1. Introduction

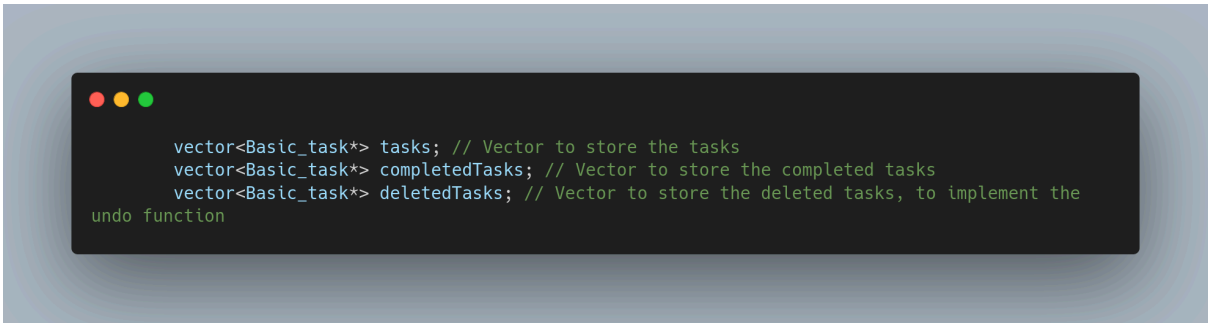
This program is DS and OOP midterm project. The project is to design a Todo lists program that have loads of functions which can used by users.

In the implementation, i spilt the project into 3 main parts, respectively basic task header file, to do list header file and the main program.
Basic task header file is handling the class declaration and definition of different type of the tasks; To do list header file is for the main implementation of different kind of functions in the To do list program, like add file, view file, etc. As for the main file is handling the UI of this program.

In the following, i will explain how to implement the functions in the program by these three files.

2. Function implementation

a. The data structure to store the task information.
We use vector to store every task, and the data type is “Basic_task” pointer, for the implementation of the polymorphism.



```
vector<Basic_task*> tasks; // Vector to store the tasks
vector<Basic_task*> completedTasks; // Vector to store the completed tasks
vector<Basic_task*> deletedTasks; // Vector to store the deleted tasks, to implement the
undo function
```

b. Type of the Task
We have three different tasks can use in the todo list program.
   i. Normal Task

```cpp
/*Implemented the normal class, just normal task the priority isn't first. The following is the
important point of implementation:
1. tm is a structure taht contains sec, min, hour, day, month, year etc., we use this in set date
2. We additionally add two private members, one sis the description of the task another is due day
3. Output type of normal_task [Normal]name|category|description|due date|is completed
*/
class Normal_task : public Basic_task{

    private:
        string *description;
        tm *dueDate;

    public:
        // Constructor, we need to allocate memory for the pointers, due_date = tm() means to
create a struct tm with default values
        Normal_task(const string& name, const string& category, const string& description = "",
const tm& dueDate = tm()):Basic_task(name, category), description(new string(description)),
dueDate(new tm(dueDate)) {}
        // Destructor, we need to free the memory allocated for the pointers
        ~Normal_task(){
            delete description;
            delete dueDate;
        }
        // Getters
        string get_descripition() const{return *description;}
        tm get_due_time() const {return *dueDate;}

        // Setters
        void set_decription(const string& New_description) {*description = New_description;}
        void set_due_time(const tm& New_due_time) {*dueDate = New_due_time;}

        // Display function for Normal_task
        void display() const override{
            cout << "[Normal] " << get_name() << "| Category: " << get_category();
            if(!get_descripition().empty()){// If the description is not empty, display it
                cout << "| Description: " << get_descripition();
            }
            // dueDate-> tm_year>0 is to check if the date is set, tm_year = 0 means the date is
not set(the default value), tm_year = 0 means 1900
            if(dueDate->tm_year > 0){// If the dueDate is correctly set, disp[lay it ]
                /*
                Range of tm_mon is 0-11, so we need to add 1 to it to display the correct month
                tm_mda is the day of the month, so we can display it directly
                tm_year is the year since 1900, so we need to add 1900 to it to display the correct
year
                */
                cout << "| Due: " << (dueDate->tm_mon+1) << "/" << dueDate->tm_mday << "/" <<
(dueDate->tm_year + 1900);
            }
            cout << "| Status: " << (is_completed() ? "Completed" : "Pending") << endl;
        }

};
```

