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

Practical 7 Specifications: Linked Lists

Release Date: 10-10-2022 at 06:00

Due Date: 16-10-2022 at 23:59

Total Marks: 110

Contents

1 General instructions

- This practical should be completed individually, no group effort is allowed.
- You may not include/import any other modules than what was already included in the given files
- If your code does not compile you will be awarded a mark of zero. Only the output of your program will be considered for marks.
- You are not allowed to plagiarise.
- You will be afforded 10 upload opportunities.

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to `https://www.up.ac.za/students/article/2745913/what-is-plagiarism`. **If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding.** Also note that the OOP principle of code reuse does not mean that you should copy and adapt code to suit your solution.

3 Introduction

3.1 Scenario

For this practical, you will take the role of a backend developer at an IT company. You were given a ticket to implement a calendar interpretation system from the data returned by the remote API (API stands for application programming interface, which is a simple way for developers to retrieve data/run remote functions from servers. It is not important for you to understand an API for this practical). The data you receive from the API is read and sent to you as a vector of data. You will need to interpret the data as events and store it in an easy-to-use way for the front-end developers. After some thought, you decided that implementing a linked list to store and manage the events would be best because of how easy it is to insert and remove data randomly within it. You also decide that sorting the data according to the start time would be better for the sake of efficiency.

Since you are an experienced developer you also decide to use some code from an old project as a library to make the development of this system a bit easier(Utils.h).

3.2 Valgrind

As a software developer, it is your job to ensure you write code that is both efficient and memory safe. Especially since it is meant to be used on a hosted server and memory could be fatal for the system. Thus for this practical, your submission will be tested for memory leaks. An easy way to check if your program has memory leaks is with a tool called Valgrind. For Linux users you would need to run the following command to install valgrind:
sudo apt-get install valgrind
Then you need to edit your makefile run command to run with valgrind instead of the plain run. To do this you can simply change your run command to execute through valgrind instead. To do this you will replace need to **"./main"** with **"valgrind –leak-check=full ./main"**
If you have a memory leak, valgrind will print a leak summary that will look something like:

```
LEAK SUMMARY:
definitely lost: 48 bytes in 1 blocks
indirectly lost: 0 bytes in 0 blocks
```

If you do have memory leaks you are not deleting memory that you have allocated.

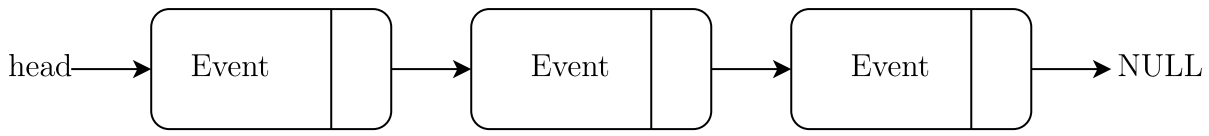
3.3 Mark Distribution

Activity	Testing	Mark Possible
Task 1	Utils	5
Task 2	Event	24
Task 3	Calendar	80
Task 4	Valgrind	1
Total		110

4 Your Tasks

A linked list is a basic and very dynamic data structure that is used to implement other data structures such as Binary Trees and Skip Lists. Your task will be to implement your own version of a linked list data structure. In your implementation, an "Event" is a node inside your linked list and "Calendar" is your linked list manager class.

The linked list data structure makes use of storing objects on the heap that point to each other. In your case, you will have "Event" nodes on the heap pointing to other nodes on the heap. It's best to think about a linked list as a graph-like structure, where nodes can point to each other. The "Calendar" class will store the first (head) node that will point to the next node and that one will point to the next and so on and so forth until we reach a node that points to NULL. Which would be the end of the linked list. By default, the head is set to NULL to indicate that the list is empty.



For this practical you will be given:

- Utils.h
- Calendar.h
- Calendar.cpp
- Event.h
- Event.cpp

You will be coding inside every file given. You are also required to write your own testing file to ensure your code is working.

