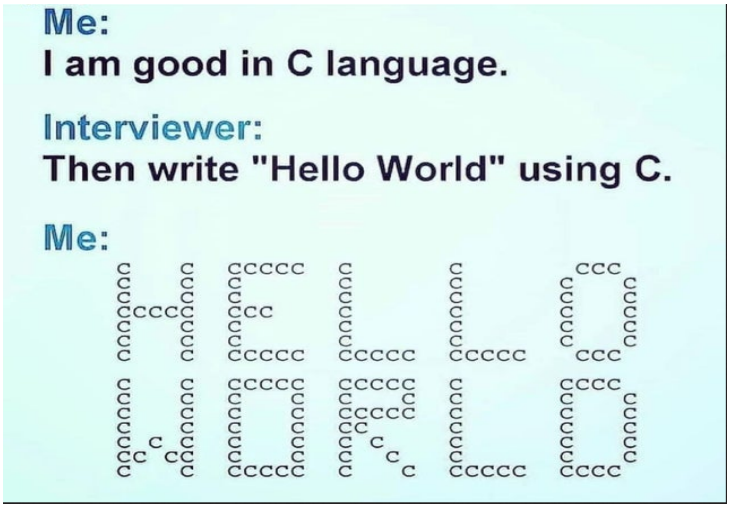
CYBR 401
Lab 4 – Structs
Adam Spanier



TASK 1 – Brains On!

In this lab we are going to examine the *structure* programming construct available to us in C. While languages like Java and Python are readily compatible with Object-Oriented Programming, C predates most OOP applications and thus isn't very OOP friendly. Though this complicates more complex OOP applications in C, the *structure* provides a primitive solution for organizing data into formalized structures in C programming.

So, what is a *structure? A structure (called a struct) is a user-defined data type that allows users to combine data items of different kinds.* In C, a strongly typed language, arrays always contain the *same type of data.* An array can arrange integers, floats, and characters, but no *other data type* can be involved in the array structure. This organization exists due to the differing sizes of different data types.

While this provides a good deal of functionality internally, there are times when the logical grouping of *different data types is useful*. Enter the *structure.* The *structure* exists so that programmers can create *custom logical groupings of data* that better represent the information that is being computed.

For instance, a *user* might have a *user ID, a username, a first name, and a last name.* While it is possible to manage *four different arrays*, each containing one item of data for a record, it would be better logically and functionally to group each record into the same memory space. This can be done using a *structure.*

TASK 2 – Make a File

Before moving forward, let's set up our .c lab file. Make a new C source file and save it as *lab4.c.* In the file, include the *stdio.h* and *stdlib.h* headers and build out a *main* method. Compile the code and make sure it all works using a *printf* statement.

TASK 3 – Structures

Alright! Let's define a structure. Structures use the following format:

```
struct myStruct {
        member definition;
        member definition;
};
```

Let's examine the code above:

struct: **T**he *struct* identifier is a **keyword** that indicates that the following declaration will be a structure definition. This keyword is used any time you declare an instance of a structure, or any time you want to pass a structure as an argument into or out of a function.

MyStruct: Indicates the *variable name* for the structure.

member definition - Indicates a *data type and variable name* internal to the struct.

When *creating* a structure, we can call each member by using *dot notation.* For instance, for the struct:

```
struct Rectangle {
        int width;
        int height;
};
```

We can create a struct via:

```
struct Rectangle r1;
```

And access data inside the rectangle via:

```
r1.width = 1;
r1.height = 2;
```

We can then get information out via:

```
printf("Width: %d\n", r1.width);
printf("Length: %d\n", r1.length);
```

It is also possible to create *arrays* of *structures.* We can do this via:

```
struct Rectangles rects[10];
```

This will create an array of Rectangle pointers of the size 10. we can then add data into this via:

```
rects[0].length = 1;
rects[0].width = 2;
rects[1].length = 1;
rects[1].width = 2;
etc.
```

Or we can use a while loop and a sentinel value:

```
struct Rectangles rects[10];
rects[10].length = -1;
int sent = 0;
int count= 0;
while(num != -1) {
        num = rects[count].length;
        if(num != -1) {
                rects[count].length = 1;
                rects[count].width = 2;
        }
        count++;
}
```

We can also print in the same way:

```
rects[10].length = -1;
int sent = 0;
int count= 0;
while(num != -1) {
        num = rects[count].length;
        if(num != -1) {
                printf("%d\n", rects[count].length);
                printf("%d\n", rects[count].width);
        }
        count++;
}
```

TASK 4 – Overview and Macros

In this lab we are going to define a *very simple structure*, declare an instance, populate the data, print the data, and finally free all allocated memory.

