




# 고급프로그래밍

## 템플릿 STL

Professor Jeong, Mun-Ho

Robot Vision & Intelligence Laboratory  
Kwangwoon University  
(02-940-5625, mhjeong@kw.ac.kr)

# Schedule

week	Topics		Homework	Quiz
1	과목소개	교과목 소개 (1), C++ 시작 (2)		
2	C++	C++ 프로그래밍의 기본(3, 3/12), 클래스와 객체(4, 3/14)	1	1
3		휴강(3/17), 객체생성과 사용(5, 3/19)	2	
4		함수와 참조(6, 3/26), 복사 생성자와 함수중복(7. 3/28)	3	2, 3
5		static friend 연산자중복(8, 4/2), 연산자중복 상속(9, 4/4)	4	4
6		상속(10, 4/9), 가상함수 추상클래스(11, 4/11)	5	5
7		 템플릿 STL(12), 표준 입출력, 파일 입출력		
8	중간고사			
9	C++	예외처리 및 C 사용, 람다식	6	6
10		멀티스레딩	7	7
11		멀티스레딩, 고급문법	8	8
12		고급문법	9	9
13	병렬 프로그래밍	병렬프로그래밍		
14		병렬프로그래밍		
15	기말고사			

# 오늘의 학습내용

- 템플릿
- STL

템플릿

# 중복 함수의 코드 중복

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

```
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 << '\t' << b << endl;

    double c=0.3, d=12.5;
    myswap(c, d); // myswap(double& a, double& b) 호출
    cout << c << '\t' << d << endl;
}
```

5	4
12.5	0.3

# 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
- 템플릿 선언

```
template <class T> 또는  
template <typename T>
```

```
3 개의 제네릭 타입을 가진 템플릿 선언  
template <class T1, class T2, class T3>
```

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

템플릿을 이용한 제네릭 함수 myswap

# 템플릿 함수

```
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;  
}
```

중복 함수들

제네릭 함수  
만들기(일반화)

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

템플릿을 이용한  
제네릭 함수

# 템플릿으로부터의 구체화

## ■ 구체화(specialization)

- 템플릿의 제네릭 타입에 구체적인 타입 지정
  - 템플릿 함수로부터 구체화된 함수의 소스 코드 생성

```
template <class 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);
}
```

myswap(a, b) 호출에 필요  
한 함수 구체화 진행

T에 **int**를 대입  
하여 구체화된  
소스 코드 생성

구체화

```
void myswap(int & a, int & b) {
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
```

```
int main() {
    int a=4, b=5;
    myswap(a, b);
}
```

컴파일 후 실행



# 예제

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

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

template <class 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;
```

myswap(int& a, int& b)  
함수 구체화 및 호출

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

myswap(double& a, double& b)  
함수 구체화 및 호출

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

myswap(Circle& a, Circle& b)  
함수 구체화 및 호출

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

# 구체화 오류

- 제네릭 타입에 구체적인 타입 지정 시 주의

두 매개 변수 a, b의 제  
네릭 타입 동일

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

```
int s=4;  
double t=5;  
myswap(s, t);
```

두 개의 매개 변수의 타  
입이 서로 다름

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

# 템플릿 장점과 제네릭 프로그래밍

- 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. 활용
  - 보편화 추세
    - Java, C# 등 많은 언어에서 활용

# 예제

- 두 값을 매개 변수로 받아 큰 값을 리턴하는 제네릭 함수 bigger()를 작성하라.

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

template <class T>
T bigger(T a, T b) { // 두 개의 매개 변수를 비교하여 큰 값을 리턴
    if(a > b)
        return a;
    else
        return b;
}

int main()
{
    int a=20, b=50;
    char c='a', d='z';
    cout << "bigger(20, 50)의 결과는 " << bigger(a, b) << endl;
    cout << "bigger('a', 'z')의 결과는 " << bigger(c, d) << endl;
}
```

```
bigger(20, 50)의 결과는 50
bigger('a', 'z')의 결과는 z
```

# 예제

- 배열과 크기를 매개 변수로 받아 합을 구하여 리턴하는 제네릭 함수 `add()`를 작성하라.

