CMSC 216

Introduction to Computer Systems

Jose Valdivia

Fall 2022

https://www.cs.umd.edu/class/fall2022/cmsc216/

Contents

1 Abstract **2**

2 Projects **3**

2.1 Project 1: Grade Calculator . . . 3
2.2 Project 2: Document Manager . . . 10
2.3 Project 3: User Interface . . . 17
2.4 Project 4: Calendar . . . 21
2.5 Project 5: Assembly Language Programming . . . 36
2.6 Project 6: Senior Shell . . . 37

3 Assignments **37**

3.1 Assignment 1 . . . 37
3.2 Assignment 2 . . . 37
3.3 Assignment 3 . . . 37
3.4 Assignment 4 . . . 37
3.5 Assignment 5 . . . 37
3.6 Assignment 6 . . . 37

---

1 Abstract

Here are my notes for CMSC 216 Projects and Assignments taken during my time at the University of Maryland in the Fall 2022. The instructor for this course was Dr. Ilchul Yoon.

This guide is designed to make it easy to visualize the concepts and guide you through each assignment. Although this guide doesn't cover any coding, it involves **pretty good ideas on how to start** the projects and assignments. **Blue** colored texts represent potential concept question on exams.

About this pdf: This pdf guide was created in LaTeX using Overleaf website. I used the template from Alex Reustle, which can be found on his Github repository at
`https://github.com/Areustle/CMSC351SP2016FLN`

About me: I am passionate about computer science and the things we can do with it. As you may know, I am a visual learner. I always believe that the best way to learn a hard concept is to actually picture it. Drawings and graphs are my best friends when it comes to understanding hard stuff. Also, it makes it so much fun when learning, isn't it?

Regrets: I regret not going to discussion sessions more often. If you ever found it not useful, try to find a discussion session that best fit to you. Never hesitate to ask questions when you don't understand a concept. Try to meet your classmates. Do study sessions. Make connections. Sharing knowledge/discussing concepts is key to succeed in this class.

Advices: There are so many resources that UMD offers. I highly recommend you try I4C or ASTS tutoring. If lectures is not for you, I recommend watching discussion recordings or YouTube videos. I will share the playlist that I used for the semester.

- `https://umdtutoring.mywconline.com/index.php` (ASTS)
- `https://inclusion.cs.umd.edu/programs/#tutoring` (I4C)
- TO-DO

Don't ever get frustrated if you get a bad grade on quizzes. Quizzes are way harder than midterms, believe me. Each worth 2.5% of your total grade, so do not worry. You only have 15 min to finish: a coding question (very lengthly) and 1-3 fill in the blank questions. Manage your time wisely. Remember, no one is perfect, it is fine to fail, but we must learn from our failures.

Coding by hand is annoying, I know, so get used to it. Practice coding questions from worksheets/past exams as much as you can. Always leave 4 to 6 empty lines for your variables declaration. Coding questions sometimes worth more than fill in the blank. So, I'd suggest doing these coding questions first and leaving the others for last.

How do I prepare for exams/quizzes? I start studying 1 week or 4 days before the exam/quizz. I print the past exams/quizzes and do the coding part by hand. If I don't understand a topic, I will discuss it with a TA or group study. **This should be a rule, before any incoming exam/quizz, always do the assigned project/exercise since 100% comes in the test.** Doing past exams should be enough to get a good grade.

2 Projects

2.1 Project 1: Grade Calculator

Overview

The program read the courses from user to calculate:

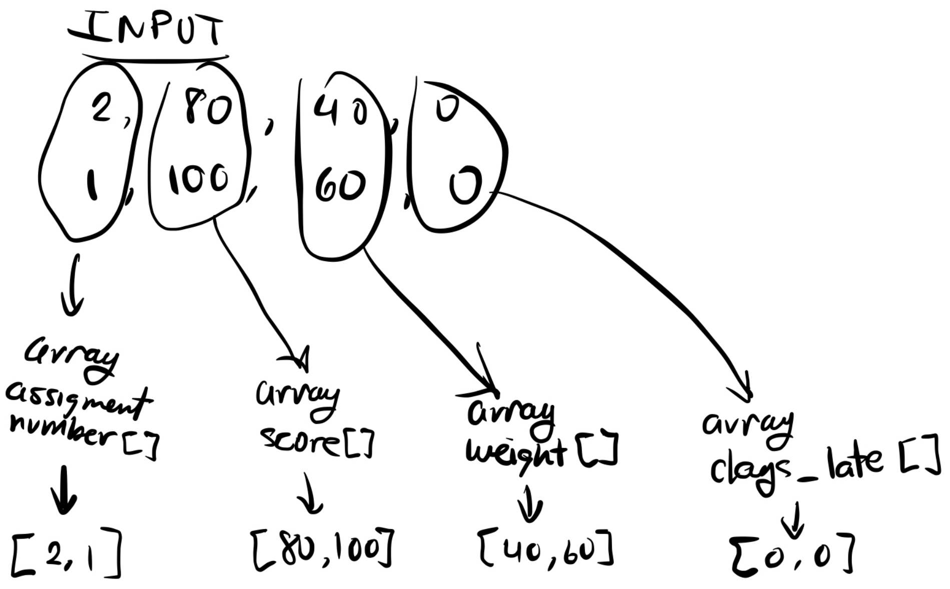
1. numeric score
2. mean
3. deviation

Objectives

- Manipulate **Arrays**
- Use functions
- Sorting output **(Bubble Sort)**

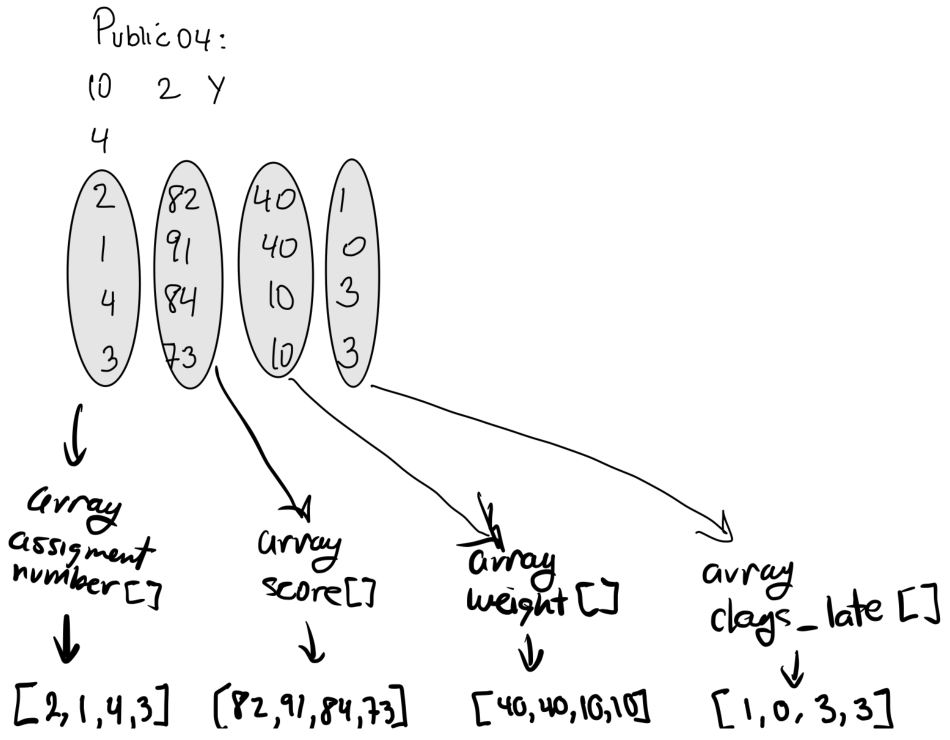
Getting User Input

The way how to store out data is important for this project. We'll use arrays of **int** for: *assignment numbers*, *scores*, *weights*, and *days late*. Let's give an example graphically.



At this point, I assumed you already retrieved: *points per day late* **(int)**, *number of assignments* **(int)**, and *statistical info*(**character**) using **scanf()**.

However, to populate our arrays, we'll need to use **scanf()** inside of a **for-loop**, looping # of assignments times. Let's see another example from public04.



The number of assignments is 4. So, we gonna loop 4 times to retrieve the data and store into 4 arrays, as displayed in the picture above.

<u>**Think about this way**:</u>
element1, element2, element3, element4 ← Loop 0
element1, element2, element3, element4 ← Loop 1
element1, element2, element3, element4 ← Loop 2
element1, element2, element3, element4 ← Loop 3

In loop **0**, element1 will be stored in *assignment number* array at index [**0**], element2 in *score* array at index [**0**], element3 in *weight* array at index [**0**], element4 in *days late* array at index [**0**].

In loop **1**, element1 in index [**1**], element2 in index [**1**], element2 in index [**1**], element2 in index [**1**].

...

Do you see the pattern?

