version: 1.2 last updated: 2024-07-19 21:00

Assignment 2 - CS Pet Salon!

Welcome to CS Pet Salon! Your task is to implement a pet salon manager involving several salons which have rooms for different types of pets. A pet salon is where pets get cared for and pampered. You will track the details of the pets being looked after in each salon!

Overview

COMP1911 Students

If you are a COMP1911 student, you will be assessed on your performance in **stages 1 and 2 ONLY** for this assignment, which will make up the performance component of your grade. Style will be marked as usual and will make up the remaining 20% of the marks for this assignment. You are NOT required to attempt stages 3 and 4. You may attempt the later stages if you wish, but you will not be awarded any marks for work beyond stage 2.

COMP1511 Students

If you are a COMP1511 student, or a COMP1911 student who filled out the COMP1511 assessment opt-in form, you will be assessed on your performance in **stages 1 through 4**, which will make up the performance component of your grade. Style will be marked as usual and will make up the remaining 20% of the marks for this assignment.

Assignment Structure

This assignment will test your ability to create, manipulate, and use linked lists to solve a variety of problems. To do this, you will be implementing a system to keep track of pet salons!

We have defined some structs in the provided code to get you started. You may modify any of the structs if you wish, but you should not need to.

`struct salon`

- Purpose:
  - To store all the information of a salon
- Fields:
  - `char salon_name[MAX_NAME_LEN]`
    - The name of the salon
  - `struct financial_summary summary`
    - A struct for all the finances relating to the salon
  - `double base_cost`
    - A double to keep track of the base cost of the salon
  - `struct pet_room *rooms`
    - A pointer to the start of the list of pet rooms
  - `struct salon *next`
    - A pointer to the next salon

`struct pet_room`

- Purpose:
  - To store all the information of a single room within the salon and the pets within the room

- Fields:
    - `char room_name[MAX_NAME_LEN]`
        - The name of the room
    - `enum pet_type pet_type`
        - The type of pet as an enum
        - Can be either `CAT` , `DOG` , `RABBIT` or `PARROT`
    - `int num_pets`
        - Number of pets to be cared for in a room
    - `struct pet_room *next`
        - The next room in the pet salon

`struct financial_summary`

- Purpose:
    - To store all the financial information of the salon
- Fields:
    - `int total_cared`
        - The number of pets that have been cared for
    - `double total_profit`
        - Total profit from the number of pets being cared for

Additionally, you can create your own enums if you would like, but you should not modify the provided `pet_type` enum.

HINT:

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

HINT:

As you complete the stages, keep in mind that 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.
- Uppercase characters (all commands will be inputted with lowercase characters). Additionally, commands will always start with a char.

Reference Implementation

To help you understand the proper behaviour of the CS Pet Salon, 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:

```
$ 1511 cs_pet_salon
```

How to get started

There are a few steps to getting started with CS Pet Salon.

1. Create a new folder for your assignment work and move into it.

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

2. Download cs_pet_salon.c here

   Or, copy these file(s) to your CSE account using the following command:

   ```
   $ 1511 fetch-activity cs_pet_salon
   ```

3. Run `1511 autotest cs_pet_salon` to make sure you have correctly downloaded the file.

```
$ 1511 autotest cs_pet_salon
```

WARNING:

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

4. Read through **Stage 1**.

About the starter code

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

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

It also contains two function stubs, `create_salon` and `create_room`, which you will need to complete in **Stage 1.1**.

Finally, the `main` function contains some comments to help guide you through **Stage 1.1**, as well as some `printf` messages which run when the program starts and ends.

These `printf` messages are:

```
Welcome to 1511 CS Pet Salon manager! =^.^=
All pet salons closed! =^.^=
```

HINT:

You can (and should!) write and add your own functions as you go.

Allowed C Features

In this assignment, **you cannot use arrays**, other than `char` arrays for strings, 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 you will need to get full marks in the assignment are:

- `int`, `char`, and `double` variables.
- Enums.
- Structs.
- If statements.
- While and for loops.
- Your own functions.
- Pointers.
- `char` arrays/strings (you are not allowed to use arrays that are not `char` arrays).
- Linked lists.
- Good Code Style!
  (Header comments, function comments, constants (`#define`'s), and whitespace and indentation.)

Using any other features will not increase your marks (and will make it more likely you make style mistakes that cost you marks).

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 COMP(1511|1911).

