CISC 3142
Programming Paradigms in C++

Ch4 – A Tour of C++:
Containers and Algorithms

(*Stroustrup – The C++ Programming Language, 4th Ed*)

Libraries

- No significant program is written in a bare programming language
- Any task can be rendered simple by the use of good libraries
- Standard library (S-L):
  - Runtime language support (allocation and run-time type info)
  - The C standard library (<string.h> becomes <cstring> to use the std namespace)
  - Strings and I/O streams
  - Containers (vector/map/etc) and algorithms (find(), sort()) - STL
  - Support for numerical computation (cmath, numeric)
  - Support for regular expression matching (regex)
  - Support for concurrent programming (threads and locks)
  - Utilities to support template metaprogramming
  - “Smart pointers” for resource management (unique_ptr and shared_ptr)
  - Special-purpose containers (array, bitset, tuple)

The S-L Headers and Namespace

- The standard-library is defined in namespace std, and provided through standard header

```
#include<string>
#include<list>
```

- It's generally not advisable to dump all names from a namespace into the global namespace. But from here on, the standard library is almost exclusively used, so std:: is not used.
- The next slide shows a list of selected S-L headers

Selected Standard Library Headers

<algorithm>	copy(), find(), sort()	§32.2	§iso.25
<array>	array	§34.2.1	§iso.23.3.2
<chrono>	duration, time_point	§35.2	§iso.20.11.2
<cmath>	sqrt(), pow()	§40.3	§iso.26.8
<complex>	complex, sqrt(), pow()	§40.4	§iso.26.8
<fstream>	fstream, ifstream, ofstream	§38.2.1	§iso.27.9.1
<future>	future, promise	§5.3.5	§iso.30.6
<iostream>	istream, ostream, cin, cout	§38.1	§iso.27.4
<map>	map, multimap	§31.4.3	§iso.23.4.4
<memory>	unique_ptr, shared_ptr, allocator	§5.2.1	§iso.20.6
<random>	default_random_engine, normal_distribution	§40.7	§iso.26.5
<regex>	regex, smatch	Ch-37	§iso.28.8
<string>	string, basic_string	Ch-36	§iso.21.3
<set>	set, multiset	§31.4.3	§iso.23.4.6
<sstream>	istrstream, ostrstream	§38.2.2	§iso.27.8
<thread>	thread	§5.3.1	§iso.30.3
<unordered_map>	unordered_map, unordered_multimap	§31.4.3.2	§iso.23.5.4
<utility>	move(), swap(), pair	§35.5	§iso.20.1
<vector>	vector	§31.4	§iso.23.3.6

Strings (std::string, or C++ strings)

- Use + for concatenation

```
string s = string{”Hello ”} + “World” + ’\n’;
// c-style strings can be implicitly converted to std:string, but not vice versa
// going the other way, use std::string.c_str(), which returns const char*, the c-style
// internal representation
```

- Mutable - supporting =, +=, and []

```
string name = "Niels Stroustrup";
string s = name.substr(6,10); // s = "Stroustrup"
name.replace(0,5,"nicholas"); // name becomes "nicholas Stroustrup"
name[0] = toupper(name[0]); // name becomes "Nicholas Stroustrup"
```

- Comparison

```
string answer = ”no”;
if (answer == ”yes”) { //…} // note in Java, one should use .equals()
```

Stream I/O

- ostream (cout) and istream (cin) for formatted I/O of built-in types
- std::getline(cin, string_variable), or cin.getline(char* s, size) for reading a whole line
- I/O for User-Defined Types

```
struct Entry {
    string name;
    int number;
};
// to write an Entry using a {“name”, number} format, overload the “<<” operator
// Note: If Entry is a class, this must be defined as a friend function to access its private data
// Here, Entry is a struct and its data members are public
ostream& operator<<(ostream& os, const Entry& e) {
        return os << "{\"" << e.name << "\", " << e.number << "}"; // \" is an escape character
}
```

