10.1 일반화와 템블릿

함수중복의 약점 매개변수의 타입만 다른 중복함수를 작성할 경우, 실수의 가능성∙코드의 길이가 길어짐∙수정소요 등 많은 문제 발생

매개변수 타입만 다른 중복된 함수들을 일반화 시킨 틀 제작 필요

일반화와 템플릿 선언 템플릿 형판

함수, 클래스코드를 찍어내듯 생산할 수 있도록 일반화시키는 도구

중복함수의 일반화 template 키워드 중복함수들을 일반화시킨 특별한 함수 제작가능 = 템플릿함수, 제네릭함수

템플릿 선언과 제네릭 타입 형식 template <제네릭타입 T>

함수…

ex) template <class T>

void mySwap(T& a, T& b) {...}

제네릭타입 선언 = 일반타입

템플릿으로부터의 구체화 구체화 중복함수들을 템블릿화하는 과정의 역과정

구체화를 통해 제네릭함수로부터 구체적인 함수 코드 제작

구체화된 함수 구체화를 통해 제작된 구체적인 함수

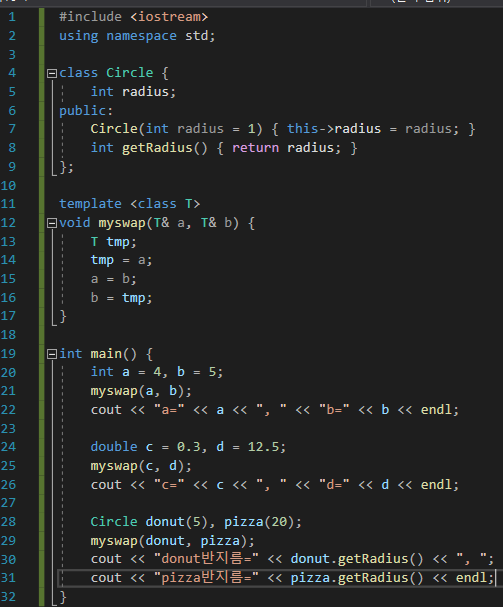
구체화 과정 1. 컴파일러는 함수호출문을 컴파일할 때 함수선언문을 찾음

2. 템블릿으로 선언된 함수 발견

3. 구체화 – 매개변수에 맞는 함수코드 제작

4. 구체화된 함수의 소스코드 컴파일, 이 함수를 호출하도록 컴파일

예제10-1



템블릿 역할 템블릿함수는 함수의 틀 컴파일X, 호출 X,

제네릭함수 선언, 구체화시키기 위한 틀

구체화 오류 제네릭타입 T에 유의해야 함 ex) template <class T> void mySwap(T& a, T& b) //두 매개변수 모두 타입T로 선언되어 있음

int s = 4;

double t = 5;

