

## 2. 배열: C에서 C++로

### 1) C 스타일 배열과 `std::array`

# 배열

## ■ 배열(array)이란?

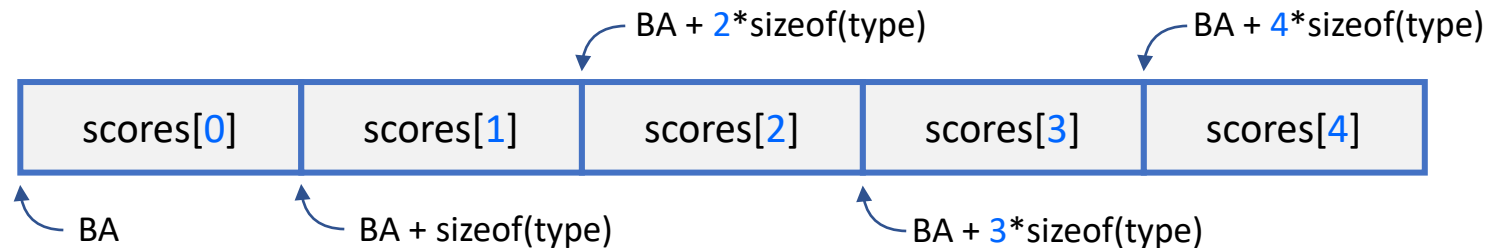
- 같은 종류의 데이터가 **연속적**으로 저장되어 있는 자료 구조

## ■ 다섯 학생의 점수를 저장하려면?

```
int score1, score2, score3, score4, score5;
```



```
int scores[5];
```



BA = 시작 주소(Base Address)

sizeof(type) = 원소 하나에 필요한 메모리 크기

# 배열

---

## ■ 배열의 특징

- **인덱스(index)**를 사용하여 원하는 **원소(element)**에 곧바로 접근 가능:  $O(1)$
- **캐시 지역성(cache locality)**
  - 배열의 각 원소는 서로 인접해 있기 때문에 하나의 원소에 접근할 때 그 근방에 있는 원소도 함께 캐시(cache)로 가져옴
- 반복문에서 배열을 사용하면 효율적인 프로그래밍이 가능
- 상수 또는 상수표현식으로 크기를 지정(크기 불변)
- 스택 메모리 영역에 할당 → 보통 1MB 할당

# C 스타일 배열

## ■ C 스타일 배열 예제 코드

```
#include <iostream>

using namespace std;

int main()
{
    int scores[5] = {50, 60, 70, 80, 90};

    int sz = sizeof(scores) / sizeof(scores[0]);
    // int sz = size(scores);

    int s = 0;
    for (int i = 0; i < sz; i++) {
        s += scores[i];
    }

    float m = (float)s / sz;

    cout << "Mean score: " << m << endl;
}
```



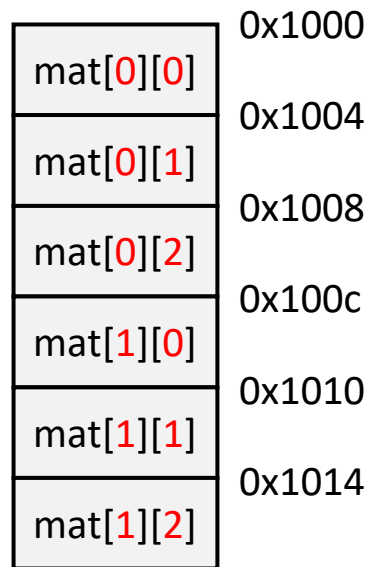
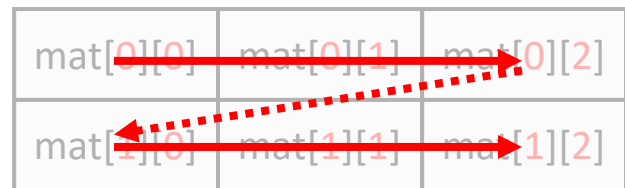
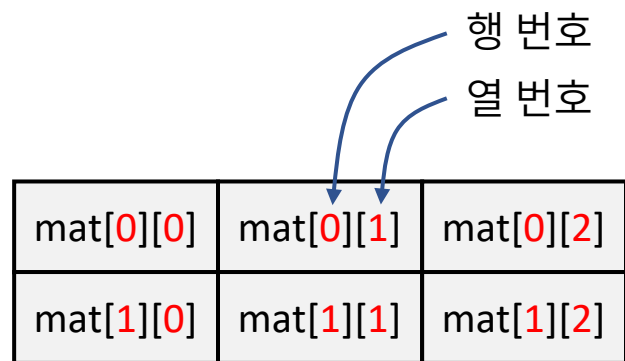
# C 스타일 배열

