



템플릿과 표준 템플릿 라이브러리(STL)

# 학습 목표

1. 일반화와 템플릿의 개념과 목적을 이해한다.
2. 템플릿으로부터 구체화의 과정을 이해한다.
3. 템플릿 함수와 템플릿 클래스를 작성하고 활용할 수 있다.
4. C++ 표준 템플릿 라이브러리(STL)에 대해 이해한다.
5. STL의 vector, map 컨테이너를 이해하고 활용할 수 있다.
6. STL의 iterator와 알고리즘 함수에 대해 이해하고 간단히 활용할 수 있다.
7. auto로 변수를 쉽게 선언하는 것을 알고 활용할 수 있다.
8. 람다식의 개념을 알고 간단한 람다식을 작성하고, 호출할 수 있다.

# 함수 중복의 약점 - 중복 함수의 코드 중복

3

```
#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 << 'Wt' << b << endl;
```

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

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

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

```
}
```

두 함수는 매개 변수만 다르  
고 나머지 코드는 동일함

동일한 코드  
중복 작성

5	4
12.5	0.3

# 일반화와 템플릿

4

- 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>
```

템플릿을 선언하  
는 키워드

제네릭 타입을  
선언하는 키워드

제네릭 타입 T 선언

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

템플릿을 이용한 제네릭 함수 myswap

# 중복 함수들로부터 템플릿 만들기 사례

5

```
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

템플릿을 선언하  
는 키워드

제네릭 타입을  
선언하는 키워드

제네릭 타입 T 선언

제네릭 함수  
만들기(일반화)

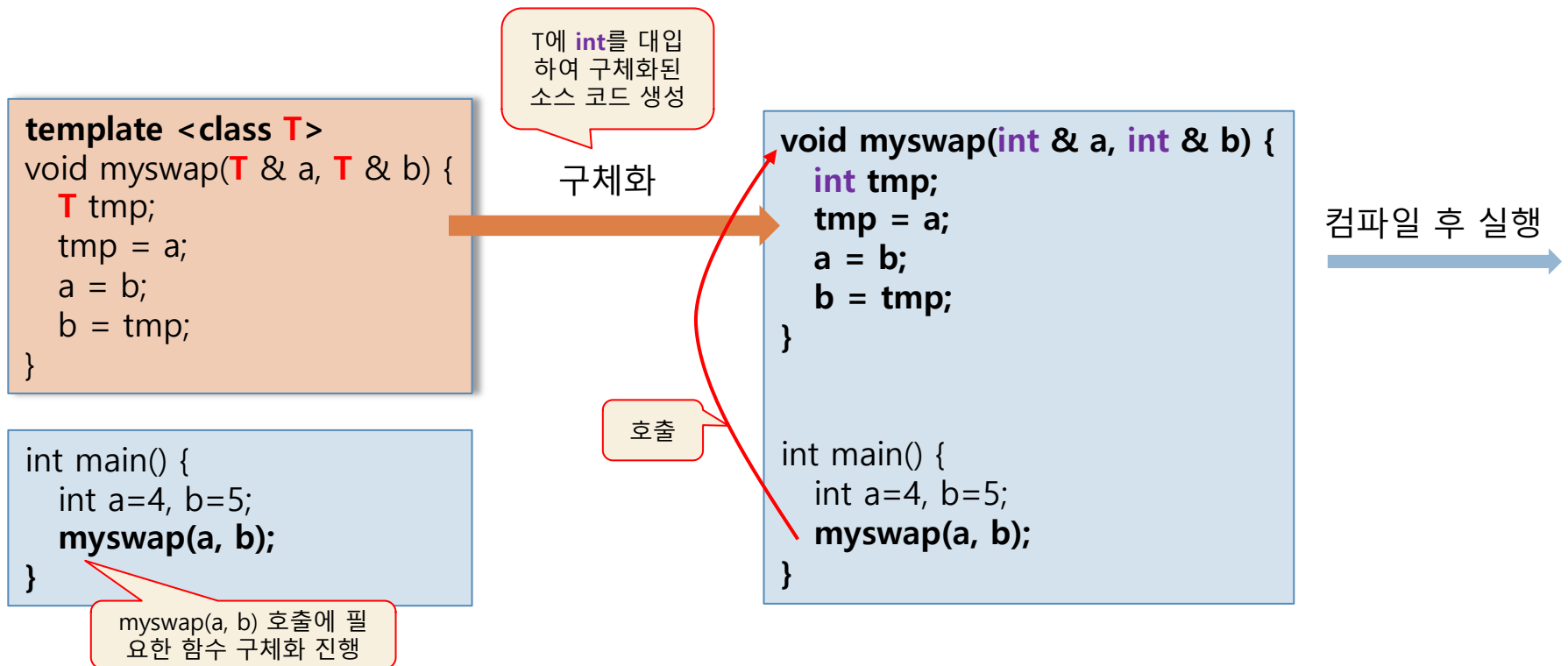
```
template <class T>  
void myswap (T &a, T &b) {  
    T tmp;  
    tmp = a;  
    a = b;  
    b = tmp;  
}
```

