Creative Software Design

Standard Template Library

Yunho Kim
yunhokim@hanyang.ac.kr
Dept. of Computer Science

Today's Topics

- Intro to Template (briefly)
- STL (Standard Template Library)
- Containters
  - std::vector, std::list
  - std::stack, std::queue
  - std::set, std::map
- Iterator
- std::string

Template

- Functions and classes can be "templated".
- This allows a function or class to **work on many different data types without being rewritten** for each one.

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

class CintPoint{
private:
    int x, y;
public:
    CintPoint(int a, int b){ x = a;  y = b;}
    void move(int a, int b){ x +=a;  y += b;}
    void print(){ cout << x << " " << y << endl;}
};

class CdoublePoint{
private:
    double x, y;
public:
    CdoublePoint(double a, double b){ x = a; y = b;}
    void move(double a, double b){ x +=a;  y += b;}
    void print(){ cout << x << " " << y << endl;}
};

int main(){

    CintPoint P1(1,2);
    CdoublePoint P2(1.1, 2.1);
    P1.print();
    P2.print();
}
```

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

template <typename T>
class Point{
private:
    T x, y;
public:
    Point(T a, T b){ x = a;  y = b;}
    void move(T a, T b){ x +=a;  y += b;}
    void print(){ cout << x << " " << y << endl;}
};

int main(){

    Point<int> P1(1,2);
    Point<double> P2(1.1, 2.1);
    P1.print();
    P2.print();
}
```

An example of class template

Standard Template Library (STL)

- STL defines powerful, template-based, reusable components.
- A collection of useful template for handling various kinds of data structure and algorithms
  - Containers: data structures that store objects of any type
  - Iterators: used to manipulate container elements
  - Algorithms: operations on containers for searching, sorting and many others

Containers

- Sequential container, Container adaptor, Associative container

- Sequential container
  - Elements are accessed by their "position" in the sequence.
  - **vector**: fast insertion at end, random access
  - **list**: fast insertion anywhere, sequential access
  - **deque** (double-ended queue): fast insertion at either end, random access

- Container adapter
  - “Adapting” the interface of underlying container to provide the desired behavior.
  - **stack**: Last In First Out
  - **queue**: First In First Out

Containers

- Associative container
  - Elements are referenced by their **key** and not by their absolute position in the container, and always sorted by keys.
  - **map**: a mapping from one type (key) to another type (value)
  - **set**: add or delete elements, query for membership…

- There are a few more containers in STL, but this course covers only the most popular ones.

std::vector - a resizable array

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

int main(void){

    vector<int> intVec(10);

    for(int i=0; i< 10; i++){
            cout << "input!";
            cin >> intVec[i];
    }

     for(int i=0; i< 10; i++){
            cout << intVec[i] << " " ;
    }
    cout << endl;
    return 0;
}
```

std::vector - a resizable array

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

int main(void){

    vector<int> intVec;
    int temp;

    for(int i=0; i< 3; i++){
        cout << "input!";
        cin >> temp;
        intVec.push_back(temp);
    }
    for(int i=0; i< (int)intVec.size(); i++){
        cout << intVec[i] << " " ;
    }
    cout << endl;
    cout << "size" << intVec.size() << endl;
    intVec.resize(intVec.size()+3);
    cout << "size" << intVec.size() << endl;
    for(int i=(int)intVec.size()-3; i< (int)intVec.size(); i++){
        intVec[i] = i;
    }
    for(int i=0; i< (int)intVec.size(); i++){
        cout << intVec[i] << " " ;
    }
    cout << endl;

    return 0;
}
```

std::vector - a resizable array

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

int main(void){

    vector<int>  intVec;
    intVec.push_back(10);
    intVec.push_back(20);
    if (intVec.empty() == true){
        cout << "size of Vector is " << intVec.size();
    }
    cout << intVec.front() << endl;
    cout << intVec.back() << endl;
    intVec.pop_back();
    cout << intVec.back() << endl;
    intVec.clear();