Drop Course & Numeric Score

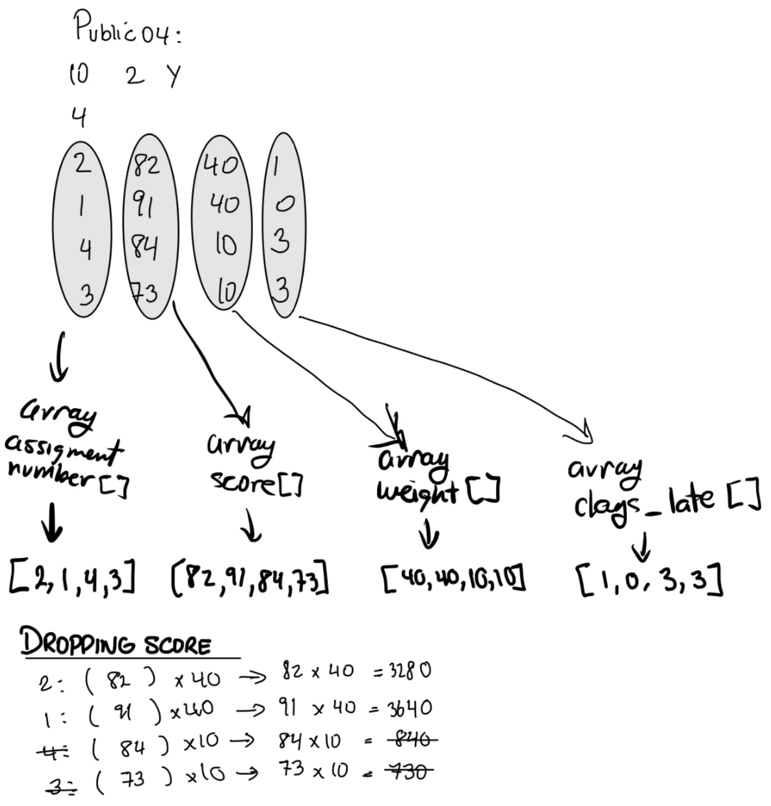
At this point, I assumed you already checked:

- **sum weights** to be equal 100
- **# assignments** to be less than **max # of assignments**.

Let's continue. The next step is to drop courses with lowest total score. This is how we calculate the total score based on its score and weight.

$$\text{Assignment \#: } score \times weight = totalscore$$

From public04 figure, let's visualize what dropping lowest total would look like.



In this case, our user chose 2 courses to drop. We see that **840** and **730** are indeed the lowest total scores.

Alright, looks straightforward. We just need to compute *score* × *weight* and store the *totalscore* in a new array. And see which are the smallest numbers.

$$\begin{bmatrix}3280 & 3640 & 840 & 730\end{bmatrix}$$

But, how are we gonna actually find the lowest scores in terms of code? Well that's your job :) but I might give you some hints.

Ok, so far our arrays would look like this:

$$\begin{bmatrix} 2 & 82 & 40 & 1 & 3280 \\ 1 & 91 & 40 & 0 & 3640 \\ 4 & 84 & 10 & 3 & 840 \\ 3 & 73 & 10 & 3 & 730 \end{bmatrix}$$

There is an algorithm to find these smallest numbers in an array and gets the position (index) of them. *(Note: Good practice for coding interviews)*

If you find the i (index) of the course with the lowest total score, that is a "big win". With that *index* you'll be able to access to that course's arrays: *assignment_numbers*, *scores*, *weights*, and *days_late*. Let's see what I mean.

$$\begin{bmatrix} 2 & 82 & 40 & 1 & 3280 \\ 1 & 91 & 40 & 0 & 3640 \\ 4 & 84 & 10 & 3 & 840 \\ 3 & 73 & 10 & 3 & 730 \end{bmatrix}$$

`The first lowest total score is 730 located at` $i = 3$
`Where` i `represents each row/a course.`

Ok, I know that the courses dropped are at index 2 and 3 but let's assume you found the index of the first lowest score, $i = 3$ **(730)**. In order to drop that course, you will need to set its *score* and *weight* to 0. How so?

$$\begin{bmatrix} 2 & 82 & 40 & 1 & 3280 \\ 1 & 91 & 40 & 0 & 3640 \\ 4 & 84 & 10 & 3 & 840 \\ 3 & 0 & 0 & 3 & 730 \end{bmatrix}$$

$$scores[i] = 0 \text{ and } weights[i] = 0$$

The same process when you find the 2nd lowest score, $i = 2$ **(840)**.

$$\begin{bmatrix} 2 & 82 & 40 & 1 & 3280 \\ 1 & 91 & 40 & 0 & 3640 \\ 4 & 0 & 0 & 3 & 840 \\ 3 & 0 & 0 & 3 & 730 \end{bmatrix}$$

$$scores[i] = 0 \text{ and } weights[i] = 0$$

Important: I would suggest copying scores and weights into new arrays before "dropping" since you will later need the original arrays back.

We almost there! What we did so far was:

- Find each lowest scores at index i
- Set scores$[i] = 0$
- Set weight$[i] = 0$

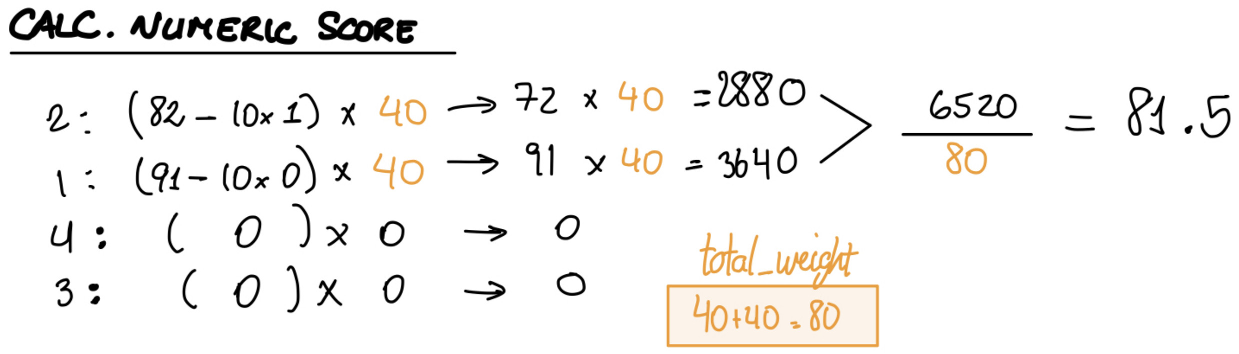
The last step is to calculate the **"Numeric Score"**.

$$(scores[i] - (points\_per\_day\_late \times days\_late[i])) \times weights[i] \mathrel{+}= total\_fixed$$

$$\frac{\sum_i^{na} total\_fixed}{\sum_i^{na} total\_weight}$$

`Where i = 0,` na `represents the # of assignments.`

Don't get it? Ok I got you. This is how would look like.



Now that you got the idea. This shouldn't be that hard to code.

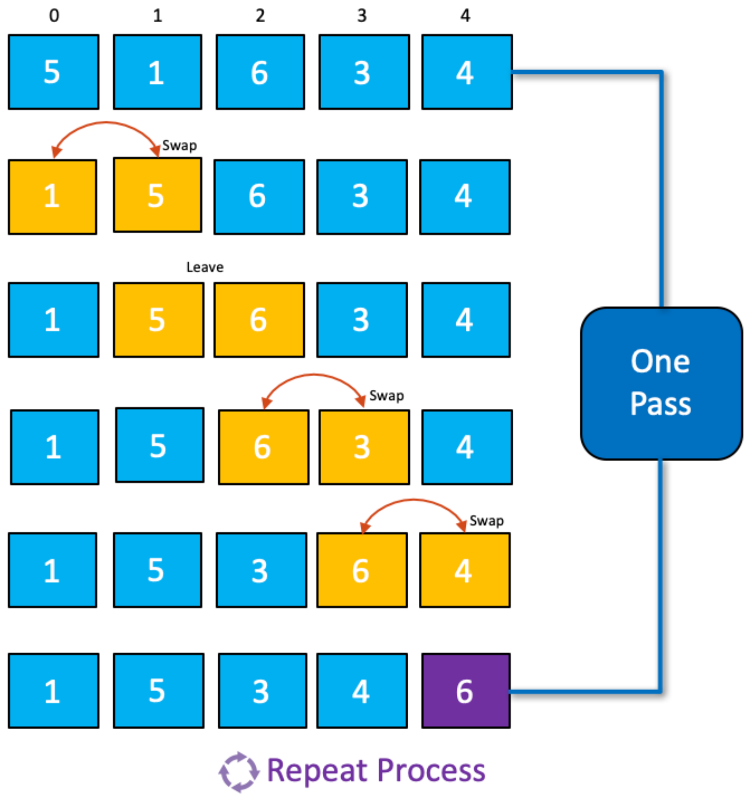
You made it! You dropped courses and calculated the numerical score of the remaining ones.

Congrats!

Sort Output Array

Let's introduce the **Bubble Sort**. Well, Bubble Sort just loop through each element of an array and checks:

```
If current_element > next_element then
    swap current_element and next_element
```



Suppose this row is your *assignment_number* array.

Pseudocode:

```
Data: Input array A[]
Result: Sorted A[]

int i,j;
N = length(A);

for j=0 to N-1 do
    for i=0 to N-1 do
        if A[i] > A[i+1] then
            temp = A[i];      // temp big number, A[i+1] small number
            A[i] = A[i+1];    // Place small number to current position
            A[i+1] = temp;    // Place big number to the next position
        endif
    endfor
endfor
```

Explanation: The algorithm will loop until the second last of the array, that's why there is $N-1$ for i.
For j, the $N-1$ just make sure that it will repeat the process enough times to sort the array.