Features that the Style Guide labeled as illegal will be penalized during marking.

FAQ

+ FAQ

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.

> WARNING:
>
> COMP1911 students are only required to complete stages 1 and 2. COMP1511 students and COMP1911 opt-in students will be assessed on all four stages.

A video explanation to help you get started with the assignment can here found here:

COMP1511 Assignment 2 Livestream (24T2)

Stage 1 ●○○ Stage 2 ●○○ Stage 3 ●●○ Stage 4 ●●● Extension

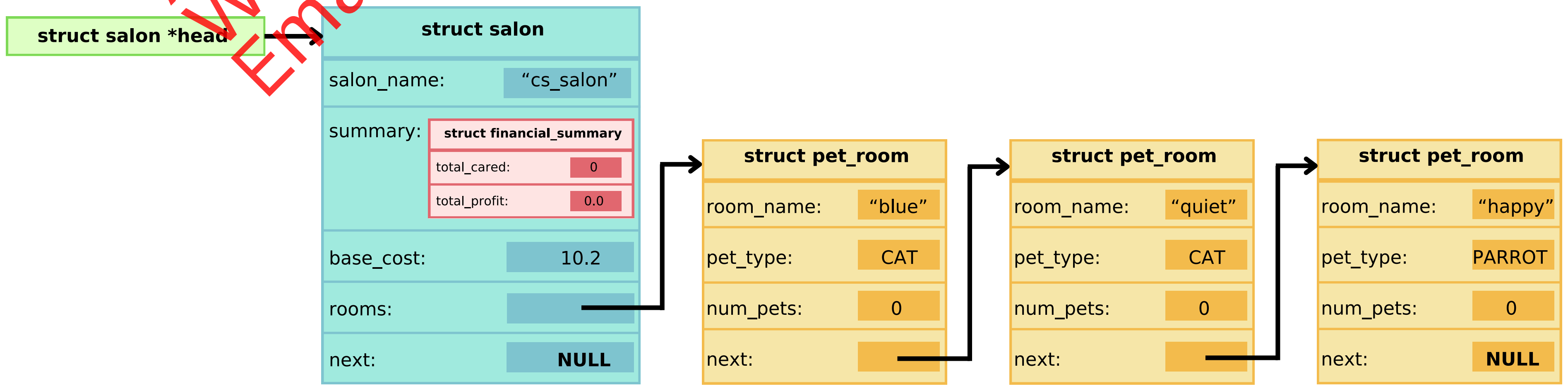
Stage 1

For **Stage 1** of this assignment, you will be implementing the command loop, as well as the commands to add pet rooms to a pet salon.

Specifically, this will include:

- Implementing the `create_salon` and `create_room` functions.
- Implementing the command loop, to scan commands until `CTRL-D` .
- Adding rooms to the end of the salon.
- Printing out all rooms in the salon.
- Handling errors.
- Inserting adjacent room.

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



Stage 1.1 - Creating a salon and room

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 salon` , and for a `struct pet_room` .

You'll find the following unimplemented functions in the starter code:

```
// TODO: what does this function do?
//
// Paramters:
//      TODO: explain what your parameters are here!
// Returns:
//      TODO: explain what your function returns here!
struct salon *create_salon(char salon_name[MAX_NAME_LEN], double base_cost) {

    // STAGE 1.1
    // TODO: malloc, initialise, and return a new salon.

    // hint: you will have to replace NULL in this return statement.

    return NULL;
}

// TODO: what does this function do?
//
// Paramters:
//      TODO: explain what your parameters are here!
// Returns:
//      TODO: explain what your function returns here!
struct pet_room *create_room(char room_name[MAX_NAME_LEN], enum pet_type pet_type) {

    // STAGE 1.1
    // TODO: malloc, initialise, and return a new room.

    // hint: you will have to replace NULL in this return statement.

    return NULL;
}
```

NOTE:

We would call both of these function "stubs", because they are just the basic structure of the `create_salon` and `create_room` functions, but without any real code inside.

You'll be writing code inside these function stubs to make them work.

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

1. Malloc's a new `struct salon` .
2. Copies the `salon_name` and `base_cost` into the corresponding struct field.
3. Initialise the nested `struct financial_summary` such that `total_cared` and `total_profit` are zero.
4. Assign `next` and `rooms` to `NULL` .
5. Returns a pointer to the malloc'd struct.