}
```

std::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(size, initial_value)
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;
}
```

std::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();
```

std::vector

- **From now on, use std::vector instead of array.**
  - Element are stored in contiguous storage, like an array.
  - Random access (by index): Fast access to any element
  - Fast addition/removal of elements at the **end** of the sequence.
  - Much more flexible and powerful than array.

- https://www.stroustrup.com/bs_faq2.html#arrays

References for STL

- std::vector
  - https://en.cppreference.com/w/cpp/container/vector
  - https://docs.microsoft.com/ko-kr/cpp/standard-library/vector-class?view=msvc-160
- STL containers
  - https://en.cppreference.com/w/cpp/container
  - https://docs.microsoft.com/ko-kr/cpp/standard-library/stl-containers?view=msvc-160
- You can find documents for any other STL features in the links in the above pages.

Iterator

- Iterator: a pointer-like object **pointing to** some element in a container.

- Iterators provide **a generalized way** to traverse and access elements stored in a container.
  - can be ++ or -- (move to next or prev element)
  - dereferenced with *
  - compared against another iterator with == or !=

- Iterators are generated by STL container member functions, such as begin() and end().

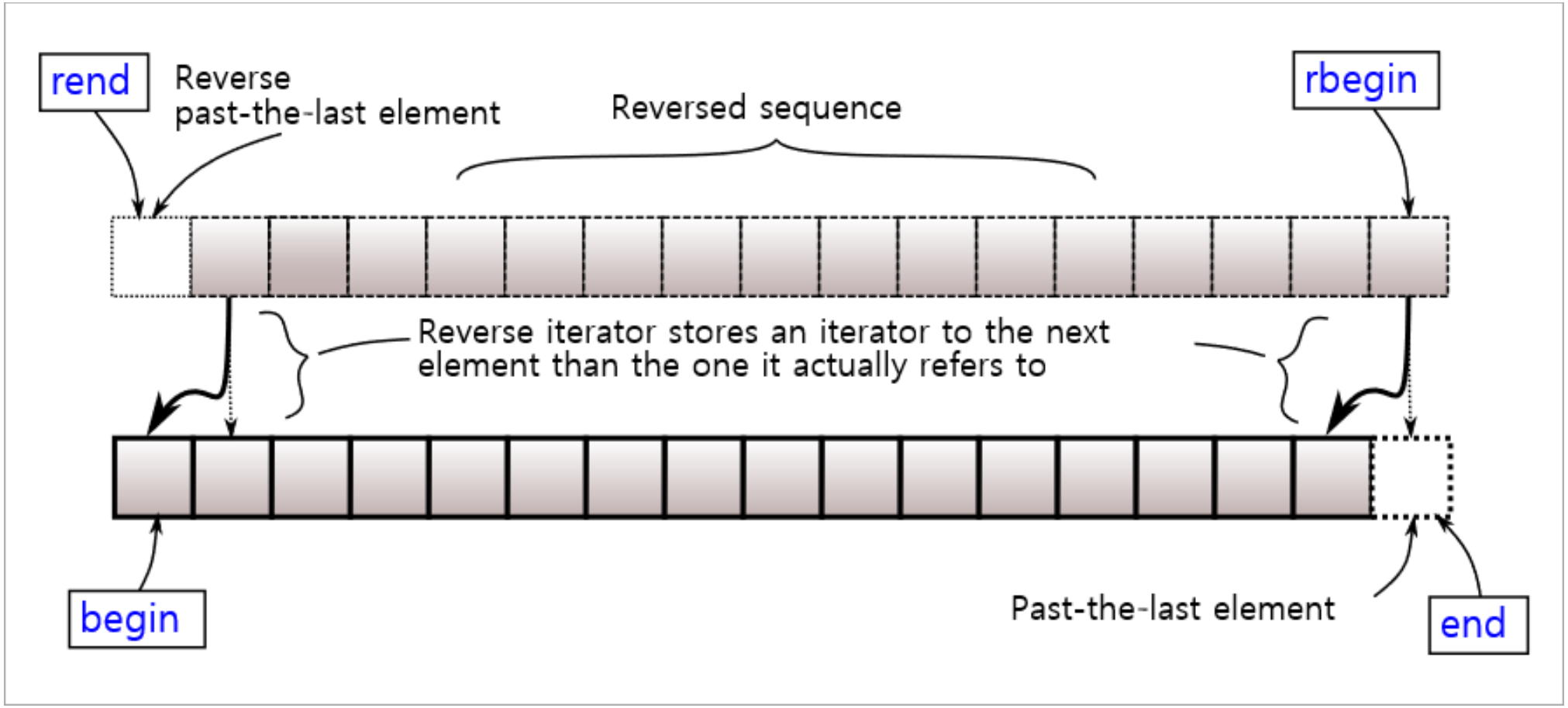
std::vector with iterator

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

void printVec(vector<int> intV, string name){

    vector<int>::iterator iter;
    cout << name << " ";
    for (iter=intV.begin(); iter != intV.end(); iter++)
        cout << *iter << " ";
    cout << endl;

}

int main(void){

    vector<int> intVec(5);
    vector<int>::iterator iter = intVec.begin();

    for(int i=0; i < 5; i++){
        *iter = i;
        iter++;
    }

    printVec(intVec, "intVec");
    intVec.insert(intVec.begin()+2, 100);
    printVec(intVec, "intVec");
    intVec.erase(intVec.begin()+2);
    printVec(intVec, "intVec");
}
```

