Where Are We?
Data Abstraction
Introduction to Sequences
Interface for a Smart Array

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, October 12, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Some material contributed by Chris Hartman

Unit Overview
Algorithmic Efficiency & Sorting

Major Topics

- ✓ Analysis of Algorithms
- ✓ Introduction to Sorting
- ✓ Comparison Sorts I
- ✓ Asymptotic Notation
- ✓ Divide and Conquer
- ✓ Comparison Sorts II
- ✓ The Limits of Sorting
- ✓ Comparison Sorts III
- ✓ Non-Comparison Sorts
- ✓ Sorting in the C++ STL

DONE

Where Are We?

Where Are We?
From the First Day of Class: Course Overview — Goals

Upon successful completion of CS 311, you should:

- Have experience writing and documenting high-quality code.
- Understand proper error handling, enabling software components to support robust, reliable applications.
- Be able to perform basic analyses of algorithmic efficiency, including use of big-*O* and related notation.
- Be familiar with various standard algorithms, including those for searching and sorting.
- Understand what data abstraction is, and how it relates to software design.
- Be familiar with standard container data structures, including implementations and relevant trade-offs.

Primary goals to be addressed for the rest of the semester

We will also discuss **this** further.

Where Are We?
From the First Day of Class: Course Overview — Topics

The following topics will be covered, roughly in order:

- Advanced C++
- Software Engineering Concepts
- Recursion
- Searching
- Algorithmic Efficiency
- Sorting

DONE

- **Data Abstraction**
- **Basic Abstract Data Types & Data Structures:**
  - **Smart Arrays & Strings**
  - **Linked Lists**
  - **Stacks & Queues**
  - **Trees (various kinds)**
  - **Priority Queues**
  - **Tables**
- Briefly: external data, graph algorithms.

Goal: Practical generic containers

A **container** is a data structure holding multiple items, usually all the same type.

A **generic** container is one that can hold objects of client-specified type.

Where Are We?
The Big Challenge

For most of the rest of the semester, we will be addressing the following problem:

- We have a collection of data items, all of the same type, that we wish to store.
- We need to be able to access items [retrieve/find, traverse], add new items [insert] and eliminate items [delete].
- It would be nice if all this were efficient in both time and space.

Solutions to this problem are called **containers**.

- There are many good ones.
- Which one we use depends on many factors, including what priority we place on the various requirements above.

We are particularly interested in **generic containers**: containers in which client code can specify the type of data to be stored.

Unit Overview
Data Handling & Sequences

We now begin a unit on (1) dealing with data using proper error handling and (2) Sequence data structures.

Major Topics

- Data abstraction
- Introduction to Sequences
- Interface for a smart array
- Basic array implementation
- Exception safety
- Allocation & efficiency
- Generic containers
- Node-based structures
- More on Linked Lists
- Sequences in the C++ STL
- Stacks
- Queues

Smart Arrays

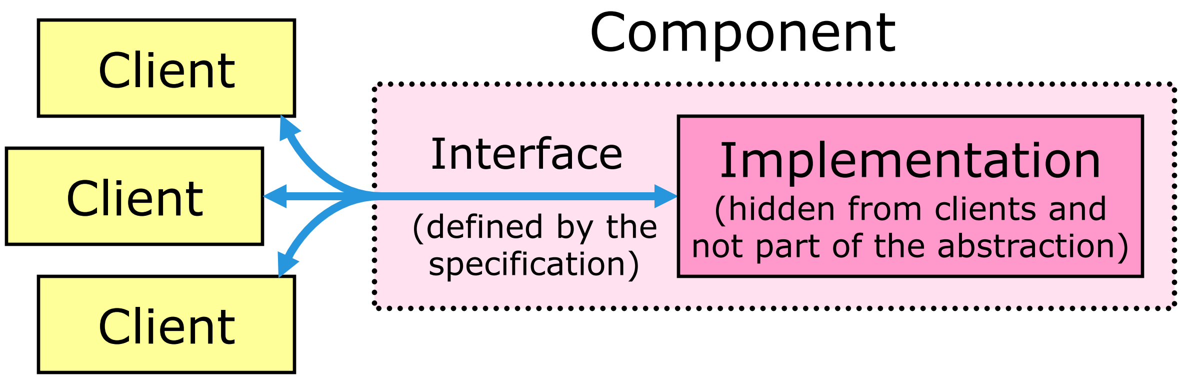
Linked Lists

After this, we will look at various kinds of trees.