mySwap(s, t); //두개의 매개변수타입이 동일해야함

템플릿의 장단점과 제네릭 프로그래밍 장점 소프트웨어의 생산성과 유연성

단점 포팅 취약

오류메시지 빈약

제네릭 프로그래밍 = 일반화 프로그래밍

보편화, 보급화

제네릭과 매크로의 차이점 매크로 복잡한 함수, 클래스 표현 한계 🡪 부작용 초래 가능성 UP

제네릭 타입 타입 안전성 확보, 실행 중의 발생 오류 차단

check time 1. 구체화

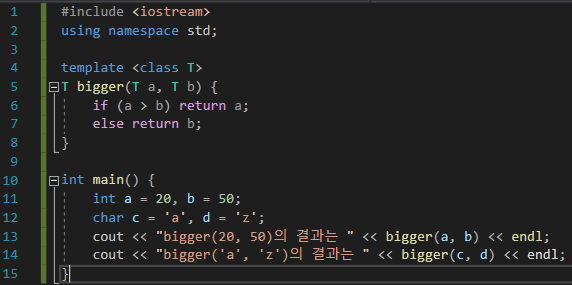
2. 컴파일

3. T 🡪 double

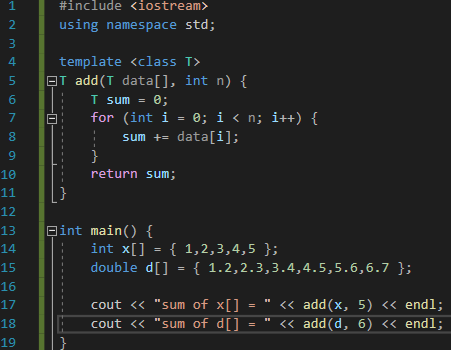
4. 3

10.2 다양한 제네릭 함수 만들기

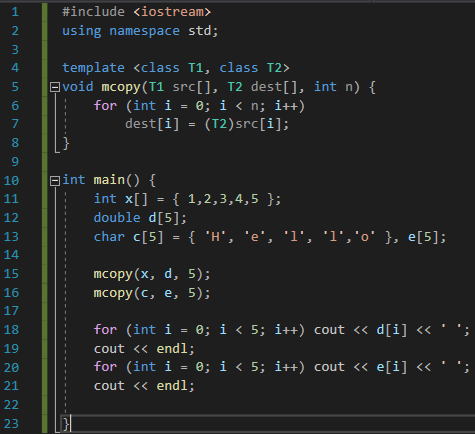
하나의 제네릭 타입을 가진 경우 두 개의 매개변수로부터 큰 값 구하기 예제10-2



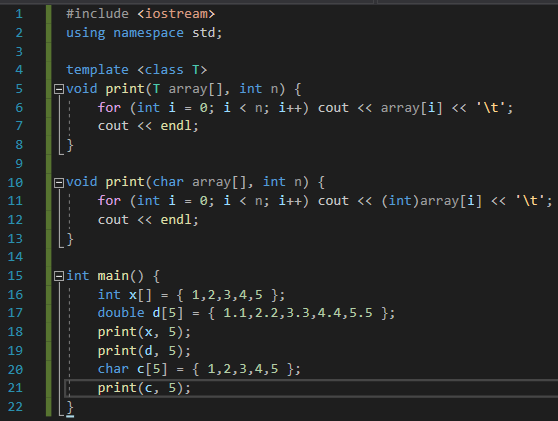
배열의 합 구하기 예제10-3



두 개 이상의 제네릭 타입을 가진 경우 하나의 배열을 다른타입의 배열로 변환해 복사 예제10-4



중복함수가 템플릿 함수보다 우선 템플릿 함수와 이름이 동일한 함수가 중복되어 있을 때, 중복된 함수를 템플릿 함수보다 우선해서 바인딩



템플릿 함수에 디폴트 매개 변수 사용 템플릿에서 함수 선언에 디폴트 매개 변수 사용

ex) void mcopy(T1 src[], T2 dest[], int n=5)

check time 1. template <class T>

T add(T x, T y) {

T n = x + y;

return T;

}

10.3 제네릭 클래스 만들기

제네릭 클래스 개요 데이터 타입을 일반화시킨 클래스

ex) int값∙double값∙char값∙Circle객체을 저장하는 스택 클래스 🡪 제네릭 스택 클래스

제네릭 클래스 선언 클래스 선언부, 구현부를 모두 template로 선언

제네릭 클래스 선언부 클래스에서 다루는 데이터타입이 하나 🡪 제네릭 타입은 T 하나만 필요함

ex) template <class T>

class MyStack {

int tos;

T data[100]; //T타입의 배열, 최대 100원소

public:

Mystack();

void push(T element); //T타입 원소 element를 data[]에 푸시

T pop();//스택의 탑에 있는 원소를 data[]에서 팝하여 리턴

};