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

template <class T>
T add(T data [], int n) { // 배열 data에서 n개의 원소를 합한 결과를 리턴
    T sum = 0;
    for(int i=0; i<n; i++) {
        sum += data[i];
    }
    return sum; // sum와 타입과 리턴 타입이 모두 T로 선언되어 있음
}

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

    cout << "sum of x[] = " << add(x, 5) << endl; // 배열 x와 원소 5개의 합을 계산
    cout << "sum of d[] = " << add(d, 6) << endl; // 배열 d와 원소 6개의 합을 계산
}

sum of x[ ] = 15
sum of d[ ] = 23.7
```

# print() 템플릿 함수의 문제점

T가 char로 구체화되는 경우, 정수 1, 2, 3, 4, 5에 대한 그래픽 문자 출력

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

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

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

print() 템플릿의 T가 int 타입으로 구체화

```
    char c[5] = {1, 2, 3, 4, 5};
    print(c, 5);
}
```

print() 템플릿의 T가 char 타입으로 구체화

char로 구체화되면  
숫자대신 문자가  
출력되는 문제 발생!

1	2	3	4	5
1.1	2.2	3.3	4.4	5.5
1	2	3	4	6

# 예제 - 중복 함수가 우선

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

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

템플릿 함수와  
중복된 print() 함수

```
void print(char array [], int n) { // char 배열을 출력하기 위한 함수 중복
    for(int i=0; i<n; i++)
        cout << (int)array[i] << '\t'; // array[i]를 int 타입으로 변환하여 정수 출력
    cout << endl;
}
```

중복된 print() 함수가  
우선 바인딩

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

```
    char c[5] = {1,2,3,4,5};
    print(c, 5);
}
```

템플릿 print() 함수로  
부터 구체화

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

주목

# 제네릭 클래스

- 제네릭 클래스 선언
- 제네릭 클래스 구현
- 클래스 구체화 및 객체 활용

```
template <class T>
class MyStack {
    int tos;
    T data [100]; // T 타입의 배열
public:
    MyStack();
    void push(T element);
    T pop();
};
```

```
template <class T>
void MyStack<T>::push(T element) {
    ...
}
template <class T>
T MyStack<T>::pop() {
    ...
}
```

```
MyStack<int> iStack; // int 타입을 다루는 스택 객체 생성
MyStack<double> dStack; // double 타입을 다루는 스택 객체 생성

iStack.push(3);
int n = iStack.pop();
```



# 예제 - 두 개의 제네릭 타입

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

template <class T1, class T2> // 두 개의 제네릭 타입
선언
class GClass {
    T1 data1;
    T2 data2;
public:
    GClass();
    void set(T1 a, T2 b);
    void get(T1 &a, T2 &b);
};

template <class T1, class T2>
GClass<T1, T2>::GClass() {
    data1 = 0; data2 = 0;
}

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

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

data1을 a에, data2를  
b에 리턴하는 함수

```
int main() {
    int a;
    double b;
    GClass<int, double> x;
    x.set(2, 0.5);
    x.get(a, b);
    cout << "a=" << a << '\t' << "b=" << b << endl;

    char c;
    float d;
    GClass<char, float> y;
    y.set('m', 12.5);
    y.get(c, d);
    cout << "c=" << c << '\t' << "d=" << d << endl;
}
```

```
a=2    b=0.5
c=m    d=12.5
```

# 함수 객체(Function Object)

- 객체를 함수처럼 사용
- 연산자 중복으로 선언함: `operator ( )`

```
2 #include <iostream>
3 using namespace std;
4
5 class Plus
6 {
7 public:
8     int operator( )(int a, int b)
9     {
10         return a + b;
11     }
12 };
13
14 int main()
15 {
16     Plus pls;
17
18     cout << "pls(10, 20): " << pls(10, 20) << endl;
19     return 0;
20 }
```

explicit call: `pls.operator()(10,20)`

implicit call

# 함수 객체

- 사용하는 이유
  - 객체 멤버 활용
  - 일반함수 보다 호출이 빠름

