Abstract Data Types
Iterators
Vector ADT
Sections 3.1, 3.2, 3.3, 3.4

Abstract Data Type (ADT)

- High-level definition of data types
- An ADT specifies
  - A *collection* of data
  - A set of *operations* on the data or subsets of the data
- ADT does not specify how the operations should be implemented
- Examples
  - vector, list, stack, queue, deque, priority queue, table (map), associative array, set, graph, digraph

Iterators

- Helps navigate through the items in a list.
- An example: iterator over `vector v`.

```
for (int i = 0; i != v.size(); i++ )
  cout << v[i] << endl;
```

Iterators (contd.)

- A generalized type that help in navigating any container
  - A way to initialize at the front and back of a list
  - A way to move to the next or previous position
  - A way to detect the end of an iteration
  - A way to retrieve the current value
- Examples:
  - Iterator type for **`vector<int>`** defined as
    - **`vector<int>::iterator itr;`**
  - Iterator type for **`list<string>`** defined as
    - **`list<string>::iterator itr;`**

Getting an Iterator

- Two methods in all STL containers
  - `begin()`
    - Returns an iterator to the first item in the container
  - `end()`
    - Returns an iterator representing the container (i.e. the position after the last item)
- Example:

```
for (int i = 0; i != v.size(); i++ )
    cout << v[i] << endl;
```

can be written using iterators as

```
for(vector<int>::iterator itr=v.begin(); itr!=v.end(); itr.???)
      cout << itr.??? << endl;
```

- What about **`???`**

Iterator Methods

- Iterators have methods
- Many methods use operator overloading
  - **`itr++ and ++itr`** → advance the iterator to next location
  - **`*itr`** → return reference to object stored at iterator **`itr`**'s location
  - **`itr1 == itr2`** → true if **`itr1`** and **`itr2`** refer to the same location, else false
  - **`itr1 != itr2`** → true if **`itr1`** and **`itr2`** refer to different locations, else false
- Previous example becomes

```
for(vector<int>::iterator itr= v.begin(); itr!= v.end(); itr++)
    cout << *itr << endl;
```

- Alternatively

```
vector<int>::iterator itr = v.begin( );
while( itr != v.end( ) )
  cout << *itr++ << endl;
```

Iterator example

```
template <typename Container>
void removeEveryOtherItem( Container & lst )
{
    typename Container::iterator itr = lst.begin( );
    while( itr != lst.end( ) )
    {
        itr = lst.erase( itr );
        if( itr != lst.end( ) )
            ++itr;
    }
}
```

const_iterator

- Returns a constant reference for **`operator*`**
- So that a function does not try to modify the elements of a constant container object.
- Note that **`c.begin()`** and **`c.end()`** functions in the example return **`const_iterator`** type.

```
template <typename Container>
void printCollection( const Container & c, ostream & out = cout )
{
    if( c.empty( ) )
        out << "(empty)";
    else
    {
        typename Container::const_iterator itr = c.begin( );
        out << "[ " << *itr++;  // Print first item

        while( itr != c.end( ) )
            out << ", " << *itr++;
        out << " ]" << endl;
    }
}
```

The Vector ADT

Generic arrays

The Vector ADT

- Extends the notion of array by storing a sequence of arbitrary objects
- Elements of vector ADT can be accessed by specifying their index.

Vectors in STL

- Collection ➔ Elements of some proper type `T`
- Operations
  - **`int size()`** ➔ returns the number of elements in the vector
  - **`void clear()`** ➔ removes all elementes from the vector
  - **`bool empty()`** ➔ returns true of the vector has no elements
  - **`void push_back ( const Object &x )`**
    - adds `x` to the end of the vector
  - **`void pop_back ( )`**
    - Removes the object at the end of the vector
  - **`Object & back ( )`**
    - Returns the object at the end of the vector
  - **`Object & front ( )`**
    - Returns the object at the front of the vector

Vectors in STL (contd.)

- More Operations
  - **`Object & operator[] ( int index )`**
    - Returns the object at location index (without bounds checking)
    - Both accessor and mutator versions
  - **`Object & at( int index )`**
    - Returns the object at location index (with bounds checking)
  - **`int capacity()`**
    - Returns the internal capacity of the vector
  - **`void reserve(int newCapacity)`**
    - Sets the new capacity of the vector
  - void **`resize(int newSize )`**
    - Change the size of the vector

Implementing Vector Class Template

- Vector maintains
  - A primitive C++ array
  - The array capacity
  - The current number of items stored in the Vector
- Operations:
  - Copy constructor
  - `operator=`
  - Destructor to reclaim primitive array.
  - All the other operators we saw earlier.

Vector Implementation (Part 1)

```
template <typename Object>
class Vector
{
  public:
    explicit Vector( int initSize = 0 )
      : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
      { objects = new Object[ theCapacity ]; }
    Vector( const Vector & rhs ) : objects( NULL )
      { operator=( rhs ); }
    ~Vector( )
      { delete [ ] objects; }

    const Vector & operator= ( const Vector & rhs )
    {
        if( this != &rhs )
        {
            delete [ ] objects;
            theSize = rhs.size( );
            theCapacity = rhs.theCapacity;

            objects = new Object[ capacity( ) ];
            for( int k = 0; k < size( ); k++ )
                objects[ k ] = rhs.objects[ k ];
        }
        return *this;
    }
```

➔Constructor

➔Copy Constructor

➔deep copy assignment

Vector Implementation (Part 2)

```
    void resize( int newSize )
    {
        if( newSize > theCapacity )
            reserve( newSize * 2 + 1 );
        theSize = newSize;
    }

    void reserve( int newCapacity )
    {
        if( newCapacity < theSize )
            return;

        Object *oldArray = objects;

        objects = new Object[ newCapacity ];
        for( int k = 0; k < theSize; k++ )
            objects[ k ] = oldArray[ k ];

        theCapacity = newCapacity;

        delete [ ] oldArray;
    }
```

Expand to twice as large because memory allocation is an expensive operation

Vector Implementation (Part 3)

```
    Object & operator[]( int index )
      { return objects[ index ]; }
    const Object & operator[]( int index ) const
      { return objects[ index ]; }

    bool empty( ) const
      { return size( ) == 0; }
    int size( ) const
      { return theSize; }
    int capacity( ) const
      { return theCapacity; }

    void push_back( const Object & x )
    {
        if( theSize == theCapacity )
            reserve( 2 * theCapacity + 1 );
        objects[ theSize++ ] = x;
    }

    void pop_back( )
      { theSize--; }
```

No error checking

Vector Implementation (Part 4)

```
    const Object & back ( ) const
      { return objects[ theSize - 1 ]; }

    typedef Object * iterator;
    typedef const Object * const_iterator;

    iterator begin( )
      { return &objects[ 0 ]; }
    const_iterator begin( ) const
      { return &objects[ 0 ]; }
    iterator end( )
      { return &objects[ size( ) ]; }
    const_iterator end( ) const
      { return &objects[ size( ) ]; }

    enum { SPARE_CAPACITY = 16 };

  private:
    int theSize;
    int theCapacity;
    Object * objects;
};
```

Same as pointers to Object