템플릿을 이용한  
제네릭 함수

# 템플릿으로부터의 구체화

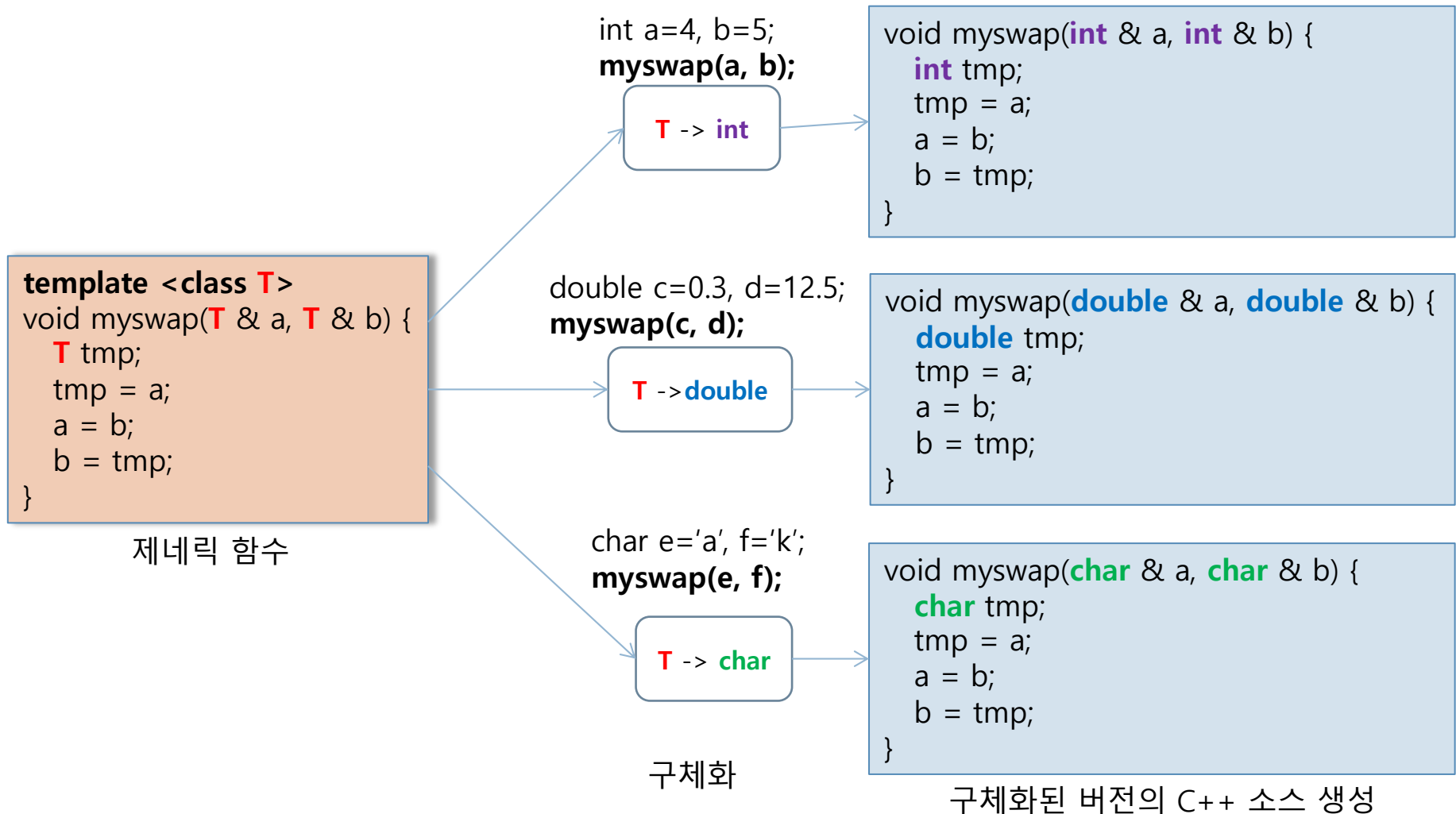
6

- 구체화(specialization)
  - ▣ 템플릿의 제네릭 타입에 구체적인 타입 지정
    - 템플릿 함수로부터 구체화된 함수의 소스 코드 생성



