

C++ 표준 템플릿 라이브러리, STL

1

- STL(Standard Template Library)
 - ▣ 표준 템플릿 라이브러리
 - C++ 표준 라이브러리 중 하나
 - ▣ 많은 제네릭 클래스와 제네릭 함수 포함
 - 개발자는 이들을 이용하여 쉽게 응용 프로그램 작성
- STL의 구성
 - ▣ 컨테이너 – 템플릿 클래스
 - 데이터를 담아두는 자료 구조를 표현한 클래스
 - 리스트, 큐, 스택, 맵, 벡터 등
 - ▣ iterator – 컨테이너 원소에 대한 포인터
 - 컨테이너의 원소들을 순회하면서 접근하기 위해 만들어진 컨테이너 원소에 대한 포인터
 - ▣ 알고리즘 – 템플릿 함수
 - 컨테이너 원소에 대한 복사, 검색, 삭제, 정렬 등의 기능을 구현한 템플릿 함수
 - 컨테이너의 멤버 함수 아님

〈표 10-1〉 STL 컨테이너의 종류

컨테이너 클래스	설명	헤더 파일
vector	가변 크기의 배열을 일반화한 클래스	<vector>
deque	앞뒤 모두 입력 가능한 큐 클래스	<deque>
list	빠른 삽입/삭제 가능한 리스트 클래스	<list>
set	정렬된 순서로 값을 저장하는 집합 클래스, 값은 유일	<set>
map	(key, value) 쌍을 저장하는 맵 클래스	<map>
stack	스택을 일반화한 클래스	<stack>
queue	큐를 일반화한 클래스	<queue>

〈표 10-2〉 STL iterator의 종류

iterator의 종류	iterator에 ++ 연산 후 방향	read/write
iterator	다음 원소로 전진	read/write
const_iterator	다음 원소로 전진	read
reverse_iterator	지난 원소로 후진	read/write
const_reverse_iterator	지난 원소로 후진	read

〈표 10-3〉 STL 알고리즘 함수들

copy	merge	random	rotate
equal	min	remove	search
find	move	replace	sort
max	partition	reverse	swap

STL과 관련된 헤더 파일과 이름 공간

3

□ 헤더파일

▣ 컨테이너 클래스를 사용하기 위한 헤더 파일

- 해당 클래스가 선언된 헤더 파일 include

예) vector 클래스를 사용하려면 `#include <vector>`

list 클래스를 사용하려면 `#include <list>`

▣ 알고리즘 함수를 사용하기 위한 헤더 파일

- 알고리즘 함수에 상관 없이 `#include <algorithm>`

□ 이름 공간

- ▣ STL이 선언된 이름 공간은 `std`

Stack

4

□ 특징

스택(Stack)은 대표적인 **LIFO(Last In First Out)** 구조이다. 따라서 가장 마지막에 넣은 데이터가 가장 먼저 사용된다. 스택의 기본함수에는 push, pop, empty, top, swap 등이 있다.

□ `#include <stack>` //스택 헤더 파일

```
#include <stack>
int main()
{
    stack<int> myStack;

    for(int i = 0 ; i < 5;i++)
    {
        myStack.push(i+10);
    }

    cout<<myStack.top()<<endl; //top 요소 출력
    myStack.pop(); //top에 있는 요소 제거
    cout<<"stack size: "<< myStack.size() << endl<<endl;
}
```

Queue

5

□ 특징

큐(Queue)은 대표적인 **FIFO(First In First Out)** 구조이다. 따라서 가장 먼저 넣은 데이터가 처음부터 시용된다. 큐의 기본함수에는 push, pop, empty, front, back, swap 등이 있다.

#include <queue> //큐 헤더 파일

```
#include<queue>

int main()
{
    queue<char> myQueue;

    myQueue.push( 'A' );
    myQueue.push( 'B' );
    myQueue.push( 'C' );

    cout << "Queue Size : " << myQueue.size() << endl;

    cout << "myQueue Items" << endl;
    while( ! myQueue.empty() )
    {
        cout << myQueue.front() << endl;
        myQueue.pop();
    }
}
```

deque

6

□ 특징

덱 자료구조는 Queue와 다른 점으로 삽입과 삭제를 한쪽이 아닌 앞, 뒤 양쪽에서 할 수 있다는 것이다. 일종의 Queue와 같다. 따라서 Deque는 Stack과 Queue의 장점을 모은 것으로 FIFO 방식과 LIFO 방식 둘 다 사용할 수 있다

```
#include <deque>
```

