Acacia Solutions

Introduction to C++ for C Programmers

Class labs and Homework exercises

Alexander Katz
11/08/2019

Table of Contents

Day 1. Introduction 3
Class Lab 1A – Stack of <int> of predefined capacity 3
Homework 1B – Fractions 3
Homework 1C – Stack of Rational (fractions) 4
Day 2. Memory Management 5
Class Lab 2A – Dynamic Stack of int 5
Homework 2B – Dynamic Stack of Rational objects 5
Homework 2C – Queue using DynamicStack 5
Day 3. Value Semantics and Operator Overloading 6
Class Lab 3A – Value semantics for Dynamic Stack 6
Class Lab 3B – Operator overloading for **Rational** class 6
Homework 3C – Dynamic String class (phase 1) 6
Day 4. More Operator Overloading; Static members 7
Class Lab 4A – Add **compare** Function to **String** class (phase 2) 7
Homework 4B – Dynamic String class (phase 3) 7
Day 5. Templates 8
Class Lab 5A – Convert **Stack** class into Template 8
Homework 5B – Generic **RingBuffer** (template) 8
Day 6. STL Containers 9
Class Lab 6A – Convert vector operations into template 9
Homework 6B – Design and implement **BiDirectionalMap<K,V>** 9
Day 7. Basic Design 10
Class Lab 7A – Design a Vending Machine 10
Homework 7B – Design a Book Catalogue 10
Day 8. Private Inheritance 11
Class Lab 8A – Sorted Vector 11
Homework 8B – Sorted List 11
Day 9. Polymorphism I 12
Lab 9A – Class Hierarchies 12
Homework 9B – Text Manipulator 12
Day 10. Exceptions 13
Lab 10 – Add Exceptions to **Stack** 13
Day 11. Interfaces (Polymorphism II) 14
Lab 11A – Design book catalog 14
Homework 11B –Drawing Images 14
Day 12. Object Model & Multiple Inheritance 14
Day 13. Final Project 15
Project A - Book Catalogue 15
Project B – Breakout Game 16

Day 1. Introduction

Class Lab 1A – Stack of <int> of predefined capacity

Define and implement a Stack container of predefined capacity (64), which holds `int` items. Define the `Stack64` class, implement it and write Unit Test.

C++ coding conventions should be followed and enforced (peer-review).

The `Stack64` class should support the following methods (member functions) :
CTOR & DTOR (Do we need them? Why?)

- Operations
  - `bool push(int)` add an item to the Stack instance
  - `bool pop(int *)` remove an item from the top of Stack
  - `bool top(int*)` peek at item at the top of Stack without removing it
- Information
  - `bool isFull()` returns true if the Stack instance is full
  - `bool isEmpty()` returns true if the Stack instance is empty
  - `size_t size()` returns the number of items in the Stack instance
  - `size_t capacity()` returns the capacity of the Stack instance

The Unit Test should comprise (at least) the following test cases:

- `Stack64` CTOR and DTOR
- Test LIFO (Last-In-First-Out) policy
- Test border cases : overflow and underflow

The exercise is to be performed in three steps:

1. Define and code-review the `Stack64` interface (Stack64.h file) - 15 min
2. Implement the class (Stack64.cpp) file - 30 min
3. UT and debug (testMU.cpp) - 1.5 hour

Homework 1B – Fractions

Define and implement a simple `Rational` class representing fractions of two integers: numerator and denominator. The Rational class should support the following functionality:

- Default CTOR, CTOR from (`int`), CTOR from (`int`, `int`)
- Operations
  - `mul(int)` multiply two fractions
    `mul(Rational)`
    - Same for add, subtract and divide operations
  - `reduce()` to lowest common denominator
- Information
  - `print()` print the fraction as rational number : {`int`, `int`}

Notes:

The Unit Test should be comprehensive. **`Rational`** class will be used in follow-on exercises. Same class declaration should be used to facilitate exchange of Unit Tests

Homework 1C – Stack of Rational (fractions)

Define and implement a simple Stack container of predefined size that stores **`Rational`** items.

- Default CTOR
- Operations
  - **`push()`** push one Rational
  - **`push()`** push array of Rationals
  - **`pop()`** pop Rational from Stack
  - **`top()`** peek at the top of the Stack
- Information
  - **`isFull()`** and **`isEmpty()`**
  - **`size()`** returns current number of items
  - **`print()`** prints the current stack
  - **`trace()`** prints the full contents of the stack (for debug)

Day 2. Memory Management

Class Lab 2A – Dynamic Stack of `int`

Define and implement a Stack container of specified capacity, which holds `int` items. The stack item array is dynamically allocated on heap. Use `Stack64` class as base for `DynamicStack` class, implement and unit-test it.