```
2 #include <iostream>
3 using namespace std;
4
5 class MoneyBox
6 {
7     int total;
8
9 public:
10     MoneyBox(int _init = 0) : total(_init) { }
11
12     int operator( )(int money)
13     {
14         total += money;
15         return total;
16     }
17 };
18
19 int main()
20 {
21     MoneyBox mb;
22
23     cout << "mb(100): " << mb(100) << endl;
24     cout << "mb(500): " << mb(500) << endl;
25     cout << "mb(2000): " << mb(2000) << endl;
26
27     return 0;
28 }
```

STL

# C++ 표준 템플릿 라이브러리(STL)

## ■ 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, 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

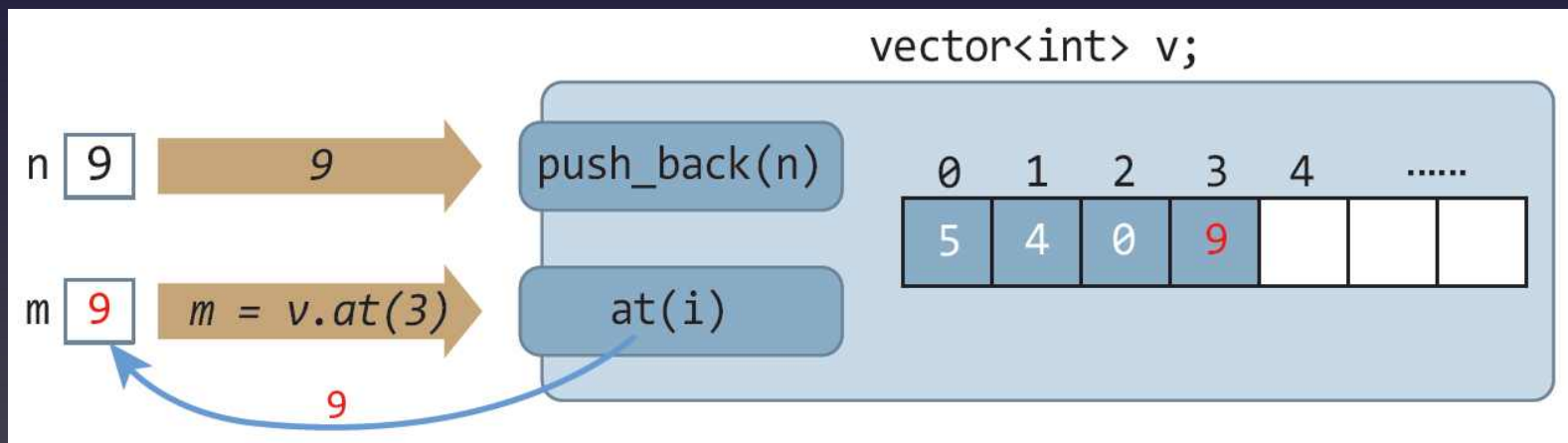
〈표 10-3〉 STL 알고리즘 함수들

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

# vector 컨테이너

## ■ 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



# vector의 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)



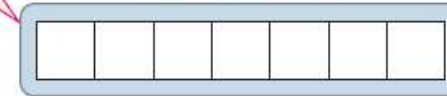
# vector 다루기 사례

vector 생성

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

정수 벡터  
생성

vector<int> v

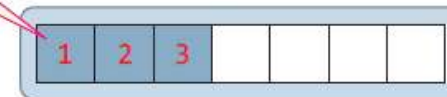


정수 원소 삽입

```
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);
```

n = 2

v

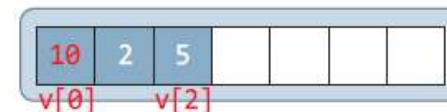


원소 값 접근

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

m = 5

v



# 예제

- string 타입의 vector를 이용하여 문자열을 저장하는 벡터를 만들고, 5개의 이름을 입력 받아 사전에 서 가장 뒤에 나오는 이름을 출력하라

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

int main() {
    vector<string> sv; // 문자열 벡터 생성
    string name;

    cout << "이름을 5개 입력하라" << endl;
    for(int i=0; i<5; i++) { // 한 줄에 한 개씩 5 개의 이름을 입력받는다.
        cout << i+1 << ">>";
        getline(cin, name);
        sv.push_back(name);
    }
    name = sv.at(0); // 벡터의 첫 원소
    for(int i=1; i<sv.size(); i++) {
        if(name < sv[i]) // sv[i]의 문자열이 name보다 사전에서 뒤에 나옴
            name = sv[i]; // name을 sv[i]의 문자열로 변경
    }
    cout << "사전에서 가장 뒤에 나오는 이름은 " << name << endl;
}
```

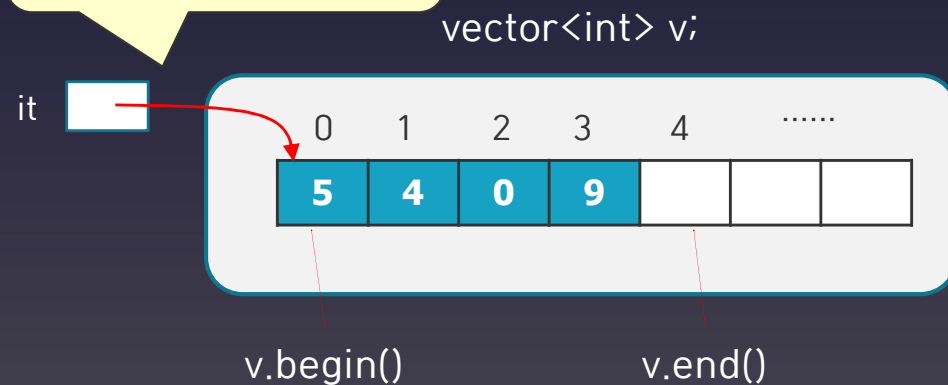
```
이름을 5개 입력하라
1>>황기태
2>>이재문
3>>김남윤
4>>한원선
5>>애슐리
사전에서 가장 뒤에 나오는 이름은 황기태
```