Alright, now is your turn to make your own bubble sort! Don't forget to swap the others arrays at the same time too. Your final output array should look like this:

$$\begin{bmatrix} 1 & 91 & 40 & 0 & 3640 \\ 2 & 82 & 40 & 1 & 3280 \\ 3 & 73 & 10 & 3 & 730 \\ 4 & 84 & 10 & 3 & 840 \end{bmatrix}$$

Note: You notice that here we didn't use the arrays with dropped courses.

Calculate Mean

Here I won't go through detail explanation since this is straightforward. However, you should know the formula to calculate the **mean**.

$$\frac{\sum_i^{na}(scores[i] - (points\_per\_day\_late \times days\_late[i]))}{na} \tag{1}$$

Where na represents the # of assignments.

There is a change in this formula. We need to apply the penalty to the scores. Also, you will hear the word **Mean** very often in `STAT 400`, if you ever take it of course. Nevertheless, you should get used to math notations such as summations/integrals since they will be used very often in your next course `CMSC 351 Algorithms` 🤯. Wow! So excited, isn't it? You already learned one of many algorithms **Bubble Sort**.

Calculate Standard Deviation

The formula for **Standard Deviation** or σ is:

$$\sqrt{\frac{\sum_i^{na}(scores[i] - mean)}{na}} \tag{2}$$

Where na represents the # of assignments.

2.2 Project 2: Document Manager

Overview

The program reads a document and fill up with lines and paragraphs. The program does also the following:

1. initialize document
2. add paragraph
3. add line
4. highlight texts
5. remove texts
6. replace texts

Objectives

- Practice **C structures** similar to **Objects in Java**
- Manipulate array of chars a.k.a. **Strings**
- Work with **Pointers**

Makefile

I won't go through much detail about `Makefile`, but don't worry, I will explain this concept in next projects. All you need to know is that `Makefile` saves you so much time when compiling your program. You won't see its effectiveness right now because our program is small, but once we go to another big project, you will see its usefulness and you will use it a lot, I promise :)

On the command prompt, these two commands are equivalent.

```
make all = gcc document.c public01.c -o public01
```

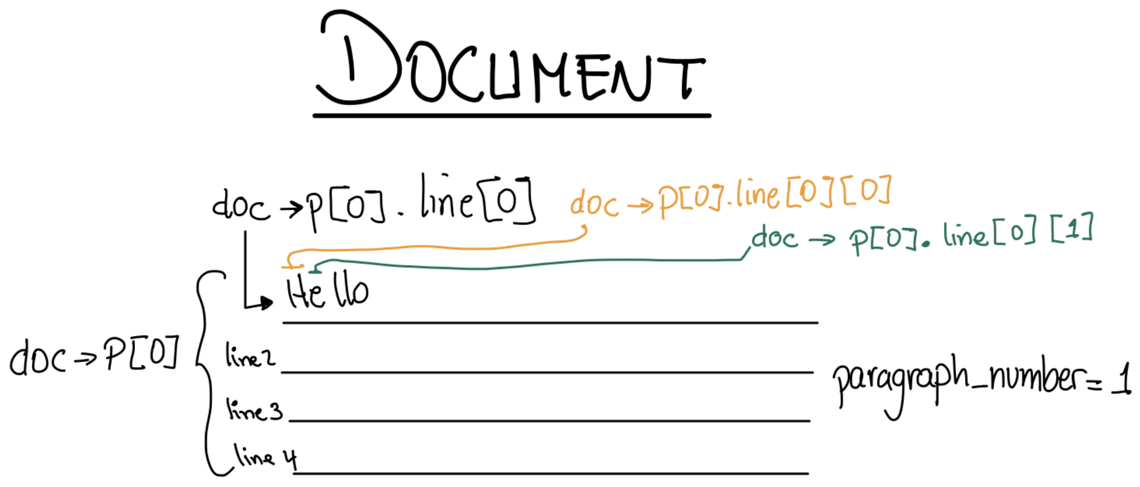
With two words `"make all"`, we can replace all the `gcc` line.
Think about it like a **shortcut** that will allow you to create all your executables instantly. And if you ever update any of your files, `"make all"` will do the job for you. So convenient!

Be aware of two `Makefile` commands:

- make all → Creates/update all executables
- make clean → Erase all executables

How to Start

First, let's think about what actually a document looks like.



A **document** has the following:

- Paragraphs
- A Name
- Number of Paragraphs

A **paragraph** has the following:

- Lines
- Number of Lines

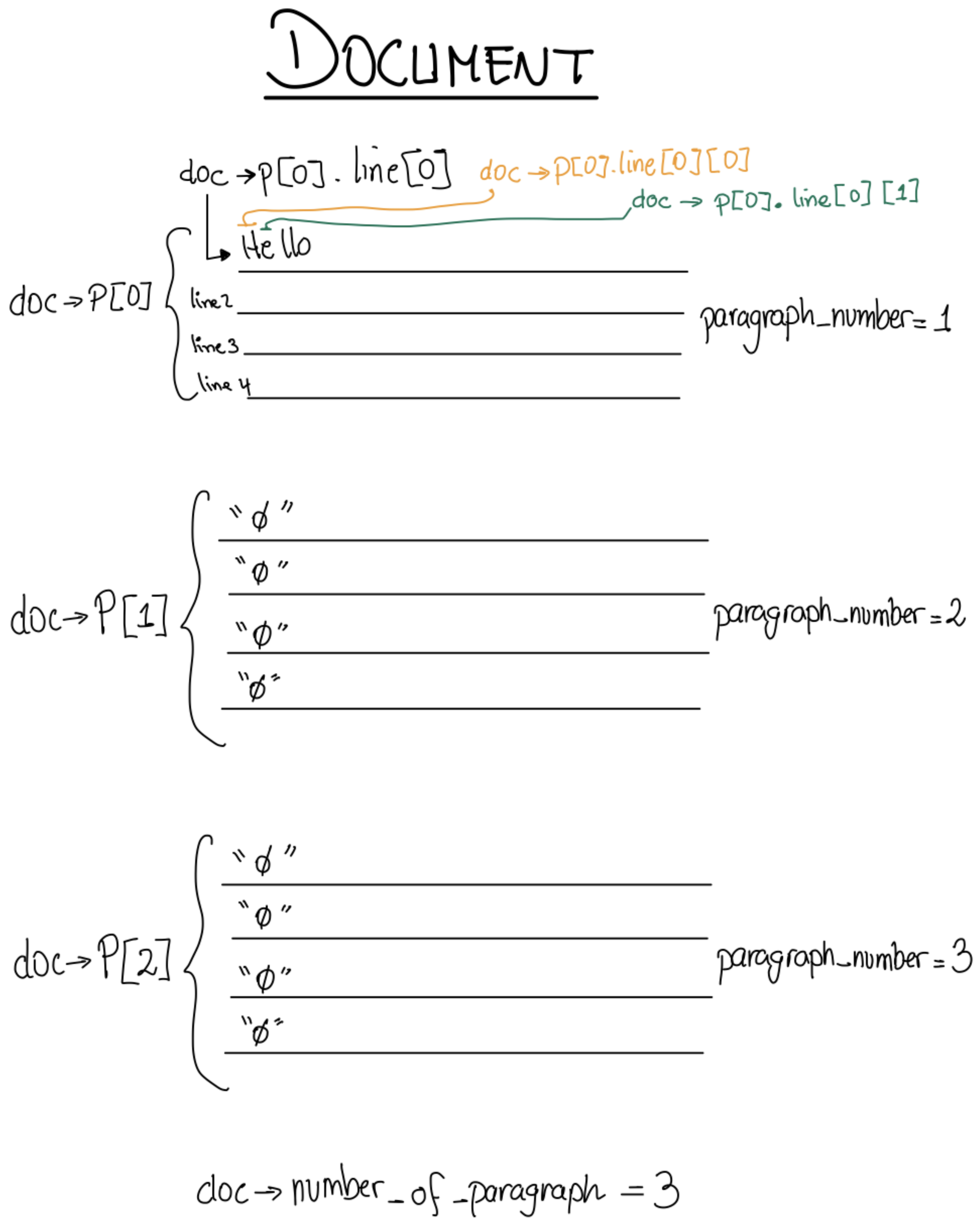
OK, 4 things we can notice from this document:

Our first paragraph is stored at the first index of the array `"doc -> p[0]"`
Our first string/line is stored at the first index of the array `"doc -> p[].line[0]"`
Our first char of a string is store at the first index of the 2D-array `"doc -> p[].line[][0]"`
Our first paragraph is denotated as `"paragraph number 1"`

Tips: Always store your data at the beginning of an array, at index $i = 0$.
Make sure you read the documentation **very carefully**. I am serious. Any small detail needs to be addressed or you will deal with debugging code which is not the fun part of coding. If possible, copy and paste the whole description of a function as a comment, break it down, and address each of them.

Alright, I think we have a pretty good insight of how the data in our document is structured.

Let's see a complete Document.



