Section 0: Tools, x86, C

CS 162

August 28, 2020

Contents

1 Vocabulary

With credit to the Anderson & Dahlin textbook (A&D):

- **stack** - The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some book-keeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed.

- **heap** - The heap is memory set aside for dynamic allocation. Unlike the stack, there is no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time.

- **process** - A process is an instance of a computer program that is being executed, typically with restricted rights. It consists of an address space and one or more threads of control. It is the main abstraction for protection provided by the operating system kernel.

- **address space** - The address space for a process is the set of memory addresses that it can use and the state associated with them. The memory corresponding to each process' address space is private and cannot be accessed by other processes, unless it is explicitly shared.

- **C** - A high-level programming language. In order to run it, C will be compiled to low level machine instructions like x86_64 or RISC-V. Note that it is often times easier to express high level ideas in C, but C cannot be used to express many details (such as register allocation).

- **x86** - A very popular family of instruction sets (which includes i386 and x86_64). Unlike MIPS or RISC-V, x86 is primarily based on CISC (Complex Instruction Set Computing) architecture. Virtually all servers, desktops, and most laptops (with Intel or AMD) natively execute x86.

Tools are important for every programmer. If you spend time learning to use your tools, you will save even more time when you are writing and debugging code. This section will introduce the most important tools for this course.

2 Make

GNU Make is program that is commonly used to build other programs. When you run `make`, GNU Make looks in your current directory for a file named `Makefile` and executes the commands inside, according to the makefile language.

```
my_first_makefile_rule:
	echo "Hello world"
```

The building block of GNU Make is a **rule**. We just created a rule, whose **target** is `my_first_makefile_rule` and **recipe** is `echo "Hello world"`. When we run `make my_first_makefile_rule`, GNU Make will execute the steps in the recipe and print "Hello world".

Rules can also contain a list of **dependencies**, which are other targets that must be executed before the rule. In this example, the `task_two` rule has a single dependency: `task_one`. If we run "`make task_two`", then GNU Make will run `task_one` and then `task_two`.

```
task_one:
	echo 1
task_two: task_one
	echo 2
```

2.1 More details about Make

- If you just run `make` with no specified target, then GNU Make will build the first target.
- By convention, target names are also file names. If a rule's file exists and the file is **newer** than all of its dependencies, then GNU Make will skip the recipe. If a rule's file does not exist, then the timestamp of the target would be "the beginning of time". Otherwise, the timestamp of the target is the **Modification Time** of the corresponding file.
- When you run "`make clean`", the "clean" recipe is executed every time, because a corresponding file named "clean" is never actually created. (You can also use the `.PHONY` feature of the makefile language to make this more robust.)
- Makefile recipes **must be indented with tabs**, not spaces.
- You can run recipes in parallel with "`make -j 4`" (specify the number of parallel tasks).
- GNU Make creates automatic rules if you don't specify them. For example, if you create a file named `my_program.c`, GNU Make will know how to compile it if you run "`make my_program`".
- There are many features of the makefile language. Special variables like `$@` and `$<` are commonly used in Makefiles. Look up the documentation online for more!

Pintos, the educational operating system that you will use for projects, has a complex build system written with Makefiles. Understanding GNU Make will help you navigate the Pintos build system.

3 Git

Git is a distributed revision control and source code management (SCM) system with an emphasis on speed, data integrity, and support for distributed, non-linear workflows. GitHub is a Git repository hosting service, which offers all of the distributed revision control and SCM functionality of Git as well as adding many useful and unique features.

In this course, we will use Git and GitHub to manage all of our source code. It's important that you learn Git, but NOT just by reading about it.

3.1 Helpful Resources

- https://try.github.io/
- Atlassian Git Cheat Sheet, especially the section *Git Basics*

3.2 Some Commands to Know

- **git init**
  Create a repository in the current directory
- **git clone <url>**
  Clone a repository from <url> into a new directory
- **git status**
  Show the working tree status
- **git pull <repo> <branch>**
  Fetch from branch <branch> of repository <repo> and integrate with current branch of repository checked out
- **git push <repo> <branch>**
  Pushes changes from local branch <branch> to remote repository <repo>
- **git add <file(s)>**
  Add file contents to the index
- **git commit -m <commit message>**
  Record changes to the repository with the provided commit message
- **git branch**
  List or delete branches
- **git checkout**
  Checkout a branch or path to the working tree
- **git merge**
  Join two or more development histories together
- **git rebase**
  Reapply commits on top of another base commit
- **git diff [--staged]**
  Show a line-by-line comparison between the current directory and the index (or between the index and HEAD, if you specify --staged).

