CSE 143

Abstract Data Types, Collections, and Iterators

[Chapter 3]

Abstraction

What is *Abstraction*?

- An idealization
- A focus on essential qualities, disregarding the "details"
- An emphasis on the *what* rather than the *how*
  `Specification` vs. `Implementation`
- A problem-solving technique

Why Abstraction?

- Abstractions helps in managing complexity
  - Don't need to know details, just interface
- Treat abstractions as "black box" components to build upon
  - Know what inputs go into box, and what outputs come out, but not what goes on inside the box
  - Hierarchical or layered decomposition

Abstraction in Programming

- The type **`int`** is an abstraction for a way of interpreting bits in memory as a number
- A *struct* is an abstraction of a collection of related data items
- A *function* is a programmer-designed abstraction for some computation
- A *module (class)* is a programmer-designed abstraction that groups functions and data together and provides an interface

Abstract Data Types (ADTs)

Composed of two parts:

- **Specification**
  - Name of new type
  - Constructors to make instances
  - Public operators on instances
- **Implementation**
  - Representation of new type
  - Implementation of public operators, constructors
  - Additional private operations, data, etc.

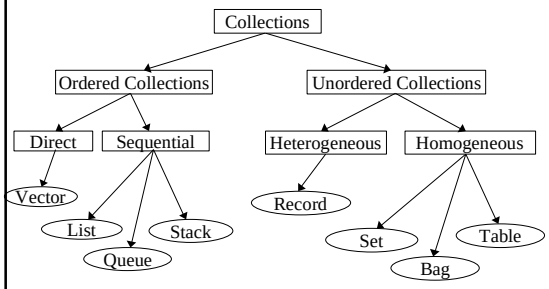
Abstract Data Types (ADTs)

- Program in terms of types appropriate to the problem being solved
  - Bank account, game board, matrix/vector math, video stream, machine tool control, …
- Separate Concerns
  - Specification as contract between client and type implementation
  - Partition large job into manageable parts

Collection ADTs

- Many standard ADTs are for *collections*
  - Data structures that manage groups of data
- Some terms:
  - Ordered vs. Unordered (list vs. set)
  - Method of access: Direct vs. Sequential
  - Type of data: Homogenous vs. Heterogeneous
  - Size: Fixed vs. Variable

Classification of ADTs



Types in C++

- Direct support for records (`struct`/`class`) and arrays
- Other ADTs must be implemented
  - No absolute standards
  - Many conventional operations and concepts
- Standard (Template) Library (new)
  - contains many ADTs
- Our Plan:
  - Look at several ADTs
  - Learn the conventional operations
  - Consider possible implementations

Arrays/Structs: Abstract?

- Most languages provide arrays and structs
  - Generally not high-level, problem-oriented
  - Use as implementation techniques for higher-level abstractions
  - Other implementation techniques: linked lists, trees, hash tables

Structs (and classes)

- Attributes
  - Nonlinear collection of heterogeneous elements (*fields*)
  - Fixed number of elements
  - Direct access
- Operations
  - Accessing elements directly through `.` operator
- Typical usages
  - C++ structs: simple collections of heterogenous data
  - C++ classes: collections of data with associated operations (use to implement true abstract types)

Arrays

- Attributes
  - Linear sequence of homogeneous elements
  - Fixed length
  - Direct access
- Operations
  - Direct indexing using `[]` operator
  - C++ arrays provide no bounds checking
  - No searching, sorting, insertion, etc.

Vector ADT

- Attributes
  - Linear sequence of homogeneous elements
  - Fixed or variable length
  - Direct access
- Operations
  - Direct indexing using [] operator
  - Others: bounds checking, slices (substrings), …
- Implementations
  - C++ array (usual)
  - Linked lists, trees (for unusual applications)

Subscripting for Vectors

- A robust Vector class would deal with subscript errors in some reasonable way.

```
Class IntVector {
public:int & operator[ ](int k);
...
private:
    int data[1..size];  // vector data;
};
...
int & IntVector::operator[ ](int k) {
   if (k<0 || k>=size) {
      deal with out-of-bounds subscript
   }
   return data[k];
}
```


Lists [Section 4.5]

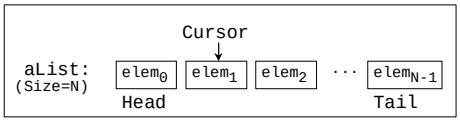
- Attributes of *list* type:
  - Linear sequence of homogeneous elements
  - Varying number of elements
  - Elements ordered (1st thing in list, 2nd, …)
- Operations on lists
  - Insert, delete elements at current position
  - Test if full, empty
  - Test if item is in the list
  - Access items in sequence