# 제네릭 함수로부터 구체화된 함수 생성 사례

7



# 예제 10-1 제네릭 myswap() 함수 만들기

8

```
#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;

    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;
}
```

myswap(int& a, int& b)  
함수 구체화 및 호출

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

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

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

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



# 구체화 오류

9

## □ 제네릭 타입에 구체적인 타입 지정 시 주의

두 매개 변수 a, b의  
제네릭 타입 동일

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

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

두 개의 매개 변수의  
타입이 서로 다름

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

# 템플릿 장점과 제네릭 프로그래밍

10

- 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 10-2 큰 값을 리턴하는 bigger() 함수 만들기 연습

11

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 10-3 배열의 합을 구하여 리턴하는 제네릭 add() 함수 만들기 연습

12

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
```

# 예제 10-4 배열을 복사하는 제네릭 함수 mcopy() 함수 만들기 연습

13

두 개의 배열을 매개 변수로 받아 배열을 복사하는 제네릭 mcopy() 함수를 작성하라.

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

// 두 개의 제네릭 타입 T1, T2를 가지는 copy()의 템플릿
template <class T1, class T2>
void mcopy(T1 src [], T2 dest [], int n) { // src[]의 n개 원소를 dest[]에 복사하는 함수
    for(int i=0; i<n; i++)
        dest[i] = (T2)src[i]; // T1 타입의 값을 T2 타입으로 변환한다.
}

int main() {
    int x[] = {1,2,3,4,5};
    double d[5];
    char c[5] = {'H', 'e', 'l', 'l', 'o'}, e[5];

    mcopy(x, d, 5); // int x[]의 원소 5개를 double d[]에 복사
    mcopy(c, e, 5); // char c[]의 원소 5개를 char e[]에 복사

    for(int i=0; i<5; i++) cout << d[i] << ' '; // d[] 출력
    cout << endl;
    for(int i=0; i<5; i++) cout << e[i] << ' '; // e[] 출력
    cout << endl;
}
```

mcopy()의 T1은 int로, T2  
는 double로 구체화

mcopy()의 T1, T2 모두 char  
로 구체화

```
1 2 3 4 5
H e l l o
```

# 배열을 출력하는 print() 템플릿 함수의 문제점

14

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

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

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

char로 구체화되면  
숫자대신 문자가  
출력되는 문제 발생!

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

print() 템플릿의 T가 int 타입으로 구체화

print() 템플릿의 T가 char 타입으로 구체화

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

# 예제 10-5 템플릿 함수보다 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] << '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[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 함수와  
중복된 print() 함수

중복된 print() 함수  
가 우선 바인딩

템플릿 print() 함수  
로부터 구체화

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

주목

# 제네릭 클래스 만들기

16

## ■ 제네릭 클래스 선언

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

## ■ 제네릭 클래스 구현

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

리턴 타입

## ■ 클래스 구체화 및 객체 활용

```
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();

dStack.push(3.5);
double d = dStack.pop();
```



# 예제 10-6 제네릭 스택 클래스 만들기

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

```
template <class T>
class MyStack {
    int tos; // top of stack
    T data [100]; // T 타입의 배열. 스택의 크기는 100
public:
    MyStack();
    void push(T element); // element를 data [] 배열에 삽입
    T pop(); // 스택의 탑에 있는 데이터를 data[] 배열에서 리턴
};
```

```
template <class T>
MyStack<T>::MyStack() { // 생성자
    tos = -1; // 스택은 비어 있음
}
```

```
template <class T>
void MyStack<T>::push(T element) {
    if(tos == 99) {
        cout << "stack full";
        return;
    }
    tos++;
    data[tos] = element;
}
```