std::vector with iterator

```
#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
}
```

Meaning of begin(), end(), rbegin(), rend()



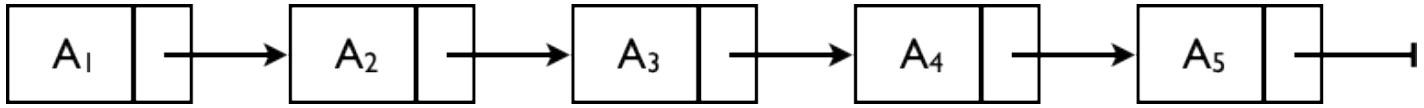
Quiz #1

- What is the expected output? (including compile/runtime error)

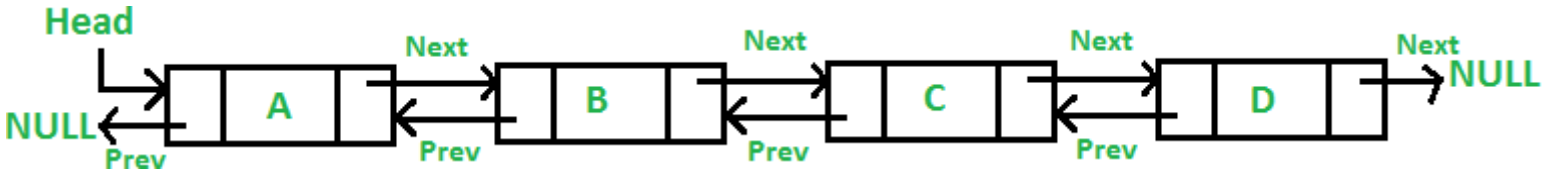
```cpp
#include <iostream>
#include <vector>

int main()
{
    // Create a vector containing integers
    std::vector<int> v = { 7, 5, 16, 8 };

    // Add two more integers to vector
    v.push_back(25);
    v.insert(v.begin(), 13);

    // Print out the vector
    std::cout << "v = { ";
    for (int n : v) {  // Range-based for loop (since C++11)
        std::cout << n << ", ";
    }
    std::cout << "}; \n";
}
```