Now, I think you're in good shape for the following functions: `init_document`, `reset_document`, `print_document`, `get_number_lines_paragraph`.

I'll see you in the next section 👋

Add Paragraph/Line After

Suppose we have three paragraphs: **paragraph 1** with 3 lines, **paragraph 2** with 2 lines, and **paragraph 3** with 3 lines. Let's visualize it:

Document
Paragraph 1, Line 1
Paragraph 1, Line 2 ← `doc->paragraphs[0]`
Paragraph 1, Line 3

Paragraph 2, Line 1 ← `doc->paragraphs[1]`
Paragraph 2, Line 2

Paragraph 3, Line 1
Paragraph 3, Line 2 ← `doc->paragraphs[2]`
Paragraph 3, Line 3

If the function `"add_paragraph_after(doc, 1)"` takes 1 as `paragraph_number`, it means that you will add a **new paragraph** after **paragraph 1**. That is, between **paragraph 1** and **paragraph 2**.

Let's see what I mean:
Document
Paragraph 1, Line 1
Paragraph 1, Line 2 ← `doc->paragraphs[0]`
Paragraph 1, Line 3

New Paragraph, Empty Line 1 ← `doc->paragraphs[1]`

Paragraph 2, Line 1 ← `doc->paragraphs[2]`
Paragraph 2, Line 2

Paragraph 3, Line 1
Paragraph 3, Line 2 ← `doc->paragraphs[3]`
Paragraph 3, Line 3

The idea here is to switch forward **paragraph 1** and **paragraph 2** so we make **"room"** for the new empty paragraph in our Document.

How would it look like in terms of code?
Well, in terms of code, you will have to copy and paste paragraph 2 and paragraph 3 to their next index in the **paragraph array** and then create a new paragraph. Let's visualize what I mean.

Step 1: Locate where we're gonna add a new paragraph. Let' say after **paragraph 1**. In order words, between paragraph 1 and 2.

<u>Document</u>
Paragraph 1, Line 1
Paragraph 1, Line 2 ← `doc->paragraphs[0]`
Paragraph 1, Line 3

Paragraph 2, Line 1 ← `doc->paragraphs[1]`
Paragraph 2, Line 2

Paragraph 3, Line 1
Paragraph 3, Line 2 ← `doc->paragraphs[2]`
Paragraph 3, Line 3

Step 2: Copy and Paste **"Paragraph 3"** to the next index. Same for **"Paragraph 2"**.

<u>Document</u>
Paragraph 1, Line 1
Paragraph 1, Line 2 ← `doc->paragraphs[0]`
Paragraph 1, Line 3

Paragraph 2, Line 1 ← `doc->paragraphs[1]`
Paragraph 2, Line 2

Paragraph 2, Line 1 ← `doc->paragraphs[2]`
Paragraph 2, Line 2

Paragraph 3, Line 1
Paragraph 3, Line 2 ← `doc->paragraphs[3]`
Paragraph 3, Line 3

Step 3: Empty the paragraph that was meant to be a **new paragraph**, which in this case is `"doc->paragraphs[1]"`

<u>Document</u>
Paragraph 1, Line 1
Paragraph 1, Line 2 ← `doc->paragraphs[0]`
Paragraph 1, Line 3

New Paragraph, Empty Line 1 ← `doc->paragraphs[1]`

Paragraph 2, Line 1 ← `doc->paragraphs[2]`
Paragraph 2, Line 2

Paragraph 3, Line 1
Paragraph 3, Line 2 ← `doc->paragraphs[3]`
Paragraph 3, Line 3

Here we go! We added a new paragraph after a specific paragraph. Now, it is your turn to convert it to code. Also, **add_line_after** has the exact same functionality.

Append Line

This is kinda straightforward. Just add a new line at the end of a paragraph.

Hint: Each paragraph tracks its number of lines.
It can help you to find the **last line** and **its index** of a paragraph. Also, you are allowed to reuse functions, for instance, **“add_line_after”** and other ones.

Replace/Highlight/Remove Text

For this section, I assume that you know about **Pointers** and its functionality. The idea here is to find a piece of text inside of our paragraph. Pointer is useful here because we can locate this piece of text and make a variable that references to it. Thus, we can modify it in any part of our code and do whatever we want to it, such as replacing, highlighting, and removing it. Pointers are so useful!

Let’s see an example:

<u>Document</u>
The first course you need to take
is cmsc131. This course will be
followed by cmsc132 (which is also based on Java). ← `here`

The next course you will take is cmsc216.
This course relies on C.

Ruby and Ocaml will be covered in cmsc330

Explanation: Let’s say we want to replace the word **“cmsc132”** to **“cmsc216”** located at `doc->paragraphs[0].lines[2]`.
First we would need to loop thru each paragraph and each line. At each line, we need to find the word **“cmsc132”**:

→ followed by **cmsc132** (which is also based on Java).

Once located with a pointer, one strategy to replace the target/word is to break down the sentence after the target and before the target. The null character "\0” is useful here.
Like this:

→ followed by \0

→ **cmsc132**

→ (which is also based on Java).

Alright, next step is to change the word cmsc132 to cmsc216.

→ followed by \0

→ **cmsc216**

→ (which is also based on Java).

Finally, putting all together and storing back to `"doc->paragraphs[0].lines[2]"`:
→ followed by **cmsc216** (which is also based on Java).

There it go! I hope you got the idea and now is your turn to back engineer it into code language. Remember, pointers are useful but also dangerous. If you modify a pointer, it also modifies the main source.

Hint: Three powerful functions in string library:

- **strstr()**
- **strcat()**
- **strcpy()**

I'll see you in the next section 👋

2.3 Project 3: User Interface

Overview

The program uses project 2 to create a user interface (like a shell where you input commands and does stuff) for the document manager. The user can call the program with either two commands:

1. user_interface
2. user_interface `filename`

The program initialize document and perform operations instructed by the commands.

Objectives

- Practice text manipulation/parsing
- Understand Standard Input (stdin), Standard Output (stdout), Standard Error (stderr)
- Understand `argc`, `argv[]` parameters from main
- Work with **File I/O**

Getting Ready

First of all, bring your `document.c` inside your project 3 directory (folder).

Second, you will be coding in a new file called `user_interface.c`. Add this line:

```
#include "document.h"
```

so we can reuse the functions from `document.c`.

We will go in depth about how this works and dependecies in the next project. For now just be aware that by using `#include + "filename"`, we can use functions from another files.

Command Line Arguments

Have you ever wonder why `main()` in C and other programming languages take parameters? Well, the reason is that not only we can run our executable file but also we (users) can interact with the program by sending inputs into it and perform some specific tasks. Let's see what I mean by this.

Besides of running our executable in the terminal, let's say:

```
> a.out
```

We can send inputs/commands into this `"a.out"`. For example:

```
> a.out do_stuff1 do_stuff2
```

a.out will received these two inputs and save them into the parameters from `main()`. **Our job is to make the program read these inputs from main's arguments and perform some specific tasks.**

In C, main() takes two parameters:

$$\texttt{int}\ \texttt{main}\ (\texttt{int}\ \overset{\text{argument count}}{\texttt{argc}}\ ,\ \texttt{char}\ \overset{\text{arg. vector}}{\texttt{*argv[]}})\ \{\ \}$$

$$\text{CMD} \Rightarrow \underbrace{\text{a.out}}_{\text{argv[0]}}\ \underbrace{2}_{\text{argv[1]}}$$

- `int argc`: represent the number of arguments/the length of **argv**
- `char *argv[]`: a pointer to an array of chars a.k.a **Array of Strings**

From the CMD (command) above, let's see what our variables are holding:

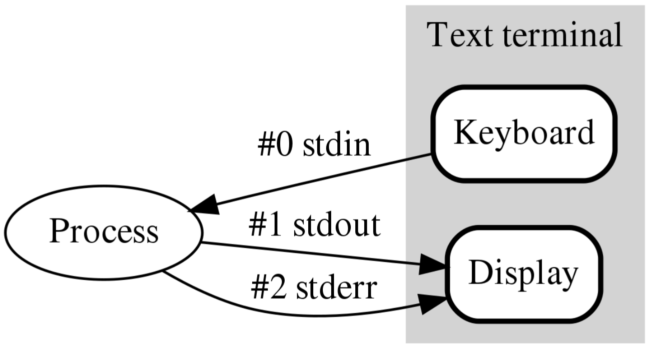
- `argc = 2`
- `argv = {"a.out", "2"}`
- `argv[0] = "a.out"`
- `argv[1] = "2"`

Note: Be aware that if we send a number as input, the number will be stored as a **string**.

Standard I/O

The Standard I/O are basically streams/communications channels for which data travels between the program and terminal.

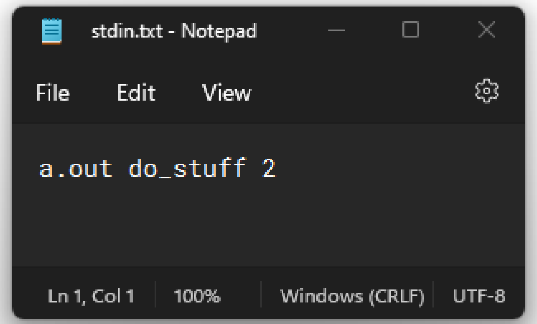
The three input/output (I/O) connections are: standard input (stdin), standard output (stdout), and standard error (stderr).