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

1. Malloc's a new `struct pet_room` .
2. Copies the `room_name` and `pet_type` into the corresponding struct fields.
3. Initialises all other fields to some either zero equivalent or `NULL` .
4. Returns a pointer to the malloc'd struct.

HINT:

`salon_name` is an array of chars (a string), so `new_salon->salon_name = salon_name` won't work!

You may need to look up the function `strcpy` . This will also be useful when completing the `create_room` function below.

Keep in mind that this also applies to `room_name` .

Error Conditions

- There is no error handling required for this stage. You'll be adding this later in **Stage 1.4 - Handle Errors**.

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. If you manually tested `./cs_pet_salon` , it would only print the lines in main.

As you can tell, this does not test the functions you just implemented. You could also write some temporary testing code to check your `create_salon` and `create_room` functions work properly.

For example, you could copy the following testing code *into your main function*:

```
//////////////////////////// TESTING CODE ////////////////////////////

    // name of pet room
    char name[MAX_NAME_LEN] = "blue";

    // create a struct room with
    //       room_name    : "blue"
    //       pet_type     : CAT
    struct pet_room *test_room = create_room(name, CAT);

    // print out all of its fields.
    printf("room name: %s\n", test_room->room_name);
    printf("number of pets: %d\n", test_room->num_pets);

    if (test_room->pet_type == CAT) {
        printf("pet type: cat\n");
    } else {
        printf("pet type: not a cat\n");
    }

    if (test_room->next == NULL) {
        printf("next field: NULL");
    }

//////////////////////////// TESTING CODE ////////////////////////////
```

This code just calls `create_room` to malloc and initialise a `struct pet_room` , and then prints out all of its fields.

If you run it, it should print out something like:

```
room name: blue
number of pets: 0
pet type: cat
next field: NULL
```

WARNING:

Don't forget to delete any testing code you wrote before moving on to the next stage!

Stage 1.2 - Add room

Now we'll be implementing the command loop, allowing your program to take in and perform different operations on the salon.

From this stage onwards, your program should run in a loop, scanning in and executing commands until `CTRL-D` . You should implement this command loop between the existing welcome and goodbye messages in the starter code.

On each iteration of the loop, your program should:

1. Print the prompt `Enter command: ` .
2. Scan in a command character.
3. Scan in any arguments following the command character and execute the command.

Each command will start with a single unique character, and may be followed by a variable number of arguments depending on the command.
The unique character for each different command and the number and type of the command arguments are specified in the relevant stages of this assignment.

HINT:

This command loop is *extremely* similar to:

- the cs_calculator lab exercise in week 4
- the command loop in stage 1.1 in assignment 1

HINT:

Make sure to put a space before the `%c` in `scanf(" %c", ...)` , so that it ignores all the preceding whitespace characters.

Also make sure to use `while (return_val == 1)` or any other method you have learnt from the lectures. Avoid using `EOF` or anything else that has not been taught.

The first command you have to implement is the **ADD ROOMS** command, which is described below.

When you run your program, a new salon should be created in main with the name `"cs_salon"` and base cost of `10.2` . The name has been already given to you in main with the variable `salon_name` so you can pass it through the `create_salon` function.

