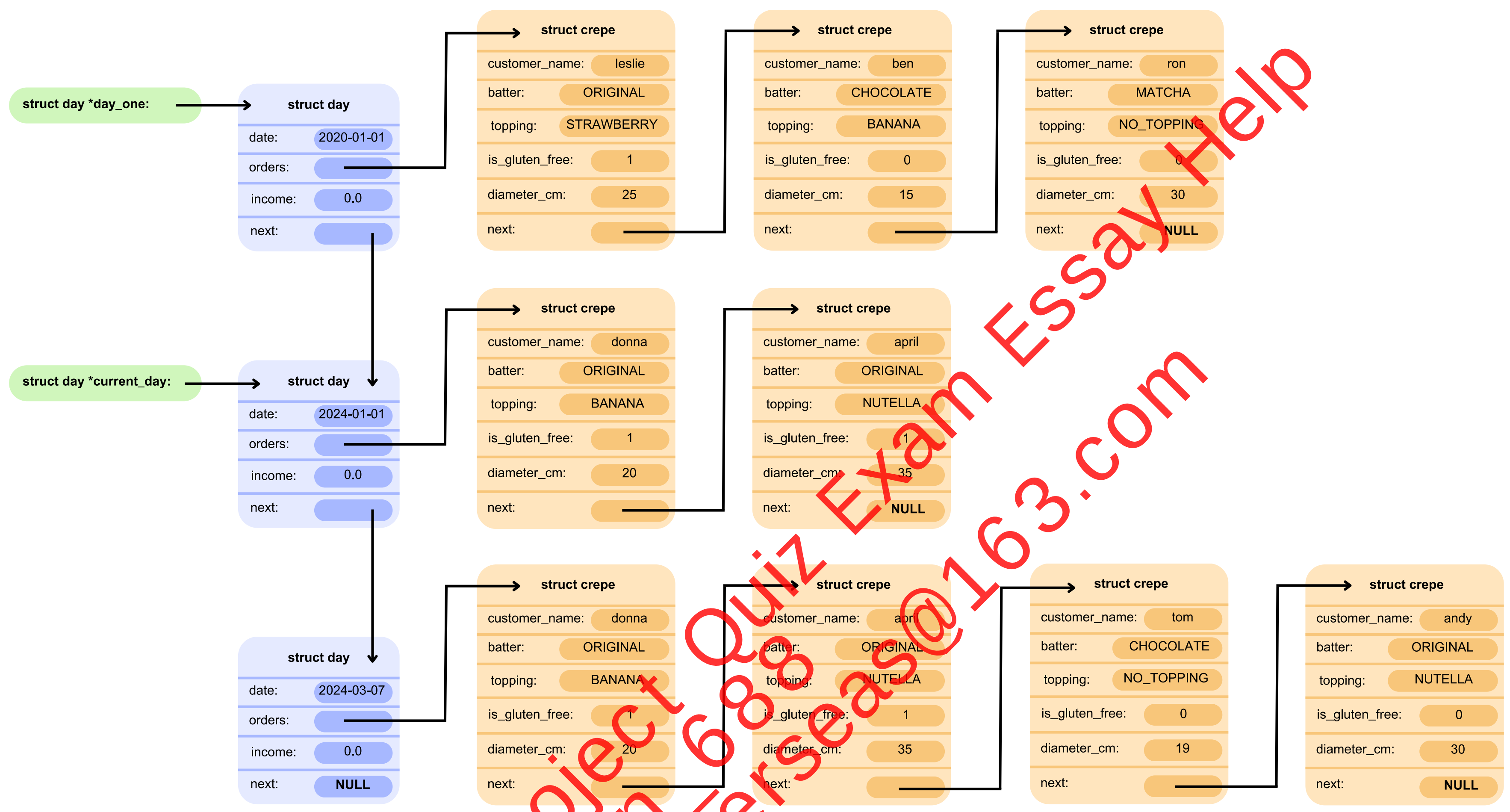
Assignment 2 - Crepe Stand!

You have been tasked with creating a crepe stand order tracker, logging daily sales of various crepes. This crepe stand has different flavour batters and toppings to choose from, and you can customise the crepe to your liking! Or, you can go with one of the crepe stand"s signature crepes - Strawberry Special, Chocolate Connoisseur, or Matcha Madness! You"ll need to record all the sales made, calculate costs and bills, display statistics about the crepe stand, and even find out when the most profitable periods of sale are. When you"re done, your crepe stand order tracker might look something like this:



Final Crepe Stand *(click on image to expand)*

Assignment Structure

This assignment will test your ability to create, use, manipulate and solve problems using linked lists. To do this, you will be implementing a crepe stand order manager using a linked list of business days, each containing a list of crepe orders.

We have defined some structs in the provided code to get you started. You may add fields to any of the structs if you wish.

`struct day`

Purpose: To store all the information about a single business day"s orders of crepes. This will form a linked list of business days by pointing to the next day (or `NULL`).

`struct crepe`

Purpose: To store all the information about a single crepe. This will form a linked list of crepes by pointing to the next one (or `NULL`).

`struct date`

Purpose: Stores a date as 3 different numbers, the year, month and date.

The following enum definitions are also provided for you. You can create your own enums if you would like, but you should not modify the provided enums.

`enum batter_type { ORIGINAL, CHOCOLATE, MATCHA };`

Purpose: To represent the different types of batter a crepe can have.

```
enum topping_type { NO_TOPPING, BANANA, STRAWBERRY, NUTELLA };
```

Purpose: To represent the different types of topping a crepe can have.

HINT:

Remember to initialise every field inside the structs when creating them (not just the fields you are using at that moment).

How to Get Started

There are a few steps to getting started with the Crepe Stand.

1. Create a new folder for your assignment work and move into it. You can follow the commands below to link and copy the files.

```
$ mkdir ass2
$ cd ass2
```

2. Run the following command below to download all 3 files, which will link the header file and the main file. This means that if we make any changes to `crepe_stand.h` or `main.c`, you will not have to download the latest version as yours will already be linked.

   Download the starter code (crepe_stand.c) here

   Download the starter code (crepe_stand.h) here

   Download the starter code (main.c) here

   or, copy this code to your CSE account using the following command

```
$ 1091 fetch_activity crepe_stand
```

3. Run `1091 autotest crepe_stand` to make sure you have correctly downloaded the file.

```
$ 1091 autotest crepe_stand
```

WARNING:

When running the autotest on the starter code (with no modifications), it is expected to see failed tests.

1. Read through **Stage 1**.

Starter Code

This assignment utilises a multi-file system. Make sure to reference the relevant material covered in the multiple_file_C on multi-file systems.

There are three files we have in Crepe Stand:

- **Main File (`main.c`):** contains code written for you to test your code in `crepe_stand.c`. You don"t need to read this, but if you"re curious about how this assignment works, feel free. It handles all the input and output for you and contains the `main` function. **You cannot change `main.c`.**
- **Header File (`crepe_stand.h`):** contains defined constants that you can use and function prototypes. It also contains header comments that explain what functions should do and their inputs and outputs. *If you are confused about what a function should do, read the header file and the corresponding specification.* **You cannot change `crepe_stand.h`.**
- **Implementation File (`crepe_stand.c`):** contains stubs of functions for you to implement. This file does not contain a `main` function, so you will need to compile it alongside `main.c`. *This is the only file you may change. You don"t need to use `scanf` or `fgets` anywhere.* If you wish to create your own **helper functions**, you can put the function prototypes at the top of this file and implement it later in the file. Note, you should place your comment just above the function definition

NOTE:

Function Stub: A temporary substitute for yet-to-be implemented code.

How to Compile Crepe Stand

To compile you should compile `crepe_stand.c` alongside `main.c`. This will allow you to run the program yourself and test the functions you have written in `crepe_stand.c`. Autotests have been written to compile your `crepe_stand.c` with the provided `main.c`.

To compile your code, use the following command:

```
$ dcc crepe_stand.c main.c -o crepe_stand
```

Once your code is compiled, you can run it with the following command:

```
$ ./crepe_stand
```

To autotest your code, use the following command:

```
$ 1091 autotest crepe_stand
```

Reference Implementation

To help you understand the expected behaviour of Crepe Stand, we have provided a reference implementation. If you have any questions about the behaviour of your assignment, you can check and compare it to the reference implementation.

To run the reference implementation, use the following command:

```
$ 1091 crepe_stand
```

You might want to start by running the `?` command:

```
 $  1091 crepe_stand
Welcome to the 1511 Crepe Stand!
Please enter today's date (YYYY-MM-DD): 2024-03-03
Enter Command: ?
=====================[  1511 Crepe Stand  ]=====================
      ===============[     Usage Info     ]===============
  ?
    Show help
  a [crepe type] [customer name] {OPTIONAL: [batter] [topping]
    [gluten free] [size]}
    Append a crepe to the end of the orders list
  p
    Print out all the crepes in the order list
  c
    Print out the amount crepes in the order list; total,
    chocolate and matcha counts.
  i [position] [crepe type] [customer name] {OPTIONAL: [batter]
    [topping] [gluten free] [size]}
    Insert a crepe to the [position] position in the list
  s
    Prints out the statistics of the day's orders
  C [position]
    Calculates the cost of the crepe in the given position
  t
    Prints out the current total income for the day
  b [customer name]
    Prints out the given customer's bill for the day
  n [date in YYYY-MM-DD format]
    Inserts a new day to the list of days
  d
    Prints out all days and associated information
  >
    Cycles to the next day in the list of days
  <
    Cycles to the previous day in the list of days
  r [position]
    Removes a crepe from the current day in the specified
     position
  R [date in YYYY-MM-DD format]
    Removes specified day in the list of days and all its
      ordered crepes.
  w
     Prints the total profit over all weekdays and all weekends
  m [year to check]
    Prints the maximum profit period in the given year
================================================================
Enter Command: Ctrl-D
Thanks for visiting the Crepe Stand!
```

About the Starter Code

The starter code `crepe_stand.c` contains some provided functions to help simplify some stages of this assignment. These functions have been fully implemented for you and should not need to be modified to complete this assignment.

These provided functions will be explained in the relevant stages of this assignment. **Please read the comments and the spec as we will suggest certain provided functions for you to use.**

It also contains function stubs for all commands used in this assignment. Each stage will specify which to implement. You will most likely still need to create your own helper functions.

`crepe_stand.h` also contains some struct definitions that you will need to use for some stages of the assignment. These will be explained as they are required.

WARNING:

Do not change the return type or parameter amount and type of the provided function stubs. `main.c` depends on these types and your code may not pass the autotests otherwise, as when testing we will compile your submitted `crepe_stand.c` with the supplied `main.c` .

> NOTE:
>
> If you wish to create your own hlper functions, you can put the function protoyepes at the top of `crepe_stand.c` and implement it later in the file. Note, you should place your comment just above the function definition.

Allowed C Features

In this assignment, **you cannot use arrays for the list of days nor the lists of crepes** and cannot use the features explicitly banned in the Style Guide.

We **strongly** encourage you to complete the assessment using only features taught in lectures up to and including weeks 8 and 9. The only C features needed to get full marks in this assignment are:

- `int` , `char` , and `double` variables.
- Enums.
- Structs.
- If statements.
- While/For loops.
- Your own helper functions.
- Pointers.
- Strings (i.e. null terminated `char` arrays).
- Linked lists.
- Standard libraries: `stdio.h` , `stdlib.h` , `ctype.h` and `string.h` .
- Good code style. (Refer to the Style marking rubric)

Using any other features will not increase your marks alone and improper use could risk style mistakes that cost you marks.

