* is a collection of object files bundled together.

When the linker is running, it searches first through all the object files to resolve any symbols. If some are still missing, it defers to any libraries that might be linked in as well, searching through those bundled object files for the missing symbol.

As an example, let's create a shared library with just our source.o file:

```
> ar r libsource.a source.o
> gcc -o linked source.o -L. -lsource
                source_2.o
> ./linked
Hello, world!
```

To parse this out a bit: `ar` is the command line tool to create a static library. It is standard to use the `.a` extension for libraries, and prepend lib to their name. To explicitly load in a library, when we invoke gcc we first pass the flag `-L.`, which tells the linker to look for libraries in a new directory (namely, the current one, with `.`). Next, the `-lsource` flag names the library to look for; gcc knows that the library will likely begin with "`lib`" and end with "`.a`", so it can fill in the rest.

Static libraries are the simplest to understand, but have some potential downsides. Another type of library is a *dynamic library*. Essentially, during linking a static library will pull into the executable being created a copy of code or data that is used in the linking code. In contrast, a dynamic library will merely include a pointer to the dynamic library in the file system, allowing the OS to pull in the correct code or data when the program is actually run (essentially, by running a mini version of the linker "just-in-time"). We won't go into further detail on dynamic libraries, but you should know that they're out there, and are quite useful in certain circumstances.

Make

Make is a build tool: it allows you to write code that can build other code by invoking compilers, linkers, and other command line utilities. It is particularly handy for complex software projects with many source files (imagine calling gcc on 200 source files by hand). Make is a complex piece of software with arcane rules and syntax, so I will only cover the very basics with examples. Given a source file `hello-world.c`, create the following file name `Makefile`:

```
# Set global variables, e.g. compilers
CC = gcc
# Targets
hello-world: hello-world.o
	$(CC) hello-world.o -o hello-world
hello-world.o: hello-world.c
	$(CC) -c hello-world.c
clean:
	rm -f hello-world hello-world.o
```

This make file has three *targets*, or a named action that it can take. Each target is specifed as

```
target-name: prerequisites
	command
```

So if we want to build an executable for the `hello-world.c` file, we invoke make

```
> make
```

The command line tool make identifies the Makefile in the current directory, finds the first target inside it, and then before running the command, runs the prerequisites. This means that the `hello-world.o` target gets run first, whose dependency is the hello-world.c source file, which make identifies in the file system. I then runs the command in the shell, interpolating the variable specifying the compiler. Once done, it returns the the hello-world target, where all prerequisites are now satisfied. Then it can invoke the compiler to link the object file, creating the executable, and then finally stopping.

If we want to invoke a particular target, for example the clean target which removes anything in the file system resulting from our build, you can do that with

```
> make clean
```

For a more realistic example of a make file, see below:

```
#Define compiler
CC=gcc
#Define linker
LD=gcc
#Compilation flags
CFLAGS= -c -std=c99 -Wall
#Link flags
LDFLAGS=
#Define addition library (like -lm)
LIBS=-lm
#Define include path (where header file could be)
INCLUDE=./
#Define sources files to compile
SOURCES=main.c struct.c vector.c
#Define generated object files from SOURCES
OBJECTS=$(SOURCES:.c=.o)
#Define name executable
EXEC=my program.exe
### End MACROS ### Begin targets
build: $(SOURCES) $(EXEC)
$(EXEC) : $(OBJECTS)
$(LD) $(LDFLAGS) $(OBJECTS) $(LIBS) -o $@
%.o: %.c
$(CC) $(CFLAGS) -I$(INCLUDE) $< -o $@
#Define additional targets
```

```
clean :
rm -rf $(OBJECTS)
all : clean build
```