University of Liberal Arts Bangladesh

(Open Ended Lab Report)

Course Code	**:**	**CSE1302**
Course Title	**:**	**Data Structures Lab**
Section	**:**	**03**
Submitted to	**:**	**Shakib Mahmud Dipto**
Submitted by	**:**	**Kawser Ahmmed**
ID	**:**	**231014036**

Problem:
Imagine you are developing a task management application, and you decide to use a linked list to implement the list of tasks. Each task has **a title, a description, a due date, and a status** (e.g., incomplete, complete). The linked list nodes store the task information, and the linked list is used to organize and manage the tasks.

Given this scenario, consider the following operations you need to perform on the linked list:

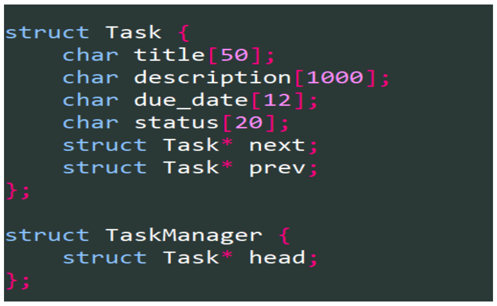
1. Add a New Task
2. Search for a Task
3. Remove Expired Tasks
4. Update Task Status
5. Display All Tasks
6. Handling Priority Tasks **[Bonus]**

Data structure

To develop this task management application, I used a doubly linked list. I think a doubly linked list is a suitable data structure for this application due to its efficient support for frequent insertions and deletions, bidirectional traversal facilitating easy backtracking and modifications, memory efficiency in a dynamic environment, and the ability to seamlessly handle priority tasks through effective sorting based on due date or status.

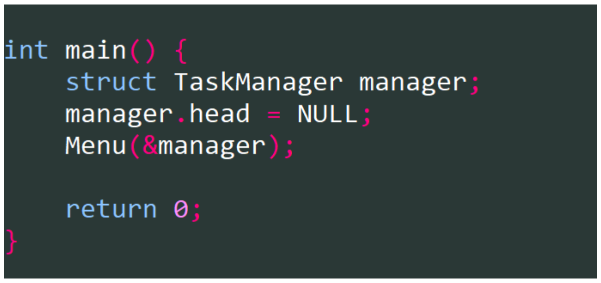
Structure and main()

- I used struct TaskManager to keep things organized. It's like a box that holds our list of tasks (struct Task* head). This helps to manage tasks without getting into the messy details of how the list works. It makes the code easier to read and allows us to add new features without messing up the whole program. So, it's like having a neat container for our tasks that makes everything simpler.by using this struct TaskManager I got another benefits function don't need return sometihg . fuction will be void();



```
struct Task {
    char title[50];
    char description[1000];
    char due_date[12];
    char status[20];
    struct Task* next;
    struct Task* prev;
};

struct TaskManager {
    struct Task* head;
};
```

In main () function I declare a (structure TaskManager type) structure that will hold the linked list address and then I have send that structure address to Menu() function to perform operation.



```
int main() {
    struct TaskManager manager;
    manager.head = NULL;
    Menu(&manager);

    return 0;
}
```

All Function ()

