

템플릿과 STL(Standard Template Library)

템플릿과 STL

- 이번 강의에서는 템플릿과 STL에 대하여 학습합니다.
- 템플릿은 함수나 클래스 코드를 찍어내듯이 생산할 수 있도록 일반화(generic)시키는 도구입니다.
 - 코드는 같지만 자료형이 다른 함수를 자동으로 생성해주는 함수 템플릿
 - 데이터 멤버의 자료형만 다른 클래스를 생성하는 클래스 템플릿
- STL(Standard Template Library)은 템플릿으로 작성된 템플릿 클래스와 함수 라이브러리입니다
- 시작하겠습니다

학습 목차

- 일반화와 템플릿
- 템플릿 함수 만들기
- 템플릿 클래스 만들기
- STL 활용
- 람다식

학습 목표

- 템플릿 문법을 이해한다
- 함수 템플릿과 클래스 템플릿을 이해하고 구현할 수 있다
- 템플릿 구체화 과정을 이해한다
- 템플릿을 활용한 응용 프로그램을 구현할 수 있다
- STL 컨테이너를 이해하고 활용할 수 있다

일반화와 템플릿

- 함수 중복의 약점 - 코드 중복

//myswap()는 매개변수만 다르고 나머지 코드는 동일, 동일한 코드 중복

```
void myswap(int& a, int& b) {
```

```
    int tmp;
```

```
    tmp = a; a = b; b = tmp;
```

```
}
```

```
void myswap(double & a, double & b) {
```

```
    double tmp;
```

```
    tmp = a; a = b; b = tmp;
```

```
}
```

```
int main() {
```

```
    int a=4, b=5;
```

```
    myswap(a, b); // myswap(int& a, int& b) 호출
```

```
    cout << a << 'Wt' << b << endl;
```

```
    double c=0.3, d=12.5;
```

```
    myswap(c, d); // myswap(double& a, double& b) 호출
```

```
    cout << c << 'Wt' << d << endl;
```

```
}
```

일반화와 템플릿

- 제네릭(generic) 또는 일반화
 - 함수나 클래스를 일반화시키고, 매개 변수 타입을 지정하여 틀에서 찍어 내듯이 함수나 클래스 코드를 생산하는 기법
- 템플릿
 - 함수나 클래스를 일반화하는 C++ 도구
 - template 키워드로 함수나 클래스 선언
 - 변수나 매개 변수의 타입만 다르고, 코드 부분이 동일한 함수를 일반화시킴
 - 제네릭 타입 - 일반화를 위한 데이터 타입

일반화와 템플릿

- 템플릿 선언

template <class T> 또는
template <typename T> //표준 C++ 라이브러리들은 내부적으로 typename 사용

3 개의 제네릭 타입을 가진 템플릿 선언

template <typename T1, typename T2, typename T3>

템플릿을 선언하는
키워드

제네릭 타입을
선언하는 키워드

제네릭 타입 T 선언

```
template < typename T>
void myswap (T & a, T & b) {
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

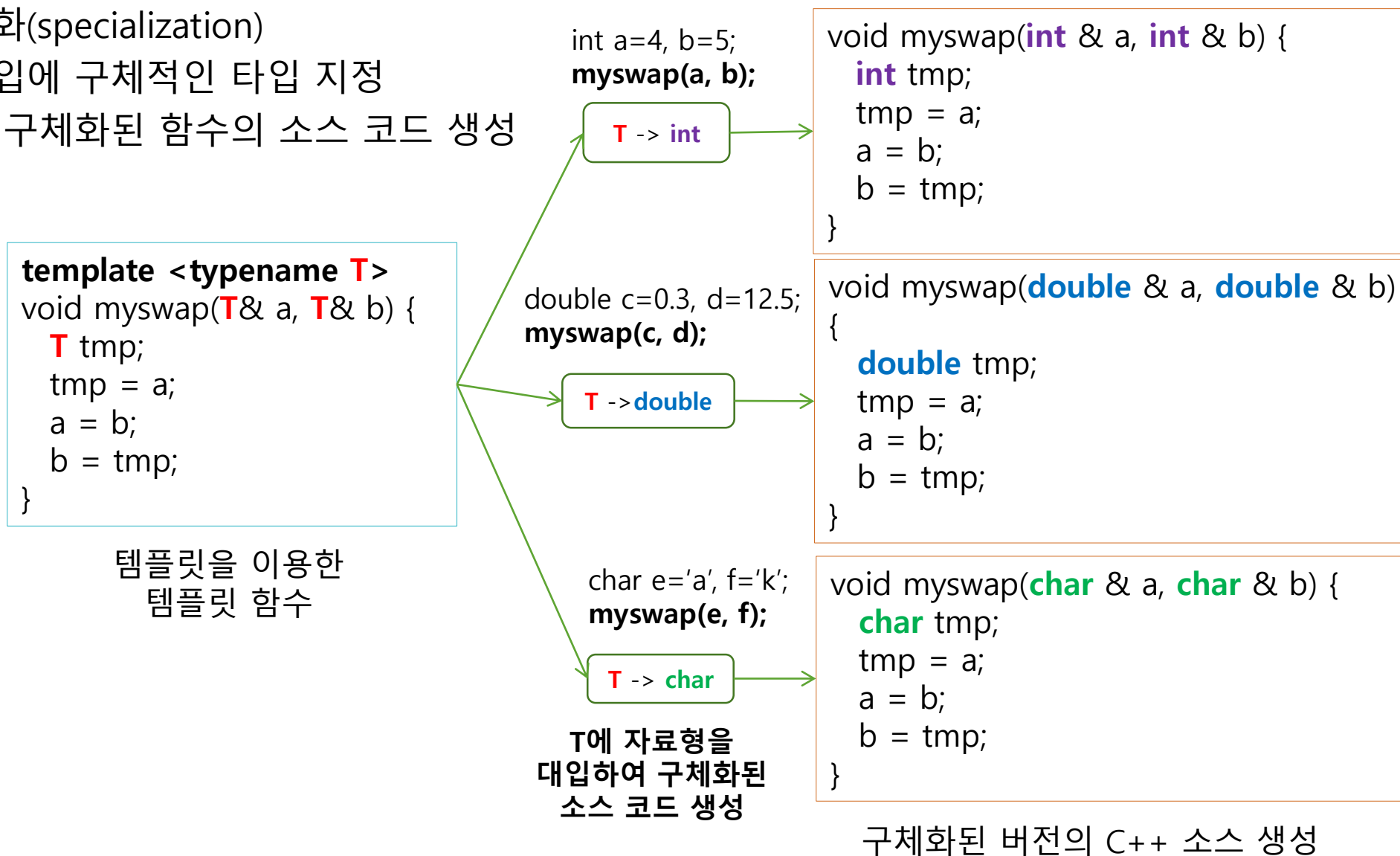
```
template <class T> void myswap (T & a, T & b){
```