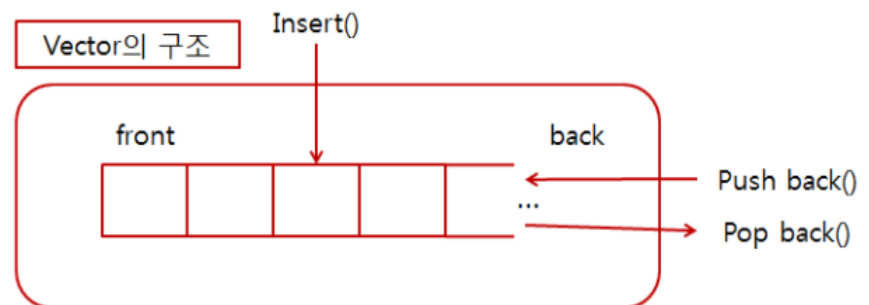


Vector

7

- **Vector(Queue)**은 **동적 배열** 구조를 C++로 구현한 것으로 **마지막에 삽입 및 삭제**가 일어나는 구조이다. 자동으로 메모리가 할당되는 배열이다
- 일반 배열과 차이점이라면 동적으로 크기가 변하고 메모리가 연속적이기 때문에 자동으로 배열의 크기를 조절할 수 있고 유연하게 객체의 추가 및 삭제가 가능하다는 점이다.
- 그러나 중간 데이터를 삭제하고 싶은 경우 Vector의 **erase** 함수를 통해 삭제할 수 있지만, 삭제가 빈번히 일어나는 경우 Vector 구조보다 링크드리스트를 쓰는 것이 효율적이다.
- 배열기반이기 때문에 임의의 원소에 접근이 가능하다.

□ **#include <vector>**



vector 클래스의 주요 멤버와 연산자

8

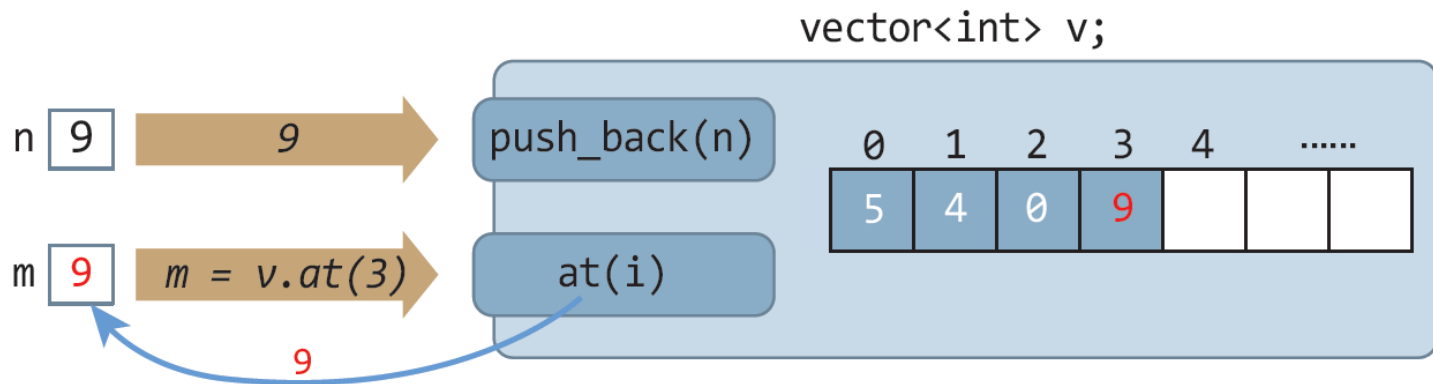
멤버와 연산자 함수	설명
<code>push_back(element)</code>	벡터의 마지막에 <code>element</code> 추가
<code>at(int index)</code>	<code>index</code> 위치의 원소에 대한 참조 리턴
<code>begin()</code>	벡터의 첫 번째 원소에 대한 참조 리턴
<code>end()</code>	벡터의 끝(마지막 원소 다음)을 가리키는 참조 리턴
<code>empty()</code>	벡터가 비어 있으면 <code>true</code> 리턴
<code>erase(iterator it)</code>	벡터에서 <code>it</code> 가 가리키는 원소 삭제. 삭제 후 자동으로 벡터 조절
<code>insert(iterator it, element)</code>	벡터 내 <code>it</code> 위치에 <code>element</code> 삽입
<code>size()</code>	벡터에 들어 있는 원소의 개수 리턴
<code>operator[]()</code>	지정된 원소에 대한 참조 리턴
<code>operator=()</code>	이 벡터를 다른 벡터에 치환(복사)

