게임프로그래밍

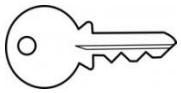
STL

박종승

Dept. of CSE, Incheon Nat. Univ.
jong@inu.ac.kr
http://ecl.inu.ac.kr

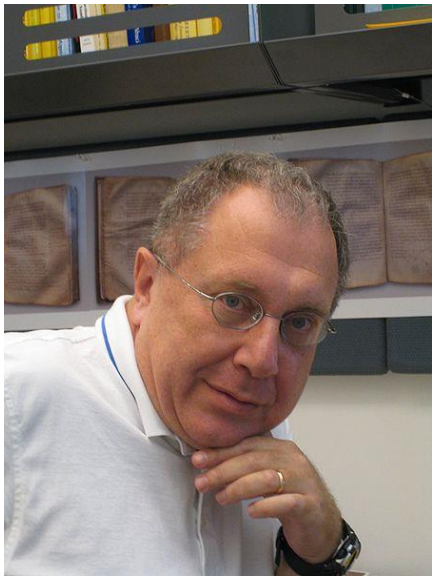
목차

- 표준 템플릿 라이브러리
- STL의 구성
- 순차적 컨테이너들
- 연관 컨테이너들
- string
- 반복자
- 컨테이너 어댑터
- 컨테이너들의 접근 함수들
- 함수객체란
- 함수 어댑터
- 알고리즘



표준 템플릿 라이브러리

- STL이란
  - 벡터, 집합, 연결 리스트, 탐색, 정렬 등
  - 소프트웨어 컴포넌트의 재사용성
  - 표준 템플릿 라이브러리
    - STL; C++ Standard Template Library
- STL의 탄생
  - by Alexander Stepanov
  - 만든 시기: 대략 1992년
  - 일반화 프로그래밍(generic programming)을 위한 C++ 언어용 라이브러리를 개발하였음
    - STL의 전신이 됨



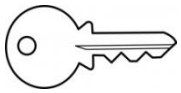
Alexander Stepanov
(1950-)

C/C++의 역사

- C의 역사 (C89, C90, C99, C11)
  - **1972**년 Ritchie가 C 언어 개발
  - 1989년 ANSI의 표준으로 C89 인증.
    - 널리 알려진 표준이며 ANSI C 또는 표준 C 라고도 함.
  - 1990년 ISO 표준으로 C90 인증.
    - C89를 약간 수정한 표준으로 사실상 C89와 동일.
  - 1999년 ISO 표준으로 C99 인증.
    - Microsoft사의 Visual Studio는 C99를 제대로 지원하지 않음. (2011년 현재)
      - C++에 더 치중하기 위함이 목적
      - C90까지는 C가 C++의 부분집합으로써의 역할을 하였음. 그러나, C99에서는 C가 더 확장된 역할을 하므로 C와 C++의 호환문제가 생김. C와 C++는 종속관계가 아님.
  - 2011년 ISO 표준으로 C11 인증.
- C++의 역사 (C++98, C++03, C++11, C++14, C++17)
  - **1983**년 Stroustrup가 C++ 언어를 개발.
  - 1994년 STL이 C++ 표준화 작업에 포함됨.
  - 1998년 ISO/IEC 표준으로 C++98 인증.
    - 현재의 대부분의 C++ 컴파일러가 이 버전에 기반함.
  - 2003년 ISO/IEC 표준으로 C++03 인증.
  - 2011년 ISO/IEC 표준으로 C++11 인증.
  - 2014년 ISO/IEC 표준으로 C++14 인증.
  - 2017년 ISO/IEC 표준으로 C++17 인증.