In your lab4.c document, **define** three macros:

- WELCOME - "\nWelcome to the Process Manager\n"
- END "\nEnding Process Manager\n\n"
- SIZE 10

Remember, these macros are located just below your **include** statements.

TASK 5 – Function Prototypes

Create the following function prototypes:

Function Name	Return	Parameters
PrintWelcome	void	None
MakeProcess	struct process	None
getID	int	None
getName	char *	None
printProcess	void	struct process
printEnd	void	None
freeProc	void	struct process

These function prototypes are located just below your Macro Definitions.

TASK 6 – Our Structure

Alrighty! Now we're ready to declare our s*tructure*. In this lab, we'll be working with a ***process*** structure. Each process has an *ID* represented as an *integer* and a *name* represented as a *string*.

Let's see if we can define the structure. First, you'll use the **struct** keyword, then give it the name **process**, create braces to contain the data, and inside the braces add an *int* variable called *id* and a *char* * variable called *name*. Finally, we'll want to end the structure definition using a ;.

Try it yourself before looking at the answer code on the LAST PAGE. If you need help, look at the *Rectangle* example I provided above.

Structures are always defined *just below* the function prototypes.

TASK 7 – printWelcome and printEnd

Let's get the two easy functions done with first. All these functions will do is call the *printf()* function using the *macros* we defined above.

Create the function header for *printWelcome* with a *return type* of *void* and no *parameters*. Inside the function, call *printf*() using the *WELCOME Macro*. Use the "%s" format string and call the *WELCOME Macro* directly from the second argument: *printf("%s", WELCOME).*

Do the exact same thing with *printEnd*, only use *END* instead: *printf("%s", END).*

Try it yourself before looking at the answers on the last page. These are easy. Don't overthink it.

Keep in mind, you should be trying to call all of these functions from the *main* method to verify their functionality before moving on.

TASK 8 – getID

Before we move on, we're going to need a couple of helper functions. The first of these is called *getID*. It takes in *no arguments* and *returns* an *integer*.

Inside the function, create an *int* called *num*. *Print* the statement, "Enter ID: " (Note the space at the end). Use the *scanf()* function to read the *user input integer* into the *num* variable ("%d", &num). Return the *num* variable.

Remember, when we read an *integer* from a user, we need to get it's *pointer* in order for *scanf* to save it. We get the pointer of *int num* via the use of the & character. This would look like:

```
scanf("%d", &num);
```

Try it before looking at the answer. See if you can get it to work.

TASK 9 – getName

The *getName* function works much like the *getID* function, only we're getting a *string* rather than an *integer*.

Remember, this is more complicated than getting an integer. Whenever we create a string, we have to first create a *char* * then we have to allocate memory for the *char* * and finally, we need to *NULL* terminate the last slot of the array.

The *getName* function will return a *char* * and take *no* args. In the function, we first define a *char* * called *temp* and assign a *malloc* call to this variable to create memory for the array. In the *malloc* call, we use the *SIZE* macro as the number of slots in the array and we multiply this times *sizeof(char)*. This first step happens on one line and will look like this:

```
char * temp = malloc(SIZE * sizeof(char));
```

After allocating memory, set the *last* slot in the allocated memory space to NULL ('\0'). We do this by referencing the *variable name* (*temp*) and using the square brackets [] to index to the last slot. We can do this using the *SIZE* macro.

```
temp[SIZE] = '\0';
```

Next we need to *print* the statement "Enter Name: " and use the *scanf* function to read console input into temp (*"%s", temp*). Then we return *temp* from the function.

Do it yourself! Answers are on the last page.

TASK 10 - makeProcess

Now we need to get a bit deeper. In this function, we will create an *instantiation* of the ***process*** structure, call both *getID* and *getName* and return the *structure* with it's data intact.

First, we'll do some CLI prettiness. Print the following prompt "\n****** GET INPUT ******\n\n" to the console using the *printf* function. This will add a few lines, indicate that we're taking input, and add a few more newlines for the *getID* and *getName* prompts.