- **git show [--format=raw] <tree-ish>**
  Show the details of anything (a commit, a branch, a tag).
- **git reset [--hard] <tree-ish>**
  Reset the current state of the repository
- **git log**
  Show commits on the current branch
- **git reflog**
  Show recent changes to the local repository

4 GDB: The GNU Debugger

GDB is a debugger that supports C, C++, and other languages. You will not be able to debug your projects effectively without advanced knowledge of GDB, so make sure to familiarize yourself with GDB as soon as possible.

4.1 Some Commands to Know

- **run, r:** start program execution from the beginning of the program. Also allows argument passing and basic I/O redirection.
- **quit, q:** exit GDB
- **kill:** stop program execution.
- **break, break x if condition:** suspend program at specified function (e.g. "`break strcpy`") or line number (e.g. "`break file.c:80`").
- **clear:** the "clear" command will remove the current breakpoint.
- **step, s:** if the current line of code contains a function call, GDB will step into the body of the called function. Otherwise, GDB will execute the current line of code and stop at the next line.
- **next, n:** Execute the current line of code and stop at the next line.
- **continue, c:** continue execution (until the next breakpoint).
- **finish:** Continue to end of the current function.
- **print, p:** print value stored in variable.
- **call:** execute arbitrary code and print the result.
- **watch; rwatch; awatch:** suspend program when condition is met. i.e. `x > 5`.
- **backtrace, bt, bt full:** show stack trace of the current state of the program.
- **disassemble:** show an assembly language representation of the current function.
- **set follow-fork-mode <mode>** (Mac OS does not support this):
  GDB can only debug 1 process at a time. When a process forks itself (creates a clone of itself), follow either the parent (original) or the child (clone). <mode> can be either `parent` or `child`.

The **print** and **call** commands can be used to execute arbitrary lines of code while your program is running! You can assign values or call functions. For example, "`call close(0)`" or "`print i = 4`". (You can actually use **print** and **call** interchangeably most of the time.) This is one of the most powerful features of gdb.

4.2 Helpful Resources

- GDB Cheat Sheet

5 Debugging Example

Take a moment to read through the code for `asuna.c`. It takes in 0 or 1 arguments. If an argument is provided, asuna uses quicksort to sort all the chars in the argument. If no argument is provided, then asuna uses a default string to sort.

```
int partition(char* a, int l, int r){
    int pivot, i, j, t;
    pivot = a[l];
    i = l; j = r+1;

    while(1){
        do
            ++i;
        while(a[i] <= pivot && i <= r);
        do
            --j;
        while( a[j] > pivot );
        if( i >= j )
            break;
        t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
    t = a[l];
    a[l] = a[j];
    a[j] = t;
    return j;
}
```

```
void sort(char a[], int l, int r){
    int j;

    if(l < r){
        j = partition(a, l, r);
        sort(a, l, j-1);
        sort(a, j+1, r);
    }
}
```

```
void main(int argc, char** argv){
    char* a = NULL;
    if(argc > 1)
        a = argv[1];
    else
        a = "Asuna is the best char!";
    printf("Unsorted: \"%s\"\n", a);
    sort(a, 0, strlen(a) - 1);
    printf("Sorted:   \"%s\"\n", a);
}
```

When asuna is run, we get the following output:

```
$ ./asuna "Kirito is the best char!"
Unsorted: "Kirito is the best char!"
Sorted  : "    !Kabceehhiiiorrssttt"

$ ./asuna
Unsorted: "Asuna is the best char!"
Segmentation fault (core dumped)
```

Use the debugging tools to find why `asuna.c` crashes when no arguments are provided.

