Standard Template Library Basic

한양대학교 컴퓨터소프트웨어학부

2017년 2학기

C++ Standard Template Library

- The Standard Template Library defines powerful, template-based, reusable components
  - That implements common data structures and algorithms
- STL extensively uses *generic programming* based on *templates*
- Divided into three components:
  - Containers: data structures that store objects of any type
  - Iterators: used to manipulate container elements
  - Algorithms: searching, sorting and many others

Generic Programming

- Generalize algorithms
  - Sometimes called “lifting an algorithm”
- The aim (for the end user) is:
  - Increase correctness
    - Thought better specification
  - Greater range of uses
    - Possibilities for re-use
  - Better performance
    - Through wider use of tuned libraries
    - Unnecessarily slow code will eventually thrown away

Lifting example (concrete algorithms)

```
double sum(double array[], int n)      // one concrete algorithm (doubles in array)
{
    double s = 0;
    for (int i = 0; i < n; ++i ) s = s + array[i];
    return s;
}

struct Node { Node* next; int data; };

int sum(Node* first)                   // another concrete algorithm (ints in list)
{
    int s = 0;
    while (first) {                    // terminates when expression is false or zero
        s += first->data;
        first = first->next;
    }
    return s;
}
```

Lifting example (abstract the data structure)

```
// pseudo-code for a more general version of both algorithms

    int sum(data)          // somehow parameterize with the data structure
    {
        int s = 0;                      // initialize
        while (not at end) {            // loop through all elements
            s = s + get value;          // compute sum
            get next data element;
        }
        return s;                       // return result
    }
```

- We need three operations (on the data structure):
  - not at end
  - get value
  - get next data element

Lifting example (STL version)

```
// Concrete STL-style code for a more general version of both algorithms
template<class Iter, class T>       // Iter should be an Input_iterator
                                    // T should be something we can + and =
T sum(Iter first, Iter last, T s)   // T is the "accumulator type"
{
    while (first!=last) {
        s = s + *first;
        ++first;
    }
    return s;
}
```

- Let the user initialize the accumulator

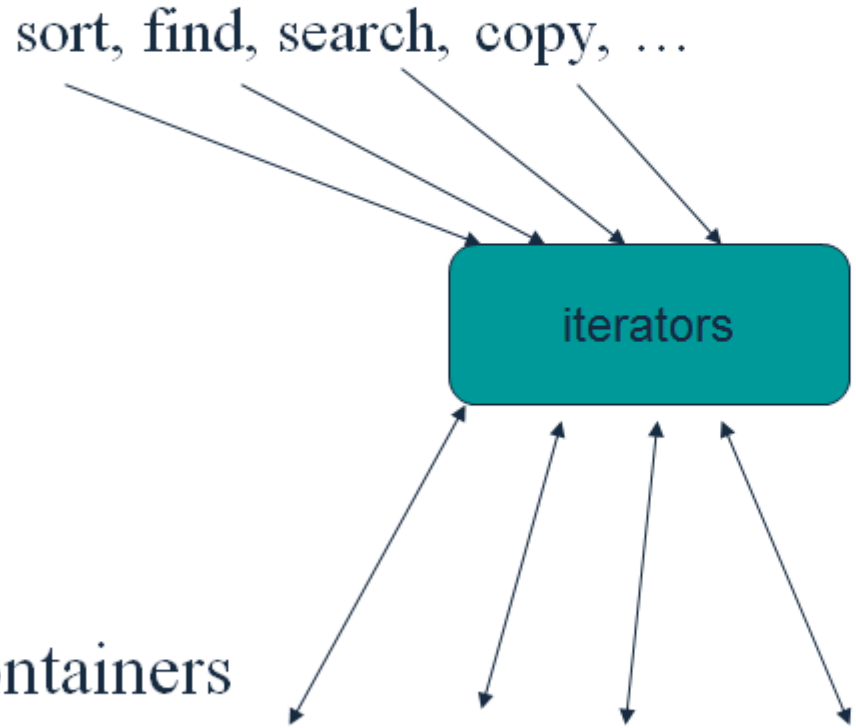
```
float a[] = { 1,2,3,4,5,6,7,8};
double d = 0;
d = sum(a,a+sizeof(a)/sizeof(*a),d);
```