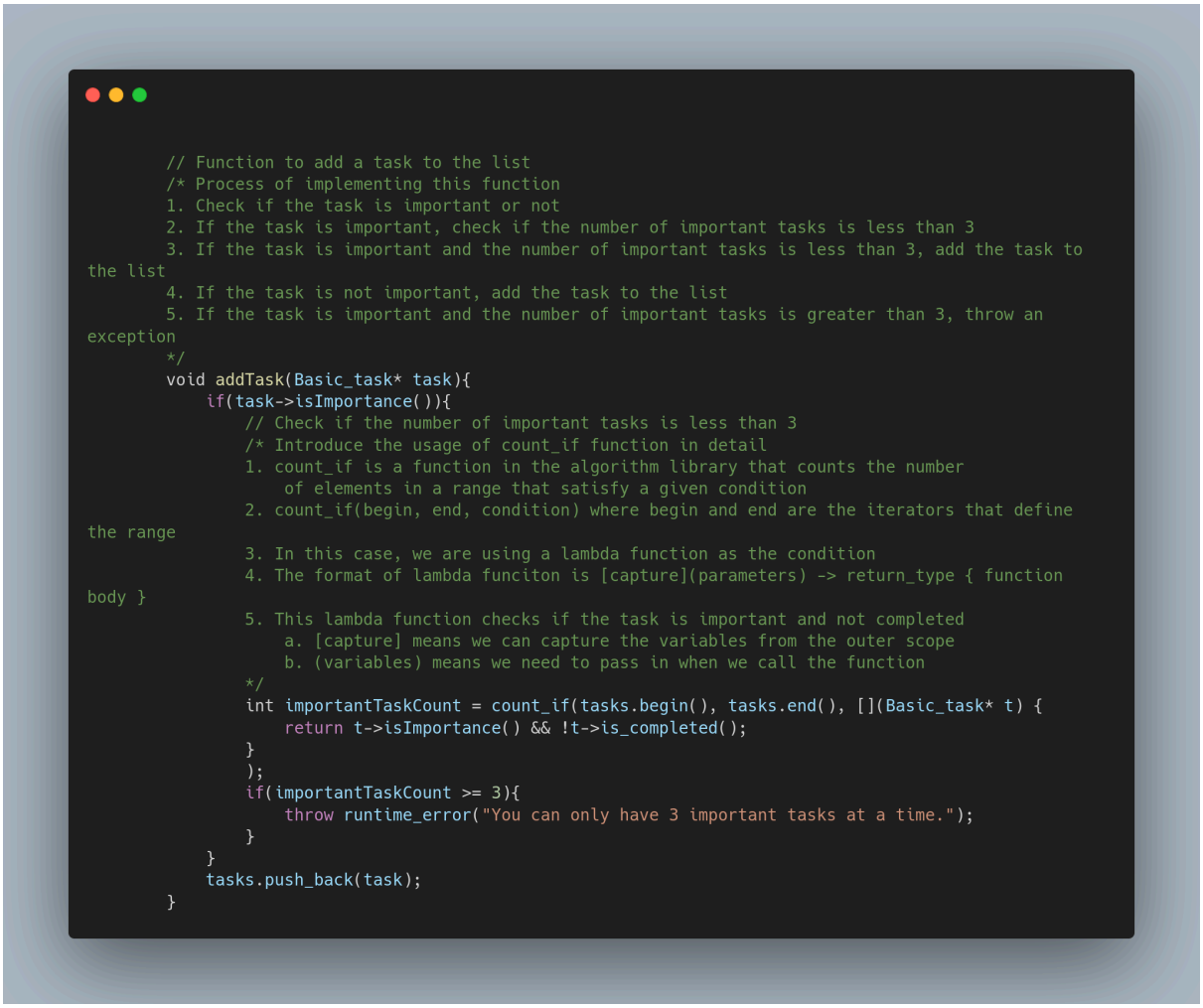
ii. Important Task

```cpp
/* Important task have the following properties
1. Important task have the highest priority, we must finish the important tasks before other tasks
2. Important task only allows have 3 in the same time.
3. The addition properties are priority and description
4. Since important tasks are the task we must finish first, we don't need to set the due date
5. The output format is [Important(priority)]name|category|description|is completed
*/
class ImportantTask : public Basic_task{
    private:
        string *description;
        int *priority;

    public:
        // Constructor, we need to allocate memory for the pointers
        ImportantTask(const string& name, const string& category, const string& description = "",
int priority = 1):Basic_task(name, category), description(new string(description)), priority(new
int(priority)) {}
        // Destructor, we need to free the memory allocated for the pointers
        ~ImportantTask(){
            delete description;
            delete priority;
        }
        // Getters
        string get_description() const{return *description;}
        int get_priority() const{return *priority;}

        // Setters
        void set_description(const string& New_description) {*description = New_description;}
        void set_priority(int New_priority) {
            if(New_priority >= 1 && New_priority <= 3){// Check if the priority is between 1 and 3
                *priority = New_priority;
            }
        }

        bool isImportance() const override {return true;}// To indicate if the task is important
task or not
        // Display function for ImportantTask
        void display() const override{
            cout << "[Important(" << get_priority() << ")]" << get_name() << "| Category: " <<
get_category();
            if(!get_description().empty()){// If the description is not empty, display it
                cout << "| Description: " << get_description();
            }
            cout << "| Status: " << (is_completed() ? "Completed" : "Pending") << endl;
        }
};
```

