UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program Design: Introduction

Assignment 3 Specifications

Release Date: 21-10-2024 at 06:00

Due Date: 07-11-2024 at 23:59

Total Marks: 320

Contents

1 General Instructions

- ***Read the entire assignment thoroughly before you start coding.***
- This assignment should be completed individually, no group effort is allowed.
- **To prevent plagiarism, every submission will be inspected with the help of dedicated software.**
- Be ready to upload your assignment well before the deadline, as **no extension will be granted.**
- If your code does not compile, you will be awarded a mark of 0. The output of your program will be primarily considered for marks, although internal structure may also be tested (eg. the presence/absence of certain functions or classes).
- Failure of your program to successfully exit will result in a mark of 0.
- Note that plagiarism is considered a very serious offence. Plagiarism will not be tolerated, and disciplinary action will be taken against offending students. Please refer to the University of Pretoria's plagiarism page at `https://portal.cs.up.ac.za/files/departmental-guide/`.
- Unless otherwise stated, the usage of c++11 or additional libraries outside of those indicated in the assignment, will **not** be allowed. Some of the appropriate files that you have submit will be overwritten during marking to ensure compliance to these requirements. **Please ensure you use c++98**
- All functions should be implemented in the corresponding `cpp` file. No inline implementation in the header file apart from the provided functions.
- The usage of ChatGPT and other AI-Related software is strictly forbidden and will be considered as plagiarism.
- Note, UML notation for variable and function signatures are used in this assignment.

2 Overview

In this assignment you will implement a simple task manager system. You will make use of linked lists and templates to implement the data structures that are commonly used in task managing systems. You will implement the following templatised data structures:

- Doubly Linked List
- Circular Double Linked List
- Dynamic queue
- Dynamic stack

3 Background

Linked lists in C++ are a dynamic data structure that allows efficient insertion and deletion of elements. Unlike arrays, which have a fixed size, a linked list grows and shrinks at runtime, with each element (called a node) containing a value and a pointer to the next node. This structure enables efficient sequential access, making it useful for tasks where the size of the data is unknown or where frequent modifications are needed. In this practical assignment, students will implement basic operations on a linked list, such as adding, removing, and traversing nodes, to develop their understanding of dynamic memory management and pointers in C++.

Templates are a feature in C++ that enable a function or a class to accept generic parameters. Rather than coding a function to accept a specific kind of input parameter type, it can be coded to accept a generic type that can then be implemented to respond to different kinds of inputs. This can be extended for classes which enable them to be parameterised with different types. Templates can be applied to both functions and classes in order to create a generic method/class once, and have it apply to many different types at compile time. By specifying a type in a template parameter, the class is compiled by using that specific datatype in the way specified. While multiple template parameters can be used (more than one type can be put in a template parameter list) for this practical, you will only use single-parameter templates.

4 Your Task:

You are required to implement the following class diagram illustrated in Figure 1. Pay close attention to the function signatures as the `h` files will be overwritten, thus failure to comply with the UML, will result in a mark of 0.