vector 컨테이너

9

□ 특징

- 가변 길이 배열을 구현한 제네릭 클래스
 - 개발자가 벡터의 길이에 대한 고민할 필요 없음
- 원소의 저장, 삭제, 검색 등 다양한 멤버 함수 지원
- 벡터에 저장된 원소는 인덱스로 접근 가능
 - 인덱스는 0부터 시작



vector 다루기 사례

vector 생성

```
vector<int> v;
```

정수 벡터
생성

vector<int> v



정수 원소 삽입

```
v.push_back(1);  
v.push_back(2);  
v.push_back(3);
```

정수 삽입

v



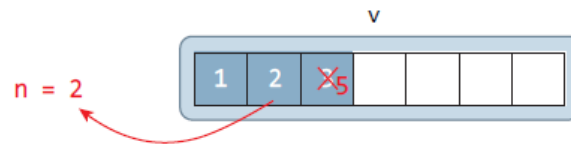
원소 개수 s
벡터의 용량 c

```
int s = v.size(); // s는 3  
int c = v.capacity(); // c는 7
```

s = 3
c = 7

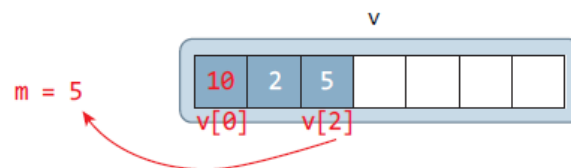
원소 값 접근

```
v.at(2) = 5;  
int n = v.at(1);
```



원소 값 접근

```
v[0] = 10;  
int m = v[2]; // m은 5
```



iterator 사용

11

□ iterator란?

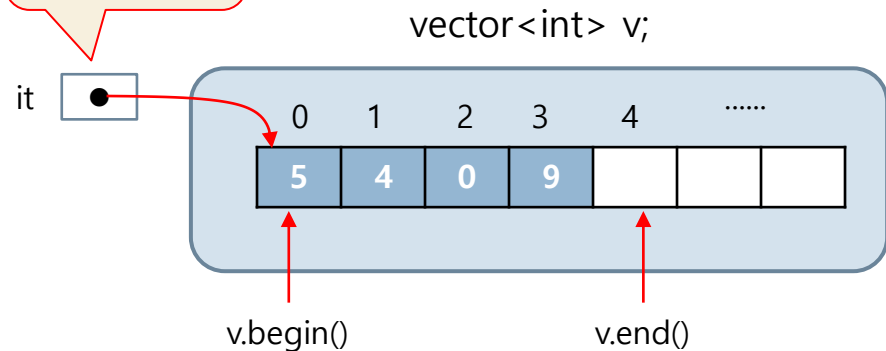
- 반복자라고도 부름
- 컨테이너의 원소를 가리키는 포인터

□ iterator 변수 선언

- 구체적인 컨테이너를 지정하여 반복자 변수 생성

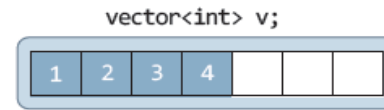
```
vector<int>::iterator it;  
it = v.begin();
```

it는 원소가 int
타입인 벡터의
원소에 대한 포
인터



벡터 생성

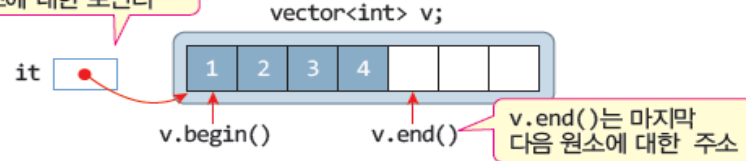
```
vector<int> v;  
for(int i=1; i<=4; i++)  
    v.push_back(i);
```



iterator 변수 선언
및 초기화

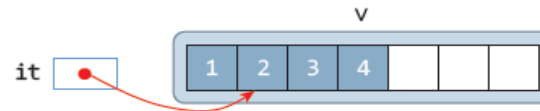
```
vector<int>::iterator it;  
it = v.begin();
```

