

Interfaces to Data, cont'd

Data Structure Implementation

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, March 26, 2008

Glenn G. Chappell

Department of Computer Science
University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Review

Where Are We? — The Big Problem

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].
- All this needs to be 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 the client can specify the type of data to be stored.

Unit Overview

Sequences and Their Implementations

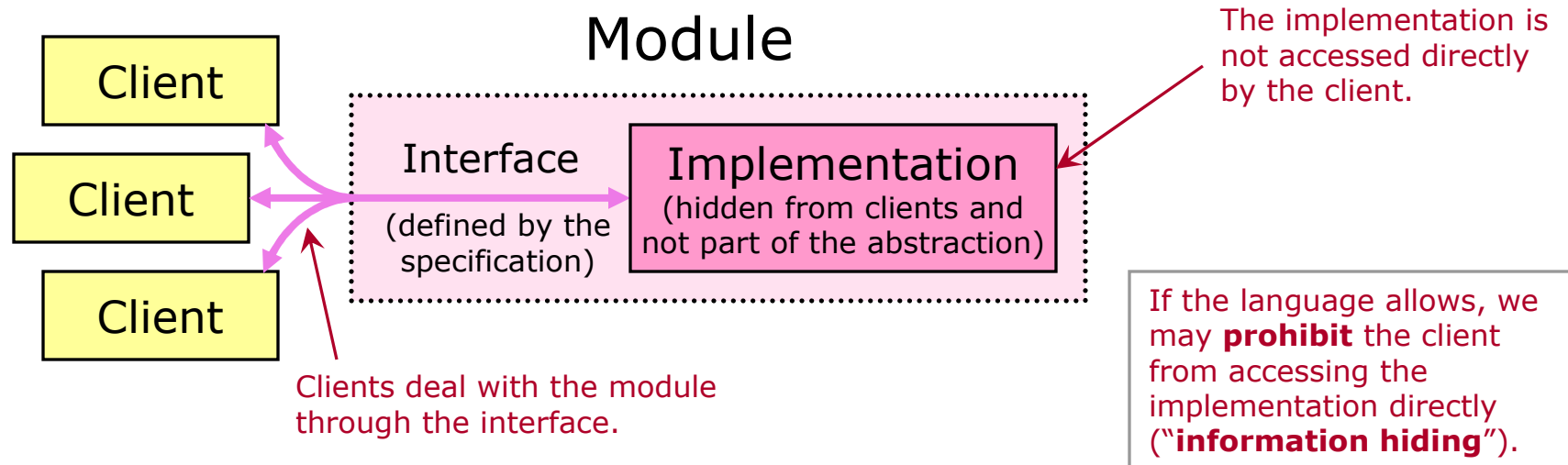
Major Topics

- ✓ • Data Abstraction
- ✓ • Sequence Data
- 1/2✓ • Interfaces to Data
 - Data Structure Implementation
 - Exception Safety
 - Allocation & Efficiency
 - Generic Containers
 - Node-Based Structures
 - Linked Lists
 - Sequences in Practice

Review

Data Abstraction [1/2]

Abstraction means separating the purpose of a module from its implementation.



We have been doing **functional abstraction**.

Next, we look at **data abstraction**.

- Data abstraction = applying the idea of abstraction to data.
- The primary concept we use is that of an **abstract data type**.

Review

Data Abstraction [2/2]

Abstract data type (ADT):

- A **collection of data**, along with a **set of operations** on that data.
- Independent of implementation & programming language.
- Examples: Sequence, SortedSequence.

Data structure:

- A construct within a programming language that stores a collection of data.
- Examples: Array, Linked List.

Class:

- A language feature in C++ and some other languages, intended to facilitate OOP.
- In C++ we *usually* implement a data structure using a class. However, we are not required to.
- Examples: `std::vector<int>`, `std::list<double>`.

Review

Sequence Data — Introduction

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 here. However, there are other ways to store Sequences.

5	3	4	2	2	8	7	4	7	5	1	2
---	---	---	---	---	---	---	---	---	---	---	---

Questions

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

Review

Sequence Data — ADT Sequence

ADT **Sequence**

- **Data**
 - An ordered sequence of values, all the same type, indexed by 0, ..., size-1.
- **Operations**
 - **CreateEmpty**
 - Creates empty Sequence (with size 0, i.e., no data).
 - **CreateSized**
 - Given a size, create a Sequence with that size.
 - **Destroy**
 - Destroys a Sequence.
 - **Copy**
 - Make a copy of a given Sequence.
 - **LookUpByIndex**
 - Given a valid index, returns Sequence item in modifiable form.
 - **Size**
 - Returns size of Sequence.
 - **Empty**
 - Returns whether the Sequence is empty, that is, has size zero.
 - **Sort**
 - Sort a Sequence, using some given comparison function.
 - **Resize**
 - Changes size of Sequence. Data for indices 0, ..., $\min(\text{old size}, \text{new size})-1$ remains identical.
 - **InsertByIter**
 - Given an iterator (or pointer?) and an item, insert the item at the specified position.
 - **RemoveByIter**
 - Given an iterator, remove the item at that position.
 - **InsertBeg**
 - Given an item, insert it 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**
 - Performs some operation on every item in the Sequence, in order.
 - **Swap**
 - Swap the values of two given Sequences.

Review

Sequence Data — SortedSequence

A SortedSequence despite being basically a Sequence in which the data are sorted, is really a rather different thing.

- Sequence is a **position-oriented** ADT.
- SortedSequence is a **value-oriented** ADT.
- In practice, the ordering of a SortedSequence is often not of primary importance. Rather, are interested in items being easy to find.

SortedSequence can be used for:

- **Set** data.
- **Table** data.
 - **Key-based look-up.**

We will get back to value-oriented ADTs later.