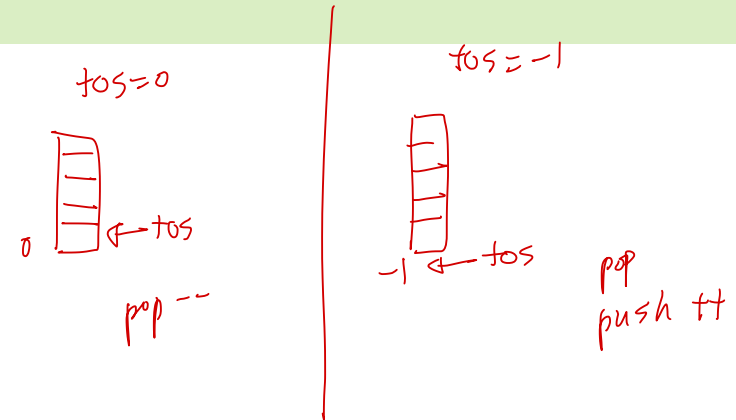
```
template <class T>
T MyStack<T>::pop() {
    T retData;
    if(tos == -1) {
        cout << "stack empty";
        return 0; // 오류 표시
    }
    retData = data[tos--];
    return retData;
}
```

```
int main() {
    MyStack<int> iStack; // int 만 저장하는 스택
    iStack.push(3);
    cout << iStack.pop() << endl;

    MyStack<double> dStack; // double 만 저장하는 스택
    dStack.push(3.5);
    cout << dStack.pop() << endl;

    MyStack<char> *p = new MyStack<char>(); // char만 저장하는 스택
    p->push('a');
    cout << p->pop() << endl;
    delete p;
}
```

3  
3.5  
a



# 예제 10-7 제네릭 스택의 제네릭 타입을 포인터나 클래스로 구체화하는 예

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

/\* 이 부분에 예제 10-6에 작성한 MyStack 템플릿 클래스 코드가 생략되었음 \*/

```
class Point {
    int x, y;
public:
    Point(int x=0, int y=0) { this->x = x; this->y = y; }
    void show() { cout << '(' << x << ',' << y << ')' << endl; }
};
```

```
int main() {
    MyStack<int*> ipStack; // int* 만을 저장하는 스택
    int *p = new int [3];
    for(int i=0; i<3; i++) p[i] = i*10; // 0, 10, 20으로 초기화
    ipStack.push(p); // 포인터 푸시
    int *q = ipStack.pop(); // 포인터 팝
    for(int i=0; i<3; i++) cout << q[i] << ' '; // 화면 출력
    cout << endl;
    delete [] p;
}
```

```
MyStack<Point> pointStack; // Point 객체 저장 스택
Point a(2,3), b;
pointStack.push(a); // Point 객체 a 푸시. 복사되어 저장
b = pointStack.pop(); // Point 객체 팝
b.show(); // Point 객체 출력
```

```
MyStack<Point*> pStack; // Point* 포인터 스택
pStack.push(new Point(10,20)); // Point 객체 푸시
Point* pPoint = pStack.pop(); // Point 객체의 포인터 팝
pPoint->show(); // Point 객체 출력
```

```
MyStack<string> stringStack; // 문자열만 저장하는 스택
string s="c++";
stringStack.push(s);
stringStack.push("java");
cout << stringStack.pop() << ' ';
cout << stringStack.pop() << endl;
}
```

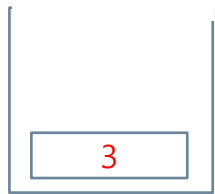
```
0 10 20
(2,3)
(10,20)
java c++
```

# 참고!! cout << a << b << c;의 실행 순서

```
cout << 1 << 2 << 3;
```

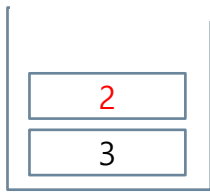
123

cout으로 << 연산자가 연속되면,  
마지막 데이터부터 거꾸로 cout의 스택에 삽입한 후  
앞에서부터 << 연산자를 순서대로 처리한다.  
이때 스택에 삽입된 데이터를 팝하여 사용한다.