제네릭 클래스 구현부 클래스이름을 “클래스이름”대신 “클래스이름<템블릿이름>”으로 사용

각 멤버함수 앞에 template<class T>를 붙여 제네릭 함수임을 선언

ex) template <class T>

void MyStack<T>::push(T element) {...}

template <class T> T MyStack<T>::pop() {...}

제네릭 클래스 구체화 제네릭 클래스 이용 시, 클래스의 이름과 함께 제네릭 타입 T에 적용할 구체적인 타입 지정

형식 클래스이름<자료형> 객체이름

ex) MyStack<int> iStack;

MyStack<double> dStack;

구체화과정 1. MyStack템플릿의 T에 int, double 적용 🡪 두 개의 구체화도니 버전의 C++클래스 소스 생성

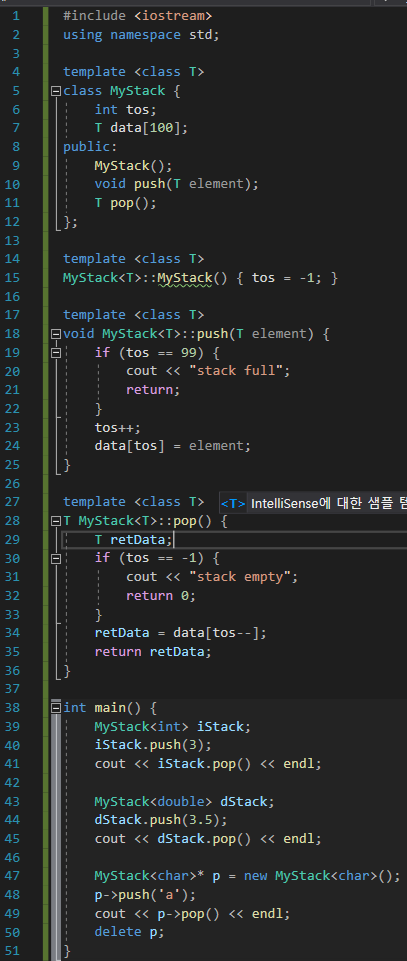
2. 두 C++ 클래스 컴파일 iStack, dStack 객체 생성

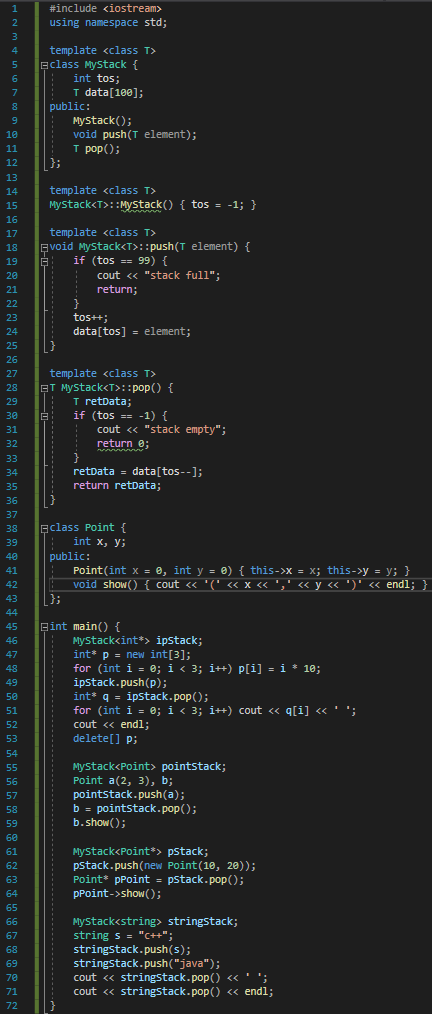
동적 생성 형식 클래스이름<자료형> \*포인터변수 = new 클래스이름<자료형>();

ex) MyStack<char>\* p = new MyStack<char>();