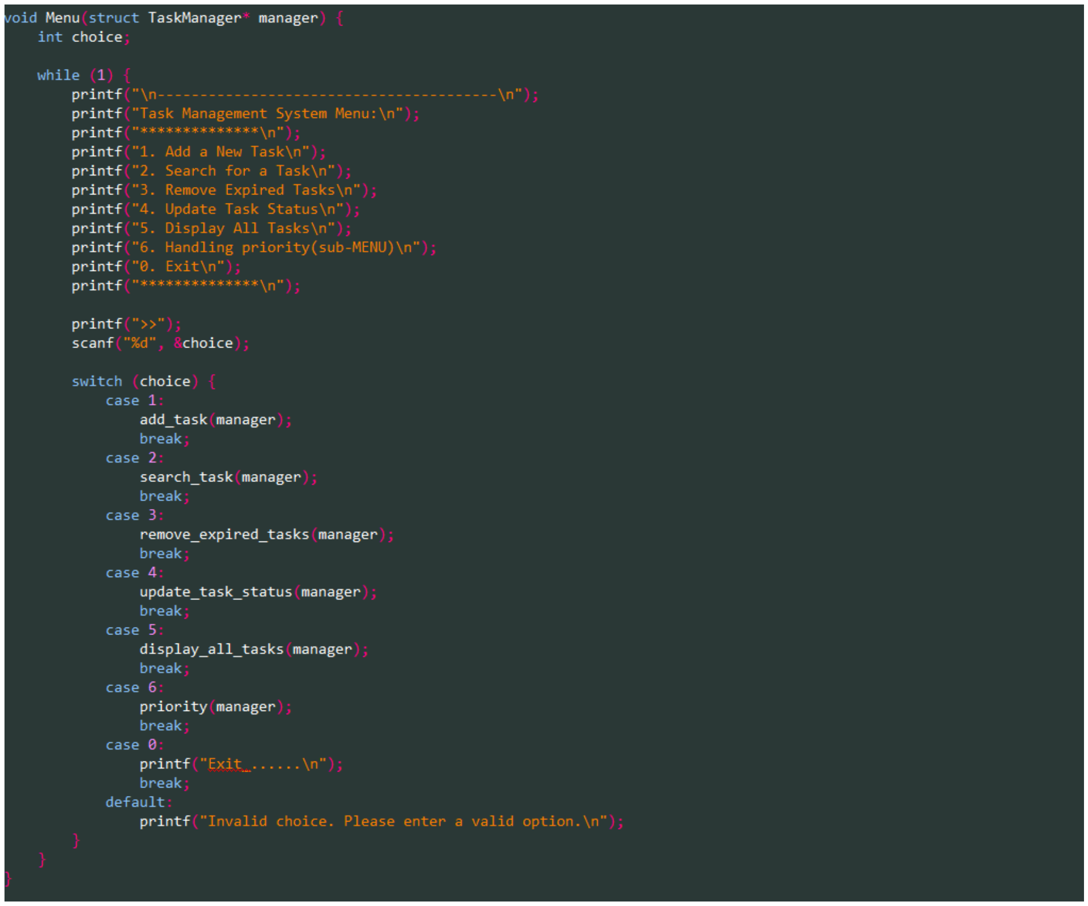
I have write down all the function I have used in my program:

- **Menu function:**
  - void Menu(struct TaskManager* manager);
- **Add task function:**
  - void add_task(struct TaskManager* manager);
- **Search task function:**
  - void search_task(struct TaskManager* manager);
- **Remove task function:**
  - void remove_expired_tasks(struct TaskManager* manager);
- **Update task function:**
  - void update_task_status(struct TaskManager* manager);
- **priority task function:**
  - void priority(struct TaskManager* manager);
- **Duedate sort function:**
  - void duedate_sort(struct TaskManager* manager);
- **By task sort:**
  - void status_sort(struct TaskManager* manager);

Menu function

First of al this menu function will revieve address of structure type data:

Menu Display: The function starts by displaying a menu for a task management system using printf statements. Each option is numbered and corresponds to a specific operation.



```c
void Menu(struct TaskManager* manager) {
    int choice;

    while (1) {
        printf("\n----------------------------------------\n");
        printf("Task Management System Menu:\n");
        printf("**************\n");
        printf("1. Add a New Task\n");
        printf("2. Search for a Task\n");
        printf("3. Remove Expired Tasks\n");
        printf("4. Update Task Status\n");
        printf("5. Display All Tasks\n");
        printf("6. Handling priority(sub-MENU)\n");
        printf("0. Exit\n");
        printf("**************\n");

        printf(">>");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                add_task(manager);
                break;
            case 2:
                search_task(manager);
                break;
            case 3:
                remove_expired_tasks(manager);
                break;
            case 4:
                update_task_status(manager);
                break;
            case 5:
                display_all_tasks(manager);
                break;
            case 6:
                priority(manager);
                break;
            case 0:
                printf("Exit_......\n");
                break;
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }
}
```

User Input: The program then prompts the user to enter a choice (scanf("%d", &choice)), and the user's input is stored in the variable choice.

Switch Statement: A switch statement is used to check the value of choice and determine the corresponding action to take.