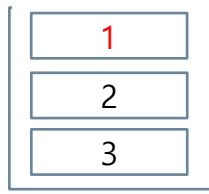
cout 스택

①



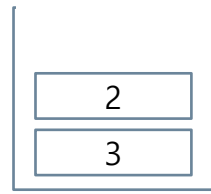
cout 스택

②



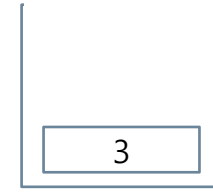
cout 스택

③



cout 스택

④ 스택 톱의 값 1로  
cout << 1 실행.  
1 출력



cout 스택

⑤ 스택 톱의 값 2로  
cout << 2 실행.  
2 출력



cout 스택

⑥ 스택 톱의 값 3로  
cout << 3 실행.  
3 출력

예제 10-7의 라인 39-40을 다음과 같이 연결하면, 잘못된 결과가 나오며, 이유를 설명하면 다음과 같다.

```
cout << stringStack.pop() << ' ' << stringStack.pop();
```

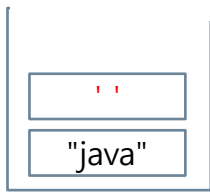
c++ java

맨 뒤에 있는 stringStack.pop() 함수를  
호출하여 리턴 값을 cout의 스택에 삽입  
하고, ' ' 문자를 cout 스택에 삽입하고,  
다시 맨 앞의 stringStack.pop() 함  
수를 호출하여 리턴 값을 cout의 스택에  
삽입한 후, 앞에서부터 << 연산자를 순  
서대로 처리한다.