Dennis Ritchie
(1941-2011)

Bjarne Stroustrup
(1950-)

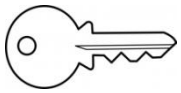


STL의 구성

- 컨테이너(container)
  - 객체들을 가지는 객체
- 반복자(iterator)
  - 포인터를 일반화한 것
- 함수객체(function object)
  - 함수처럼 동작하는 객체임
- 어댑터(adapter)
  - 위의 한 컴포넌트의 인터페이스를 바꾸는 컴포넌트임
  - 예: reverse_iterator는 한 반복자를 다른 반복자로 바꾸는 컴포넌트임
    - reverse_iterator는 순서가 반대방향인 점을 제외하면 원래의 반복자와 동일
  - 종류: 컨테이너 어댑터, 반복자 어댑터, 함수 어댑터
- 알고리즘(algorithm)
  - 공통적인 작업을 수행하는 함수들

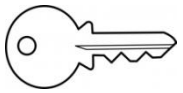
컨테이너

- 컨테이너의 종류
  - 순차적 컨테이너: vector, list, deque
  - 연관적 컨테이너: map, multimap, set, multiset
- 사용법 익히기
  - 모두 대부분 유사하게 사용됨
- 배우기 순서
  - vector
  - deque와 list
  - set와 map



순차적 컨테이너들

- 순차적 컨테이너들: vector, deque, list
  - vector, deque, list 의 나열 순서대로 구현 복잡도가 증가함.
    - 일반적으로 vector를 많이 사용
  - 헤더파일: <vector>, <deque>, <list>
- vector
  - 원소의 추가나 삭제가 마지막 위치에서만 이루어지는 경우에 효율적임
  - 임의 접근 연산자 '[]' 지원
- deque
  - vector 기능 + 맨 처음 위치에서의 원소의 추가나 삭제까지 고려
  - 임의 접근 연산자 '[]' 지원
- list
  - deque의 기능 + 중간의 위치에서의 추가나 삭제도 고려
  - 이중 연결 리스트로 구현되어서 모든 위치에서의 삽입과 삭제가 효율적임
  - 그러나, 임의 접근 연산자 '[]' 지원하지 않음.



vector

- vector의 용도
  - 동일 타입의 선형적인 나열. 길이는 가변적임.
  - 연산자 '[]'를 사용하여 모든 위치의 데이터를 임의 접근 가능
  - 나열된 모든 원소에 대한 임의접근 시간이 빠르고 원소의 추가나 삭제가 배열의 마지막에서만 이루어지는 경우에 가장 적합함
- 사용법
  - 헤더파일 '<vector>' ⟶ #include <vector>
- 비교

정적 배열을 사용한 예

```
#define SIZE 10
int arr[SIZE];
for (int i=0; i<SIZE; i++) arr[i] = i;
```

장: 메모리를 생성하고 반환하지 않아도 됨
단: 배열 크기를 상수로 지정해야 함

동적 배열을 사용한 예

```
int size = 10;
int* arr = new int[size];
for (int i=0; i<size; i++) arr[i] = i;
delete[] arr;
```

장: 배열 크기를 가변적으로 조절할 수 있음
단: 메모리를 생성하고 반환해야 함

vector를 사용한 예

```
int size = 10;
std::vector<int> arr(size);
for (int i=0; i<size; i++) arr[i] = I;
```

vector’

01.VectorSimple

- vector의 생성
  - 다양한 방법들

```
vector<int> v0; //빈 벡터를 생성
vector<int> v1(3); //크기가 3이고 각각 디폴트값으로 0인 벡터를 생성
vector<int> v2(5, 2); //크기가 5이고 모두 값 2를 가지는 벡터를 생성
vector<int> v4(v2); //v2의 복제본 벡터를 생성
vector<int> v5(v4.begin()+1, v4.begin()+3); //v4의 부분 복제본 벡터를 생성. (v4[1],v4[2]).
```

- vector의 접근
  - iterator
    - v.begin() 벡터의 첫 번째 위치
    - v.end() 벡터의 마지막 위치의 다음 위치

```
vector<int>::iterator iter;
cout << "v2 =" ;
for (iter = v2.begin(); iter != v2.end(); iter++)
    cout << " " << *iter;
cout << endl;
```

- 다양한 vector 접근 함수들
  - push_back, size, [], at, pop_back, pointer

push_back, pop_back, size

- push_back
  - 벡터의 뒤쪽에 값을 추가

```
vector<int> v1;
v1.push_back(8);
if (v1.size() != 0) cout << "마지막 원소: " << v1.back() <<endl; //8
v1.push_back(9);
if (v1.size() != 0) cout << "새 마지막 원소: " << v1.back() <<endl; //9
```