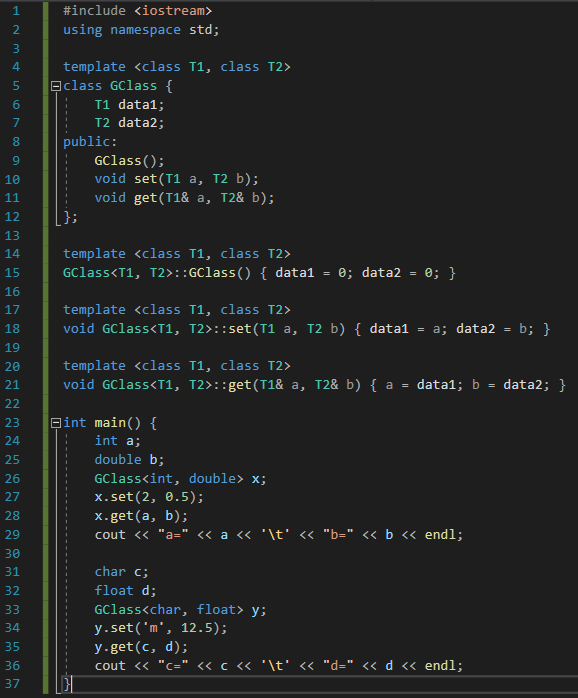
함수의 매개변수 타입이 제네릭 클래스일 때 제네릭 클래스가 매개변수인 함수 선언 형식 반환타입 함수이름(클래스이름<자료형> 매개변수이름)

ex) void popAll(MyStack<int> s) {...}

예제10-6 

예제10-7 

두 개 이상의 제네릭 타입을 가진 제네릭 클래스 템플릿을 사용해 2개 이상의 제네릭타입을 가진 제네릭 클래스를 만들 수 있음

예제10-8 

check time 1. (1) template <class T>

class TestClass {

T a;

public:

void set(T a);

T get();

};

template <class T>

void TestClass<T>::set(T a) { x = a; }

template <class T>

T TestClass<T>::get() { return x; }

1. (2) TestClass<int> tmp;

10.4 C++ 표준 템플릿 라이브러리(STL)와 활용

표준 템플릿 라이브러리 개요 표준 템플릿 라이브러리 Standard Template Library

템블릿으로 작성된 많은 제네릭 클래스와 함수 라이브러리

컨테이너 템플릿 클래스

데이터 저장∙검색하기 위해 담아두는 자료구조를 구현한 클래스

리스트, 큐, 스택, 맵, 셋, 벡터

순차컨테이너 연속적인 메모리 공간에 순서대로 값을 저장하는 읽는 컨테이너

컨테이너 어댑터 다른 컨테이너를 상속받아 기능을 제한하거나 변형한 컨테이너

stack, queue

연관어댑터 ‘키’로 ‘값’을 저장하고, ‘키’로 검색하여 ‘값’을 알아내는 컨테이너

set, map

iterator 컨테이너 원소에 대한 포인터

반복자

컨테이너의 원소들을 하나씩 순회접근하기 위한 컨테이너 원소에 대한 포인터

알고리즘 템플릿 함수

컨테이너의 원소에 대한 함수

컨테이너 클래스의 멤버함수 아님

헤더 파일 std 이름 공간 STL은 이름 공간에 작성되어 있음 🡪 using namespace std;

해당 템플릿이 선언된 헤더파일 include ex) vertor 클래스 – #include <vector>

알고리즘 함수 – #include <algorithm>

vertor 컨테이너 활용 가변길이배열을 구현한 제네릭 클래스

인덱스는 0부터 시작

원소 저장∙삭제∙검색

vector 객체 생성 형식 vector<자료형> 객체이름

ex) vector<int> v;

vector에 원소 삽입 push\_back() 입력값을 벡터의 맨 마지막에 삽입

ex) v.push\_back(1);

v.push\_back(2);

v.push\_back(3);

vector의 원소값 읽기 및 변경 at() 벡터의 원소 접근

ex) v.at(2) = 5;

[]연산자 벡터의 원소 접근

ex) v[2] = 5;

vector의 원소 개수 알아내기 size() 벡터의 원소 개수 반환

ex) v.size();