By default, our program (process) uses **stdin** to read input (from keyboard) and uses **stdout** and **stderr** to output a message to the terminal (display). Unless we explicity redirect input/output to another communication channel/file.

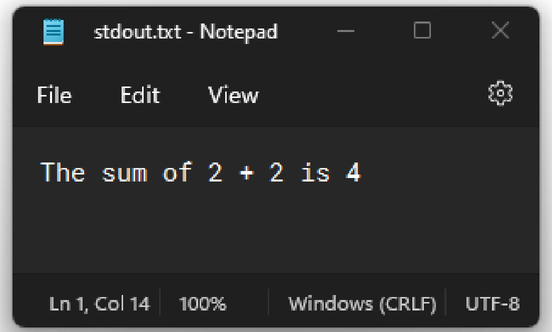
The concept is broad and we will review redirection in project 6, but the easy way to understand is to think Standard I/O as a file.

- stdin: a file that contains whatever input from the keyboard



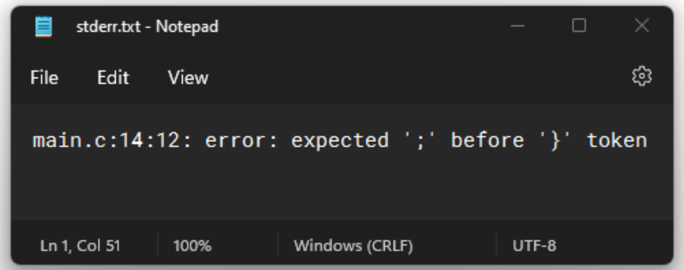
```
a.out do_stuff 2
```

- stdout: a file that contains output from program



```
The sum of 2 + 2 is 4
```

- stderr: a file that contains error message from our program



```
main.c:14:12: error: expected ';' before '}' token
```

I am sure that your professor will go over this with clearly code examples of how to use stdin, stdout, stderr. Setting **stdin** to a file variable is the first **Act** of this project.

Act I

We know that our program executes with either no input or some input:

1. user_interface
2. user_interface doc1.txt

The inputs from the Command Line Argument will tell our program how to read:
If **1** is done, `user_interface` will run and read commands from **stdin**.
If **2** is done, `user_interface` will run and read commands from the file **doc1.txt**

File I/O

One of the important functions you will be using for this project are:

- fopen() & fclose()
- fputs()
- fprintf()
- fgets()
- sscanf()
- atoi()
- strcmp()

Make sure you fully understand what each of these functions do and what they do return.

Act II

Once our program is told how to read from either **stdin** or **an external file**. The next step is to actually read **each line** of the specified file (stdin or external file) with one of the functions mentioned above File I/O in a while loop. There're gonna be 3 cases:

Case 1: If the line is empty, skip it then go to next loop
One of the functions from File I/O returns the number of entries read, which can be so helpful to check if the line was empty or not.

Case 2: If the first non-white character contains '#', it is an comment then go to next loop
The problem here is that we cannot use **strstr()** to find the first '#' in the **line**. If there's just one character before '#' then the comment is not valid anymore. For example:

```
>           #let's print it (this is a valid comment)
>     a     #let's print it (this is not a valid comment)
```

However, there is another approach by traversing the whole line (character by character) until it encounters the first non-white character.

Case 3: Otherwise execute command
Here is where the most of your program functionality comes. I will not give more info about since the project.pdf has a really good documentation that you can follow. Do your best!

Note: Don't forget to print out **'>'** before each loop.

2.4 Project 4: Calendar

Overview

The project uses Dynamic Memory Allocation and the data structure Linked List to create a calendar application that has:

1. Days: a calendar can have number of days and specific number of events
2. Events: each day can hold several events, ordered according to the comp_func function e.g event duration

Objectives

- **Makefile**
- Practice Dynamic Allocation
- Learn **Linked List**
- Use **function pointers**

Intro to Makefile

For project 4, you're asked to develop your first own `Makefile`. As recap, `Makefile` are useful to replace all the `gcc` line:

```
> gcc hashtable.c public01.c
```

With just two words, it will allow you to create all your executables instantly:

```
> make all
```

However, there are four concepts that we should be aware of:

- Executables
- Compilation into Object files
- Dependecies
- Macros

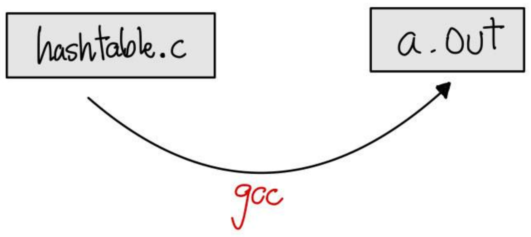
Executables: are our runnable files.
They are typically our **a.out** or whatever name you set it. They are two ways to create our executable files:

1. gcc hashtable.c public01.c
2. gcc hashtable.c public01.c -o public01

If we run **1**, `gcc` will create by default an executable file named **a.out**.
If we run **2**, `gcc` uses the flag **-o** to create an executable file with a given name, this case **public01**.

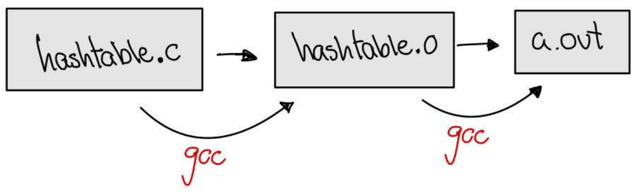
By typing on the console **a.out** or **public01**, we will run our program.

Object files: We're always been skipping one step when doing `gcc`. So far we were compiling from source code **(hashtable.c)** to an executable file **(a.out)**:



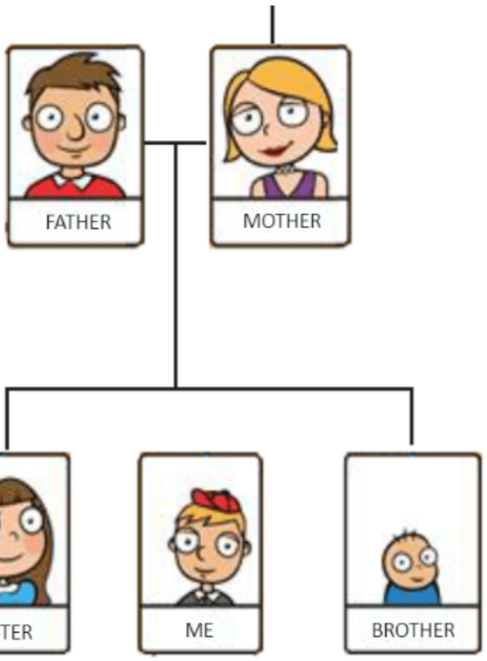
However, in reality, C compiles our source code to an object file **(hashtable.o)**, then to an executable file:

```
> gcc -c hashtable.c
> gcc hashtable.o
```
← .c to .o file
← .o to executable file **"a.out"**

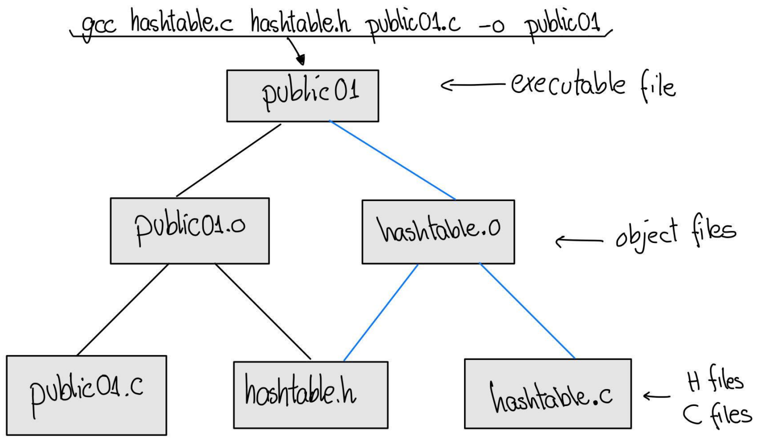


This is important because `Makefile` uses this step to create our executables.

Dependecies: are all files that depends on a file in order to compile.
An analogy: your parent's dependecies are probably you or your little brothers. You and little brothers are dependents of your parent.



In the same way, (H) header files and (C) source files are mostly dependents of (O) object files. And O files's dependecies are mostly H and C files.



Whenever you start a project's `Makefile`, always try to draw this tree dependecies. It will make your life easier. Remember, C and H files will be always at the bottom of the tree. From there, you start building the O files. Finally, the top of the tree is your executable file. One way to figure the file at the top of the tree is by asking: **What file has the main()?** In our example is `public01` which is our executable. Only one can be at the top.