Concept of Linked List

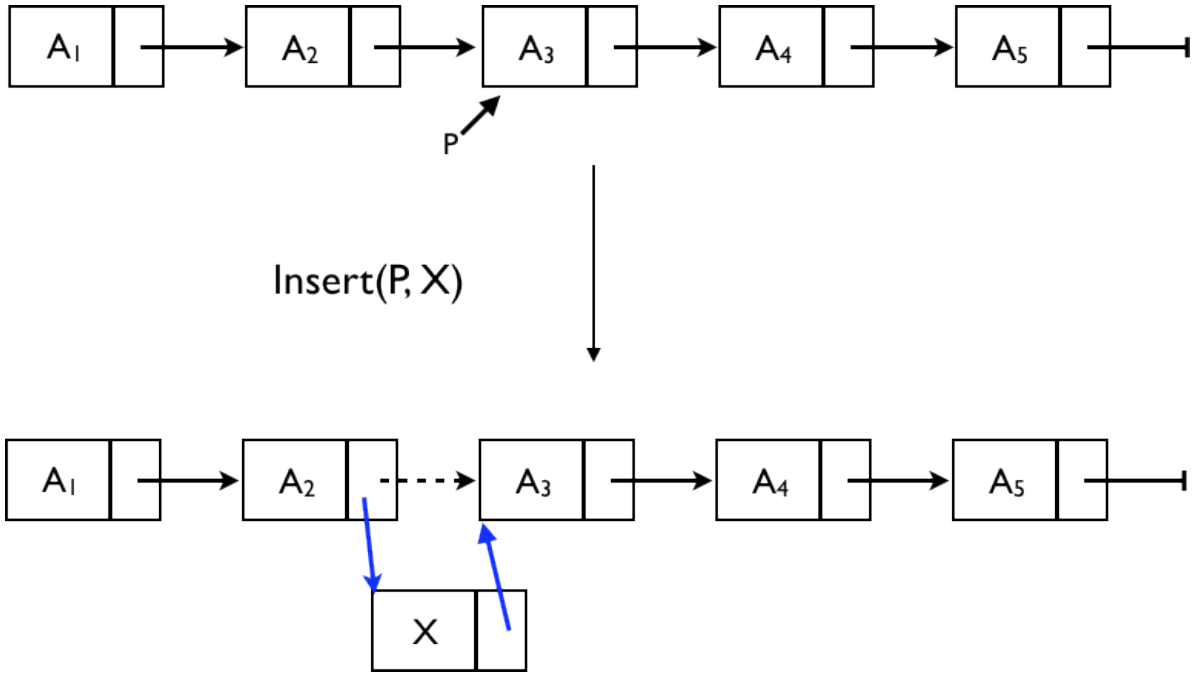
- Singly linked list: A node consists of the data and a link to the next node.

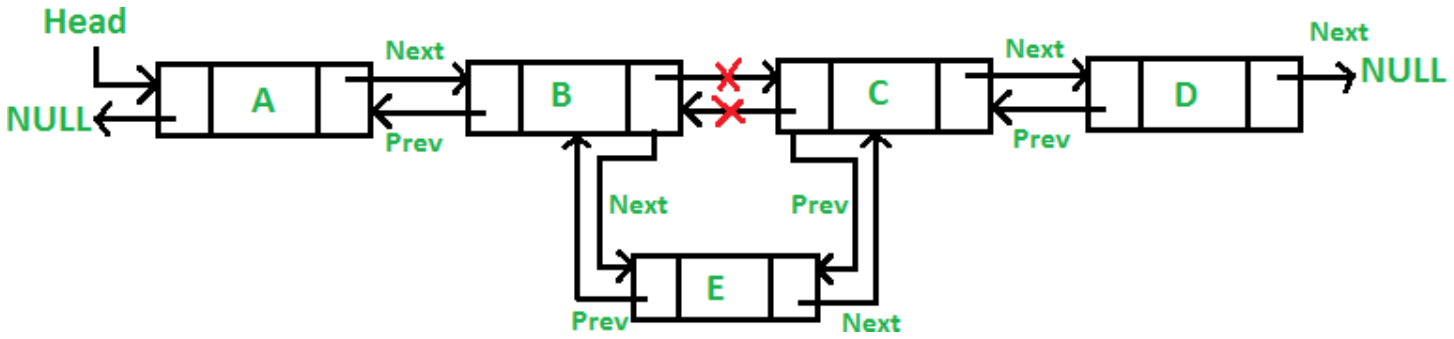


- Doubly linked list: with links to prev. & next node.

