Unit Overview
Handling Data & Sequences

Major Topics

✓ Data abstraction
✓ Introduction to Sequences
  ▪ Smart arrays
    ✓ Array interface
    ▪ Basic array implementation
    ▪ Exception safety
    ▪ Allocation & efficiency
    ▪ Generic containers
  ▪ Linked Lists
    ▪ Node-based structures
    ▪ More on Linked Lists
  ▪ Sequences in the C++ STL
  ▪ Stacks
  ▪ Queues
Review
Where Are We? — The Big Problem

Our problem for much of the rest of the semester:

- **Store**: a collection of data items, all of the same type.
- **Operations**:
  - Access items [one item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- All this needs to be efficient in both time and space.

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

**Generic containers**: those in which client code can specify the type of data stored.
Abstraction: Separate the purpose of a module from its implementation.

Recall: Function, class, or other unit of code. Generally smaller than a package.

We have been doing functional abstraction. Now we look at data abstraction.
In **data abstraction**, we separate the various aspects of dealing with data, from the implementation of the data:

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

**Important concepts**

- Abstract data type (ADT).
- Interface.
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 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>`.
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 should be complete.
- All required operations should be possible.

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.
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.

Questions
- What operations do we perform on Sequences?
- How can we implement a Sequence?
- How do we decide which implementation best fits any given circumstance?
Review
Introduction to Sequences — ADT Sequence, Definition

**ADT Sequence**

- **Data**
  - An ordered sequence of values, all 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(old size, 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**
  - Exchange the values of two given Sequences.
SortedSequence: like Sequence, except that items are kept sorted. Despite superficial similarity, a SortedSequence is fundamentally a different kind of thing from a Sequence.

- In practice, the ordering of a SortedSequence is often of little importance. Rather, are interested in items being **easy to find**.
- Sequence is a **position-oriented** ADT.
- SortedSequence is a **value-oriented** ADT.

SortedSequence can be used for:

- **Set** data.
- **Table** data.  

We will get back to value-oriented ADTs later in the semester.
Review
Array Interface — 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”.

26 Oct 2009
CS 311 Fall 2009
Review
Array Interface — 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
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 client code.

The public interface is all that client code sees.  
- Every operation should be implemented so that clients can use it.  
- Make no functions available to client code 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.
Basic Array Implementation
General

Call our class “SmArray”.

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 SmArray 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.
Basic Array Implementation Details

What data members should class **SmArray** have?
- Size of the array: `size_type size_;`
- Pointer to the array: `value_type * data_;`

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 `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.

Note: This design has a serious (but not obvious) problem, as we will see.
TO DO

- Write some of class SmArray, as described.

Done. See smarray.h, smarray.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.
Exception Safety
Refresher — Error Handling

An **error condition** (or “error”) is a condition occurring during runtime that cannot be handled by the normal flow of execution.

- Not necessarily a bug or a user mistake.
- Example: Could not read file.

Three ways to deal with a possible error condition in a function:

- **Prevention**
  Client code must prevent the error (precondition).

- **Containment**
  Fix the problem inside the function.

- **Signal the Client Code**
  Idea: When we cannot fulfill our postconditions.

Methods for signaling an error condition to the client code:

- **Return an error code**
- **Set a flag, checked by a separate function**
- **Throw an **exception****
Exceptions are objects that are “thrown”, generally to signal error conditions.

- We catch exceptions using a try ... catch construction.
- “throw” backs out of blocks & functions, until a matching catch is found.
- An uncaught exception terminates the program.

```cpp
Foo * makeAFoo() // throw(std::bad_alloc)
{ return new Foo(2, 3); }

void myFunc() // throw()
{
    Foo * p;
    try {
        p = makeAFoo();
    }
    catch (std::bad_alloc & e) {
        allocationSuccessful = false;
        cout << "Oops! Message: " << e.what() << endl;
    }
```

Catch by reference.
We can throw our own exceptions, using “throw”.

```cpp
class Foo {
public:
    int & operator[](int index) // May throw std::range_error
    {
        if (index < 0 || index >= arraySize)
            throw std::range_error("Foo: index out of range");
        return theArray[index];
    }
private:
    int * theArray;
    std::size_t arraySize;
};
```

We do not do this very much. And we only do it when we must signal the client code that an error condition has occurred.
Exception Safety
Refresher — Introduction to Exceptions [3/4]

We can catch **all** exceptions, using “...”.

- In this case, we do not get to look at the exception, since we do not know what type it is.

```java
try {
    myFunc4(17);
}
catch (...) {
    fixThingsUp();
    throw;
}
```

- Inside any `catch` block, we can **re-throw the same exception** using `throw` with no parameters.
The following can throw in C++:

- “throw” throws.
- “new” may throw `std::bad_alloc` if it cannot allocate.
- A function that (1) calls a function that throws, and (2) does not catch the exception, will throw.
- Functions written by others may throw. See their doc’s.

The following do not throw:

- Built-in operations on built-in types.
  - Including the built-in `operator[]`.
- Deallocation done by the built-in version of “delete”.
  - Note: “delete” also calls destructors. These can throw.
- C++ Standard I/O Libraries (default behavior)

If a destructor is called between a throw and a catch, and that destructor throws, then the program terminates.

Therefore, **destructors should not throw.**
Exception Safety
TO BE CONTINUED ...

Exception Safety will be continued next time.