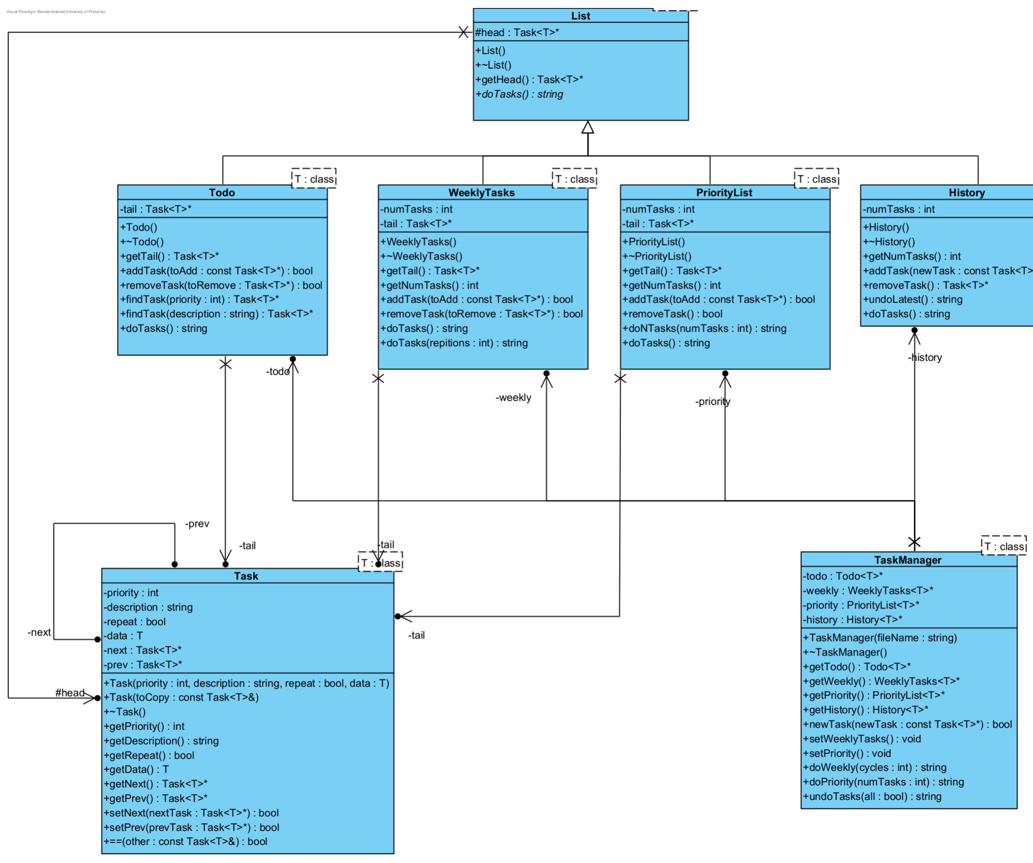

Figure 1: Class diagrams

Note, the importance of the arrows between the classes are not important for COS 110 and will be expanded on in COS 214.

4.1 Task<T>

The Task class is a template class which will be used as the ”nodes” in the linked lists.

4.1.1 Members

- - next: Task<T>*
  - This member variable is the pointer to the next task in the list.
- - prev: Task<T>*
  - This member variable is the pointer to the previous task in the list.
- - priority: int
  - This member variable represents the priority of the task.
  - A lower numbers correspond to higher priority tasks.
- - description: string
  - This member variable represents a description of the task.
- - repeat: bool
  - This member variable indicates if the task is a weekly task.
- - data: T
  - This member variable represents some data needed to complete the task.

4.1.2 Functions

- + Task(priority: int, description: string, repeat: bool, data: T)
  - This is the constructor for the Task class.
  - Set the member variables to the corresponding parameters.
  - Set next and prev to NULL.
- + Task(toCopy: const Task<T>&)
  - This is the copy constructor for the Task class.
  - Set the member variables to the corresponding variables of toCopy.
  - Set next and prev to NULL.
- + ∼Task()
  - This is the destructor for the Task class.
  - It should not deallocate the next and prev member variables, therefore leave it blank.

- + getPriority(): int
  - Return the priority member variable.
- + getDescription(): string
  - Return the description member variable.
- + getRepeat(): bool
  - Return the repeat member variable.
- + getData(): T
  - Return the data member variable.
- + getNext(): Task<T>*
  - Return the next member variable.
- + getPrev(): Task<T>*
  - Return the prev member variable.
- + setNext(nextTask: Task<T>*): bool
  - Set the next member variable to nextTask.
  - Return true.
- + setPrev(prevTask: Task<T>*): bool
  - Set the prev member variable to prevTask.
  - Return true.
- + operator==(other: const Task<T>&): bool
  - This method checks if the current object's member variables are equal to other. It should not check if next and prev are equal.
  - Return true if all are equal and false otherwise.

4.2 List<T>

The List class is a template class and will serve as the base class for Todo, WeeklyTasks, PriorityList and History.

4.2.1 Members

- # head: Task<T>*
  - This member variable stores the pointer to the head of the linked list (the first task in the list).

4.2.2 Functions

- + List()
  - This is the constructor for the List class.
  - Set head to NULL
- + *~List()*
  - This is the destructor for the List class.
  - It should be left blank.
- + getHead(): Task<T>*
  - Return the head member variable.
- + *doTasks(): string*
  - This is a pure virtual function.

4.3 Todo<T>

The Todo class is a template class and inherits publicly from List. It is a doubly linked list.

4.3.1 Members

- - tail: Task<T>*
  - This member variable stores the pointer to the tail of the linked list (the last task in the list).

4.3.2 Functions

- + Todo()
  - This is the constructor for the Todo class.
  - Set tail to NULL.
  - Remember to call the base class's constructor.
- + ~Todo()
  - This is the destructor for the Todo class.
  - It should deallocate all the tasks in the list.
- + getTail(): Task<T>*
  - Return the tail member variable.
- + addTask(toAdd: const Task<T>*): bool

  - This method appends (insert at the tail) a new task to the list.
  - If toAdd is not NULL, create a copy of toAdd and append it to the list and return true. Otherwise return false.
  - Remember Todo is a doubly linked list and to consider all edge cases.