Regardless, you **must** still follow the style guide. You may also be unable to get help from course staff if you use features not taught in DPST1091.

Features that the style guide strongly discourages or bans will be penalised during marking.

Your Tasks

This assignment consists of four stages. Each stage builds on the work of the previous stage, and each stage has a higher complexity than its predecessor. You should complete the stages in order.

- **Stage 1.1 - Basic setup:**

  Implement code to create a single struct crepe and a single struct day.

- **Stage 1.2 onwards - Implement the commands:**

  You will need to implement some commands, which will allow you to add, modify and display information on the list of crepes and different days.

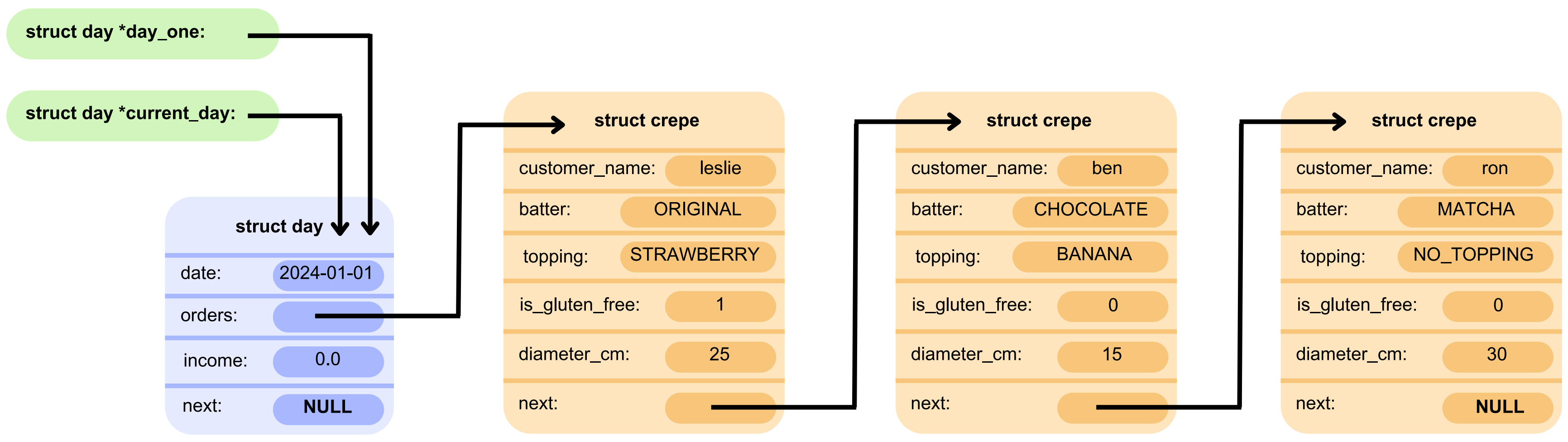
> NOTE:
>
> You can assume that your program will **NEVER** be given: - A non-existent command. - Command arguments that are not of the right type. - An incorrect number of arguments for the specific command. - Commands will always start with a `char` . Additionally, all scanning and printing will be handled for you in `main.c` , so you should not need to worry about the above.

Stage 1

For Stage 1 of this assignment, you will be implementing some basic commands to set up the crepe stand! Specifically, this will include:

- Implementing the `create_crepe` & `create_day` functions.
- Appending crepes to a list of orders in a `struct day` .
- Printing out all crepes ordered on a given day.
- Handling any invalid inputs.
- Counting the number of crepes in any given day.

By the end of this stage, your linked list of crepes will look something like:



Stage 1: A linked list of crepe orders in a day *(click on image to expand)*

Stage 1.1 Creating a day and a crepe

As you might have found by now, it can be really useful to have a function that takes the input for a linked list node, calls `malloc` and initialises all the fields of the node. So, in **Stage 1.1**, we will be implementing functions that do exactly that for a `struct day` and for a `struct crepe` .

You"ll find the following unimplemented function stubs in `crepe_stand.c` :

```
struct day *create_day(struct date new_date) {
    // TODO: implement this function
    printf("Create Day not yet implemented.\n");
    exit(1);
}

struct crepe *create_crepe(char *customer_name,
                           enum batter_type batter,
                           enum topping_type topping,
                           int is_gluten_free,
                           int diameter_cm) {
    // TODO: implement this function
    printf("Create Crepe not yet implemented.\n");
    exit(1);
}
```

`create_day` is called in `main.c` after scanning in today"s date from the user. `create_crepe` will be called by you, in `crepe_stand.c` , whenever a crepe should be added to the list of orders.

Your task is to complete the `create_day` function, so that it:

1. Creates a new `struct day` (using `malloc`).
2. Copies the `new_date` into the corresponding struct fields.
3. Initialises all other fields to some reasonable value.
4. Returns a pointer to the newly created `struct day` .

Your also then need to complete the `create_crepe` function, so that it:

1. Creates a new `struct crepe` (using `malloc`).
2. Copies the `customer_name` , `batter` , `topping` , `is_gluten_free` and `diameter_cm` into the corresponding struct fields.
3. Initialises all other fields to some reasonable value.
4. Returns a pointer to the newly created `struct crepe` .

Error handling

- No error handling is required for **Stage 1.1**.

Assumptions

- The date given will always be a valid date in the format `YYYY-MM-DD` .
- The initial income is always `0.0` .
- Initally there are no orders.

Testing

There are no autotests for **Stage 1.1**.

Instead, you may want to double check your work by compiling your code using `dcc` and making sure there are no warnings or errors.

Stage 1.2 Appending Crepes

Now it"s time to start adding crepes to your orders! When you run your program, `create_day` will be called for you, and if implemented appropriately, the first (and current) day"s list of orders will start empty.

Customers can order their custom crepe with their chosen design, or, they can order a signature crepe to save some time! There are three to choose from - **Matcha Madness**, **Strawberry Special** and **Chocolate Connoisseur**.

Their details are as follows:

Crepe Menu

Below is the detailed menu for our signature crepes.

Signature Crepe	**Matcha Madness**	**Strawberry Special**	**Chocolate Connoisseur**
Batter Type	Matcha	Original	Chocolate
Topping	Strawberry	Strawberry	Nutella
Gluten-Free	No	Yes	No
Size	Medium (25cm)	Medium (25cm)	Medium (25cm)

The **append crepes** command is inputted as follows:

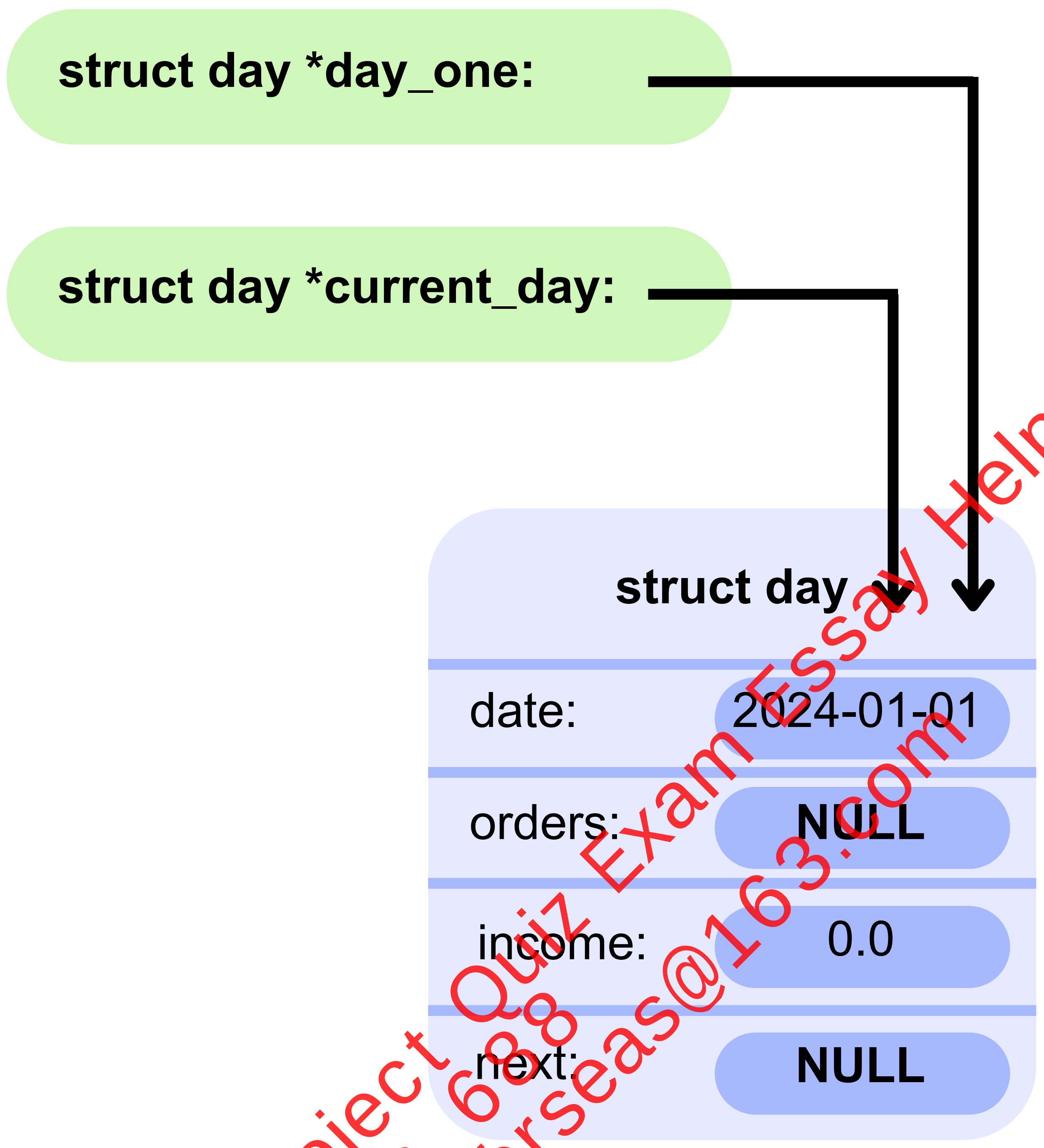
Command: Append crepe to current day"s orders

```
Enter Command: a [crepe type] [customer name] {OPTIONAL: [batter] [topping] [gluten free] [size]}
```

The `a` command takes in either 2 or 6 arguments depending on whether it is a custom crepe or a signature crepe. `[crepe type]` will be: - `custom` , for a custom crepe, - `matcha` , for a Matcha Madness crepe, - `strawberry` , for a Strawberry Special crepe, or - `chocolate` , for a Chocolate Connoisseur crepe.