## ■ C 스타일 2차원 배열

- 2x3 행렬  $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$  의 원소 합?

```
int mat[2][3] = {  
    {1, 2, 3},  
    {4, 5, 6}  
};
```

```
int s = 0;  
for (int r = 0; r < 2; r++) {  
    for (int c = 0; c < 3; c++) {  
        s += mat[r][c];  
    }  
}
```



실제 메모리  
저장 순서  
(행 단위)

# std::array

## ■ std::array란?

- C++에서 C 스타일 배열을 대체하는 **고정 크기** 컨테이너 (C++11)
- 원소의 타입과 배열 크기를 매개변수로 사용하는 클래스 템플릿
- <array> 헤더 파일에 정의되어 있음

```
template<class T, std::size_t N>  
struct array;
```

```
template <class T, std::size_t N>  
class array  
{  
public:  
    T data[N];  
  
    T& operator[](int index)  
    {  
        return data[index];  
    }  
  
    ...  
};
```

[Pseudo-code]

# std::array

---

## ■ std::array 특징

- C 스타일 배열처럼 사용할 수 있는 [] 연산자 오버로딩을 제공
- 대입 연산자 지원 (깊은 복사)
- 배열 크기를 정확하게 알 수 있음. array::size().
- 반복자 지원

# std::array

## ■ std::array 예제 코드

```
#include <array>
#include <iostream>

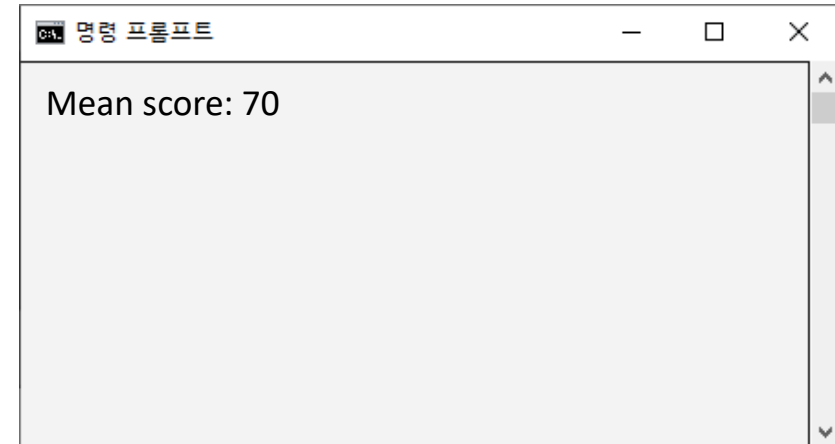
using namespace std;

int main()
{
    array<int, 5> scores = {50, 60, 70, 80, 90};

    int s = 0;
    for (int i = 0; i < scores.size(); i++) {
        s += scores[i];
    }

    float m = (float)s / scores.size();

    cout << "Mean score: " << m << endl;
}
```





# std::array

---

## ■ std::array 단점

- 배열의 크기를 명시적으로 지정해야 함
- 항상 스택 메모리를 사용
- 고정 크기 배열이 아닌 가변 크기 배열을 더 선호함

} → std::vector

## 2. 배열: C에서 C++로

### 2) 동적 배열과 `std::vector`

# 동적 배열

## ■ 동적 메모리 할당(dynamic memory allocation)

- 프로그램 실행 중 필요한 크기의 메모리 공간을 할당하여 사용하는 기법
- 동적으로 할당한 메모리는 사용이 끝나면 명시적으로 (할당된) 메모리를 해제해야 함
- C 언어: `malloc()` 또는 `calloc()` 함수로 메모리 할당하고, `free()` 함수로 메모리 해제
- C++ 언어: `new` 연산자로 메모리 할당하고, `delete` 연산자로 메모리 해제

## ■ 동적 메모리 할당을 이용한 동적 배열 생성 및 해제

- `new[]` 연산자를 이용하여 동적 배열을 위한 메모리를 할당하고, `delete[]` 연산자를 이용하여 동적 배열 메모리를 해제!

```
int* ptr = new int[3];  
// ptr을 배열처럼 사용. (e.g.) ptr[0], ptr[1], ptr[2]  
delete [] ptr;
```