- pop_back
  - 벡터의 마지막의 원소를 삭제

```
vector<int> v1;
v1.push_back(1);
cout << v1.back() << endl; //1
v1.push_back(2);
cout << v1.back() << endl; //2
v1.pop_back();
cout << v1.back() << endl; //1
```

- size
  - 벡터의 현재의 길이를 리턴

```
vector<int> v1;
v1.push_back(1);
int i = (int)v1.size();
cout << "벡터 길이: " << i << endl; //1
v1.push_back(2);
i = (int)v1.size();
cout << "새 벡터 길이 " << i << endl; //2
```

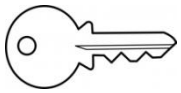
[], at, pointer

- [],at
  - []와 at 모두, 명시한 위치에서의 벡터 원소에 대한 참조를 리턴

```
vector<int> v1;
v1.push_back(1);
v1.push_back(2);
int& i = v1[0];
cout << "첫 번째 원소: " << i << endl; //1
int& j = v1.at(1);
cout << "두 번째 원소: " << j << endl; //2
```

- pointer
  - 벡터의 한 원소로의 포인터 타입.
    - 이 포인터를 사용하여 원소의 값을 수정할 수도 있다.

```
vector<int> v;
v.push_back(1);
v.push_back(2);
vector<int>::pointer ptr = &v[0];
cout << *ptr << endl; //1
ptr++;
cout << *ptr << endl; //2
*ptr = 4;
cout << *ptr << endl; //4
```



알고리즘

- 알고리즘
  - 공통적인 작업을 수행하는 함수들을 제공
    - 검색, 정렬, 비교, 수정 등을 위한 알고리즘을 제공함
  - 헤더파일: '<algorithm>', '<functional>', '<numeric>'
- 약 100개의 알고리즘 함수들을 지원
  - 예
    - 선형 검색: find
    - 최소와 최대: min, max
    - 구간의 정렬: sort
  - 더 자세히 : http://www.cplusplus.com/reference/algorithm/

sort

- sort
  - 함수 원형

```
template <class RandomAccessIterator>
  void sort ( RandomAccessIterator first, RandomAccessIterator last );

template <class RandomAccessIterator, class Compare>
  void sort ( RandomAccessIterator first, RandomAccessIterator last, Compare comp );
```

  - 기능
    - 범위 [first,last) 내의 요소들을 정렬
    - 비교함수객체
      - 디폴트: 오름차순('operator<')으로 정렬
      - 세 번째 인자로 주어진 'comp'에 따라 정렬
        - » 비교함수객체는 동일타입의 두 인자값을 취함
        - » 두 인자값 중 첫 번째가 먼저이면 true를 리턴함

sort'

01.VectorSimple

- sort
  - 간단 사용

```
vector<int> v1;
for (int i=0; i<8; i++)
    v1.push_back( rand() );
sort(v1.begin(), v1.end());
cout << "오름차순으로 정렬된 벡터 = " ;
for (vector<int>::iterator iter = v1.begin(); iter != v1.end(); iter++)
    cout << *iter << " ";
```

  - 인자
    - sort 함수의 세 번째 인자를 생략하면 오름차순으로 정렬됨
      - 디폴트인 less<int>가 사용됨
    - 내림차순으로 정렬하려면

```
sort(v1.begin(), v1.end(), greater<int>());
```

    - 지원하는 비교함수객체
      - less<int>(), greater<int>(), less_equal<int>(), greater_equal<int>()

sort”

- sort
  - 사용자 정의 함수를 사용

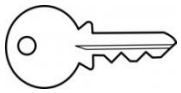
```
bool comp(int x, int y) { return x > y ; }
sort(v1.begin(), v1.end(), comp);
```

: greater<int>() 와 동일

  - 사용자 정의 클래스를 사용

```
class compObj {
public:
    bool operator()(int x, int y) { return x > y ; }
};
sort(v1.begin(), v1.end(), compObj());
```