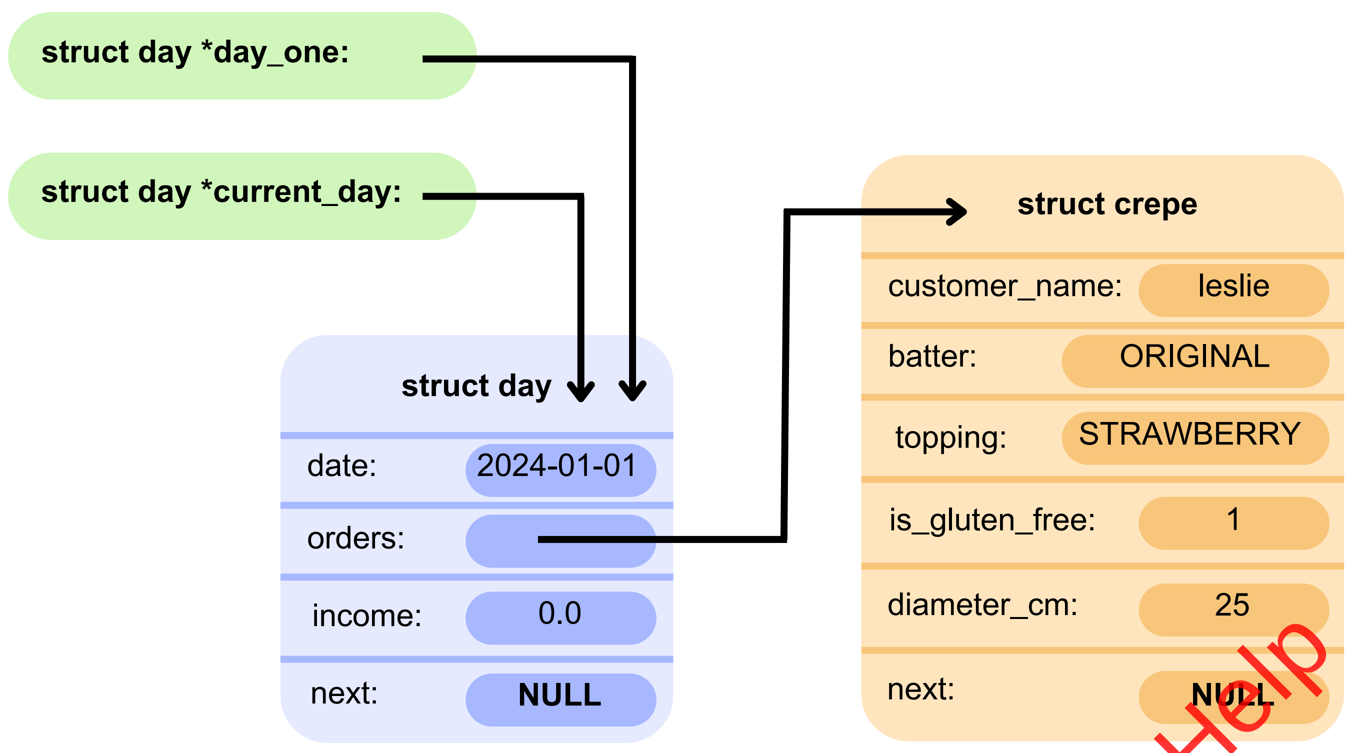
> **NOTE:**
>
> This input is all already sacnned in for you in the command loop function (`command_loop`) located in `main.c` .

Before appending any crepes, the current day"s list of orders will be empty, looking something like this:



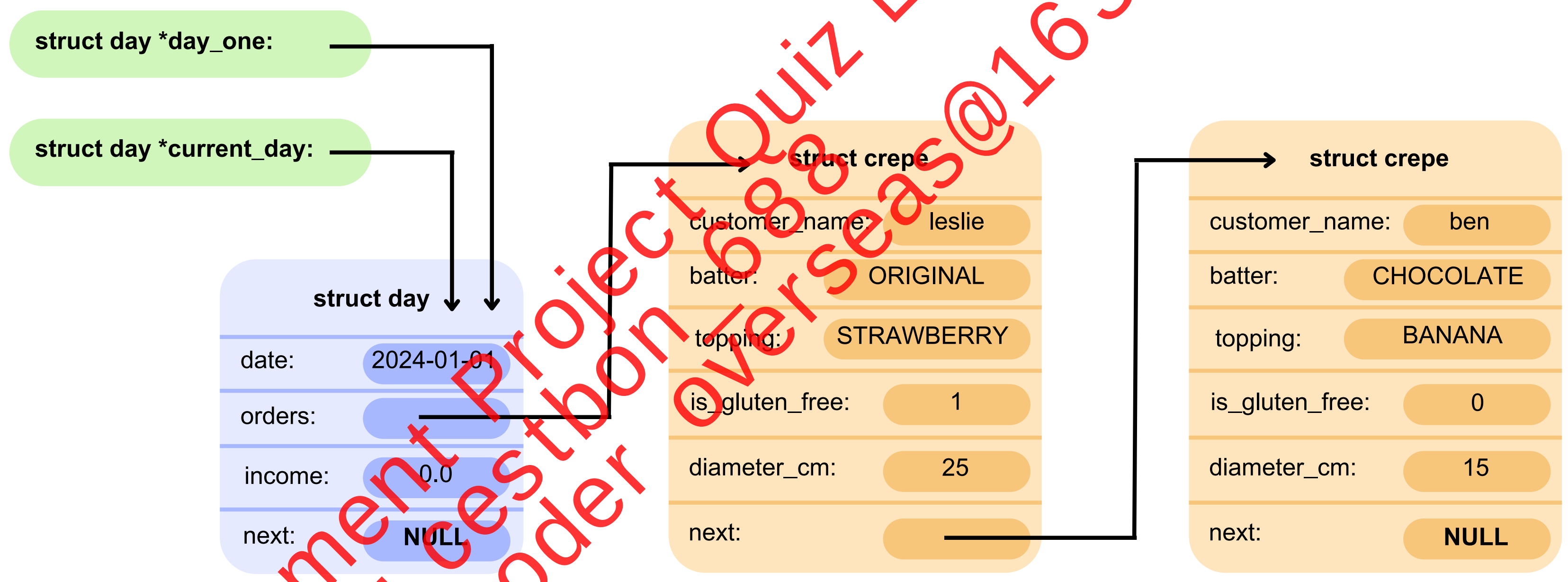
Stage 1.2: A single struct day with no orders *(click on image to expand)*

After appending one crepe, the current day"s list of orders will have that one crepe, looking something like this:



Stage 1.2: A single struct day with one order *(click on image to expand)*

Any other crepes added will all be appended to the the end of the list, i.e. a tail insertion, looking something like this:



Stage 1.2: A single struct day with two orders *(click on image to expand)*

Your task is to implement the functions `append_crepe_custom` which will append a newly created custom crepe to the end of the orders list and `append_crepe_signature` which will will append a newly created signature crepe to the end of the orders list. Note, you do not need to scan in the command and input as this is already done for you in `main.c` You will find the following function stubs in `crepe_stand.c` :

```
int append_crepe_custom(struct day *current_day,
                        char *customer_name,
                        int batter,
                        int topping,
                        int is_gluten_free,
                        int diameter_cm) {
    // TODO: implement this function
    printf("Append Crepe Custom not yet implemented.\n");
    exit(1);
}

int append_crepe_signature(struct day *current_day,
                           char *customer_name,
                           char *crepe_type) {
    // TODO: implement this function
    printf("Append Crepe Signature not yet implemented.\n");
    exit(1);
}
```

HINT:

Make sure to read the function prototypes for `append_crepe_custom` and `append_crepe_signature` in `crepe_stand.h` for clear explanations regarding function parameters and their corresponding functionalities.

NOTE:

`current_day` contains a pointer to the head of the list of its orders, called `orders` . This is what you will append the new custom crepe to.

For both of these functions, your program should create a new `struct crepe` using your `create_crepe` function and then append it to the end of the current day"s order list (i.e. perform a tail insertion). Both functions return an `int` - this value indicates whether or not the appending was successful or not. We will handle error checking in **Stage 1.4**, so for now you can return the constant `VALID_CREPE` (defined in `crepe_stand.h`). `main.c` will handle all input and output (including any `printf` , `scanf` , or `fgets` calls) for you already.

HINT:

The function `strcmp` will be useful to tell what kind of signature crepe `crepe_type` is. You can see how to use `strcmp` from the `string.h` library

here

Error handling

- No error handling is required for **Stage 1.2**

Assumptions

- `crepe_type` in `append_crepe_signature` will always be one of `matcha` , `strawberry` or `chocolate` .
- The same customer can make multiple orders, but no two different customers will have the same name.

Examples

+ Example 1.2.1: Add one signature crepe

+ Example 1.2.2: Add one custom crepe

+ Example 1.2.3: Add many crepes; signature and custom

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 01_02 crepe_stand
```

Stage 1.3 - Printing out the current day"s orders

Now we want to be able to display the current day"s orders and all the information about each order!

Command: Print crepes

```
Enter Command: p
```

This command takes no arguments.

When the `p` command is run, your program should print out all of the crepes in the current day"s list of orders.

There is one function to implement for **Stage 1.3**, called `print_crepes`. The function stub in `crepe_stand.c` reads as follows:

```
void print_crepes(struct day *current_day) {
    // TODO: implement this function
    printf("Print Crepes not yet implemented.\n");
    exit(1);
}
```

Two functions have been provided for you in `crepe_stand.c` to help print the list of crepes: `print_single_crepe` (prints a single crepe) and `print_arrow` (prints an arrow to connect the crepes).

HINT:

You may find it helpful to edit your `struct crepe`, adding a field that stores the crepe"s position in the list of orders.

If there are no crepes in the list, the following message should be printed:

```
No crepes have been ordered yet!
```

Otherwise, crepes should be printed as follows:

```
Enter Command: p
---------------------
1. leslie's crepe
Matcha flavour
Toppings: strawberries
gluten free? no
Size: 25cm
---------------------
    |
    v
---------------------
2. ann's crepe
Original flavour
Toppings: strawberries
gluten free? yes
Size: 25cm
---------------------
    |
    v
---------------------
3. ron's crepe
Matcha flavour
Toppings: bananas
gluten free? yes
Size: 20cm
---------------------
```

NOTE:

The last crepe order should have no arrow after it.

Examples

+ Example 1.3.1: Print a day with no orders

+ Example 1.3.2: Print a day with a few orders (signature and custom)

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 01_03 crepe_stand
```

Stage 1.4 Handling Errors

Now we will need to make sure that if a customer tries to place an order that we cannot fulfill, that we do not take that order! So from **Stage 1.4** onwards, you will need to handle error checking for given inputs when creating a crepe and will need to edit the code you've written from earlier stages.

You will need to check for the following errors:

- Invalid batter
- Invalid topping
- Invalid gluten-free option
- Invalid size

In your `append_crepe_custom` function, you will need check if any of these are invalid and if they are, return the corresponding constant.

These constants are defined for you in `crepe_stand.h` :

```
#define VALID_CREPE 0
#define INVALID_BATTER -2
#define INVALID_TOPPING -3
#define INVALID_GFO -4
#define INVALID_SIZE -5
```

You are currently returning `VALID_CREPE` always. You will now need to check the inputs to see if you should return an invalid constant instead and `main.c` will then handle the printing accordingly.

To check whether each field is valid or not, you must perform the following checks **in the below order**:

- `batter` is invalid if it is not one of `ORIGINAL` , `CHOCOLATE` or `MATCHA` .
- `topping` is invalid if it is not `NO_TOPPING` , `BANANA` , `STRAWBERRY` or `NUTELLA` .
- `is_gluten_free` is invalid if it is not 0 or 1.
- `size` is invalid if it is not between 10cm (inclusive) and 40cm (not inclusive).

If a crepe is invalid, it should not be added to the list of crepes.

Assumptions

- You can assume the length of `customer_name` will be less than `MAX_STR_LEN` (not including the null terminating character) and you do not need to check if it is valid or not.
- You don't need to do error checking to see if a customer has the same name as another customer, you can assume it is the same customer.