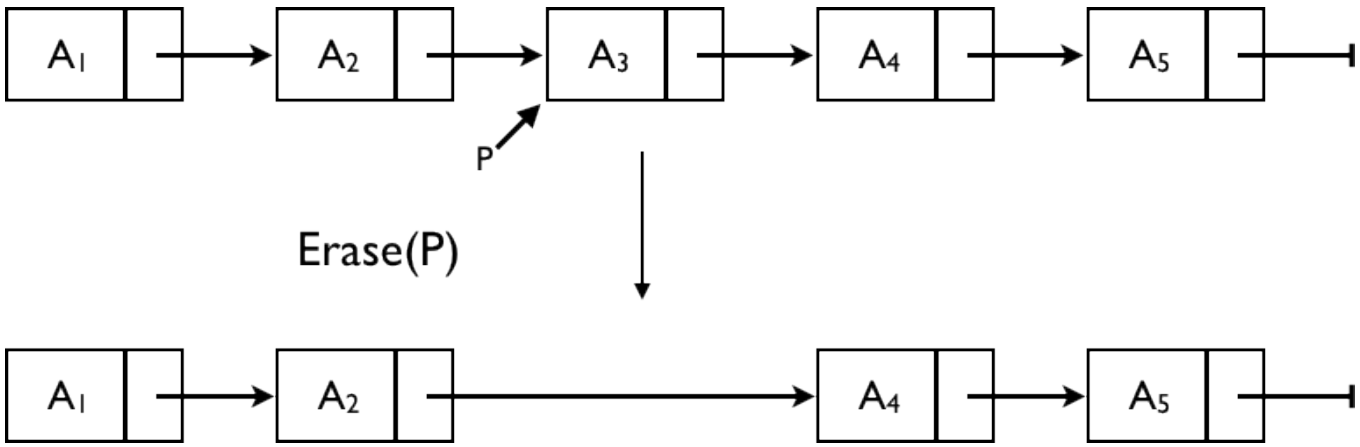


Concept of Linked List: insert





Concept of Linked List: erase



std::list

- Implemented as a doubly-linked list.
  - Non-contiguous storage.
- Sequential access
  - One should iterate from a known position (like begin() or end()) to access to some element.
- Fast addition/removal of elements **anywhere** of the sequence.

std::list – an insert and erase example

```
void printList(list<int> intV){

    list<int>::iterator iter;
    for (iter = intV.begin(); iter != intV.end(); iter++){
        cout << *iter << " ";
    }
    cout << endl;
}

int main(){

    list<int> intL(5);
    list<int>::iterator iter = intL.begin() ;
    for (int i=0; i < 5; i++){
        *iter = i;
        iter++;
    }
    printList(intL);

    iter = intL.begin();
    iter++;
    iter = intL.insert(iter, 100);
    printList(intL);

    iter++; iter++;
    cout << *iter << endl;
    intL.erase(iter);
    printList(intL);

    return 0;
}
```

An iterator that points to the first of the newly inserted elements.

```
0 1 2 3 4
0 100 1 2 3 4
2
0 100 1 3 4
```