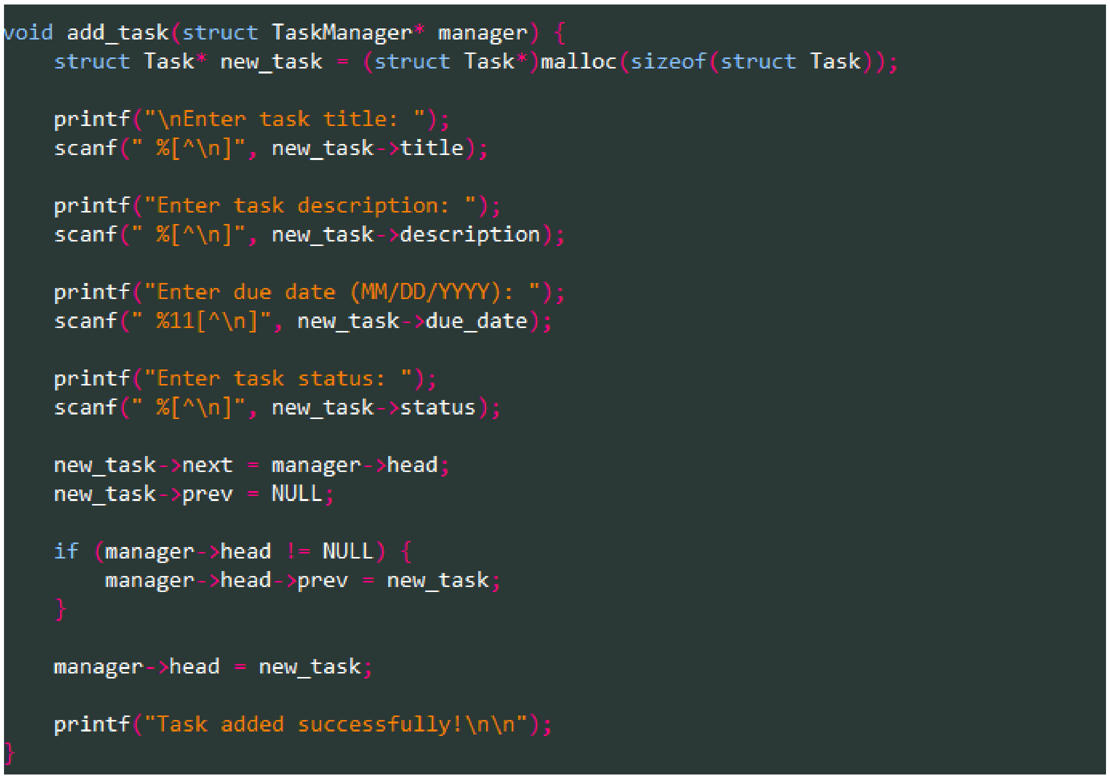
Handling User Choices: For each possible value of choice, there is a corresponding case statement. Depending on the user's input, the program calls different functions (add_task, search_task, etc.) to perform specific tasks.
Infinite Loop: The while (1) loop ensures that the menu keeps displaying, and the user can keep choosing operations until they decide to exit by entering '0'.If the user enters an invalid choice, the program provides feedback by printing an error message and asking the user to enter a valid option.

<u>Add task function</u>

This function also recieve a structure to operation in this structe:

<u>Memory Allocation:</u>

The function begins by dynamically allocating memory for a new task using malloc.This ensures that each task has its own space in memory.



```
void add_task(struct TaskManager* manager) {
    struct Task* new_task = (struct Task*)malloc(sizeof(struct Task));

    printf("\nEnter task title: ");
    scanf(" %[^\n]", new_task->title);

    printf("Enter task description: ");
    scanf(" %[^\n]", new_task->description);

    printf("Enter due date (MM/DD/YYYY): ");
    scanf(" %11[^\n]", new_task->due_date);

    printf("Enter task status: ");
    scanf(" %[^\n]", new_task->status);

    new_task->next = manager->head;
    new_task->prev = NULL;

    if (manager->head != NULL) {
        manager->head->prev = new_task;
    }

    manager->head = new_task;

    printf("Task added successfully!\n\n");
}
```

User Input:

Special care is taken to handle spaces in the input strings using formatted scanf statements.

Task Linking:

The new task is linked to the existing list. It is set as the new head (next points to the current head, and prev is set to NULL as it becomes the head).

Update Previous Head:

If there was an existing head (meaning the list was not empty), the previous head's prev pointer is updated to point to the new task.

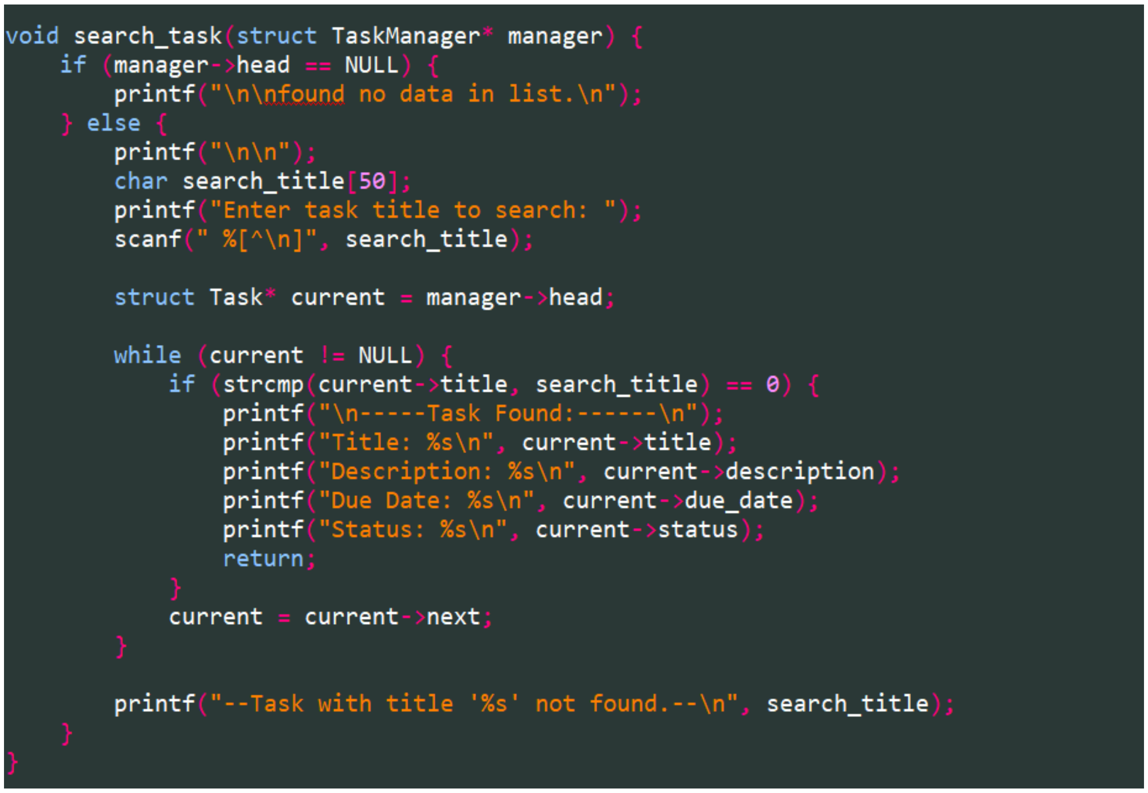
Update Head Pointer:

- The head pointer of the task manager is updated to point to the new task, effectively making it the new head of the list.

- Finally, a success message is printed to notify the user that the task has been added successfully.

Search task function

The search_task function will search for a specific task within the task manager's list. The function begins by checking whether the list is empty, and if so, it informs the user that there is no data to search. In the case where the list contains tasks, the user is prompted to input the task title they are looking for



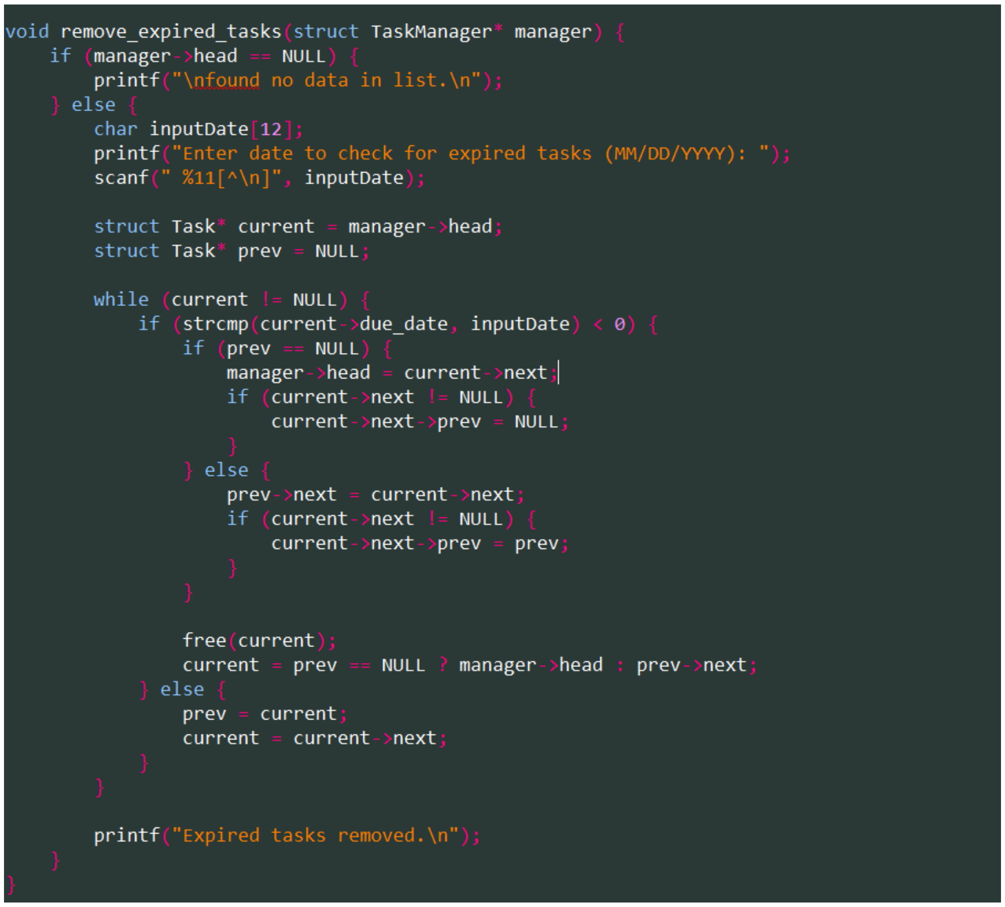
```c
void search_task(struct TaskManager* manager) {
    if (manager->head == NULL) {
        printf("\n\nfound no data in list.\n");
    } else {
        printf("\n\n");
        char search_title[50];
        printf("Enter task title to search: ");
        scanf(" %[^\n]", search_title);

        struct Task* current = manager->head;

        while (current != NULL) {
            if (strcmp(current->title, search_title) == 0) {
                printf("\n-----Task Found:------\n");
                printf("Title: %s\n", current->title);
                printf("Description: %s\n", current->description);
                printf("Due Date: %s\n", current->due_date);
                printf("Status: %s\n", current->status);
                return;
            }
            current = current->next;
        }

        printf("--Task with title '%s' not found.--\n", search_title);
    }
}
```