```
template <typename T> void myswap (T & a, T & b){
```

템플릿을 이용한 함수 myswap

일반화와 템플릿

- 템플릿으로 부터의 구체화(specialization)
 - 템플릿의 제네릭 타입에 구체적인 타입 지정
 - 템플릿 함수로 부터 구체화된 함수의 소스 코드 생성



일반화와 템플릿

- 구체화 오류
 - 제네릭 타입에 구체적인 타입 지정 시 주의

```
//두 매개변수 a, b의 제네릭 타입 동일  
template <typename T>  
void myswap(T & a, T & b)
```

```
int s=4;  
double t=5;  
myswap( s, t ); //두 개의 타입이 서로 다름
```

컴파일 오류. 템플릿으로부터
myswap(int &, double &) 함수를 구체화할 수 없다.

일반화와 템플릿

- 템플릿 장점
 - 함수 코드의 재사용
 - 높은 소프트웨어의 생산성과 유용성
- 템플릿 단점
 - 포팅에 취약 : 컴파일러에 따라 지원하지 않을 수 있음
 - 컴파일 오류 메시지 빈약, 디버깅에 많은 어려움
- 제네릭 프로그래밍
 - generic programming
 - 일반화 프로그래밍이라고도 부름
 - 제네릭 함수나 제네릭 클래스를 활용하는 프로그래밍 기법
 - C++에서 STL(Standard Template Library) 제공
 - 보편화 추세
 - Java, C# 등 많은 언어에서 활용

템플릿 함수

- 데이터를 교환하는 함수 템플릿 myswap()

```
class Circle {
    int radius;
public:
    Circle(int radius=1) { this->radius = radius; }
    int getRadius() { return radius; }
};
```

```
template <typename T>
void myswap(T & a, T & b) {
    T tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

```
int main() {
    int a=4, b=5;
    myswap(a, b);
    cout << "a=" << a << ", " << "b=" << b << endl;

    double c=0.3, d=12.5;
    myswap(c, d);
    cout << "c=" << c << ", " << "d=" << d << endl;

    Circle donut(5), pizza(20);
    myswap(donut, pizza);
    cout << "donut반지름=" << donut.getRadius() << ", ";
    cout << "pizza반지름=" << pizza.getRadius() << endl;
}
```

```
a=5, b=4
c=12.5, d=0.3
donut반지름=20, pizza반지름=5
```

템플릿 함수

- 명시적 자료형 결정

```
template <typename T>
T smaller (T first, T second) {
    if (first < second) {
        return first;
    }
    return second;
}
```

cout << smaller(23, 67.2); //error, 인수는 동일한 자료형이어야 함

//명시적으로 자료형을 지정하여 해결

cout << **smaller<double>**(23, 67.2); // 23이 double형으로 변환

- 특수화(Specialization)

- c string에는 < 연산자 적용 불가
 - smaller 템플릿 함수 사용할 수 없음
- 템플릿 특수화를 사용하여 해결
 - 특정 자료형으로 함수를 정의하여 활용
 - 함수 앞에 template<> 사용
 - 원본 템플릿 함수를 참조할 수 있어야 한다

템플릿 함수

- 특수화

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

template <typename T>
T smaller(const T &first, const T &second){
    if (first < second){
        return first;
    }
    return second;
}
```

//원본 템플릿을 반드시 참조할 수 있어야 한다 - 함수 템플릿 특수화

```
template <>
const char *smaller<const char *>(const char * const &first, const char *const &second){
    if (strcmp(first, second) < 0) {
        return first;
    }
    return second;
}
```

```
int main(){
    const char *s1 = "Hallym";
    const char *s2 = "Software";
    string str1 = "Hello";
    string str2 = "Hi";

    cout << "Smaller (Hello , Hi): " << smaller (str1, str2) << endl;
    cout << "Smaller (Hallym, Software)" << smaller(s1, s2) << endl;
    // 또는 smaller<const char *>(s1, s2)
    return 0;
}
```

템플릿 함수

- 함수 템플릿 변형
 - 자료형이 아닌 템플릿 매개변수
 - 함수 템플릿으로 자료형이 아닌 값을 정의할 수도 있음
 - 이렇게 값으로 정의해서 사용할 경우, 매개변수를 전달하는 것과 비슷한 역할
 - 자료형이 아니라 값을 정의하는 것이므로 자료형은 고정

```
#include <iostream>
using namespace std;
//함수 템플릿 변형 - 템플릿 매개변수
template <typename T, int size>
decltype(auto) add(T const(&data)[size]) { //&data : T 타입의 배열에 대한 참조, T 타입의 원소 size개를 갖는다
    T sum = 0;
    for (int i = 0; i < size; i++) {
        sum += data[i];
    }
    return sum; //컴파일러가 함수의 리턴 타입을 자동으로 추론
}
```

- 템플릿 함수 반환 타입
 - 다음과 같이 반환 타입을 auto로 할 수 있음
auto add(T const(&data)[size])
 - auto로 하면 레퍼런스와 const가 사라져 복사 연산 수행
 - decltype(auto)으로 하면 const와 레퍼런스 지시자 그대로 존재