- + removeTask(toRemove: Task<T>*): bool
  - This method removes a toRemove from the list.
  - If the list is empty or if toRemove is NULL, return false.
  - Remember Todo is a doubly linked list and to consider all edge cases.

- + findTask(priority: int): Task<T>*
  - This method should search the linked list for a task with the passed in priority.
  - It should return the first task with that priority.
  - If no task is found with the corresponding priority, return NULL.

- + findTask(description: string): Task<T>*
  - This method should search the linked list for a task with the passed in description.
  - It should return the first task with that description.
  - If no task is found with the corresponding description, return NULL.

- + doTasks(): string
  - This method returns a string with all the tasks in the list.
  - The string should be formatted as follows for each task:

```
                description\n
```

  - Where description represent the description of the task and \n a newline.

4.4 WeeklyTasks<T>

The WeeklyTasks class is a template class which inherits publicly from List. It is a circular linked list. This list represents tasks that need to be completed on a weekly basis.

4.4.1 Members

- - tail: Task<T>*
  - This member variable stores the pointer to the tail of the linked list (the last task in the list).

- - numTasks: int
  - This member variable represents the number of nodes currently in the list.

4.4.2 Functions

- + WeeklyTasks()
  - This is the constructor for the WeeklyTasks class.
  - Set tail to NULL.
  - Set numTasks to 0.
  - Remember to call the base class's constructor.
- + ∼WeeklyTasks()
  - This is the destructor for the WeeklyTasks class.
  - It should deallocate all the tasks in the list.
- + getTail(): Task<T>*
  - Return the tail member variable.
- + getNumTasks(): int
  - Return the numTasks member variable.
- + addTask(toAdd: const Task<T>*): bool
  - This method adds a new task to the list.
  - If toAdd is not NULL, create a copy of toAdd and add it to the list and return true. Otherwise return false.
  - The tasks in the list should be ordered in alphabetical order, based on their descriptions.
  - If a new task is alphabetically equivalent to a task already in the list, it should be added before the task already in the list.
  - For example assume you have four tasks with the following descriptions: B, D, A, A. Then the linked list should be in the following order H →A→ A → B → D → T. Where H is the head and T the tail.
  - Increment numTasks.
  - Only link the nodes using the next member of Task.
  - Remember WeeklyTasks is a circular linked list and to consider all edge cases.
- + removeTask(toRemove: Task<T>*): bool
  - This method removes toRemove from the list.
  - If the list is empty or if toRemove is NULL, return false.
  - Decrement numTasks.
  - Remember Todo is a circular linked list and to consider all edge cases.
- + doTasks(): string

- This method builds a string that represents the tasks being done.
- Each task should be added to the string with the following format:

```
Task: description Data: data COMPLETED\n
```

- This method should only ”do” each task once.

- + doTasks(repetitions: int): string
  - This method builds a string that represents the tasks being done.
  - Each task should be added to the string with the following format:

```
Task: description Data: data COMPLETED\n
```

  - This method should only ”do” each task repetitions times.

4.5 PriorityList<T>

The PriorityList is a template class which inherits publicly from List. This is a queue. Tasks are enqueued so that the tasks with a higher priority get done first.

4.5.1 Members

- - tail: Task<T>*
  - This member variable stores the pointer to the tail of the linked list (the last task in the list).
- - numTasks: int
  - This member variable represents the number of nodes currently in the queue.

4.5.2 Functions

- + PriorityList()
  - This is the constructor for the PriorityList class.
  - Set tail to NULL.
  - Set numTasks to 0.
  - Remember to call the base class’s constructor.
- + ∼PriorityList()
  - This is the destructor for the PriorityList class.
  - It should deallocate all the tasks in the list.
- + getTail(): Task<T>*
  - Return the tail member variable.

- + getNumTasks(): int
  - Return the numTasks member variable.
- + addTask(toAdd: const Task<T>*): bool
  - This method enqueue a new task to the queue.
  - If toAdd is not NULL, create a copy of toAdd and add it to the list and return true. Otherwise return false.
  - The new task should be added to the head of the list
  - Increment numTasks.
  - The nodes should be linked with the next and prev member variables.
  - Remember PriorityList is a queue and is a doubly linked list and to consider all edge cases.
- + removeTask(): bool
  - This method dequeues the task at the tail of the list.
  - If the list is empty or if toRemove is NULL, return false.
  - Decrement numTasks.
  - Remember PriorityList is a queue and is a doubly linked list and to consider all edge cases.
- + string doNTasks(numTasks: int): string
  - This method builds a string that represents the tasks being done.
  - Each task should be added to the string with the following format:

    ```
    Task: description Priority: priority Data: data COMPLETED\n
    ```

  - When the task is added to the string, remove it from the queue.
  - numTasks represent the number of tasks that should be done.
- + string doTasks(): string
  - This method builds a string that represents the tasks being done.
  - Each task should be added to the string with the following format:

    ```
    Task: description Priority: priority Data: data COMPLETED\n
    ```

  - When the task is added to the string, remove it from the queue.
  - All the tasks should be done.