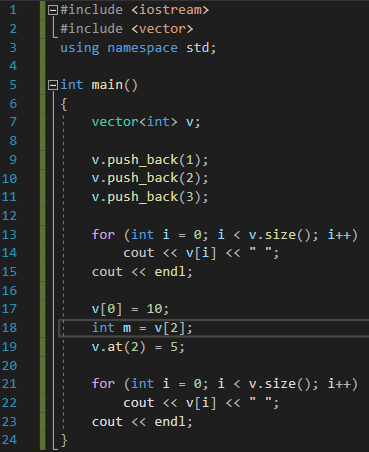
vector의 원소 삭제 erase() iterator와 함께 사용해야함

ex) vector<int>::iterator it; //정수벡터의 원소를 가리키는 포인터

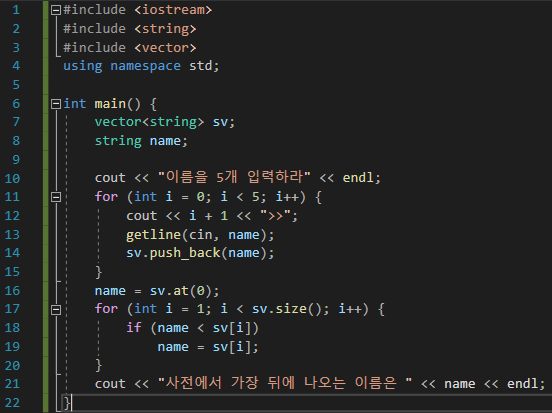
it = v.begin(); //it은 벡터 v의 첫번째 원소를 가리킴

it = v.erase(it); //백터 v에서 첫번째 원소 삭제 후 erase()의 리턴값으로 it재설정

예제10-9



예제10-10



iterator사용 iterator 컨테이너 안에 있는 원소들을 하나씩 순차적으로 접근하기 위한 원소에 대한 포인터

생성 시, 컨테이너 템플릿에 구체적인 타입 지정

형식 vector<자료형>::iterator 이터레이터객체

ex) vector<int>::iterator it; //현재 어떤 원소도 가리키고 있지 않음

벡터객체.begin() 백터객체의 첫번째 원소의 주소리턴

형식 이터레이터객체 = 백터객체.begin()

ex) it = v.begin();

포인터연산 ++ 다음 원소

ex) it++;

-- 이전 원소

ex) it--;

벡터객체.end() 백터객체의 마지막 원소의 다음 위치에 대한 주소리턴

형식 이터레이터객체 = 백터객체.end()

ex) it = v.end();

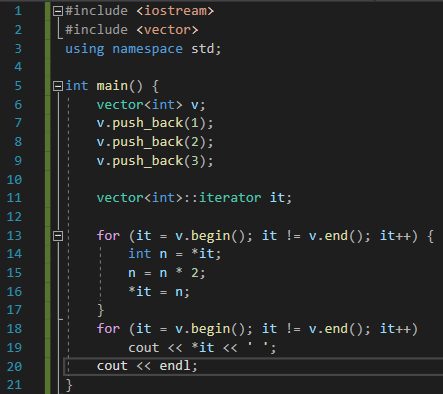
모든 원소를 검색하여 출력하는 for문 vector<int>::iterator it;

for (it = v.begin(); it != v.end(); it++) {

int n = \*it;

cout << n << endl;

}

예제10-11 

map컨테이너의 활용 ‘키’와 ‘값’쌍을 원소로 저장, 키를 통해 값을 검색하는 제네릭 컨테이너

#include <map> 필요

map 컨테이너 생성 및 원소 삽입 생성 map<자료형, 자료형> 맵객체이름

ex) map<string, string> dic;

삽입 insert(), make\_pair() 형식 맵객체.insert(make\_pair(키,값))