```
int main() {
    int x[5] = { 1,2,3,4,5 };
    double d[6] = { 1.2, 2.3, 3.4, 4.5, 5.6, 6.7 };
```

```
// 함수 템플릿을 호출할 때 컴파일러는 T의 구체적인 값 뿐만 아니라 size 값 추론
cout << "sum of x[] = " << add(x) << endl; //또는 add<int>(x);
cout << "sum of d[] = " << add(d) << endl;
}
```

템플릿 함수 오버로딩

- 이름이 같지만 시그니처가 다른 여러 개의 함수
- 일반적으로 템플릿 자료형은 같지만 매개변수의 수가 다른 함수를 만들어낼 때 활용

```
template <typename T>
T smallest(const T& first, const T& second){
    if (first < second){ return first; }
    return second;
}
```

```
template <typename T>
T smallest(const T& first, const T& second, const T& third){
    return smallest(smallest(first, second), third);
}
```

```
int main(){
    cout << "17, 12 에서 작은값 : ";    cout << smallest(17, 12) << endl;
    cout << "8.5, 4.1, 19.75 에서 작은값 : "; cout << smallest(8.5, 4.1, 19.75) << endl;
    return 0;
}
```

템플릿 함수 보다 중복 함수 우선

1	2	3	4	5	
1.1	2.2	3.3	4.4	5.5	6.6
1	2	3	4	5	

템플릿 함수와
중복된 print() 함수

중복된 print() 함수가
우선 바인딩

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

```
template <typename T>
void print(T array [], int n=5) {
    for(int i=0; i<n; i++)
        cout << array[i] << 'Wt';
    cout << endl;
}
```

```
void print(char array [], int n) { // char 배열을 출력하기 위한 함수 중복
    for(int i=0; i<n; i++)
        cout << (int)array[i] << 'Wt'; // array[i]를 int 타입으로 변환하여 정수 출력
    cout << endl;
}
```

```
int main() {
    int x[] = {1,2,3,4,5};
    double d[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
    print(x);
    print(d, 6);
```

```
    char c[5] = {1,2,3,4,5};
    print(c, 5); //print<char>(c, 5); 단, 제네릭 타입을 명시하면 템플릿 함수가 먼저 실행
}
```

템플릿 print()
함수로부터 구체화

print(c, 5); //print<char>(c, 5); 단, 제네릭 타입을 명시하면 템플릿 함수가 먼저 실행

템플릿 클래스

- 멤버 변수 타입, 멤버 함수의 매개변수 또는 리턴 타입을 템플릿 매개변수 받음
- 클래스 템플릿 선언과 구현 모두 템플릿 매개변수가 있어야 함
- 템플릿 클래스 사용 시 제네릭 타입에 적용할 구체적인 타입 지정 해야 함

```
template <typename T> //템플릿 클래스 선언
class Stack{
private:
    T* ptr;
    int capacity;
    int size;
public:
    Stack(int capacity);
    ~Stack();
    void push(const T& element);
    T pop();
};
```

```
template <typename T> //템플릿 클래스 구현
void Stack<T>::push(const T& element){
    if (size < capacity){
        ptr[size] = element;
        size++;
    }
    else{
        cout << "stack is full." << endl;
        return;
    }
}
```

```
//템플릿 클래스 구체화 & 활용
Stack<int> stack(10); //구체적인 타입 지정
stack.push(5);
Stack<char> cstack(20);
cstack.push('a');
```

템플릿 클래스

```
//stack 템플릿 클래스 선언과 구현
template <typename T> //템플릿 클래스 선언
class Stack{
private:
    T* ptr;
    int capacity;
    int size;
public:
    Stack(int capacity);
    ~Stack();
    void push(const T& element);
    T pop();
};

template <typename T> //제네릭 클래스 구현
Stack<T> ::Stack(int cap): capacity(cap), size(0){
    ptr = new T[capacity];
}

template <typename T>
Stack <T> :: ~Stack() { delete[] ptr; }
```

```
template <typename T>
T Stack <T> ::pop(){
    if (size > 0){
        T temp = ptr[size - 1];
        size--;
        return temp;
    }
    else{
        cout << "stack is empty." << endl;
        return 0;
    }
}
```

```
int main(){
    Stack<int> stack(10);
    stack.push(5);
    stack.push(6);
    stack.push(7);
    stack.push(3);
    cout << stack.pop() << endl;
    cout << stack.pop();
    return 0;
}
```

템플릿 클래스

두 개의 제네릭 타입을 갖는 템플릿 클래스

// 두 개의 제네릭 타입 선언

```
template <typename T1, typename T2>
```

```
class GClass {
```

```
    T1 data1;
```

```
    T2 data2;
```

```
public:
```

```
    GClass(T1 a, T2 b);
```

```
    void set(T1 a, T2 b);
```

```
    void get(T1 &a, T2 &b);
```

```
};
```

```
template <typename T1, typename T2>
```

```
GClass<T1, T2>::GClass(T1 a, T2 b) : data1(a), data2(b) {}
```

```
template <typename T1, typename T2>
```

```
void GClass<T1, T2>::set(T1 a, T2 b) { data1 = a; data2 = b; }
```

```
template <typename T1, typename T2>
```

```
void GClass<T1, T2>::get(T1 &a, T2 &b) { a = data1; b = data2; }
```

```
int main() {
    int a; double b;
    GClass<int, double> x(30, 3.5);
    x.get(a, b);
    cout << "a=" << a << 'Wt' << "b=" << b << endl;
    x.set(2, 0.5);

    char c; float d;
    GClass<char, float> y('c', 3.4);
    y.get(c, d);
    cout << "c=" << c << 'Wt' << "d=" << d << endl;

    y.set('m', 12.5);
    y.get(c, d);
    cout << "c=" << c << 'Wt' << "d=" << d << endl;
}
```

템플릿 클래스

- 비 타입 매개변수 - 템플릿 매개변수에 변수 선언

