Ve 280

Programming and Introductory Data Structures

Standard Template Library:
Sequential Containers;
Container Adapters;
Associative Containers

Outline

- Two Other Sequential Containers: deque and list
- Container Adapters
- ADT: Dictionary
- Associative Container: `map`

deque

- Pronounced as “deck”. Means double-ended queue
- Based on arrays
- Supports fast random access.
- Fast insert/delete at front or back.
- To use, `#include <deque>`

Similarity between deque and vector

- Initialization method
  - `deque<T> d; deque<T> d(d1);`
  - `deque<T> d(n,t)`: create `d` with `n` elements, each with value `t`
  - `deque<T> d(b,e)`: create `d` with a copy of the elements from the range denoted by iterators `b` and `e`
- size(), empty()
- push_back(), pop_back()
- random access through subscripting: `d[k]`
- begin(), end(), insert(p, t), erase(p)
- Operations on iterators
  - *iter, ++iter, --iter, iter1 == iter2, iter1 != iter2, etc.

Differences of deque over vector

- It supports insert and remove at the beginning

- `d.push_front(t)`
  - Add element with value `t` to **front** of `d`

- `d.pop_front()`
  - Remove the **first** element in `d`

list

- Based on a doubly-linked lists
- Supports only bidirectional **sequential** access.
  - If you want to visit the 15th element, you need to go from the beginning and visit every one between the 1st and the 15th.
- Fast insert/delete at any point in the list.

- To use, `#include <list>`

Similarity between list and vector

- Initialization method
  - `list<T> l; list<T> l(li);`
  - `list<T> l(n,t)`: create `l` with `n` elements, each with value `t`
  - `list<T> l(b,e)`: create `l` with a copy of the elements from the range denoted by iterators `b` and `e`
- size(), empty()
- push_back(), pop_back()
- begin(), end()
- Operations on iterators
  - *iter, ++iter, --iter, iter1 == iter2, iter1 != iter2, etc.

Insert: insert(p, t)
Remove: erase(p)

Differences of list over vector

- Does not support subscripting

```
list<string> li(10, "abc");
li[1] = "def"; // Error!
```

- No iterator arithmetic for list

```
list<int>::iterator it;
it+3; // Error! To move, use ++/--
```

- No relational operation <, <=, >, >= on iterator of list

```
list<int>::iterator it1, it2;
it1 < it2; // Error!
           // To compare, use == or !=
```

Differences of list over vector

- It supports insert and remove at the beginning

- `l.push_front(t)`
  - Add element with value `t` to **front** of `l`

- `l.pop_front()`
  - Remove the **first** element in `l`

Which Sequential Container to Use?

- `vector` and `deque` are fast for random access, but are not efficient for inserting/removing at the middle
  - For example, removing leaves a hole and we need to shift all the elements on the right of the hole
  - For vector, only inserting/removing at the back is fast
  - For deque, inserting/removing at both back and front is fast

- `list` is efficient for inserting/removing at the middle, but not efficient for random access
  - It is based on linked list. Accessing an item requires traversal

General Rules of Thumb

- Use `vector`, unless you have a good reason to prefer another container.
- If the program requires random access to elements, use a `vector` or a `deque`.
- If the program needs to insert or delete elements in the middle, use a `list`.
- If the program needs to insert or delete elements at the front and the back, but not in the middle, use a `deque`.
- If the program needs both random access and inserting/deleting at the middle, the choice depends to the predominant operation (whether it does more random access or more insertion or deletion).

Outline

- Two Other Sequential Containers: deque and list
- Container Adaptors
- ADT: Dictionary
- Associative Container: `map`

Adaptor

- An adaptor is a mechanism for making one thing act like another
  - A container adaptor takes an existing container type and makes it act like a different abstract data type
- Three sequential container adaptors
  - stack ✓
  - queue ✓
  - priority_queue
- To use stack, `#include <stack>`
- To use queue, `#include <queue>`

Initializing an Adaptor

- `A a;`
  - The default adaptor. Create an empty object
  - E.g. `stack<int> stk;`
- `A a(c);`
  - Take a container and make a copy of that container as its underlying value
  - By default, both stack and queue are implemented using deque, so if you use a sequential container `c` to initialize, `c` must be a deque type

```
deque<int> deq(10, 1);
stack<int> stk(deq);
```

Initializing an Adaptor

- How do we use a vector to initialize a stack?

```
// Assume ivec is vector<int>
stack<int,vector<int> > stk(ivec);
```

Note the **space**. Otherwise, treated as an extraction operator – an error!