- 동적 메모리 할당은 힙(heap) 메모리 영역을 사용하므로 대용량 배열도 할당 가능

# 동적 배열

## ■ 동적 배열 예제 코드

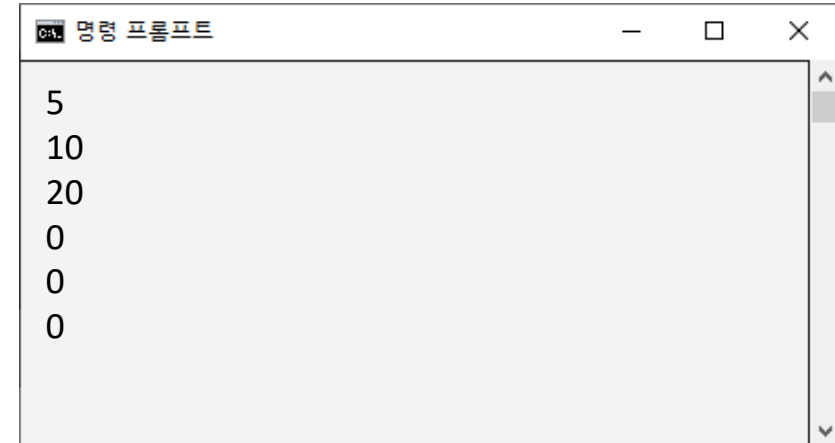
```
#include <iostream>

int main()
{
    int n;
    std::cin >> n;

    // 동적 배열 할당 및 초기화
    int* ptr = new int[n] {};
    ptr[0] = 10;
    ptr[1] = 20;

    // ptr을 배열처럼 사용
    for (int i = 0; i < n; i++) {
        std::cout << ptr[i] << std::endl;
    }

    // 동적 배열 해제
    delete [] ptr;
}
```



# 동적 배열

## ■ 메모리를 자동으로 해제하는 동적 배열 클래스

```
class DynamicArray
{
private:
    unsigned int sz;
    int* arr;

public:
    DynamicArray(int n) : sz(n) {
        arr = new int[sz] {};
    }

    ~DynamicArray() { delete [] arr; }

    unsigned int size() { return sz; }

    int& operator[] (const int i) { return arr[i]; }
    const int& operator[] (const int i) const { return arr[i]; }
};
```

```
int main()
{
    DynamicArray da(5);
    da[0] = 10;
    da[1] = 20;
    da[2] = 30;

    for (int i = 0; i < da.size(); i++)
        cout << da[i] << ", ";
    cout << endl;
}
```

# 동적 배열

## ■ 템플릿으로 구현한 동적 배열 클래스

```
template <typename T>
class DynamicArray
{
private:
    unsigned int sz;
    T* arr;

public:
    DynamicArray(int n) : sz(n) {
        arr = new T[sz] {};
    }

    ~DynamicArray() { delete [] arr; }

    unsigned int size() { return sz; }

    T& operator[] (const int i) { return arr[i]; }
    const T& operator[] (const int i) const { return arr[i]; }
};
```

```
int main()
{
    DynamicArray<int> da(5);
    da[0] = 10;
    da[1] = 20;
    da[2] = 30;

    for (int i = 0; i < da.size(); i++)
        cout << da[i] << ", ";
    cout << endl;
}
```

# std::vector

## ■ std::vector란?

- C++에서 C 스타일 배열을 대체하는 **가변 크기** 컨테이너
  - 초기화 과정에 데이터의 크기를 지정하지 않아도 됨
  - 배열의 크기를 확장할 수 있음
- 원소의 타입을 매개변수로 사용하는 클래스 템플릿
- <vector> 에 정의되어 있음

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

```
template <class T,
          class Allocator = std::allocator<T>>
class vector
{
private:
    T* data;
    unsigned int size;
    unsigned int capacity;
    ...
};
```

[Pseudo-code]

## 2. 배열: C에서 C++로

### 3) `std::vector` 사용법과 동작 방식



# std::vector 사용법과 동작 방식

## ■ std::vector 객체 생성과 원소 참조

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

using namespace std;

int main()
{
    // 벡터 생성
    vector<int> v1;           // int 값을 저장할 비어 있는 벡터 생성
    vector<int> v2(10);       // int 값 10개를 저장할 벡터 생성하고 0으로 초기화
    vector<int> v3(10, 1);    // int 값 10개를 저장할 벡터 생성하고 1로 초기화
    vector<int> v4 {10, 20, 30, 40, 50}; // 유니폼 초기화(uniform initialization)

    vector<int> v5(v4);       // v4를 복사하여 v5 생성
    vector<int> v6(v4.begin(), v4.begin() + 3); // v4의 처음 3개 원소를 이용하여 v6 생성