Examples

+ Example 1.4.1: Attempting to append invalid crepes

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 01_04 crepe_stand
```

Stage 1.5 - Counting Number of Crepes Ordered on the Current Day

We need a way to quickly tell how many crepes have been ordered so far today! So in **Stage 1.5** you will be implementing a feature that calculates how many crepes have been ordered on the current day.

Command: Count number of crepes in current day

```
Enter Command: c
```

This command takes no arguments.

There is one function to implement for **Stage 1.5**, called `count_crepes` . The function stub in `crepe_stand.c` reads as follows:

```c
int count_crepes(struct day *current_day) {
    // TODO: implement this function
    printf("Count Crepes not yet implemented.\n");
    exit(1);
}
```

- `struct day *current_day` is a pointer to the current day of crepe stand operations.
- `count_crepes` returns the number of crepes in the `current_day` "s list of orders (passed in via the `struct day` pointer, `current_day`).

Printing will be handled based on the return value in `main.c` .

Examples

＋ Example 1.5.1: Counting an empty list of orders

＋ Example 1.5.2: Counting a list of orders with one crepe only

＋ Example 1.5.3: Counting multiple orders

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 01_05 crepe_stand
```

Testing and Submission

Remember to do your own testing

Are you finished with this stage? If so, you should make sure to do the following:

- Run `1091 style` and clean up any issues a human may have reading your code. Don"t forget -- **20%** of your mark in the assignment is based on style and readability!
- Autotest for this stage of the assignment by running the `autotest-stage` command as shown below.
- Remember -- *give early and give often*. Only your last submission counts, but why not be safe and submit right now?

```
$ 1091 style crepe_stand.c
$ 1091 autotest-stage 01 crepe_stand
$ give dp1091 ass2_crepe_stand crepe_stand.c
```

Stage 2

In **Stage 2** of this assignment, you will be using more advanced linked list concepts to add and move nodes within a linked list and perform operations using the values stored inside those nodes.

Specifically, this will include:

- Inserting a crepe into the current day's list of crepe orders at a given position.
- Calculating statistics about the crepes that have been ordered so far on the current day.
- Calculating the price of a single crepe based on its ordered details.
- Calculating the total income of the current day's orders.
- Calculating a given customer"s bill for the current day.

HINT:

Additionally, many of the commands in this stage partially overlap in logic. You will be assessed on how well you utilise functions to break up and structure your code and reduce repetition.

Stage 2.1 - Inserting Crepes

Currently, we have a way of appending a crepe to the end of the current day's list of orders, but there"s no way to insert a crepe anywhere else in your list. What if you got so busy making crepes you forgot to enter a customer"s order, and you"ll need to make it before crepes ordered later? Or what if you had an extra special order that needed to come before all the other crepes?

That"s why you will be implementing the Insert Crepe command for **Stage 2.1**, which will insert a new crepe at the specified position, rather than at the end of the orders list.

Command: Insert crepe into current day's orders at given position

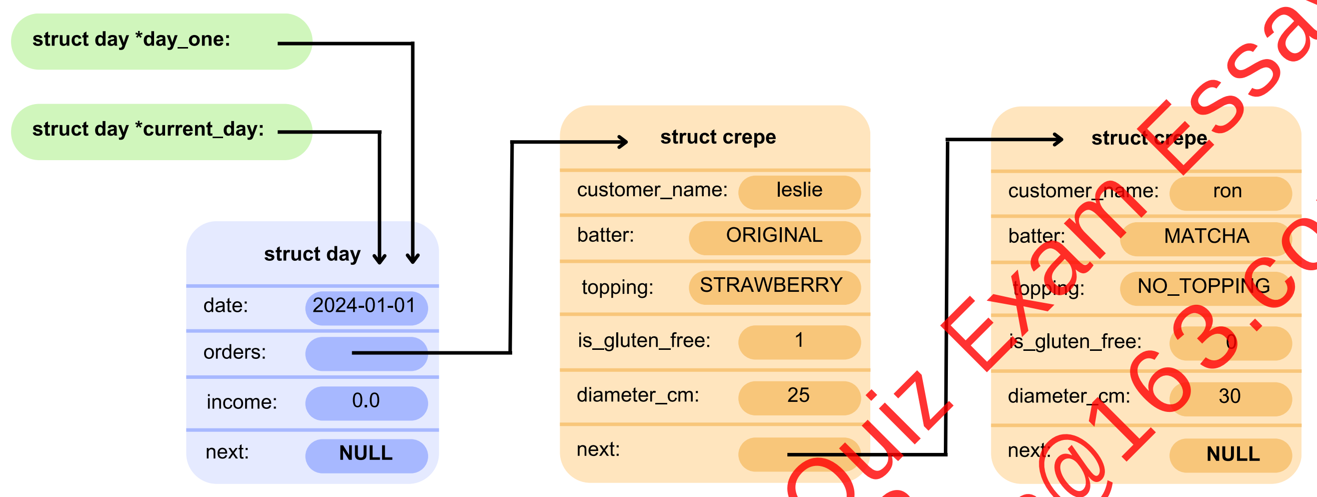
```
Enter Command: i [position] [crepe type] [customer name] {OPTIONAL: [batter] [topping] [gluten free] [size]}
```

The `i` command takes in either 3 or 7 arguments, similar to appending a crepe. This depends on whether a custom crepe or a signature crepe has been ordered.

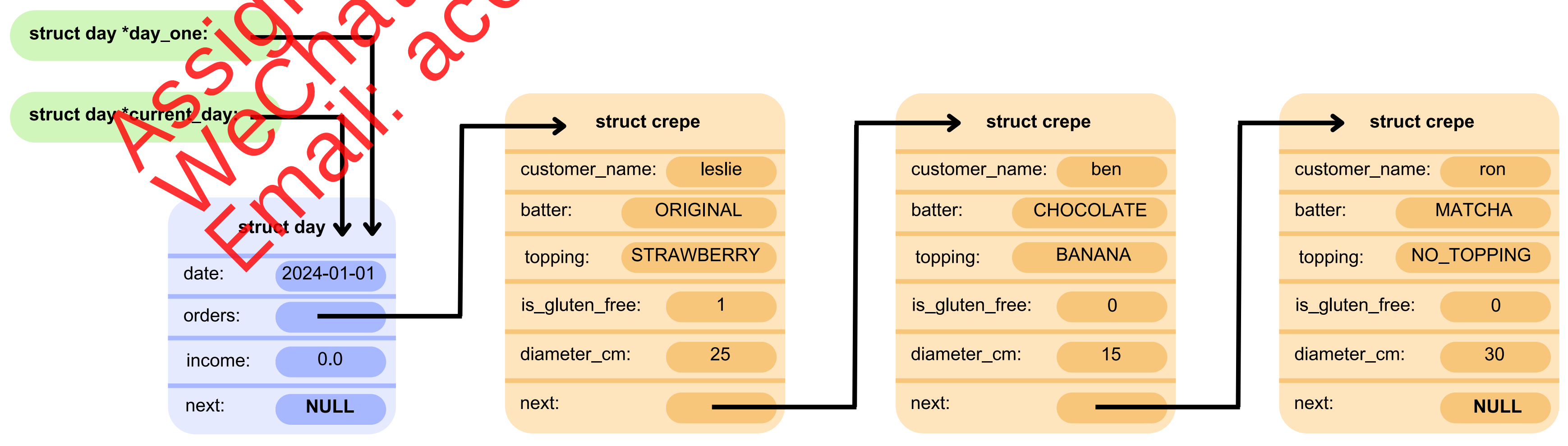
`[crepe type]` will be:

- `custom` , for a custom crepe,
- `matcha` , for a Matcha Madness crepe,
- `strawberry` , for a Strawberry Special crepe, or
- `chocolate` , for a Chocolate Connoisseur crepe.

Inserting a new crepe should look something like this:



Stage 2.1: Before inserting a new crepe in the middle of the orders list *(click on image to expand)*



Stage 2.1: After inserting a new crepe in the middle of the orders list *(click on image to expand)*

There are two functions in `crepe_stand.c` that you will have to implement for **Stage 2.1**.

1. `insert_crepe_custom` will insert a newly created custom crepe to the specified `position` in the current day"s orders list.
2. `insert_crepe_signature` will append a newly created signature crepe (Matcha Madness, Strawberry Special or Chocolate Connoisseur) to the specified `position` in the orders list.

You will find the following function stubs in `crepe_stand.c` :

```
int insert_crepe_custom(struct day *current_day,
                        int position,
                        char *customer_name,
                        int batter,
                        int topping,
                        int is_gluten_free,
                        int diameter_cm) {
    // TODO: implement this function
    printf("Insert Crepe Custom not yet implemented.\n");
    exit(1);
}

int insert_crepe_signature(struct day *current_day,
                           int position,
                           char *customer_name,
                           char *crepe_type) {
    // TODO: implement this function
    printf("Insert Crepe Signature not yet implemented.\n");
    exit(1);
}
```

NOTE:

Both of these functions contain a parameter named `current_day` which contains a pointer to the head of the list of its orders, called `orders`. This is what you will append the new custom crepe to.

For both of these functions, your program should create a new `struct crepe` and then insert it into the current day"s list of crepe orders after the given `position`. **The list is indexed from 1, meaning the head of the list is position 1. The `position` given means that the new crepe should take that position in the list**, e.g. if the position was `1`, the new crepe should become the new head of the current day"s list of orders.

Both functions return an `int` - this value indicates whether or not the appending was successful or not. As we handled error checking in **Stage 1.4**, you will now need to check if any of the arguments provided are invalid. If they are, you will need to return the appropriate constant, defined in `crepe_stand.h`:

```
#define VALID_CREPE 0
#define INVALID_BATTER -2
#define INVALID_TOPPING -3
#define INVALID_GFO -4
#define INVALID_SIZE -5
#define INVALID_POSITION -6
```

`main.c` will then handle the printing accordingly.

If a crepe is invalid, it should not be inserted to the list of crepes.

Error handling

Your program should handle the same errors as **stage 1.4**.

Additionally, your program should also check if `position` is invalid. `position` is invalid if it is less than 1. If the position is greater than the length of the list, it should be appended to the tail of the list.

Assumptions

- If the given `position` is 1, the new crepe should become the head of the list.
- If the given `position` is larger than the length of the list + 1, the new crepe should be added to the tail of the list.
- `crepe_type` in `insert_crepe_signature` will always be one of `matcha`, `strawberry` or `chocolate`.
- You don"t need to do error checking to see if a customer has the same name as another customer, you can assume it is the same customer.

Clarifications

- The list is indexed from 1, meaning the head of the list is position 1. The `position` given means that the new crepe should take that position in the list, and should point to the crepe that previously held that position.