- We can use vector, list, and deque to build stack
- We can only use deque and list to build queue
  - Cannot use vector, because queue adaptor requires pop_front()

Operations of Stack Adaptor

- `s.empty()`
  - Returns true if the stack is empty; false otherwise
- `s.size()`
  - Returns a count of the number of elements of the stack
- `s.pop()`
  - Removes, but does not return, the top element from the stack
- `s.push(item)`
  - Places a new top element of the stack
- `s.top()`
  - Returns a reference to the top element of the stack

Note: although stack is implemented using another container, you cannot use other operations. For example, cannot call push_back().

Example

```
stack<int> stk;
for(int i=0; i<5; i++)
   stk.push(i);
while(!stk.empty()) {
   cout << stk.top() << endl;
   stk.pop();
}
```

What's the output?

Operations on Queue Adaptor

- `q.empty()`
  - Returns true if the queue is empty; false otherwise
- `q.size()`
  - Returns a count of the number of elements of the queue
- `q.push(item)`
  - Places a new element at the end of the queue
- `q.pop()`
  - Removes, but does not return, the front element from the queue
- `q.front()`
  - Returns a reference to the front element of the queue
- `q.back()`
  - Returns a reference to the back element of the queue

Outline

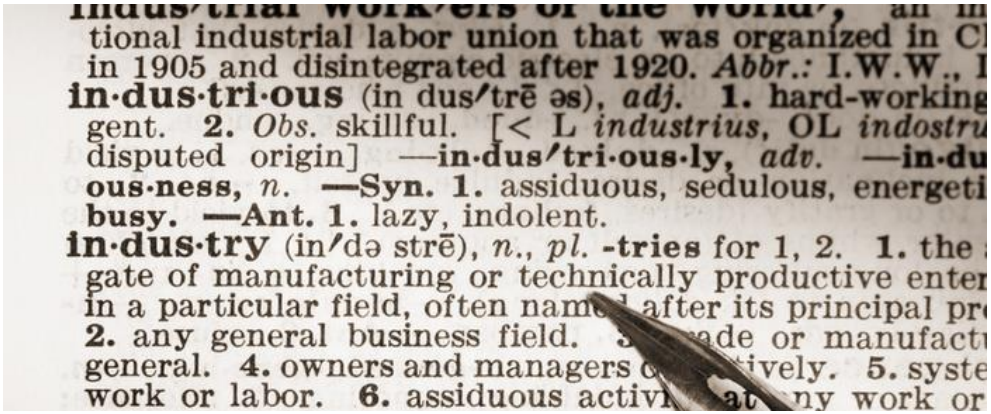
- Two Other Sequential Containers: deque and list
- Container Adaptors
- ADT: Dictionary
- Associative Container: `map`

Dictionary

- How do you use a dictionary?
  - Look up a "word" and find its meaning.
- We also have an **ADT** of dictionary.
  - It is a collection of pairs, each containing a **key** and an **value**

`(key, value)`

  - **Important**: Different pairs have different keys.


tional industrial labor union that was organized in C
in 1905 and disintegrated after 1920. *Abbr.*: I.W.W., I
in·dus·tri·ous (in dus′trē əs), *adj.* **1.** hard-working
gent. **2.** *Obs.* skillful. [< L *industrius*, OL *indostru*
disputed origin] —**in·dus′tri·ous·ly**, *adv.* —**in·du**
ous·ness, *n.* —**Syn. 1.** assiduous, sedulous, energeti
busy. —**Ant. 1.** lazy, indolent.
in·dus·try (in′də strē), *n.*, *pl.* **-tries** for 1, 2. **1.** the
gate of manufacturing or technically productive enter
in a particular field, often nam after its principal pr
2. any general business field. ade or manufact
general. **4.** owners and managers tively. **5.** syste
work or labor. **6.** assiduous activi at ny work or

Dictionary

- Key space is usually more regular/structured than value space, so easier to search.

- Dictionary is optimized to quickly **add** **`(key, value)`** pair and **retrieve value** by key.

Methods

- **`Value find(Key k)`**: Return the value whose key is **`k`**. Return **`Null`** if none.
- **`void insert(Key k, Value v)`**: Insert a pair **`(k, v)`** into the dictionary. If the pair with key as **`k`** already exists, update its value.
- **`Value remove(Key k)`**: Remove the pair with key as **`k`** from the dictionary and return its value. Return **`Null`** if none.
- **`int size()`**: return number of pairs in the dictionary.