iii. Recurring Task

```cpp
/*Following are the properties of the recurring task
1. Recurring task is a task that happen in same time interval, like every day,
 every week, every month, we can define the interval by the user
2. Recurring rask is inherited from the normal task, but have additiona; properoties
 like the interval ofthe next task, and the date of the next task
3. The output format is [Recurring (every ...)]name|category|description|due date|is completed
4. In recurring task, we don't need to set the due date, because the due date is the next occurrence
*/
class RecurringTask : public Normal_task{
    private:
        int* recurrenceDays; // Number of days between each occurrence
        tm* nextOccurance; // Date of the next occurrence
    public:
        // Constructor, we need to allocate the memory for the pointers
        RecurringTask(const string& name, const string& category, const tm& startDate, const string&
description = "",  int recurrenceDays = 1) :Normal_task(name, category, description, startDate),
recurrenceDays(new int(recurrenceDays)), nextOccurance(new tm(startDate)) {}
        // Destructor, we need to free the memory allocated for the pointers
        ~RecurringTask(){
            delete recurrenceDays;
            delete nextOccurance;
        }

        // Getters
        int getRecurranceDays() const{return *recurrenceDays;}
        tm getNextOcurance() const{return *nextOccurance;}

        // Set the reccurance to true
        bool isRecurring() const override{return true;}

        // Indicate the next occurrence of the task
        void markNectOccurance() override{
            // Add todays date with the recurrence days, because we initial the next occurrence with
the start date
            nextOccurance -> tm_mday += *recurrenceDays;
            // Use tm's built in function to check if the date is valid, mktime() helps us
automatically adjust the date, for example: 1/32 -> 2/1
            // time_t mktime(tm* timeptr);The input us a tm pointer
            mktime(nextOccurance);// So we don't need to dereference nextOccurance
            // Have to reset the completed to false, because if we finish the task before the next
occurrence, the completed is true and the next occurrence is also inherited to be true
            set_completed(false);
        }
        void display() const override {
            cout << "[RECURRING (every " << getRecurranceDays() << " days)] " << get_name()
                << " | Category: " << get_category();

            if (!get_descripition().empty()) {
                cout << " | Description: " << get_descripition();
            }

            if (nextOccurance->tm_year > 0) {
                cout << " | Due: " << (nextOccurance->tm_mon + 1) << "/" << nextOccurance->tm_mday <<
"/" << (nextOccurance->tm_year + 1900);
            }

            cout << " | Status: " << (is_completed() ? "Completed" : "Pending") << endl;
        }

};
```