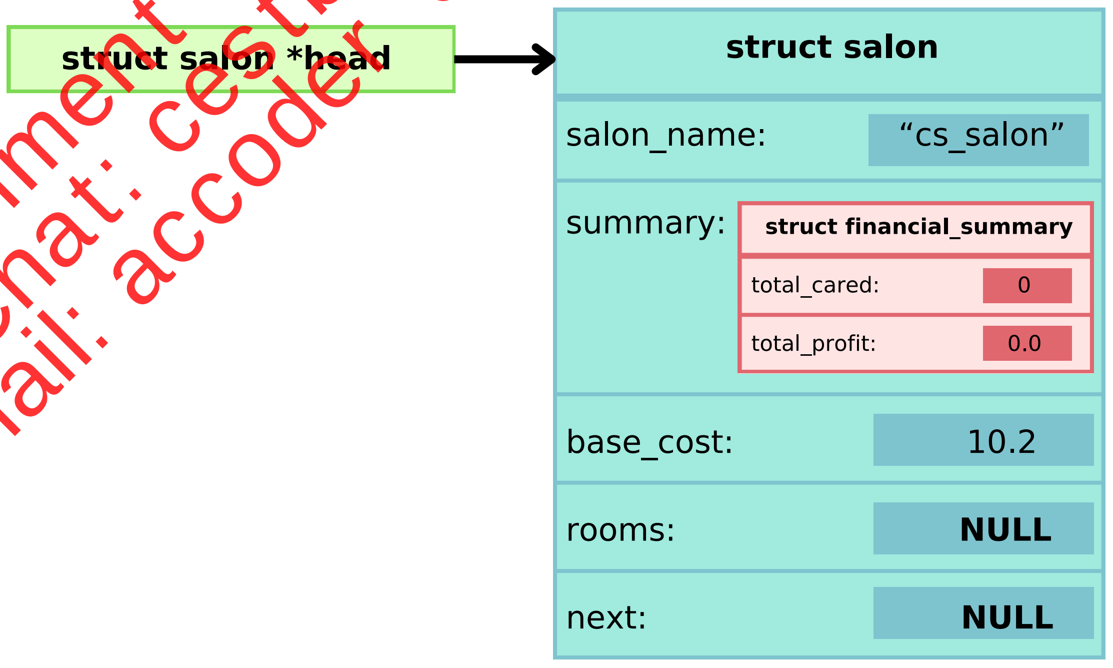
HINT:

It would be a good idea to add the following lines in `main` above your command loop:

```
char salon_name[MAX_NAME_LEN] = "cs_salon";
struct salon *head = create_salon(salon_name, 10.2);
```

This salon will start out empty - with no rooms.

It should look something like this:



To make your salon more useful, we need a way of adding a room to the end of the list of rooms.

Command: Add room

```
a [room_name] [pet_type]
```

Description

The `a` command takes in 2 arguments:

- a string called `room_name` ,

- an `enum pet_type` called `pet_type` .

Some helper functions have been provided to help you scan in these arguments:

- `void scan_name(char string[MAX_NAME_LEN])`
- `enum pet_type scan_pet_type()`

You can find more information on these here: [Provided helper functions](#).

> WARNING:
>
> Make sure to use these provided helper functions rather than reading in the inputs yourself!

When the `a` command is entered, your program should create a new room containing the `room_name` , `pet_type` , and `num_pets` , then **append** it to the end of the list of rooms inside the salon (in other words, insert the new room at the end of the salon's `rooms` linked list).

Finally, it should out a message to confirm the command was successful:

`"Room: '[room_name]' added!\n"`

You should replace `[room_name]` with the `room_name` you scanned in.

For example, if we have just started the program and we use the `a` command once, it would look like:

```
Welcome to 1511 CS Pet Salon manager! =^.^=
Enter command: a blue cat
Room: 'blue' added!
```

Our linked list should now look like this:

A salon with one room.

If we then run the `a` command again:

```
Enter command: a happy dog
Room: 'happy' added!
```

then our linked list should now look like:

A salon with two rooms.

Provided helper functions

Two helper functions have been provided for this stage: `void scan_name(char string[MAX_NAME_LEN])` :

- `char string[MAX_NAME_LEN]` : used to scan the `room_name` .

`enum pet_type scan_pet_type()` :

- Scans the `pet_type` and returns it as an `enum pet_type` .
- Returns `INVALID_PET_TYPE` if the `pet_type` did not correspond to one of the valid pet types.

You should use these functions to help you scan in the `room_name` and `pet_type` arguments.

Remember that the arguments must be scanned in the correct order, so your code to scan arguments will look something like the following:

```c
// Create variables to scan arguments into
char room_name[MAX_NAME_LEN];
enum pet_type type;

// Arguments are in order: [room_name] [pet_type]

// 1. Scan room_name first
scan_name(room_name);
// 2. Then scan the pet_type
type = scan_pet_type();

// We've scanned in all the arguments! Now we can use them in our code
```

Error Conditions

- There is no error handling required for this stage. You'll be adding this later in **Stage 1.4 - Handle Errors**.

Assumptions

- `salon_name` will always be less than `MAX_NAME_LEN` including their null terminator at the end.
- `salon_name` will not contain any whitespace. For example `my_salon` is a valid title but `my salon` is not. We will not test any invalid `salon_name` or `room_name` inputs so you do not need to account for this in your program.
- `salon_name` will not contain any quotations. For example `bobs` is a valid title but `bob's` is not. We will not test any invalid `salon_name` or `room_name` inputs so you do not need to account for this in your program.
- All input will be in lowercase. Uppercase characters will not be tested.
- `pet_type` will be entered as a lowercase string and automatically converted to the correct `enum pet_type` for you by the `scan_pet_type` function, when the function is used.