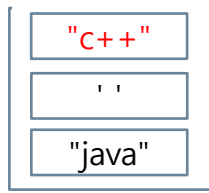
cout 스택

①



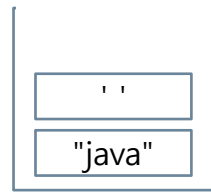
cout 스택

②



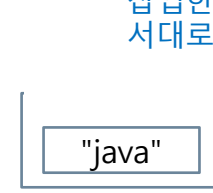
cout 스택

③



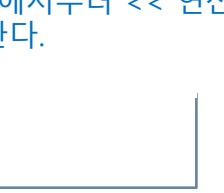
cout 스택

④ 스택 톱에  
있는 "c++" 출력



cout 스택

⑤ 스택 톱에  
있는 ' ' 출력



cout 스택

⑥ 스택 톱에  
있는 "java" 출력

# 예제 10-8 두 개의 제네릭 타입을 가진 클래스 만들기

20

```
#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);
};
```

data1을 a에, data2를  
b에 리턴하는 함수

```
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;
}
```

```
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

# C++ 표준 템플릿 라이브러리, STL

21

- 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

# STL과 관련된 헤더 파일과 이름 공간

23

## □ 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

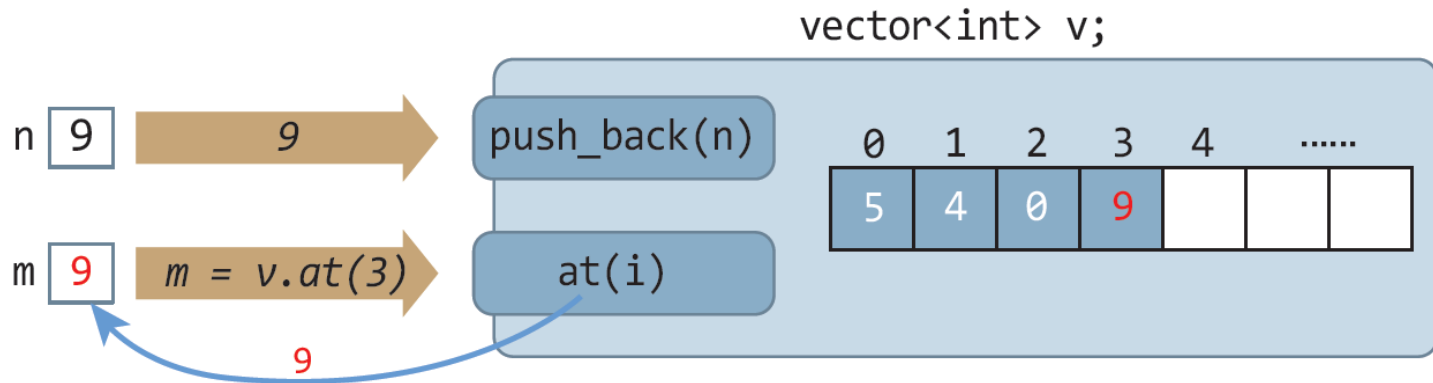
- ▣ STL이 선언된 이름 공간은 `std`

# vector 컨테이너

24

## □ 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 클래스의 주요 멤버와 연산자

25

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)

] 원소를 가리키는 반복자

# vector 다루기 사례

vector 생성

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

정수 벡터  
생성

vector<int> v

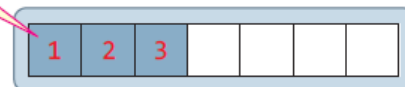


정수 원소 삽입

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

정수 삽입

v



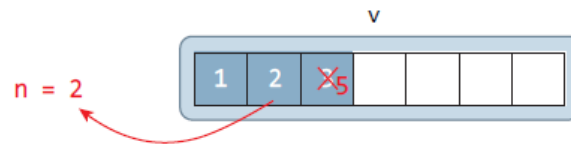
원소 개수 s  
벡터의 용량 c

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

s = 3  
c = 7

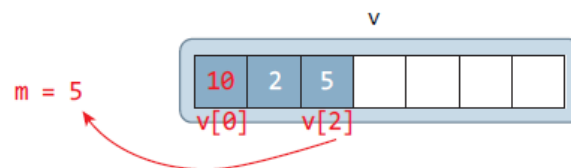
원소 값 접근

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



원소 값 접근

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



# 예제 10-9 vector 컨테이너 활용하기

27

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

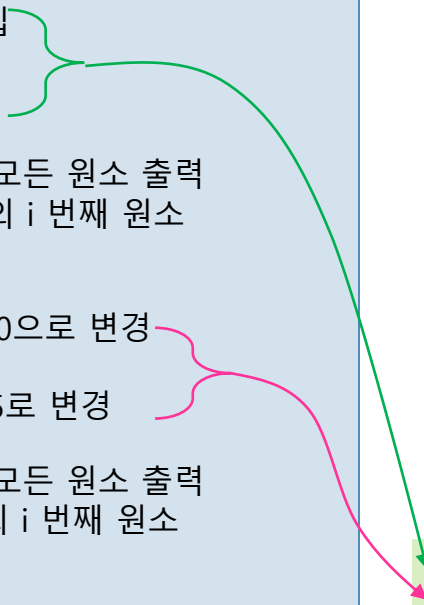
int main() {
    vector<int> v; // 정수만 삽입 가능한 벡터 생성

    v.push_back(1); // 벡터에 정수 1 삽입
    v.push_back(2); // 벡터에 정수 2 삽입
    v.push_back(3); // 벡터에 정수 3 삽입

    for(int i=0; i<v.size(); i++) // 벡터의 모든 원소 출력
        cout << v[i] << " "; // v[i]는 벡터의 i 번째 원소
    cout << endl;

    v[0] = 10; // 벡터의 첫 번째 원소를 10으로 변경
    int n = v[2]; // n에 3이 저장
    v.at(2) = 5; // 벡터의 3 번째 원소를 5로 변경

    for(int i=0; i<v.size(); i++) // 벡터의 모든 원소 출력
        cout << v[i] << " "; // v[i]는 벡터의 i 번째 원소
    cout << endl;
}
```



1 2 3  
10 2 5

# 예제 10-10 문자열을 저장하는 벡터 만들기 연습

28

string 타입의 vector를 이용하여 문자열을 저장하는 벡터를 만들고, 5개의 이름을 입력 받아 사전에서 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

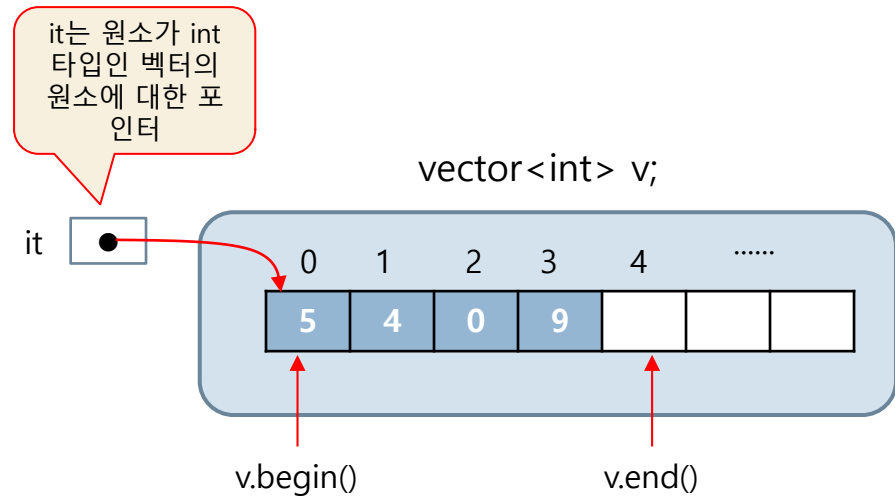
사전에서 가장 뒤에 나오는 이름은 황기태

# iterator 사용

29

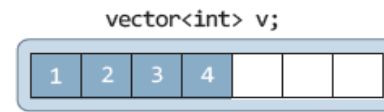
- iterator란?
  - ▣ 반복자라고도 부름
  - ▣ 컨테이너의 원소를 가리키는 포인터
- iterator 변수 선언
  - ▣ 구체적인 컨테이너를 지정하여 반복자 변수 생성