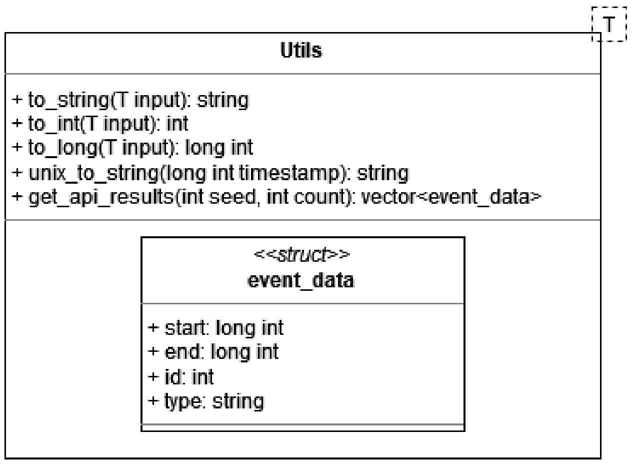
During development, you decided it would be best to mock the API data for easier development. Thus the function "get_api_results" will generate random API results for you to work with.

4.1 Unix Time Explained

The unix time stamp is a simple way for computers to track time. Using a long integer they count every second since 1970 Jan 1st. For example, the time "2022/10/1 12:30:30" would be expressed as the integer "1664620230". To experiment with this standard, you may use the site: `https://www.unixtimestamp.com/`