Corresponding string	**enum pet_type**
"cat"	`CAT`
"dog"	`DOG`
"rabbit"	`RABBIT`
"parrot"	`PARROT`
any invalid string	`INVALID_PET_TYPE`

Note that you don't need to worry about `INVALID_PET_TYPE` until **Stage 1.4**.
Until then, you can assume that the returned `enum pet_type` will never be `INVALID_PET_TYPE` .

Examples

+ Example 1.2.1: Add one room

+ Example 1.2.2: Add many rooms

NOTE:

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

```
1511 autotest-stage 01_02 cs_pet_salon
```

Stage 1.3 - Printing rooms in the salon

Now we want a way to display the salon and all its rooms.

Command: Print rooms

```
p
```

Description

The `p` command takes no arguments.

When the `p` command is run, your program should print out all rooms in the current salon, from head to tail.

The `print_one_room` function has been provided for you to format and print a single room. More information on its usage can be found below. After all the rooms are printed, the following line should be printed:

```
All the rooms listed above are in salon '[salon_name]'.
```

where `[salon_name]` is the name of salon containing the rooms being printed.

If there are no rooms in the salon, you should print the following message instead:

```
There are no pet rooms in this salon!
```

Provided helper functions

One helper function has been provided for this stage:

```
void print_one_room(int position, struct pet_room *room)
```

- `int position` : the position of where the room is in the salon. (In other words, if this room was the first in the linked list, the position would be 1).
- `struct pet_room *room` : a pointer to a pet room.

This function will print out the required information for the pet room in the correct format. Your job is to determine the correct values to pass to this function.

This means you don't need to worry about copying the exact output format for the `p` command. To match the autotests exactly, you should loop through the list of pet rooms and call this function for each of them.

Error Conditions

- There is no error handling required for this stage. You'll be adding this later in **Stage 1.4 - Handle Errors**.

Examples

+ Example 1.3.1: Print an empty salon

+ Example 1.3.2: Print a salon with several rooms

WARNING:

If you are failing autotests in this stage, it might be because of your implementation of stage 1.2.

NOTE:

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

```
1511 autotest-stage 01_03 cs_pet_salon
```

Stage 1.4 - Handle Errors

Once you've reached this stage, you should be able to add rooms to your salon and print them! Nice work!

However, the people using your salon system can still make mistakes! Let's consider this example:

```
Welcome to 1511 CS Pet Salon manager! =^.^=
Enter command: a blue cat
Room: 'blue' added!
Enter command: a blue cat
Room: 'blue' added!
Enter command: p
/--------------------------------\
Room name: blue
    Room position: 1
    Pet type: cat
    Num of pets in room: 0/10
\-------------- | --------------/
                V
/--------------------------------\
Room name: blue
    Room position: 2
    Pet type: cat
    Num of pets in room: 0/10
\-------------- | --------------/
                V
All the rooms listed above are in salon 'cs_salon'.
Enter command: Ctrl-D
All pet salons closed! =^.^=
```

It does not make sense to have two rooms with the same name - this can be confusing for the owner! We need to ensure that the room names are unique. Also, there are only a few types of pets that the salon can look after, so we want to make sure that the input for the type of pet is actually valid for the salon.

In this stage you will be modifying your code from **Stage 1.3** so we make sure only valid rooms can be added to your salon.

Error Conditions

When running the `a` command, you scanned in the `room_name` and `pet_type` for the new room.

DANGER:

You should still always scan in *every* argument for a command.

Once you've scanned them all in, then you can start to check for errors.

If any one of the following conditions are met, then you should **not** append a new room to the linked list. You should **instead** print out an error message:

- If there is already a room in the linked list that contains the same `room_name`, the following error should be printed: `Error: This room name already exists!`
- If `pet_type` (returned by the `scan_pet_type` function) is `INVALID_PET_TYPE`, the following error should be printed: `Error: Unfortunately, this salon does not cater for this pet type!`

Using the `strcmp` function in `string.h` might be useful for this step!

HINT:

Some advice for Stage 2.1: Splitting up the work

A lot of the commands you'll be implementing will first have to scan in a number of arguments, then check that those arguments are valid, and *only* then run the actual command, if those arguments were all valid.