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



벡터 생성

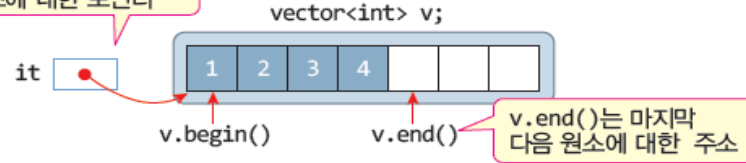
```
vector<int> v;  
for(int i=1; i<=4; i++)  
    v.push_back(i);
```



iterator 변수 선언  
및 초기화

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

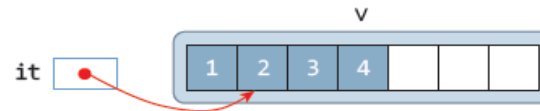
it는 int 타입 벡터의  
원소에 대한 포인터



iterator 증가

```
it++;
```

$n = v[1]$



원소 읽기

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

$n = 2$

$n = 2 \times v[1]$

원소 쓰기

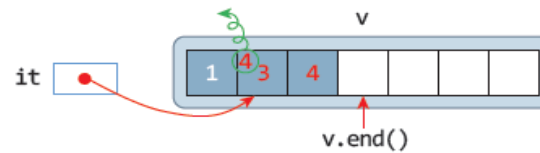
```
n = n*2;  
*it = n;
```



원소 삭제

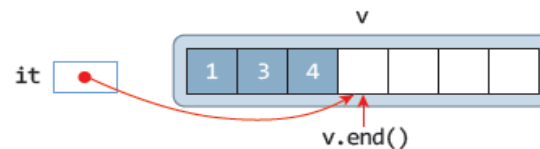
```
it = v.erase(it);
```

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



끝으로 옮기기

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



# 예제 10-11 iterator를 사용하여 vector의 모든 원소에 2 곱하기

31

```
#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;
}
```

# map 컨테이너

32

## □ 특징

- ('키', '값')의 쌍을 원소로 저장하는 제네릭 컨테이너
  - 동일한 '키'를 가진 원소가 중복 저장되면 오류 발생
- '키'로 '값' 검색
- 많은 응용에서 필요함
- #include <map> 필요

## □ 맵 컨테이너 생성 예

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

33

멤버와 연산자 함수	설명
<code>insert(pair&lt;&gt; &amp;element)</code>	맵에 '키'와 '값'으로 구성된 pair 객체 element 삽입
<code>at(key_type&amp; 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&amp; 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&amp; key]()</code>	맵에서 '키' 값에 해당하는 원소를 찾아 '값' 리턴
<code>operator=()</code>	맵 치환(복사)

# 예제 10-12 map으로 영한 사전 만들기

34

map 컨테이너를 이용하여  
(영어, 한글) 단어를 쌍으로  
저장하고, 영어로 한글을 검  
색하는 사전을 작성하라.

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

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

int main() {
    map<string, string> dic; // 맵 컨테이너 생성. 키는 영어 단어, 값은 한글 단어

    // 단어 3개를 map에 저장
    dic.insert(make_pair("love", "사랑")); // ("love", "사랑") 저장
    dic.insert(make_pair("apple", "사과")); // ("apple", "사과") 저장
    dic["cherry"] = "체리"; // ("cherry", "체리") 저장

    cout << "저장된 단어 개수 " << dic.size() << endl;

    string eng;
    while (true) {
        cout << "찾고 싶은 단어>> ";
        getline(cin, eng); // 사용자로부터 키 입력
        if (eng == "exit")
            break; // "exit"이 입력되면 종료

        if(dic.find(eng) == dic.end()) // eng '키'를 끝까지 찾았는데 없음
            cout << "없음" << endl;
        else
            cout << dic[eng] << endl; // dic에서 eng의 값을 찾아 출력
    }
    cout << "종료합니다..." << endl;
}
```

# STL 알고리즘 사용하기

35

## □ 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 10-13 sort() 함수를 이용한 vector 소팅

36

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;
}
```

주목

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

# auto를 이용하여 쉬운 변수 선언

37

## □ 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 i; (X) 초기값을 줘야함*

*double* ← `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

38

## □ 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  
int
```

## ▣ 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;
```



vector<int>::iterator

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

# 예제 10-14 auto를 이용한 변수 선언

39

auto를 사용하여 변수를 선언하는 다양한 사례를 보인다.