.

The function then performs a linear search through the linked list, comparing each task's title with the user-inputted title using the strcmp function. If a task with a matching title is found, the function prints detailed information about that task, including its title, description, due date, and status. This provides the user with a clear presentation of the task's details.
On the other hand, if the search loop completes without finding a match, the function communicates this to the user by printing a message stating that the task with the specified title was not found. This ensures that the user is aware when their search did not yield .

Remove task function

The remove_expired_tasks function serves the purpose of removing tasks from the task manager's list that have exceeded their due dates. Initially, the function checks whether the list is empty, delivering a message to the user if no data is present. Subsequently, the user is prompted to input a date (in MM/DD/YYYY format) for identifying expired tasks. The function then iterates through the linked list, comparing the due date of each task with the user-inputted date.



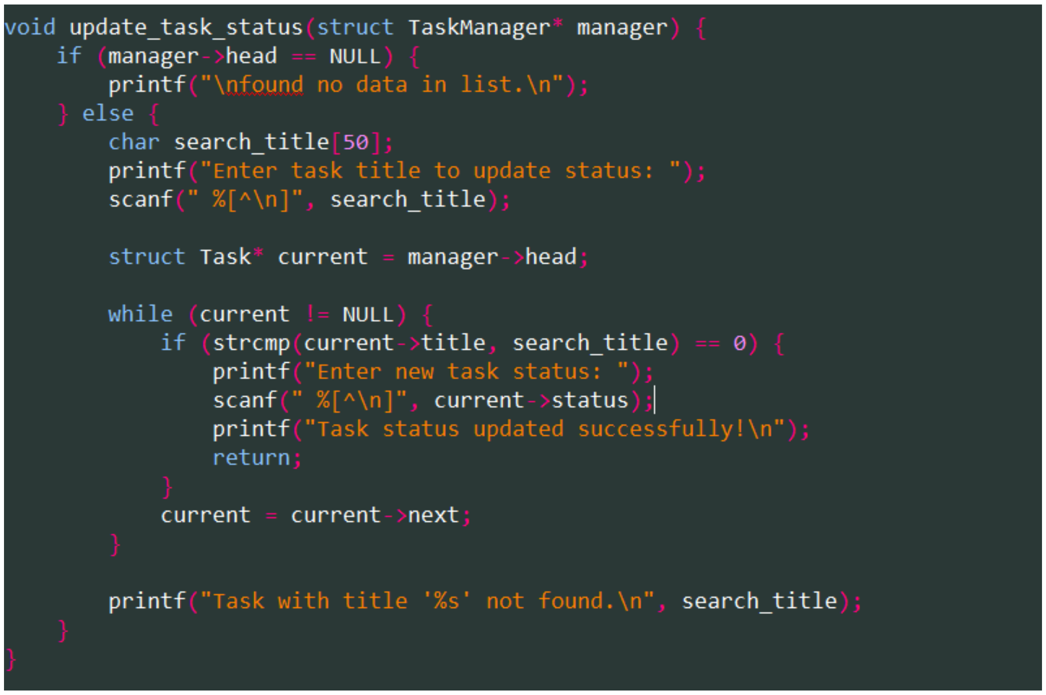
```c
void remove_expired_tasks(struct TaskManager* manager) {
    if (manager->head == NULL) {
        printf("\nfound no data in list.\n");
    } else {
        char inputDate[12];
        printf("Enter date to check for expired tasks (MM/DD/YYYY): ");
        scanf(" %11[^\n]", inputDate);

        struct Task* current = manager->head;
        struct Task* prev = NULL;

        while (current != NULL) {
            if (strcmp(current->due_date, inputDate) < 0) {
                if (prev == NULL) {
                    manager->head = current->next;
                    if (current->next != NULL) {
                        current->next->prev = NULL;
                    }
                } else {
                    prev->next = current->next;
                    if (current->next != NULL) {
                        current->next->prev = prev;
                    }
                }

                free(current);
                current = prev == NULL ? manager->head : prev->next;
            } else {
                prev = current;
                current = current->next;
            }
        }

        printf("Expired tasks removed.\n");
    }
}
```

For tasks that are found to have expired, the function efficiently removes them from the list. This removal process involves adjusting pointers to bypass the expired task and releasing the memory occupied by the task using free. Notably, the function handles both cases where the expired task is the head of the list and situations where it is located elsewhere within the list, ensuring that the linked list structure remains intact.

