Programming in Modern C++

Module M44: C++ Standard Library (STL): Part 2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Overview of Standard Library components of C++
- Learnt fundamentals of generic programming

Module Objectives

- To understand Standard Template Library (STL)
- To understand common containers (data structure) and their use

Module Outline

1. The STL
   - Policy Parameterization
2. Common Standard Library Components
   - `vector`
   - `list`
   - `map`
   - `set`
3. Module Summary

The STL

Source:

- Chapter 20 The STL (containers, iterators, and algorithms), Bjarne Stroustrup
- Chapter 21 The STL (maps and algorithms), Bjarne Stroustrup

STL: Standard Template Library

- Part of the ISO C++ Standard Library
- Has four components:
  - `containers`
  - `iterators`
  - `algorithms`
  - `functions`
- Mostly non-numerical
  - Only 4 standard algorithms specifically do computation
    - `accumulate`, `inner_product`, `partial_sum`, `adjacent_difference`
- Handles textual data as well as numeric data
  - For example, `string`
- Deals with organization of code and data
  - Built-in types, user-defined types, and data structures
- Optimizing disk access is among its original uses
  - Performance is always a key concern

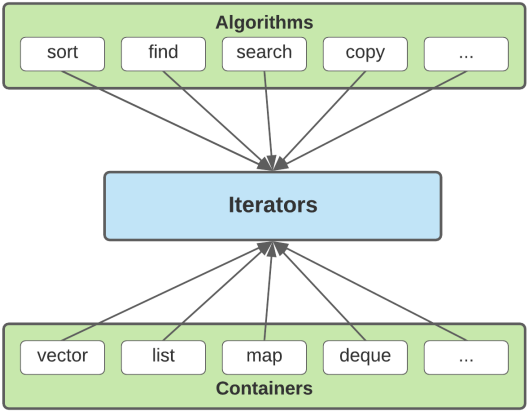
The STL

- Designed by Alex Stepanov
- General aim: *Most general*, *most efficient*, *most flexible* representation of concepts (ideas, algorithms)
  - Represent separate concepts separately in code
  - Combine concepts freely wherever meaningful
- General aim to make programming *like math*
  - or even *Good programming is math*
  - works for integers, for floating-point numbers, for polynomials, for ...

The STL

- An ISO C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org: Boost provides free peer-reviewed portable C++ source libraries. It has several containers include a number of non-standard containers (like `stable_vector`, `flat_(multi)map/set` associative containers, `slist`, `static_vector`, and `small_vector`
    - MSVC STL: Microsoft VC++ Standard Library has been released as open source: Open Sourcing MSVC's STL, 2019
    - Dinkumware Standard C++ Library
    - SGI
- The best known and most widely used example of generic programming

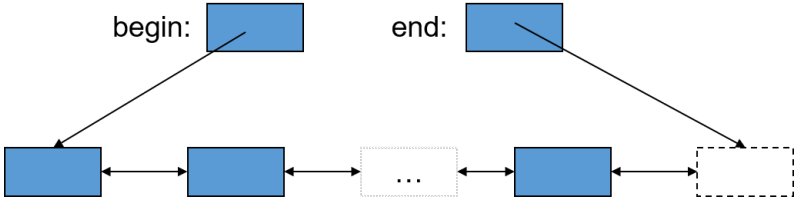
Basic Model: Algorithms ==> Iterators <== Containers: Recap (Module 43)



- **Separation of Concerns**
  - *Algorithms* manipulate data, but do not know about *Containers*
  - *Containers* store data, but do not know about *Algorithms*
  - *Algorithms* and *Containers* interact through *Iterators*
  - Each *Container* has its *own iterator types*

Basic Model: Iterators: Recap (Module 43)

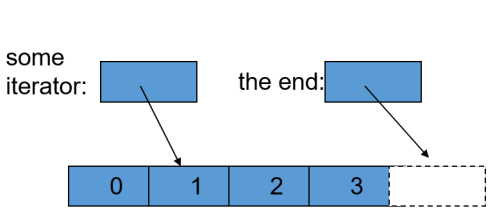
- A pair of iterators defines a sequence
  - The *beginning* (points to the *first element* – if any)
  - The *end* (points to the *one-beyond-the-last element*)