First, to compile `asuna.c`, run

```
$ gcc -g asuna.c -o asuna
```

The first step in debugging a seg fault is often times seeing which line it occurred in. You might immediately see which line the problem occurs by running the program in `gdb` with `run` or `r`. To get a more holistic view, you can also get the backtrace of the error with `gdb` using the `backtrace` or `bt` command immediately after using `run`.

```
$ gdb ./asuna
(gdb) r  # runs the program fully until the segfault, because no breakpoints are set
(gdb) bt # get backtrace
(gdb) k  # kill the program being run
```

The following is similar to the backtrace you should see when running `backtrace`:

```
Backtrace
#0  0x0000555555554738 in partition (
    a=0x555555554914 "Asuna is the best char!", l=0, r=22) at asuna.c:20
#1  0x00005555555547cc in sort (a=0x555555554914 "Asuna is the best char!",
    l=0, r=22) at asuna.c:34
#2  0x0000555555554870 in main (argc=1, argv=0x7fffffffe0f8) at asuna.c:47
```

Notice that the backtrace points to an error in the `partition` function, specifically the line `a[i] = a[j]`. We can inspect this bug closer now that we know where its located by using `gdb` or `cgdb`. We can either set the breakpoint to be on `partition` or the actual faulting line.

```
(gdb) b asuna.c:20 # set a breakpoint on the faulting line
(gdb) r # runs the program until the breakpoint
(gdb) n # runs the next line, which segfaults
```

At this point, notice that

1. This line performs 2 operations: a read from `a[j]` and a write to `a[i]`.
2. Earlier in the program we already execute a `a[j]` in `partition:12`.
3. If we run `asuna` with the default argument (`"Asuna is the best char!"`) passed in as an user argument, no segfault occurs.

The fact that #1 and #2 are simultaneously true points to a problem with the write to `a[i]`, which is most likely a memory issue. #3 implies that memory is somehow different when using a default argument vs an user provided argument. In `gdb`, we can print the address of the string `a` when using the default argument compared to an user provided argument.

```
(gdb) print a
$1 = 0x4007c4 "Asuna is the best char!"
(gdb) r "Test user argument" # rerun the program with a user arg
The program being debugged has been started already.
Start it from the beginning? (y or n) y
(gdb) print a
$2 = 0x7fffffffe6fa "Test user argument"
```

Notice how the address of the default argument is so much lower than that of the user provided argument. This is because the default argument is in the static region of the program. The segfault occurs because memory in the static region cannot be modified. When a string is declared as part of the program such as in `main:6`, that string is compiled into the code and stored in static memory. See this Stackoverflow post for a more detailed explanation of this bug.

Below we provide a cleaned up and fixed version of the the same program. Our solution is to malloc an array on the heap for the argument to `partition` and `strcpy` the string into that array.

```c
void swap (char* arr, int first, int second) {
        char temp = arr[first];
        arr[first] = arr[second];
        arr[second] = temp;
}

```

```c
int partition(char* arr, int left_bound, int right_bound){
	int pivot = arr[left_bound];
	// Initialize to starting bounds we won't use
	int left_loc = left_bound;
	int right_loc = right_bound + 1;

	while(left_loc < right_loc){
		// Make forward progress on every iteration
		// so use do while loops
		do {
			left_loc++;
		// Find the leftmost elem greater than the pivot
		} while (left_loc <= right_bound && arr[left_loc] <= pivot);

		// Make forward progress on every iteration
		// so use do while loops
		do {
			right_loc--;
		// Find the rightmost elem less than the pivot
		} while (right_loc > left_loc && arr[right_loc] > pivot);
		// If there are elements to switch swap them
		if(left_loc < right_loc) {
			swap (arr, left_loc, right_loc);
		}
	}
	swap (arr, left_bound, right_loc);
	return right_loc;
}

void sort(char* arr, int left_bound, int right_bound){
	if(left_bound < right_bound){
		// divide and conquer
		int split_point = partition(arr, left_bound, right_bound);
		sort(arr, left_bound, split_point-1);
		sort(arr, split_point+1, right_bound);
	}
}

void main(int argc, char** argv){
	const char* no_args = "Asuna is the best char!";
	char* arr = NULL;
	if(argc > 1) {
		arr = malloc (strlen (argv[1]) * sizeof (char));
		strcpy (arr, argv[1]);
	} else {
		arr = malloc (strlen (no_args) * sizeof (char));
		strcpy (arr, no_args);
	}
	printf("Unsorted: \"%s\"\n", arr);
	sort(arr, 0, strlen(arr) - 1);
	printf("Sorted:   \"%s\"\n", arr);
	// Really not necessary because this is main but
	// might as well free all your mallocs
	free (arr);
}
```