List Implementations

Operation	Array	Sngl Link Lst	Dbl Linked
Create	O(1)	O(1)	O(1)
Delete all	O(1)	O(n)	O(n)
Calc length			
Find item			
Access kth item			
Get next item			
Get prev item			
Insert at curr pos			
Delete at curr pos			

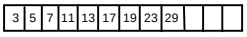
List Iteration - One Design

- *Head*: First element in list
- *Tail*: Last element in list
- *Cursor*: "Current" element of list
- *Size*: Number of elements in list



List Implementation

- Store data in an array



- Advance: increment cursor
- Reset: set cursor to 0
- Insert at cursor: shift elements up beyond cursor to make room for new elements
- Delete at cursor: shift elements down to overwrite element

Implementation Critique

- Every instance of List class has overhead for iteration, even if iteration is never used
- Only one iteration can be performed at a time on any single list
  - What if we want two active cursors? (i.e., binary search)
- Better: create an *iterator* object to hold data and provide operations needed to access elements of the list in sequence.

Iterators

- Define an interator class
  - Contains cursor and operations to reset, advance, and test for end of container
- An iterator object is associated with a particular container object
- Idea of an iterator is found in many programming languages
  - Many possible choices of specific operations to provide
  - Implementation details depend on the specific programming language

Iterator Example

- Suppose we have class IntList (list of integers) and associated IntListIter (iterator for IntList). Compute the sum of the list as follows

```
IntList L;          // integer list
IntListIter it(L)   // iterator for L
int sum = 0;
...
it.reset();              // reset iterator
while (it.moreData()) {
   sum = sum + it.current();
   it.advance();
}
```


IntList and IntListIter

- Everything in class IntListIter needs access to representation of class IntList
- Declare class IntListIter as friend of IntList
- No other changes to IntList
- Example: Assume IntList implemented with array

```
class IntList {
  ...
  friend class IntListIter;
  ...
private;
  const int maxData = 100;
  int data[0..maxData]; // values stored in
  int nData;            // data[0..nData-1]
};
```


IntListIter Representation

- Key idea: IntListIter holds a reference to the associated IntList and a current position (cursor).

```
class IntListIter {
...
private:
  IntList * theList;    // theList bound to
                        // this iterator
  int currentPos;       // cursor
};
```


IntListIter Interface

```
class IntListIter {
public:
  // Construct this IntListIter bound to
  // IntList L and reset cursor
  IntListIter(IntList * L);
  // reset cursor to beginning of theList
  void reset( );
  // = “more elements in the list”
  bool moreData( );
  // = value of element at cursor
  int current( );
  // advance to next element
  void advance( );
  ...
```

Iterator Creation and Reset

- Initialization and reset only involve variables in the IntListIter itself in this example

```
// Construct this IntListIter bound to
// IntList L and reset cursor
IntListIter::IntListIter(IntList * L){
  theList = L;
  currentPos = 0;

// reset cursor to beginning of theList
void IntListIter::reset( ) {
  currentPos = 0;
}
```


Iteration

- Decision: What happens if iteration runs off end of the IntList? Answer for this example: assert fails.

```
// = "more elements in the list"
bool IntListIter::moreData( ) {
  currentPos < theList->nData;
}
// = value of element at cursor
int IntListIter:: current( ) {
  assert(moreData( ));
  return theList->data[currentPos];
}
// advance to next element
void IntListIter:: advance( ) {
  assert(moreData( )); currentPos++;
}
```


Cleaning up Class IntList

- Once we have an iterator class for IntList, we may be able to reduce access to IntList internals in other parts of the program.
- In particular, stream output can be implemented without knowing representation of IntList.
  - Remove `friend ostream operator<<( … )` from IntList

Cleaning up Class IntList

- New implementation of stream output

```
ostream & operator<<(ostream&s,
                     const IntList &L) {
  IntListIter it(&L);
  s << '<';
  // it.reset( ) done in constructor
  if (it.moreData( )) {
    s << it.current( ); it.advance( );
    while(it.moreData( )) {
      s << ',' << it.current( );
      it.advance( );
    }
  }
  return (s << '>');
}
```


Iterator Issues

- This is a very simple implementation of an iterator
- Issues for production-quality iterator classes
  - What happens if items are added to/deleted from list during iteration? Iterator should at least not crash.
  - Should the iterator return copies of the list values or references to list elements?
  - Iterator algorithms are more complex for complicated data structures
  - Exercise: Re-implement IntListIter for an IntList class with a linked-list representation.