Example Overloading operator>>

```
istream& operator>>(istream& is, Entry& e) { // read { "name" , number } pair. Note: formatted with { " " , and }
    char c, c2;
    if (is>>c && c=='{' && is>>c2 && c2=='"') {   // start with a { ", note whitespace in middle will be skipped
        string name;                             // the default value of a string is the empty string: ""
        while (is.get(c) && c!='"')              // anything before a " is part of the name, including spaces
            name+=c;                             // note: is>>c skips whitespace, but is.get(c) doesn't
        if (is>>c && c==',') {
            int number = 0;
            if (is>>number>>c && c=='}') {       // read the number and a }
                    e = {name ,number};          // assign to the Entry
                    return is;
            }
        }
    }
    is.setstate(ios_base::failbit);  // register the failure in the stream, setstate(): sets error flags
    return is;
}
```

Containers

- Classes with the main purpose of holding objects
- string could be considered a container of characters
- Providing suitable containers for a given task and supporting them with useful operations are important steps in any program
- Examples
  - vector
  - list
  - map
  - unordered_map

vector

- Elements are stored contiguously in memory
- Initialization (with default zero-initialization for elements)

```
vector<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

```
vector<int> v1 = {1, 2, 3, 4}; // size is 4
vector<string> v2; // size is 0
vector<Shape*> v3(23); //size is 23; initial element value: nullptr
vector<double> v4(32,9.9); // size is 32; initial element value: 9.9
```

- Access via subscripting [], or range-for loop (uses iterators)

```
void print_book(const vector<Entry>& book) {
        for (int i = 0; i!=book.size(); ++i)
                cout << book[i] << '\n';
}
```

```
void print_book(const vector<Entry>& book) {
    for (const auto& x : book) // for "auto" see §2.2.2
            cout << x << '\n';
}
```

More vector operations

- Adding a new element at the end

  for (Entry e; cin>>e;) // ends when **EOF** is entered, **^Z** on Windows, **^D** on Unix

  phone_book.push_back(e); // O(1) operation

- Copying via assignments and initializations

  vector<Entry> book2 = phone_book; // copy ctor or copy assignment?

- If copying is too expensive, consider references/pointers, or move operations

- vector<T> allows vectors of any types

Range Checking

- vector doesn't check range when [] is used (might be implementations dependent)

```
void silly(vector<Entr y>& book) {
        int i = book[book.size()].number; // book.size() is out of range, but no error is given
}
```

- However vector's .at(i) member function does the check (compare to Java ArrayList's .get(i))

  book.at[book.size()] will throw an exception: out_of_range

- A good practice to have a catch-all block in main()

```
int main()
try {
        // your code
}
catch (out_of_rang e) {
        cerr << "range error\n";
}
catch (...) { // "..." ellipsis is for variadic parameters
        cerr << "unknown exception thrown\n";
}
```

list

- Implemented as a doubly-linked list in standard library
- O(1) operation for inserting/deleting an element
- Initialization

```
list<Entry> phone_book = {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

- Typical use is to search the list for an element with a given value

```
int get_number(const string& s) {
        for (const auto& x : phone_book)
             if (x.name==s)
                    return x.number;
        return 0; // use 0 to represent "number not found"
}
```

list – iterator, insert/delete an item

- Use iterator to find an item (O(n) operation)

```
int get_number(const string& s) { // or list<Entry>::iterator get_item(const string& s)
        for (auto p = phone_book.begin(); p!=phone_book.end(); ++p)
                if (p->name==s) // an iterator can be used as if it's a pointer
                        return p->number; // or return p
        return 0; // use 0 to represent "number not found" , or return phone_book.end()
}
// .begin() returns the first, .end() returns the one-past-the-last element
```

- Insert/Erase (O(1) operation)

```
void f(const Entry& ee, list<Entry>::iterator p, list<Entry>::iterator q) {
        phone_book.insert(p,ee); // add ee before the element referred to by p
        phone_book.erase(q); // remove the element referred to by q
}
```

map

- Allows quick search on (key, value) pairs, also known as an associative array or a dictionary
- Implemented by a balanced binary tree - lookup cost is O(log(n))
- Initialization

```
map<string,int> phone_book {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

- When indexed by its key, a map returns the corresponding value

```
int get_number(const string& s) {
        return phone_book[s]; // if key=s is not found, it will be entered into the map with
                              // default value, 0. To avoid that, use find() and insert() instead of []
}
```

unordered_map

- Do better than O(log(n)) in lookup?
- unordered_map uses a hash table instead of an ordering function such as <
- Initialization (#include <unordered_map>)