```
template <typename T, int capacity=10> //템플릿 매개변수에 int 선언, default값도 가능
```

```
class Stack{
private:
    T *ptr;
    int size;
public:
    .....
};
```

```
template <typename T, int capacity>
Stack<T, capacity>::Stack() {
    size=0;
    ptr = new T[capacity];
}
```

```
template <typename T, int capacity>
void Stack<T, capacity>::push(const T &element){ ..... }
```

```
template <typename T, int capacity>
T Stack<T, capacity>::pop() { }
```

```
int main()
```

```
{
    Stack<int, 15> istack;
    Stack<double, 20> dstack;
    Stack<char> cstack;
```

```
    istack.push(5);
    istack.push(6);
    dstack.push(7.5);
    dstack.push(3.2);
    istack = dstack; //error, 동일한타입이 아님
    cout << istack.pop() << endl;
    cout << dstack.pop();
    return 0;
}
```

템플릿 클래스

- 템플릿 클래스와 static 멤버 변수

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

template<typename T>
class SimpleStatic{
public:
    inline static T mem=0;
    static void AddMem(T num){
        mem += num;
    }
    static void ShowMem(){
        cout<<mem<<endl;
    }
};

//inline을 사용하지 않을 경우
//template<typename T>
//T SimpleStatic<T>::mem=0;

int main(){
    SimpleStatic<int>::AddMem(50);
    SimpleStatic<int>::AddMem(50);
    SimpleStatic<int>::ShowMem();
}
```

템플릿 클래스

- 클래스 템플릿 특수화
 - 특정한 타입에 대하여 템플릿을 다르게 구현할 수 있다

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

template<typename T> //원본 템플릿
class Simple{
    T mem;
public:
    Simple(T value):mem(value){ }
    T getMem(){
        return mem;
    }
};
```

```
template<> //템플릿 특수화
class Simple<int>{
    int mem;
public:
    Simple(int value):mem(value){ }
    int getMem(){
        return mem+mem;
    }
};

int main(){
    Simple<double> s1(45.3); //원본 템플릿 사용
    Simple<int> s2(50); //템플릿 특수화 사용
    cout<<"s1 : "<<s1.getMem()<<endl;
    cout<<"s2 : "<<s2.getMem()<<endl;
}
```

헤더파일과 소스파일의 구분

```
#ifndef __POINT_TEMPLATE_H_
#define __POINT_TEMPLATE_H_

template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0);
    void ShowPosition() const;
};
#endif
```

헤더파일

```
#include <iostream>
#include "PointTemplate.h"
using namespace std;
int main(void)
{
    Point<int> pos1(3, 4);
    pos1.ShowPosition();
    Point<double> pos2(2.4, 3.6);
    pos2.ShowPosition();
    Point<char> pos3('P', 'F');
    pos3.ShowPosition();
    return 0;
}
```

소스파일 1

```
#include <iostream>
#include "PointTemplate.h"
using namespace std;

template <typename T>
Point<T>::Point(T x, T y) : xpos(x), ypos(y) { }

template <typename T>
void Point<T>::ShowPosition() const
{
    cout<< '[' << xpos<< ", " << ypos<< " ]' << endl;
}
```

소스파일 2

기준에 의해서 헤더파일과 소스파일을 잘 구분하였다.
그러나 컴파일 오류가 발생한다! 이유는?

헤더파일과 소스파일의 구분

```
#ifndef _POINT_TEMPLATE_H_
#define _POINT_TEMPLATE_H_

template <typename T>
class Point
{
private:
    T xpos, ypos;
public:
    Point(T x=0, T y=0);
    void ShowPosition() const;
};
#endif
```

헤더파일

```
#include <iostream>
#include "PointTemplate.h"
using namespace std;
int main(void)
{
    Point<int> pos1(3, 4);
    pos1.ShowPosition();
    Point<double> pos2(2.4, 3.6);
    pos2.ShowPosition();
    Point<char> pos3('P', 'F');
    pos3.ShowPosition();
    return 0;
}
```

소스파일 1

위의 소스파일을 컴파일 할 때 **Point<int>**, **Point<double>**, **Point<char>** 템플릿 클래스가 만들어져야 한다. 따라서 **Point** 클래스 템플릿의 정의 부에 대한 정보도 필요로 한다.

```
#include <iostream>
#include "PointTemplate.h"
#include "PointTemplate.cpp"
using namespace std;
int main(void)
{
    . . . .
    return 0;
}
```

해결책 1. 클래스 템플릿의 구현 부를 담고 있는 소스파일을 포함시킨다.

해결책 2. 헤더파일에 클래스 템플릿의 구현 부를 포함시킨다.

C++ 표준 템플릿 라이브러리, STL

- STL(Standard Template Library)
 - 표준 템플릿 라이브러리
 - C++ 표준 라이브러리 중 하나
 - 제네릭 클래스와 제네릭 함수 포함
- STL의 구성
 - 컨테이너 – 템플릿 클래스
 - 데이터를 담아두는 자료 구조를 표현한 클래스
 - 리스트, 큐, 스택, 맵, 셋, 벡터
 - 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

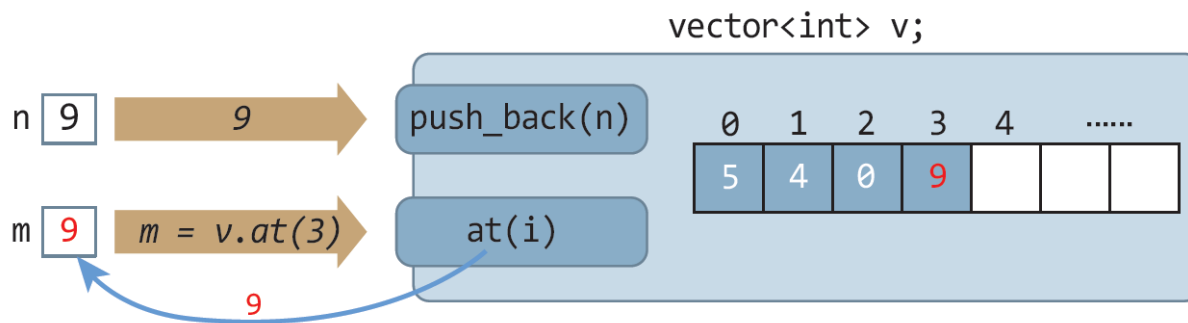
STL과 관련된 헤더 파일과 이름 공간

- 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`

vector 컨테이너

특징

- 가변 길이 배열을 구현한 템플릿 클래스 – 내부 크기 조절
- 연속된 메모리 공간에 원소 저장 – 순차 컨테이너
- 원소의 저장, 삭제, 검색 등 다양한 멤버 함수 지원
- 벡터에 저장된 원소는 인덱스로 접근 가능
 - 인덱스는 0부터 시작



vector 클래스의 주요 멤버와 연산자

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 활용

