C++0x

VS 2010 기능을 중심으로

김진욱 (http://rein.kr)

C++ 98/03 의 문제와 C++ 0x의 대안

1. 간단한(?) 벡터 순회 코드

```
std::vector<
  std::pair<std::string, std::string>> m;
// variable for the iterator of m
const std::vector<std::pair<std::string,
  std::string>>::iterator mend = m.begin();

for(std::vector<std::pair<std::string,
   std::string>>::iterator mi = m.begin();
  mi != mend; ++mi) {
  // do something
}
```

문제 1

복잡한 타입 선언

함수포인터 혹은 템플릿을 쓰는 경우, 복잡한 구조체 계층을 따라가기 힘들다
기존 해결책: typedef

기존 해결책

```
typedef std::vector<
  std::pair<std::string, std::string>>
    mapType;
typedef mapType::iterator mapIter;

const mapIter mend = m.end();
for(mapIter mi = m.begin(); mi != mend;
  ++mi) {
  // …
}
```

auto, decltype

타입 추론을 이용해서 선언 하기

```
auto x = m.begin();
const auto x2 = m.end();
const decltype(m.end()) x3 = m.end();

const auto mend = m.end();
for(auto mi = m.begin(); mi != mend; ++mi){
  // do something
}
```

auto, decltype

template type 추론

```
template<typename A0, typename A1>
auto Add(const A0 a0, const A1 a1)
    -> decltype(a0 + a1)
{
  return a0 + a1;
}

cout << Add(1, 2) << endl
 << Add(0.1, 1) << endl
 << Add(string("Hello,"), "world") << endl;
```

2. 문자열 벡터 길이 순으로 정렬하기

```
std::vector<std::string> v;
// v를 문자열의 길이로 정렬하자
std::sort(v.begin(), v.end(), ???);
```

문자열 벡터 길이 순으로 정렬하기

C 스타일 함수

```
// 전역 함수
bool strlencmp(const string& s1,
  const string& s2) {
    return s1.size() < s2.size()
}

// 실제 함수 호출
sort(v.begin(),v.end(), strlencmp);
```

문자열 벡터 길이 순으로 정렬하기

C++ 함수 객체

```
struct StringLengthComparator {
  bool operator( )(const string& s1,
    const string& s2) const {
      return s1.size() < s2.size();
  }
};

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

문자열 벡터 길이 순으로 정렬하기

Boost Library (lambda, bind)

```
#include <boost/lambda/lambda.hpp>
#include <boost/bind.hpp>

sort(v.begin(), v.end(),
  boost::bind(
    &string::size, boost::lambda::_1)
  < boost::bind(
      &string::size, boost::lambda::_2)
);
```

문제 2

간단한 함수 표현이 없다

STL 알고리즘을 사용하기 힘들다
함수 객체를 쓰는 건 너무 번잡하다
함수 포인터는 오버헤드가 있다
기존 해결책: Boost lambda? Boost bind?

제시한 해결책의 문제점

C 함수, C++ 함수 객체
- 구현을 위한 코드가 두 개로 쪼개진다
- 이름 충돌을 피해야 한다 (함수/클래스 선언)

Boost lambda, bind
- 외부 라이브러리가 필요하다
- 긴 컴파일 시간 / 복잡한 오류 메시지
- 기본적인 경우를 빼면 번잡한 문법

λ the ultimate

```
sort(v.begin(), v.end(),
 [ ](const string& s1, const string& s2){
    return s1.size() < s2.size();
});
```

lambda

Syntax

```
auto functor = // 선언
   [&v0, &v1, v2] (A0 a0, A1& a1)
     -> return_type
   {
       function_body;
   };

functor(a0Val, a1Ref); // 호출
```

lambda

Equivalent Structure Definition

```
struct Functor {
    T0& v0_, T1& v1_, const T2 v2_;

    Functor(T0& v0, T1& v1, const T2& v2)
     : v0_(v0), v1(_v1), v2_(v2) {
    }

    return_type operator()(A0 a0, A1& a1) {
      // body
    }
};
```

lambda

Definition Syntax: Capture List

```
[&v0, &v1, v2] (A0 a0, A1 a1,…)
  -> return_type
{
    function_body;
};
```

lambda

Capture List: Equivalent Definition

```
struct Functor {
    T0& v0_, T1& v1_, const T2 v2_;