: greater<int>() 와 동일

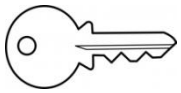


deque

02.DequeSimple

- deque
  - 사용자 정의 구조체 타입에 대한 deque를 선언하고,
  - 앞쪽과 뒤쪽에서 삽입과 삭제를 하는 예

```
struct MyStruct {
    int myInt;
};
deque<MyStruct> dq;

MyStruct m;
m.myInt = 4; dq.push_front(m);    //dq = 4;
m.myInt = 3; dq.push_back(m);     //dq = 4, 3
m.myInt = 5; dq.push_front(m);    //dq = 5, 4, 3
dq.pop_front();                   //dq = 4, 3
dq.pop_back();                    //dq = 4
```

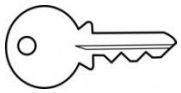


list

03.ListSimple

- list
  - 정수형 list를 선언하고, 값들을 임의의 위치에 채우는 예
  - insert: 값을 삽입하는 함수 * 주어진 위치 바로 앞에 삽입함
    - insert(삽입할 위치, 삽입할 값)
    - insert(삽입할 위치, 삽입횟수, 삽입할 값)
    - insert(삽입할 위치, 반복자의 first, 반복자의 last) //삽입할 값의 범위 [first, last)

```
list<int> c1;
c1.push_back(2); c1.push_back(1); c1.push_back(3); //c1 = 2, 1, 3

list<int>::iterator iter;
iter = c1.begin(); iter++;
c1.insert(iter, 8); //c1 = 2, 8, 1, 3

iter = c1.begin(); iter++; iter++;
c1.insert(iter, 2, 9); //c1 = 2, 8, 9, 9, 1, 3

list<int> c2;
c2.push_back(4); c2.push_back(5); c2.push_back(6); //c2 = 4, 5, 6
c1.insert(++c1.begin(), c2.begin(), --c2.end()); //c1 = 2, 4, 5, 8, 9, 9, 1, 3
```



연관 컨테이너들

- 연관 컨테이너들
  - 데이터를 연관성에 의해서 접근
  - map, multimap, set, multiset
  - 헤더파일: <map>, <set>

map

04.MapSimple

- map
  - 키와 값의 쌍들의 정렬된 집합
  - 쌍들은 키에 따라서 사용자가 정의하는 비교자에 의해서 정렬됨
    - 디폴트: less<T>
  - map은 일대일 대응관계를 지원함
    - 특정 키에 한 값만 연관될 수 있음
  - 예

```
map<const char*, int> m1;
m1["january"] = 31; m1["february"] = 28; m1["april"] = 30;
cout << "february" << " = " << m1["february"] << endl; //february = 28

cout << "march" << " = " << m1["march"] << endl; //march = 0

map<const char*, int>::iterator p = m1.find("january");
cout << p->first << " = " << p->second << endl; //january = 31
```

map'

- map
  - 정렬 순서도 사용자 정의 비교함수를 지정하여 바꿀 수 있음

```
struct MyStringCompare {
    bool operator()(const char* s1, const char* s2) const {
        return strcmp(s1, s2) < 0;
    }
};

map<char *, int, MyStringCompare> m1;
m1["january"] = 31; m1["february"] = 28; m1["april"] = 30;
//m1 = (april,30), (february,28), (january,31)
```

s1이 s2보다 앞에 오는 순서라면 true를 리턴해야 함

multimap

04.MapSimple

- multimap
  - 일대다 대응관계를 지원함
    - 특정 키에 대해서 여러 값들이 연관될 수 있음
  - 예: 정수와 문자열의 쌍에 대한 multimap

```
multimap<int, const char*> m;
m.insert(pair<int, const char* const>(2, "two"));
m.insert(pair<int, const char* const>(3, "three"));
m.insert(pair<int, const char* const>(2, "zwei")); //"zwei"= Two in the German language
cout << '₩t' << "#elements with key 2 = " << m.count(2) << endl; //2
multimap<int, const char*>:: const_iterator iter = m.find(2);
cout << '₩t' << "the first element with key 2 = " << iter->second << endl; //two
```

set

- set
  - 빠른 연관 조회(lookup)에 적합
  - set 내의 각 객체는 유일해야 하며 중복은 허용되지 않음
    - 집합의 특성상 중복된 값을 삽입하면 무시됨
  - 조회 시에 연산자'=='를 사용하여 객체의 일치 여부를 확인할 수 있음
  - 삽입 시에 객체들은 사용자가 정의한 비교자에 따라서 정렬되어 보관됨
    - 비교자는 디폴트로 less<Type>이 사용되어, 오름차순으로 정렬됨
    - 정렬 방식을 바꾸려면 'set<int, greater<int> >' 와 같이 두 번째 인자값을 지정하면 됨

set'