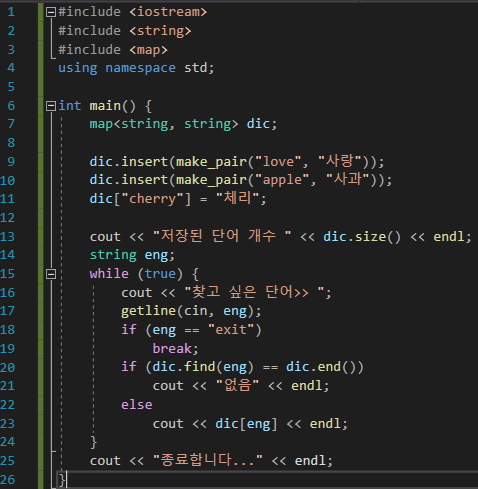
ex) dic.insert(make\_pair("love", "사랑"));

[]연산자 형식 맵객체[키] = 값

ex) dic["love"] = "사랑";

키로 검색하여 값 알아내기 []연산자 형식 맵객체[키]

ex) string kor = dic["love"];

예제10-12 

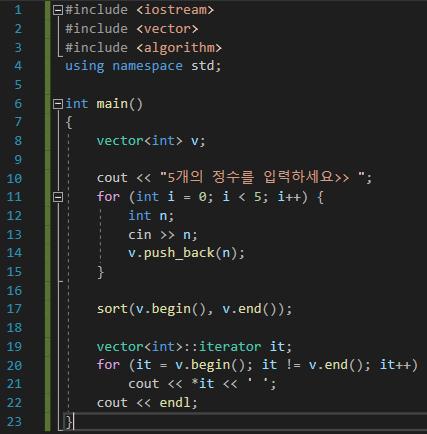
STL알고리즘 사용하기 전역 함수로 선언되어 있음

템플릿으로 작성됨

iterator와 함께 작동함 ex) vector<int> v;

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

sort(v.begin(), v.begin()+3);

예제10-13 

10.5 auto와 람다식

auto를 이용한 쉬운 변수 선언 auto키워드 변수 선언문으로부터 변수의 타입을 추론해 결정

복잡한 형식의 변수 선언 간소화, 오타∙번거로움 감소

기본 사례 ex) auto pi = 3.14;

ex) auto n = 3; //int

auto\* p = &n; //int\*

ex) int n = 10;

int& ref = n;

auto ref2 = ref; //int&

함수의 리턴타입으로부터 추론하여 변수 타입 선언 ex) int square(int x) { return x \* x; }

auto ret = square(3); //int

STL 템플릿에 활용 템플릿에 auto를 사용하면, 복잡한 변수 선언 간소화

ex) vector<int>::iterator it; //정수벡터의 원소를 가리키는 포인터

for (it = v.begin(); it != v.end(); it++) {

cout << \*it << endl;

}

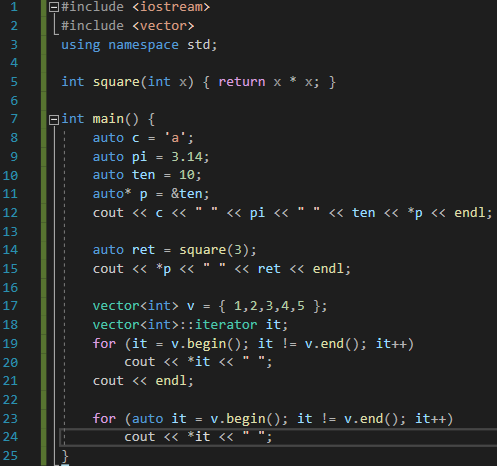
🡪

for(auto it = v.begin(); it != v.end(); it++) {

cout << \*it << endl;

}

예제10-14



람다 대수와 람다식 람다식 수학의 함수를 단순하게 표현하는 방법

이름없는 함수

C++에서 람다식 선언 람다식 이름없는 익명함수, 람다함수

형식 [캡쳐리스트](매개변수리스트) -> 리턴타입 { 함수코드 } ->리턴타입 은 생략가능

ex) [](int x, int y) -> int {return x + y; };

[](int x, int y) {cout << x + y; };