4.2 Task 1

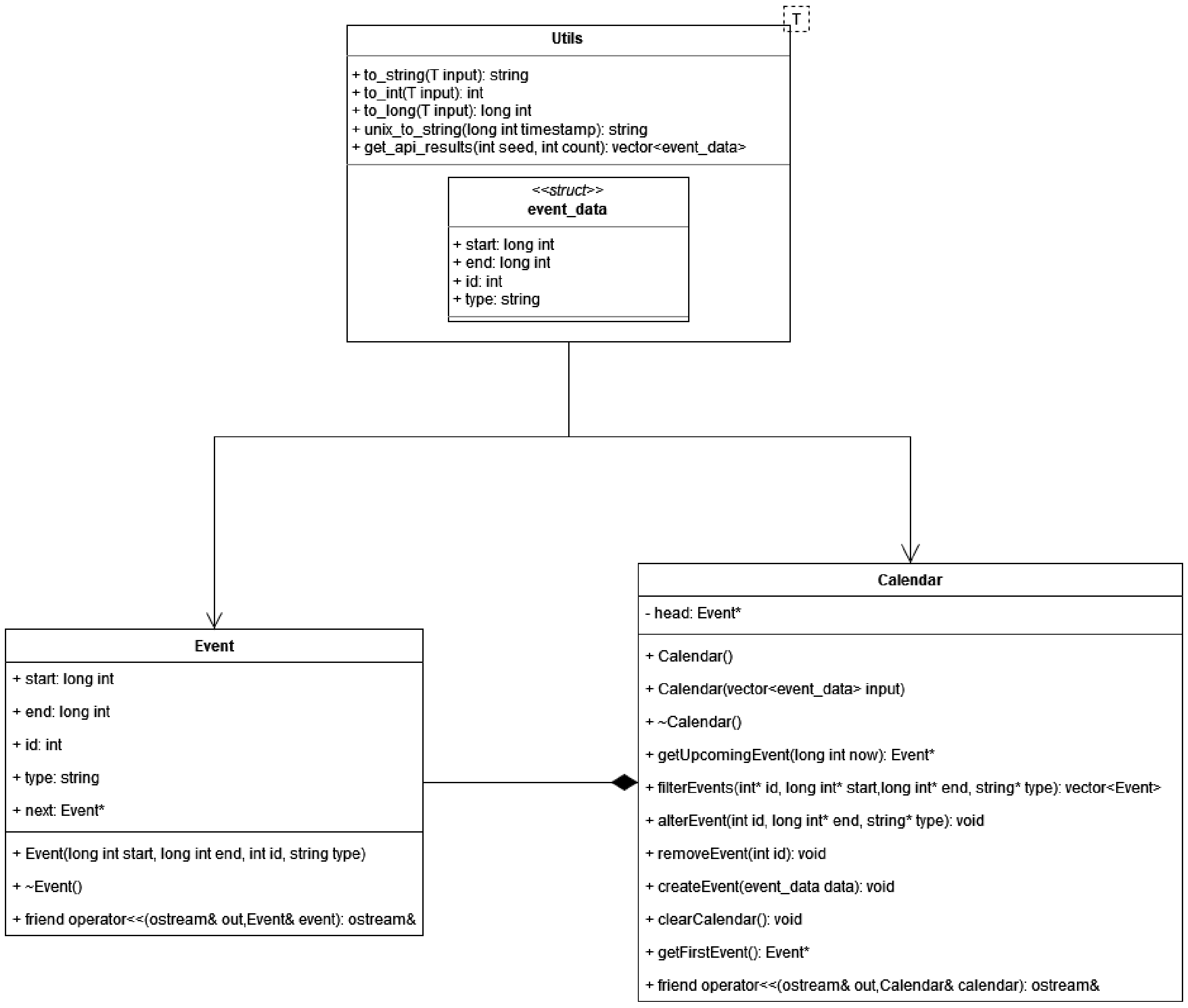
4.2.1 Utils.h


T
Utils
+ to_string(T input): string
+ to_int(T input): int
+ to_long(T input): long int
+ unix_to_string(long int timestamp): string
+ get_api_results(int seed, int count): vector<event_data>
<<struct>>
event_data
+ start: long int
+ end: long int
+ id: int
+ type: string

First you will need to implement the general to_string(T input), to_int(T input) and to_long(T input) functions inside the **"Utils.h"** file. **Note that Utils is not a class. It is a collection of static functions and an event_data struct.**

- **static string to_string(T input)**
  This is a template function that converts the given template input data into a string using stringstream
- **static int to_int(T input)**
  This is a template function that converts the given template input data into an integer using stringstream.
- **static int long to_long(T input)**
  This is a template function that converts the given template input data into a long using stringstream.
- **static string unix_to_string(long int timestamp)**
  This is a given function that will convert a given unix timestamp into a readable string.
- **static vector<event_data> get_api_results(int seed, int count)**
  This is a given function that will generate a vector of random event_data.
  Note: The random values generated will differ from platform to platform due to the way each platform calculates a random value.

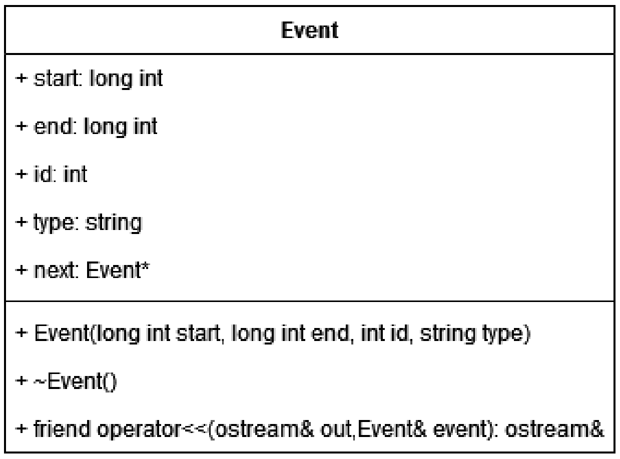
Final UML



T
Utils
+ to_string(T input): string
+ to_int(T input): int
+ to_long(T input): long int
+ unix_to_string(long int timestamp): string
+ get_api_results(int seed, int count): vector<event_data>
<<struct>>
event_data
+ start: long int
+ end: long int
+ id: int
+ type: string
Event
+ start: long int
+ end: long int
+ id: int
+ type: string
+ next: Event*
+ Event(long int start, long int end, int id, string type)
+ ~Event()
+ friend operator<<(ostream& out,Event& event): ostream&
Calendar
- head: Event*
+ Calendar()
+ Calendar(vector<event_data> input)
+ ~Calendar()
+ getUpcomingEvent(long int now): Event*
+ filterEvents(int* id, long int* start,long int* end, string* type): vector<Event>
+ alterEvent(int id, long int* end, string* type): void
+ removeEvent(int id): void
+ createEvent(event_data data): void
+ clearCalendar(): void
+ getFirstEvent(): Event*
+ friend operator<<(ostream& out,Calendar& calendar): ostream&

4.3 Task 2

4.3.1 Event



Event
+ start: long int
+ end: long int
+ id: int
+ type: string
+ next: Event*
+ Event(long int start, long int end, int id, string type)
+ ~Event()
+ friend operator<<(ostream& out,Event& event): ostream&

The Event class will act as both the node in our linked list and the data holder. Thus it will be storing data and a pointer to the next node.