4.6 History<T>

The History is a template class which inherits publicly from List. This is a stack. It represents a history of the completed tasks and will allow undo operations.

4.6.1 Members

- - numTasks: int
  - This member variable represents the number of nodes currently in the stack.

4.6.2 Functions

- + History()
  - This is the constructor for the History class.
  - Set numTasks to 0.
  - Remember to call the base class's constructor.
- + ∼History()
  - This is the destructor for the History class.
  - It should deallocate all the tasks in the list.
- + getNumTasks(): int
  - Return the numTasks member variable.
- + addTask(toAdd: const Task<T>*): bool
  - This method adds a new task to the stack.
  - If toAdd is not NULL, create a copy of toAdd and add it to the list and return true. Otherwise return false.
  - The new task should be added to the head of the list
  - Increment numTasks.
  - The nodes should be linked with the next member variables.
  - Remember History is a stack and to consider all edge cases.
- + removeTask(): Task<T>*
  - This method removes (pop) a task from the stack.
  - It should remove the head of the stack (the top task).
  - If the stack is empty, return NULL.
  - Remember History is a stack and to consider all edge cases.
- + undoLatest(): string
  - This method builds a string that represents a single task being undone.
  - Remove a task from the stack.
  - The removed task should be added to a string with the following format:

```
Task: description Priority: priority Data: data UNDID\n
```

- + doTasks(): string
  - This method builds a string that represents all the tasks being undone.
  - The removed task should be added to a string with the following format:

    ```
    Task: description Priority: priority Data: data UNDID\n
    ```

  - Remember to remove all the tasks in the stack.

4.7 TaskManager<T>

The TaskManager is a template class and will manage all the data structures described above.

4.7.1 Members

- - todo: Todo<T>*
  - This member variable represents all the tasks that should be done.
- - weekly: WeeklyTasks<T>*
  - This member variable represents all the tasks that should be done on a weekly basis.
- - priority: PriorityList<T>*
  - This member variable represents a priority queue so that tasks with a higher priority get done first.
- - history: History<T>*
  - This member variable represents the history of completed tasks.

4.7.2 Functions

- + TaskManager(fileName: string)
  - This is the constructor of the TaskManager class.
  - fileName represents the name of a textfile which contains the tasks that should be done. Each line in the textfile represents a Task.
  - The textfile has the following format:

    ```
    Priority#Description#Repeat#Data
    ```

  - Repeat will be True or False.
  - Initialise the todo list.
  - Add each task to the todo list.

  - Initialise weekly, priority and history to NULL.

- + ∼TaskManager()
  - This is the destructor for the TaskManager class.
  - It should deallocate todo, weekly, priority and history.

- + getTodo(): Todo<T>*
  - Return the todo member variable

- + getWeekly(): WeeklyTasks<T>*
  - Return the weekly member variable

- + getPriority(): PriorityList<T>
  - Return the priority member variable

- + getHistory(): History<T>*
  - Return the history member variable

- + newTask(const Task<T>* newTask): bool
  - This method should add newTask to the todo list.
  - If newTask got added to todo, return true. Otherwise return false.

- + setWeeklyTasks(): void
  - This method set the weekly list.
  - Initialise the weekly member variable
  - Traverse the todo list to and add all the tasks that should be repeated (repeat = true) to the weekly list.
  - If a task got added to the weekly list, it should be removed from the todo list.

- + setPriority(): void
  - This method sets the priority list.
  - Initialise the priority list.
  - All the tasks in the todo list, that do not repeat, should be added to the priority queue. The tasks with a lower priority should be added first.
  - If a task got added to the priority list, it should be removed from the todo list.

- + doWeekly(int cycles): string
  - This method should return a string that represents the weekly tasks done.
  - Cycles represents how many times the weekly tasks should be done.