Examples

+ Example 2.1.1: Insert a signature and a custom crepe at the head

+ Example 2.1.2: Insert into the middle of the list

+ Example 2.1.3: Insert at the end of the list

+ Example 2.1.4: Handling errors when inserting

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 02_01 crepe_stand
```

Stage 2.2 - Crepe Stand Statistics

Now that we've started making crepes, we'd like to get some information about all the crepes ordered so far on the current day! In **Stage 2.2**, you'll be tasked with calculating some statistics:

- How many crepes were ordered in total?
- How many were original batter, chocolate batter, and matcha batter?
- How many crepes were gluten-free?
- What was the most popular topping?
- How many small, medium and large crepes were there?

Command: Calculate current day''s statistics

```
Enter Command: s
```

There is one function in `crepe_stand.c` that you will have to implement for **Stage 2.2**. `get_stats` should calculate all the statistics listed above, and store them into a `struct stats` to return back to `main.c`, which will handle all the printing of the statistics for you.

You will find the following struct definition in `crepe_stand.h`:

```
struct stats {
    int total_crepes;
    int original;
    int chocolate;
    int matcha;
    int gluten_free;
    enum topping_type popular_topping;
    int small;
    int medium;
    int large;
};
```

You will find the following function stub in `crepe_stand.c`:

```
struct stats get_stats(struct day *current_day) {
    // TODO: implement this function
    printf("Get Stats not yet implemented.\n");
    exit(1);
}
```

- `struct day *current_day` is a pointer to the current day of crepe stand operations.
- `get_stats` should return a `struct stats` representing the `current_day` ''s statistics.

To implement `get_stats`, you will have to look through all the crepes on the current day, provided to you as a pointer to the `current_day`, and calculate the above listed statistics. You will need to create a `struct stats`, and store into each of the fields the appropriate stat. e.g. for the field `total_crepes`, this should be set to the total number of crepes stored on the current day.

- Any field that returns an `int` should be set to the number of crepes matching that field.
- The field `popular_topping` should be set to the `enum topping_type` that was ordered the most frequently. If no topping was ordered more than another, then `topping` should be set to `NO_TOPPING`.
- Small crepes have diameter: 10cm <= diameter < 20cm.
- Medium crepes have diameter: 20cm <= diameter < 30cm.
- Large crepes have diameter: 30cm <= diameter < 40cm.

Clarifications

- If no crepes have been ordered on the current day, the field `total_crepes` should be set to 0, and `main.c` will handle the printing accordingly.
- The field `popular_topping` should be set to `NO_TOPPING` in both the case where `NO_TOPPING` itself is the most popular topping, and when there is no topping type that has been ordered more than another.

Examples

+ Example 2.2.1: No crepes ordered, calculate stats

+ Example 2.2.2: Lots of crepes ordered, calculate stats

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 02_02 crepe_stand
```

Stage 2.3 - Calculate Price of a Single Crepe

We've started making crepes, but to make sure we don't start going out of business, we're unfortunately going to need to charge for our crepes! So, in **Stage 2.3**, you will be finding out the price of a single crepe, after being given its position in the list of orders.

Command: Calculate the price of a single crepe

```
Enter Command: C [position]
```

There is one function in `crepe_stand.c` that you will have to implement for **Stage 2.4**. `calculate_price` will calculate the cost of a single crepe in the current day's list of orders, from the provided `position`.

You will find the following function stub in `crepe_stand.c`:

```c
double calculate_price(struct day *current_day, int position) {
    // TODO: implement this function
    printf("Calculate Price not yet implemented.\n");
    exit(1);
}
```

- `struct day *current_day` is a pointer to the current day of crepe stand operations.
- `position` is the position in the orders list of the crepe to calculate the price of. Position is indexed from 1, in the same way as earlier stages.
- `calculate_price` should return a `double` representing the price of the specified crepe.

Crepes are priced as follows:

Ingredient	Price

Original Batter	$8.00
Chocolate Batter	$8.00
Matcha Batter	$9.50
Banana Topping	Extra $2.00
Strawberry Topping	Extra $2.00
Nutella Topping	Extra $3.50
Gluten-free	Extra $0.60
Upgrade to Medium Size	Extra $2.50
Upgrade to Large Size	Extra $3.50

Error handling

- If the `position` given does not correspond to a crepe in the current day"s list of orders, the constant `INVALID_POSITION` should be returned instead. `INVALID_POSITION` is defined in `crepe_stand.h` for you.

Assumptions

- Medium crepes have diameter: 20cm <= diameter < 30cm.
- Large crepes have diameter: 30cm <= diameter < 40cm.

Examples

+ Example 2.3.1: Calculate the price of a valid crepe

+ Example 2.3.2: Position does not correspond to a crepe in list

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 02_03 crepe_stand
```

Stage 2.4 - Calculate Total Income for the Day

Now we would like to know how much income we've made so far today! So in **Stage 2.4**, you will need to calculate the total income (the sum of the cost of each crepe in the list of orders).

Command: Calculate total income for the current day

```
Enter Command: t
```

There is a function in `crepe_stand.c` that you will have to implement for **Stage 2.4**. `total_income` will calculate the total income made on the current day then return the result for `main.c` to print. The total income is the sum of the cost of all crepes ordered for a day.

You will find the following function stub in `crepe_stand.c` :

```
double total_income(struct day *current_day) {
    // TODO: implement this function
    printf("Total Income not yet implemented.\n");
    exit(1);
}
```

If no orders have been placed, you should return `0` , and `main.c` will handle the printing for you.

Examples

+ Example 2.4.1: No crepes ordered yet

+ Example 2.4.2: One crepe ordered

+ Example 2.4.3: Multiple crepes ordered

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 02_04 crepe_stand
```

Stage 2.5 - Calculate a Customer"s Bill for the Day

If a customer has multiple orders, they'd like to know what their total bill is! So in **Stage 2.5**, you will be given a customer name, and will need to find how many crepes they ordered and what their bill is! The bill is calculated as the sum of the costs of each crepe ordered by the specified customer.

Command: Calculate a customer"s bill for the current day

```
Enter Command: b [customer name]
```

There is one function in `crepe_stand.c` that you will have to implement for **Stage 2.5**. `customer_bill` will need to calculate the given customer"s total bill for the day and the amount of crepes they ordered. These will be stored into a `struct bill` , then the result will be returned for `main.c` to print.

You will find the following struct definition in `crepe_stand.h` :

```
struct bill {
    double total_price;
    int crepe_count;
};
```

- `total_price` represents the total cost for the customer, calculated as the sum of each individual crepe the customer ordered.
- `crepe_count` represents the total number of crepes ordered by the customer.

You will find the following function stub in `crepe_stand.c` :

```
struct bill customer_bill(struct day *current_day, char *customer_name) {
    // TODO: implement this function
    printf("Customer Bill not yet implemented.\n");
    exit(1);
}
```

- `struct day *current_day` is a pointer to the current day of crepe stand operations.
- `char *customer_name` is the name of the customer to calculate the bill of.

If the specified customer has not ordered any crepes (i.e. no crepes in the current day"s list of orders match the customer name), then the field `crepe_count` should be set to 0, and `main.c` will handle the printing.

Examples

+ Example 2.5.1: No crepes ordered yet

+ Example 2.5.2: One crepe ordered

+ Example 2.4.3: Multiple crepes ordered

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 02_05 crepe_stand
```

Testing and Submission

Remember to do your own testing

Are you finished with this stage? If so, you should make sure to do the following:

- Run `1091 style` and clean up any issues a human may have reading your code. Don"t forget -- **20%** of your mark in the assignment is based on style and readability!
- Autotest for this stage of the assignment by running the `autotest-stage` command as shown below.
- Remember -- *give early and give often*. Only your last submission counts, but why not be safe and submit right now?