c. Add task



```
        // Function to add a task to the list
        /* Process of implementing this function
        1. Check if the task is important or not
        2. If the task is important, check if the number of important tasks is less than 3
        3. If the task is important and the number of important tasks is less than 3, add the task to
the list
        4. If the task is not important, add the task to the list
        5. If the task is important and the number of important tasks is greater than 3, throw an
exception
        */
        void addTask(Basic_task* task){
            if(task->isImportance()){
                // Check if the number of important tasks is less than 3
                /* Introduce the usage of count_if function in detail
                1. count_if is a function in the algorithm library that counts the number
                    of elements in a range that satisfy a given condition
                2. count_if(begin, end, condition) where begin and end are the iterators that define
the range
                3. In this case, we are using a lambda function as the condition
                4. The format of lambda funciton is [capture](parameters) -> return_type { function
body }
                5. This lambda function checks if the task is important and not completed
                    a. [capture] means we can capture the variables from the outer scope
                    b. (variables) means we need to pass in when we call the function
                */
                int importantTaskCount = count_if(tasks.begin(), tasks.end(), [](Basic_task* t) {
                    return t->isImportance() && !t->is_completed();
                }
                );
                if(importantTaskCount >= 3){
                    throw runtime_error("You can only have 3 important tasks at a time.");
                }
            }
            tasks.push_back(task);
        }
```

d. View task

e. Mark task completed

```
        // Function to mark a task as completed
        /*Procedure to implement this function
        1. Check if the input index is valid, if not, throw an exception
        2. If the input index is valid, change the  status of the task tocompleted and put it into the
completed task list
        3. For the recurring
            a. Keep the task in the list
            b. Automatically calculate the next happening date
            c. Reset the status of the task to not completed
        <tip> b and c we can do this by using the function in the recurring task class called
markNextOccurance
        */
       void markTaskCompleted(int index){
            if(index < 0 || index >= tasks.size()){
                throw runtime_error("Invalid task index.");// After throw, it will stop executing this
function
            }
            tasks[index]->set_completed(true); // Set the task as completed
            completedTasks.push_back(tasks[index]); // Add the task to the completed tasks list

            if(tasks[index]->isRecurring()){ //If the tasks we have completed is recurring task
                tasks[index]->markNectOccurance(); // Mark the next occurrence of the task
            }
       }
```

f. Edit task

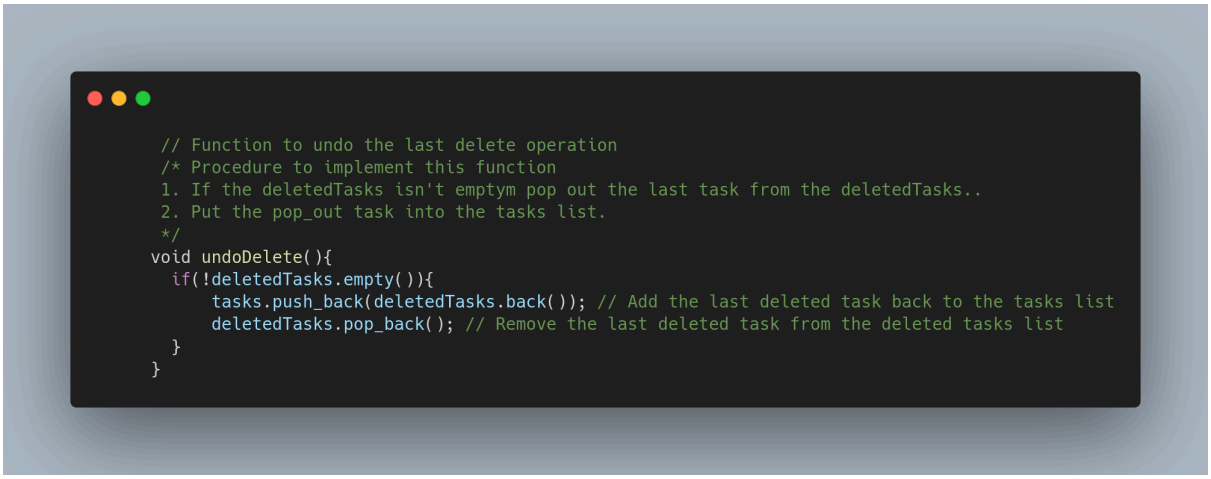
```
       // Function to edit a task in the list
        /*Procedure to implement this function
        1. Check if the input index is valid, if not, throw an exception
        2. Input new task name, category.
        */
       void editTask(int index, const string& newName, const string& newCategory){
        if(index < 0 || index >= tasks.size()){
            throw runtime_error("Invalid task index.");// If the index is invalid, throw runtime_error
        }
        tasks[index]->set_name(newName); // Set the new name for the task
        tasks[index]->set_category(newCategory); // Set the new category for the task
       }
```