Lifting example

- Almost the standard library accumulate
  - I simplified a bit for terseness
- Works for
  - arrays
  - **vectors**
  - **lists**
  - **istreams**
  - …
- Runs as fast as "hand-crafted" code
  - Given decent inlining
- The code's requirements on its data has become explicit
  - We understand the code better

Basic model

- Algorithms

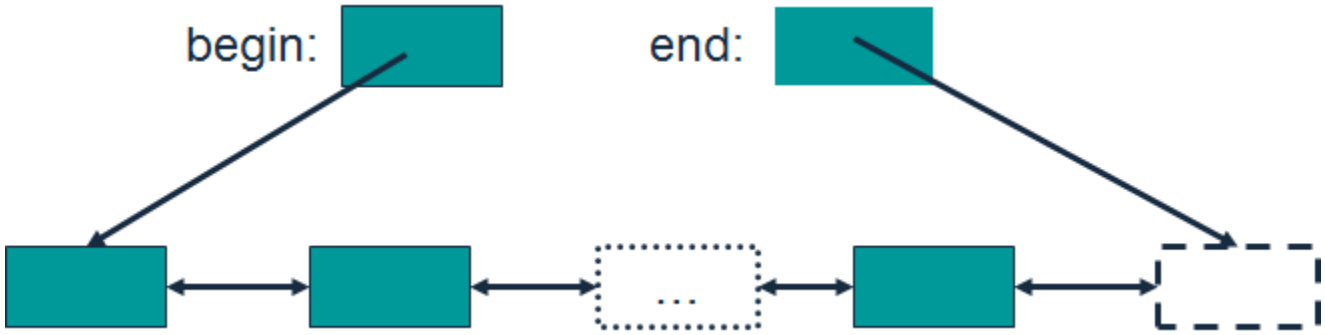


- Containers

- Separation of concerns
  - Algorithms manipulate data, but don't know about containers
  - Containers store data, but don't know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types

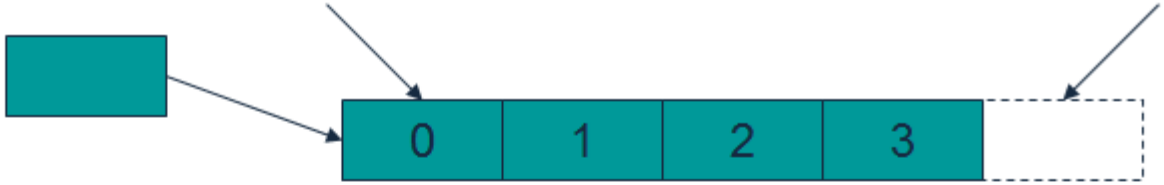
Basic Model (cont.)

- 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)



Containers
(holds sequences in different ways)

- **vector**

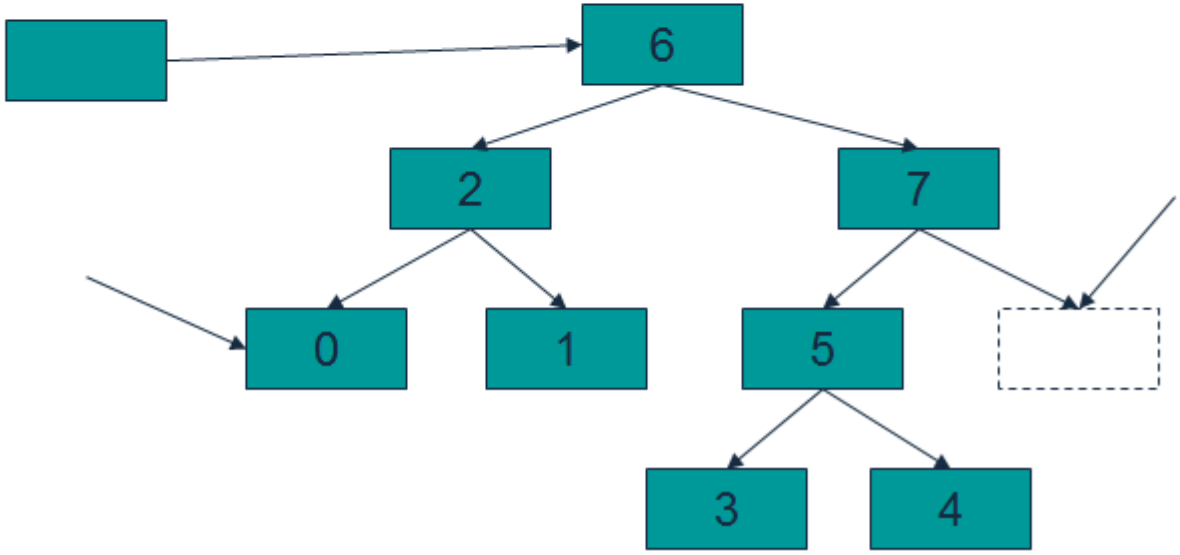


- **list**

(doubly linked)