The `DynamicStack` class should support at least the following methods (member functions):

- CTOR, DTOR
- Operations
  - `bool push(int)` add an item to the Stack instance
  - `bool pop(int &)` remove an item from the top of Stack
  - `bool top(int &)` peek at item at the top of Stack without removing it
- Information
  - `bool isFull()` returns true if the Stack instance is full
  - `bool isEmpty()` returns true if the Stack instance is empty
  - `size_t size()` returns the number of items in the Stack instance
  - `size_t capacity()` returns the capacity of the Stack instance

Modify Unit Test accordingly and add new tests (at least)

- Test copy assignment operator

The exercise is to be performed in three steps:

1. Define and code-review the `DynamicStack` interface (DynStack.h file) – 15 min
2. Implement the class (DynStack.cpp) file - 30 min
3. UT and debug (testMU.cpp) - 1.5 hour

Homework 2B – Dynamic Stack of Rational objects

Create a `DynamicRationalStack` of `Rational` (fraction) objects. The Stack will start with initial capacity and will grow on demand by a given factor, defaulted to 1.5

Use `DynamicStack` from class lab as a base and modify accordingly: the class declaration, definition (implementation) and unit-test. Add the following methods:

- `drain()` move items from the Stack to another one
- `copy()` copy items from the Stack to another one

Homework 2C – Queue using DynamicStack

How to implement a Queue using Stack class?

The Queue constructor should specify the size of queue. Make sure to keep the semantics of all information methods: `size(), capacity(), isFull()` and `isEmpty()`.

Day 3. Value Semantics and Operator Overloading

Class Lab 3A – Value semantics for Dynamic Stack

Modify **`DynamicRationalStack`** (homework) to correctly implement value semantics, i.e. implement the Copy Constructor (CCTOR) and Copy Assignment operator op=() to correctly perform allocation and copy of items array.

- CTOR, DTOR, CCTOR, op=()
- Keep **`drain()`** method

Modify Unit Test accordingly and add new tests

- Test copy CCTOR and assignment operator
- Test value semantics

Class Lab 3B – Operator overloading for **`Rational`** class

Amend **`Rational`** (from Day 1 homework) to include the following operators:

- Members
  - op+=, op-=, op*=, op/=
  - op<, op>, op==, op!=
- Global operators
  - op+, op-, op*, op/
  - op<<

Amend unit-test to test the operators.

Homework 3C – Dynamic String class (phase 1)

Implement a dynamic String class that will dynamically allocate memory to hold characters.

- Default CTOR, CTOR(const char*)
- CCTOR, DTOR and op=
- Members
  - **`const char* get()`** return pointer to character string
- Global
  - **`int compare(…)`** compare two Strings in C fashion (strcmp)
  - **`operator<<`** stream op<< for printing the string
- Information
  - **`size_t length()`** returns the number of characters in the String instance

Use the provided header file, and update it as needed.
Add new tests to verify the functionality of the **`String`** class.

Day 4. More Operator Overloading; Static members

Class Lab 4A – Add `compare` Function to `String` class (phase 2)

Modify `String` (homework) to compare both case-sensitive and case-insensitive, according to settings. Add setting comparison mode methods.
In addition keep statistics about number of `String` objects currently used.

- Comparison mode
  - `bool isCaseSensitive()` return current comparison sensitivity
  - `void setCaseSensitive()` set new comparison sensitivity mode
  - `int compare(…)` compare two Strings according to current mode
  - `operator<(…)` overload comparison operator less-than
- Information
  - `size_t objectCount()` returns the number of current String objects

Use the provided header file, and update it as needed.

Modify Unit Test accordingly and add new tests

- Test setting sensitivity mode
- Verify number of current objects

Homework 4B – Dynamic String class (phase 3)

Modify `String` (phase 2) to add the following functionality:

- Concatenation operations to support the following expressions:
  - `String += String`
  - `c = String + String`
  - `c = "string" + String`
  - `c = String + "string"`
- Substrings and Subscripts
  - `operator()(size_t a_start, size_t a_leng)`
    - Returns a substring starting at `start` position and has a length of `leng`, e.g.
    - `String str = "123456789"; str(4,3)` returns string "567"
  - `bool contains(String const&)`, which returns true if a substring is found
  - `operator[size_t i]`, which returns i'th character in a `String`
  - Find the first and last occurrence of a character in the `String`
- Static member and mem-functions (get/set) to control allocation alignment of the `String` data. Allocations should be a multiple of this value (default = 32)
- Find (and return) a first duplicate character in the `String`. Return 0 in case all characters in are unique.

Use the provided header file, and update it as needed.
Add new tests to verify **all** the functionality of the `String` class.

Day 5. Templates

Class Lab 5A – Convert `Stack` class into Template

Modify `Stack` (fixed size stack Day 1 homework) to template class.