    Functor(T0& v0, T1& v1, const T2& v2)
      : v0_(v0), v1(_v1), v2_(v2)
    {
    }

    return_type operator()(A0 a0, A1& a1) {
      // body
    }
};
```

lambda

Declaration Syntax: Argument List

```
[&v0, &v1, v2] (A0 a0, A1& a1)
  -> return_type
{
    function_body;
};
```

lambda

Argument List: Equivalent Definition

```
struct Functor {
    T0& v0_, T1& v1_, const T2 v2_;

    Functor(T0& v0, T1& v1, const T2& v2)
     : v0_(v0), v1(_v1), v2_(v2) {
    }

    return_type operator()(A0 a0, A1& a1) {
      // body
    }
};
```

lambda

Definition Syntax: Return Type

```
[&v0, &v1, v2] (A0 a0, A1& a1)
  -> return_type
{
    function_body;
};
```

lambda

Return Type: Equivalent Definition

```
struct Functor {
    T0& v0_, T1& v1_, const T2 v2_;

    Functor(T0& v0, T1& v1, const T2& v2)
     : v0_(v0), v1(_v1), v2_(v2) {
    }

    return_type operator()(A0 a0, A1& a1) {
      // body
    }
};
```

lambda

lambda 함수의 타입 문제

```
// empty lambda function
auto f = [](){};
decltype(f) f2 = f;// ok
decltype(f) f3 = [](){}; // ERROR!
```

lambda

lambda 함수의 타입 문제

- ✓ lambda 함수는 익명함수다
- ✓ 선언 될 때마다 “새로운 타입”이 된다
- ✓ 똑같은 선언을 해도, 타입은 다르다

그렇다면 lambda 함수는 어떻게 저장하는가?

lambda

lambda 함수의 저장/전달

```
auto f = [](){};
std::function<void (void)> = [](){};
template<typename F> void foo(F f) …
```

람다 함수를 저장하려면,

- ✓ auto 변수를 쓰거나
- ✓ std::function<R (A0, A1, …)>을 쓰거나
- ✓ 템플릿 인자로 써야

한다

lambda

Timer Service (1/3)

```
struct Runnable {
public:
  virtual void Run() = 0;
};
// 언젠가는 해당 작업을 실행한다
class TimerService {
public:
    void Schedule(Runnable* job,
                  DateTime t);
};
```

lambda

Timer Service (2/3)

```
template<typename F>
struct LambdaRunnable : public Runnable {
    F fun;
    lambdaRunnable(F& f) : fun(f) { }
    virtual void Run() { fun(); }
};

template<typename F>
Runnable* MakeRunnable(F& f) {
    return new LambdaRunnable<F>(f);
}
```

lambda

Timer Service (3/3)

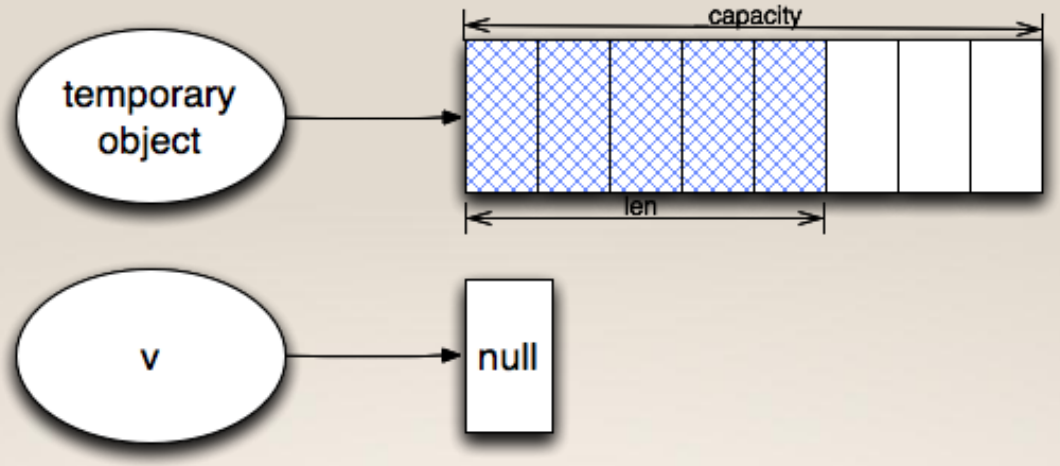
```
executorService.Schedule(
   // function as argument
   MakelambdaRunnable(
      [&val0, &val1, val2, …] ( ) {
         // some function body
      }
   ),
   someDateTime // datetime
);
```

3. 거대 벡터 만들어서 반환하기