- **set**

(a kind of tree)



Containers (cont.)

- Three types of containers
  - Sequence containers:
    - linear data structures such as vectors and linked lists
  - Associative containers:
    - non-linear containers such as hash tables
  - Container adapters:
    - constrained sequence containers such as stacks and queues
- Sequence and associative containers are also called first-class containers

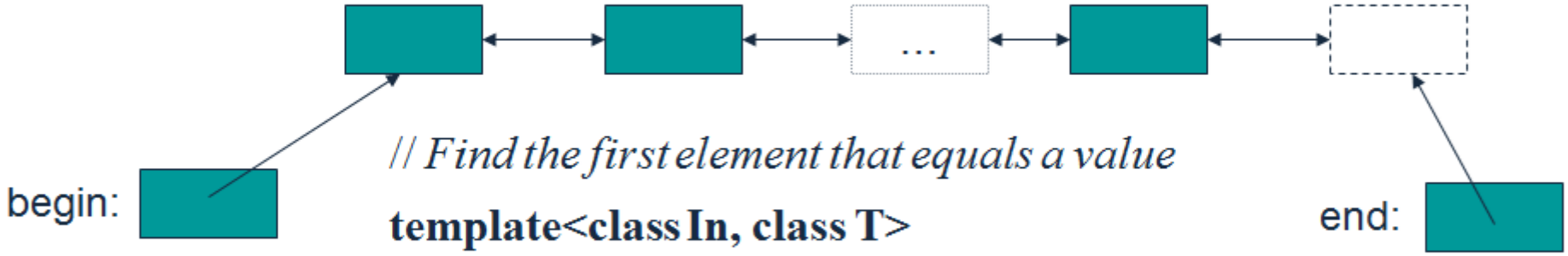
Iterators

- **Iterators are pointers to elements of first-class containers**
  - Type **const_iterator** defines an iterator to a container element that *cannot* be modified
  - Type **iterator** defines an iterator to a container element that *can* be modified
- **All first-class containers provide the members functions** begin() **and** end()
  - return iterators pointing to the first and one-past-the-last element of the container

Iterators (cont.)

- If the iterator **it** points to a particular element, then
  - -**it++ (or ++it)** points to the next element and
  - ***it** refers to the value of the element pointed to by **it**
- The iterator resulting from end() can only be used to detect whether the iterator has reached the end of the container
- We will see how to use begin() and end() in the next slides

The simplest algorithm: **find()**




```
// 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)        // find an int in a vector
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found x */ }
    // …
}
```

We can ignore ("abstract away") the differences between containers

find()

generic for both element type and container type

```
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 */ }
    // ...
}
```

How to Use C++ template

STL is a template library. Templates are used for generic programming in C++.

- You can specify a type (= a class) in < >.
- Both functions or classes can be templated.
- We will learn how to make templated functions or classes later. Now just need to understand how to use them.

How to Use C++ template

STL is a template library. Templates are used for generic programming in C++.

- You can specify a type (= a class) in < >.
- Both functions or classes can be templated.

```
template <class T>
const T& max(const T& a, const T& b) {
  return (a < b) ? b : a;
}

int main() {
  int a = 10;
  int ma = max(a, 0);       // Same as max<int>(a, 0) .
  double b = 0.9, mb;
  mb = max(b, 1);           // Compile error: (double, int).
  mb = max(b, 1.0);         // Okay: both are double.
  mb = max<double>(b, 1);   // Okay.
  return 0;
}
```

How to Use C++ template

- You can specify a type (= a class) in < >.

```
template <class T>
struct Complex {
  T real, imag;
  Complex(const T& r, const T& i) : real(r), imag(i) {}
  Complex(const Complex& c) : real(c.real), image(c.imag) {}
  void Print() const;
  Complex Multiply(const Complex& c) const;
}

int main() {
  Complex<int> ci(0, 1);
  Complex<double> cr(1.0, 2.0);
  Complex<double> cr2 = cr.Multiply(Complex<double>(2.0, 0.0));
  return 0;
}
```

C++ :: and namespace

:: is used to specify the namespace or the class membership.