05.SetSimple

- set
  - 예

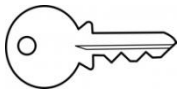
```
set<int> s1;
int A[8] = {4, 2, 1, 2, 1, 2, 4, 1};
for (int i=0; i<8; i++) s1.insert(A[i]); //s1 = 1, 2, 4

set<int> s2;
int B[5] = {5, 2, 3, 3, 1};
for (int i = 0; i < 5; i++) s2.insert( B[i] ); //s2 = 1, 2, 3, 5

set<int> s3;
set_union(s1.begin(),s1.end(), s2.begin(),s2.end(), inserter(s3,s3.begin()));
//s3 = (A 합집합 B) = 1, 2, 3, 4, 5

set<int> s4;
set_intersection(s1.begin(),s1.end(), s2.begin(),s2.end(), inserter(s4,s4.end()));
//s4 = (A 교집합 B) = 1, 2
```

multiset

05.SetSimple

- multiset
  - 중복 객체들을 허용한다는 점 외에는 set과 동일

```
multiset<int, less<int> > s1;
int A[8] = {4, 2, 1, 2, 1, 2, 4, 2};
for (int i = 0; i < 8; i++) s1.insert( A[i] ); //s1 = 1, 1, 2, 2, 2, 2, 4, 4

s1.insert(s1.begin(), 3); //s1 = 1, 1, 2, 2, 2, 2, 3, 4, 4

s1.erase(2); //s1 = 1, 1, 3, 4, 4
```



string

06.StringSimple

- string
  - C++에서 제공하는 문자열을 위한 안전하고 간편한 클래스
    - 문자열을 다루기 위한 기존의 'char*' 를 대치하기 위함
  - 헤더파일 <string>
  - 이름공간 std 내에 정의되어 있음
    - 따라서 "using namespace std;" 또는 "std::string" 식으로 언급해야 함

- string 기본
  - string 선언

```
string my_string;
string my_string("starting value");
```

  - string I/O

```
cin >> my_string; //공백으로 분리된 한 문자열만 입력됨
getline(cin, my_string, '₩n'); //한 라인 전체가 입력됨
```

string‘

- string 조작
  - string 붙이기
    - ‘+’, ‘+=’ 등의 연산자를 사용

```
string my_string1 = "a string";
string my_string2 = " is this";
string my_string3 = my_string1 + my_string2;
cout<<my_string3<<endl; // "a string is this"
```

  - string 비교하기

```
string passwd;
getline(cin, passwd, '₩n');
if (passwd == "xyzzy") //기대하는 대로 의미적으로 올바르게 수행됨
    cout<<"Access allowed";
```

  - string 길이 구하기
    - 주의: ‘char*’에서와 다르게, string은 null로 끝나지 않을 수도 있음!!

```
string my_string1 = "ten chars.";
int len = my_string1.length(); // or .size();
```

string‘’

- string 조작
  - string의 각 문자들 접근하기

```
for (unsigned i = 0; i < my_string.length(); i++)
    cout << my_string[i];

for (string::iterator my_iter = my_string.begin(); my_iter != my_string.end(); my_iter++)
    cout << *my_iter;
```

  - find 함수
    - int find(string pattern, int position);
      - 찾을 문자열 “pattern”을 위치 “position”에서부터 시작하여 검색함
      - 일치되면 바로 그 위치를 리턴함; 없으면 특별한 값 “string::npos”을 리턴함
      - 실제로 리턴타입은 int가 아닌 size_type 임 (unsigned int).
    - 함수 rfind 는 가장 뒤쪽에서부터 검색을 시작함.
      - 일치되는 “pattern”이 1번만 있다면, find와 동일한 결과임.
  - substr 함수
    - string substr(int position, int length);
      - 문자열의 일부를 따서 새로운 문자열을 생성함

```
string my_string = "abcdefghijklmnop";
string first_ten_of_alphabet = my_string.substr(0, 10);
cout<<"The first ten letters of the alphabet are " <<first_ten_of_alphabet;
```

string‘”

