OOP Final Project Report

Omar El Hajj (ose6432)

ENGR-UH 2510 Final Project Report

March 3rd 2025

1. Introduction

Whether you're managing assignments as a student or keeping up with deadlines as a professional, staying organized is crucial. With the increasing number of tasks people juggle daily, a structured system is essential to ensure efficiency and productivity. To address this challenge, this project introduces a Task Manager developed in C++. The system provides users with a simple yet powerful way to create, organize, and track tasks. This report details the design, implementation, and key insights gained from building this system. The project utilizes Object-Oriented Programming (OOP) concepts such as inheritance, polymorphism, and templates to create a scalable and efficient solution. The system is designed with a user-friendly console interface that allows easy task management.

2. Approach & System Design

The **Task Manager** is designed to provide a seamless experience for users looking to manage their tasks effectively. The core functionalities include:

- Adding tasks with a **title, description, and due date**.
- Displaying tasks in a **clean, readable format**.
- Marking tasks as **completed** with a simple command.
- **Reordering** tasks dynamically based on priority.
- **Saving and loading tasks** from a file for persistence.

Key Design Considerations

The system is built using a modular approach:

1. **Task Class:** Represents an individual task, encapsulating its properties and behaviors.
2. **UrgentTask Class:** Extends the Task class, adding urgency-specific features.
3. **TaskManager Class (Template-Based):** Handles dynamic storage, task management, and priority changes.
4. **FileHandler Class:** Manages reading and writing tasks to a file for persistent storage.
5. **Main Function:** Provides an interactive console-based menu for user interaction.

OOP concept	**Implementation**	**Where in the code**	

Inheritance	Extensible design allowing future specialized task types	- UrgentTask class inherits from Task.
		- UrgentTask overrides the display method.
			class UrgentTask : public Task {
			public:
			UrgentTask(std::string t, std::string d, std::string date);
			void display() const override; // Override the display method
			};

Templates	TaskManager<T> is a generic class, making reusable for various task types	- TaskManager<T> is defined as a template class.
		- Used to manage Task and UrgentTask objects.
			template <typename T>
			class TaskManager {
			private:
			T** tasks; // Array of pointers to tasks
			// ...
			public:
			void addTask(T* task); // Adds a task of type T
			// ...
			};

Polymorphism	Task and UrgentTask objects are treated as Task objects but behave differently.	- Task has a virtual display method.
		- UrgentTask overrides display.
		- TaskManager calls display on Task* objects, and the correct method is called based on the actual object type.
			In the task class:
			virtual void display() const;
			
			In the urgent task class:
			void display() const override;

Classes	Encapsulation of data and behavior into Task, UrgentTask, and TaskManager.	- Task class: Encapsulates task details (title, description, due date, completed status). - UrgentTask class: Adds urgency-specific behavior. - TaskManager class: Manages a collection	

		of tasks.

3. Solution: Code Highlights & Optimizations

1. Task Class

The task class is the backbone of the project, ensuring that all task details are managed efficiently.

```
class Task {
protected:
    string title, description, dueDate;
    bool completed;
public:
    Task(string t, string d, string date) : title(t), description(d), dueDate(date), completed(false) {}
    void markCompleted() { completed = true; }
    bool isCompleted() const { return completed; }
    void display() const {
        cout << (completed ? "✅" : "❌") << " " << title << " (Due: " << dueDate << ")\n";
    }
};
```

2. Dynamic Task Storage & Expansion

A key challenge was ensuring that the task list remains scalable. The TaskManager class uses dynamic memory allocation to store tasks, doubling the size when necessary.

```
void TaskManager<T>::expandArray() {
    int newCapacity = capacity * 2;
    T** newTasks = new T*[newCapacity];
    for (int i = 0; i < size; i++)
        newTasks[i] = tasks[i];
    delete[] tasks;
    tasks = newTasks;
    capacity = newCapacity;
}
```

3. Changing Task Priority

```
void TaskManager<T>::changeTaskPriority(int oldIndex, int newIndex) {
   if (oldIndex < 0 || oldIndex >= size || newIndex < 0 || newIndex >= size) {
      cout << "Invalid move\n";
      return;
   }
   T* temp = tasks[oldIndex];
   if (oldIndex < newIndex) {
      for (int i = oldIndex; i < newIndex; i++)
         tasks[i] = tasks[i + 1];
   } else {
      for (int i = oldIndex; i > newIndex; i--)
         tasks[i] = tasks[i - 1];
   }
   tasks[newIndex] = temp;
   std::cout << "Task priority updated\n";
}
```

Optimization

1. **Optimization 1:** Dynamic Array Expansion. File Name / Line Number(s): [TaskManager.h / 36-38, TaskManager.cpp / 45-56]

   The TaskManager dynamically expands its array of tasks when it runs out of space, doubling the capacity each time. This avoids frequent reallocations and improves performance by reducing the overhead of memory management.