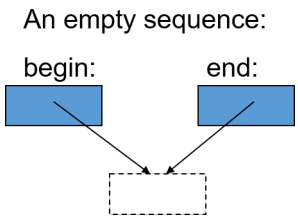


- An iterator is a type that supports the *iterator operations*
  - `++` Go to next element
  - `*` Get value
  - `==` Does this iterator point to the same element as that iterator?
- Some iterators support more operations (for example, `--`, `+`, and `[ ]`)

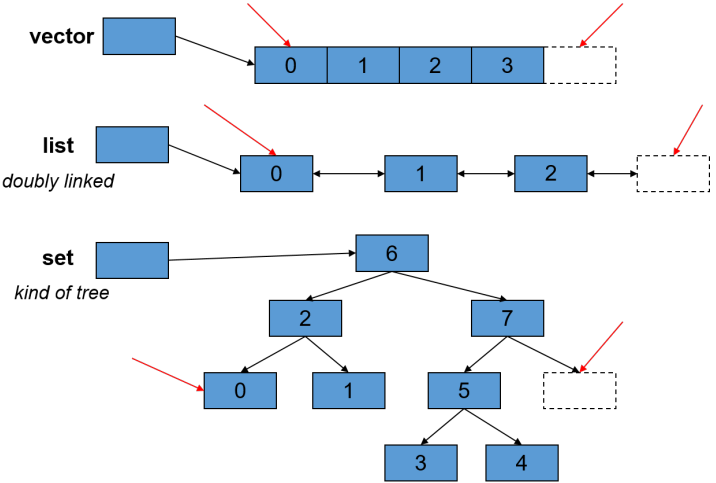
Basic Model: Algorithms + Iterators: Recap (Module 43)

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is *one past the last element*
  - not *the last element*
  - That is necessary to elegantly represent an empty sequence
  - One-past-the-last-element is not an element
    - ▷ You can compare an iterator pointing to it
    - ▷ You cannot dereference it (read its value)
- Returning the end of the sequence is the standard idiom for *not found* or *unsuccessful*





Basic Model: Containers + Iterators: Recap (Module 43)



Algorithm: find(): Recap (Module 43)


```
// Find the first element that equals a value
template<class In, class T>
In find(In first, In last, const T& val) {
    while (first != last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x) { // works for vector of ints
    vector<int>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end())  /* we found x */
    // ...
}

void f(list<string>& v, string x) { // works for list of strings
    list<string>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end())  /* we found x */
    // ...
}

void f(set<double>& v, double x) { // works for set of doubles
    set<double>::iterator p = find(v.begin(), v.end(), x);
    if (p != v.end())  /* we found x */
    // ...
}
```

Algorithm: find_if(): Recap (Module 43)

```
// Find the first element that matches a criterion (predicate)
template<class In, class Pred>
In find_if(In first, In last, Pred pred) {
    while (first != last && !pred(*first)) ++first;
    return first;
}

void f(vector<int>& v) {
    vector<int>::iterator p = find_if(v.begin(), v.end, Odd()); // Here, a predicate takes
                                                                // one argument and returns a bool
    if (p != v.end()) { /* we found an odd number */ }
    // ...
}
```

- A predicate (often of one argument) is a function or a function object returns a `bool` given the argument/s. For example

```
// A function
bool odd(int i) { return i % 2; } // % is the remainder (modulo) operator
odd(7);                           // call odd: is 7 odd?

// A function object (Module 40)
struct Odd { bool operator()(int i) const { return i % 2; } };
Odd odd;    // make an object odd of type Odd
odd(7);     // call odd: is 7 odd?
```

Policy Parameterization

- When we have a useful algorithm, we may want to *parameterize* it by a *policy*
  - For example, we need to *parameterize sort* by the *comparison criteria*

```
struct Record {
    string name;   // standard string for ease of use
    char addr[24]; // old C-style string to match database layout
    // ...
};
vector<Record> vr;
// ...
sort(vr.begin(), vr.end(), Cmp_by_name()); // sort by name
sort(vr.begin(), vr.end(), Cmp_by_addr()); // sort by addr

// Different comparisons for Rec objects
struct  Cmp_by_name {
    bool operator()(const Record& a, const Record& b) const
        { return a.name < b.name; } // look at the name field of Record
};
struct  Cmp_by_addr {
    bool operator()(const Record& a, const Record& b) const
        { return 0 < strncmp(a.addr, b.addr, 24); } // look at the addr field of Record
};
// Note how the comparison function objects are used to hide ugly and error-prone code
```