Following the removal of expired tasks, the function provides feedback to the user, indicating that the removal process has been successfully executed.

Update task function

The update_task_status function facilitates the modification of a task's status within the task manager's list. To begin, the function checks whether the list is empty (manager->head == NULL). In case of an empty list, it informs the user that there is no data to work with. In situations where the list is not empty, the user is prompted to input the title of the task for which they want to update the status.



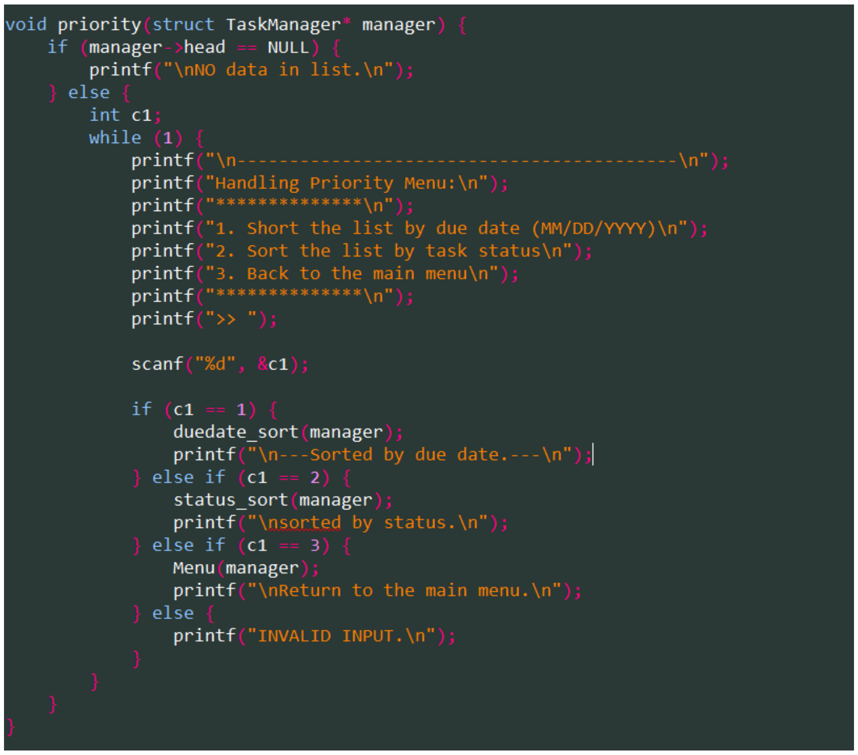
```
void update_task_status(struct TaskManager* manager) {
    if (manager->head == NULL) {
        printf("\nfound no data in list.\n");
    } else {
        char search_title[50];
        printf("Enter task title to update status: ");
        scanf(" %[^\n]", search_title);

        struct Task* current = manager->head;

        while (current != NULL) {
            if (strcmp(current->title, search_title) == 0) {
                printf("Enter new task status: ");
                scanf(" %[^\n]", current->status);
                printf("Task status updated successfully!\n");
                return;
            }
            current = current->next;
        }

        printf("Task with title '%s' not found.\n", search_title);
    }
}
```

Utilizing a while loop, the function iterates through the linked list, searching for a task with a title matching the user-inputted title. Upon finding the matching task, the function prompts the user to enter the new task status. Subsequently, it updates the task's status with the user-provided information and prints a success message, indicating that the task status has been updated.

If the loop completes without finding a matching title, the function communicates this to the user, specifying that a task with the provided title was not found.

priority task function

The priority function introduces a dynamic and interactive priority handling menu to the task management system, enabling users to efficiently manage and sort tasks based on specific criteria. The function initiates by checking if the task manager's list is empty, ensuring that users are informed if there is no data to prioritize. Assuming the list contains tasks, the function enters an infinite loop where users are presented with a priority handling menu. This menu offers choices to sort the list by due date (Option 1), sort the list by task status (Option 2), or return to the main menu (Option 3).



```c
void priority(struct TaskManager* manager) {
    if (manager->head == NULL) {
        printf("\nNO data in list.\n");
    } else {
        int c1;
        while (1) {
            printf("\n------------------------------------------\n");
            printf("Handling Priority Menu:\n");
            printf("**************\n");
            printf("1. Short the list by due date (MM/DD/YYYY)\n");
            printf("2. Sort the list by task status\n");
            printf("3. Back to the main menu\n");
            printf("**************\n");
            printf(">> ");

            scanf("%d", &c1);

            if (c1 == 1) {
                duedate_sort(manager);
                printf("\n---Sorted by due date.---\n");
            } else if (c1 == 2) {
                status_sort(manager);
                printf("\nsorted by status.\n");
            } else if (c1 == 3) {
                Menu(manager);
                printf("\nReturn to the main menu.\n");
            } else {
                printf("INVALID INPUT.\n");
            }
        }
    }
}
```