6 x86 Assembly

In the projects for this class, you will write an operating system for a 32-bit x86 machine. The class VM (and probably your laptop) use a 64-bit x86 processor (i.e., an x86-64 processor) that is capable of executing 32-bit x86 instructions. There are significant differences between the 64-bit and 32-bit versions of x86. For this worksheet, **we will focus on the 32-bit x86 ISA** because that is the ISA you will have to read when working on the projects. Remember that if you compile programs on your local machine or directly in the class VM (not in Pintos), the result will be in x86-64 assembly.

6.1 Registers

The 32-bit x86 ISA has 8 main registers: `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, and `ebp`. You can omit the "e" to reference the bottom half of each register. For example, `ax` refers to the bottom half of `eax`. `esp` is the stack pointer and `ebp` is the base pointer. Additionally, `eip` is the instruction pointer, similar to the program counter in MIPS or RISC-V.

x86 also has *segment registers* (`cs`, `ds`, `es`, `fs`, `gs`, and `ss`) and *control registers* (e.g., `cr0`). You can think of segment registers as offsets when accessing memory in certain ways (e.g., `cs` is for instruction fetches, `ss` is for stack memory), and control registers as configuring what features of the processor are enabled (e.g., protected mode, floating point unit, cache, paging). **We won't focus on them in this worksheet, but you should know that they exist.** In particular, Pintos sets these up carefully upon startup in `pintos/src/threads/start.S`, so look there if you are interested. Keep in mind that there are special restrictions as to how these registers are used as operands to instructions.

6.2 Syntax

Although the x86 ISA specifies the registers and instructions, there are two different syntaxes for writing them out: Intel and AT&T. Instruction operands are written in a different order in each syntax, which can make it confusing to read one syntax if you are used to the other. For this worksheet, **we will focus on the AT&T syntax** because it is the version used by the toolchain we are using (`gcc`, `as`).

In the AT&T syntax:

- Registers are preceded by a percent sign (e.g., `%eax` for the register `eax`)
- Immediates are preceded by a dollar sign (e.g., `$4` for the constant 4)
- For many (but not all!) instructions, use parentheses to dereference memory addresses (e.g., `(%eax)` reads from the memory address in `eax`)
- You can add a constant offset by prefixing the parentheses (e.g., `8(%eax)` reads from the memory address `eax` + 8)
- Source operands typically precede destination operands, for instructions with two operands.

Instructions are often suffixed by a letter to specify the size of operands. Use the suffix `b` to work with 8-bit *bytes*. Use the suffix `w` to work with 16-bit *words*. Use the suffix `l` to work with 32-bit *longwords* (or *doublewords*). (Analogously, on the x86-64 ISA, append `q` to work with 64-bit *quadwords*). If you omit the suffix, the assembler will add it for you.

Some examples:

- `addw %ax, %bx`: Add the word in `ax` to the word in `bx`, and store the result in `bx`.
- `addl %eax, %ebx`: Add the longword in `eax` to the longword in `ebx`, and store the result in `ebx`.
- `addl (%eax), %ebx`: Add the longword in memory at the address in `eax` to the longword in `ebx`, and store the result in `ebx`.
- `addl 12(%eax), %ebx`: Add the longword in memory at the address `eax` + 12 to the longword in `ebx`, and store the result in `ebx`.
- `subl $12, %esp`: Subtract the constant 12 from the longword in `esp`, and store the result in `esp`.

Notice that you don't need special instructions to load from/store to memory. Some other useful instructions are `and`, `or`, and `xor`. An especially common instruction is `mov`:

- `movl %eax, %ebx`: Copy the longword in `eax` into `ebx`.
- `movl $4, %ecx`: Set the longword in `ecx` to 4.
- `movl 4, %ecx`: Read the longword in memory at address 4 and store the result in `ecx`.
- `movl %edx, -8(%ecx)`: Write the longword in `edx` to memory at the address `ecx` − 8.

For the instructions `lea` and `leal`, which you will find in Pintos, the parenthesis notation for memory works differently. They calculate an absolute memory address given a register and offset.