# 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



벡터 생성

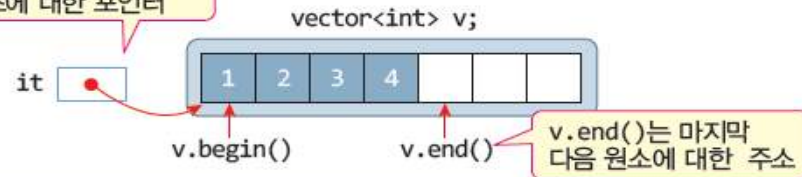
```
vector<int> v;  
for(int i=1; i<=4; i++)  
    v.push_back(i);
```



iterator 변수 선언  
및 초기화

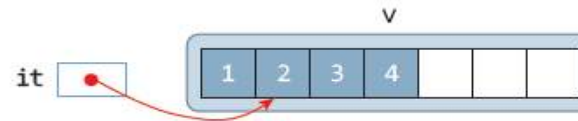
```
vector<int>::iterator it;  
it = v.begin();
```

it는 int 타입 벡터의  
원소에 대한 포인터



iterator 증가

```
it++;
```



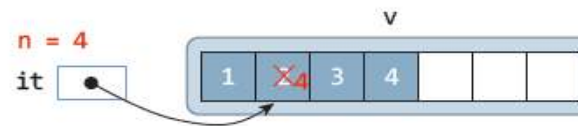
원소 읽기

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

n = 2

원소 쓰기

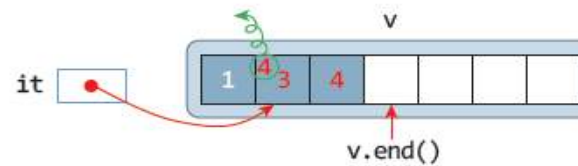
```
n = n*2;  
*it = n;
```



원소 삭제

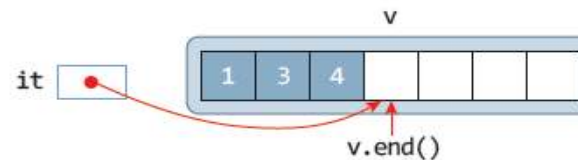
```
it = v.erase(it);
```

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



끝으로 옮기기

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



# 예제 - iterator

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

int main() {
    vector<int> v; // 정수 벡터 생성
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);

    vector<int>::iterator it; // 벡터 v의 원소에 대한 포인터 it 선언

    for(it=v.begin(); it != v.end(); it++) { // iterator를 이용하여 모든 원소 탐색
        int n = *it; // it가 가리키는 원소 값 리턴
        n = n*2; // 곱하기 2
        *it = n; // it가 가리키는 원소에 값 쓰기
    }

    for(it=v.begin(); it != v.end(); it++) // 벡터 v의 모든 원소 출력
        cout << *it << ' ';
    cout << endl;
}
```

# iterator vs. const iterator

- iterator , constant iterator

```
vector<int>::iterator iter;  
vector<int>::const_iterator citer;
```

- constant iterator로 컨테이너 요소 수정 불가

```
vector<int> v(8) ;  
:  
vector<int>::iterator iter;  
vector<int>::const_iterator citer;  
  
iter = v.begin();  *iter = 100;  
  
citer = v.begin();  
*citer = 100;    // Error !
```

- constant vector는 const\_iterator에 의해 접근 가능

# 예제

```
#include <vector>
#include <algorithm>

void PrintVector(const vector<int> & v)
{
    // vector<int>::iterator iter = v.begin(); // Compile Error
    vector<int>::const_iterator citer = v.begin();

    for( ; citer != v.end(); citer++)
        cout << *citer << " ";
    cout << endl;
}
```

```
int main()
{
    vector<int> v(8);

    for (int i = 0; i < v.size(); i++)
        v[i] = i + 1 ;

    PrintVector( v );

    return 0;
}
```

# 실습 1 - vector 초기화

- 아래 프로그램에서 각 `vector`가 출력되도록 프로그램을 수정하십시오.

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

void main()
{
    vector<int> v1(10);
    vector<int> v2(10,7);
    vector<int> v3(v2);
    int ar[]={1,2,3,4,5,6,7,8,9};
    vector<int> v4(&ar[2],&ar[5]);
}
```



# 실습 1 - 답

- 아래 프로그램에서 각 `vector`가 출력되도록 프로그램을 수정하시오.