it는 int 타입 벡터의
원소에 대한 포인터



iterator 증가

```
it++;
```



원소 읽기

```
int n = *it;
```

n = 2

원소 쓰기

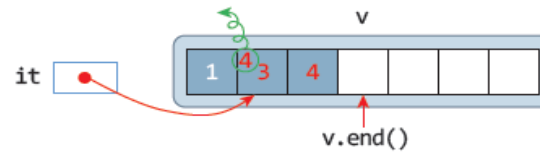
```
n = n*2;  
*it = n;
```



원소 삭제

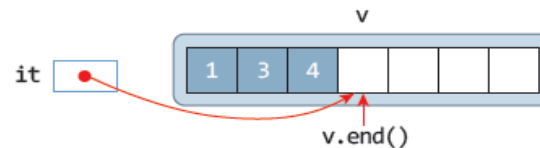
```
it = v.erase(it);
```

v.erase(it)는 it가 가리키는 원소를
삭제한 후 다음 원소에 대한 포인터 리턴



끝으로 옮기기

```
it = v.end();
```



Sort 알고리즘

13

□ 특징

- ▣ sort 함수는 C++ STL에서 제공하는 함수로 정렬에 이용.
- ▣ # include <algorithm> 헤더사용

□ 함수인자 사용법

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

첫 번째 인자: iterator(or 포인터)에서 정렬을 시작 할 부분

두 번째 인자: iterator(or 포인터)에서 정렬을 마칠 부분

이 때 중요한 점은 [begin,end) 범위로 정렬을 한다는 것이다. (begin <= 범위 <end)

Sort 알고리즘

14

□ 특징

- ▣ sort 함수는 세 번째 인자를 사용자자 정의한 함수를 호출할 수 있다.

□ 함수인자 사용법

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

첫 번째 인자, 두 번째 인자 이전과 동일

세 번째 인자 : 정렬을 어떤 식으로 할 것인지 알려주는 함수

이 세 번째 인자를 compare 함수라고 많이들 부르던데, 이 부분에는 **bool형 함수나 혹은 수식**(수식도 0또는 1이 나오므로)을 넣어 줄 수 있다.

함수를 사용하는 경우, 두 개의 매개 변수를 받도록 선언하면 sort 함수에서 자동으로 값을 할당한다.

Vector assign 메서드

15

- 벡터에 새로운 내용을 집어 넣는다. 벡터에 이전에 있었던 원소들은 모두 삭제하고, 인자로 받은 새로운 내용을 집어 대입된다.
- 형식 `assign (원소의 개수, 값);`
- `vector<int> v;`
- `v.assign(5, 0);` // 벡터 `v` 에 0을 5개 대입한다
- `v.assign(배열변수, 배열변수+5);` 배열로 부터 5개 인수

Vector pair

16

- Pair 는 두 개의 객체(first, second) 를 하나로 묶어주는 역할을 하는 struct로 데이터의 쌍을 표현할 때 사용된다. 주로 vector와 묶어 이용된다.
- 문법] pair <[type first],[type second]> p1;
- 사용할 데이터 타입 first, second 를 넣고, pair의 이름을 선언한다.
- make_pair(value1, value2); //두 값을 가진 pair 생성
- Pair는 초기에 따로 초기화 하지 않으면, make_pair 멤버함수를 사용해 원소를 할당한다.
- 인자 값 사용
p.first, p.second 를 사용하여 두 인자를 구분하며, 인자의 값을 반환.