2. **Optimization 2:** Lazy Initialization for TaskManager File Name / Line Number(s): [TaskManager.h / 36-38, TaskManager.cpp / 45-56]

   The TaskManager uses lazy initialization to delay creating its internal array (tasks) until the first task is added. This reduces memory usage if the TaskManager is unused, improving efficiency. It also avoids unnecessary memory allocation during construction, making the program faster when the TaskManager isn’t used immediately.

3. **Optimization 3:** Inline function. File Name / Line Number(s): [TaskManager.h / 51]

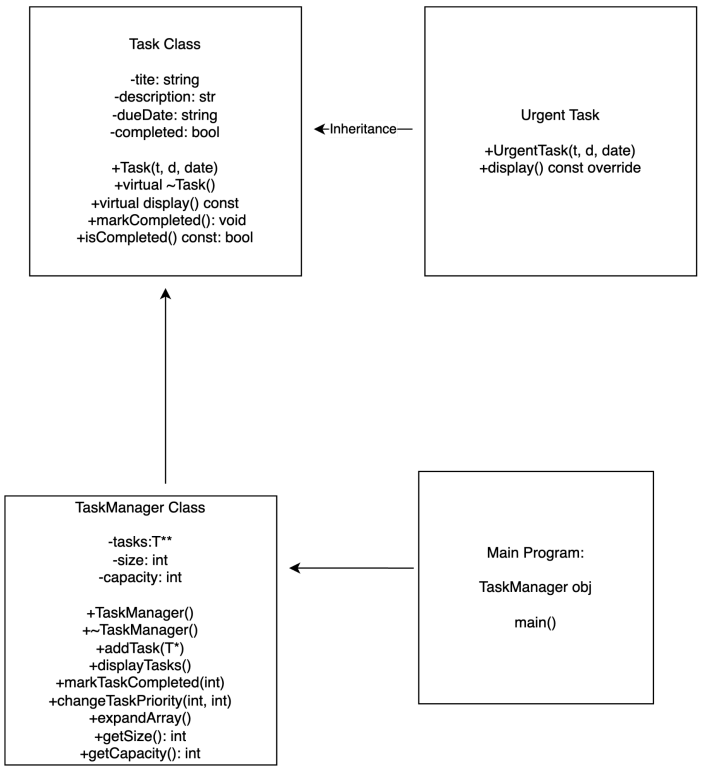
   The inline keyword on getSize() reduces function call overhead by allowing the compiler to directly insert the function’s code wherever it's called. This results in faster execution, particularly when accessing the task count frequently.

4. **Optimization 4:** Avoiding Unnecessary Copies. File Name / Line Number(s): [TaskManager.h / 24-26, TaskManager.cpp / 45-56]

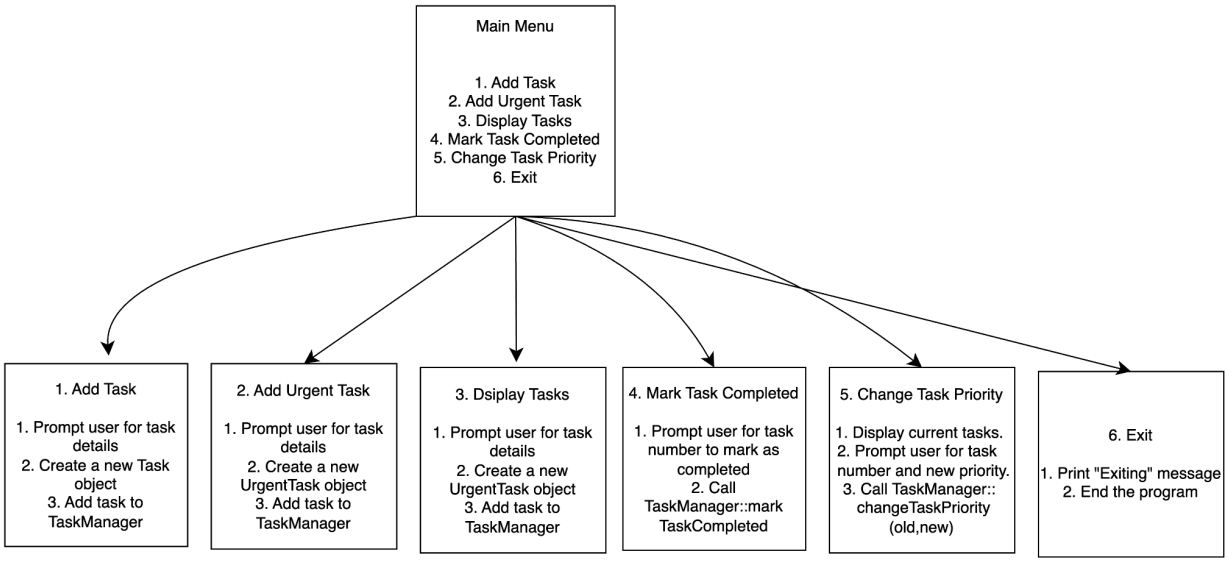
   The TaskManager stores pointers to tasks (T** tasks) instead of copying task objects. This reduces memory usage and improves performance, especially for large numbers of tasks, by avoiding expensive copy operations.