- CTOR, DTOR, CCTOR and op=; global op<<
- Operations
  - **push** add an item to the Stack
  - **pushAll** add array of items to the Stack
  - **pop** remove an item from the top of Stack
  - **top** peek at item at the top of Stack without removing it
- Information
  - **`isFull`** returns true if the Stack instance is full
  - **`isEmpty`** returns true if the Stack instance is empty
  - **`size`** returns the number of items in the Stack instance
  - **`capacity`** returns the capacity of the Stack instance
- Operations
  - **`drain`** transfer items to another stack

Steps:

1. Create and review `Stack` class template definition – 30 min
2. Implement functions – 1 hour
3. Modify Unit Test accordingly and add test with integers and `Rational` objects.

Homework 5B – Generic `RingBuffer` (template)

Implement and unit-test a generic `RingBuffer` class, a.k.a. circular queue. The `RingBuffer` should be of constant capacity, defined at creation.

- CTOR, DTOR, CCTOR and op=; global op<<
- Operations
  - **enqueue** add an item to the `RingBuffer`
  - **dequeue** remove an item from the top of `RingBuffer`
  - **front** peek at item at the front of `RingBuffer` without removing it
- Information
  - **`isFull/isEmpty`**
  - **`size/capacity`**
- Utility
  - **`print/trace`** print the current/all queue items accordingly

Create UT with different item types to store in the buffer.

Day 6. STL Containers

Class Lab 6A – Convert vector operations into template

Write and unit-test the following functions, using standard C++ container library. Demonstrate use of traversal, using both subscript operator and iterators.

- **`VectorFillWithSquares`** – fill a vector<T> with squared values of T
- **`VectorPrint`** – print vector<T> items. What <T> should support?
- **`VectorSelect(in, out, predicate)`** - move items that match the predicate to another container. Implement with both predicate function and function object
- **`FirstDuplicate(vector<T>)`** – find first duplicate object in a vector.
  - What <T> should support?
  - What is the run-time and space complexity of your solution?

Each function should be tested on more than one variant of <T>.

Homework 6B – Design and implement `BiDirectionalMap<K,V>`

C++ **`std::map`** container supports mapping of Key to Value. In other words, the user can find and access a Value via a Key.
`BiDirectionalMap<K,V>` should support both type of access:

- Given a Key, return Value
- Given a Value, return Key

There way **`std::map`** methods and operators work, might affect the bi-directional map. Try to create a list of such possible issues and deal with them in design and implementation.

At least the following mem-functions should be implemented:

- CTOR, DTOR, etc
- Operations
  - **insert** add a <K,V> pair to the **`BiDirectionalMap`**
  - **remove** remove an <K,V> pair from the **`BiDirectionalMap`**
  - **atKey** given a Key – return the Value
  - **atValue** given a Value – return the Key
- Information
  - **`empty`**
  - **`size`**

Bonus Questions:

- Can we implement iterators to **`BiDirectionalMap`**? How>
- What type of iterators? Why?

Day 7. Basic Design

Class Lab 7A – Design a Vending Machine

Design a vending machine for soft drinks. Follow the steps:

1) Create two lists:
   a) Different parts (components)of a vending machine
   b) Various scenarios of working with a vending machine
2) Prepare answers to questions about your lists:
   a) Nouns – what they represent?
   b) Verbs – what they represent?
   c) Who are the actors?
   d) What are the relations between various Nouns in your list>
   e) What are the relations between various Nouns and Verbs in your list?
3) Prepare class diagram (block-diagram)
   a) Draw block-diagrams on a piece of paper.
   b) Prepare a story about your vending machine and its design

Homework 7B – Design a Book Catalogue

Design a Catalogue system for books, as follows:

- Each book has one author, one publisher, year of publishing and ISBN (key)
- Operations on a Catalogue:
  - Find book by author
  - Find book by publisher
  - Find book by word/words in title

Prepare class diagram. You may use one of the following cloud-based tools:

- Draw.io – https://www.draw.io/
- LucidChart – https://www.lucidchart.com
- Visual Paradigm – https://online.visual-paradigm.com/

Deliverables:

1. Class-diagram PDF file
2. Prepare a story to present your design in class

Day 8. Private Inheritance

Class Lab 8A – Sorted Vector

Implement a sorted integer vector by extending `std::vector<int>`.

- Prepare `SortedVector` class design for code-review in class.
  - Consider which operations on sorted vector may break the sorting.
- Required functionality:
  - insert
  - remove
  - `iterator` / `const_iterator`
  - front / back
  - size
  - operator[]

Implement and unit-test the implementation

Homework 8B – Sorted List

- Complete the implementation for `SortedVector` template as homework.
- Design and implement `SortedList` class template by inheriting from `std::list`.
- Implement a function that can use either (`SortedVector` or `SortedList`) container to store randomly generated numbers, as well as `std::vector` and `std::list` .