- string 수정
  - erase 함수

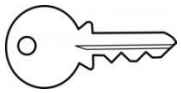
```
string my_string = "remove aaa";
my_string.erase(7, 3); // erases aaa

my_string.erase(0, my_string.length()); //delete an entire string
```

  - insert 함수

```
string my_string = "ade";
my_string.insert(1, "bc"); // my_string is now "abcde"
cout<<my_string<<endl;
```

- C-style 문자열 얻기
  - c_str(): const char* string.c_str();
    - 문자열을 ‘char*’ 타입으로 리턴함. 마지막 문자열을 null임.
    - 주의: const 타입으로 리턴하므로 리턴받은 값을 수정하지 말것!
- Wide characters
  - std::basic_string

```
typedef basic_string<char> string; //string의 원래 정의
typedef basic_string<wchar_t> wstring; //예시
```



반복자

- 반복자
  - 컨테이너의 내용에 접근할 수 있는 일반화된 포인터임
  - 반복자도 포인터에서와 같은 방법으로 사용
    - 증가 및 감소시키거나 '*' 연산자로 가리키는 값을 참조할 수 있음
  - 컨테이너와 알고리즘을 밀착시키는 역할
    - 각 컨테이너는 반복자를 제공하며 알고리즘은 반복자 인자에 대해서 작성되어 있음
  - 비교연산자의 사용: vector나 deque는 가능함. list는 불가능함.

```
for (iter = v1.begin(); iter < v1.end(); iter++)  // list는 불가능함
for (iter = v1.begin(); iter != v1.end(); iter++)  // vector,deque,list 모두 가능함
```

- 예

```
vector<int> v1(10);
for (unsigned i = 0; i < v1.size(); i++)
    v1[i] = i;

vector<int>::iterator iter;
for (iter = v1.begin(); iter != v1.end(); iter++)
    cout << *iter << " ";
cout << endl;
```

반복자의 카테고리

- 입력/출력 반복자(input/output iterator)
  - 한 번에 하나를 읽을/쓸 수 있고 앞으로만 전진할 수 있다.
- 전방향 반복자(forward iterator)
  - 입력/출력 반복자 모두를 모두 상속한 파생 클래스
  - 연산자 지원: 지정: '=', 비교: '==', '!=', 역참조(dereference): '*', 전진: '++'
- 양방향 반복자(bidirectional iterator)
  - 전방향 반복자를 상속한 파생 클래스
  - 후진하는 기능을 추가시켰음.
  - 연산자 지원: 후진: '--'
  - 해당 컨테이너: **list**, **set**, multiset, **map**, multimap
- 임의접근 반복자(random access iterator)
  - 양방향 반복자를 상속한 파생 클래스
  - 임의의 거리만큼 전진 또는 후진하는 기능을 추가시켰음
    - C++에서의 포인터를 임의접근 반복자로 생각해도 됨
  - 연산자 지원: 임의 전진 또는 후진: '+', '-', '+=', '-=', 임의의 위치에 대한 역참조: '[]', 비교: '<', ' >', '>=', '<='
  - 해당 컨테이너: **vector**와 **deque**

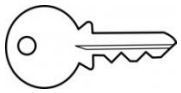
참고: 반복자 어댑터

- 반복자 어댑터
  - 삽입을 위한 반복자 어댑터에는 insert_iterator, back_insert_iterator,
  - front_insert_iterator가 있다.
  - 역방향을 위한 반복자 어댑터에는 임의접근 반복자를 위한 reverse_iterator와 양방향 반복자를 위한 reverse_bidirectional_iterator의 두 개가 있다.
- 예
  - reverse_iterator를 선언하고, 값들을 역방향으로 출력하는 예

```
vector<int> v1(10);
for (unsigned i = 0; i < v1.size(); i++) v1[i] = i;

vector<int>::reverse_iterator riter;
for (riter = v1.rbegin(); riter != v1.rend(); riter++)
    cout << *riter << " ";
cout << endl;
```