    // 벡터 원소 참조
    for (int i = 0; i < v6.size(); i++)
        cout << v6[i] << endl;
```

# std::vector 사용법과 동작 방식

- std::vector를 이용한 2차원 벡터 생성과 원소 참조

```
// 정수형 2차원 배열. 2행 3열. 0으로 초기화.
vector<vector<int>> mat1(2, vector<int>(3, 0));

// 유니폼 초기화
vector<vector<int>> mat2 {{1, 2, 3}, {4, 5, 6}};

// 2차원 벡터 출력
for (int r = 0; r < mat2.size(); r++) {
    for (int c = 0; c < mat2[r].size(); c++) {
        cout << mat2[r][c] << " ";
    }
    cout << endl;
}
```

# std::vector 사용법과 동작 방식

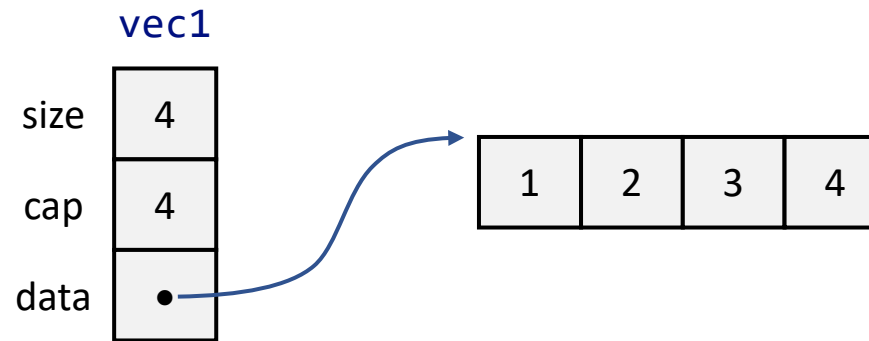
## ■ std::vector 주요 멤버 함수와 기능

멤버 함수	설명
operator []	특정 위치 원소의 참조를 반환
front()	첫 번째 원소의 참조를 반환
back()	마지막 원소의 참조를 반환
push_back()	마지막에 원소를 추가
emplace_back()	push_back()과 같지만 객체의 복제나 이동이 없어서 효율적
pop_back()	마지막 원소를 삭제 (마지막 원소를 반환하지 않음)
insert()	특정 위치에 원소를 삽입
erase()	특정 위치의 원소를 삭제
clear()	모든 원소를 삭제
size()	원소의 개수를 반환
empty()	벡터가 비어 있으면 true를 반환

# std::vector 사용법과 동작 방식

- std::vector::push\_back() 동작 방식

```
vector<int> vec1 {1, 2, 3, 4};
```

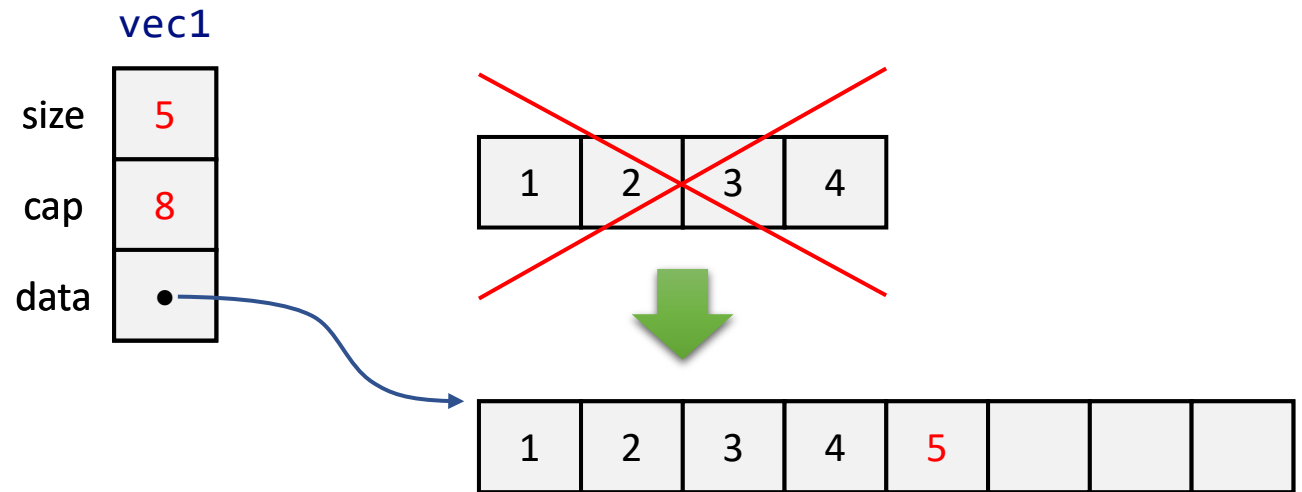


# std::vector 사용법과 동작 방식

## ■ std::vector::push\_back() 동작 방식

