ECE2800J

Programming and Elementary Data Structures

Standard Template Library:

Sequential Containers

Learning Objectives:

Know how to use the STL sequential containers

Know which one to choose for a specific application

Outline

- Overview of Standard Template Library
- STL Sequential Container: `vector`
  - Some Basic Operations
  - Iterator
  - Operations with Iterator
- Two Other Sequential Containers: `deque` and `list`

Standard Template Library (STL)
Overview

- We have talked about containers
  - C++ has a **standard template library (STL)** that provides us with an easy way to define containers
- STL defines powerful, template-based, reusable components that implements common data structures and algorithms
- Divided into three components:
  - Containers: data structures that hold a collection of objects of a specified type
  - Iterators: used to examine and navigate container elements
  - Algorithms: searching, sorting and many others

Containers in STL

- The STL provides three kinds of containers:
  - **Sequential Containers**: let the programmer control the order in which the elements are stored and accessed. The order does not depend on the values of the elements
  - **Associative Containers**: store elements based on their values. The order depends on the value of the elements
  - **Container Adapters**: take an existing container type and make it act like a different type

Sequential Containers

- There are three sequential containers:
  - `vector`: based on arrays.
    - Supports fast random access.
    - Fast insert/delete at the back. Inserting or deleting at other position is slow.
  - `deque` (double-ended queue): based on arrays.
    - Supports fast random access.
    - Fast insert/delete at front or back.
  - `list`: based on a doubly-linked lists
    - Supports only bidirectional **sequential** access.
    - Fast insert/delete at any point in the list.

Which statements are true?

Select all the correct answers.

- **A.** As the STL provides an implementation of sequential containers, there's no reason to provide new implementations for them.
- **B.** We should use the STL containers when possible.
- **C.** A container need not be sequential.
- **D.** None of the above.

Outline

- Overview of Standard Template Library
- STL Sequential Container: `vector`
  - Some Basic Operations
  - Iterator
  - Operations with Iterator
- Two Other Sequential Containers: `deque` and `list`

Vector

- `vector` is a widely used STL container
  - A collection of objects of a **single** type, each of which has an associated integer index.
  - We can create a vector of ints, a vector of strings, etc.

- To use a vector, include the appropriate header and namespace.

```
#include <vector>
using namespace std;
```

Vector

- vector is a template. We need to specify the type of objects the vector contains.

```
vector<int> ivec; // holds ints
vector<IntSet> isvec; // holds IntSets
```

Initializing Vector

- `vector<T> v1;`
  - Construct an **empty** vector `v1` that holds objects of type `T`
  - E.g., `vector<int> v1;`
- `vector<T> v2(v1);`
  - Copy constructor.
  - E.g., `vector<int> v2(v1);`
- `vector<T> v3(n, t);`
  - Construct `v3` that has `n` elements with value `t`.
  - E.g., `vector<int> v3(10, -1);`
  - `vector<string> v4(2, "abc");`

Size of Vector

- `v.size() // number of elements in v`
- `size()` return a value of `size_type` corresponding to the vector type.
- `vector<int>::size_type`
  - A **companion type** of vector
  - Essentially an unsigned type (unsigned int or unsigned long)
  - **Note**: not `vector::size_type`
- Why companion types?
  - To make the type machine-independent

Size of Vector

- Generally, you can convert `size_type` into `unsigned int`

```
unsigned int s = v.size();
```

- However, using `int` is not recommended

```
int s = v.size(); // not good
```

- If you only want to know whether the vector is empty or not, you can use
  - `v.empty() // true if v is empty`

Add/Remove Element to/from Vector

- Add: `v.push_back(t)`
  - Add element with value `t` to **end** of `v`
- Example

```
vector<int> v;
for(int i = 0; i <5; i++)
     v.push_back(i);
// v is 0,1,2,3,4
```

- Remove: `v.pop_back()`
  - Remove the last element in `v`. No argument. Returns void. `v` must be non-empty

Container Elements Are Copies

- There is no relationship between the element in the container and the value from which it was copied.
- What is the value of `v[0]`?

```
vector<int> v;
int a = 3;
v.push_back(a); // v[0] is 3 now
a = 5; // What is v[0] now?
```

- Subsequent changes to the value that was copied have no effect on the element in the container, and vice versa.

Subscripting Vector

- `v[n]`: returns element at position `n` in `v`

```
vector<int>::size_type ix;
for(ix=0; ix!=ivec.size(); ++ix)
 ivec[ix]=0;
```

- Subscripting does not add elements.

```
vector<int> ivec; // empty vector
for(vector<int>::size_type ix=0; ix!=10; ++ix)
  ivec[ix] = 0; // Error!
```

- An element must exist in order to subscript it.

Good Practice

```
vector<int>::size_type ix;
for(ix=0; ix!=ivec.size(); ++ix)
  ivec[ix]=0;
```

- **Note**: we call the `size` member in the `for` rather than calling it once before the loop and remembering its value.
- Why?
  - Because vector can grow dynamically by adding new elements
  - By putting `size` in `for`, we test on the most current size. It is safer.
- Will it be slow?
  - No! size() is an inline function
  - Inline function: expanded "in line". Avoid function call overhead.