Review

Interfaces to Data

An interface should be **complete**.

- All relevant operations should be *possible*.
 - But what is “relevant”? Sometimes we place restrictions, as we will see with ADTs Stack and Queue.

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

- Avoid unnecessary functionality.

An interface should be **convenient**.

- Avoid making the interface a pain to use.

We want to facilitate **efficiency**.

- Allow the data to be dealt with efficiently.

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

- Avoid restricting possible implementations and internal data types.

- Some of the above goals can be at odds with each other. Designing a good, useful interface can be difficult.

These two often pull in opposite directions.

These two *can* pull in opposite directions.

Interfaces to Data, cont'd

An Interface for a Sequence — Start

Now design a good interface for a Sequence class, based roughly on ADT Sequence.

- At this point, we can think in terms of an array implementation (although the interface should not *require* this). Later, we will look at other possible implementations of Sequences.

Basic Ideas

- Use a C++ class. An object of the class implements a single Sequence.
- Most (at least) of the ADT Sequence operations should be implemented using class member functions.
- Use iterators, operators, ctors, and the dctor in the usual ways.
- *Every* function should exist in order to implement, or somehow make possible, an ADT operation.

Interfaces to Data, cont'd

An Interface for a Sequence — By ADT Operation

Use iterators to handle positions, traversing.

ADT Operations

- CreateEmpty
 - Default ctor.
- CreateSized
 - Ctor given size.
- Destroy
 - Dctor.
- Copy
 - Copy ctor & copy assignment.
- LookUpByIndex
 - Bracket operator.
- Size
 - Member function `size`.
- Empty
 - Member function `empty`.
- Sort
 - Handle externally, using iterators. Use iterator-returning member functions `begin` and `end`.
- Resize
 - Member function `resize`.
- InsertByIter, InsertBeg, InsertEnd
 - Member function `insert` does InsertByIter.
 - Use in conjunction with iterator-returning functions to do InsertBeg, InsertEnd.
- RemoveByIter, RemoveBeg, RemoveEnd
 - As above, using `remove`.
- Splice
 - Call `resize`, then copy using `op[]`.
- Traverse
 - Use iterator-returning member functions `begin` and `end`.
- Swap
 - Member function `swap`.

Interfaces to Data, cont'd

An Interface for a Sequence — Summary

Ctors & Dctor

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

Member Operators

- Copy assignment
- Bracket

Global Operators

- *None.*

Associated Global Functions

- *None.*

Named Public Member Functions

- **size**
- **empty**
- **begin**
- **end**
- **resize**
- **insert**
- **remove**
- **swap**

Interfaces to Data, cont'd

An Interface for a Sequence — Details

For most of the member functions in our class, it is pretty obvious what the function prototype should look like.

Some exceptions:

- **insert**
 - Takes an iterator and an item.
 - Inserts the item just before the position referenced by the iterator.
 - Return value is an iterator to the inserted item.
- **remove**
 - Takes an iterator.
 - Removes the item referenced by the iterator.
 - Return value is an iterator to the item following the one removed.
- **swap**
 - Takes another Sequence, by reference.
 - Exchanges the values of this Sequence and the given one.
 - No return value.

Data Structure Implementation

Introduction

In C++ we usually implement a data structure using a **class**.

- Operations are usually implemented using member functions.
- Some operations may need to be global functions, but they are still associated with the class, and are defined in the class's header and/or source files.
- Sometimes we need helper classes. These are probably not visible to the client.

The public interface is all the client sees.

- Every operation should be implemented so that the client can use it.
- Make no functions available to the client that do not implement publicly available operations.
- In C++ this means that we give our class no public member functions that do not implement publicly available operations. We also do not declare global helper functions in the header.
- We can write any *private* functions we might need.
- We may wish to define public types, to help the client deal with the data.

Data Structure Implementation

Implementing a Sequence — General

Call our class “`Sequence`”.

What type should a data item be?

- Use `int` for the value type (for now).
 - You will make it generic in Assignment 5.

What type should the size of a Sequence be?

- Use `std::size_t`.

How should we store the data?

- Use a dynamically allocated array of `ints`.
- Note: We could have used a separate RAII class, like `IntArray`.

How should we implement the iterators?

- Use pointers (`int *`, `const int *`).

Have member types, as in STL containers: `value_type`, `size_type`, `iterator`, `const_iterator`.

- This allows us to easily tell what a value is for.
- Also, we can easily change (say) the value type.

Data Structure Implementation

Implementing a Sequence — Details

What data members should class `Sequence` have?

- Size of the array: `size_type size_;`
- Pointer to the array: `value_type * data_;`

Note: There are serious (but non-obvious) problems in this design, as we will see.

What class invariants should it have?

- Member "`size_`" should be nonnegative.
- Member "`data_`" should point to an `int` array, allocated with `new []`, owned by `*this`, holding (at least) `size_ ints`.

What should `operator[]` return? Should it be `const` or not?

- We need two versions: non-`const` and `const`.
- The non-`const` version returns `value_type &`.
- The `const` version returns `const value_type &`.

What should `begin`, `end` return? Should they be `const` or not?

- As with `operator[]`, we need two versions.
- Non-`const` versions return `iterator`, `const` versions return `const_iterator`.

What about the Big Three? Can we use silently written functions?

- No. We are directly managing an owned resource.

Data Structure Implementation

Implementing a Sequence — Do It

TO DO

- Write *some* of class `sequence` as described.

*Done. See `sequence.h`,
`sequence.cpp`, on the web page.*

Note: We will be writing and improving this class in various ways in the next few days. Your job in Assignment 5 will be to finish it, including turning it into a generic container class.