Policy Parameterization

- *Lambda* or *closure object* may be used to parameterize a *policy*

```
vector<Record> vr;
// ...
sort(vr.begin(), vr.end(),
    [] (const Record& a, const Record& b) // lambda expression as policy [C++11]
        { return a.name < b.name; }       // sort by name
    );

sort(vr.begin(), vr.end(),
    [] (const Record& a, const Record& b)           // lambda expression as policy [C++11]
        { return 0 < strncmp(a.addr, b.addr, 24); } // sort by addr
    );

// A lambda expression is an anonymous function - a function without a name
```

Policy Parameterization

- Use a *named object* as argument
  - If you want to do something complicated
  - If you feel the need for a comment
  - If you want to do the same in several places
- Use a *lambda expression* as argument [C++11]
  - If what you want is short and obvious
- Choose based on *clarity of code*
  - There are no performance differences between function objects and lambdas
  - Function objects (and lambdas) tend to be faster than function arguments

Common Standard Library Components

Common Standard Library Headers

- `<iostream>` `// I/O streams, cout, cin, ...` (**Module 42**)
- `<fstream>` `// file streams` (**Module 42**)
- Containers
  - `<string>` `// many` **Modules**
  - `<vector>` `// many` **Modules**
  - `<map>`
  - `<list>`
  - `<set>`
  - `<unordered_map>` `// hash table [C++11]`
  - ...
- `<algorithm>` `// sort, copy, ...`
- `<numeric>` `// accumulate, inner_product, ...`
- `<functional>` `// function objects` (**Module 40**)

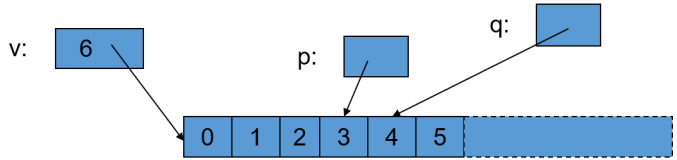
More components will be covered in weeks 10-12 in the course of discussions on [C++11] onward

vector

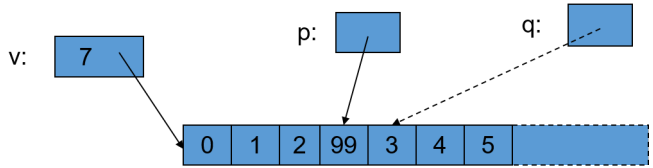

```
template<class T> class vector {
    T* elements;
    // ...
    using value_type = T;
    using iterator = ???; // the type of an iterator is implementation defined
                          // and it (usefully) varies (for example, range checked iterators)
                          // a vector iterator could be a pointer to an element
    using const_iterator = ???;

    iterator begin();             // points to first element
    const_iterator begin() const;

    iterator end();               // points to one beyond the last element
    const_iterator end() const;

    iterator insert(iterator p, const T& v); // insert a new element v before p

    iterator erase(iterator p);              // remove element pointed to by p
};
```

insert() into vector

```
vector<int>::iterator p = v.begin(); ++p; ++p; ++p;
vector<int>::iterator q = p; ++q;
```

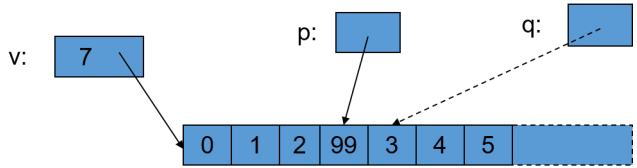



```
p = v.insert(p,99); // leaves p pointing at the inserted element
```

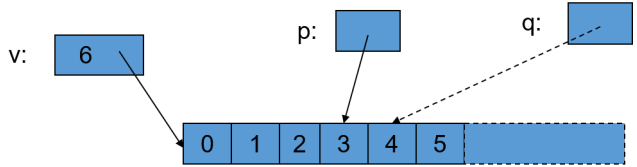



```
// Note: q is invalid after the insert()
// Note: Some elements moved; all elements could have moved
```

erase() from vector




```
p = v.erase(p); // leaves p pointing at the element after the erased one
```




```
// Note: vector elements move when we insert() or erase()
// Note: Iterators into a vector are invalidated by insert() and erase()
```