```
#include <vector>
```

```
using namespace std;
```

```
int main(){
```

```
    //1) 벡터 생성
```

```
    vector<int> v;           // 정수만 저장하는 빈 벡터 생성
```

```
    vector<double> vd(10);   // 실수를 10개 저장할 수 있는 빈 벡터 생성
```

```
    vector<char> vc(5, 'a'); // 'a'로 초기화 되는 크기 5인 벡터 생성
```

```
    vector<int> intVector({1,2,3,4,5,7}); // initializer-list로 초기원소 지정
```

```
    vector<double> doubleVector = {2.3, 4.5, 6.5, 7.6, 8.7, 8.9}; // 유니폼 초기화 사용
```

```
    //2) 벡터 원소 저장, emplace_back() 메소드는 복사나 이동 작업을 수행하지 않음
```

```
    v.emplace_back(10); // v.push_back(10); 벡터에 정수 1 삽입
```

```
    v.emplace_back(20); // v.push_back(20); 벡터에 정수 2 삽입
```

```
    v.emplace_back(30); // v.push_back(30); 벡터에 정수 3 삽입
```

```
    //3) 벡터 원소 값 접근, at() 또는 []
```

```
    v[0] = 1; // 벡터의 첫 번째 원소를 1로 변경
```

```
    v.at(2) = 5; // 벡터의 3 번째 원소를 5로 변경
```

```
    int n = v[2]; // 또는 n=v.at(2), 벡터 3번째 원소를 n에 저장
```

```
    cout << "벡터 용량 : " << v.capacity() << endl;
```

```
    cout << "벡터 원소 개수 : " << v.size() << endl;
```

```
    //4) 벡터에 저장된 원소 출력
```

```
    for (auto i = 0; i < v.size(); i++)
```

```
        cout << v[i] << " ";
```

```
    cout << endl;
```

```
    //5) 벡터에 저장된 원소 출력 – 범위기반 for
```

```
    for (auto &i : vc)
```

```
        cout << i << " ";
```

```
    cout << endl;
```

```
}
```

vector 컨테이너 활용

```
class Person{
    char *name;
    int id;
public:
    Person(int id, const char *name);
    Person(Person &&p) noexcept;
    ~Person();
};

int main(){
    vector<Person> vp;
    //push_back & emplace_back 비교
    cout << "=1=====" << endl;
    vp.push_back(Person(1, "hallym")); //이동 생성자 호출
    cout << "=2=====" << endl;
    vp.emplace_back(2, "software"); //이동 생성자 호출 없음, 권장
}
```

vector 컨테이너 활용

```

#include <iostream>
#include <initializer_list>
#include <vector>
using namespace std;
class InitCon {
private:
    vector<double> dv;

public:
    //initializer_list 생성자 :
    InitCon(initializer_list<double> value) {
        for (auto data : value)
            dv.emplace_back(data);
    }

    void show() const{
        for (auto &data : dv)
            cout << data << " ";
        cout << endl;
    }
};

int main() {
    InitCon ic = { 2.3, 4.5, 7.8, 11.2, 56.3, 6.7 }; //객체 생성
    ic.show();
}

```

▪ initializer-list 생성자

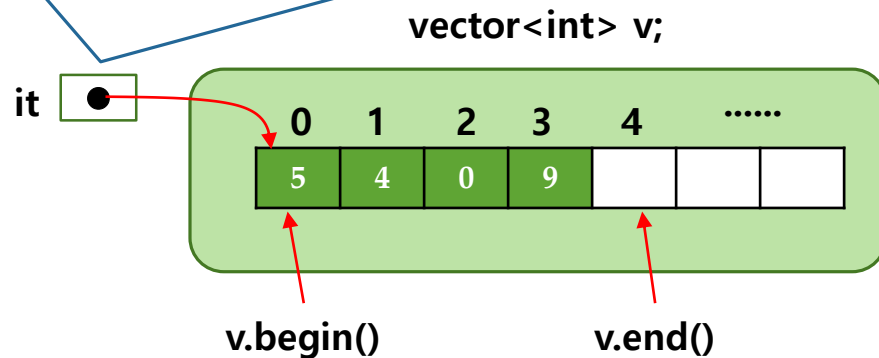
- `initializer_list<T>` 타입을 첫번째로 받고 다른 파라미터가 없거나
- 디폴트 값을 가진 매개변수를 추가로 받는 생성자

iterator 사용

- iterator란?
 - 반복자라고도 부름
 - 컨테이너의 원소를 가리키는 포인터
- iterator 변수 선언
 - 구체적인 컨테이너를 지정하여 반복자 변수 생성

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

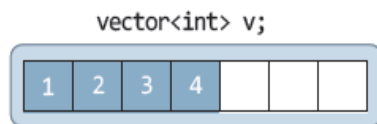
it는 원소가 int 타입인 벡터의 원소에 대한 포인터



iterator 사용

벡터 생성

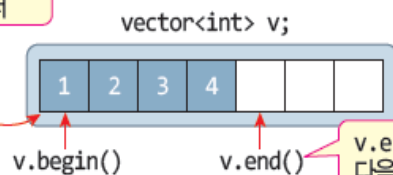
```
vector<int> v;
for(int i=1; i<=4; i++)
    v.push_back(i);
```



iterator 변수 선언
및 초기화

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

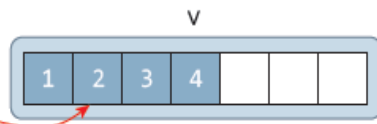
it는 int 타입 벡터의
원소에 대한 포인터



v.end()는 마지막
다음 원소에 대한 주소

iterator 증가