- A::B means B is in a namespace/class A.
- ::B means B belongs the global namespace (most C library).

```
#include <math.h>
namespace my_namespace {

class MyClass {
  void FunctionA(int i);
  // ...
};

void MyClass::FunctionA(int i) { /* ... */ }
void FunctionB(double v, MyClass* a) { /* ... */ }

}  // namespace my_namespace

int main() {
  my_namespace::MyClass a;
  my_namespace::FunctionB(1.25, &a);
  double v = ::cos(0.0);
  return 0;
}
```

`vector` - a resizable array

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

vector<int> va;                      // Make an empty array: va = []
                                  // push_back(v)
va.push_back(10);                    // va = [10]
va.push_back(20);                    // va = [10, 20]
assert(va.size() == 2);           // size()
assert(va.empty() == false);      // empty()
assert(va.front() == 10);         // front()
assert(va.back() == 20);          // back()
va.pop_back();                    // pop_back()
va.clear();                       // clear()

vector<double> vb(10, 0.0);    // vb = [0.0, 0.0, ..., 0.0]
vb.resize(20);                    // resize(sz)
for (int i = 0; i < vb.size(); ++i) vb[i] = i * 0.5;  // operator[](i)

vector<double> vc;
vc.reserve(vb.size());            // reserve(sz)
for (int i = 0; i < vb.size(); ++i) vc.push_back(vb[i] * 2);
```

`vector` - a resizable array

- Iterator : access the elements in the container iteratively in order.
  - Const and non-const types : `const_iterator` and `iterator`.
  - In many cases, it can considered as a pointer to an element.

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

int main(void) {
// vector(sz)
vector<int> v(10);
for (int i = 0; i < v.size(); ++i) v[i] = i;

// begin(), end()
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
  cout << " " << *it;
}
// Output:  0 1 2 3 4 5 6 7 8 9

// rbegin(), rend()
for (vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it) {
  cout << " " << *it;
}
// Output:  9 8 7 6 5 4 3 2 1 0
}
```

`vector` - a resizable array

- You can make a vector of strings or other classes.

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

struct Complex { double real, imag; /* ... */ };

// ...
vector<string> vs;
for (int i = 0; i < 10; ++i) cin >> vs[i];
// vector(sz, init)
vector<string> vs2(5, "hello world");

vector<Complex> v1(10);
vector<Complex> v2(10, Complex(1.0, 0.0));
Complex c(0.0, 0.0);
v2.push_back(c);
for (int i = 0; i < v2.size(); ++i) {
  cout << v2[i].real << "+" << v2[i].imag << "i" << endl;
}
```

vector - a resizable array

- Even a vector of vectors of a class is possible.

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

vector<vector<int> > vi(10);  // Note vector<vector<int>> => Error.
for (int i = 0; i < vi.size(); ++i) vi[i].resize(5, 0);

for (int i = 0; i < vi.size(); ++i) {
  for (int j = 0; j < vi[i].size(); ++j) cout << " " << vi[i][j];
  cout << endl;
}

// vector<vector<int> > vi(10, vector<int>(5, 0));  would this work?
```

`vector` - a resizable array

- Sometimes you may want to use a vector of pointers.

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

class Student;

vector<Student*> vp(10, NULL);
for (int i = 0; i < vp.size(); ++i) {
  vp[i] = new Student;
}

// After using vp, all elements need to be deleted.