- **Event(long int start,long int end,int id,string type)**
  The constructor must initialize all the members appropriately.
- **~Event()**
  The destructor will not do anything.
- **friend ostream& operator«(ostream& out,Event& event)**
  The overload of the stream extraction operator conveniently puts the formatted output into the ostream object it operates on. This function must insert the event type which is followed by the start and end. The start and end must be converted to string using the unix_to_string function. Do not add a newline at the end.
  So the formatting should be (Take note of added spaces and the split between start and end using '->'):

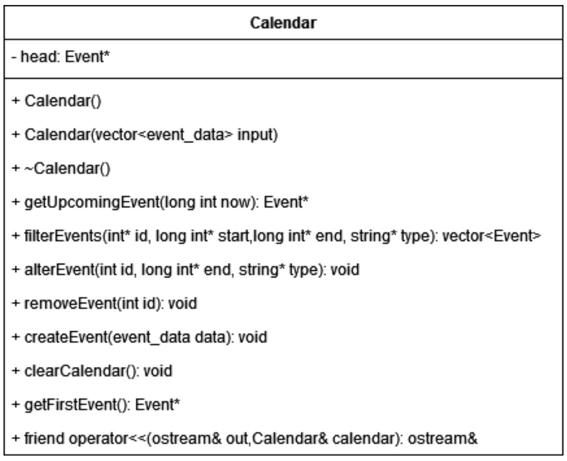
  ```
  {Type} unix_to_string( {Start} )->unix_to_string( {End} )
  ```

  Example of data filled in:

  ```
  Class Test 2025/08/29 01:17:34->2025/12/19 15:29:53
  ```

4.4 Task 3

4.4.1 Calendar



```
Calendar
- head: Event*
+ Calendar()
+ Calendar(vector<event_data> input)
+ ~Calendar()
+ getUpcomingEvent(long int now): Event*
+ filterEvents(int* id, long int* start,long int* end, string* type): vector<Event>
+ alterEvent(int id, long int* end, string* type): void
+ removeEvent(int id): void
+ createEvent(event_data data): void
+ clearCalendar(): void
+ getFirstEvent(): Event*
+ friend operator<<(ostream& out,Calendar& calendar): ostream&
```

The Calendar class is the manager for our linked list.

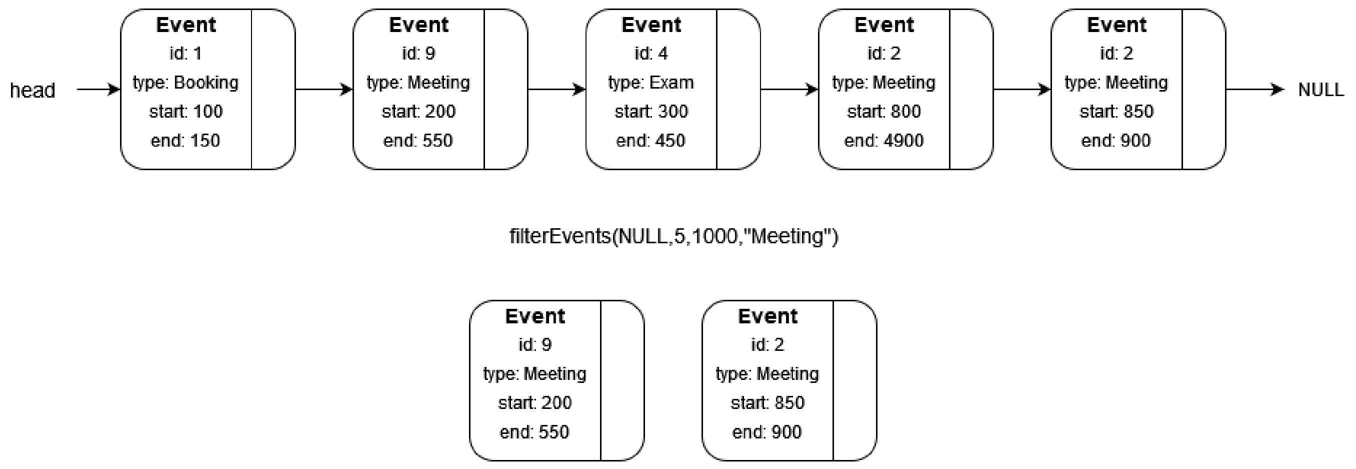
- **Calendar()**
  The default constructor only needs to set the head to NULL.

