Basic Array Implementation

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, October 14, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu
© 2005–2020 Glenn G. Chappell
Some material contributed by Chris Hartman
Review
Our problem for most of the rest of the semester:

- **Store**: A collection of data items, all of the same type.
- **Operations**:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- **Time & space efficiency** are desirable.

A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.
Unit Overview
Data Handling & Sequences

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
Review
Data Abstraction

Abstract data type (ADT):
- A collection of data, along with a set of operations on that data.
- Independent of implementation and 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 feature in C++ and some other programming languages, aimed at facilitating OOP.
- In C++, we often implement a data structure using a class. However, we are not required to.
- Examples: std::vector<int>, std::list<double>.
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.

```
5 3 4 2 2 8 7 4 7 5 1 2
```

We defined an ADT **Sequence**.

- Data. An ordered list, all items the same type, indexed by 0, ..., size–1.
- Operations. CreateEmpty, CreateSized, Destroy, Copy, LookUpByIndex, Size, Empty, Sort, Resize, InsertByPos, RemoveByPos, InsertBeg, RemoveBeg, InsertEnd, RemoveEnd, Splice, Traverse, Swap.
We wish to implement a Sequence in C++ using a **smart array**. It will know its size, be able to copy itself, etc. It will also be able to change its size.

**Basic Ideas**

- Use a C++ class. An object of the class implements a single Sequence.
- 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.

You will finish this implementation in Project 5.
Review
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`.

- **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`.

 std::remove exists and does something different. We could name this member “remove”, but that might lead to confusion.
Review
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 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.
Basic Array Implementation
Basic Array Implementation
Introduction

We will implement our data structure as a C++ class. Its interface will consist of the public members of the class.

Note. There is nothing wrong with global functions—friends of the class, perhaps—being part of the interface; but our interface happens not to involve any.

- Example. A string class might implement concatenation via a global operator+.

The public interface is all that client code sees.

- Every operation must be doable through this interface.
- Every function available to client code exists in order to implement one or more publicly available operations.
- We can write any private functions we might feel like writing.
- As a convenience, we can define public member types, to help client code deal with the data.
Basic Array Implementation
Design Decisions [1/2]

Call our class FSArray (Frightfully Smart Array).

What type should an array item be?
- Use \texttt{int} for the value type.
- This is just for now. You will make it generic in Project 5.

What type should the size of an array be?
- Use \texttt{std::size_t} for the size type.

How should we store the data?
- Store the data in a dynamically allocated array of \texttt{int}.
- Note. We could have used a separate RAII class, like \texttt{IntArray}.

How should we implement the iterators?
- Use pointers for iterators (\texttt{int *}, \texttt{const int *}).

What member types should we define?
- \texttt{value_type}, \texttt{size_type}, \texttt{iterator}, \texttt{const_iterator}. 
Basic Array Implementation
Design Decisions [2/2]

What data members should our array class have?
- Size of the array: `size_type _size;
- Pointer to the array: `value_type * _data;

What class invariants should it have?
- Member `_size` is nonnegative.
- Member `_data` points to an `int` array, allocated with `new []`, owned by `*this`, holding `_size` ints.

What should `operator[]` return? Should it be const or not?
- We need two versions: non-const and const.
- These return `value_type &`, `const value_type &`, respectively.

What should `begin`, `end` return? Should they be const or not?
- As with `operator[]`, we need two versions: non-const and const.
- These return `iterator`, `const_iterator`, respectively.

Can we use automatically generated versions of the Big Five?
- No. We are directly managing an owned resource.
Basic Array Implementation

CODE

TO DO

- Write a skeleton form of class FSArray.
  - The package header & source files: #ifndef, #include, etc.
  - The class definition.
  - Definitions of all public types.
  - Prototypes and dummy definitions for all public functions.
- As time permits, begin implementing functionality.
  - Declarations of data members and comments indicating class invariants.
  - Definitions for functions that do not copy/move/swap or resize the array.
  - Definitions for member functions push_back & pop_back.

Done. See fsarray.h & fsarray.cpp.
Also see fsarray_main.cpp for a program to compile the package with.

We will improve FSArray over the next few days. In Project 5 you will turn it into a generic container and finish it.

2020-10-14