Macros: are, in terms of code, like a variable that holds data.
We can use macros to represent pieces of our commands. One of the most used macros are:

- **CC** : will store **gcc** command
- **CFLAGS** : will store some options for this course a.k.a gcc alias
- **PROGS** : will store the name of all your executables

Targets: are similar to Macros but instead they hold **compilation commands** that tell Makefile which files will be compiled either to O files or executable files.
There are 3 types of targets:

- **Object targets:** here is where you compile your C file to O file and save it into this target. First, pick one C file of your tree dependecies. So now, you will compile it to O file by doing its target. The first line of the target **comes all the depedencies of that C file**, it can be either C files or H files. The second line comes the actual command to compile from C file to O file, e.g `"gcc -c hashtable.c"`
- **Executable targets:** similar to object targets. Here is where you compile O file to an executable file. The first line comes **all the O files from the tree dependecies**. The second line comes the actual command to compile from O file to an executable file, e.g `"gcc hashtable.o -o hashtable"`
- **Other targets:** e.g "**clean**" and "**all**". Do you remember that we can create our executable by typing: `make all`. So, these targets is what Makefile uses to create all your files. The same way: `make clean`, it will erase all files created by Makefile.
  There is another one called **.PHONY**. It basically tells the Makefile that "**clean**" and "**all**" targets are not files. The reason for this is because Makefile treats all targets as files, however, "**clean**" and "**all**" targets are not files, they are just commands.

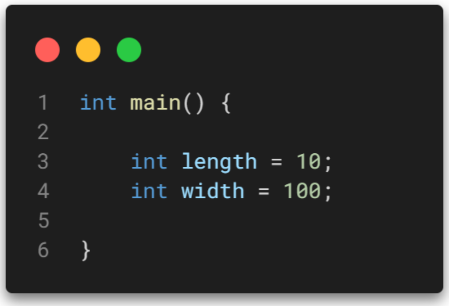
Don't forget to include...

joke 😂

Dynamic Allocation

We have been always allocated our variables automatically. Before going to the differences and when to use dynamic allocation, we must know these three concepts first:

Automatic Allocation: are variables declared, as local variables or function parameters, inside a scope. For example:



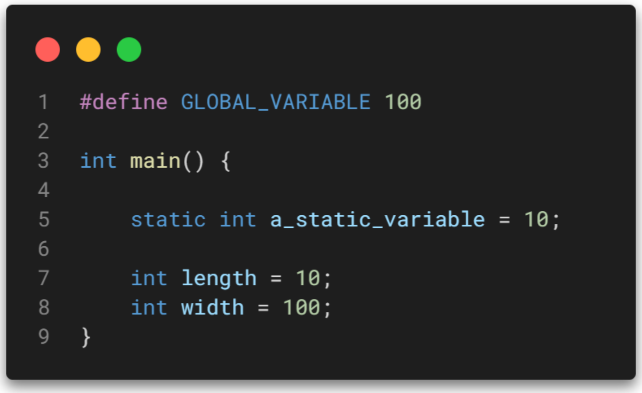
```c
int main() {

    int length = 10;
    int width = 100;

}
```

These variables are stored in the **stack** (an area in the memory) — We'll see more about stack in Project 5 — The variable is automatically allocated in the stack, and it's automatically freed when we exit the scope. Once you move out of the scope, the value of the variable is undefined, and it is an error to access it.

An example would be when you're in main and trying to print a variable declared in a function, which will give you an error.

This is a quick briefing but we'll see in depth about memory storage in Project 5: Assembly.

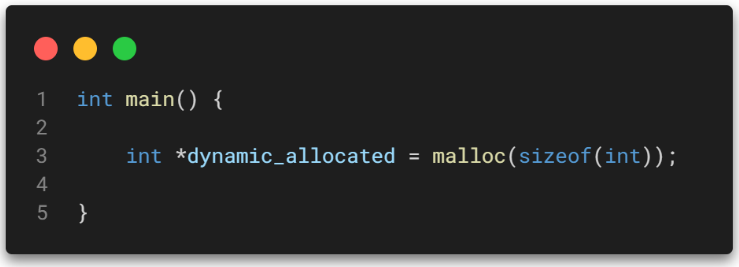
Static Allocation: are variables declared, as global or static, in any part of the program. For example:



```c
#define GLOBAL_VARIABLE 100

int main() {

    static int a_static_variable = 10;

    int length = 10;
    int width = 100;
}
```

These variables are stored in the **data segment** (an area in the memory) — again, we'll see more about this in Project 5 — All you need to know is that these variables are allocated once in the data segment and fixed (cannot be changed). These variables can be accessed from any part of the code, and its lifetime remains until the program ends.

Dynamic Allocation: are variables declare with **`malloc()`** or **`calloc()`**. For example:



```c
int main() {

    int *dynamic_allocated = malloc(sizeof(int));

}
```

These variables are stored in the **heap** (an area in the memory), and these remain in the memory until **`free()`** is called. In other words, you control the lifetime of the variable.

Automatic vs Static vs Dynamic Allocation:

One of the reasons to use dynamic instead of static/automatic allocation is:

- When you cannot determine the maximum amount of memory to use at compile time
- When you want to allocate a very large object
- When you want to build data structures (containers) without a fixed upper size
- When you want the lifetime of an object to be independent of scope, you need dynamic storage duration

These bullet points are the main reason why we're gonna use dynamic allocation in this project for some **objects** and **strings** but not for **ints**. Because these variabes are subject to change their memory size (no fixed), we will need to dynamic allocate these variables.

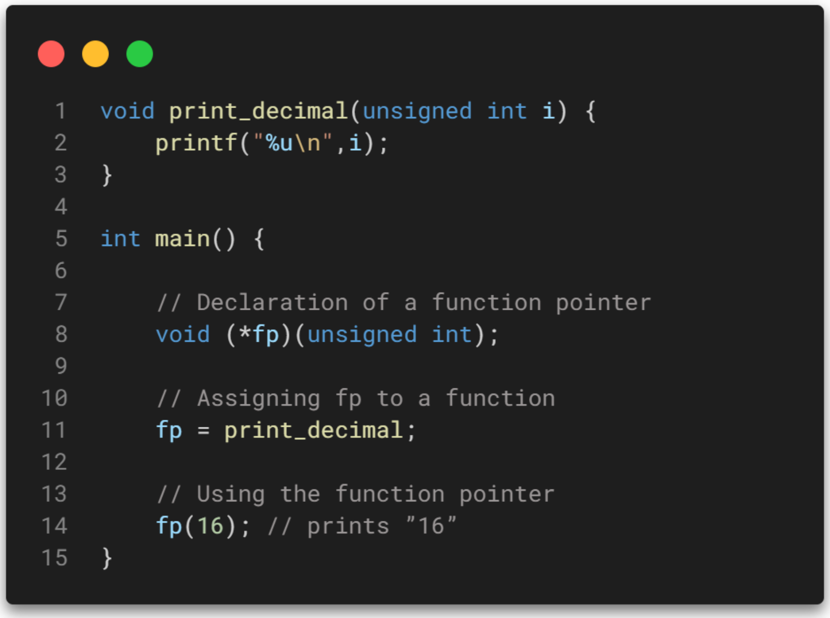
Malloc vs Calloc:

Malloc and Calloc does the same thing but have different outcome.

- Calloc: creates a block of memory for dynamic allocation, but memory allocated are uninitialized. In other words, it contains **garbage value**.
- Malloc: creates a memory space for dynamic allocation, and memory are initialized to **0**.

Function Pointers

Like **pointers to variables**, we can also have **pointers to functions**. We know that variables are allocated in the memory and has an address. Functions are also stored in the memory with an address, so we can access them with pointers. I believe this is a good example on how to declare, initialize, and call function pointers.



```c
void print_decimal(unsigned int i) {
    printf("%u\n",i);
}

int main() {

    // Declaration of a function pointer
    void (*fp)(unsigned int);

    // Assigning fp to a function
    fp = print_decimal;

    // Using the function pointer
    fp(16); // prints ”16”
}
```

Here are the steps to write the declaration of a function pointer:

1. Write the function prototype: `void fp(unsigned int)`
2. Add parentheses around the function name: `void (fp)(unsigned int)`
3. Add an asterisk next to the function name: `void (*fp)(unsigned int)`