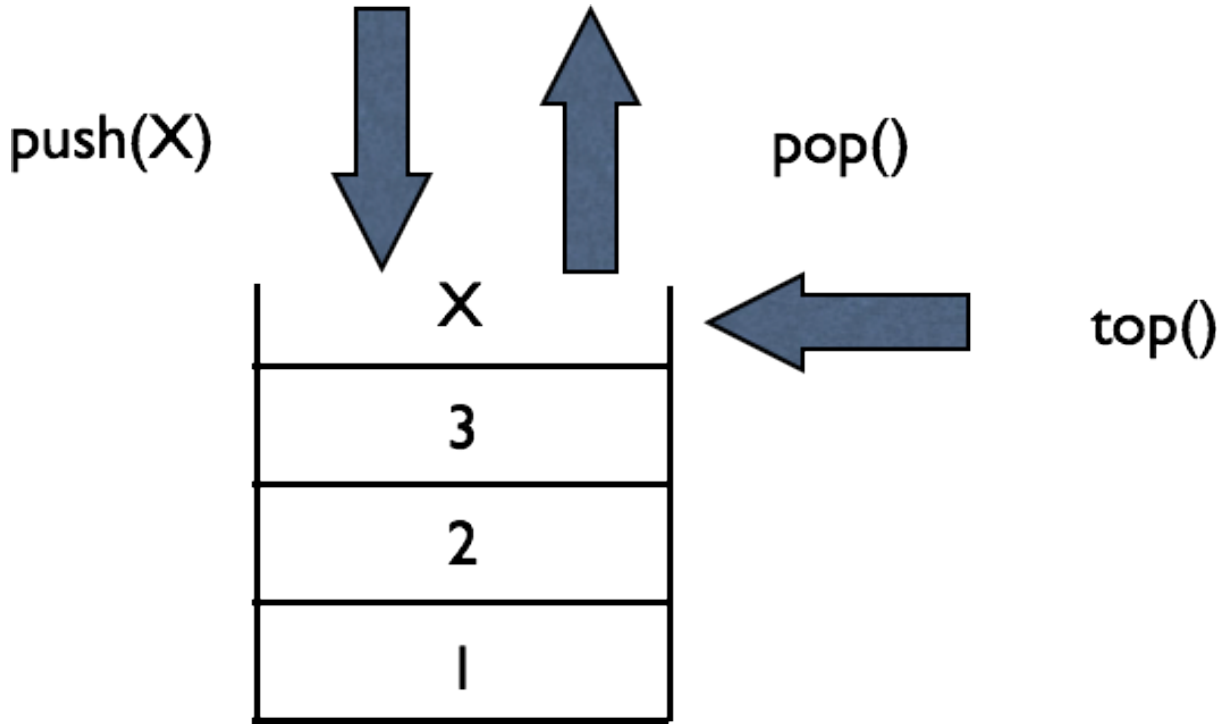
std::list – a remove example

```
#include <list>
#include <iostream>
#include <algorithm>
using namespace std;

int main(){

    list<int> lt;
    lt.push_back(10);
    lt.push_back(20);
    lt.push_back(30);
    lt.push_back(40);

    list<int>::iterator iter;
    for(iter=lt.begin(); iter != lt.end(); iter++)
        cout << *iter << ' ';
    cout << endl;

    iter = lt.begin();
    iter++;
    iter++;
    cout << *iter << endl;

    list<int>::iterator iter2 = lt.erase(iter);
    cout << *iter2 << endl;

    lt.remove(10);
    for(iter=lt.begin(); iter != lt.end(); iter++)
        cout << *iter << ' ';
    cout << endl;
    return 0;
}
```

An iterator pointing to the new location of the element that followed the last element erased by the function call.

```
10 20 30 40
30
40
20 40
```

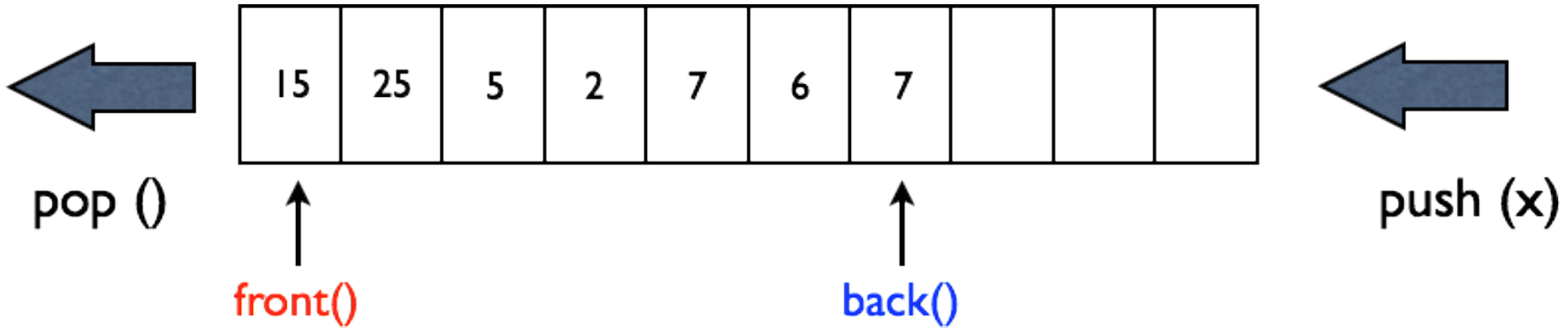
Concept of Stack : Last In First Out



std::stack - example

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

int main(){

    stack<int> st;