```
it++;
```



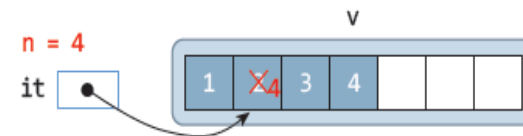
원소 읽기

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

n = 2

원소 쓰기

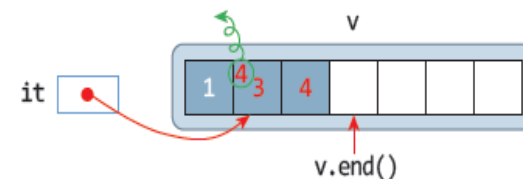
```
n = n*2;
*it = n;
```



원소 삭제

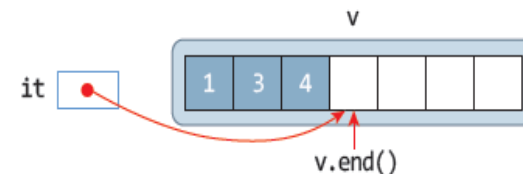
```
it = v.erase(it);
```

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



끝으로 옮기기

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



iterator를 사용하여 vector 원소 접근 & 삭제

```
#include <vector>
int main() {
    vector<string> vs={"java", "c++", "software", "script"}; //문자열 벡터 생성
    vector<string>::iterator it; //문자열 벡터 원소에 대한 포인터 it 선언
```

```
//1) iterator를 사용하여 모든 원소 탐색
cout << "--- 문자열 벡터 원소 출력 ---" << endl;
for (it = vs.begin(); it != vs.end(); it++)
    cout << *it << endl;
```

```
string str;
cout << "--- 문자열 벡터 원소 제거 ---" << endl;
cout << "제거할 문자열 >> ";
getline(cin, str);
for (it = vs.begin(); it != vs.end(); it++) {
    if (*it == str) {
        vs.erase(it); //2) 원소 제거
        break;
    }
}
}
```

```
cout << "--- 문자열 벡터 원소 삽입 ---" << endl;
cout << "삽입 위치를 찾기 위한 문자열 >> ";
getline(cin, str);
for (it = vs.begin(); it != vs.end(); it++){
    if (*it == str){
        cout << "삽입 할 문자열 >> ";
        getline(cin, str);
        vs.insert(it, str); //3) 벡터 내 it 앞에 원소 삽입
        break;
    }
}
```

pair & tuple 템플릿 클래스

- pair 템플릿 클래스
 - <utility> 헤더 파일에 정의
 - 두 값을 그룹으로 묶는 클래스
 - pair에 담긴 값은 first, second public 데이터 멤버로 접근
- tuple 템플릿 클래스
 - 여러 개의 하나로 묶어서 저장
 - 각각의 타입도 따로 지정
 - <tuple> 헤더 파일에 정의

pair & tuple 클래스 활용

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

```
using namespace std;
```

```
int main(){
```

//1) pair 활용

```
pair<string, int> mypair1("hello", 5);
```

```
pair<string, int> mypair2;
```

```
mypair2.first = "hello"; //멤버 변수를 사용하여 직접 값을 대입
```

```
mypair2.second = 6;
```

```
pair<string, int> mypair3=make_pair("hello", 7);
```

```
auto mypair4 = pair("hello", 8); //c++17, 템플릿 매개변수 추론
```

```
if(mypair2 == mypair3){
```

```
    cout<<"true"<<endl;
```

```
}
```

```
else{
```

```
    cout<<"false"<<endl;
```

```
}
```

//2) tuple 활용

```
tuple t1(16, "test", true); //c++17, 템플릿 매개변수 추론
```

```
auto [i, str, b]=t1; //c++17 구조적 바인딩, tuple분리
```

```
cout<<i<<" "<<str<<" "<<b<<endl;
```

//get<> : tuple 원소 접근

```
cout<<get<0>(t1)<<" "<<get<1>(t1)<<" "<<get<2>(t1)<<endl;
```

//멤버 변수 value : tuple 원소 개수

```
cout<<"Tuple size = "<<tuple_size<decltype(t1)>::value <<endl;
```

```
}
```

map 컨테이너

- ('키', '값')의 쌍을 원소로 저장하는 제네릭 컨테이너
 - 동일한 '키'를 가진 원소가 중복 저장되면 오류 발생
 - 요소들은 기본적으로 키를 기반으로 오름차순 정렬 – 정렬 연관 컨테이너
- '키'로 '값' 검색
- Map 클래스 주요 멤버

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

map 컨테이너 활용(1/2)

```
#include <map>
```

```
int main(){
```

```
    //1) map 객체 생성
```

```
    map<string, int> std;
```

```
    //비어 있는 map 객체 생성
```

```
    map<string, int> scores = {"Mary", 88}, {"Lucie", 98}; //유니폼 초기화 사용
```

```
    map<string, int>::iterator iter;
```

```
    //반복자 생성
```

```
    //2) map 원소 추가 하기
```

```
    scores.insert({"Mary", 88}); //initializer-list 사용
```

```
    scores["Robert"] = 77; //[]연산자 : 지정한 키가 존재하면 기존 값을 새로운 값으로 대체, 없으면 추가
```

