COMP 2012 Object-Oriented Programming and Data Structures

Lab 5 Template and Operator Overloading

Menu

- Introduction
- Download
- Lab Tasks
- Submission Guidelines

Page maintained by

TANG, Mingkai
Last Modified: 11/03/2022 09:32:40

Homepage

Course Homepage

Introduction

In lab 5, we will implement a sorted sequence that can support different data types. `int`, `char`, and a self-defined type will be used for testing. You can practice template class and operator overloading in this lab. You can download the skeleton code HERE.

Background

Imagine that you are a teacher in a kindergarten. You want to line up the kids in your class **from higher priority to lower priority**. The priority is defined as follows:

1. Shorter kids have a higher priority than taller kids.
2. If two kids have the same height, girls have a higher priority than boys.
3. If two kids have the same height and gender, the kid whose name is lexicographically smaller has a higher priority
4. If two kids have the same height, the same gender, and the same name, the heavier kid has a higher priority.
5. If all criteria above are identical, they have the same priority. It means these kids can be sorted in arbitrary order.

The concept of **lexicographically smaller** is defined as follows. Given two strings `A = a_0 a_1 a_2 a_n ` and `B = b_0 b_1 b_2 ... b_m`, the string `A` is lexicographically smaller than `B` if and only if one of the following two conditions is satisfied:

1. There exist an integer `k`, such that `k le n` and `k le m ` and `a_0=b_0, a_1=b_1...a_{k-1}=b_{k-1}, a_k < b_k`.
2. ` n < m ` and `a_0=b_0, a_1=b_1...a_{n-1}=b_{n-1}, a_n=b_n`.

For example, the string "Alice" is lexicographically smaller than "Bob" and "Tom" is lexicographically smaller than "Tommy".

After you finish planning the queuing order, unfortunately, some new children may join the class, and you need to plan again.

Therefore, you want to implement a class `SortedSequence` to maintain a sorted sequence dynamically. In the beginning, the sequence is empty. The class needs to support two operations:

- Add a new element to the sequence.

- Output all elements in the sequence.

Enum, Function, and Class Introduction:

This section will introduce each enum, function, and class in detail.

Enum Gender

```
enum Gender {
  Female,
  Male
};
```

This is defined in `kid.h`. We use this enum to represent the gender of the kid.

Class Kid

```
class Kid {
  public:
    Kid();
    Kid(int height, Gender gender, const char *name, int weight);
    Kid(const Kid &kid);
    ~Kid();

    // Operator functions
    Kid& operator=(const Kid& other);
    bool operator<(const Kid& other) const;

    // Accessors
    int height() const;
    Gender gender() const;
    const char* name() const;
    int weight() const;

  private:
    int height_;
    Gender gender_;
    char* name_;
    int weight_;
};
```

This class is defined in `kid.h/cpp`. Class `Kid` stores some information about a kid. In addition, it has two operator functions:

```
Kid& operator=(const Kid& other);
```

This function performs a deep copy of `Kid`. It has been implemented in the code.

```
bool operator<(const Kid& other) const;
```

You need to implement this function in Task 1. It checks whether 'this' kid has a higher priority than the 'other' kid. If yes, return true; otherwise, return false.

Class SortedSequence

```
template <typename T>
class SortedSequence{
  public:
    SortedSequence();
    ~SortedSequence();

    void add(const T &a);

    std::string toString() const;

  private:
    T *data_;
    int capacity_;
    int size_;
};
```

This class is defined in `sorted_sequence.h`. It maintains a sorted sequence.

- `data_`: This pointer points to the memory space.
- `capacity_`: This variable represents the maximum number of elements stored in the current memory space.
- `size_`: This variable represents the current number of elements of the sequence.

In the beginning, we don't know how much space needs to be allocated to `data_`. We will use a special mechanism to store elements incrementally, like the `vector` in the STL library, which we will learn later in this course. More details of the mechanism will be shown in Task 2.

```
void add(const T &a);
```

This function adds a new element to the sequence and maintains the sequence in a sorted status. You need to implement it in Task 2.

```
std::string toString() const;
```

This function transforms a `SortedSequence` instance to a string. It has been implemented in the code. The return string can be used to output the information without directly accessing the data member of `SortedSequence`.

Other Functions

```
template <typename T>
std::ostream& operator<<(std::ostream& os, const SortedSequence& t);
```

This function is implemented in `sorted_sequence.h`. Using this function, we can have a convenient way to print out the information of `SortedSequence`, like `std::cout << sorted_sequence << std::endl;`

```
template <typename T>
std::ostream& operator<<(std::ostream& os, const Kid& k);
```

This function is implemented in `kid.h/cpp`. It can output the information of `Kid` by the statement like `std::cout << kid << std::endl;`.

Lab tasks

You can download the skeleton code HERE.

Task 1

```
bool operator<(const Kid& other) const;
```

Implement this operator function of the class `Kid` in `kid.cpp`. The return boolean should be whether 'this' kid has a higher priority than the 'other' kid.

Hint: You can implement the function based on this scheme in order:

1. If two kids have different heights, return whether 'this' kid is **shorter** than the other kid.
2. If two kids have different genders, return whether 'this' kid is female.
3. If two kids have different names, return whether 'this' kid's name is lexicographically smaller than the 'other' kid's name.
4. If two kids have different weights, return whether 'this' kid is heavier than the 'other' kid.
5. Otherwise, return false.

You can use `strcmp()` to compare two C-string. The usage of `strcmp` can be found in this LINK.