One possible way of splitting up the logic would be to make a function that takes in the inputs for a new room after they have been scanned in and does the error checking. You might also have a function for handling the adding room command, and *inside of this function* you can call your error checking function to check a name is valid before it is added.

This is not the only approach to structuring your code.
So if you have other ideas that you think are better, feel free to go with those instead!

Clarifications

- If more than one error occurs, only the first error should be addressed by printing an error message. For example, if there is already a room name that exists and the pet type does not exist - only the room name error should be printed out.
- As before, you can assume there is no whitespace within a `room_name` and do not need to do any error handling for this.
- As before, uppercase characters are not being tested.
- This is the same for all future commands with error checking.

Examples

+ Example 1.4.1: Attempt to add room name twice

+ Example 1.4.2: Invalid pet_type and room_name

NOTE:

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

```
1511 autotest-stage 01_04 cs_pet_salon
```

Stage 1.5 - Insert adjacent room

We want to make sure all pets are comfortable at their stay at the salon. We noticed that putting some pets adjacent to pets of the same type helps with this! For example when making a new room and we already see that there's a room existing for cats, we want to place this new room next to that existing room. This is similar to **Stage 1.2 - Add room**, however position matters for this command.

Command: Insert adjacent room

```
i [room_name] [pet_type]
```

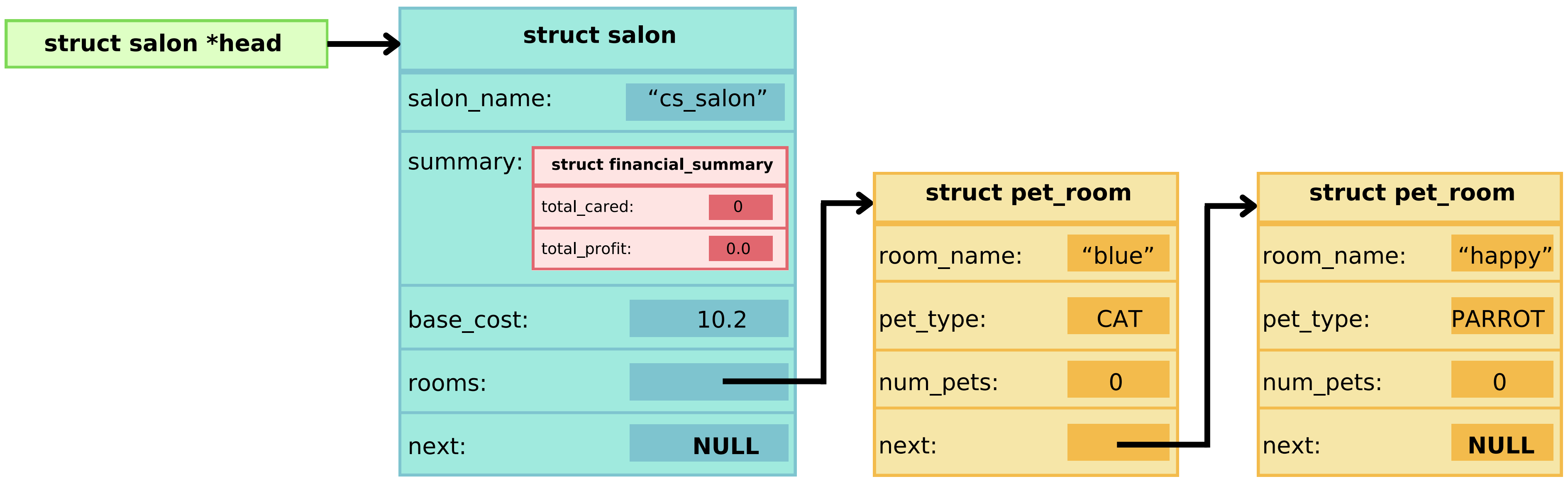
Description

The `i` command is similar to the `a` command, except the position of the room matters.

It should create a new `struct pet_room` containing `room_name` , `pet_type` and `num_pets` . If a room exists of the same `pet_type` , then the new room should be inserted after that existing room. If the same pet_type is not found, then the room should be inserted at the end.