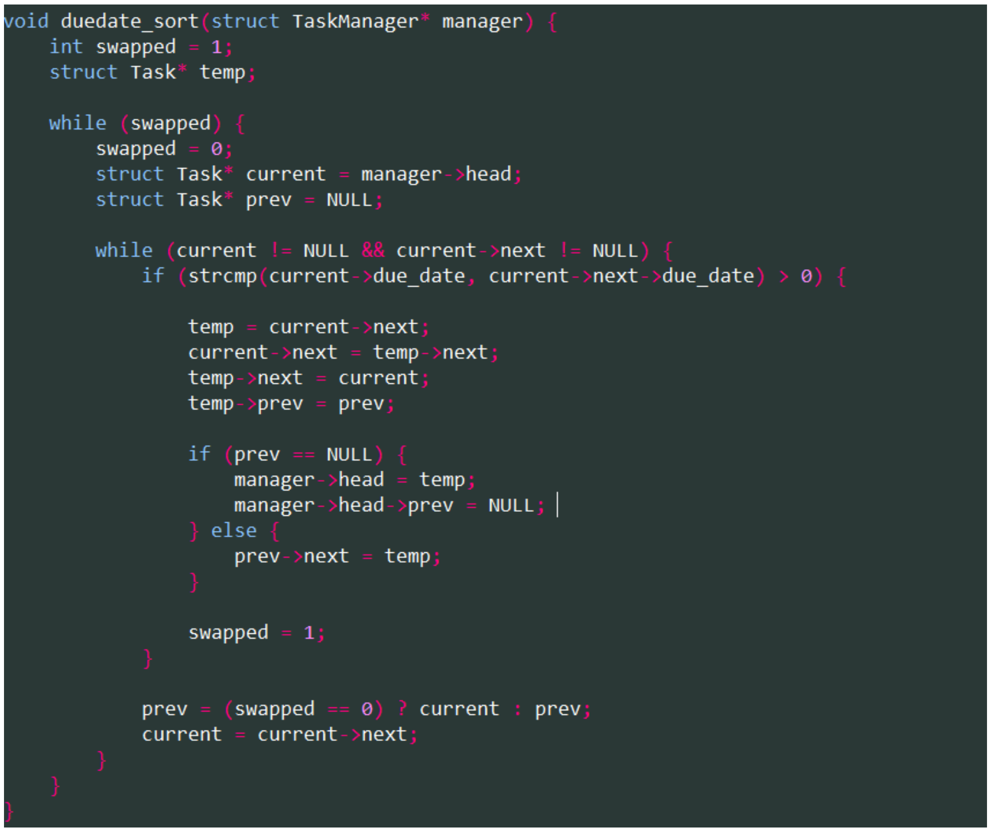
Upon receiving the user's input, the function executes the corresponding action. If the user chooses to sort by due date, the duedate_sort function is called, and a success message is displayed. Similarly, if the user opts for sorting by task status, the status_sort function is invoked, and a confirmation message is printed. Alternatively, if the user selects Option 3, the function returns to the main menu using the Menu function, with an accompanying message indicating the return.

The function ensures robust error handling by providing feedback in the case of invalid user input, printing a clear message to notify the user of the input's invalidity.

<u>Due date sort function</u>

The duedate_sort function implements a sorting algorithm to arrange tasks within the task manager's list based on their due dates. This function employs the Bubble Sort algorithm, a straightforward comparison-based sorting method. The primary objective is to compare adjacent tasks in the linked list based on their due dates and swap them if they are in the wrong order. The sorting process continues until no more swaps are needed, signifying that the list is now in ascending order by due date.

Within the function, a flag, swapped, is initialized to 1, initiating the first pass through the list. A temporary task pointer, temp, is also declared to facilitate task swapping. The sorting process takes place in a nested loop, traversing the linked list and comparing adjacent tasks' due dates.