Data Abstraction

Data Abstraction
Introduction [1/2]

Abstraction: Considering a software component in terms of *how* and *why* it is used, separate from its implementation.


Client
Client
Client
Component
Interface
(defined by the specification)
Implementation
(hidden from clients and not part of the abstraction)

We have been doing **functional abstraction**.

Now we look at **data abstraction**.

Data Abstraction
Introduction [2/2]

When we do **data abstraction**, we consider data in terms of *how* and *why* it is used, separate from its implementation.

So we need to consider:

- The conceptual form of the data.
- The **operations** available on the data.
- The method used to access the data.

Important Concepts

- Interface
- Abstract data type (ADT)

Data Abstraction
ADTs, Data Structures, Classes

An **abstract data type** (**ADT**) is:

- a **collection of data**, along with
- a **set of operations** on that data.

ADTs are independent of implementation, and even of programming language.

ADTs are heavily used in software development, but often they are not explicitly mentioned.

A **data structure** is a construct within a programming language that stores a collection of data.

C++ and some other programming languages include **classes**, which facilitate object-oriented programming.

Classes are often used to implement data structures. However, one can implement data structures without using classes.

Data Abstraction
ADT Example

Suppose we want to specify an ADT that holds three pieces of information. Assume they have the same type, but make no other assumptions.

Call this ADT *Triple*.

What operations might Triple have? *Here are some possibilities:*

- $\boxed{\text{Create with unspecified values.}}$
- Create with specified values.
- $\boxed{\text{Destroy.}}$
- Copy (create or assign).
- $\boxed{\text{Get a value.}}$
- $\boxed{\text{Change a value.}}$
- Output (if each item can be output).

Just these four allow us to do all the other operations, too.

Eliminate any of the four, and this is no longer true.

We say these four form a **complete**, **minimal** interface. *See the next slide.*

We *might* store Triple data in an obvious data structure: three variables.

And we *might* implement this in C++ as a class with three data members—or an array member—and member functions implementing the various Triple operations.

Data Abstraction
Good Interfaces

When we implement a data structure, the idea of abstraction requires that we have a well defined **interface**.

Designing a good interface can be difficult. Here are some characteristics of a good interface.

An interface must be **complete**.

- All required operations are *possible*.

We often strive for interfaces that are **minimal**.

- Without redundant functionality.

We like interfaces that are **convenient**.

- The interface is not a pain to use.

These two often pull in opposite directions.

We want to **facilitate efficiency**.

- Interface allows data to be dealt with efficiently.

We often want our interface to be **generic**.

- Avoid restricting possible implementations and internal data types.

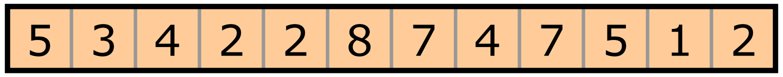
These two may pull in opposite directions.

Introduction to Sequences

Introduction to Sequences
What is a Sequence?

A **Sequence** is a collection of items that are in some order.

We will restrict our attention to **finite** Sequences in which all items have the same type.



It may help to think of an array. However, there are other ways to store Sequences—for example, a Linked List.

Issues

- What operations do we perform on Sequences?
- How can we implement a Sequence?
- How do we decide which implementation best fits any given circumstance?

Introduction to Sequences
ADT Sequence — Definition

Position: think "iterator", although iterators may or may not be used in practice.

ADT **Sequence**

- Data
  - An ordered list, all items the same type, indexed by 0, …, *size*–1.
- Operations
  - CreateEmpty
    - Create **empty** (size 0) Sequence.
  - CreateSized
    - Create Sequence of given size.
  - Destroy
    - Destroy Sequence.
  - Copy
    - Make copy of a Sequence.
  - LookUpByIndex
    - Given a valid index, return item—in modifiable form, if appropriate.
  - Size
    - Return size of Sequence.
  - Empty
    - Is Sequence empty?
  - Sort
    - Sort items with some comparison.
  - Resize
    - Change size. Items 0, …, min(*old_size*, *new_size*)–1 unchanged.
  - InsertByPos
    - Insert given value at a given position.
  - RemoveByPos
    - Remove item at a given position.
  - InsertBeg
    - Insert given value at the beginning.
  - RemoveBeg
    - Remove the first item.
  - InsertEnd
    - Like insertBeg, but at the end.
  - RemoveEnd
    - Like removeBeg, but at the end.
  - Splice
    - Move a contiguous subsequence from one Sequence to another.
  - Traverse
    - Perform an operation on every item.
  - Swap
    - Exchange values of two Sequences.