For example if we ran the following commands:

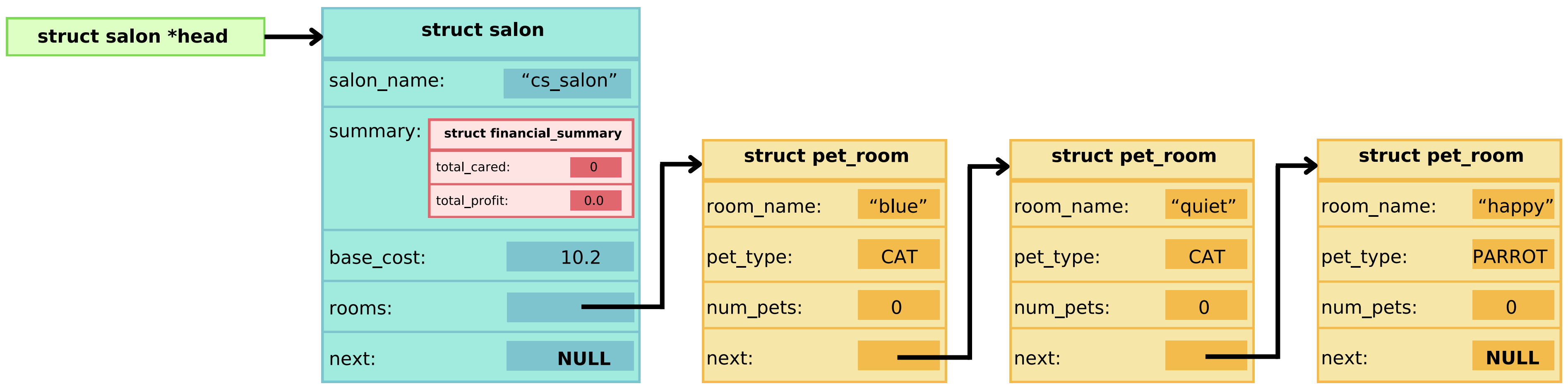
- `a blue cat`
- `a happy parrot` resulting in the following.



A salon with rooms

Then used our new command:

- `i quiet cat` the room would be inserted after the same pet_type is found as seen below.



A salon after using the insert command

After successful insertion, your program should print the following message:

`Room: '[room_name]' inserted!`

Provided helper functions

See **Stage 1.2 - Add room** for information on how to scan the `room_name` and `pet_type` .

Error Conditions

Just like command `a` : **add room**, there are restrictions on the rooms that can be inserted into the salon. Specifically, if any of the previous mentioned error conditions are met, then the room should not be inserted, and an error message should be printed out instead. Please refer to **Error Conditions in Stage 1.4 - Handle Errors** for more information.

Examples

+ Example 1.5.1: Insert adjacent room

+ Example 1.5.2: Insert pet_type when pet_type exists

+ Example 1.5.3: Insert several of same pet_type

+ Example 1.5.4: Insert errors

NOTE:

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

```
1511 autotest-stage 01_05 cs_pet_salon
```

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 `1511 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?

```
$ 1511 style cs_pet_salon.c
$ 1511 autotest-stage 01 cs_pet_salon
$ give cs1511 ass2_cs_pet_salon cs_pet_salon.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 COMP1511.

  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 COMP1511. 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 COMP1511** is **not permitted**.

  For example, do not place your assignment in a public GitHub repository after COMP1511 is over.

  Rationale:COMP1511 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 COMP1511 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 COMP1511	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_cs_pet_salon` .

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 cs1511 ass2_cs_pet_salon cs_pet_salon.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 `cs_pet_salon.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.

COMP1911

100% for Performance	Completely working implementation of Stage 1 and Stage 2.
55% for Performance	Completely working implementation of Stage 1.

COMP1511

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**	**4**
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/thought out functions, but they are mostly more than 50 lines	A b m d in fu a li
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 any illegal elements indicated in the style guide	**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 COMP1511	Paying another person to complete work. Submitting another person's work without their consent.

Allowed C Features

In this assignment, there are no restrictions on C Features, except for those in the style guide. 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 COMP1511. Features that the Style Guide identifies as illegal will result in a penalty 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 **02 August 2024 20:00:00**. 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.

Change Log

Version 1.0
(2024-07-12 10:00)
- Assignment Released

Version 1.1
(2024-07-12 15:00)
- Correct typo in 2.1 error condition for when 0 or less pets are added.

Version 1.2
(2024-07-19 21:00)
- Correct typo in 1.2 code snippet to refer to correct provided scan_name function.