    st.push(10);
    st.push(20);

    cout << st.top() << endl;
    st.pop();
    cout << st.top() << endl;
    st.pop();

    if (st.empty())
        cout << "no data in the stack " << endl;
    return 0;
}
```

Concept of Queue : First In First Out



std::queue - example

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

int main(void){

    queue<int> q;
    cout << "size : " << q.size() << endl;

    q.push(10);
    q.push(20);
    q.push(30);

    cout << "size : " << q.size() << endl;
    cout << "front : " << q.front() << endl;
    cout << "back : " << q.back() << endl << endl;

    while(!q.empty()){
        cout << q.front() << endl;
        q.pop();
    }
    return 0;
}
```

```
size : 0
size : 3
front : 10
back : 30

10
20
30
```

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()`

std::map

- Contains key-value pairs with **unique keys**.
- Associative: Elements are referenced by their key, and always sorted by keys.
- Accessing with keys is efficient.

std::map - example

```
#include <map>
#include <iostream>

using namespace std;


int main(void){

  map <string, double> m;
  for (int i=0; i<4; i++)  m.insert(make_pair("string"+to_string(i), 0.5*i));
  for (map<string, double>::iterator it = m.begin(); it !=m.end(); ++it){
      cout << " " << it->first << "," << it->second << endl ;
  }

  m.insert(make_pair("apple", 10));
  m["orange"] = 3.14;
  m["string0"] = 111;

  for (map<string, double>::iterator it = m.begin(); it !=m.end(); ++it){
      cout << " " << it->first << "," << it->second << endl ;
  }

  map<string, double>::iterator it;
  it = m.find("apple");

  cout << "output " << it->first << " " << (*it).second << endl;
  m.clear();
  return 0;
}
```

std::set

- Contains **unique keys**.
- Associative: Elements are referenced by their key, and always sorted by keys.
- Accessing with keys is efficient.

std::set - example

```
#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; // s: 0 10 20 30 40 50 60 70 80 90
}
cout << s.size();
cout << s.empty();

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

Quiz #2

- What is the expected output? (including compile/runtime error)

```cpp
#include <iostream>
#include <map>
#include <string>

void print_map(const std::string& comment, const std::map<std::string, int>& m){
    std::cout << comment;
    // const_iteration is a read-only iterator
    std::map<std::string, int>::const_iterator it;
    for (it = m.begin(); it != m.end(); it++) {
        std::string key = it->first;
        int value = it->second;
        std::cout << key << " = " << value << "; ";
    }
    std::cout << "\n";
}

int main(){
    // Create a map of three strings (that map to integers)
    std::map<std::string, int> m { {"CPU", 10}, {"GPU", 15}, {"RAM", 20}, };

    print_map("Initial map: ", m);

    m["CPU"] = 25;  // update an existing value
    m["SSD"] = 30;  // insert a new value

    print_map("Updated map: ", m);
}
```

Iterator again

- Iterators provide **a generalized way** to traverse and access elements stored in a container.
- Iterators serve as **an interface** for various kinds of containers.
- Passing and returning iterators makes an algorithms more generic, because the algorithms will work for **any** containers.

Algorithm

- Many useful algorithms are available
  - sort
  - min, max, min_element, max_element
  - binary_search

std::sort