```
unordered_map<string,int> phone_book {
        {"David Hume",123456},
        {"Karl Popper",234567},
        {"Bertrand Arthur William Russell",345678}
};
```

- Subscripting [] is supported

```
int get_number(const string& s) {
        return phone_book[s]; // will insert default value as well, if unmatched
}
```

- Default hash function is provided by the S-L for all basic types, strings, and some library types, and you can provide your own in the form of a hash function object.

Container Overview

- Standard Container Summary

vector<T>	A variable-size vector (§31.4)
list<T>	A doubly-linked list (§31.4.2)
forward_list<T>	A singly-linked list (§31.4.2)
deque<T>	A double-ended queue (§31.2)
set<T>	A set (§31.4.3)
multiset<T>	A set in which a value can occur many times (§31.4.3)
map<K,V>	An associative array (§31.4.3)
multimap<K,V>	A map in which a key can occur many times (§31.4.3)
unordered_map<K,V>	A map using a hashed lookup (§31.4.3.2)
unordered_multimap<K,V>	A multimap using a hashed lookup (§31.4.3.2)
unordered_set<T>	A set using a hashed lookup (§31.4.3.2)
unordered_multiset<T>	A multiset using a hashed lookup (§31.4.3.2)

Algorithms

- They are generally function templates for operating on containers

```
bool operator<(const Entry& x, const Entry& y) { // less than
        return x.name<y.name; // order Entrys by their names
}
void f(vector<Entry>& vec, list<Entry>& lst) {
        sort(vec.begin(),vec.end()); // use < for order
        unique_copy(vec.begin(),vec.end(),lst.begin()); // don't copy adjacent
                                                        equal elements
}
```

- The input sequence of elements is represented by a pair of iterators [b, e)
- For output, use back_inserter(Container& x) instead to always add new elements at the end
  - This avoids overwriting the target, as lst.begin() in the example above will do
  - Supported when the container has a push_back member function (e.g. vector, deque, list)

Use of Iterators

- Many algorithms return iterators:

```
bool has_c(const string& s, char c) {          // does s contain the character c? If not,
    return find(s.begin(),s.end(),c)!=s.end(); } // find() returns s.end()
```

- Another example

```
template<typename T>
using Iterator = typename T::iterator;
template<typename C, typename V>
vector<Iterator<C>> find_all(C& c, V v) {
// find all occurrences of v in c
    vector<Iterator<C>> res;
    for (auto p = c.begin(); p!=c.end(); ++p)
        if (*p==v)
            res.push_back(p);
    return res; // res holds all locations of matches
}
```

Usage

```
void test() {
    vector<string> vs { "red", "blue", "green", "green",
            "orange", "green" };
    for (auto p : find_all(vs,"green"))
        if (*p!="green")
            cerr << "vector bug!\n";
    for (auto p : find_all(vs,"green"))
        *p = "vert"; // change all “green” to “vert”
}
```

- An algorithm operates through iterators and knows nothing about the container
- A container supplies iterators on request and knows nothing about the algorithms

Iterator Types

- Iterators are objects and have many types
- The implementation is hidden from you
- They have common semantics
  - Applying ++ to any iterator yields an iterator that refers to the next element
  - * yields the element to which the iterator refers (similar to dereference)
- Each container knows its iterator types and make them available:
  - iterator: mutable when dereferenced (list<Entry>::iterator)
  - const_iterator: immutable when dereferenced (e.g. vector<Entry>::const_iterator)

Stream Iterators

- I/O streams also have iterators

```
ostream_iterator<string> oo {cout}; // write strings to cout, passed in the initializer list
int main() {
        *oo = "Hello, "; // meaning cout<<"Hello, "
        ++oo;
        *oo = "world!\n"; // meaning cout<<"world!\n"
}
```

- For input, we need a pair of iterators to represent a sequence [begin, end)

```
istream_iterator<string> ii {cin};
istream_iterator<string> eos {}; // the default istream_iterator marks the end() (sentinel)
```

- I/O stream iterators are typically not used directly, but passed to algorithms

Example of Stream Iterators

- Reads all strings from a file, saves to a new file while removing all duplicates