```
    scores.try_emplace("Dog", 90); //c++ 17 추가, 지정한 키가 존재하면 아무 동작도 하지 않음
```

```
    scores.emplace("Bird", 99); //복사나 이동 작업을 수행하지 않음
```

```
    scores.insert(make_pair("John", 52)); //insert() 메소드 사용 시 키가 이미 존재하는지 검사할 수 있음
```

```
    //3) map 원소 추가 결과 확인
```

```
    if (auto result = scores.insert({"Cat", 89}); result.second){ //c++17 if문 이니셜라이저
```

```
    //if(auto [iter, success] = scores.insert({"Cat", 89}); success) -> c++17 구조적 바인딩 적용
```

```
        cout << "Insert succeeded!!!" << endl;
```

```
    }
```

```
    else{
```

```
        cout << "Insert failed!!" << endl;
```

```
    }
```

map 컨테이너 활용(2/2)

//4) map 원소 접근

```
cout << "학생 이름과 성적" << endl;
for (iter = scores.begin(); iter != scores.end(); iter++){
    cout << setw(10) << left << iter->first << " ";
    cout << setw(4) << iter->second << endl;
}
```

```
for (const auto &[key, value] : scores){//C++ 17 구조적 바인딩 & 범위 기반 for 사용
    cout << setw(10) << key << " " << setw(4) << value << endl;
}
```

```
for (const auto &value : scores){//범위 기반 for
    cout << setw(10) << value.first << " " << setw(4) << value.second << endl;
}
```

//5) 특정 원소 검색

```
auto it= scores.find("Dog"); //키값에 해당하는 원소를 가리키는 iterator 리턴
if(it != end(scores)){
    cout<<"Dog = " <<it->second<<endl;
}
else{
    cout<<"error"<<endl;
}
}
```


알고리즘

- 반복자를 사용하여 연산을 처리
- 모든 컨테이너에 적용할 수 있는 전역 함수

```
#include <iostream>
#include <vector>
#include <algorithm> // for_each() 알고리즘 함수 사용
using namespace std;

void print(int n) {
    cout << n << " ";
}

int main() {
    vector<int> v = { 1, 2, 3, 4, 5 };

    //1) for_each() 함수는 벡터 v의 첫번째 원소부터 끝까지 검색하면서
    // 각 원소에 대해 print(int n) 호출, 매개 변수 n에 각 원소 값 전달
    for_each(v.begin(), v.end(), print);

    //람다식을 사용하여 벡터의 모든 원소 출력
    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
}
```

알고리즘 활용

```

C:\WINDOWS\system32\cmd.exe
원본 벡터
17 10 13 10 15 11

오름차순으로 정렬된 벡터
10 10 11 13 15 17

역순으로 벡터 원소 출력
17 15 13 11 10 10

2 회전 후 벡터 원소 출력
13 11 10 10 17 15

임의 순서로 정렬된 벡터 원소 출력
17 11 10 10 13 15

원소 값 10의 개수: 2
짝수 값을 갖는 원소 개수: 2

Found 15 in vector? true
계속하려면 아무 키나 누르십시오 . . .

```

```

#include <vector>
#include <algorithm>

void print(int value){ cout << value << " "; }
bool isEven(int value){ return (value % 2 == 0); }

int main(){
    vector<int> vec;
    int arr[5] = { 5, 7, 9, 11, 13 }
    vec.push_back(17); vec.push_back(10);
    vec.push_back(13); vec.push_back(10);
    vec.push_back(15); vec.push_back(11);

    for_each(arr, arr + 5, print);
    cout << "원본 벡터" << endl;
    for_each(vec.begin(), vec.end(), print);

    cout << "오름차순으로 정렬된 벡터" << endl;
    sort(vec.begin(), vec.end());
    for_each(vec.begin(), vec.end(), print);

```

알고리즘 활용

```
cout << "역순으로 벡터 원소 출력" << endl;
reverse(vec.begin(), vec.end());
for_each(vec.begin(), vec.end(), print);

cout << "2 회전 후 벡터 원소 출력" << endl;
rotate(vec.begin(), vec.begin()+2, vec.end());
for_each(vec.begin(), vec.end(), print);

cout << "임의 순서로 정렬 된 벡터 원소 출력" << endl;
random_shuffle(vec.begin(), vec.end());
for_each(vec.begin(), vec.end(), print);

cout << "원소 값 10의 개수: ";
cout << count(vec.begin(), vec.end(), 10);

cout << "짝수 값을 갖는 원소 개수: ";
cout << count_if(vec.begin(), vec.end(), isEven);

cout << "Found 15 in vector? " << boolalpha;
cout << binary_search(vec.begin(), vec.end(), 15) << endl; //이진 탐색

return 0;
}
```

람다

- 람다 대수와 람다식
 - 람다 대수에서 람다식은 수학 함수를 단순하게 표현하는 기법

x, y 를 더하는 수학 함수 f

$$f(x, y) = x + y$$

함수 f 의 람다식

$$(x, y) \rightarrow x + y$$

람다식 f 계산

$$\begin{aligned} & ((x, y) \rightarrow x + y)(2, 3) \\ &= 2 + 3 \\ &= 5 \end{aligned}$$

- C++ 람다
 - 익명의 함수 만드는 기능으로 C++11에서 도입
 - 람다식, 람다 함수로도 불림
 - C#, Java, 파이썬, 자바스크립트 등 많은 언어들이 도입하고 있음

C++에서 람다식 선언

- 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 */
};

(a) 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
  
```

(b) 람다식 작성 및 호출 사례

간단한 람다식 만들기

예제1) 매개변수 x, y의 합을 출력하는 람다식 만들기

매개변수 x, y의 합을 출력하는 람다식

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

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

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

auto로 람다식 저장 및 호출

예제 2) auto로 람다식 다루기

auto를 이용하여 변수 love에 람다식을 저장하고, love를 이용하여 람다식 호출

```
int main() {  
    auto love = [](string a, string b) {  
        cout << a << "보다 " << b << "가 좋아" << endl;  
    };  
    love("돈", "너");    // 람다식 호출  
    love("냉면", "만두"); // 람다식 호출  
}
```

- * auto를 이용하여 람다식을 변수에 저장하는 사례
- * 람다식의 형식은 컴파일러만 알기 때문에, 개발자가 람다식을 저장하는 변수의 타입을 선언할 수 없음!