```
$ 1091 style crepe_stand.c
$ 1091 autotest-stage 02 crepe_stand
$ give dp1091 ass2_crepe_stand crepe_stand.c
```

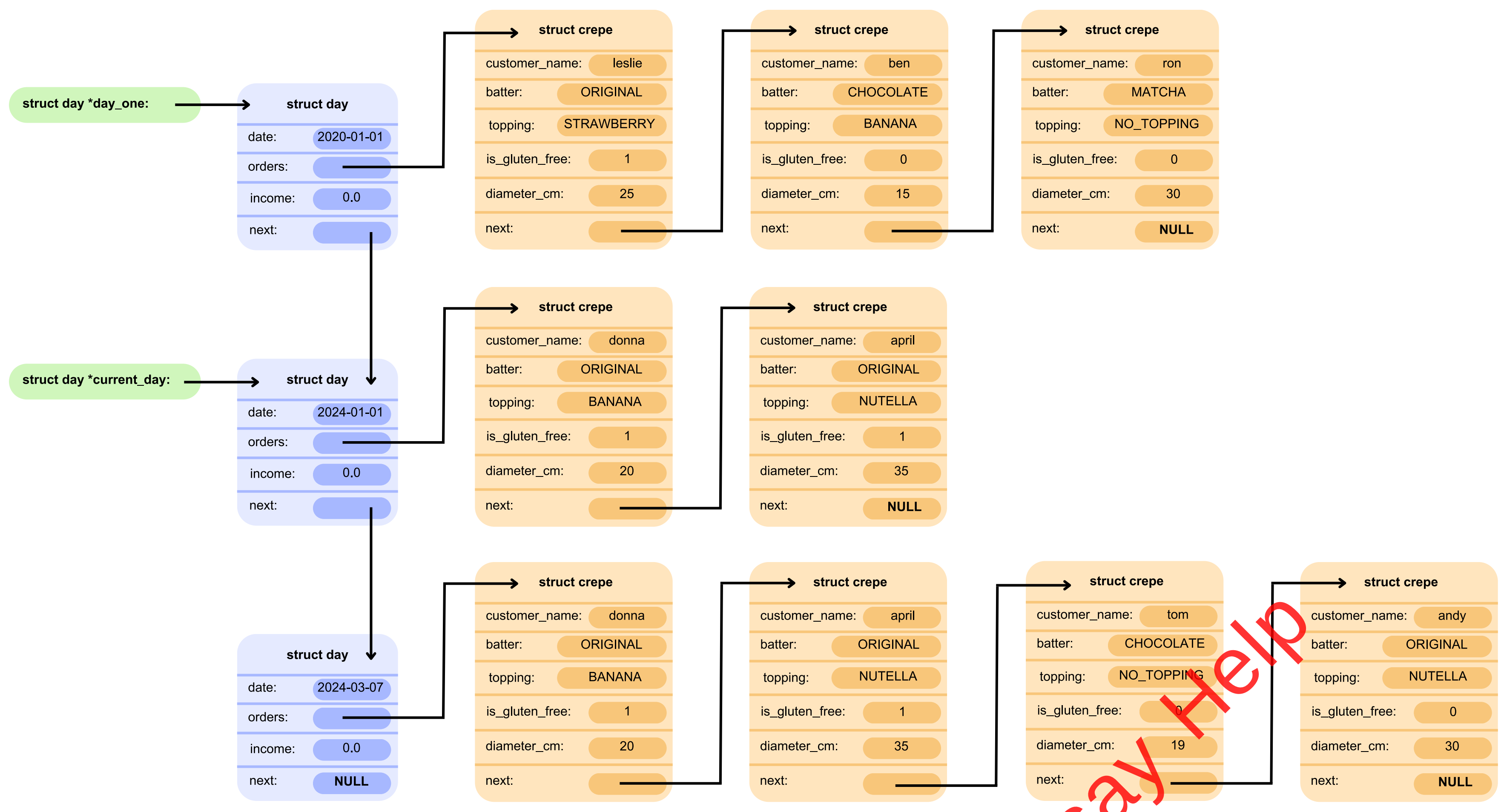
Stage 3

In **Stage 3** of this assignment, you will be manipulating 2D linked lists by adding more than just the current day to the list of days. You will also be managing your memory usage by freeing memory and preventing memory leaks.

Specifically, this will include:

- Creating and inserting a new day to the list of days.
- Moving between different days to change the current day.
- Printing all days in the list of days and some of their associated information.
- Removing a specified crepe from the current day"s list of orders.
- Removing a specified day from the list of days.

By the end of this stage, your crepe stand will look something like:



Stage 3: A linked list of days, each with a linked list of crepes *(click on image to expand)*

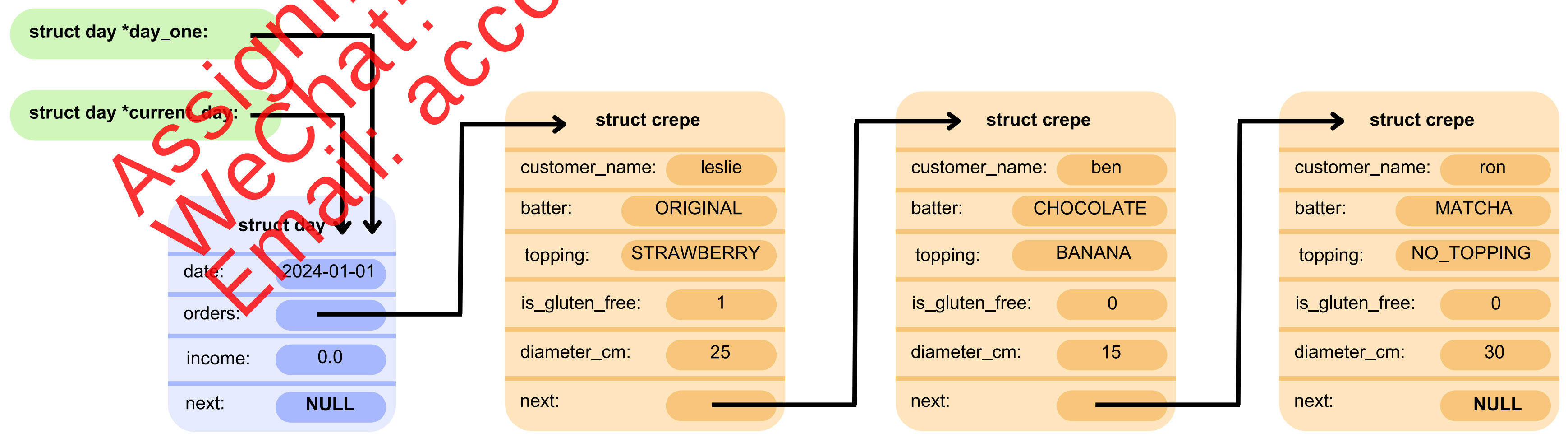
Stage 3.1 - Inserting a New Day

We currently only have one day of crepe sales, the `current_day`, and we would like to be open for more than just today, so we will add more days. We also need to be able to add previous days, so that we can record previous sales, allowing us to keep better track of how our crepe stand performs. Whenever a day is inserted, it should be inserted in chronological order.

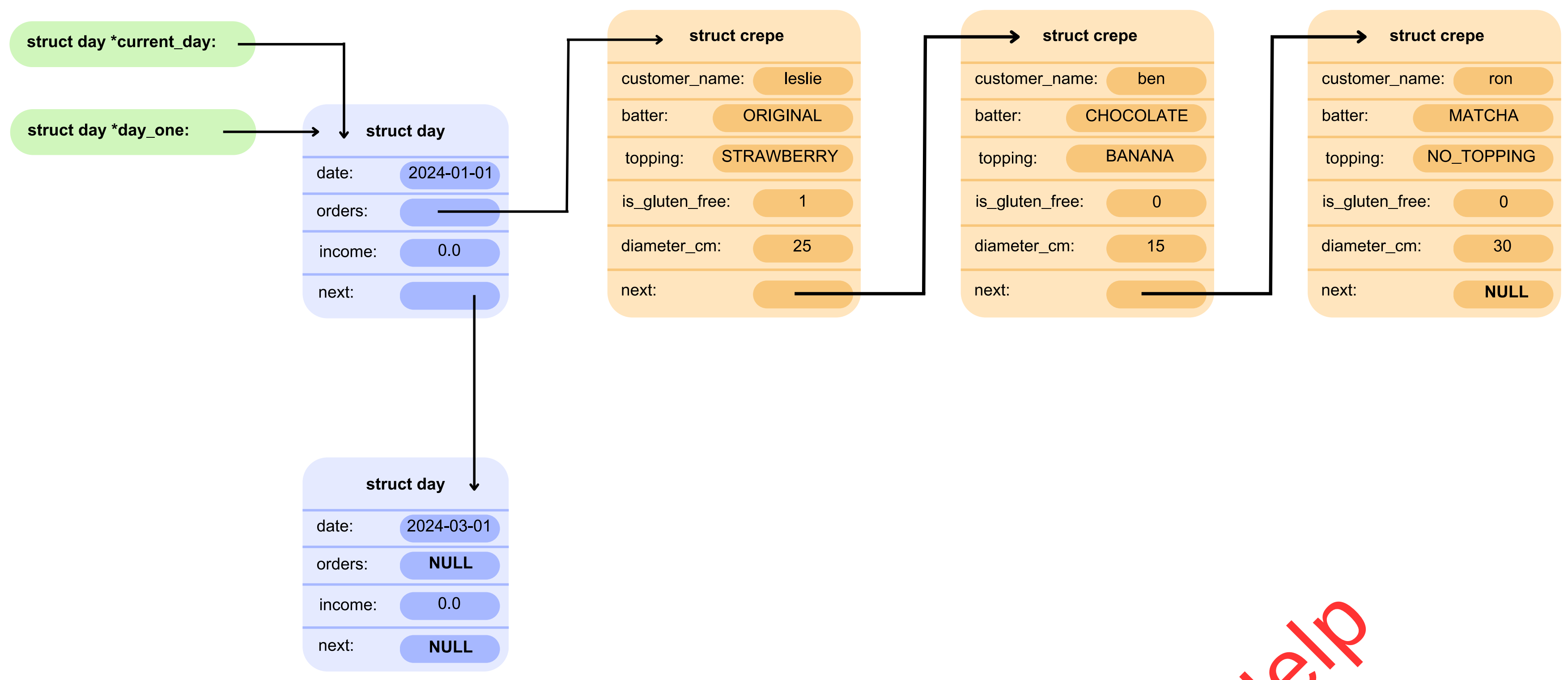
Command: New day

```
Enter Command: n [date in YYYY-MM-DD format]
```

Inserting a new day should look something like this:



Stage 3.1: A linked list of one day before inserting a new day *(click on image to expand)*



Stage 3.1: A linked list of days after inserting a new day *(click on image to expand)*

There is one function in `crepe_stand.c` that you will have to implement for **Stage 3.1**. `new_day` should create a new day (`create_day` from 1.1 will be helpful here), and insert the new day into the list of days in chronological order, e.g. the date `2020-01-01` should be inserted before `2024-01-01` . `new_day` should return a pointer to the head of the list of days.

You will find the following function stub in `crepe_stand.c` :

```
struct day *new_day(struct day *day_one, struct date date) {
    // TODO: implement this function
    printf("New Day not yet implemented.\n");
    exit(1);
}
```

Assumptions

- If the date of the new day provided corresponds to an existing day in the list of days, `NULL` should be returned instead.
- Any date provided will be valid

Examples

+ Example 3.1.1: Insert a day into the end of the list

+ Example 3.1.2: Insert a day into the start of the list

+ Example 3.1.3: Insert a day into the middle of the list

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 03_01 crepe_stand
```

Stage 3.2 - Print All Days in the List

Now that we have inserted some new days, we would like to be able to see all this information! In this stage you will be printing out each day in the list of days, and for each you will need to find the following information:

- The date of the day as a `struct date`.
- Whether or not the day is the current day.
- The most popular signature crepe ordered on the day.
- The total income made on the day.

Command: Display days

```
Enter Command: d
```

There is one function to implement for **Stage 3.2**, called `display_days`. The function stub in `crepe_stand.c` reads as follows:

```c
void display_days(struct day *day_one, struct day *current_day) {
    // TODO: implement this function
    printf("Display Days not yet implemented.\n");
    exit(1);
}
```

Two functions have been provided for you in `crepe_stand.c` to help print the list of crepes: `print_single_day` (prints a single day) and `print_arrow` (prints an arrow to connect the crepes).

`print_single_day` is given as follows:

```c
void print_single_day(int is_current_day,
                      struct date date,
                      char most_pop,
                      double income) {
    printf("---------------------\n");
    if (is_current_day) {
        printf("!CURRENT DAY!\n");
    }
    printf("Date: %04d-%02d-%02d\n", date.year, date.month, date.day);
    if (most_pop == SIG_STRAWBERRY) {
        printf("Most popular signature crepe: Strawberry Special\n");
    } else if (most_pop == SIG_MATCHA) {
        printf("Most popular signature crepe: Matcha Madness\n");
    } else if (most_pop == SIG_CHOCOLATE) {
        printf("Most popular signature crepe: Chocolate Connoisseur\n");
    } else if (most_pop == NO_SIG_SOLD) {
        printf("There were no signature crepes sold!\n");
    } else if (most_pop == NO_SIG_MOST_POP) {
        printf("No signature crepe was more popular than another!\n");
    } else {
        printf("ERROR: %c, unknown most popular character\n", most_pop);
    }
    printf("Total income: $%.2lf\n", income);
    printf("---------------------\n");
}
```

There are also constants defined in `crepe_stand.c` you can use to supply as arguments to `print_single_day`, defined as:

```c
#define SIG_STRAWBERRY 's'
#define SIG_MATCHA 'm'
#define SIG_CHOCOLATE 'c'
#define NO_SIG_SOLD 'n'
#define NO_SIG_MOST_POP 'N'
```

Days should be printed in the following format:

```
Enter Command: d
---------------------
!CURRENT DAY!
Date: 2020-01-01
Most popular signature crepe: Strawberry Special
Total income: $13.10
---------------------
    |
    v
---------------------
Date: 2021-01-01
There were no signature crepes sold!
Total income: $8.00
---------------------
    |
    v
---------------------
Date: 2022-01-01
There were no signature crepes sold!
Total income: $0.00
---------------------
Enter Command:
```

NOTE:

The last day should have no arrow after it.

Assumptions

- There will always be at least one `struct day` in the list of days
- Custom crepes with the same details as signature crepes are not considered signature crepes

Examples

+ Example 3.2.1: Print a single day with some orders

+ Example 3.2.2: Print multiple days

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 03_02 crepe_stand
```

Stage 3.3 - Move Between Days

Now we need to move between the different days, changing which one is the current day. This way we can add crepe sales to future days, and back up any previous days sales by adding their sales as well.

Command: Next day