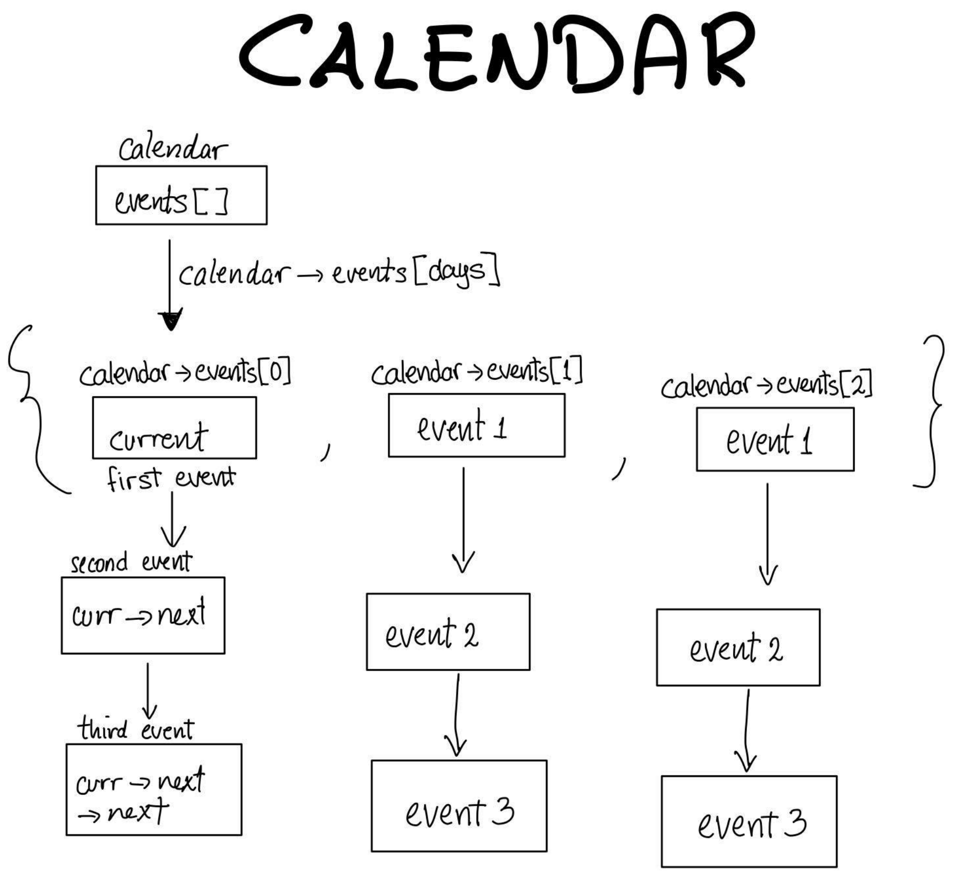
It's critical to have the asterisk inside the parentheses since this tells that the **fp** is a function pointer. If outside, this is a simple function that returns a pointer (we don't want this).

You're not gonna be using oftenly other than on this project but be aware of how to declare and initialize it since it's a potential question on exam.

A Start

It is suggested that you start this project by building first the Makefile. Before coding the implementation of this project, it's indispensable to know **structs** and **how to access their fields**. With structs, we can create and manipulate Linked Lists, which is the whole point of this project.

Let's visualize what our Project is about.



A **calendar** has the following:

- Name — char
- Array of events — array of linked lists
- Number of Days and — int
- Number of total events — int
- Comparission function — function pointer
- Function that frees variables — function pointer

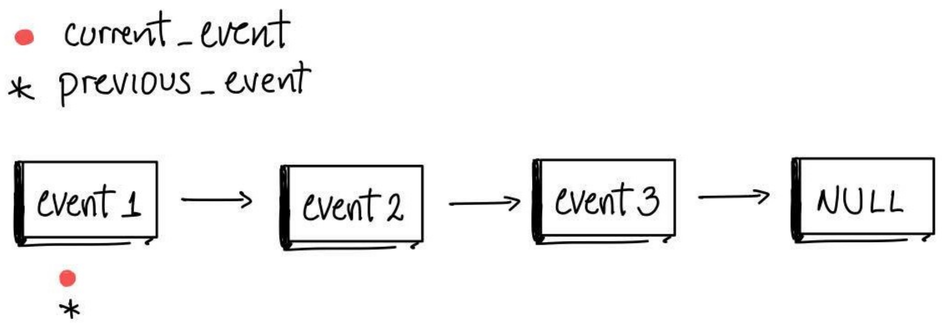
An **event** has the following:

- Name — string
- Start time in militar format — int
- Duration in minutes — int
- Info — function pointer
- Next event — pointer to an object Event

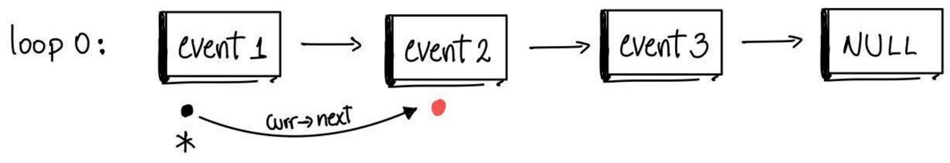
Explanation: We observe 2 things from the main skeleton of our Document.

1. The **Document** has an array of Events in which each index represents a day.
2. Each **Event** has a pointer that points to the next Event.

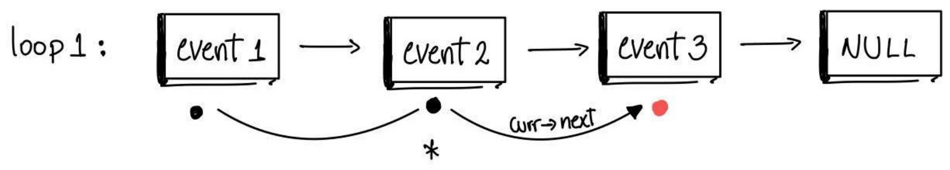
First of all, before going to the next section, we must know how to traverse in a Linked List.



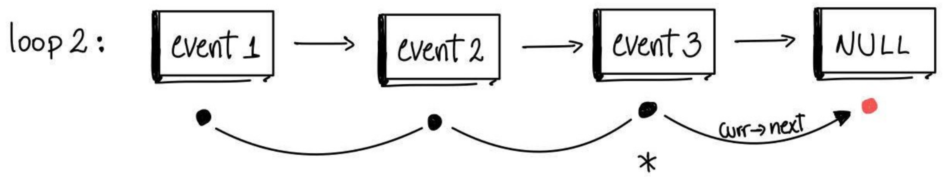
In order to traverse a Linked List, we would need to create two pointers: **current** and **previous** nodes. One advatange of having two nodes is that we can have track of two nodes in the Linked List at the same time and access either one when it's convenient. Before traversing, the two nodes must be set equal to **"event 1"**.



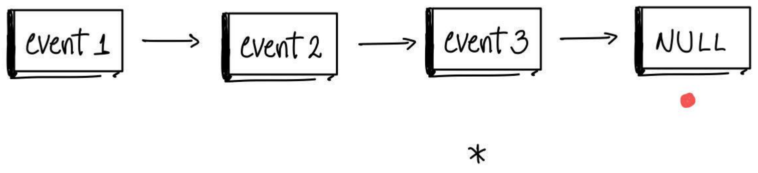
Next step is to loop. When **current** (event 1) is not a NULL node, it means that we haven't reached to the end of the Linked List and will procced to the next loop. So first, we will save the **current** node into **previous** node so we can keep tracking **event 1**. Finally, we set **current** node to point to the next node. Now, our **current** node will point to **event 2** and **previous** points event 1. This is the end of the first loop and proceed the next loop.



Same process. Check for **current** (event 2) is not a NULL. Save **current** (event 2) into **previous** node. Set **current** to point the next node (event 3). So far, **current** points event 3, and **previous** points event 2. Proceed the next loop.



Check for **current** (event 3) is not a NULL. Save **current** (event 3) into **previous** node. Set **current** to point the next node (NULL). So far, **current** points NULL, and **previous** points event 3. Proceed the next loop.



Check for **current** (NULL) is not a NULL. Upps! it is NULL. Therefore, terminate the loop. So far, **current** points NULL, and **previous** points event 3. Here is where **previous** node is so useful because now we have a reference of the **last node of the Linked List** or **whatever node was the last when loop terminated**.

Add Event

Adding an new event into the Linked List is tricky. The documentation asks us to add it in increasing order according to the **comparission function**:

```
int comparission_function(event 1, event 2);
```

As we can see, the comparission function can take any **Object** and returns an **int**. The comparission function will access an Object' field to compare with the second Object's field to later return a number. Be aware that we don't know what it is comparing, but we do know that, whatever it is comparing, it will return an number **(int)**. Let's see what I mean:

Assume the function access the **duration_minutes** field from both **Events** and returns the difference of these two.

```
> event 1's duration = 20 min

> event 2's duration = 15 min

> The function will return:  20 − 15 = 5
```

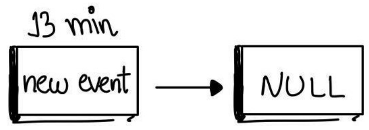
We clearly see three main points:

- If the function returns greater than 0, it means that event 1's duration is greater than event 2's duration.
- If the function returns less than 0, event 1's duration is less than event 2's duration.
- If the function returns 0, both event's duration are equal.

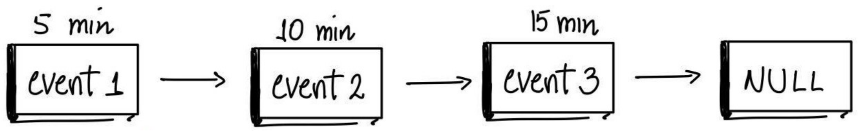
Having that in mind, now we are ready to add our new event in our Linked List. There're gonna be three cases:

Case 1: Adding between two nodes in the Linked List

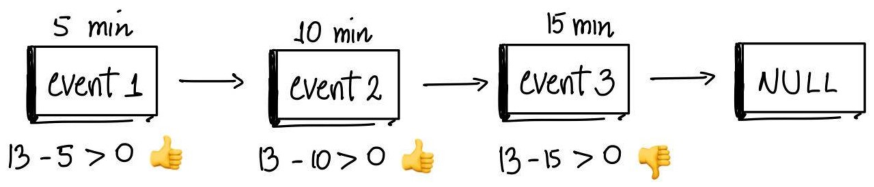
First, we create our new event with their fields populated. I assumed you dynamic allocated the fields asked in the documentation.