Other Basic Operations on Vector

- `v1 = v2` //replace elements in v1 by a copy of
  // elements in v2
- `v.clear()` // makes vector v empty
- `v.front()` // Returns a reference to the first element
  // in v. v must be non-empty!
- `v.back()` // Returns a reference to the last element in v.
  // v must be non-empty!

Outline

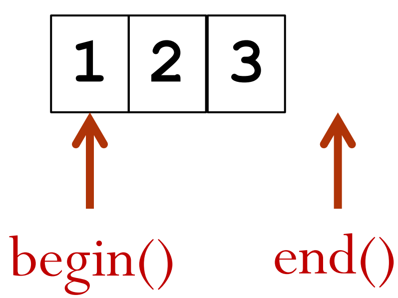
- Overview of Standard Template Library
- STL Sequential Container: `vector`
  - Some Basic Operations
  - Iterator
  - Operations with Iterator
- Two Other Sequential Containers: deque and list

Iterators

- Each container type has a companion **iterator** type.
  - It lets us examine elements and navigate in the container.
- Iterators are more general than subscripts: All of the library containers define iterator types, but only a few of them support subscripting.
- Declare an iterator for vector:
  - E.g., `vector<int>::iterator it;`
- An iterator is a generalization of pointer.
  - They are pointers to the elements of containers.

How to Link Iterator to Vector?

- Use two member functions `begin()` and `end()` of vector
- `v.begin()` returns an iterator pointing to the first element of vector
  - `vector<int>::iterator it = v.begin();`
- `v.end()` returns an iterator positioning to **one-past-the-end** of the vector
  - It does not denote an actual element in vector



end()

- `v.end()` is used to indicate when we have processed all the elements in vector

- If the vector is empty, the iterator returned by `begin` is the same as the iterator returned by `end`

Operations on Iterator

- Dereference operator
  - `*iter`: let us access the element to which the iterator refers
  - You can **read**/**write** through `*iter`
- Increment/decrement operator
  - `++iter`, `iter++`: advance to the next item in vector
  - `--iter`, `iter--`: go back to the previous item

Note: you cannot dereference or increment iterator returned by end()

- `iter == iter2` and `iter != iter2`: test whether two iterators point to the same data item

Example

- Sum all the elements of the `vector<int> ivec`.

```
int sum = 0;
vector<int>::iterator it;
for(it=ivec.begin(); it != ivec.end(); ++it)
   sum += *it;
```

- **Question**: what happens when `ivec` is empty? what is the sum?
- Why using iterator instead of subscripting?
  - All container types have associated iterator types, but not all of them have subscripting.

Aside: `auto` keyword

- Used for type inference. Allows developers to write less code
- Enables developers to declare a variable without explicitly specifying its data type. Instead, the compiler deduces the type of the variable from the initialization expression

- Sum all the elements of the `vector<int> ivec`.

```
int sum = 0;
for(auto it=ivec.begin(); it != ivec.end(); ++it)
// auto is of the type vector<int>::iterator
  sum += *it;
```

Aside: `auto` keyword

- Some more examples

```
auto i = 5; // i is int type
auto k = 'C'; // k is char type
```

- An auto variable cannot be left uninitialized because its type is deduced from the initializer:

```
auto a; // ERROR
```

Aside: Range-based Loops

```
for(range_declaration : range_expression)
    loop_statement
```

- **`range_expression`**
  - Any expression that represents a suitable sequence
- **`range_declaration`**
  - A declaration of a named variable, whose type is the type of the element of the sequence represented by **`range_expression`**, or a reference to that type
  - Often uses the `auto` specifier for automatic type deduction

Aside: Range-based Loops

- Sum all the elements of the `vector<int> ivec`.

```
int sum = 0;
for(int v : ivec) // also: for(auto v : ivec)
   sum += v;
```

- Use reference to change value in the sequence

```
for(int &v : ivec)
   v = 42;
```

const_iterator

- Using iterator could change the values in the vector.
- `const_iterator` is another iterator type. However, it **cannot** be used to change values.
  - It can only be used for reading, but not writing to, the container elements …
  - … because dereferencing a const_iterator is a const object.
  - Note: its own value can be changed, e.g., we can increment it.

```
vector<string>::const_iterator it;
for(it=text.begin(); it!=text.end(); ++it) {
  cout << *it << endl; // fine
  *it = " "; // error: *it is const
}
```

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)`
- vector supports relational operation on iterator
  - Not all containers support relational operation on iterator
- 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

- Overview of Standard Template Library
- STL Sequential Container: `vector`
  - Some Basic Operations
  - Iterator
  - Operations with Iterator
- Two Other Sequential Containers: deque and list

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`
- **Note**: **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.
  - **Note**: 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=x.begin()`, `e=x.end()`, the range includes all the elements in `x`

Initializing with a Range of Elements

- We can use this form of initialization to copy just 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` **right 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

- Overview of Standard Template Library
- STL Sequential Container: `vector`
  - Some Basic Operations
  - Iterator
  - Operations with Iterator
- Two Other Sequential Containers: `deque` and `list`

deque

- Pronounced as "deck". Means <u>d</u>ouble-<u>e</u>nded <u>que</u>ue
- 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).

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