```
Enter Command: >
```

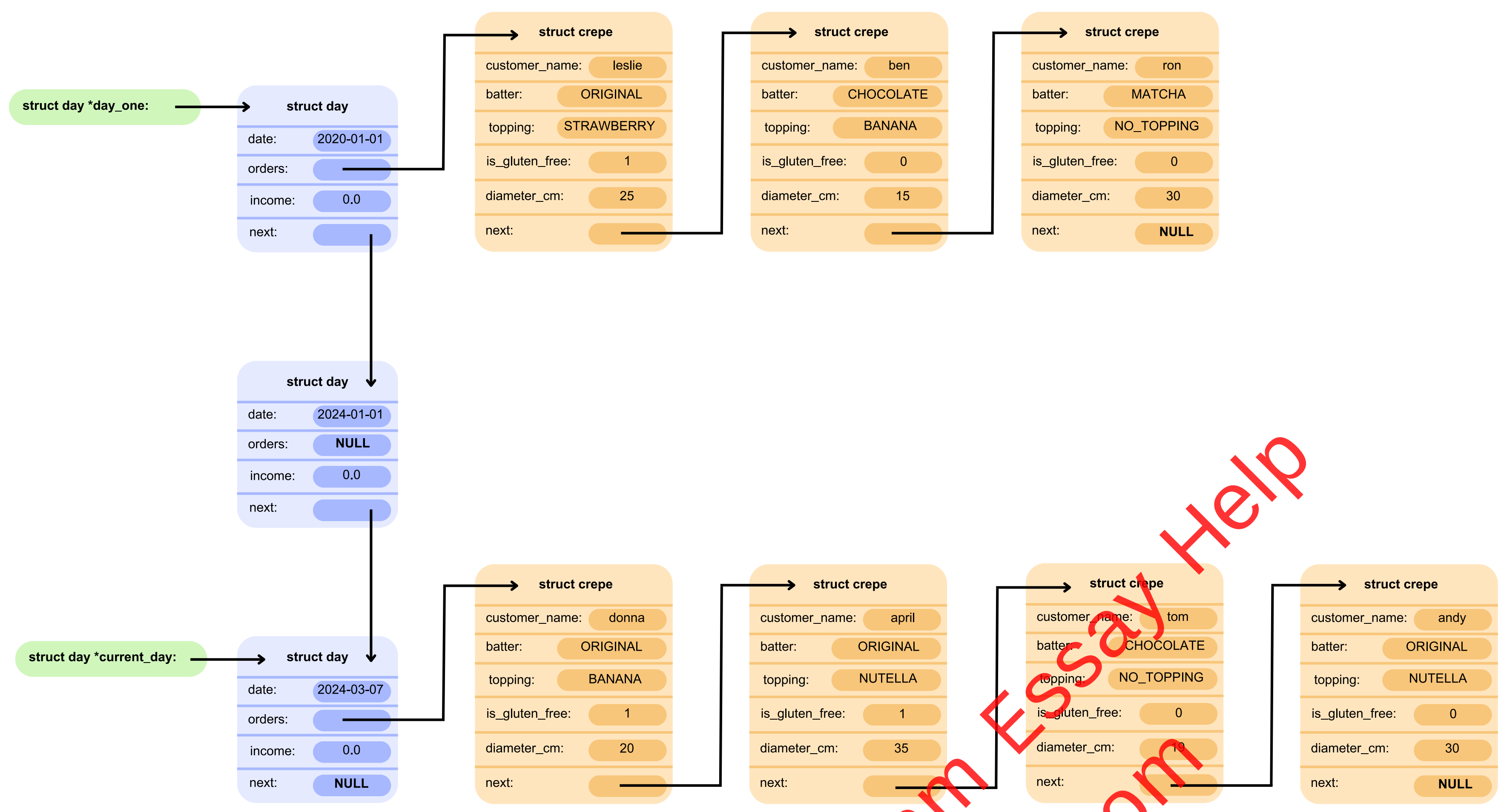
Command: Previous day

```
Enter Command: <
```

When `>` is entered, the current day should change to the next day in the list. If the next day is `NULL`, the current day should cycle back to the first day.

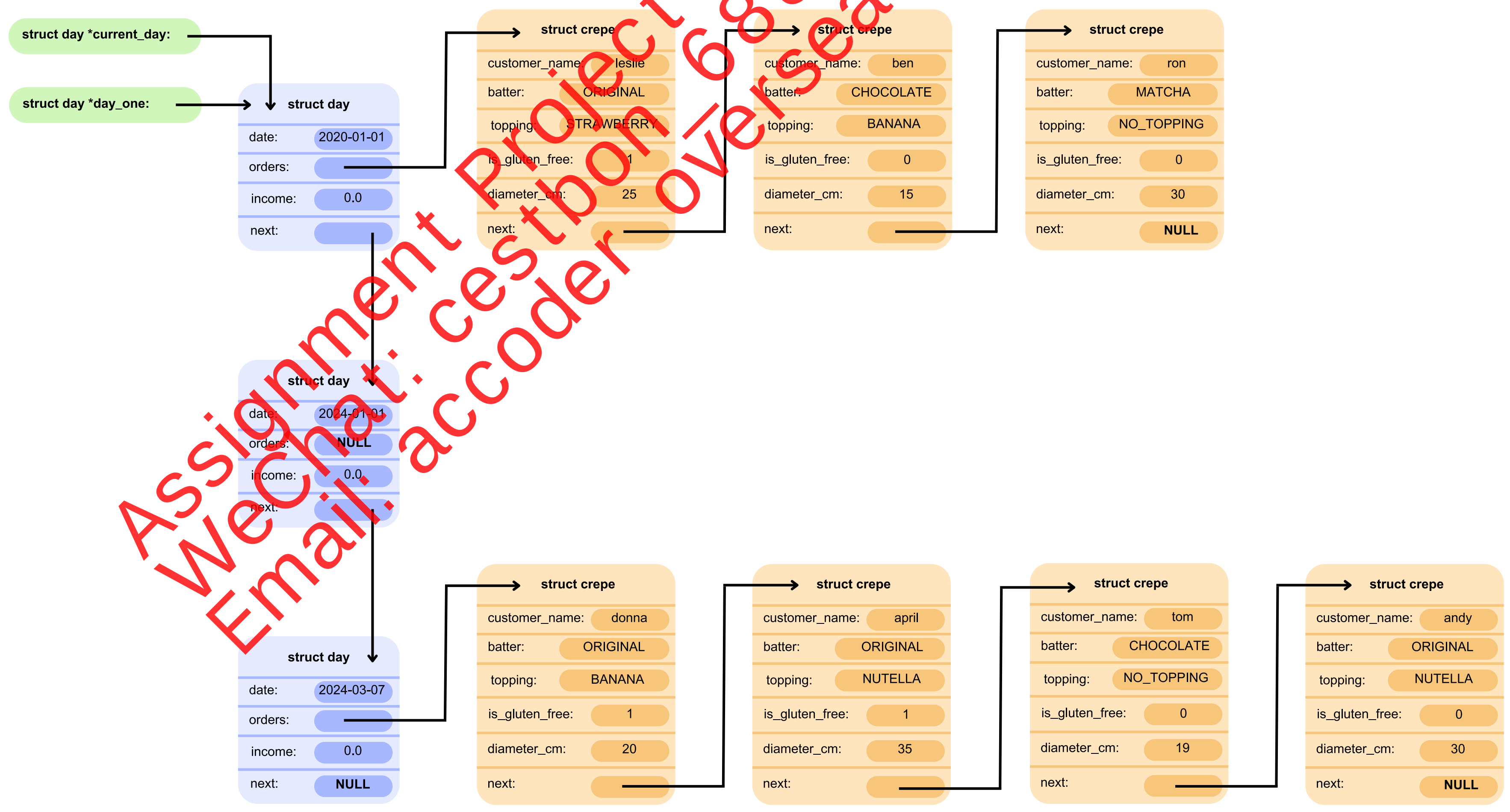
When `<` is entered, the current day should change to the previous day in the list. If the current day was the first day in the list, the new current day should be the last day in the list.

For example, before entering `>` , the current day was the last day in the list:



Stage 3.3: A linked list of days each with a linked list of crepes, current day is the last day in the list. *(click on image to expand)*

After entering `>` , the current day is the first day in the list:



Stage 3.3: A linked list of days each with a linked list of crepes, current day is the first day in the list. *(click on image to expand)*

There is one function to implement for **Stage 3.3**, called `cycle_days` . The function stub in `crepe_stand.c` reads as follows:

```c
struct day *cycle_days(char command,
                       struct day *day_one,
                       struct day *current_day) {
    // TODO: implement this function
    printf("Cycle Days not yet implemented.\n");
    exit(1);
}
```

Assumptions

- There will always be at least one `struct day` in the list of days.
- Cycling forward or backward when there is only one day will still print the corresponding message in `main.c` , and `current_day` should stay the same.

Examples

+ Example 3.3.1: Cycle forward

+ Example 3.3.2: Cycle backward

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 03_03 crepe_stand
```

Stage 3.4 - Remove a Crepe

Whoops! We added a crepe by accident, and now need to get rid of it from the current day"s orders list!

Command: Remove Crepe

```
Enter Command: r [position]
```

There is one function to implement for **Stage 3.4**, called `remove_crepe` . The function stub in `crepe_stand.c` reads as follows:

```
int remove_crepe(struct day *current_day, int position) {
    // TODO: implement this function
    printf("Remove Crepe not yet implemented.\n");
    exit(1);
}
```

`remove_crepe` should return a constant indicating whether or not the crepe was removed successfully. These constants are defined in `crepe_stand.h` . `remove_crepe` should return `VALID_CREPE` if the crepe was successfully removed.

Error handling

- `remove_crepe` should return `INVALID_POSITION` if the provided position does not correspond to a crepe in the current day"s orders list.

Examples

+ Example 3.4.1: Remove a crepe

+ Example 3.4.2: Attempt to remove crepe at invalid position

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 03_04 crepe_stand
```

Stage 3.5 - Remove a Day

Now we need to remove a whole day! In this stage, you will need to remove a day and all its ordered crepes from the list of days.

Command: Remove day

```
Enter Command: R [date of day to remove in YYYY-MM-DD format]
```

In **stage 3.5**, you will need to implement the functions `remove_day` and `free_crepe_stand` .

The function stub for `remove_day` in `crepe_stand.c` reads as follows:

```c
struct day *remove_day(struct day *day_one,
                       struct day **current_day,
                       struct day date) {
    // TODO: implement this function
    printf("Remove Day not yet implemented.\n");
    exit(1);
}
```

If the day to remove is the only day in the list, you should free it, and create a new day with the date `2024-01-01` . This way, even if we delete the only day in the list, we can still keep using all of our other crepe stand commands.

Furthermore, if the current day is the day being removed it will need to be appropriately changed. In this case your program should cycle the `current_day` forward, in the same way that cycling forward occurs in **Stage 3.3**, e.g. if the current day is being deleted, current day should then become the day after.

Additionally, when the program ends, all malloc'd memory should be freed, and there should be no memory leaks. To do this, you will need to implement the following stub function in `crepe_stand.c` :

```c
void free_crepe_stand(struct day *day_one) {
    // TODO: implement this function
}
```

This function is called at the end of the `main` function in `main.c` , occurring after the user hits `CTRL + D` . Currently, it is not doing anything, so you will need to make sure that this function frees all malloc'd memory.

Error handling

- If no day in the list matches the given `date` , you should return `NULL` instead and `main.c` will handle the printing.

Assumptions

- All crepes in the day to delete's orders list should be freed as well.

Clarifications

- From now on, you should check for memory leaks by compiling with `dcc --leak-check crepe_stand.c main.c -o crepe_stand` .
- Autotests and marking tests from this stage onwards onwards will check for memory leaks.

Examples

+ Example 3.5.1: Remove Only Day

+ Example 3.5.2: Remove First Day

+ Example 3.5.3: Remove Last Day

+ Example 3.5.4: Remove Middle Day

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 03_05 crepe_stand
```

Testing and Submission

Remember to do your own testing

Are you finished with this stage? If so, you should make sure to do the following:

- Run `1091 style` and clean up any issues a human may have reading your code. Don't forget -- **20%** of your mark in the assignment is based on style and readability!
- Autotest for this stage of the assignment by running the `autotest-stage` command as shown below.
- Remember -- *give early and give often*. Only your last submission counts, but why not be safe and submit right now?

```
$ 1091 style crepe_stand.c
$ 1091 autotest-stage 03 crepe_stand
$ give dp1091 ass2_crepe_stand crepe_stand.c
```

Stage 4

This stage is for students who want to challenge themselves, and solve more complicated linked lists and programming problems, such as:

- Calculating the total income for all weekends and the total income for all weekdays.
- Calculating the maximum profit period of a given year

Stage 4.1 - Comparing Weekday vs Weekend Sales

In **Stage 4.1**, we'd like to calculate the profit made across all weekdays and the profit made across all weekends.

Command: Compare weekday and weekend

```
Enter Command: w
```

There is one function in `crepe_stand.c` that you will have to implement for **Stage 4.1**. `compare_weekday_end` should calculate the total profit made over all weekdays and the total profit made over all weekends, and return this information in a `struct profits` .