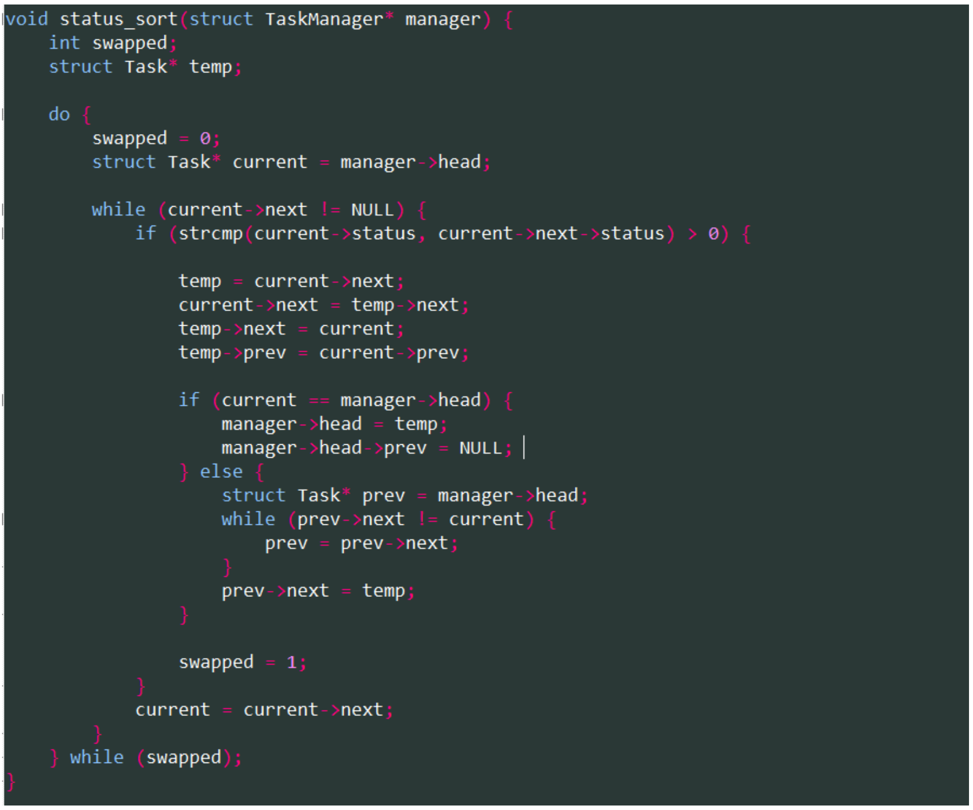


```c
void duedate_sort(struct TaskManager* manager) {
    int swapped = 1;
    struct Task* temp;

    while (swapped) {
        swapped = 0;
        struct Task* current = manager->head;
        struct Task* prev = NULL;

        while (current != NULL && current->next != NULL) {
            if (strcmp(current->due_date, current->next->due_date) > 0) {

                temp = current->next;
                current->next = temp->next;
                temp->next = current;
                temp->prev = prev;

                if (prev == NULL) {
                    manager->head = temp;
                    manager->head->prev = NULL;
                } else {
                    prev->next = temp;
                }

                swapped = 1;
            }

            prev = (swapped == 0) ? current : prev;
            current = current->next;
        }
    }
}
```

If a pair of tasks is found to be out of order, the function performs the necessary swaps, adjusting pointers to maintain the integrity of the linked list. This includes updating the next and prev pointers of the swapped tasks and ensuring that the head pointer of the list is appropriately updated if the new head is involved in the swap.
The sorting process continues until a pass through the list is completed without any swaps, indicating that the list is now sorted.

By task sort function

The status_sort function is will sort tasks within the task manager's list based on their status. This function implements the Bubble Sort algorithm, a straightforward and comparison-based sorting method. The key idea is to compare adjacent tasks in the linked list by their status and swap them if they are in the wrong order. The sorting process continues until a pass through the list is completed without any swaps, indicating that the list is now sorted.



```
void status_sort(struct TaskManager* manager) {
    int swapped;
    struct Task* temp;

    do {
        swapped = 0;
        struct Task* current = manager->head;

        while (current->next != NULL) {
            if (strcmp(current->status, current->next->status) > 0) {

                temp = current->next;
                current->next = temp->next;
                temp->next = current;
                temp->prev = current->prev;

                if (current == manager->head) {
                    manager->head = temp;
                    manager->head->prev = NULL;
                } else {
                    struct Task* prev = manager->head;
                    while (prev->next != current) {
                        prev = prev->next;
                    }
                    prev->next = temp;
                }

                swapped = 1;
            }
            current = current->next;
        }
    } while (swapped);
}
```

In the function, a flag swapped is initialized to 0, and a temporary task pointer temp is declared to facilitate task swapping. The sorting process takes place within a do-while loop, traversing the linked list and comparing adjacent tasks' statuses.

If a pair of tasks is found to be out of order, the function performs the necessary swaps, adjusting pointers to maintain the integrity of the linked list. This includes updating the next and prev pointers of the swapped tasks and ensuring that the head pointer of the list is appropriately updated if the new head is involved in the swap.

The do-while loop ensures that the sorting process continues until a pass through the list is completed without any swaps, ensuring that the list is fully sorted.