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

int square(int x) { return x*x; }

int main() {
    // 기본 타입 선언에 auto 활용
    auto c = 'a';           // c는 char 타입으로 결정
    auto pi = 3.14;         // pi는 double 타입으로 결정
    auto ten = 10;          // ten은 int 타입으로 결정
    auto *p = &ten;         // 변수 p는 int* 타입으로 결정
    cout << c << " " << pi << " " << ten << " " << *p << endl;

    // 함수의 리턴 타입으로 추론
    auto ret = square(3);   // square() 함수의 리턴 타입이 int 이므로 ret는 int로 결정
    cout << *p << " " << ret << endl;

    vector<int> v = { 1,2,3,4,5 }; // 벡터 v에 5개의 원소, 1,2,3,4,5 삽입
    vector<int>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
        cout << *it << " "; // 1 2 3 4 5 출력
    cout << endl;

    // 템플릿에 auto를 사용하여 간소화
    for (auto it = v.begin(); it != v.end(); it++)
        cout << *it << " "; // 1 2 3 4 5 출력
}
```

~~vector<int> v::iterator it(X)~~

두 코드는  
동일

a 3.14 10 10  
10 9  
1 2 3 4 5  
1 2 3 4 5

# 람다

40

## □ 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

41

## □ C++의 람다식의 구성 익명 함수

### ▣ 4 부분으로 구성

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

캡처 리스트    매개변수 리스트    생략 가능    함수 바디

[ ] ( ) -> 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

42

## 예제 10-15 매개변수 $x$ , $y$ 의 합을 출력하는 람다식 만들기

매개변수  $x$ ,  $y$ 의 합을 출력하는 람다식은 다음과 같이 작성

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

$x$ 에 2,  $y$ 에 3을 전달하여 람다식이 바로 실행된다.

```
#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
}
```

합은 5

# auto로 람다식 저장 및 호출

43

## 예제 10-16 auto로 람다식 다루기

auto를 이용하여 변수 love에 람다식을 저장하고, love를 이용하여 람다식을 호출하는 사례이다.

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

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
}
```

람다식

돈보다 너가 좋아  
냉면보다 만두가 좋아

- \* 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

44

## 예제 10-17 반지름이 r이 원의 면적으로 리턴하는 람다식 만들기

지역 변수 pi의 값을 받고, 매개변수 r을 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
}
```

람다식

면적은 28.26

# 캡처 리스트에 참조를 활용하는 람다식

45

## 예제 10-18 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

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

    cout << "합은 " << sum;
}
```

합은 5

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

# 예제 10-19 STL for-each() 함수를 이용하여 벡터의 모든 원소 출력

46

STL에 들어 있는 for-each() 함수는 컨테이너의 각 원소를 검색하는 함수이며, 3번째 매개변수로 주어진 함수를 호출한다.

```
#include <iostream>
#include <vector>
#include <algorithm> // for_each() 알고리즘 함수를 사용하기 위함
using namespace std;

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

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

    // for_each()는 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);
}
```

it.begin() (X)  
v.begin() (O)

1 2 3 4 5

# STL 템플릿에 람다식 활용

47

## 예제 10-20 STL 함수 for\_each()와 람다식을 이용하여 벡터의 모든 원소 출력

STL에 들어 있는 for\_each() 함수는 컨테이너의 각 원소를 검색하는 함수이며, 3번째 매개변수로 주어진 함수를 호출한다.

```
#include <iostream>
#include <vector>
#include <algorithm> // for_each() 알고리즘 함수를 사용하기 위함
using namespace std;

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

    // for_each()는 벡터 v의 첫번째 원소부터 끝까지 검색하면서,
    // 각 원소에 대해 3번째 매개변수인 람다식 호출. 매개변수 n에 각 원소 값 전달
    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
}
```

1 2 3 4 5

람다식 호출.  
매개변수 n에는 벡터  
의 각 원소 전달