```
std::vector<BigObj> MakeBigVector( … ) {
    std::vector<BigObj> x;
    for(int i = 0; I < N; ++i) {
       x.push_back(BigObj);
       x.back().a = …;
       x.back().b = …;
       x.back().c = new BYTE[…];
       …
    }
}
std::vector<BigObj> vbo = MakeBigVector();
```

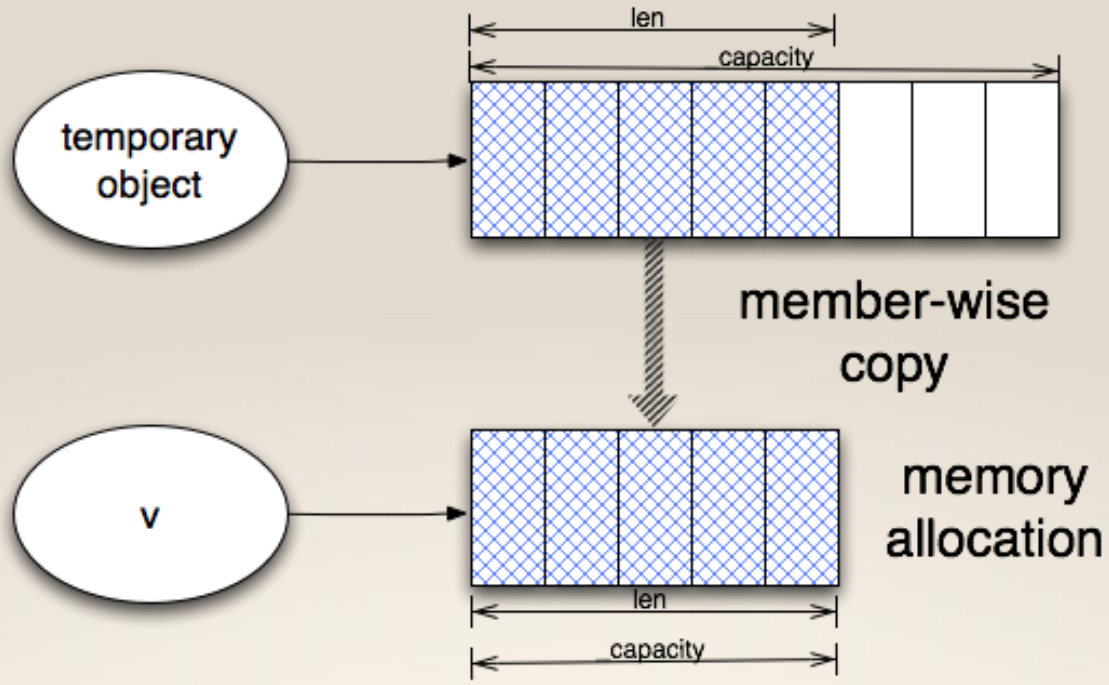
무슨 일이 일어날까?

복사가 일어나기 전 상태



무슨 일이 일어날까?

복사할 때 일어나는 일



무슨 일이 일어날까?

실제 코드

```
vector<BigObj> tmp; // function return val.
vector<BigObj> v;

