CIS-255 Home http://www.c-jump.com/bcc/c255c/c255syllabus.htm

C++ Standard Library Part II

---

1. STL containers

STL containers replicate behavior of *data structures* commonly used in programming:

Sequence containers:

- dynamic string
- dynamic array: vector
- linked list
- queue
- stack

Associative containers:

- tree: set
- multiset
- associative array: map
- multimap

- heap: priority_queue
- bit storage: bitset

---

2. std::stack

A stack is a container that permits to insert and extract its elements only from the top of the container:

```
#include <cassert>
#include <stack>

using namespace std;

int main (int argc, char* argv[])
{
    stack< int > st;

    st.push( 100 );           // push number on the stack
    assert( st.size() == 1 ); // verify one element is on the stack
    assert( st.top() == 100 );// verify element value

    st.top() = 456;           // assign new value
    assert( st.top() == 456 );

    st.pop();                 // remove element
    assert( st.empty() == true );

    return 0;
}
```

---

3. std::set

A set is a container that holds *unique* elements.

The elements in **std::set** are always sorted.

```
#include <cassert>
#include <iostream>
#include <set>
using namespace std;
int main (int argc, char* argv[])
{
    set< int > iset;   // set of unique integer numbers

    iset.insert( 11 ); // populate set with some values
    iset.insert( -11 );
    iset.insert( 55 );
```

```
    iset.insert( 22 );
    iset.insert( 22 );
    if ( iset.find( 55 ) != iset.end() ) { // is value already stored?
        iset.insert( 55 );
    }
    assert( iset.size() == 4 ); // sanity check :-)
    set< int >::iterator it;
    for ( it = iset.begin(); it != iset.end(); it++ ) {
        cout << " " << *it;
    }
    return 0;
}
// Output:  -11 11 22 55
```

---

4. std::pair structure

`std::pair< T1, T2 >`

is a C++ structure that holds one object of type **T1** and another one of type **T2**:

```
#include <cassert>
#include <string>
#include <utility>
using namespace std;
int main (int argc, char* argv[])
{
    pair< string, string > strstr;
    strstr.first = "Hello";
    strstr.second = "World";

    pair< int, string > intstr;
    intstr.first = 1;
    intstr.second = "one";

    pair< string, int > strint( "two", 2 );
    assert( strint.first == "two" );
    assert( strint.second == 2 );

    return 0;
}
```

- A pair is much like a container that holds exactly two elements.
- The pair is defined in the standard header named **utility**.

5. std::pair data members

In mathematics, a *tuple* is a sequence of values, or tuple components, each component of a

specified type. A tuple containing *n* components is known as an *n-tuple*.

```
std::pair< typename T, typename U >
```

Thus, **std::pair** supports *duples*.

A **pair** has two public members, **first** and **second**.

```
template< typename T, typename U >
struct pair {
    typedef T first_type;
    typedef U second_type;
    T first;
    U second;

    pair(); // default constructor

    // construct from specified values:
    pair( T const& x, U const& y );

    // construct from compatible pair:
    template< typename V, typename W > pair( pair<V, W> const& pr );
};
```

---

6. std::pair construction

```
std::pair< typename T, typename U >
```

Default construction

Construction from two items

Construction from another pair (even with other types)

The function **make_pair(item1, item2)** makes a pair.

```
template< typename T, typename U >
struct pair {

    pair();

    pair( T const& x, U const& y);

    template< typename V, typename W > pair( pair<V, W> const& pr);
};
```

---

7. Functions returning a pair

Functions that need to return two values often return a **pair**:

```
pair< bool, double > result = do_a_calculation();

if ( result.first ) {
    do_something_more( result.second );

} else {
    report_error();
}
```

8. std::map

- Supports association
- A map stores pairs of a *key type* and a *value type*
- Provides fast access to a value when given a key.
- Uses trees, so fast means `O(log(`*`num items in map`*`))`
- Must be able to compare keys using operator<
- Map supports iteration in order of keys,

  because map items are always *sorted* by its keys.

9. std::map construction

- Declarations look like this:

```
map< string, int > mymap;
```

- Other constructors for copying, or construction from a range of pairs (e.g. a vector).

```
vector< pair < string, int > > myvect;
//...
map< string, int > mymap2( myvect.begin(), myvect.end() );
```

10. Adding items to a map

- Use operator[ ] to access items

```
map< string, int > agemap;
string name = "fred";
agemap[ name ] = 45;      // "fred" --> 45
int age = agemap[ name ];
++agemap[ name ];         // now "fred" --> 46
```

- Note: use of operator[ ] will put items in if they aren't there!
- Generally, this is very useful, occasionally a pain.

```
map< string, int > visit_count;
string name = "fred";
++visit_count[ name ]; // Works fine!!
```

11. Looking at a map

- Use **count(key)** to see if key is in the map
- For a map, **count( )** will always return **0** or **1**

```
map< string, int > agemap;
string name = "fred";

int age = agemap[ name ]; // Always succeeds, might return 0

if ( agemap.count(name) == 0 ) {
    // name is not in map
}
```

12. map iterators

Use iterators to specify items or ranges:

```
typedef map< string, int > MapT;
typedef MapT::const_iterator MapIterT;

MapT amap;
// Print out map contents in alphabetical order:
for ( MapIterT mit = amap.begin(); mit != amap.end(); ++mit ) {
```

```
        cout
            << mit->first
            << " "
            << mi->second
            << endl
            ;
    }
```

---

13. map::find

Use **find(*key*)** to get an iterator for a specific key:

```
typedef map<string, int> MapT;
typedef MapT::const_iterator MapIterT;

MapT amap;
MapIterT result = amap.find( "Fred" );

if ( result != amap.end() ) {
    // Print out map in alpha. order, starting with Fred
    for( MapIterT mit = result; mit != amap.end(); mit++ ) {
        cout
            << mit->first
            << " "
            << mit->second
            << endl
            ;
    }
}
```