Vector pair, tuple - 2쌍, 3쌍 값 묶기

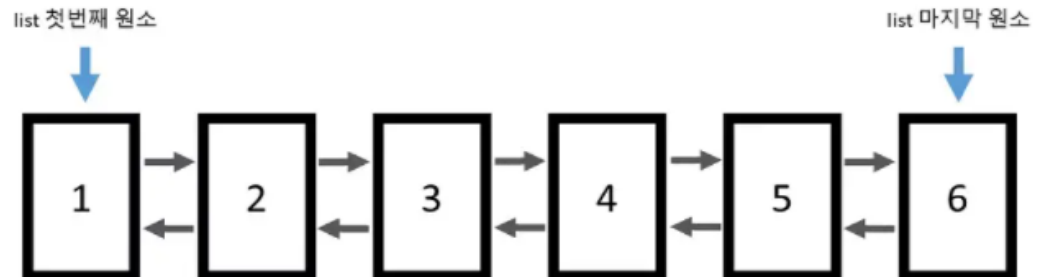
17

- 서로 연관이 있는 데이터끼리 처리하기 쉽고 직관적이게 알 수 있게 값을 묶을 수 있다.
- c++에서는 **pair**과 **tuple**을 지원한다.
- **2쌍의 값**을 묶고 싶다면 **pair**
- **3쌍의 값**을 묶고 싶다면 **tuple** 헤더 : **#include <tuple>**
- - tuple은 값을 읽을 시 **get**을 사용
 get<튜플에서 읽어올 값 인덱스>(튜플객체변수)
 '<>'의 들어가는 인덱스 값은 튜플 이므로 3개 라면, 0~2
 까지 이다.

STL List

□ list 클래스의 탐색 (연결리스트)

- 리스트는 double linked list로 구현되어 있다.
- 노드가 양 쪽으로 모두 연결 되어 있으며 삽입/삭제가 자주 발생하는 경우에 용이하다.
- 리스트(list) 의 경우 양방향 연결 구조로 임의의 위치에 있는 원소에 접근을 바로 할 수 없다. list 자체에서는 시작 원소와 마지막 원소의 위치만을 기억하기 때문에, 임의의 위치에 있는 원소에 접근하기 위해서는 하나씩 링크를 따라가야 한다.



STL List

□ 추가 및 삭제

- `push_front(element)` : 리스트 제일 앞에 원소 추가
- `pop_front()` : 리스트 제일 앞에 원소 삭제
- `push_back(element)` : 리스트 제일 뒤에 원소 추가
- `pop_back()` : 리스트 제일 뒤에 원소 삭제
- `insert(iterator, element)` : iterator가 가리키는 부분 "앞"에 원소를 추가
- `erase(iterator)` : iterator가 가리키는 부분에 원소를 삭제

□ 조회

- `*iterator` : iterator가 가리키는 원소에 접근
- `front()` : 첫번째 원소를 반환
- `back()` : 마지막 원소를 반환

□ 기타

- `empty()` : 리스트가 비어있으면 true 아니면 false를 반환
- `size()` : 리스트 원소들의 수를 반환

map 컨테이너

20

□ 특징

- ('키', '값')의 쌍을 원소로 저장하는 제네릭 컨테이너
 - 동일한 '키'를 가진 원소가 중복 저장되면 오류 발생
- '키'로 '값' 검색
- #include <map> 필요
- map <key, value> map1; //MAP 기본 형태

□ 맵 컨테이너 생성 예

- 영한 사전을 저장하기 위한 맵 컨테이너 생성 및 활용
 - 영어 단어와 한글 단어를 쌍으로 저장하고, 단어로 검색

```
// 맵 생성
Map<string, string> dic;           // 키는 영어 단어, 값은 한글 단어

// 원소 저장
dic.insert(make_pair("love", "사랑")); // ("love", "사랑") 저장
dic["love"] = "사랑";                 // ("love", "사랑") 저장

// 원소 검색
string kor = dic["love"];             // kor은 "사랑"
string kor = dic.at("love");          // kor은 "사랑"
```

map 클래스의 주요 멤버와 연산자

21

멤버와 연산자 함수	설명
<code>insert(pair<> &element)</code>	맵에 '키'와 '값'으로 구성된 pair 객체 element 삽입
<code>at(key_type& key)</code>	맵에서 '키' 값에 해당하는 '값' 리턴
<code>begin()</code>	맵의 첫 번째 원소에 대한 참조 리턴
<code>end()</code>	맵의 끝(마지막 원소 다음)을 가리키는 참조 리턴
<code>empty()</code>	맵이 비어 있으면 true 리턴
<code>find(key_type& key)</code>	맵에서 '키' 값에 해당하는 원소를 가리키는 iterator 리턴
<code>erase(iterator it)</code>	맵에서 it가 가리키는 원소 삭제
<code>size()</code>	맵에 들어 있는 원소의 개수 리턴
<code>operator[key_type& key]()</code>	맵에서 '키' 값에 해당하는 원소를 찾아 '값' 리턴
<code>operator=()</code>	맵 치환(복사)