```
// need to include <fstream>, <set>, <iterator>
int main() {
        string from, to;    // for storing input, output filenames
        cin >> from >> to; // get source and target file names from keyboard input
        ifstream is {from}; // input stream for file "from"
        ofstream os {to}; // output stream for file "to "
        // read input – set is used to avoid duplicates – range ctor uses [begin, end) of ifstream
        set<string> b { istream_iterator<string>{is}, istream_iterator<string>{} };
        // copy to output – each string will be ended with a newline as delimiter
        copy( b.begin(), b.end(), ostream_iterator<string>{os,"\n"} );
        return !is.eof() || !os; // return error state (§2.2.1, §38.3)
        // Note: to return 0 (success), both is.eof() and os need to be true (i.e. read the entire input
        // file and there is no error writing to the ofstream)
}
```

Predicates

- Sometimes we need to make part of algorithm's action a parameter
  - e.g. we use find() to look for element that fulfills a specified requirement
  - e.g. the requirement could be "larger than 42"
  - Such requirement is a *predicate*
- Example

```
void f(map<string,int>& m) {
        auto p = find_if(m.begin(),m.end(),Greater_than{42}); // ... }
```

- Here Greater_than is a function object holding 42, the value to be compared to

```
struct Greater_than {
        int val;
        Greater_than(int v) : val{v} { }
        bool operator()(const pair<string,int>& r) { return r.second>val; }
};
// Note: a map's entry can be passed/accessed as a pair: (iterator->first, iterator->second)
```

Algorithm Overview

- Selected Standard Algorithms

p=find(b, e, x)	**p** is the first **p** in [**b**:**e**) so that ∗**p==x**
p=find_if(b, e, f)	**p** is the first **p** in [**b**:**e**) so that **f(**∗**p)==true**
n=count(b, e, x)	**n** is the number of elements ∗**q** in [**b**:**e**) so that ∗**q==x**
n=count_if(b, e, f)	**n** is the number of elements ∗**q** in [**b**:**e**) so that **f(**∗**q)==true**
replace(b, e, v, v2)	Replace elements ∗**q** in [**b**:**e**) so that ∗**q==v** by **v2**
replace_if(b, e, f, v2)	Replace elements ∗**q** in [**b**:**e**) so that **f(**∗**q)** by **v2**
p=copy(b, e, out)	Copy [**b**:**e**) to [**out**:**p**)
p=copy_if(b, e, out, f)	Copy elements ∗**q** from [**b**:**e**) so that **f(**∗**q)** to [**out**:**p**)
p=unique_copy(b ,e, out)	Copy [**b**:**e**) to [**out**:**p**); don't copy adjacent duplicates
sort(b, e)	Sort elements of [**b**:**e**) using **<** as the sorting criterion
sort(b, e, f)	Sort elements of [**b**:**e**) using **f** (a predicate) as the sorting criterion
(p1,p2)=equal_range(b, e, v)	[**p1**:**p2**) is the subsequence of the sorted sequence [**b**:**e**) with the value **v**; basically a binary search for **v**
p=merge(b, e, b2, e2, out)	Merge two sorted sequences [**b**:**e**) and [**b2**:**e2**) into [**out**:**p**)

Chapter-end Advice

[1] Don’t reinvent the wheel; use libraries; §4.1.

[2] When you have a choice, prefer the standard library over other libraries; §4.1.

[3] Do not think that the standard library is ideal for everything; §4.1.

[4] Remember to **#include** the headers for the facilities you use; §4.1.2.

[5] Remember that standard-library facilities are defined in namespace **std**; §4.1.2.

[6] Prefer **string**s over C-style strings (a **char***; §2.2.5); §4.2, §4.3.2.

[7] **iostream**s are type sensitive, type-safe, and extensible; §4.3.

[8] Prefer **vector<T>**, **map<K,T>**, and **unordered_map<K,T>** over **T[]**; §4.4.

[9] Know your standard containers and their tradeoffs; §4.4.

[10] Use **vector** as your default container; §4.4.1.

[11] Prefer compact data structures; §4.4.1.1.

[12] If in doubt, use a range-checked vector (such as **Vec**); §4.4.1.2.

[13] Use **push_back()** or **back_inserter()** to add elements to a container; §4.4.1, §4.5.

[14] Use **push_back()** on a **vector** rather than **realloc()** on an array; §4.5.

[15] Catch common exceptions in **main()**; §4.4.1.2.

[16] Know your standard algorithms and prefer them over handwritten loops; §4.5.5.

[17] If iterator use gets tedious, define container algorithms; §4.5.6.