g. Delete task

```
        // Function to check every task in the list
        /*
        1. We can determine what kind of task we want to display.ADJ_FREQUENCY
        2. We can display
            a. All the tasks
            b. Base on the category
            c. Base on completed or not
            d. Only show the important tasks
        3. Based on the above conditions, we can display the tasks, and the input parameters are the
condition
        4. The input parameter means
            a. categoryFilter: the category we want to display, if it is empty, we display all the
tasks
            b. showCompleted: If we show the completed tasks or not
            c.importantOnly: If only show the important tasks or not
        */
        void viewTasks(const string& categoryFilter = "", bool showCompleted =false, bool importantOnly
= false) const{
            cout << endl << "---- Task List ----" << endl;
            int count = 0; // Count for the matching tasks
            for(const auto& task : tasks){
                // The filter condition for the tasks
                bool categoryMatch = categoryFilter.empty()|| (task->get_category() == categoryFilter);
//If the category is empty or the task category is the same as the filter category
                bool completedMatch = showCompleted || !(task->is_completed()); //Show already
completed or tasks isn't completed
                bool importantMatch = !importantOnly || task->isImportance(); //Show donb't select
important task or is important task
                if(categoryMatch && completedMatch && importantMatch){
                    cout << count+1 << ". ";
                    task->display();
                    count++; // Increase the count for the matching tasks
                }

            }
            if(count == 0){// No matching tasks
                cout << "No tasks found matching the criteria." << endl;
            }
        }
```

```
       // Function to delee a task from the list
        /*Procedure to implement this function
        1. Check if the input index is valid, if not, throw an exception
        2. If the input index is valid, delete the task from the list and put it into the deleted task
list
        */
       void deleteTask(int index){
        if(index < 0 || index >= tasks.size()){// If the index is invalid
            throw runtime_error("Invalid task index.");// Throw runtime_error
        }
        // Push the tak into deletedTasks
        deletedTasks.push_back(tasks[index]); // Add the task to the deleted tasks list
        // Remove the task from tasks
        tasks.erase(tasks.begin()+index); // Remove the task from the list
       }
```

h. Undo task



```
// Function to undo the last delete operation
/* Procedure to implement this function
1. If the deletedTasks isn't emptym pop out the last task from the deletedTasks..
2. Put the pop_out task into the tasks list.
*/
void undoDelete(){
  if(!deletedTasks.empty()){
      tasks.push_back(deletedTasks.back()); // Add the last deleted task back to the tasks list
      deletedTasks.pop_back(); // Remove the last deleted task from the deleted tasks list
  }
}
```

3. Conclusion

I think the hardest part of implementing this program is how you arrange the purpose of every file, in past, i always put the whole program in the same file, but i find out that not only do i constantly confused of the code i write, the debug part is also very hard to do since the whole thing just stuck together.
In this program, i try to split to three files, and each file have very specific purpose, not only does it very clear to the part what i have to code, but also enhance the ability of class, constructor, inheritance and polymorphism taught in the past 8 weeks. And i also find some cool built-in function of c++, like “tm” struct to record and calculate the time, and c++’s lambda function etc.
I’m very appreciate to use this project to enhance my programming ability and that’s all of my report.

My github repo for this program:
https://github.com/ModernHuman0531/Data-Structures-and-Object-oriented-Programming_2025/tree/main/Todo_list