Task 2

```
void add(const T &a);
```

Implement this member function in `sorted_sequence.h`. In the test case, `int`, `char`, and `Kid` variables will be used to test this function.

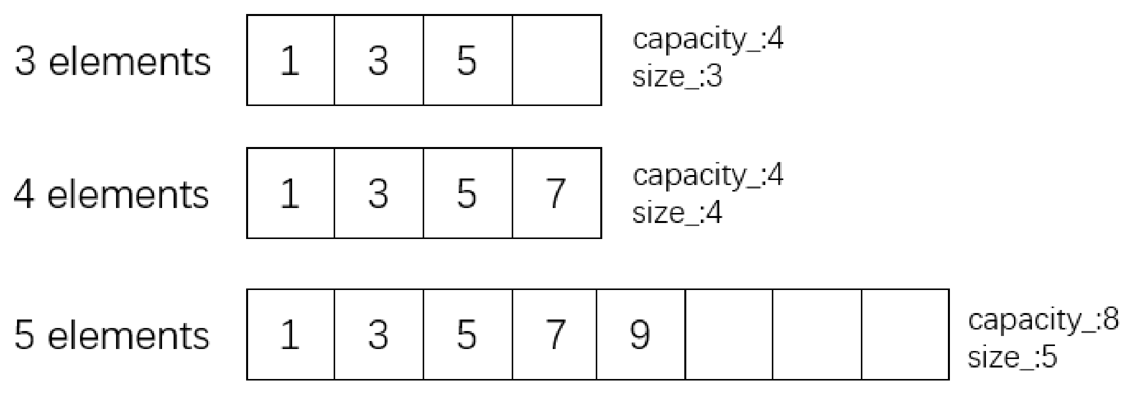
Initially, the sequence is empty. `data_` is a `nullptr`, `capacity_ = 0`, `size_ = 0`.

The function can be split into two phases, the memory allocation phase, and the element insertion phase.

In the memory allocation phase, you need to check whether the space of `data_` allows for adding a new element. If yes, go to the element insertion phase directly. Otherwise, allocate new space for `data_`. There are two cases:

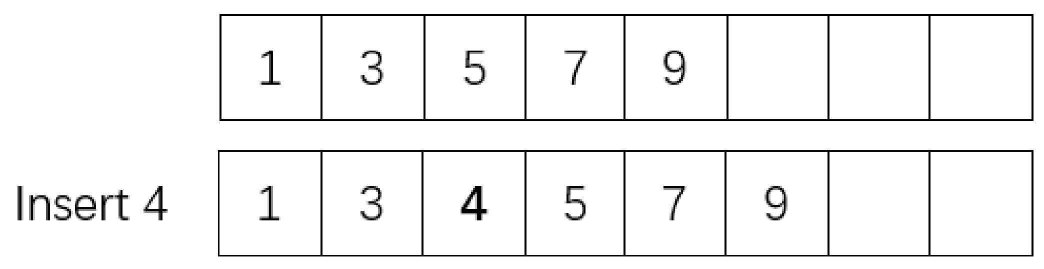
- If `data_` is a `nullptr`, allocate 1 space to the `data_` and `capacity_ = 1`.
- If `data_` is not a `nullptr`, allocate the space of `2 * capacity_`. Copy the content from the old space to the new space. Delete old space. Update `capacity_ = 2 * capacity_`.

Here are examples of the memory space of a 3-element sequence, a 4-element sequence, and a 5-element sequence (The figure assumes that the `size_` has been updated. The update of `size_` will be described in the element insertion phase):



After the memory allocation phase, the memory is enough for the insertion.

In the element insertion phase, We need to insert the elements into the sequence at the proper position such that the sequence remains in order. In addition, Update `size_` to `size_ + 1`. Here is an example:



Expected Output

The expected output is shown in `test_1_expected_output.txt` and `test_2_expected_output.txt`.

Submission and Grading

Deadline: 10 minutes after your lab session, Nov 3/4, 2022.

Please submit the following files to ZINC by zipping the following 2 files. ZINC usage instructions can be found here.

- `kid.cpp`
- `sorted_sequence.h`

There are 3 test cases in total. Test case 3 is the same as test case 2, respectively, except that test case 3 checks memory leakage.

Notes:

- You may submit your file multiple times, but only the last submission will be graded. **You do NOT get to choose which version we grade.** If you submit after the deadline, a late penalty will be applied according to the submission time of your last submission.
- Submit early to avoid any last-minute problems. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day, especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we will grade your latest submission with all test cases after the deadline.
- In the grading report, pay attention to various errors reported. For example, **under the "make" section, if you see a red cross, click on the STDERR tab to see the compilation errors.** You must fix those before seeing any program output for the test cases below.
- Make sure you submit the correct file yourself. You can download your file back from ZINC to verify. Again, **we only grade what you uploaded last to ZINC**.

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online auto-grader ZINC. It won't be graded if we cannot compile your work. Therefore, for parts you cannot finish, just put in a dummy implementation so that your whole program can be compiled for ZINC to grade the other parts you have done.

Change Log

22:30 - 28/10/2022 :

- Modify step 1 of the hints in Task 1 from "If two kids have different heights, return whether 'this' kid is **taller** than the other kid" to "If two kids have different heights, return whether 'this' kid is **shorter** than the other kid".
- Add some comments for Windows users in the makefile, including in `create_directory:` and `clean:`. It does not affect your code.

9:30 - 03/11/2022 :

- Fix the bug in the makefile that cannot detect the modification of the ".h" files. Change from `-include $(DEPS)` to `-include $(OBJ_DIR)/$(DEPS)`. It does not affect your code.