---

14. map::insert

Use **insert( )** to put in a new item *only* if it isn't there:

```
#include <cassert>
#include <string>
#include <map>

using namespace std;
typedef map<string, int> MapT;
typedef MapT::const_iterator MapIterT;

int main()
{
    MapT amap;
    pair< MapIterT, bool> result =
        amap.insert( make_pair( "Fred", 45 ) );

    assert( result.second == true );
```

```
    assert( result.first->second == 45 );

    result = amap.insert( make_pair( "Fred", 54 ) );

    // Fred was already in the map, and result.first
    // simply points there now:
    assert( result.second == false );
    assert( result.first->second == 45 );
}
```

---

15. map::erase

Use **erase(*key*)** to remove an item:

```
typedef map< string, int > MapT;
MapT amap;
//...

int how_many_erased = amap.erase( "Fred" );

if ( how_many_erased == 1 ) {
    // Fred was in the map, now he's not.
}
```

---

16. Map summary

- A map is a container that holds unique pairs of *keys* and *values*.
- The elements in **std::map** are always sorted by its keys.
- Each element of the map is formed by the combination of the *key value* and a *mapped value*.
- Map iterators access both the key and the mapped value at the same time.

```
#include <cassert>
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main (int argc, char* argv[])
{
    map< string, string > phone_book;
    phone_book[ "411" ] = "Directory";
    phone_book[ "911" ] = "Emergency";
    phone_book[ "508-678-2811" ] = "BCC";
    if ( phone_book.find( "411" ) != phone_book.end() ) {
        phone_book.insert(
            make_pair(
                string( "411" ),
```

```
                string( "Directory" )
            )
        );
    }
    assert( phone_book.size() == 3 );
    map< string, string >::const_iterator it;
    for ( it = phone_book.begin(); it != phone_book.end(); ++it ) {
        cout
            << " " << it->first
            << " " << it->second
            << endl
            ;
    }
    return 0;
}
/* Output:
 411 Directory
 508-678-2811 BCC
 911 Emergency
*/
```

---

17. std::multimap

- Similar to map, but allows multiple values for one key
- Doesn't provide operator[ ]
- **insert( )** returns an iterator, since it can't fail
- Still supports iteration in order of keys,

  but no order is assumed on the values.
- Now **count(*key*)** can return any size, corresponding to the number of elements with the given *key*.
- multimap has **find(*key*)**, but this is not as useful as **equal_range(*key*)**

---

18. std::multimap::equal_range

---

equal_range(*key*) returns a range that includes all elements for a given key:

```
typedef multimap< string, int > MMapT;
typedef MMapT::const_iterator MMIterT;

MMapT amap;
pair< MMIterT, MMIterT > result = amap.equal_range( "Fred" );

// Print out all values for the key named "Fred"
```

```
    for( MMIterT mit = result.first; mit != result.second; mit++ ) {
        cout
            << mit->first
            << " "
            << mit->second
            << endl
            ;
    }
```

---

19. STL Algorithms

- Containers are nice, but we want more!
- We want to *find, remove, sort,* etc.
- We also want to use these functions on any container.
- STL provides all of the above with a help of iterators.

---

20. Iterators revisited

- Iterators are distinguished by the access type they provide:
  - Output
  - Input
  - Forward
  - Bidirectional
  - Random
- A regular C++ pointer to an array is a *random access iterator!*

---

21. Iterator types and functionality

If p is an iterator, the following semantics are supported,

	Iterator Type

where

Access Type	Output	Input	Forward	Bidirectional	Random
Write	`*p=x`		`*p=x`	`*p=x`	`*p=x`
Read		`x=*p`	`x=*p`	`x=*p`	`x=*p`
Pointer		`p->f`	`p->f`	`p->f`	`p->f`

Iteration	`++p`	`++p`	`++p`	`++p`
				`--p`
					`++p p+n`
					`p-n p+=n`
					`p-=n`
					`--p`

Comparison		`p==q`
		`p!=q`
			`p==q`
			`p!=q`
				`p==q`
				`p!=q`
					`p==q p>q`
					`p<q p>=q`
					`p<=q`
					`p!=q`

- *x* is an item pointed by the iterator,
- *q* is another iterator,
- *f* is a data field in a struct,
- *n* is an integer number.

---

22. std::sort

- Sort the range between two iterators
- Iterators must be random access
- Items pointed to must have operator<

```
template< typename RandomIterT >
void sort( RandomIterT first, RandomIterT last );

template< typename RandomIterT, typename PredicateT >
void sort( RandomIterT first, RandomIterT last, PredicateT pr );
```

---

23. std::sort example

```
#include< algorithm>
#include< vector>

using namespace std;

int arr[ 100 ];
sort( arr, arr + 100 );
```

```
vector v1;
sort( v1.begin(), v1.end() );
```

---

24. Predicates

- A function returning a bool is a predicate.
- An object which overloads operator( ) to return bool is also a predicate.
- Some algorithms take predicates and do useful things with them.

---

25. std::count_if and predicate function

```
#include <algorithm>

bool less_than_7( int value )
{
    return value < 7;
}

vector< int > v1;
int count_less = std::count_if( v1.begin(), v1.end(), less_than_7 );
```

---

26. std::count_if and predicate object

```
class less_than {
public:
    less_than( int t )
    : m_thresh( t )
    {
    }

    bool operator( )( int v )
    {
        return v < m_thresh;
    }

private:
    int m_thresh;
};//class less_than

vector< int > v1;
int x = 14;
```

```
int count_less = std::count_if( v1.begin(), v1.end(), less_than( x ) );
```