Ways of traversing a vector


```
for(int i = 0; i < v.size(); ++i)                          // index style. why int?
    ... // do something with v[i]

for(vector<T>::size_type i = 0; i < v.size(); ++i)         // index style, always correct
    ... // do something with v[i]

for(vector<T>::iterator p = v.begin(); p != v.end(); ++p) // iterator style
    ... // do something with *p
```

- Know both ways (iterator and subscript)
  - The subscript style is used in essentially every language
  - The iterator style is used in C (pointers only) and C++
  - The iterator style is used for standard library algorithms
  - The subscript style does not work for lists (in C++ and in most languages)
- Use either way for vectors
  - There are no fundamental advantages of one style over the other
  - But the iterator style works for all sequences
  - Prefer `size_type` over plain `int`
    - ▷ pedantic, but quiets compiler and prevents rare errors

Ways of traversing a vector


```
for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p) // iterator style
    ... // do something with *p

for(vector<T>::value_type x : v)                        // iterator style, range for [C++11]
    ... // do something with x

for(auto& x : v)                                        // iterator style, range for [C++11]
    ... // do something with x
```

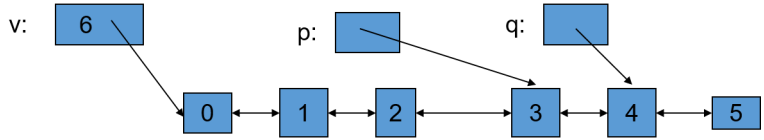
- Range for [C++11]
  - Use for the simplest loops
    - Every element from begin() to end()
  - Over one sequence
  - When we do not need to look at more than one element at a time
  - When we do not need to know the position of an element

list: Doubly Linked List

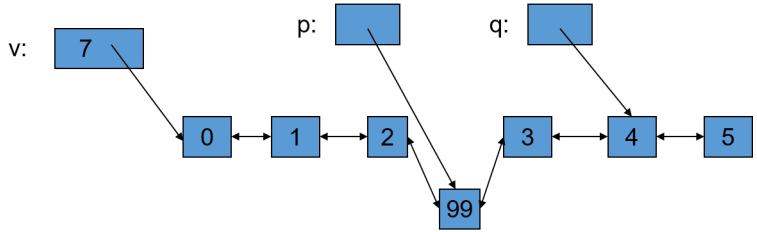

```
template<class T> struct Link {
    T value;
    Link* pre;
    Link* post;
}
template<class T> class list {
    Link* elements;
    // ...
    using value_type = T;
    using iterator = ???; // the type of an iterator is implementation defined
                          // and it (usefully) varies (for example, range checked iterators)
                          // a vector iterator could be a pointer to a link node
    using const_iterator = ???;

    iterator begin();             // points to first element
    const_iterator begin() const;

    iterator end();               // points to one beyond the last element
    const_iterator end() const;

    iterator insert(iterator p, const T& v); // insert a new element v before p

    iterator erase(iterator p);              // remove element pointed to by p
};
```

insert() into list


```
list<int>::iterator p = v.begin(); ++p; ++p; ++p;
list<int>::iterator q = p; ++q;
```

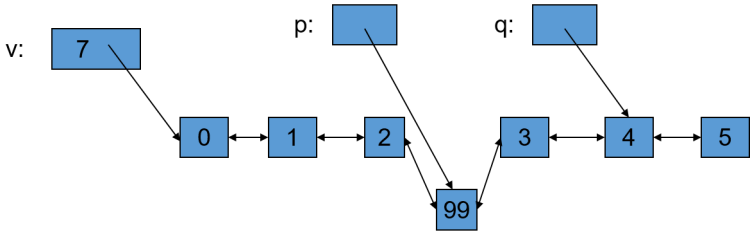



```
p = v.insert(p,99); // leaves p pointing at the inserted element
```

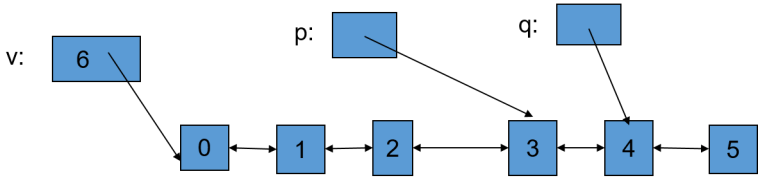



```
// Note: q is unaffected
// Note: No elements moved around
```

erase() from list




```
p = v.erase(p); // leaves p pointing at the element after the erased one
```




```
// Note: list elements do not move when we insert() or erase()
```

