CSCI 335
Software Design and Analysis III
Lecture 6: Lists, Stacks, Queues and the STL

Professor Anita Raja

09-15-22

1

1

Agenda

- Log complexity
  - Binary Search
  - Fast Exponentiation
- Lists/Stacks/Queues
  - Vector vs List in STL
  - Iterators
  - erase

2

2

List/Stack/Queue ADTs

- Already covered in 235
- We will emphasize the STL and use of iterators

3

3

What is Abstract Data Type?

- Mathematical abstractions (list, sets, graphs); set of objects with set of operations (add, remove, size).
- Implementation not specified; up to designer.
- C++ class allows for ADT implementation with appropriate hiding of details.

4

4

What is STL?

- C++ includes in its library an implementation of common data structures called Standard template library
- List ADT is one of the data structures (collections of containers)

5

5

Vector vs. List in the STL

- STL vector:
  - + Constant time indexing
  - + Fast to add data at the end (not front).
  - - Slow to add data in the "middle"
- STL list:
  - + Implemented as a doubly linked list
  - + Fast insertion/removal of items in any position
  - - No indexing

Both are inefficient for searches!

6

6

Instantiation

- int size() const //returns the number of elements in the container
- void clear() //removes all the elements in the container
- bool empty() const //returns true if the container contains no elements and false otherwise

7

7

Vector vs. List in the STL

- STL vector and list:
  - Adding and removing element from end of list as well as accessing front item in the list is in constant time.

```
void push_back(const Object &x) //adds x at the end of the list
void pop_back() //removes object at the end of the list
const Object &back() const  // returning the object at the end of the list
(an accessor  that returns a reference is also provided)
const Object &front() const // returning the object at the front of the list
(an accessor that returns a reference is also provided)
```

- STL list only:

```
void push_front(const Object &x)//adds x to the front of the list
void pop_front()//removes the object at front of list
```

8

8

Vector vs. List in the STL

- STL vector only:
  - Efficient indexing

```
Object & operator[](int idx) //  returns object at index idx with no bounds checking (an accessor that returns const ref. also provided)
Object &at(int idx) // returns object at index idx with bounds checking (an accessor that returns const ref. also provided)
```

  - Efficient view and change internal capacity

```
int capacity() const //returns internal capacity of vector Sec 3.4
void reserve(int new_capacity)//sets new capacity and possibly avoid expansion of vector Sec 3.4
```

9

9

Container

- A holder object that stores a collection of other objects (its elements).
- Implemented as class templates:
  - Allows great flexibility in the types supported as elements.
- Manages the storage space for its elements and provides member functions to access them
  - either directly or through iterators (reference objects with similar properties to pointers).
- Containers replicate structures very commonly used in programming:
  - dynamic arrays (vector), queues stacks , heaps (priority_queue), linked lists, trees (set), associative arrays (map)…
- Many containers have several member functions in common, and share functionalities.

10

10

Iterators (STL)

- In STL, position represented by iterator

```
list<string>::iterator itr1;
vector<int>::iterator itr2;
```

  - … note that book will use *iterator* as a shorthand
- Basic operations

```
iterator begin();   // first item of container
iterator end();     // endmarker of container; position after last item; returns an iterator that is out of bounds

for (int i=0; i != v.size(); ++i) //no iterators
     cout << v[i] << endl;
```

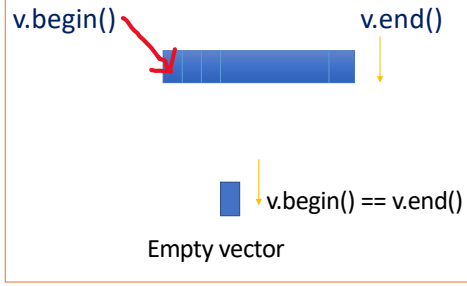
Tests if loop counter is out-of-bounds

11

11

Iterator semantics

vector<int> v; // Same semantics for list, etc.



12

12

Iterators (STL)

Basic operations

```
iterator begin();  // first item
iterator end();    // position after last item
```

```
// Print out a vector using iterators.
for (vector<int>::iterator itr = v.begin();  itr != b.end();
itr.??)
     cout << itr.?? << endl;
```

13

13

Iterator Methods

itr++ //advances the iterator itr to next location. Both prefix and postfix forms allowable
++itr
*itr // returns a reference to the object stored at itr's location. Returned reference may or may not be modifiable
itr1 == itr2 //returns true if iterators itrs 1 and itr2 refer to same location and false otherwise.
itr1 != itr2 //returns true if iterators itrs 1 and itr2 refer to different locations and false otherwise.

```
//Code to print
for (vector<int>::iterator itr = v.begin(); itr != v.end(); ++itr)
     cout << *itr << endl;
// or using operator overloading to access the current item and advance to next item using itr++
vector<int>::iterator itr = v.begin();
while (itr != v.end())
    cout << *itr++ << endl;
```

14

14

Operations that require iterators

```
// Insert prior
//constant time operator for list but not for vector; return val is
an iterator representing position of inserted item
iterator insert(iterator pos, const Object &x);

// Delete at, return next, invalidates pos
iterator erase(iterator pos);
//Removes all items from start up to but not including end
iterator erase(iterator start, iterator end);
//Removes entire list
c.erase(c.begin(),c.end());
```

- Example: Routine that erases every other item on list or vector. How would we program this?