v._ptr = new BigObj[tmp._len];
v._len = v._capacity = tmp._capacity;
for(unsigned i = 0; I < v._len; ++i) {
    v._ptr[i] = tmp._ptr[i]; // expensive!
}
```

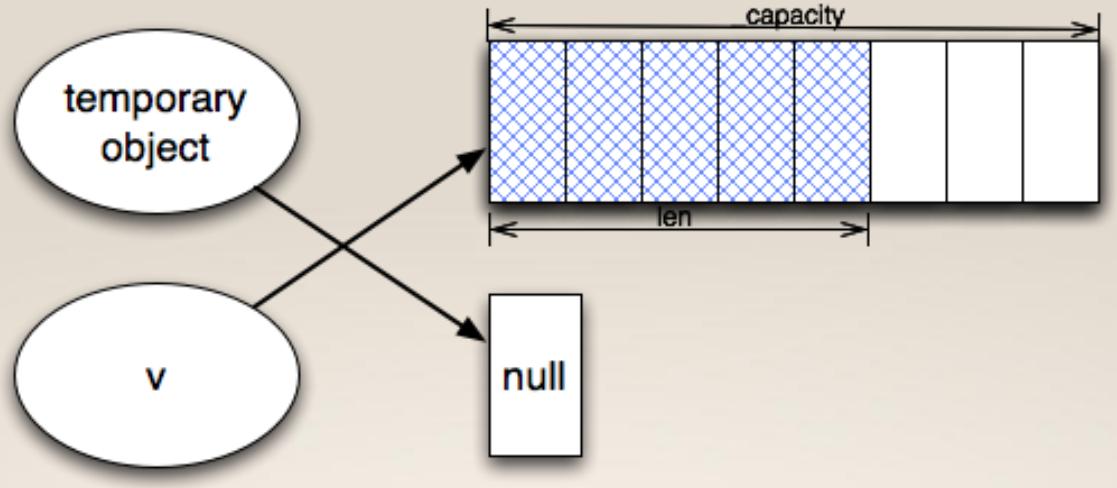
문제 3

불필요한 복사/생성

더 이상 쓰지 않을 객체를 생성하고 복사하는 오버헤드
기존 해결책: 컴파일러 최적화 믿기 (RVO/NRVO; 대부분 지원)
혹은 결과값 받을 컨테이너를 미리 전달하기

더 나은 해법

가볍게 복사하기?



임시 값 가볍게(?) 복사하기

코드로 표현하면

```
vector<BigObj> v = MakeBigVector(…);

// tmp는 MakeBigVector()가 반환한 임시 객체
swap(v._ptr, tmp._ptr);
swap(v._len, tmp._len);
swap(v._capacity, tmp._capacity);
```

이를 위해 임시 값을 표현할 문법이 필요하다

r-value reference

What is r-value?

r-value 는 주소가 없는 값을(임시 값) 말한다

하나의 C++ 표현식 안에서만 유효하다

```
// r-values: 대입식의 우변
string(“hello”);
42;
x + “, world” // x 는 std::string 객체
foo(); // string foo()인 함수가 있다고 치자
// errors
&string(“hello”);
&42;
&(x + ”, world”);
```

r-value reference

새 문법, 새 생성자

```
vector<T>&& // r-value 참조 문법
vector::vector(vector&& v); // 새 생성자

// std::string의 경우, 문자열을 합성하는
// 연산도 최적화 되어 있다
string operator+(string&& _l, string&& _r);
string operator+(string& _l, string&& _r);
string operator+(string&& _l, string& _r);
```

이걸 **move** semantic 이라 부른다

r-value reference

두 종류의 복사 생성자

```
vector::vector(const vector<T>& v) {
    this->_ptr = new T[v._len];
    this->_len = this->_capacity = v._len;
    for(unsigned i = 0; i < _len; ++i)
        this->_ptr[i] = v._ptr[i];
}
// 실제 벡터 구현과는 다르다; 이건 예제
// 실제론 생성 + 복사가 아니라 복사 생성
```

원래 C++에 있던 복사 생성자

r-value reference

두 종류의 복사 생성자

```
vector::vector(vector<T>&& v) {
   std::swap(v._ptr, this->_ptr);
   std::swap(v._len, this->_len);
   std::swap(v._capacity, this->_capacity);
}
```

임시 객체 관련 최적화가 이루어진
새 복사 생성자

r-value reference

두 종류의 복사 생성자

- ✓ 첫 번째 생성자가 const vector<T>&, vector<T>&, const vector<T>&& 를 받는다
- ✓ 두 번째 생성자는 movable(!)한 임시 객체만 받는다 (move constructor)
- ✓ 대입 연산자도 비슷하게 두 가지 형태가 존재한다

r-value reference

Supplement: Named r-value problem – use move( )

```
struct X {
    std::string s;
    X(const X& x) : s(x.s) { }
    X(X&& x) : s(std::move(x.s)) { }
};
```

- ✓ 단순하지 않은 멤버 변수를 복사하는 경우, 내부 구현을 몰라서 단순히 swap( )할 수가 없다
- ✓ std::move 를 써서 r-value 임을 알려주자

문제 4

```
char buffer[N]; // 상수 N
…
assert(N >= 512); // 일정 크기 이상이어야
…
```

컴파일 타임 오류를 만들기가 힘들다

컴파일 시간에 특정 조건이 만족되는지 체크하려면 별도의 hack을 써야 한다
기존 해결책: #pragma, Boost StaticAssert, 길이 0인 배열 …

static_assert

컴파일 타임 assertion

