Ve 280

Programming and Introductory Data Structures

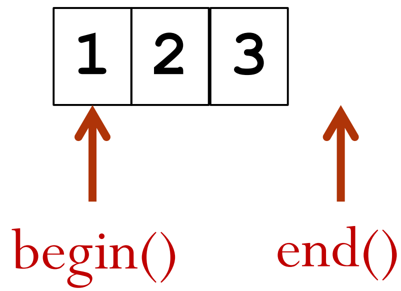
Standard Template Library:
Sequential Containers;
Container Adapters;
Associative Containers

Outline

- STL Sequential Container: `vector`
  - Iterator
  - Operations with Iterator
- Two Other Sequential Containers: deque and list
- Container Adapters
- ADT: Dictionary
- Associative Container: `map`

Review: Iterator

- `vector<int>::iterator it;`
- a generalization of pointer
- begin(), end(), dereference, ++, ==



Iterator Arithmetic

- vector supports iterator arithmetic
  - Not all containers support iterator arithmetic
- `iter+n`, `iter-n`
  - `n` is an integral value
  - adding (subtracting) a value `n` to (from) an iterator yields an iterator that is `n` positions forward (backward)
- We can use iterator arithmetic to move an iterator to an element directly
  - Example: go to the middle

```
vector<int>::iterator mid;
mid = vi.begin() + vi.size()/2;
```

Relational Operation on Iterator

- >, >=, <, <=
- E.g., `while(iter1 < iter2)`
- One iterator is less than (<) another if it refers to an element whose position in the container is **ahead** of the one referred to by the other iterator.

- To compare, iterators must refer to elements in the **same** container or one past the end of the container (i.e., `c.end()`).

Outline

- STL Sequential Container: `vector`
  - Iterator
  - Operations with Iterator
- Two Other Sequential Containers: deque and list
- Container Adaptors
- ADT: Dictionary
- Associative Container: `map`

Initializing with a Range of Elements

- `vector<T> v(b, e);`
  - Create vector `v` with a copy of the elements from the range denoted by iterators `b` and `e`
- <u>**Note**</u>: **iterator range** is denoted by a pair of iterators `b` and `e` that refer to two elements, or to one past the last element, in the same container.
  - <u>**Note**</u>: the range includes `b` and each element **up to but not including** `e`.
  - It is denoted as `[b, e)`
  - If `b = e`, the range is empty
  - If `b=v.begin()`, `e=v.end()`, the range includes all the elements in `v`

Initializing with a Range of Elements

- We can use this form of initialization to copy only a subsequence of the other container
- Example

```
// assume v is a vector<int>
vector<int>::iterator mid;
mid = v.begin()+ v.size()/2;

// front includes the 1st half of v, from begin
// up to but not including mid
vector<int> front(v.begin(), mid);

// back includes the 2nd half of v from mid
// to end
vector<int> back(mid, v.end());
```

Initializing with a Range of Elements

- `vector<T> v(b, e);`

- We can even use another container type to initialize

```
deque<string> ds(10, "abc");
vector<string> vs(ds.begin(), ds.end());
```

Initializing with a Range of Elements

- Since pointers are iterators, the iterator range can also be a pair of pointers into a built-in array

```
int a[] = {1, 2, 3, 4};
unsigned int sz = sizeof(a)/sizeof(int);
vector<int> vi(a, a+sz);
```

- Note
  - **`sizeof(obj)`**, **`sizeof(type name)`**: return the size in bytes of an object or type name
  - If **`obj`** is an array name, **`sizeof(obj)`** is the total size in byte in that array
- <u>Question</u>: what is the value of **`sz`**?

Initializing with a Range of Elements

```
int a[] = {1, 2, 3, 4};
unsigned int sz = sizeof(a)/sizeof(int);
vector<int> vi(a, a+sz);
```

- `a` points to the first element in array `a`
- `a+sz` points to the location one past the end of array `a`
- Thus, the entire array `a` is copied

Another Way to Add Value: insert()

- `v.insert(p,t)`
  - Inserts element with value `t` **before** the element referred to by iterator `p`.
  - Returns an iterator referring to the element that was added.
- We can use insert to insert at the beginning of vector

```
vector<int> iv(2, 1);
iv.insert(iv.begin(), -1);
```

- We can also insert at the end

```
iv.insert(iv.end(), 3);
```

Erase Element: erase()

- `v.erase(p)`
  - Removes element referred to by iterator `p`
  - Returns an iterator referring to the element **after** the one deleted, or an **off-the-end** iterator if p referred to the last element
  - p cannot be an **off-the-end** iterator
  - Example use: find an element and erase it

Outline

- STL Sequential Container: `vector`
  - Iterator
  - Operations with Iterator
- Two Other Sequential Containers: deque and list
- Container Adaptors
- 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()
- Operations on iterators
  - *iter, ++iter, --iter, 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, etc.

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!
```

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

```
list<int>::iterator it1, it2;
it1 < it2; // Error!
```

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 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 access or more insertion or deletion).

Outline

- STL Sequential Container: `vector`
  - Iterator
  - Operations with Iterator
- 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 the container `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 push_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, but does not remove, 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, but does not remove, the front element of the queue
- `q.back()`
  - Returns, but does not remove, the back element of the queue

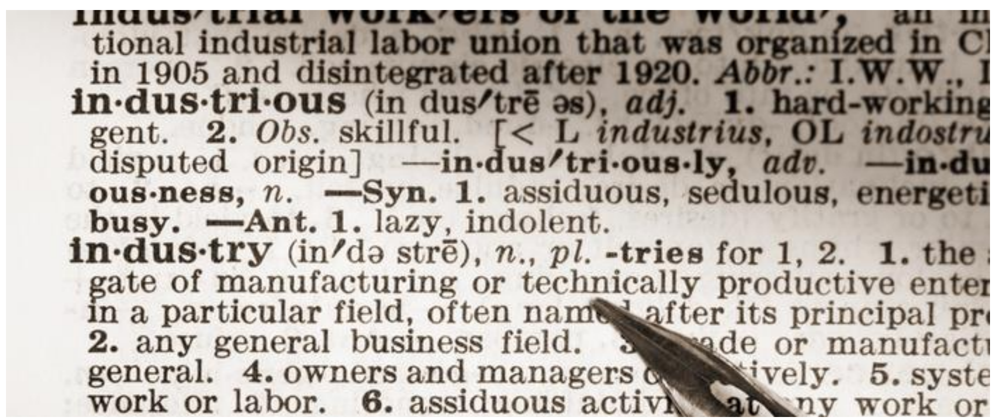
Outline

- STL Sequential Container: `vector`
  - Iterator
  - Operations with Iterator
- 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)`
  - <u>**Important**: Different pairs have different keys.</u>



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

- STL Sequential Container: `vector`
  - Iterator
  - Operations with Iterator
- 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 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;
```

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