vector vs. list

- By default, use a `vector`
  - We need a reason not to
  - We can *grow* a vector (for example, using `push_back()`)
  - We can `insert()` and `erase()` in a `vector`
  - Vector elements are *compactly stored* and *contiguous*
  - For small vectors of small elements all operations are fast
    - ▷ compared to lists
- If we do not want elements to move, use a list
  - We can *grow* a list (for example, using `push_back()` and `push_front()`)
  - We can `insert()` and `erase()` in a `list`
  - List elements are *separately allocated*
- Note that there are more containers like
  - `map`
  - `unordered_map` [C++11]

map: An associative array

- For a vector, we subscript using an integer
- For a map, we can define the subscript to be (just about) any type
- After vector, map is the most useful standard library container
  - Maps (and/or hash tables) are the backbone of scripting languages
- A map is really an *ordered balanced binary tree*, by default ordered by < (less than)

```
// note the similarity to vector and list
template<class Key, class Value> class map {
    // ...
    using value_type = pair<Key, Value>; // a map deals in (Key, Value) pairs

    using iterator = ???; // Some implementation defined type - probably a pointer to a tree node
    using const_iterator = ???;

    iterator begin(); // points to first element
    iterator end();   // points to one beyond the last element

    Value& operator[](const Key&); // get Value for Key; creates pair if necessary, using Value()
    iterator find(const Key& k);   // is there an entry for k?

    pair<iterator, bool> insert(const value_type&); // insert new (Key, Value) pair
                                                    // the bool is false if insert failed
    void erase(iterator p);        // remove element pointed to by p
};
```

map: Example: Simple Use


```
#include <iostream>
#include <map>
using namespace std;

int main() { map<char,int> myMap; // Key = char, Value = int

    myMap['a'] = 10;                              // initializing a map. using index
    myMap.insert(pair<char, int>('c', 30));       // using insert. myMap['c'] = 30;
    map<char, int>::iterator it = myMap.begin();
    myMap.insert(it, pair<char, int>('b', 20));  // using insert with hint. myMap['b'] = 20
    myMap['d'] = 40;

    for(it = myMap.begin(); it! = myMap.end(); ++it) // print myMap
        cout << it->first << " => " << it->second << endl;

    it = myMap.find('c'); // search myMap for key = 'c'
    if (it != myMap.end())
        cout << "Value of myMap['c'] = " << it->second << endl;
}
a => 10
b => 20
c => 30
d => 40
Value of myMap['c'] = 30
```

map: Example: Counting Words


```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() { map<string, int> words; // keep (word, frequency) pairs. Key type = string, Value type = int
    for (string s; cin >> s;)
        ++words[s]; // words is indexed by string, words[s] returns int&, the int values are set to 0
    for (const auto& p: words)  // Iterating the map in [C++11] style
        cout << p.first << ": " << p.second << "\n";
}
```

Input: words

```
Twinkle, twinkle, little star
How I wonder what you are
Up above the world so high
Like a diamond in the sky
Twinkle, twinkle, little star
How I wonder what you are
```

```
Twinkle, twinkle, little star
How I wonder what you are
Up above the world so high
Like a diamond in the sky
Twinkle, twinkle, little star
How I wonder what you are
```

Output: : frequency

```
How: 4          above: 2        little: 4       twinkle,: 4
I: 4            are: 4          sky: 2          what: 4
Like: 2         diamond: 2      so: 2           wonder: 4
Twinkle,: 4     high: 2         star: 4         world: 2
Up: 2           in: 2           the: 4          you: 4
a: 2
```

set

- sets are containers that store *unique elements* following a specific order
- In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique
- Internally, the elements in a set are always sorted following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare)
- set containers are generally slower than unordered_set containers
- Sets are typically implemented as *binary search trees*

```
template<class T> class set {
    // ...
    using value_type = T;

    using iterator = ???; // Some implementation defined type - probably a pointer to a tree node
    using const_iterator = ???;

    iterator begin(); // points to first element
    iterator end();   // points to one beyond the last element

    pair<iterator, bool> insert(const value_type&); // insert new (Key, Value) pair
                                                    // the bool is false if insert failed
    iterator erase(const_iterator p);        // remove element pointed to by p
};
```

Module Summary

- Learnt Standard Template Library (STL) with common components
- Learnt useful containers and their use