Example

- Collection of student records in the class
  - (key, value) = (student name, linear list of assignment and exam scores)
  - All keys are distinct

- Operations
  - Get the value whose key is John Adams.
  - Insert a record for the student whose name is Diana Ross.

Implementation

- Method #1: using an array
  - Just like our **`IntSet`**
  - The difference is that each array element is a **`(key, value)`** pair
  - If keys can be sorted, can use either sorted array or unsorted array on keys
- Method #2: using a linked list
  - Each node now stores both the key and value
  - The differences over **`IntList`** are:
    - When inserting, it needs to verify there is no duplicated key. If key already exists, update the value entry
    - It removes on key, not just the first and last element

Outline

- Two Other Sequential Containers: deque and list
- Container Adaptors
- ADT: Dictionary
- Associative Container: `map`

Introduction

- Elements in an associative container are stored and retrieved by a **key**, in contrast to elements in a sequential container, which are stored and accessed sequentially by their position within the container
- Two primary associative container types: `map` and `set`
  - Elements in a `map` are (key, value) pairs
  - `set` contains only a key and supports efficient queries to whether a given key is present
- Applications
  - `map`: dictionary
  - `set`: store a collection of distinct values efficiently. For example, the distinct English words in an article

Associative vs. Sequential Containers

- Associative containers share many, but not all, of the operations on sequential containers
  - They do not have the front(), push_front(), pop_front(), back(), push_back(), or pop_back() operations
- Common operations
  - `C<T> c; // creates an empty container`
  - `C<T> c1(c2); // copies elements from c2 into c1`
    `// c2 must be the same type as c1`
  - `C<T> c(b, e); // b and e are iterators denoting a`
    `// sequence. Copy elements from the sequence into c`
  - begin(), end(), size(), empty(), clear(), =

Associative vs. Sequential Containers

- The associative container types define additional operations
- The big difference: for associative containers, elements are ordered by key
- There is one important consequence of the fact that elements are ordered by key:
  - When we iterate across an associative container, we are guaranteed that the elements are **accessed in key order**, irrespective of the order in which the elements were placed in the container.

Map

- map is also known as **associative array**
- It stores (key, value) pair
- To use, `#include <map>`
- Constructors
  - `map<k, v> m;` `// Create an empty map named m`
    `// with key and value types k and v.`
  - E.g., **`map<string,int> word_count`**
  - `map<k, v> m(m2);` `// Create m as a copy of m2;`
    `// m and m2 must have the same key and value types`
  - `map<k, v> m(b, e);` `//Create m as a copy of the`
    `//elements from the range denoted by iterators b and e`

Constraints on the Key Type

- Since elements in map are ordered by keys, we require that key type has an extra operation: **strict weak ordering**

Examples:
- < for int
- alphabetical order for string

- Strict weak ordering:
  - Think as less than (<)
- Technically
  - Yield false when we compare a key with itself
  - Given two keys, they cannot both be "less than" each other
  - Satisfy transitive property: if k1<k2 and k2<k3, then k1<k3
  - If we have two keys, neither of which is "less than" the other, then they are treated as equal

Preliminaries: the pair Type

- A simple companion type, holding two data values
- It is a template. Need to supply two type names

```
pair<string,string> spair; // hold two strings
```

- `pair<T1, T2> p1;`
  - Create a pair with two elements of types T1 and T2. The elements are value-initialized (use default constructor for class type; 0 for built-in type)
- `pair<T1, T2> p1(v1, v2);`
  - Create a pair with types T1 and T2. Initialize the first member from v1 and the second from v2.
  - **`pair<string, int> count("blue", 2);`**

Preliminaries: the pair Type

- We can access the two data members in the pair
  - `p.first //` return the **reference** to the first member
  - `p.second //` return the **reference** to the second member
  - They are **public**

- `make_pair(v1,v2)`
  - Create a new pair from the values v1 and v2. The type of the pair is inferred from the types of v1 and v2

```
pair<string,string> name = make_pair("John", "Adams");
```

Map Iterator

- Dereferencing a map iterator yields a **pair** in which first member holds the **const key** and second member holds the **value**

```
map<string, int>::iterator it =
                                word_count.begin();
```

- ***it** is a reference to a **pair<const string, int>** object
  - It refers to neither the key nor the value
- To access key, use **it->first**

```
cout << it->first;
```

- However, first member is a **const key**, so we cannot change it

```
it->first = "new key"; // Error!
```

Map Iterator

```
map<string, int>::iterator it =
                                word_count.begin();
```

- To access value, use **it->second**

```
cout << it->second;
```

- We can change value through iterator

```
it->second = 2;
```

Adding Elements to a map

- There are two ways:
  - Using the subscript operator
  - Using the insert member

Insert Using Subscripting

- If key `k` is not in the map `m`, you can insert `(k,v)` using

```
m[k] = v;
```

- Example

```
map <string, int> word_count; // empty map
// insert element with key "Anna";
// then assign 1 to its value
word_count["Anna"] = 1;
```

- You insert a pair **`("Anna", 1)`** into **`word_count`**.