```
char buffer[N];
static_assert(N >= 512,
  “not enough buffer”);
```

static_assert(조건, 오류 내용) 형태로 정의하면, 컴파일 시간에 조건을 검사하고, 오류인 경우 오류 내용을 출력한다

C++의 문제 5

```
std::vector<std::set<int>> v;
std::map<int,
   std::pair<int, std::string>> m;
```

템플릿을 중첩으로 선언할 때 >> 사이에 공백이 필요하다

템플릿을 중첩해서 사용하는 건 흔한 일이지만, 저렇게 쓰면 컴파일 오류
기존 해결책: typedef 혹은 공백 삽입

template 선언 형식 개선

>> 는 연산자로만 쓰는 게 아니다

```
// 다음은 C++ 0x 에선 정상적으로 컴파일 된다
std::vector<std::set<int>> v;
std::map<int,
    std::pair<int, std::string>> m;
```

꺾쇠 괄호 사이에 공백 없이 >> 로 닫아도, 정상적으로
템플릿 선언임을 인식하게 수정되었다

C++의 문제 6

```
// #define NULL 0; // it's what we have
void foo(char*);
void foo(int);
foo(NULL); //calls foo(int), not foo(char*)
```

NULL 값이 형 전환

NULL 이 0으로 정의되어, 자동으로 정수 혹은 부동 소수점 값이 될 수 있다
기존 해결책: 코드 리뷰, 강제 타입 캐스팅

nullptr

NULL 포인터를 대체하기 위한 새로운 상수

```
// #define NULL 0; // it's what we have
void foo(char*);
void foo(int);
foo(NULL); //calls foo(int), not foo(char*)
foo(nullptr); // calls foo(char*)
```

C++ 전역으로 정의된 상수 **nullptr** 를 NULL대신 쓰면 된다.

문제 7

```
enum E0 { V0 = 0, V1 = 1, … };
enum E1 { V0 = 10, x = 1, … };
(V1 == x); // 서로 다른 enum을 비교함
sizeof(v1) = ? // 구현에 따라 다름
```

type-safe 하지 않다
이름 충돌이 일어난다
크기를 지정할 수 없다

enum 은 그다지 안전하게 쓸 수 없다
기존 해결책: 코드 리뷰

Strongly-typed Enum

타입을 갖는 enum

```
// 크기를 BYTE로 제한하는 정의
enum ByteEnum : BYTE {
  val0 = 0, val1 = 1,
};
```

VS2010에는 크기를 (=타입)
지정하는 문법만 구현

Strongly-typed Enum

In C++0x

```
// C++0x의 정의를 따른 코드지만 VS 2010에서는
// val0 가 두 번 정의되었다고 오류를 냄
enum class SomeEnum { val0 = 0, val1 = 1, };
enum class OtherEnum { val0 = 0 };
```

C++ 0x에서는 enum class 의 경우,
별도의 "스코프"가 생겨서 val0 가 중복 선언이 아니다

C++0x의 표준 라이브러리

tuple

<tuple>

```
tuple<int, char, std::string> x;
get<0>(x) = 1;
int y = get<0>(x) + get<1>(x);
auto z = make_tuple(42, ‘*’, “*”);
x = z; // char* const converted to string
```

std::pair의 수평적인 확장

get<N>(tuple_val) 은 tuple_val이 상수인지에 따라 적절한 참조를 반환한다

고정 길이 배열

<array>

```
array<int, 4> a = { 1, 2, 3, 4 };
cout << "Length: " << a.size() << endl
  << "First: " << a.front()
  << ", 2nd: " << a.at(1)
  << ", 3rd: " << a[2]
  << ", last: " << a.back() << endl;
```

C 배열과 거의 같다 ([ ] 로 인덱싱 가능)

C 배열 만큼 효율적이면서 C++에서 쓰기 좋음

STL 알고리즘 에서 요구하는 함수들을 제공

Smart Pointer

어디서 메모리를 반환해야 할까?