4. System Architecture (Diagram)

To visualize the Task Manager's workflow, below is a high-level flowchart representing the system's structure:



Here is a flowchart explaining how the program works:



5. Demo of the code

Adding Task

```
TO DO LIST MENU
1. Add Task
2. Add Urgent Task
3. Display Tasks
4. Mark Task Completed
5. Change Task Priority
6. Exit
enter choice: 1
enter title: demo 1
enter description: demo
enter due date (eg. dd-mm-yyyy): 20-09-2025
```

Adding an Urgent Task

```
enter choice: 2
enter title: URGENT demo 2
enter description: urgent demo
enter due date (eg. dd-mm-yyyy): 20-09-2026
```

Displaying Tasks

```
enter choice: 3
```

```
your to-do list:
1. ❌ demo 1 (Due: 20-09-2025)
   demo
2. ❌ URGENT! URGENT demo 2 (Due: 20-09-2026)
   urgent demo
```

Marking Tasks as completed

```
enter choice: 4
enter task number to mark completed: 1
task marked as completed
```

Displaying Tasks After Marking as Completed

```
enter choice: 3
your to-do list:
1. ✅ demo 1 (Due: 20-09-2025)
   demo
2. ❌ URGENT! URGENT demo 2 (Due: 20-09-2026)
   urgent demo
```

Changing tasks priority

```
enter choice: 5
current tasks:
1. demo 1
2. URGENT demo 2
enter task number to move: 2
enter new priority position: 1
task priority updated
```

6. Challenges and self evaluation and checklist

Challenges Faced:

- **Memory Management Issues**: Initial implementations had memory leaks due to improper deallocation of dynamically allocated objects.
- **User Input Handling**: Ensuring that invalid inputs (such as negative task indices) do not crash the program required implementing error handling mechanisms.

- **Dynamic Task Prioritization**: Moving tasks around while maintaining the correct order was a challenge that required implementing an efficient priority adjustment function.

Code:	Points	Student's Evaluation	Instructor's Evaluation
Readability	3	3	
Optimization	3	2	
OOP Concepts	3	3	
Output	3	3	
Correctness	3	3	
Report:			
Engaging	3	3	
Grammar	3	3	
Structure	3	3	
Visual	3	3	
Content	3	3	
References	3	2	
Total	33	31	

Check the list below as they are completed using the [x] symbol.

- ☑ ~~Read this document.~~
- ☑ ~~Understand the evaluation scheme.~~
- ☑ ~~Place code files into the **code/** directory.~~
- ☑ ~~Ensure code files can be executed on a standard linux terminal and available tools (bash, g++ compiler and linker, make)~~
- ☑ ~~Provide a screenshot of your code's output.~~
- ☑ ~~Submit your report in **Report.md**.~~
- ☑ ~~Complete the self-assessment according to the defined evaluation scheme.~~
  - ☑ ~~Complete the self-assessment table (2.2) below.~~
  - ☑ ~~Enumerate location of implemented concepts in section (2.3) below.~~

7. Conclusion

This Task Manager demonstrates the use of Object-Oriented Programming by integrating efficient data structures, dynamic memory management, and modular design.

Future improvements could include:

Graphical User Interface (GUI) for a better user experience.
Task categorization with labels and filters.
File-based storage for saving and loading tasks locally, improving accessibility and persistence.

This project reinforced key C++ concepts and demonstrated how software engineering principles apply in real-world development.

8. References

1. GeeksforGeeks Virtual function in C++: https://www.geeksforgeeks.org/virtual-function-cpp/
2. Draw.io for flowchart creation: https://www.drawio.com
3. Flask.io for online to-do list: https://flask.io/new
4. GeeksforGeeks Inheritance in C++: https://www.geeksforgeeks.org/inheritance-in-c/
5. Dynamic Memory Allocation in C++: https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/