[](int x, int y) {cout << x + y; }(2, 3);

캡쳐리스트 람다식 외부에 선언된 변수 목록

람다식에서 사용하고자 할 때 나열

[x] 변수 x의 값 활용

[&x] 참조변수 x의 활용

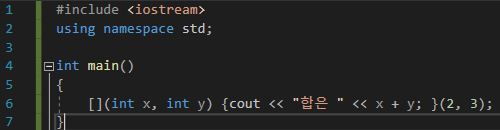
[=] 모든 변수의 값 활용

[&] 모든 참조 변수 활용

매개변수리스트 보통함수의 매개변수 리스트와 같음

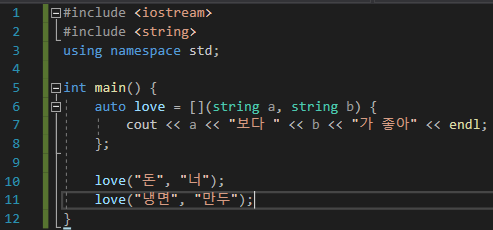
함수 바디 람다식이 호출될 때 실행되는 코드

함수를 작성하는 방법과 동일

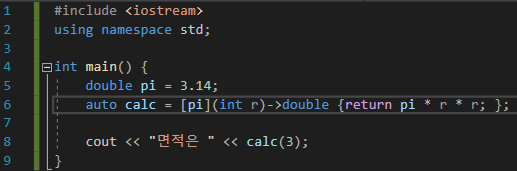
예제10-15 

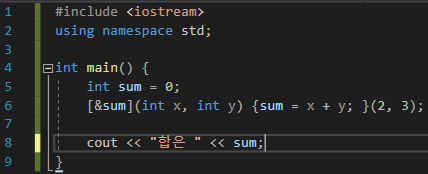
auto로 람다식 저장 및 호출 람다식의 형식은 컴파일러에게만 알려져 있음, 실행기에는 알려져 있지 않음 🡪 저장 불가

auto를 통해 람다식 저장 변수 선언 가능

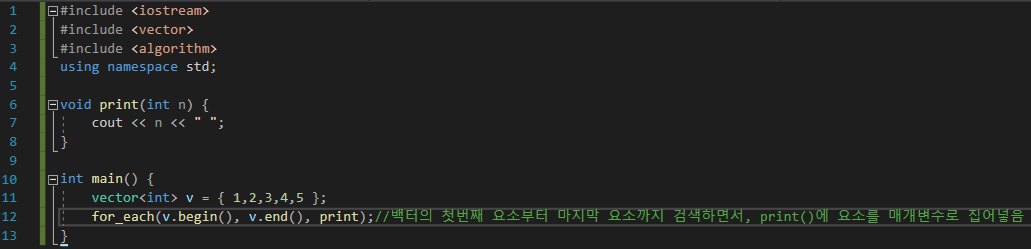
예제10-16 

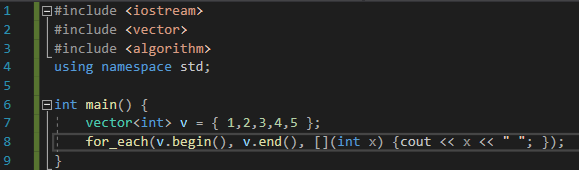
캡쳐리스트와 리턴타입을 가지는 람다식 만들기 캡쳐리스트 이용 🡪 주변의 non-static변수의 값을 복사, 참조가능

예제10-17 

예제10-18 

STL템플릿에 람다식 활용 STL템플릿에 람다식 매우 유용

예제10-19 

예제10-20 

예제10-20이 더 깔끔함

C++에서람다식의 의미 간단하고 짧게 최적화된 코드 작성

1. 한 번만 호출하고 재사용하지 않기 때문에 함수에 이름을 붙일 필요가 없는 경우
2. STL 알고리즘 함수의 매개변수에 연산코드를 넘기는 경우