- `leal 8(%eax), %ebx`: Sets `ebx` to `eax` + 8. You can think of this as setting `ebx` to the memory address that `movl 8(%eax), %ebx` would read from.

6.3 Practice: Clearing a Register

Write an instruction that clears register `eax` (i.e., stores zero in `eax`).

There are various possibilities:

```
xorl %eax, %eax
subl %eax, %eax
movl $0, %eax
```

6.4 Calling Convention

The **caller** does the following:

1. Push the arguments onto the stack, in reverse order. After this step, the top of the stack must be 16-byte aligned — add padding before pushing arguments, if necessary, so that this is true.
2. Push the return address and jump to the function you are trying to call.
3. When the callee returns, the return address is gone but the arguments are still on the stack.

The **callee** does the following, and must preserve `ebx`, `esi`, `edi`, and `ebp`:

1. (Typical, but not required) Push `ebp` onto the stack, and store current `esp` into `ebp`.
2. Compute the return value and store it in `eax`.
3. Restore `esp` to its value at the time the callee began executing.
4. Pop the return address off of the stack and jump to it.

6.5 Instructions Supporting the Calling Convention

- `pushl %eax` is equivalent to:

  ```
  subl $4, %esp
  movl %eax, (%esp)
  ```

- `popl %eax` is equivalent to:

  ```
  movl (%esp), %eax
  addl $4, %esp
  ```

- `call $0x1234`: push the return address (address of the next instruction of the caller) onto the stack and jump to the specified address (address of the callee).
- `leave` is equivalent to:

```
movl %ebp, %esp
popl %ebp
```

- `ret` pops a longword off of the stack (typically a return address) and jumps to it.

`pushal` pushes `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, and `edi` to the stack, and `popal` pops values off of the stack and stores them in those registers. They are useful to switch context or handle interrupts.

6.6 Practice: Reading Disassembly

`file.c`:

```
int global = 0;

int callee(int x, int y) {
  int local = x + y;
  return local + 1;
}

void caller(void) {
  global = callee(3, 4);
}
```

When `gcc` compiles this file, with optimizations off, it outputs:
`file.s`:

```
callee:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %edx
        movl    12(%ebp), %eax
        addl    %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        addl    $1, %eax
        leave
        ret

caller:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   $4
        pushl   $3
        call    callee
        addl    $8, %esp
        movl    %eax, global
        nop
```

```
            leave
            ret
```

What does each instruction do? Mark the prologue(s), epilogue(s), and call sequence(s).

- First three instructions of `callee` are the prologue: save `esp` and allocate space for locals.
- The next two `movl` instructions read the function arguments off of the stack into registers.
- The `addl` instruction computes `x + y`.
- The next `movl` instruction stores the result at `ebp` − 4, the stack memory allocated for the `local` variable.
- The next `movl` reads the value of `local` into a register, and the following `addl` instruction adds one to it. Now the return value is in `eax`.
- The final two instructions are the epilogue: restore `esp`, pop the return address off the stack, and jump to it.
- The first two lines of `caller` are the prologue: save `esp`.
- The next four lines are the call sequence for calling `callee`: set up stack, call function, and clean up stack.
- The next `movl` instruction stores the return value into the address of the `global` variable.
- The `nop` appears to be an artifact of `gcc`—we're compiling with optimizations off, so the compiler doesn't optimize this out (although it would on any other optimization level)
- The last two instructions are the the epilogue: restore `esp`, pop the return address of the stack and jump to it.

6.7 Practice: x86 Calling Convention

Sketch the stack frame of `helper` before it returns.

```
void helper(char* str, int len) {
 char word[len];
 strncpy(word, str, len);
 printf("%s", word);
 return;
}

int main(int argc, char *argv[]) {
 char* str = "Hello World!";
 helper(str, 13);
}
```

```
 13
 str
 return address
 saved EBP
 \0   <--------  word[len]
 !
```

```
d
l
...
l
e
H
```

7 C Programs

7.1 Calling a Function in Another File

Consider a C program consisting of two files:

`my_app.c`:

```
#include <stdio.h>

int main(int argc, char** argv) {
  char* result = my_helper_function(argv[0]);
  printf("%s\n", result);
  return 0;
}
```

`my_lib.c`:

```
char* my_helper_function(char* string) {
  int i;
  for (i = 0; string[i] != '\0'; i++) {
    if (string[i] == '/') {
      return &string[i + 1];
    }
  }
  return string;
}
```

You build the program with `gcc my_app.c my_lib.c -o my_app`.