- + doPriority(numTasks: int): string
  - This method should return a string that represents the priority tasks done.
  - numTasks represents how many of the tasks in the priority queue should be done.
  - Initialise the history member variable.
  - Each task that get done should be added to the history stack.
  - If the priority queue is empty return the following string: "Priority Queue Empty".
- + undoTasks(all: bool): string
  - This method undo the tasks done.
  - all reperesents if all tasks or only a single task should be undone.
  - If the priority list is NULL, initialise it.
  - Each task undone should be re-added to the priority queue.
  - If the history stack is empty return the following string: "Nothing to Undo".

5 Testing

As testing is a vital skill that all software developers need to know and be able to perform daily, 10% of the assignment marks will be allocated to your testing skills. To do this, you will need to submit a testing main (inside the `main.cpp` file) that will be used to test an Instructor Provided solution. You may add any helper functions to the main.cpp file to aid your testing. In order to determine the coverage of your testing the gcov [1] tool, specifically the following version *gcov (Debian 8.3.0-6) 8.3.0*, will be used. The following set of commands will be used to run gcov:

```
g++ --coverage *.cpp -o main
./main
gcov -f -m -r -j ${files}
```

This will generate output which we will use to determine your testing coverage. The following coverage ratio will be used:

$$\frac{\text{number of lines executed}}{\text{number of source code lines}}$$

We will scale this ration based on class size.

The mark you will receive for the testing coverage task is determined using Table 1:

Coverage ratio range	% of testing mark
0%-5%	0%
5%-20%	20%
20%-40%	40%

[1] For more information on gcov please see `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

40%-60%	60%
60%-80%	80%
80%-100%	100%

Table 1: Mark assignment for testing

Note the top boundary for the Coverage ratio range is not inclusive except for 100%. Also, note that only the functions stipulated in this specification will be considered to determine your testing mark. Remember that your main will be testing the Instructor Provided code and as such, it can only be assumed that the functions outlined in this specification are defined and implemented.

As you will be receiving marks for your testing main, we will also be doing plagiarism checks on your testing main.

6 Implementation Details

- You must implement the functions in the header files exactly as stipulated in this specification. Failure to do so will result in compilation errors on FitchFork.
- You may only use **c++98**.
- You may only utilize the specified libraries. Failure to do so will result in compilation errors on FitchFork.
- Do not include using `namespace std` in any of the files.
- You may only use the following libraries:
  - string
  - sstream
  - fstream
  - iostream
- You are supplied with a makefile, and all the required header files and the cpp files with the required *# ifndef, # define, # endif* blocks.
- The compilation strategy followed is the same as the preferred approach in the lectures.

7 Upload Checklist

The following c++ files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- task.h

- task.cpp
- list.h
- list.cpp
- todo.h
- todo.cpp
- weekly.h
- weekly.cpp
- priority.h
- priority.cpp
- history.h
- history.cpp
- manager.h
- manager.cpp
- `main.cpp`
- `testingFramework.h` and `testingFramework.cpp` if you used these files.

The files should be in the root directory of your zip file. In other words, when you open your zip file you should immediately see your files. They should not be inside another folder.

8 Submission

You need to submit your source files on the FitchFork website (`https://ff.cs.up.ac.za/`). All methods need to be implemented (or at least stubbed) before submission. Your code should be able to be compiled with the following command:

```
g++ *.cpp -o main
```

and run with the following command:

```
./main
```

Remember your `h` file will be overwritten, so ensure you do not alter the provided `h` files.
You have 10 submissions and your best mark will be your final mark. Upload your archive to the Practical 2 slot on the FitchFork website. Submit your work before the deadline. **No late submissions will be accepted!**

9 Accessibility

Figure 1 Show the class diagram of the seven classes to be implemented in this practical:

- Task<T>
- List<T>
- Todo<T>
- WeeklyTasks<T>
- PriorityList<T>
- History<T>
- TaskManager<T>

The member variables and functions are discussed in Section 4. The following relationships exists between the classes:

- Todo, WeeklyTasks, PriorityList, and History inherits from List.
- Todo, WeeklyTasks, PriorityList, and History have a relationship with TaskManager.
- List has a relationship with Task.
- Todo has a relationship with Task.
- WeeklyTasks has a relationship with Task.
- PriorityList has a relationship with Task.
- History has a relationship with Task.
- Task has a relationship with itself.

The doTasks in the WeeklyTasks class should have the following format:

```
Task:_(description)_Data:_(data)_COMPLETED\n
```

The doTasks and doNTasks in the PriorityList class should have the following format:

```
Task:_(description)_Priority:_(priority)_Data:_(data)_COMPLETED\n
```

The doTasks and undoLatest in the History class should have the following format:

```
Task:_(description)_Priority:_(priority)_Data:_(data)_UNDID\n
```