Insert Using Subscripting

```
map <string, int> word_count; // empty map
// insert element with key "Anna";
// then assign 1 to its value
word_count["Anna"] = 1;
```

- What really happens is
  - **`word_count`** is searched for the element whose **`key`** is **`Anna`**. The element is not found.
  - A new (key, value) pair is inserted. key = "Anna". Value is value-initialized to 0.
  - The newly inserted element is fetched and is given the value 1.

Subscripting a map

- Subscripting a map behaves quite differently from subscripting an array or vector
  - Using an index (key) that **does not exist** adds an element with that index to the map
- If the key exists, the value associated with the key is returned. We can read and write to the value

```
cout << word_count["Anna"];
++word_count["Anna"]; // fetch the element
                      // and add one to it
```

- Subscripting a vector = dereferencing a vector iterator
- Subscripting a map ≠ dereferencing a map iterator

Use Subscript Behavior in a Smart Way

```
// count #times each word occurs from input
map<string, int> word_count;
// empty map from string to int
string word;
while (cin >> word)
  ++word_count[word];
```

Question: what's the behavior for the first time we encounter a word?

- The first time we encounter a word, a new element indexed by word is created and inserted into map
  - Its value is initialized with zero
- Then, the value of that element is immediately incremented. So, the count is the (correct) value of one
- If word is already in the map, then its value is incremented.

insert()

- `m.insert(e)`
  - `e` is a (key, value) pair. If the key is not in `m`, insert the pair. If the key is in `m`, then `m` is unchanged

```
word_count.insert(make_pair("Anna", 1));
```

insert()

- `m.insert(e)`
  - Returns a pair of (map iterator, bool)
    - map iterator refers to the element with key
    - bool indicates whether the element was inserted or not.

```
// count #times each word occurs from input
map<string, int> word_count;
while (cin >> word) {
  pair<map<string, int>::iterator, bool> ret =
      word_count.insert(make_pair(word, 1));
  if (!ret.second) // word already in word_count
      ++ret.first->second; // increment count
}
```

Finding and Retrieving a map Element

- The subscript operator provides the simplest method of retrieving a value
- But, it has a side effect. What is it?
  - If that key is not already in the map, then subscript inserts an element with that key.
- How can we determine if a key is present without causing it to be inserted?
  - **`m.find(k)`**

find()

- `m.find(k)`
  - Returns an iterator to the element indexed by key `k`, if there is one
  - Otherwise, returns an off-the-end iterator (i.e., end()) if the key is not present

```
int occurs = 0;
map<string,int>::iterator it =
  word_count.find("abc");
if (it != word_count.end())
  occurs = it->second;
```

erase()

- `m.erase(iter)`
  - Removes element referred to by the iterator `iter` from `m`. `iter` must refer to an actual element in `m`; it must not be equal to `m.end()`.
  - Returns void.
- `m.erase(k)`
  - Removes the element with key `k` from `m` if it exists
  - Otherwise, do nothing
  - Returns the number of elements removed. For map, this is either 0 or 1

```
if (word_count.erase(rm_word)) // rm_word is a key
  cout << "ok: " << rm_word << "removed\n";
else cout << rm_word << " not found!\n";
```

Iterate across a map

- map has `begin()` and `end()`, with which we can traverse the map
- Example: print all the elements in `word_count`

```
map<string, int>::iterator it;
for(it=word_count.begin();
    it!=word_count.end(); ++it)
    cout << it->first << " occurs "
      << it->second << " times";
```

- The output prints the words in **alphabetical order**.
  - **Note**: When we use an iterator to traverse a map, the iterators yield elements in **ascending key order**.

Reference

- **C++ Primer (4th Edision)**, by *Stanley Lippman, Josee Lajoie, and Barbara Moo*, Addison Wesley Publishing (2005)
  - Chapter 3.3 Library vector Type
  - Chapter 9 Sequential Containers
  - Chapter 10 Associative Containers