```
void sort(RandomAccessIterator first, RandomAccessIterator last);
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
```

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;


int main(void){

    vector<int> v;
    int input;
    cin >> input;
    while (input != 0) {
        v.push_back (input);
        cin >> input;
    }

    sort(v.begin(), v.end());

    for (int i = 0; i < (int)v.size(); i++)
        cout << v[i] << "\n";

    return 0;
}
```

std::min, std::max, std::min_element, std::max_element

```cpp
#include <vector>
#include <iostream>
#include <algorithm>
#include <cstdlib>  //for rand() and srand()
#include <ctime>    //for time()
using namespace std;

int main(){
const int a = 10, b = 15;
int minv = min(a,b);
int maxv = max(a,b);
cout << minv << " " << maxv << endl;

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

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

random_shuffle(v.begin(), v.end());
for (int i = 0; i < (int)v.size(); ++i)
    cout << " " << v[i];
cout << endl;

sort(v.begin(), v.end());
for (int i = 0; i < (int)v.size(); ++i)
    cout << " " << v[i];
cout << endl;

return 0;
}
```

std::string - constructor

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

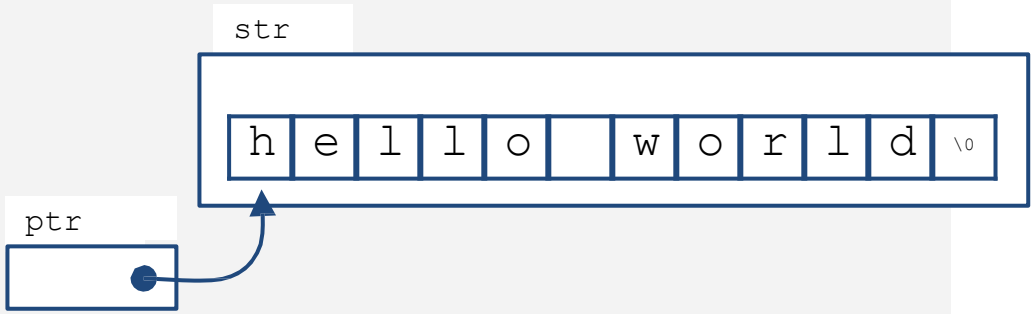
```cpp
#include <iostream>

using namespace std;


int main(void){

    string one("Lottery Winner!");        //string (const char *s)
    cout << one << endl;

    string two(20, '$');                  //string (size_type n, char c)
    cout << two << endl;

    string three(one);                   //string (const string & str)
    cout << three << endl;
    one += "Ooops!";
    cout << one << endl;

    return 0;
}
```

```
Lottery Winner!
$$$$$$$$$$$$$$$$$$$$
Lottery Winner!
Lottery Winner! Oops!
```

(Recall) std::string - c_str()

- Returns a pointer to a null-terminated string array representing the current value of the string object.

```
#include <string>
#include <cassert>
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 https://en.cppreference.com/w/cpp/string/basic_string
// resize(), substr(), find(), etc.
```



(Recall) std::string - input

```
std::string str;

std::cin >> str; // read a word (separated by a space, tab, enter)

std::getline(cin, str); // read characters until the default
                        // delimiter '\n' is found

std::getline(cin, str, ':'); // read characters until the delimiter
                             // ':' is found
```

(Recall) std::string - input

- Note that `std::string` automatically resize to the length of target string.

```
char fname[10];
string lname;
cin >> fname;       // could be a problem if input size > 9 characters
cin >> lname;       // can read a very, very long word
cin.getline(fname, 10);  // may truncate input
getline(cin, lname);     // no truncation
```

std::string - input from file

```
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
int main()
{
     using namespace std;
     ifstream fin;
     fin.open("tobuy.txt");
     if (fin.is_open() == false)
     {
        cerr << "Can't open file. Bye.\n";
        exit(EXIT_FAILURE);
     }
     string item;
     int count = 0;
     getline(fin, item, ':');
     while (fin)  // while input is good
     {
        ++count;
        cout << count <<": " << item << endl;
        getline(fin, item,':');
     }
     cout << "Done\n";
     fin.close();
     return 0;
}
```

std::string - find

```
#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.
```

std::string - substr

```
#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.
```

Quiz #3

- What is the expected output? (including compile/runtime error)

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

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

  string str2 = str.substr(3, 5);
  size_t pos = str.find("in");
  string str3 = str.substr(pos);

  cout << str2 << "\n" << str3 << endl;
}
```