Assume that this new event has a duration of 13 minutes and it is pointing to a NULL node.



Assume our Linked List has 3 **events** with their respective duration time in ascending order. As we traverse thru the Linked List, we're gonna compare the duration time so we know where to place it.

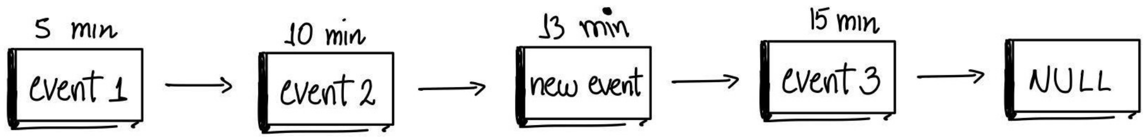


Now, we traverse the Linked List as previously mentioned. We check for node to be **not NULL** and comp_func be **greater than 0**. The process is the following:

Is **event 1** not NULL? Yes	`&&`	Is 13 - 5 greater than 0? Yes, so go to next loop
Is **event 2** not NULL? Yes	`&&`	Is 13 - 10 greater than 0? Yes, so go to next loop
Is **event 3** not NULL? Yes	`&&`	Is 13 - 15 greater than 0? Nope, so stop loop.

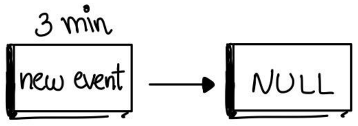
Good, since we stopped the loop, now we have tracked the **last node** (event 3 — 15 min) when loop terminated and the most important the **previous** (event 2 —10 min) node.

Last step is to add **new event** between **event 3** and **event 2**. The following output should look like:

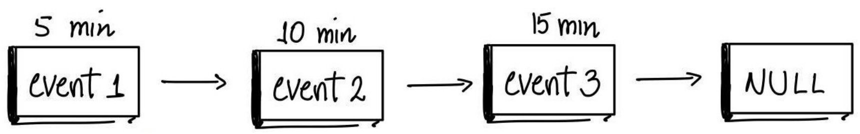


Case 2: Adding at the beginning of the Linked List

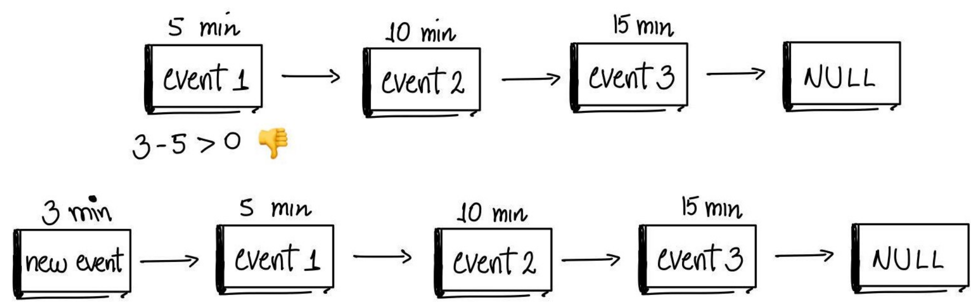
This case is different from **Case 1** since we cannot add between **event 1** and another node (because it doesn't exist). So far we know that when the traverse loop ends, we will populate our **current** and **previous** node. If these two are equal, it means that we need to place the **`new event`** at the beginning of the Linked List. Suppose we have another new Event initialized.



Assume our new event has a duration of 3 minutes and pointing to NULL.



And the same Linked List mentioed in the previous Case 1.



We do the same process from **Case 1**:

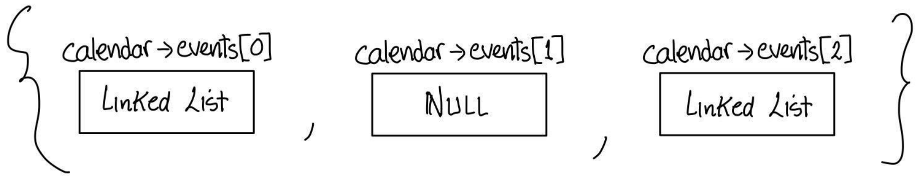
Is **event 1** not NULL? Yes **&&** Is 3 - 5 greater than 0? No, so stop loop.

After the traversing, we'll get our **current** and **previous** nodes. If **current** and **previous** are equal, it means that both are pointing to the beginning of the Linked List. Finally, we grab the **new_event** node and **set its next pointing to the Linked List**. The new Linked List **(new_event + initial Linked List)** have to be saved into the Events array in whatever day it was specified.

Case 3: Adding an event into a empty Linked List in a specific day.

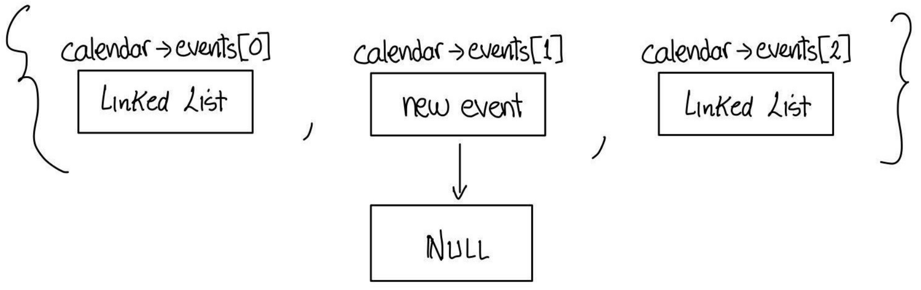
This case is straightforward. We know that our Calendar has an **array of Events** where each index represents a day that has an **Linked List**. If there's **no event** (empty Linked List) at day specified, then we set this **new event** as the first event in that day.

Let's say we were asked to add a new event at day 2 (second index of the array).



We see that our array of Linked List has **3 days**. The first day **(index 0)** is populated with a Linked List. The third day **(index 2)** is also populated. However, the second day is NULL or empty Linked List.

So we just set `calendar->events[1]` to be equal to the **new event**:



Note: Don't forget to add +1 to the total events of the Calendar **for every Case**.

Remove Event

In order to remove an event by name, you must loop thru the whole calendar to find the event and free it. Note that **Remove** event means to **free** it. There will be different **Cases**.

Case 1: Once we found our event to remove, if the event has info pointer and free_info function, then we use free_info function to free this event.

Case 2: Once we found our event to remove, if the event is the only event of the day, then free it. For example, we want to remove event 1, which is the only event in this Linked List.

```
Events[1] = event 1 -> NULL
```

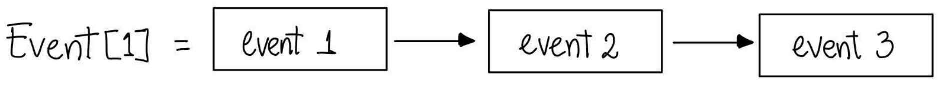
Case 3: Once we found our event to remove, if the event is the first event of the day, then free it. For example, we want to remove event 1, but there are more events after it.

```
Events[1] = event 1 -> event 2 -> event 3 -> NULL
```

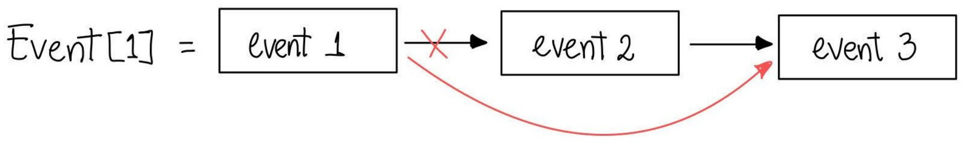
Case 4: Once we found our event to remove, if the event is between other events, free it. For example, we want to remove event 2, but it's between two events (event 1 and event 3).

```
Events[1] = event 1 -> event 2 -> event 3 -> NULL
```

Important: When you remove an event, your Linked List needs to be still "linked". This will solve several release and secret tests. Let's see what I mean.



Let's say we have an array of Events. In the array Events[1] (at day 2), we have a Linked List of 3 events, and we want to remove **event 2**. The following process is:



Let's say your **current** node is pointing to **event 2**, and **previous** to **event 1**. Before you free event 2, you must link **previous** (event 1) to **current's next** (event 3).

This is because if you free event 2 first, then event 1's next will point to nothing, and since event 2 is the only one pointing to event 3 and is freed, then event 3 wil never be accessed because no node points to it.

Note: Don't forget to free event 2's name and event 2, decrement total events, and return SUCCESS. These same steps for every **Case**.

Destroy Calendar

TODO

2.5 Project 5: Assembly Language Programming

Overview

In this project, you will write MIPS assembly code that corresponds to a provided C code.

Objectives

- Write assembly code (potential coding question)
- Understand translation of high-level to low-level language
- Learn Stack Memory

2.6 Project 6: Senior Shell

Overview

The project will clone a shell that will support boolean operations, pipes, and file redirections.

Objectives

- Learn Binary Tree and Traversing
- Manipulate file descriptors — File Redirection
- Execute program: `execvp()`
- Unix I/O

3 Assignments

3.1 Assignment 1

3.2 Assignment 2

3.3 Assignment 3

3.4 Assignment 4

3.5 Assignment 5

3.6 Assignment 6