Introduction to Sequences
ADT SortedSequence — Introduction

Sometimes we want to ensure that a Sequence is always sorted.

This changes the operations available. Operations that mess up the ordering are now disallowed. New operations, that make use of the ordering, become possible.

Therefore, we define another ADT, SortedSequence.

Roughly, a SortedSequence is a Sequence in which the items are always kept sorted according to some comparison function.

Introduction to Sequences
ADT SortedSequence — Draft

If we eliminate the problems, how can we add new items?

ADT **SortedSequence** (draft)

- Data
  - A **sorted** list, all items the same type, indexed by 0, …, *size*−1.
- Operations
  - CreateEmpty
    - Create empty SortedSequence.
  - CreateSized
    - Create SortedSequence of given size.
  - Destroy
    - Destroy SortedSequence.
  - Copy
    - Make copy of a SortedSequence.
  - LookUpByIndex
    - Given a valid index, return item—in modifiable form, if appropriate.
  - Size
    - Return size of SortedSequence.
  - Empty
    - Is SortedSequence empty?
  - Sort ← Either problematic or pointless
    - Sort items with some comparison.

Iffy

Problems

- Resize
  - Change size. Items 0, …, min(*old_size*, *new_size*)−1 unchanged.
- InsertByPos
  - Insert given value at a given position.
- RemoveByPos
  - Remove item at a given position.
- InsertBeg
  - Insert given value at the beginning.
- RemoveBeg
  - Remove the first item.
- InsertEnd
  - Like insertBeg, but at the end.
- RemoveEnd
  - Like removeBeg, but at the end.
- Splice
  - Move a contiguous subsequence from one SortedSequence to another.
- Traverse
  - Perform an operation on every item.
- Swap
  - Exchange values of two SortedSequences.

Introduction to Sequences
ADT SortedSequence — Improved

ADT **SortedSequence** (final)

- Data
  - A list, **in ascending order by some comparison function**, all items the same type, indexed by 0, …, *size*–1.
- Operations
  - CreateEmpty
    - Create empty SortedSequence.
  - Destroy
    - Destroy SortedSequence.
  - Copy
    - Make copy of a SortedSequence.
  - LookUpByIndex
    - Given a valid index, return item—in **non-modifiable** form.
  - Size
    - Return size of SortedSequence.
  - Empty
    - Is SortedSequence empty?
  - **InsertByValue**
    - Given a value, insert it.
  - **RemoveByValue**
    - Given a value, remove item(s) in SortedSequence having an equivalent value, if any exist.
  - RemoveByPos
    - Remove item at a given position.
  - RemoveBeg
    - Remove the first item.
  - RemoveEnd
    - Like removeBeg, but at the end.
  - Traverse
    - Perform an operation on every item.
  - Swap
    - Exchange values of two SortedSequences.
  - **Find**
    - Given a value, find item(s) in SortedSequence having an equivalent value, if any exist.

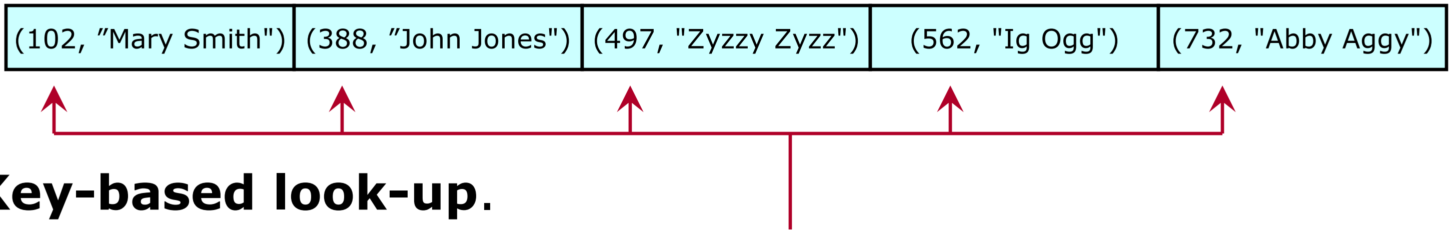
Introduction to Sequences
ADT SortedSequence — What is it For?

The ordering of a SortedSequence is usually not of interest for its own sake. Rather, we want items to be *easy to find by value*.