for (int i = 0; i < vp.size(); ++i) delete vp[i];
vp.clear();
```

Other Vector-like Containers

- List, stack, queue, and deque (double-ended queue).

	vector	**list**	**stack**	**queue**	**deque**
Random access	operator[] at()	-	-	-	operator[] at()
Sequential access	front() back()	front() back()	top()	front() back()	front() back()
Iterators	begin(), end() rbegin(), rend()	begin(), end() rbegin(), rend()	-	-	begin(), end() rbegin(), rend()
Adding elements	push_back() insert()	push_front() push_back() insert()	push()	push()	push_front() push_back() insert()
Deleting elements	pop_back() erase() clear()	pop_front() pop_back() erase() clear()	pop()	pop()	pop_front() pop_back() erase() clear()
Adjusting size	resize() reserve()	resize()	-	-	resize()

`set` - a container for keys

- Contains a set of keys, and accessing with keys is very efficient.

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

set<int> s;
for (int i = 0; i < 10; ++i) s.insert(i * 10);

for (set<int>::const_iterator it = s.begin(); it != s.end(); ++it) {
  cout << " " << *it;
}
assert(s.size() == 10);
assert(s.empty() == false);

set<int>::iterator it, it_low, it_up;
assert(s.find(123) == s.end());  // s: 0 10 20 30 40 50 60 70 80 90
it = s.find(50);                 //                    ^it
it_low = s.lower_bound(30);      //             ^it
it_up = s.upper_bound(60);       //                          ^it
s.erase(it_low, it_up);          // s: 0 10 20 70 80 90
s.clear();                       // s:
```

map - a container for key-value pairs

- Contains a set of key-value pairs that can be accessed by keys.

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

map<int, double> m;
for (int i = 0; i < 4; ++i) m.insert(make_pair(i, 0.5 * i));

for (map<int, double>::iterator it = m.begin(); it != m.end(); ++it) {
  cout << " " << it->first << "," << it->second;
}
m.insert(make_pair(0, 10.0));  // Since (0,0) already exists, no change.
// operator[](key) :
// (*((this->insert(make_pair(x,mapped_type()))).first)).second
m[10] = 3.141592;       // This adds a new key-value pair (10, 3.141592).
m[0] = 10.0;            // This updates the value for key 0.
cout << m[3] << endl;   // This outputs 1.5 which is the value for key 3.
cout << m[11] << endl;  // Note this adds (11,0.0) and prints 0.0.

map<int, double>::iterator it;
assert(m.find(123) == m.end());  // m: (0,0.0) (1,0.5) (2,1.0) (3,1.5)
it = m.find(2);                  //                      ^it
cout << " " << it->first << "," << it->second;  // Prints 2,1.0

m.clear();
```

`pair` - a pair of values

- A tuple that contains two values; first and second.

```
#include <string>
#include <utility>

using namespace std;

pair<string, int> p;
p.first = "hello";
p.second = 10;

pair<int, int> p1(10, 20);
pair<int, string> p2 = make_pair(5, "hi");  // make_pair(first, second)
```

Other associative containers

- Multiset and multimap allows duplicate keys.

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

set<int> s;
multiset<int> ms;
map<int, int> m;
multimap<int, int> mm;

for (int i = 0; i < 10; ++i) {
  int key = i / 2;
  pair<int, int> pk(key, i);
  s.insert(key), ms.insert(key), m.insert(pk), mm.insert(pk);
}
assert(s.size() == 5);    // 0, 1, 2, 3, 4
assert(ms.size() == 10);  // 0, 0, 1, 1, 2, 2, 3, 3, 4, 4,
assert(m.size() == 5);    // (0,0), (1,2), (2,4), (3,6), (4,8)
assert(mm.size() == 10);  // (0,0), (0,1), (1,2), (1,3), (2,4), (2,5),
                          // (3,6), (3,7), (4,8), (4,9)
```

Algorithm library

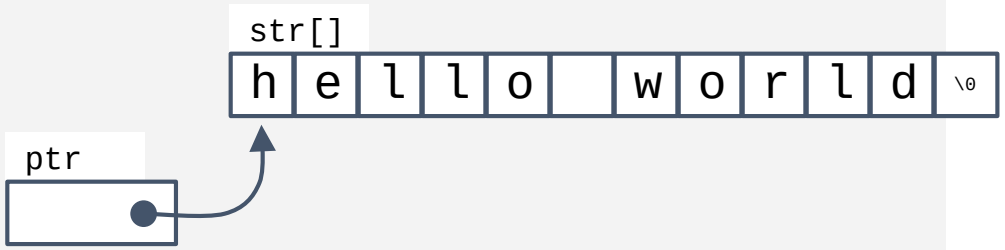
- Many useful algorithms are available.

```
#include <algorithm>
#include <ctime>    // For time() function.
#include <cstdlib>  // For rand() and srand() function.
using namespace std;

// int RandomNumber() { return (rand()%100); }

// .....
const int a = 10, b = 15;
int minv = std::min(a, b), maxv = std::max(a, b);  // min(a, b), max(a, b)

vector<int> v(10);
for (int i = 0; i < v.size(); ++i) v[i] = 2 * i;

vector<int>::iterator it;
it = std::min_element(v.begin(), v.end());  // min_element(b, e)

srand((unsigned int) time(NULL));
std::random_shuffle(v.begin(), v.end());  // random_shuffle(b, e)
std::sort(v.begin(), v.end());            // sort(b, e)
```