array 컨테이너

22

- 고정길이의 배열을 표현할 때 사용 하는 C++11에 추가된 array container 이다. (std::array), 고정배열 처럼 배열의 크기는 컴파일 타임에 설정.
- 기존 배열과 마찬가지로 stack에 저장이 된다. 다른 container (vector, deque ...)들과 마찬가지로 여러 유용한 멤버 함수를 사용할 수 있다.
- 임의접근 가능([], at())
- array 컨테이너 배열이 함수에 전달될 때 포인터로 형 변환 하지 않으므로 배열 길이정보를 사용할 수 있다.

```
#include <array>
```

auto를 이용하여 쉬운 변수 선언

23

□ C++에서 auto

▣ 기능

- C++ 11부터 auto 선언의 의미 수정 : 컴파일러에게 변수선언문에서 추론하여 타입을 자동 선언하도록 지시
- C++ 11 이전까지는 스택에 할당되는 지역 변수를 선언하는 키워드

▣ 장점

- 복잡한 변수 선언을 간소하게, 긴 타입 선언 시 오타 줄임

□ auto의 기본 사용 사례

```
auto pi = 3.14;    // 3.14가 실수이므로 pi는 double 타입으로 선언됨
auto n = 3;        // 3이 정수이므로 n을 int 타입으로
auto *p = &n;      // 변수 p는 int* 타입으로 추론
```

```
int n = 10;
int &ref = n;      // ref는 int에 대한 참조 변수
auto ref2 = ref;  // ref2는 int& 변수로 자동 선언
```

auto의 다른 활용 사례

24

□ 다른 활용 사례

- ▣ 함수의 리턴 타입으로부터 추론하여 변수 타입 선언

```
int square(int x) { return x*x; }  
...  
auto ret = square(3); // 변수 ret는 int 타입으로 추론
```

- ▣ STL 템플릿에 활용

- vector<int>iterator 타입의 변수 it를 auto를 이용하여 간단히 선언

```
vector<int>::iterator it;
```

```
for (it = v.begin(); it != v.end(); it++)  
    cout << *it << endl;
```



```
for (auto it = v.begin(); it != v.end(); it++)  
    cout << *it << endl;
```


람다

25

□ C++ 람다

- ▣ 익명의 함수 만드는 기능으로 C++11에서 도입
 - 람다식, 람다 함수로도 불림
 - `lambda`는 람다 표현식 또는 람다 함수, 익명 함수(anonymous function)로 불리며, 성질은 함수 객체(functor)와 동일하다.
 - 익명 함수로 몸통은 있지만 이름이 없는 함수이다.

□ 람다문법

`[] () { } ()`

`[캡처] (매개변수) { 함수 동작 } (호출인자)`

□ 예]

```
[] (int a, int b) { cout << a + b << endl; } (10, 20) //람다함수
```

□ 람다식

람다 함수를 호출시 전달 값

C++에서 람다식 선언

26

□ C++의 람다식의 구성

▣ 4 부분으로 구성

- 캡처 리스트 : 람다식에서 사용하고자 하는 함수 바깥의 변수 목록
- 매개변수 리스트 : 보통 함수의 매개변수 리스트와 동일
- 리턴 타입
- 함수 바디 : 람다식의 함수 코드

[] () -> 리턴타입 { /* 함수 코드 작성 */ };

람다식의 기본 구조

```
[ ](int x, int y) { cout << x + y; }; // 매개변수 x, y의 합을 출력하는 람다 작성  
[ ](int x, int y) -> int { return x + y; }; // 매개변수 x, y의 합을 리턴하는 람다 작성  
[ ](int x, int y) { cout << x + y; } (2, 3); // x에 2, y에 3을 대입하여 코드 실행. 5 출력
```

람다식 작성 및 호출 사례

간단한 람다식 만들기

27

매개변수 x, y의 합을 출력하는 람다식 만들기

매개변수 x, y의 합을 출력하는 람다식은 다음과 같이 작성

```
[](int x, int y) { cout << x + y; }; // x, y의 합을 출력하는 람다식
```

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

int main() {
    int sum = 0; // 지역 변수

    // 람다 함수 선언과 동시에 호출(x=2, y=3 전달)
    [](int x, int y) { cout << "합은 " << x + y; } (2, 3); // 5 출력

    //캡처리스트
    [&sum](int x, int y) { sum = x + y; } (2, 3); // 합 5를 지역변수 sum에 저장

    cout << "합은 " << sum;

}
```