캡처 리스트와 리턴 타입을 가지는 람다식

예제 3) 반지름이 r 이 원의 면적으로 리턴하는 람다식 만들기

지역 변수 π 의 값을 받고, 매개변수 r 을 이용하여 반지름 값을 전달받아, 원의 면적을 계산하여 리턴하는 람다식을 작성하고, 람다식을 호출하는 코드를 프로그램을 작성하라.

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

int main() {
    double pi = 3.14; // 지역 변수

    auto calc = [pi](int r) -> double { return pi*r*r; };

    cout << "면적은 " << calc(3); // 람다식 호출. 28.26출력
}
```

* 캡처 리스트와 리턴타입을 가지는 람다식 연습

캡처 리스트에 참조를 활용하는 람다식

예제 4) 캡처 리스트에 참조 활용. 합을 외부에 저장하는 람다식 만들기

지역 변수 `sum`에 대한 참조를 캡처 리스트를 통해 받고, 합한 결과를 지역변수 `sum`에 저장한다.

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

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

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

지역변수 `sum`에
대한 참조

* 캡처 리스트를 통해 지역 변수의 참조를 받아 지역 변수를 접근하는 연습

활용1.

- 배열의 순서를 역순으로 할 수 있는 템플릿 함수를 작성하시오. 템플릿 함수를 만든 뒤에는 int, double, char, string 자료형을 사용해서 함수를 테스트한다. 데이터를 교환하는 템플릿 함수, 배열의 요소를 출력하는 템플릿 함수도 함께 만들어서 활용하도록 한다

```
int arr1[] = { 3, 7, 2, 12, 14 };
double arr2[] = { 22.7, 14.2, 3.8, 12.23, 11.2 };
char arr3[] = { 'C', 'a', 'B', 'E', 'N', 'Q' };
string arr4[] = { "John", "Lu", "Mary", "Su" };
```

```
C:\WINDOWS\system32\cmd.exe
Original array
3 7 2 12 14
Reversed array
14 12 2 7 3

Original array
22.7 14.2 3.8 12.23 11.2
Reversed array
11.2 12.23 3.8 14.2 22.7

Original array
C a B E N Q
Reversed array
Q N E B a C

Original array
John Lu Mary Su
Reversed array
Su Mary Lu John

계속하려면 아무 키나 누르십시오 . . .
```

활용 2.

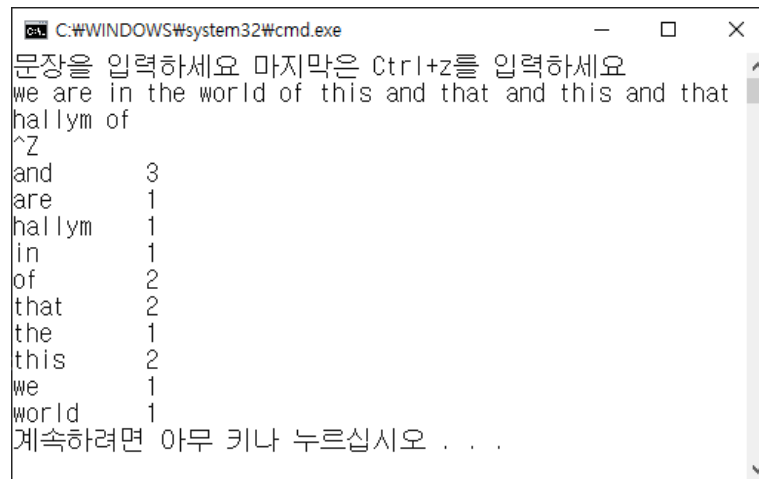
- map을 활용하여 단어 개수 세기

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

```
int main(){
```

```
    return 0;
```

```
}
```



A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window shows the following text: "문장을 입력하세요. 마지막은 Ctrl+z를 입력하세요", "we are in the world of this and that and this and that", "hallym of", "^Z", and a word frequency list: "and 3", "are 1", "hallym 1", "in 1", "of 2", "that 2", "the 1", "this 2", "we 1", "world 1". Below the list is the text "계속하려면 아무 키나 누르십시오 . . .".

활용 3.

- map 컨테이너를 이용하여 (영어, 한글) 단어를 쌍으로 저장하고, 영어로 한글을 검색하는 사전을 작성하시오

```
저장된 단어 개수 3
찾고 싶은 단어>> apple
사과
찾고 싶은 단어>> lov
없음
찾고 싶은 단어>> love
사랑
찾고 싶은 단어>> exit
종료합니다...
```

- 벡터를 사용하여 책을 저장하고 저자와 년도로 검색하는 프로그램을 작성하시오

```
class Book {
    string title;    // 책 이름
    string author;  // 저자
    int year;       //출판 년도
public:
    Book() {}
    Book(string title, string author, int year) ;
    string getAuthor();    int getYear() { return year; }
    void show();
};
```

```
class BookManager {
    vector<Book*> v;
    void searchByAuthor(); //저자로 검색
    void searchByYear(); //년도로 검색
    void bookIn(); //저장
    void finish();
public:
    void run();
};
```

Q & A

- “템플릿과 STL”에 대한 학습이 끝났습니다.
- 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.
- 과제와 퀴즈(제한 기간내 2회 응시 가능)가 있습니다.
- 수고하셨습니다.^^