16

16

Example: Using erase on a list

```
// @ lst: A given list, or any object type that supports iterators and
//        erase.
// The function deletes every other element from lst, starting from
// the first item.
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;
    }
}
```

17

17

Example: Using erase on a list

```
// @ lst: A given list, or any object type that supports iterators and
//        erase.
// The function deletes every other element from lst, starting from
// the first item.
template <typename Container>
void RemoveEveryOtherItem(Container &lst) {
    auto itr = lst.begin();
    while (itr != lst.end()) {
      itr = lst.erase(itr);
      if (itr != lst.end()) ++itr;
    }
}
```

18

18

Operations that require iterators

- More efficient for list or vector?
- List<int>
  - 0.039 sec for 800,000 item list; 0.073 for 1,600,000 item list
  - O(n) because each of the calls to erase take constant time
- Vector<int>
  - ~5 min for 800,000 vector of integers, ~20min for 1,600,000-item vector.
  - $O(n^2)$ each call on erase is inefficient taking O(N) time.

19

19

const_iterators

- Is there a need for a const iterator?
  - Note: *itr is a reference to the object at iterator's position
  - i.e. *itr is not just the value of the item that the iterator is viewing but also the item itself
- Check generic routine that runs for both vector and list:

```
//change all the items in collection to specified value
template <typename Container, typename Object>
void Change(Container &c, const Object &value) {
    auto itr = c.begin();
    while (itr != c.end())
        *itr++ = value;
}
```

- Good example of generic, type-independent code but some issues….

20

20

const_iterators

- Any problems?

```
void Print(const list<int> &lst, ostream & out = cout)
{
 typename Container::iterator itr = lst.begin(); // ERROR
    while (itr != lst.end()) {
        out  << *itr << endl;
        *itr = 0;//this is suspect
        itr++;
    }
}
```

21

21

Review C++ details: Parameter Passing

- C and Java use call-by-value: actual argument is copied into the formal parameter.
- C++ (large complex objects) – copying is inefficient and value may need to be changed.
- C++ has 3 ways to pass parameters
  - Call-by-value: Small objects that will not be changed by function.
    - double average(double a, double b);
  - Call-by-reference: All objects that may be changed by function.
    - void swap(double &a, double &b); //call-by-reference
  - Call-by-constant-reference : Large objects that will not be changed by function and are expensive to copy.
    - String randomItem(const vector<string> &arr); //call-by-constant-reference

22

22

const_iterators

- Life is good.

```
void Print(const list<int> &lst, ostream & out = cout)
{
   typename Container::const_iterator itr = lst.begin();
    while (itr != lst.end()) {
        out  << *itr << endl;
        // *itr = 0; can't change value
        itr++;
    }
}
```

23

23

const_iterators

- STL provides both iterator and const_iterator
- `operator*()` for const_iterator returns *constant reference*
  - `iterator begin();`
  - `const_iterator begin() const;`
  - `iterator end();`
  - `const_iterator end() const;`

Note: The two versions of begin can be in the same class only because of the const-ness of a method (whether an accessor or mutator) is considered to be part of the signature. Trick is overloading operator[]

24

24

Printing a container

```
// @ c: a given container.
// @ out: an output stream.
// Prints out the container on the output stream.
template <typename Container>
void Print(const Container &c, ostream &out = cout)  {
if (c.empty()) {
     out << "(empty)"
Else {
     typename Container::iterator itr = begin(c); //item is
a Container::const_iterator
      out << "[" << *itr++; //print first item
      while (itr != end(c))
         out << ", " << *itr++;
        out << " ]" << endl;
  }
}
```

25

25

STL also provides…

```
// Sample usage for lists:
// list<int> lst;
// if (begin(lst) == end(lst)) cout << "empty…" << endl;
// Already exists in the STL, not need to rewrite it.

template<typename Container>
auto begin(Container &c) -> decltype(c.begin()) {
      return c.begin();
}

template<typename Container>
auto begin(const Container &c) -> decltype(c.begin()) {
  return c.begin();
}
```

The return type for begin is deduced to be the type of c.begin()

Using begin(c) has the advantage that it allows generic code to work on containers that have begin/end members but also those that don't but can be later augmented with appropriate non-member functions.

26

26

Summary

- List, Stack, Queue ADT Intro
- Next week:
  - Implementation of vector, list, const iterator, error conditions
  - Trees
- To Do:
  - HW1 due today
  - Keep up with the reading. Expect to spend on average atleast 6 hours/week outside class. Chapter 3, 4.
  - Short BB multiple choice exercise next week up to Lecture 6 contents.

27

27