C/C++ character string

- A string is basically an array of characters (char []).
- C standard requires a string must be terminated with 0 ('\0').

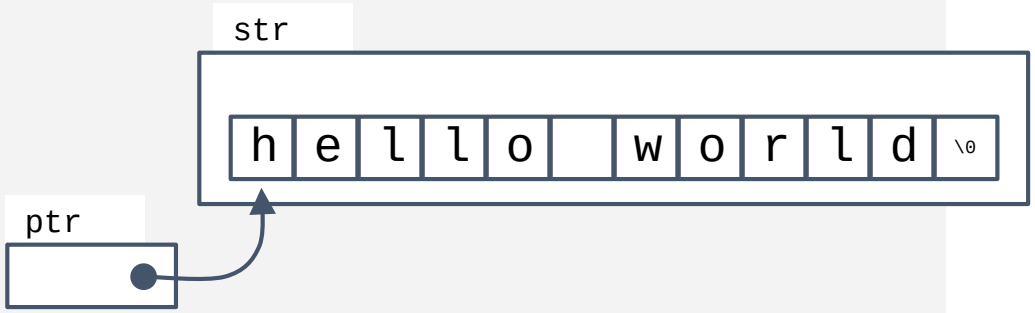
```
int main() {
  char str[] = "hello world";

  char* ptr = str;
  while (*ptr != '\0') {
    printf("%c", *ptr++);
  }
  return 0;
}
```



C++ std::string

- In C++, STL provides a powerful string class.

```
#include <string>

std::string str = "hello world";
const char* ptr = str.c_str();
printf("%s\n", ptr);

// ...


std::string str1 = str + " - bye world";
assert(str1 == "hello world - bye world");

assert(str.length() > 10);
assert(str[0] == 'h');
str[0] = 'j';
str.resize(5);
assert(str == "jello");

// check out http://www.cplusplus.com/reference/string/string/
// resize(), substr(), find(), etc.
```



C++ std::string - find

```
size_t find(const string& str, size_t pos = 0) const;
size_t find(char c, size_t pos = 0) const;
[from http://www.cplusplus.com/]
```

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

int main() {
  string str("There are two needles in this haystack with needles.");
  string str2("needle");
  size_t found;

  if ((found = str.find(str2)) != string::npos) {
    cout << "first 'needle' found at: " << int(found) << endl;
  }
  str.replace(str.find(str2), str2.length(), "preposition");
  cout << str << endl;
  return 0;
}
```

```
first 'needle' found at: 14
There are two prepositions in this haystack with needles.
```

C++ std::string - substr

```
string substr(size_t pos = 0, size_t n = npos) const;
[from http://www.cplusplus.com/]
```

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

int main() {
  string str = "We think in generalities, but we live in details.";
               // quoting Alfred N. Whitehead

  string str2 = str.substr(12, 12);  // "generalities"
  size_t pos = str.find("live");     // position of "live" in str
  string str3 = str.substr(pos);     // get from "live" to the end

  cout << str2 << ' ' << str3 << endl;
}
```

```
generalities live in details.
```

C++ Stream IO

- In C++, iostream provides basic input/output streaming.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  string str;
  int i;
  double d;

  cin >> str;  // takes key-inpunt until enter is pressed.
  cin >> i >> d;

  cout << "i = " << i << ", d = " << d << ", str=" << str << endl;
  return 0;
}
```

Summary

- C++ namespace
- Using C++ template classes and functions
- C++ standard template library (STL)
  - Containers : vector, set, map, multiset, multi-map
  - Iterators : iterator, const_iterator
  - String : string - find, substr, etc.
  - Algorithm : min, max, sort, random_shuffle, etc.
  - Stream IO : cin, cout, cerr, <<, >>


Q&A