What can we do with efficient look-up by value?

- First, we can store **Set data**. In a Set, we only care *whether* an item is in the container, not *where* it is.

Now suppose we have a SortedSequence whose items are pairs, and a comparison function that compares only the *first parts* of each pair. What is this good for?



- **Key-based look-up**.
  - The first part of each pair is the **key**.
- “Arrays” (kind of), where the thing between the brackets does not have to be a nonnegative integer.
- That is, **Tables** (a.k.a. *dictionaries*, *associative arrays*, *maps*).

Introduction to Sequences
ADT SortedSequence — Position vs. Value Orientation

Despite the similarities of Sequence and SortedSequence, there is a fundamental difference.

- Sequence handles an item primarily according to its **position** (index or iterator) in the container.
- SortedSequence handles an item primarily according to its **value**.

Two Kinds of ADTs

- Sequence is a **position-oriented** ADT.
- SortedSequence is a **value-oriented** ADT.

SortedSequence is a bit inadequate as a value-oriented ADT.

- Typically, we do not care about SortedSequence being a Sequence.
- Rather, we want to use it to store Set or Table data.
- Maybe we should break it away from its Sequence origins?

Questions (to be examined later)

- What do we really want from a value-oriented ADT?
- How does one implement these in efficient ways?

Interface for a Smart Array

Interface for a Smart Array
Introduction

We wish to implement a Sequence in C++ using a **smart array**.

- It will know its size, be able to copy itself, etc.
  - As in Project 2.
- It will also be able to *change* its size.
  - We did not allow for this in Project 2.
  - Recall that the ADT has resize and various insert/remove operations.

We will work on this for several days. You will finish it in Project 5.

Basic Ideas

- Use a C++ class. An object of the class implements a single Sequence.
- ADT operations will be implemented as member functions, global functions, or combinations of these and Standard Library functions.
- Use iterators, operators, ctors, and the dctor in conventional ways.
- *Every* function in the interface should exist in order to implement, or somehow make possible, an ADT operation.

Interface for a Smart Array By ADT Operation

ADT Operations

- CreateEmpty
  - Default ctor.
- CreateSized
  - Ctor given size.
- Destroy
  - Dctor.
- Copy
  - Copy ctor, copy assignment.
  - Also optimizations: move ctor, move assignment.
- LookUpByIndex
  - Bracket operator.
- Size
  - Member function `size`.
- Empty
  - Member function `empty`.
- Sort
  - Handle externally, with iterators. Use member functions `begin` & `end` and `std::sort` or `std::stable_sort`.

`std::remove` exists and does something different. We could name this member "`remove`", but that might lead to confusion.

- Resize
  - Member function `resize`.
- InsertByPos
  - Member function `insert`.
- RemoveByPos
  - Member function `erase`.
- InsertBeg
  - `insert` with `begin`.
- RemoveBeg
  - `erase` with `begin`.
- InsertEnd
  - Member function `push_back`.
- RemoveEnd
  - Member function `pop_back`.
- Splice
  - Call `resize`, then copy data with `op[]` or `std::copy`.
- Traverse
  - Use member functions `begin` & `end`.
  - This enables range-based for-loops.
- Swap
  - Member function `swap`.

Interface for a Smart Array
Summary

Ctors & Dctor

- Default ctor
- Ctor given size
- Copy ctor
- Move ctor
- Dctor

Member Operators

- Copy assignment
- Move assignment
- Bracket

Global Operators

None

Named Global Functions

None

Named Public Member Functions

- `size`
- `empty`
- `begin`
- `end`
- `resize`
- `insert`
- `erase`
- `push_back`
- `pop_back`
- `swap`

All design decisions so far have been made exactly the same as in `std::vector`—except that `vector` has other members, too.

Interface for a Smart Array
Details

For three of the member functions, it may not be so obvious what the prototype should look like:

- `insert`
  - Takes an iterator and an item.
  - Inserts the item just *before* the item referenced by the iterator—or at the end, if the given iterator is the just-past-the-end iterator.
  - Return value is an iterator referencing the inserted item.
- `erase`
  - Takes an iterator.
  - Removes the item referenced by the iterator.
  - Return value is an iterator to the item following the one removed—or the end iterator, if the item removed was the last in the Sequence.
- `swap`
  - Takes another Sequence, by reference.
  - Exchanges the values of this Sequence and the given one.
  - No return value.
  - *We discussed such a member function earlier in the semester.*