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

- High-level definition of data types
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
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
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
Iterators

- Helps navigate through the items in a list.
Iterators

- Helps navigate through the items in a list.

- An example: iterator over `vector v`.

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

- A generalized type that help in navigating any container
Iterators (contd.)

- A generalized type that help in navigating any container
  - A way to initialize at the front and back of a list
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
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
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
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:
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
  ```cpp
  vector<int>::iterator itr;
  ```
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
Getting an Iterator

- Two methods in all STL containers
Getting an Iterator

- Two methods in all STL containers
  - `begin()`
    - Returns an iterator to the first item in the container
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)
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
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:
  ```cpp
  for (int i = 0; i != v.size(); i++)
    cout << v[i] << endl;
  ```
  can be written using iterators as

  ```cpp
  for(vector<int>::iterator itr=v.begin(); itr!=v.end(); itr++)
    cout << itr.??? << endl;
  ```
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:
  ```cpp
  for (int i = 0; i != v.size(); i++)
    cout << v[i] << endl;
  ```

  can be written using iterators as

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

- What about ???
Iterator Methods
Iterator Methods

- Iterators have methods
Iterator Methods

- Iterators have methods
- Many methods use operator overloading
Iterator Methods

- Iterators have methods
- Many methods use operator overloading
  - `itr++` and `++itr` ➔ advance the iterator to next location
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
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
**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
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

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

- Alternatively
  ```cpp
  vector<int>::iterator itr = v.begin();
  while( itr != v.end() )
      cout << *itr++ << endl;
  ```
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;
    }
}
template<typename Container>
void printCollection( const Container & c, ofstream & 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;
    }
}
const_iterator

- Returns a constant reference for operator*
const_iterator

- Returns a constant reference for operator*

- So that a function does not try to modify the elements of a constant container object.
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.

```cpp
1 template <typename Container>
2 void printCollection( const Container & c, ostream & out = cout )
3 {
4   if( c.empty() )
5       out << "(empty)";
6   else
7     {}
8     
9       typename Container::const_iterator itr = c.begin();
10      out << "[ " << *itr++; // Print first item
11     
12     while( itr != c.end() )
13       out << ", " << *itr++;
14      out << "]" << endl;
15     }
```
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
Vectors in STL

- Collection $\rightarrow$ Elements of some proper type $T$
Vectors in STL

- Collection ➔ Elements of some proper type T
- Operations
Vectors in STL

- Collection ➔ Elements of some proper type T
- Operations
  - `int size()` ➔ returns the number of elements in the vector
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 elements from the vector
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 elements from the vector
  - `bool empty()` ➔ returns true if the vector has no elements
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 elements from the vector
  - `bool empty()` ➔ returns true if the vector has no elements
  - `void push_back ( const Object &x )` ➔ adds x to the end of the vector
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
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 elements from the vector
  - `bool empty()` ➔ returns true if 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
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 elements from the vector
  - `bool empty()` ➔ returns true if 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.)
Vectors in STL (contd.)

- More Operations
Vectors in STL (contd.)

- More Operations
  - `Object & operator[]( int index )`
    - Returns the object at location index (without bounds checking)
    - Both accessor and mutator versions
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)
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
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
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
Implementing Vector Class Template

- Vector maintains
  - A primitive C++ array
  - The array capacity
  - The current number of items stored in 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
  - \texttt{operator=}
  - Destructor to reclaim primitive array.
  - All the other operators we saw earlier.
Vector Implementation (Part 1)

```cpp
template<typename Object>
class Vector {

public:

    explicit Vector( int initSize = 0 )
        : theSize( initSize ), theCapacity( initSize + SPARE_CAPACITY )
          { objects = new Object[ theCapacity ]; }  // Constructor

    Vector( const Vector & rhs ) : objects( NULL )
        { operator=( rhs ); }  // Copy Constructor

    ~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;
    }

};
```
Vector Implementation (Part 2)

27
28     void resize( int newSize )
29     {
30         if( newSize > theCapacity )
31             reserve( newSize * 2 + 1 );
32             theSize = newSize;
33     }
34
35     void reserve( int newCapacity )
36     {
37         if( newCapacity < theSize )
38             return;
39     
40             Object *oldArray = objects;
41         
42             objects = new Object[ newCapacity ];
43             for( int k = 0; k < theSize; k++ )
44                 objects[ k ] = oldArray[ k ];
45         
46             theCapacity = newCapacity;
47         
48             delete [ ] oldArray;
49     }

Expand to twice as large because memory allocation is an expensive operation
Vector Implementation (Part 3)

```
50 Object & operator[]( int index )
51    { return objects[ index ]; }
52
53 const Object & operator[]( int index ) const
54    { return objects[ index ]; }
55
56 bool empty() const
57    { return size() == 0; }
58
59 int size() const
60    { return theSize; }
61
62 int capacity() const
63    { return theCapacity; }
64
65 void push_back( const Object & x )
66    {
67        if( theSize == theCapacity )
68            reserve( 2 * theCapacity + 1 );
69            objects[ theSize++ ] = x;
70    }
71
72 void pop_back( )
73    { theSize--; }
74
75 No error checking
```
Vector Implementation (Part 4)

```
71 72 73 74 75 76 77
const Object & back ( ) const
{ return objects[ theSize - 1 ]; }

typedef Object * iterator;
typedef const Object * const_iterator;

78 79 80 81
iterator begin( )
{ return &objects[ 0 ]; }

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

82 83 84 85
iterator end( )
{ return &objects[ size( ) ]; }

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

86 87
enum { SPARE_CAPACITY = 16 };

88

89 90 91 92
private:
int theSize;
int theCapacity;
Object * objects;
93
```