1. What is the bug in the above program? (Hint: it's in `my_app.c`.) `my_helper_function` is not declared in `my_app.c`, so the compiler (incorrectly) guesses that its return type is `int`. Because `sizeof(int)` = 4 but `sizeof(char*)` = 8 in the Student VM, this results in a segfault.

2. How can we fix the bug? Declare `my_helper_function` with the proper signature above `main`.

7.2 Including a Header File

Suppose we add a header file to the above program and revise `my_app.c` to `#include` it.

`my_app.c`:

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

int main(int argc, char** argv) {
  char* result = my_helper_function(argv[0]);
  printf("%s\n", result);
  return 0;
}
```

`my_lib.h`:

```
char* my_helper_function(char* string);
```

You build the program with `gcc my_app.c my_lib.c -o my_app`.

1. Suppose that we made a mistake in `my_lib.h`, and declared the function as `char* my_helper_function(void);`. Additionally, the author of `my_app.c` sees the header file and invokes the function as `my_helper_function()`. Would the program still compile? What would happen when the function is called? The program would compile but the compiler would not pass an argument to the callee even though it is expecting one, causing it to read some value on the stack (`%ebp` offset by 8).
2. What could the author of `my_lib.c` do to make such a mistake less likely? Also `#include "my_lib.h"` at the top of `my_lib.c`.

7.3 Using #define

Suppose we add a `struct` and `#ifdef` to the header file:

`my_app.c`:

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

int main(int argc, char** argv) {
  helper_args_t helper_args;
  helper_args.string = argv[0];
  helper_args.target = '/';

  char* result = my_helper_function(&helper_args);
  printf("%s\n", result);
  return 0;
}
```

`my_lib.h`:

```
typedef struct helper_args {
#ifdef ABC
  char* aux;
#endif
  char* string;
  char target;
} helper_args_t;

char* my_helper_function(helper_args_t* args);
```

`my_lib.c`:

```
#include "my_lib.h"

char* my_helper_function(helper_args_t* args) {
  int i;
  for (i = 0; args->string[i] != '\0'; i++) {
    if (args->string[i] == args->target) {
      return &args->string[i + 1];
    }
  }
  return args->string;
}
```

You build the program with:

```
$ gcc -c my_app.c -o my_app.o
$ gcc -c my_lib.c -o my_lib.o
$ gcc my_app.o my_lib.o -o my_app
```

Convince yourself that this program outputs the same thing as the one in 7.2.

1. What is the size of the `helper_args_t` struct? 16 bytes
2. Suppose we add the line `#define ABC` at the top of `my_lib.h`. Now what is the size of the `helper_args_t` structure? 24 bytes
3. Suppose we leave `my_lib.h` unchanged (no `#define ABC`). But, suppose we instead use the following commands to build the program:

   ```
   $ gcc -DABC -c my_app.c -o my_app.o
   $ gcc -c my_lib.c -o my_lib.o
   $ gcc my_app.o my_lib.o -o my_app
   ```

   The program will now either segfault or print something incorrect. What went wrong? The code in `my_app.c` sees a different definition of `helper_args_t` than `my_lib.c`, causing them to write/read `string` at different offsets from the pointer to the `args` structure.

7.4 Using #include Guards

Suppose we split `my_lib.h` into two files: `my_helper_function.h`:

```
#include "my_helper_args.h"

char* my_helper_function(helper_args_t* args);
```

`my_helper_args.h`:

```
typedef struct helper_args {
  char* string;
  char target;
} helper_args_t;
```

1. What happens if we include the following two lines at the top of `my_app.c`?

   ```
   #include "my_helper_function.h"
   #include "my_helper_args.h"
   ```

Compiler encounters an error because `helper_args_t` is defined twice.

2. How can we fix this? (Hint: look up `#include` guards.)
Use an `#include` guard. `my_helper_function.h`:

```
#ifndef MY_HELPER_FUNCTION_H_
#define MY_HELPER_FUNCTION_H_

#include "my_helper_args.h"

char* my_helper_function(helper_args_t* args);

#endif
```

Similar for `my_helper_args.h`.