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

void main()
{
    vector<int> v1(10);
    vector<int> v2(10,7);
    vector<int> v3(v2);
    int ar[]={1,2,3,4,5,6,7,8,9};
    vector<int> v4(&ar[2],&ar[5]);
}
```

```
void main()
{
    :
    vector<int>::iterator it;
    for(it = v1.begin(); it != v1.end(); it++)
        cout << *it << " ";
    cout << endl;
    for(it = v2.begin(); it != v2.end(); it++)
        cout << *it << " ";
    cout << endl;
    for(it = v3.begin(); it != v3.end(); it++)
        cout << *it << " ";
    cout << endl;
    for(it = v4.begin(); it != v4.end(); it++)
        cout << *it;
}
```

# vector Sorting I

- 알고리즘 함수
  - 템플릿 함수
  - 전역 함수
    - STL 컨테이너 클래스의 멤버 함수가 아님
  - iterator와 함께 작동
- sort() 함수 사례
  - 두 개의 매개 변수
    - 첫 번째 매개 변수 : 소팅을 시작한 원소의 주소
    - 두 번째 매개 변수 : 소팅 범위의 마지막 원소 다음 주소

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

```
...
```

```
sort(v.begin(), v.begin()+3); // v.begin()에서 v.begin()+2까지, 처음 3개 원소 정렬
```

```
sort(v.begin()+2, v.begin()+5); // 벡터의 3번째 원소에서 v.begin()+4까지, 3개 원소 정렬
```

```
sort(v.begin(), v.end()); // 벡터 전체 정렬
```

# 예제

- 정수 벡터에 5개의 정수를 입력 받아 저장하고, sort()를 이용하여 정렬하는 프로그램을 작성하라.

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

int main() {
    vector<int> v; // 정수 벡터 생성

    cout << "5개의 정수를 입력하세요>> ";
    for(int i=0; i<5; i++) {
        int n;
        cin >> n;
        v.push_back(n); // 키보드에서 읽은 정수를 벡터에 삽입
    }

    // v.begin()에서 v.end() 사이의 값을 오름차순으로 정렬
    // sort() 함수의 실행 결과 벡터 v의 원소 순서가 변경됨
    sort(v.begin(), v.end());

    vector<int>::iterator it; // 벡터 내의 원소를 탐색하는 iterator 변수 선언

    for(it=v.begin(); it != v.end(); it++) // 벡터 v의 모든 원소 출력
        cout << *it << ' ';
    cout << endl;
}
```

```
5개의 정수를 입력하세요>> 30 -7 250 6 120
-7 6 30 120 250
```

# vector Sorting II

- 사용자 정의 함수로 Sorting

```
#include <vector>
#include <algorithm>
:
bool custom_func(const T& i, const T& j);
:
sort(vector.begin( ), vector.end( ), custom_func);
```

## 실습 2

- 입력한 문자열을 문자열 길이의 오름차순으로 출력하도록 프로그램을 완성하시오(단, 사용자 정의 함수 이용).

```
int main()
{
    vector<string> v;
    vector<string>::iterator it;

    v.push_back("cccc");
    v.push_back("aa");
    v.push_back("bbb");
    v.push_back("dddd");

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

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

## 실습 2 - 답

- 입력한 문자열을 문자열 길이의 오름차순으로 출력하도록 프로그램을 완성하시오.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

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

# vector Sorting III

- 함수 객체를 이용한 `sorting`

```
#include <vector>
#include <algorithm>
:
struct custom_func    //function object
{
    :
};
sort(vector.begin( ), vector.end( ), custom_func( ));
```

## 실습 3

- 입력한 문자열을 문자열 길이의 오름차순으로 출력하도록 프로그램을 완성하시오 (단, 함수 객체 이용).

```
int main()
{
    vector<string> v;
    vector<string>::iterator it;

    v.push_back("cccc");  v.push_back("aa");
    v.push_back("bbb");   v.push_back("dddd");

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

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



## 실습 3 - 답

- 입력한 문자열을 문자열 길이의 오름차순으로 출력하도록 프로그램을 완성하시오.

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

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