Day 9. Polymorphism I

Lab 9A – Class Hierarchies

1. Implement a common base for **`SortedVector`** and **`SortedList`** – **`SortedContainer`**.
2. Implement a function that can use either sorted container to store randomly generated values.
    a. Design and implement random number generation hierarchy. Implement range (to, from) random number generator.
3. Design a **`Messager`** class using Encode hierarchy with **`UppercaseEncoder`** and **`LowercaseEncoder`** derived classes.

Homework 9B – Text Manipulator

Implement a text manipulator application that can read a message (a collection of text lines) either from file or interactively from console, apply various transformations to it and output the result to either file or console.

Transformations should include: Uppercase, Lowercase, Rot13

Note:
Message transformations should be based on class lab 3 above.

Day 10. Exceptions

Lab 10 – Add Exceptions to `Stack`

Add exceptions to `Stack` class:

- Overflow thrown when `Stack` is full
- Underflow thrown when `Stack` is empty

Stack exceptions should be based on STL `std::runtime_error`.

Modify Unit-Test to catch only known exceptions.

Day 11. Interfaces (Polymorphism II)

Lab 11A – Design book catalog

Design a book catalog with Indeces and Parsers.

Data File Format

The cataloge input file has the following format.

- First line contains column headers
- Eeach subsequent line is a record of one book with the following fields separated by '|' character :
  - ISBN – unique identification of the book
  - Book Title – string of characters, including white space
  - Book Author – in case of several authors, only the first is provided
  - Year of publication
  - Publisher

In-Memory Representation

The application will parse and store in memory a compact representation of the books catalog. Publishers and Authors will be stored only once in the memory, although they may appear in multiple books.

Homework 11B –Drawing Images

- Implement an Image (matrix of pixels) class, based on NetPBM format. See https://en.wikipedia.org/wiki/Netpbm_format.
- Implement **`Shape`** hierarchy, with methods to draw themselves on an **`Image`**
  - **`VerticalLine`**
  - **`HorisontalLine`**
  - **`Rectangle`**
  - **`Square`**
  - **`Circle`**
- Create an application that can read and display a PBM file (use P3 format)
- Create an application that can add shapes to an image

Day 12. Object Model & Multiple Inheritance

- Continue work on Homework 11B.
- Refactor the class hierarchy to support and use **`IDrawable`** and **`IDrawingSurface`** interfaces.

Day 13. Final Project

Two possible projects: Book Catalogue and Breakout Game.
The choice of the specific project is left to the Mentor decision.

Project A - Book Catalogue

You will implement a book catalogue search engine. The catalogue will be built by parsing and loading data files in textual format. Nonetheless, the design should allow future extension to load data from various sources.

Catalogue Input Data File Format

For the test you will be given a text file with the following format:
First line contains column headers, each subsequent line is a record of one book with the following fields separated by '|' character.

- ISBN – unique identification of the book
- Book title – string of characters, including white space
- Book Author – in case of several authors, only the first is provided
- Year of publication – string
- Publisher

The application will load, parse and create in-memory representation of the books catalog. Publishers and Authors will be stored only once in the memory, although they may appear in multiple books. The data structures used should be optimized for space and time to execute the queries described below.

Queries

User can query the catalog using one of the following:

1. Search for a book by ISBN
2. Search for books by title or part of it, where part is a complete word or words.
3. Optionally specify a word or words that shall not be present in the title, by prefixing it with a minus ('-') character.

Deliverables

- Class Design Document
- Source code of the application, including
  - Makefile + Unit test for cardinal components
  - Interactive Query application

Implementation Steps

1. Class Design and Design Review (peer and in-classroom)
2. Implement cardinal components, like Container(s), Indeces, Query, etc and unit-test them
3. Refactor class design to use Interfaces like IQuery, IIndex, etc
4. Book catalogue main program
5. Prepare project presentation: documents and story script

Project B – Breakout Game

The Breakout Game (https://en.wikipedia.org/wiki/Breakout_(video_game)) is one of the first arcade games developed and published by Atari. You can play a modern on-line version at https://www.gamesxl.com/breakout.

You will create breakout game, using SFML (Simple and Fast Multimedia Library). Use the following link to access the code, documentation and tutorials: http://www.sfm-dev.org

Project Definition

Design and implement a simple breakout bricks game and expand it to include different types of bricks.

Deliverables

- Class Design Document
- Source code of the application, including
  - Makefile
  - Interactive application

Implementation Steps

1. Class Design and Design Review (peer and in-classroom)
2. Phase 1 – Implement a bouncing ball
3. Phase 2 – Extend design to include bricks and collision detection
4. Phase 3 - Expand design to include additional types of bricks: exploding, solid, needs multiple hits, etc
5. Prepare project presentation: documents and story script