```
void mainLoop(DateTime& now) {
    vector<Obj*> objs;
    …

    Obj* obj = new Obj(…);
    AddTimerEvent(obj, now + delta);
    objs.push_back(obj);

    for(auto oi = objs.begin();
        oi != objs.end(); ++oi)
      (*oi)->UpdateState(now);
}
```

Smart Pointer

shared_ptr<T> from <memory>

- ✓ shared_ptr<T>는 리퍼런스 카운팅을 한다
- ✓ 여러 곳에서 참조해도, 마지막으로 참조가 끝나는 곳에서 "자동적으로" 삭제 된다
- ✓ -> 연산자로 원래 포인터처럼 쓸 수 있다

Smart Pointer

shared_ptr<T> 를 써서 쉽게 짜기

```
void mainLoop(DateTime& now) {
    vector<shared_ptr<Obj>> objs;
    …

    shared_ptr<Obj> obj(new Obj());
    AddTimerEvent(obj, now + delta);
    objs.push_back(obj);

    for(auto oi = objs.begin();
        oi != objs.end(); ++oi)
      (*oi)->UpdateState(now);
}
```

Smart Pointer

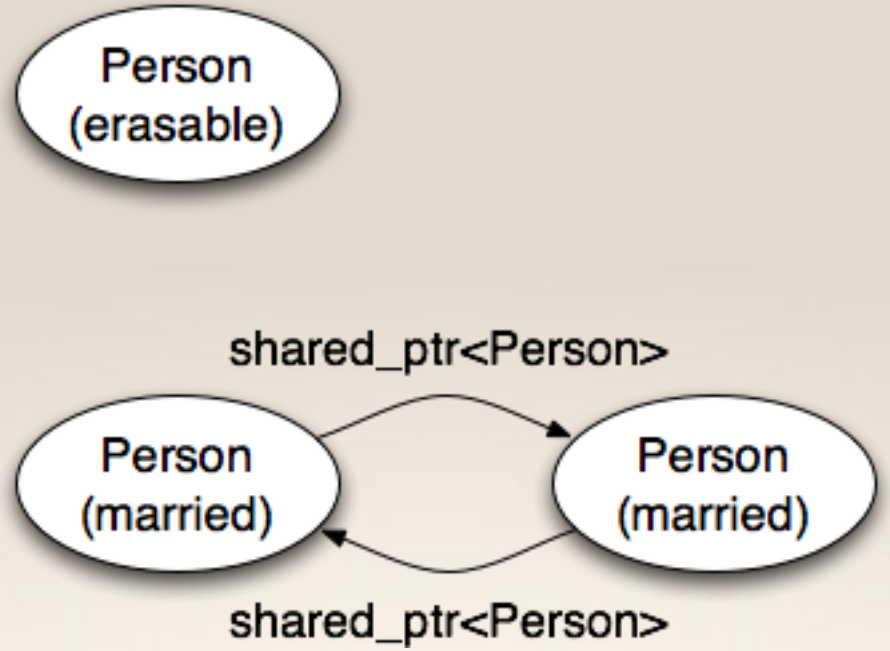
GetPartner의 반환 값을 지울 수 있는가?

```
struct Person : public obj {
    string name;
    shared_ptr<Person> partner;
};

shared_ptr<Person>
    GetPartner(shared_ptr<Person> person) {
    return person->partner;
}
```

Smart Pointer

상호 참조는 지울 수 없다



Smart Pointer

weak_ptr<T> from <memory>

```
struct Person : public obj {
    string name;
    weak_ptr<Person> partner;
};

shared_ptr<Person>
    GetPartner(shared_ptr<Person> person) {
    return person->partner.lock();
}
```

Smart Pointer

unique_ptr<T> from <memory>

기존의 문제가 많던 auto_ptr<T>를 대체하는
unique_ptr<T> 가 추가되었다

Hash Table

<unordered_map>, <unordered_set>

```
unordered_map<int, string> stringMap;

stringMap.insert(make_pair(42, “*”));
stringMap.find(40); // == stringMap.end()
stringMap[42] == “*”;
stringMap.at(42) == “*”;
stringMap.erase(42);
```

정규 표현식

<regex>

```
regex pattern("(\\d{4}[- ]){3}\\d{4}");
regex_match("1234-5678-1122-3344", e);// #t
regex_match(“123-45678-1122-3344”, e);// #f
```

정규 표현식을 C++에서 사용하는 몇 가지 문자열 형태(C 문자열, string, wstring)과 매칭 해주는 함수들을 제공한다

Perl/PHP 처럼 매칭/검색/변경을 지원한다

Q&A