  - 뒤에서 앞으로 향하며 오름차순 정렬하므로, v1을 내림차순 정렬함

```
//sort(v1.begin(), v1.end());
sort(v1.rbegin(), v1.rend());
```



컨테이너 어댑터

- 컨테이너 어댑터
  - stack
  - queue, priority_queue
  - 헤더파일: <stack>, <queue>

- stack
  - LIFO(Last In, First Out) 데이터 구조인 스택으로 동작되도록 함
  - stack은 push_back과 pop_back 멤버함수를 가지는 어떤 컨테이너에도 적용될 수 있음
    - vector, deque, list가 이러한 컨테이너에 해당함.
    - deque가 주로 사용됨 (디폴트)

컨테이너 어댑터'

- queue
  - FIFO(First In, First Out) 데이터 구조인 큐로 동작되도록 함
  - queue는 push_back과 pop_front 멤버함수를 가지는 어떤 컨테이너에도 적용될 수 있음
    - deque와 list가 이러한 컨테이너에 해당한다.
    - deque가 주로 사용됨 (디폴트)
- priority_queue
  - 가장 앞에 위치한 원소가 가장 큰 원소이도록 큐를 유지함
  - priority_queue의 가장 앞의 위치는 "back"임
    - 원소 삭제는 "back"에서 수행됨
  - priority_queue는 임의접근 반복자를 가지면서 front, push_back, pop_back 멤버함수를 가지는 어떤 컨테이너에도 적용될 수 있다.
    - vector와 deque가 이러한 컨테이너에 해당한다.
    - vector가 주로 사용됨 (디폴트)

stack,queue 예

07.StackQueueSimple

- stack
  - 10개의 정수를 stack에 push하고, 각각을 pop하는 예.
    - 9에서 0까지의 값들이 순서대로 출력됨

```
stack<int, deque<int> > stack;
for (int i = 0; i < 10; i++) stack.push(i);

while (stack.size()) { cout << stack.top() << " "; stack.pop(); }
cout << endl;
```

- queue
  - 10개의 정수를 queue에 push하고, 각각을 pop하는 예.
    - 0에서 9까지의 값들이 순서대로 출력됨

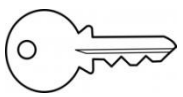
```
queue<int, deque<int> > queue;
for (int i = 0; i < 10; i++) queue.push(i);

while (queue.size()) { cout << queue.front() << " "; queue.pop(); }
cout << endl;
```