You will find the following struct definition in `crepe_stand.h` :

```
struct profits {
    double weekday;
    double weekend;
};
```

You will find the following function stub in `crepe_stand.c` :

```
struct profits compare_weekday_end(struct day *day_one) {
    // TODO: implement this function
    printf("Compare weekday end not yet implemented.\n");
    exit(1);
}
```

To calculate profit for any given day, it is the total income made that day minus $35 per day to run the crepe stand on weekdays, and minus $50 per day to run the crepe stand on weekends.

To find out whether a date is a weekday or weekend, there are several different formulas you can use, however you should pick one that is applicable to the Gregorian calendar.

Assumptions

- Any dates used in autotesting will be after the year 1582.

Examples

+ Example 4.1.1: Comparing many sales on weekdays and weekends

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 04_01 crepe_stand
```

Stage 4.2 - Finding the Maximum Profit Period of a Given Year

In **Stage 4.2**, we would like to find the longest stretch of time within a given year with the most profits. This is so that we can better prepare for the future as we'll know when our most profitable time periods are!

Command: Maximum profit period in year

```
Enter Command: m [year to check in format YYYY]
```

There is one function in `crepe_stand.c` that you will have to implement for **Stage 4.2**. `max_profit_period` should find the start date and end date of the maximum profit period and the total profit made during that period. It should then store this into a `struct max_profit` and return it.

You will find the following struct definition in `crepe_stand.h` :

```
struct max_profit {
    struct date start_date;
    struct date end_date;
    double profit;
};
```

You will find the following function stub in `crepe_stand.c` :

```
struct max_profit max_profit_period(struct day *day_one, int year) {
    // TODO: implement this function
    printf("Max Profit Period not yet implemented.\n");
    exit(1);
}
```

To calculate profit for a given day, it is the total income made that day minus $35 per day to run the crepe stand on weekdays, and minus $50 per day to run the crepe stand on weekends.

The maximum profit period is the streak of consecutive days in a year with the highest summed profit. If all days in the year have negative profit, the maximum profit should be that of the least loss.

For example:

- Date: 2024-01-01, Profit: $12
- Date: 2024-03-03, Profit: $35
- Date: 2024-03-06, Profit: -$12
- Date: 2024-04-04, Profit: $20
- Date: 2024-05-05, Profit: -$35
- Date: 2024-06-06, Profit: $30

The max profit period here is from 2024-01-01 to 2024-04-04 with a total profit during this period of $55.

Clarifications

- If the maximum profit period only occured on one single day, the `year` field in `end_date` should be set to `0` .

Error handling

- If the year given is not a valid year (i.e. there are no days with the given year), then both `year` fields in `start_date` and `end_date` should be set to `0` .

Assumptions

- Any dates used in autotesting will be after the year 1582.
- You may assume that no two spans will have the same max profit.

Examples

+ Example 4.2.1: One day of orders only

+ Example 4.2.2: Multiple dates in year

+ Example 4.2.3: All negative profits in year

NOTE:

You may like to autotest this section with the following command:

```
1091 autotest-stage 04_02 crepe_stand
```

Testing and Submission

Remember to do your own testing

Are you finished with this stage? If so, you should make sure to do the following:

- Run `1091 style` and clean up any issues a human may have reading your code. Don"t forget -- **20%** of your mark in the assignment is based on style and readability!
- Autotest for this stage of the assignment by running the `autotest-stage` command as shown below.
- Remember -- *give early and give often*. Only your last submission counts, but why not be safe and submit right now?

```
$ 1091 style crepe_stand.c
$ 1091 autotest-stage 04 crepe_stand
$ give dp1091 ass2_crepe_stand crepe_stand.c>
```

Assessment

Assignment Conditions

- **Joint work** is **not permitted** on this assignment.

  This is an individual assignment.

  The work you submit must be entirely your own work. Submission of any work even partly written by any other person is not permitted.

  The only exception being if you use small amounts (< 10 lines) of general purpose code (not specific to the assignment) obtained from a site such as Stack Overflow or other publicly available resources. You should attribute the source of this code clearly in an accompanying comment.

  Assignment submissions will be examined, both automatically and manually for work written by others.

  Do not request help from anyone other than the teaching staff of DPST1091.

Do not post your assignment code to the course forum - the teaching staff can view assignment code you have recently autotested or submitted with give.

Rationale: this assignment is an individual piece of work. It is designed to develop the skills needed to produce an entire working program. Using code written by or taken from other people will stop you learning these skills.

- The use of **code-synthesis tools**, such as **GitHub Copilot**, is **not permitted** on this assignment.

  The use of **Generative AI** to generate code solutions is not permitted on this assignment.

  Rationale: this assignment is intended to develop your understanding of basic concepts. Using synthesis tools will stop you learning these fundamental concepts.

- **Sharing, publishing, distributing** your assignment work is **not permitted**.

  Do not provide or show your assignment work to any other person, other than the teaching staff of DPST1091. For example, do not share your work with friends.

  Do not publish your assignment code via the internet. For example, do not place your assignment in a public GitHub repository.

  Rationale: by publishing or sharing your work you are facilitating other students to use your work, which is not permitted. If they submit your work, you may become involved in an academic integrity investigation.

- **Sharing, publishing, distributing your assignment work after the completion of DPST1091 is not permitted**.

  For example, do not place your assignment in a public GitHub repository after DPST1091 is over.

  Rationale:DPST1091 sometimes reuses assignment themes, using similar concepts and content. If students in future terms can find your code and use it, which is not permitted, you may become involved in an academic integrity investigation.

Violation of the above conditions may result in an academic integrity investigation with possible penalties, up to and including a mark of 0 in DPST1091 and exclusion from UNSW.

Relevant scholarship authorities will be informed if students holding scholarships are involved in an incident of plagiarism or other misconduct. If you knowingly provide or show your assignment work to another person for any reason, and work derived from it is submitted - you may be penalised, even if the work was submitted without your knowledge or consent. This may apply even if your work is submitted by a third party unknown to you.

If you have not shared your assignment, you will not be penalised if your work is taken without your consent or knowledge.

For more information, read the UNSW Student Code , or contact the course account. The following penalties apply to your total mark for plagiarism:

0 for the assignment	Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 for the assignment	Submitting any other person's work. This includes joint work.
0 FL for DPST1091	Paying another person to complete work. Submitting another person's work without their consent.

Submission of Work

You should submit intermediate versions of your assignment. Every time you autotest or submit, a copy will be saved as a backup. You can find those backups here , by logging in, and choosing the yellow button next to `ass2_crepe_stand` .

Every time you work on the assignment and make some progress, you should copy your work to your CSE account and submit it using the `give` command below.

It is fine if intermediate versions do not compile or otherwise fail submission tests.

Only the final submitted version of your assignment will be marked.

You submit your work like this:

```
$ give dp1091 ass2_crepe_stand crepe_stand.c
```

Assessment Scheme

This assignment will contribute 25% to your final mark.

80% of the marks for this assignment will be based on the performance of the code you write in `crepe_stand.c` .

20% of the marks for this assignment will come from manual marking of the readability of the C you have written. The manual marking will involve checking your code for clarity, and readability, which includes the use of functions and efficient use of loops and if statements.

Marks for your performance will be allocated roughly according to the below scheme.

100% for Performance	Completely Working Implementation, which exactly follows the spec (Stage 1, 2, 3 and 4).
85% for Performance	Completely working implementation of Stage 1, 2 and 3.
65% for Performance	Completely working implementation of Stage 1 and Stage 2.
35% for Performance	Completely working implementation of Stage 1.

Marks for your style will be allocated roughly according to the scheme below.

Style Marking Rubric

	0	1	2	3
Formatting (/5)				
Indentation (/2) - Should use a consistent indentation scheme.	Multiple instances throughout code of inconsistent/bad indentation	Code is mostly correctly indented	Code is consistently indented throughout the program	
Whitespace (/1) - Should use consistent whitespace (for example, 3 + 3 not 3+ 3)	Many whitespace errors	No whitespace errors		
Vertical Whitespace (/1) - Should use consistent whitespace (for example, vertical whitespace between sections of code)	Code has no consideration for use of vertical whitespace	Code consistently uses reasonable vertical whitespace		
Line Length (/1) - Lines should be max. 80 characters long	Many lines over 80 characters	No lines over 80 characters		
Documentation (/5)				
Comments (incl. header comment) (/3) - Comments have been used throughout the code above code sections and functions to explain their purpose. A header comment (with name, zID and a program description) has been included	No comments provided throughout code	Few comments provided throughout code	Comments are provided as needed, but some details or explanations may be missing causing the code to be difficult to follow	Comments have been used throughout the code above code sections and functions to explain their purpose. A header comment (with name, zID and a program description) has been included
Function/variable/constant naming (/2) - Functions/variables/constants names all follow naming conventions in style guide and help in understanding the code	Functions/variables/constants names do not follow naming conventions in style guide and help in understanding the code	Functions/variables/constants names somewhat follow naming conventions in style guide and help in understanding the code	Functions/variables/constants names all follow naming conventions in style guide and help in understanding the code	
Organisation (/5)				

Function Usage (/4) - Code has been decomposed into appropriate functions separating functionalities	No functions are present, code is one main function	Some functions are present, but functions are all more than 50 lines	Some functions are present, and all functions are approximately 50 lines long	Most code has been moved to sensible/thoought out functions, but they are mostly more than 50 lines
Function Prototypes (/1) - Function Prototypes have been used to declare functions above main	Functions are used but have not been prototyped	All functions have a prototype above the main function or no functions are used		
Elegance (/5)				
Overdeep nesting (/2) - You should not have too many levels of nesting in your code (nesting which is 5 or more levels deep)	Many instances of overdeep nesting	<= 3 instances of overdeep nesting	No instances of overdeep nesting	
Code Repetition (/2) - Potential repetition of code has been dealt with via the use of functions or loops	Many instances of repeated code sections	<= 3 instances of repeated code sections	Potential repetition of code has been dealt with via the use of functions or loops	
Constant Usage (/1) - Any magic numbers are #defined	None of the constants used throughout program are #defined	All constants used are #defined and are used consistently in the code		
Illegal elements				
Illegal elements - Presence of illegal elements including: Global Variables, Static Variables, Labels or Goto Statements	**CAP MARK AT 16/20**			

Note that the following penalties apply to your total mark for plagiarism:

0 for the assignment	Knowingly providing your work to anyone and it is subsequently submitted (by anyone).
0 for the assignment	Submitting any other person's work. This includes joint work.
0 FL for DPST1091	Paying another person to complete work. Submitting another person's work without their consent.

If you choose to disregard this advice, you **must** still follow the style guide.

You also may be unable to get help from course staff if you use features not taught in DPST1091. Features that the Style Guide strongly discourages or bans will be penalised during marking. You can find the style marking rubric above. Please note that this assignment must be completed using only **Linked Lists** . Do not use arrays in this assignment.

Due Date

This assignment is due 19 April 2024 20:00:00. However, as per Sasha's announcement on the Ed forum the revised due date is **26 July 2024 20:00:00**. No short special considerations will be granted on top of this. For each day after that time, the maximum mark it can achieve will be reduced **by 5%** (off the ceiling).

- For instance, at **1 day past the due date**, the maximum mark you can get is **95%**.
- For instance, at **3 days past the due date**, the maximum mark you can get is **85%**.
- For instance, at **5 days past the due date**, the maximum mark you can get is **75%**.
  No submissions will be accepted after 5 days late, unless you have special provisions in place.