- **Calendar(vector<event_data> input)**
  The overloaded constructor first needs to set the head to NULL then it needs to iterate through the passed vector and call the *createEvent()* function for each even_data object.

- **~Calendar()**
  The destructor calls the *clearCalendar()* function.

- **Event* getUpcomingEvent(long int now)**
  This function will need to iterate through your linked list and return the first element whose start time is larger than the value passed. For example, if the list contains 2 items, the first starting in 1980 and the second starting in 2023. If the passed-in value is before 1980 (ie 1979) you must return the first element. If the passed now is after 1980 but before 2023 (ie. 2022) you must return the second one.
  Return NULL if the list is empty or if there does not exist an event whose start time is after the passed value.

- **vector<Event> filterEvents(int* id, long int* start,long int* end, string* type)**
  This function will return a vector of events whose members match the filters passed. If a parameter is not set to NULL you must use it as a filter for the events to add. For example, if the type and start parameter are not NULL you must return a vector of every event of that type after the starting time. If all parameters are null you must return a vector with all the events. If there doesn't exist any events with those parameters you must return an empty vector.



  In the above example, we asked that they filter all events that start after 5, end before 1000 and that are "Meeting" types. Thus only 2 nodes will be added to the vector.
  Note: When comparing start and end we compare before and after not at that time.

- **void alterEvent(int id, long int* end, string* type)**
  This function will need to alter the event whose id matches the passed id. You may assume only 1 node with that id will exist. If the end/type parameter is not NULL you must change the events end/type to the new end/type value. Do not delete the event while altering it.

- **void removeEvent(int id)**
  This function will need to remove the event of that id from the linked list.

- **void createEvent(event_data data)**
  This function will create a new event and place it inside the linked list. The location of the event inside the list is determined using the start time. The list must stay sorted by start time (oldest to newest). If 2 events have the same start time you must place the new event after the old one.

- **void clearCalendar()**
  This function should delete every event inside the linked list and set the head to NULL. Valgrind will be used to ensure all the Events are deleted properly.

- **Event* getFirstEvent()**
  This function returns the head pointer.

- **friend ostream& operator«(ostream& out,Calendar& calendar)**
  The overload of the stream extraction operator conveniently puts the formatted output into the ostream object it operates on. This function must print a counter alongside the event using the following formatting.

  ```
  { counter }: { Event }{ newline }
  ```

  Example with data filled in:

  ```
  0: Testing 1970/01/01 00:00:01->1970/01/01 00:00:02
  1: Booking 2022/09/27 14:36:41->2022/12/16 19:52:51
  2: Meeting 2023/01/03 20:01:35->2023/02/22 05:53:37
  3: Semester Test 2023/02/06 21:53:55->2023/04/12 03:39:16
  ```

5 Submission

Your code must be able to compile and run using the c++98 compiler.

You need to submit your source files on the Fitch Fork website (https://ff.cs.up.ac.za/). Only the following files should be in a zip archive named uXXXXXXXX.zip where XXXXXXXX is your student number:

- Utils.h
- Event.h
- Event.cpp
- Calendar.h
- Calendar.cpp

Ensure that your archive is correct and that your files are not within another folder within the archive.
For this practical, you are not allowed to use any other includes than the ones provided.

Allowed includes:

- string
- iostream
- sstream
- vector
- cstdlib

You have 10 submissions and your best mark will be your final mark. Upload your archive to the Practical 7 slot on the Fitch Fork website. **No late submissions will be accepted!**