priority_queue 예

08.PriorityQueueSimple

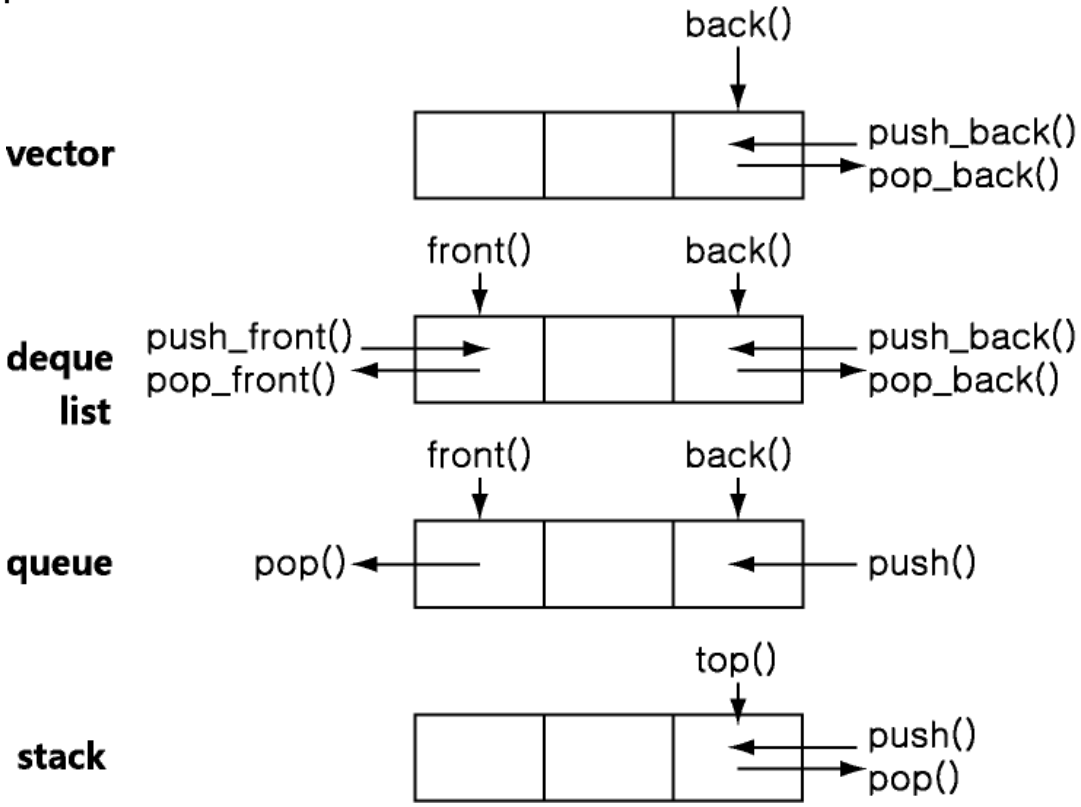
- priority_queue
  - 10개의 정수를 priority_queue에 push하고, 각각을 pop하는 예.
    - 항상 가장 큰 수가 pop되므로 큰 수부터 작은 수의 순서로 출력됨
      - 생성자의 세 번째 인자는 디폴트로 less<T>이며, 가장 큰 수부터 출력됨
        - » greater<T>로 바꾸면 가장 작은 수부터 출력됨

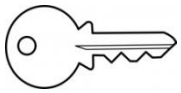
```
priority_queue<int, vector<int>, less<int> > pqueue;
for (int i = 0; i < 10; i++) pqueue.push(rand());

while (pqueue.size()) { cout << pqueue.top() << " "; pqueue.pop(); }
cout << endl;
```



컨테이너들의 접근 함수들

- 접근 함수들의 요약
  - vector, deque, list
  - queue, stack





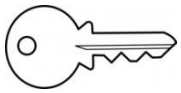
함수객체란

- 함수객체(function object)
  - 'operator()' 함수를 겹지정하는 클래스 또는 구조체의 객체
  - 함수처럼 동작하는 객체
    - 객체이므로 함수와는 달리 생성할 수 있다.
    - 함수처럼 동작하므로 값을 리턴한다.

```
class Matrix {
public:
        Matrix(int r, int c) { /* 생략 */ };
        //...생략
        int operator()(int, int) const;
private:
        int m_matrix[5][5];
        int m_row;
        int m_col;
};

int Matrix::operator()(int r, int c) const {
        if ( r > 0 && r <= m_row && c > 0 && c <= m_col) return m_matrix[r][c];
        else return 0;
}
void main() {
        Matrix m(10, 10);
        //...생략
        int element = m(5, 5);
}
```

함수객체란'

- 사용
  - 알고리즘은 함수객체를 인자로 받아들여서 알고리즘의 디폴트 동작을 바꿀 수 있도록 하고 있음
  - 산술연산, 비교연산, 논리연산의 기능을 위한 여러 함수객체들을 지원함
    - 산술연산을 위해서 plus, minus, times, divides, modulus, negate를 제공
    - 비교연산을 위해서 equal_to, not_equal_to, greater, less, greater_equal, less_equal을 제공
    - 논리연산을 위해서 logical_and, logical_or, logical_not를 제공
  - 더 자세히 : http://www.cplusplus.com/reference/std/functional/



함수 어댑터

- 함수 어댑터
  - 기존의 함수객체를 사용하여 다양한 함수객체를 만들 수 있도록 함
  - 헤더파일 '<functional>'
  - bind1st, bind2nd
    - 이항 함수객체를 단항 함수객체로 바꾸는 함수 어댑터
  - not1, not2
    - 주어진 함수객체의 연산 결과를 부정(negation)하여 리턴하는 함수객체로 바꿈
  - 더 자세히 : http://www.cplusplus.com/reference/std/functional/