After printing the prompt above, we'll need to create an instance of the process struct. To do this we'll use the following code:

```
struct process p1;
```

Please note, *ANY TIME* you indicate a *struct type*, you have to use the *struct keyword.* This includes the creation of *struct* definitions and instances, function headers, and function prototypes.

Next, let's set *p1's id integer* to the return value of *getID*. We do this by calling into *p1* using dot notation and using the assignment operator with a *getID* call. This looks something like this:

```
p1.id = getID();
```

Do the same with *p1.name* and *getName*.

When you've set both the *ID* and *Name* variables inside the *Process* structure using *getID* and *getName*, return *p1*.

If you are having any trouble, see the last page for answers. BUT! Try it yourself first!

TASK 11 – printProcess

Now that we've created a ***process***, we should also build out a function that can print one. To do this, we'll use the *printProcess* function. This function will return a void datatype and take in a *struct process* input parameter. Call the input structure *p1*.

In the function, print the following string, "\n****** PRINT OUT ******\n\n". Then print the process ID using the following format string, "Process ID: %d\n". Finally print the process Name using this format string, "Process Name: %s\n". Note, you'll need to provide the right *variable* input into the %d and %s format strings above.

There is no return from this function.

TASK 12 – freeProc

The final custom function we'll be building is a function that *frees* up the allocated memory inside of a process. This function returns void and takes in a *struct process.* Call the incoming parameter *p1*. In the function simply call the free() function with the *p1.name* variable.

TASK 13 – Main!

For the final step, let's call the program in the appropriate order. The program should run as follows:

1. Call the welcome function
2. Create a *process* called *p1: struct process p1;*
3. Call the *makeProcess* function and catch its output in *p1*
4. Call the *printProcess* function with *p1* as an arg.
5. Call *freeProc* with *p1* as the arg
6. Call the *printEnd* function
7. Return 0

Write the main function by adding a function call then testing. Each function call builds on the calls before, so build from 1 to 7, not in a random order.

Debug as you go. ***Using an ID of 1234 and a Name of SVCHOST,*** Your final console output should look like this:

```
Welcome to the Process Manager

****** GET INPUT ******

Enter ID: 1234
Enter Name: SVCHOST

****** PRINT OUT ******

Process ID: 1234
Process Name: SVCHOST

Ending Process Manager
```

TASK 14 - Submit

When you have your code working properly, save the file and submit your *lab4.c* file on Canvas. If you have any struggles or questions, shoot me an email!

Be sure your file layout is as follows:

1. Include statements
2. Define statements
3. Function prototypes
4. Structure statements
5. Main method
6. Function defined in same order as function prototypes

ANSWERS

NOTE: These solutions aren't the only viable solutions. If you find another way to do it, please let me know! I love seeing different approaches.

TASK 4

```
#define WELCOME "\nWelcome to the Process Manager\n"
#define END "\nEnding Process Manager\n\n"
#define SIZE 10
```

TASK 5

```
void printWelcome();
struct process makeProcess();
int getID();
char * getName();
void printProcess(struct process);
void printEnd();
void freeProc(struct process);
```

TASK 6

```
struct process {
  int id;
  char * name;
};
```

TASK 7

```
void printWelcome(){
  printf("%s", WELCOME);
}
```

```
void printEnd() {
  printf("%s", END);
}
```

TASK 8

```c
int getID() {
  int num;
  printf("Enter ID: ");
  scanf("%d", &num);
  return num;
}
```

TASK 9

```c
char * getName(){
  char * temp = malloc(SIZE * sizeof(char));
  temp[SIZE] = '\0';
  printf("Enter Name: ");
  scanf("%s", temp);
  return temp;
}
```

TASK 10

```c
struct process makeProcess() {
  printf("\n****** GET INPUT ******\n\n");
  struct process p1;
  p1.id = getID();
  p1.name = getName();
  return p1;
}
```

TASK 11

```c
void printProcess(struct process p1){
  printf("\n****** PRINT OUT ******\n\n");
  printf("Process ID: %d\n", p1.id);
  printf("Process Name: %s\n", p1.name);
}
```

TASK 12

```c
void freeProc(struct process p1){
  free(p1.name);
}
```

TASK 13

```c
int main(){
  printWelcome();
  struct process p1;
  p1 = makeProcess();
  printProcess(p1);
  freeProc(p1);
  printEnd();
  return 0;
}
```