```
vector<int> vec1 {1, 2, 3, 4};
```

```
vec1.push_back(5);
```



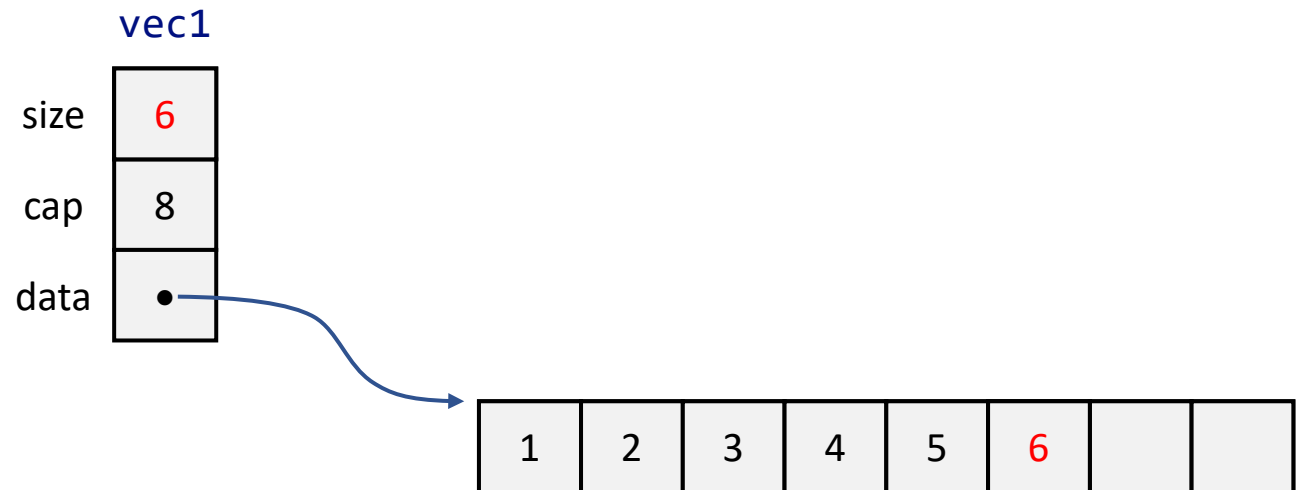
# std::vector 사용법과 동작 방식

## ■ std::vector::push\_back() 동작 방식

```
vector<int> vec1 {1, 2, 3, 4};
```

```
vec1.push_back(5);
```

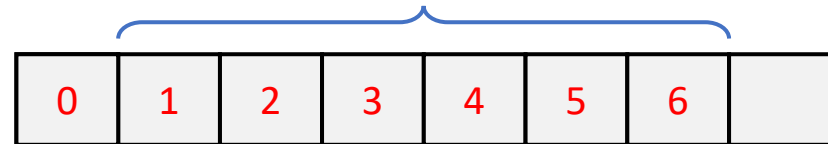
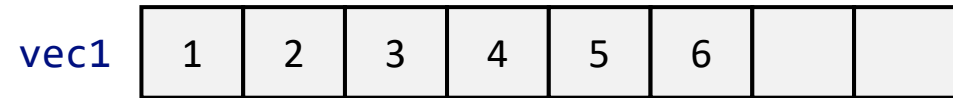
```
vec1.push_back(6);
```



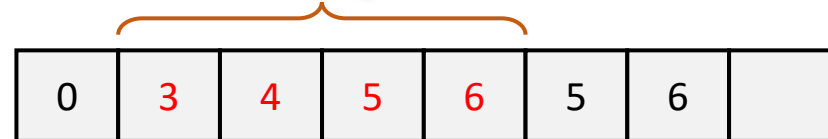
# std::vector 사용법과 동작 방식

- std::vector::insert()와 std::vector::erase() 동작 방식

```
vec1.insert(vec1.begin(), 0);
```



```
vec1.erase(vec1.begin() + 1, vec1.begin() + 3);
```



vec1.begin()

vec1.end()

size = 5

# std::vector 사용법과 동작 방식

## ■ std::vector 사용 예제

```
int main()
{
    vector<int> vec1 {1, 2, 3, 4};
    cout << vec1.capacity() << ":" << vec1.size() << endl;

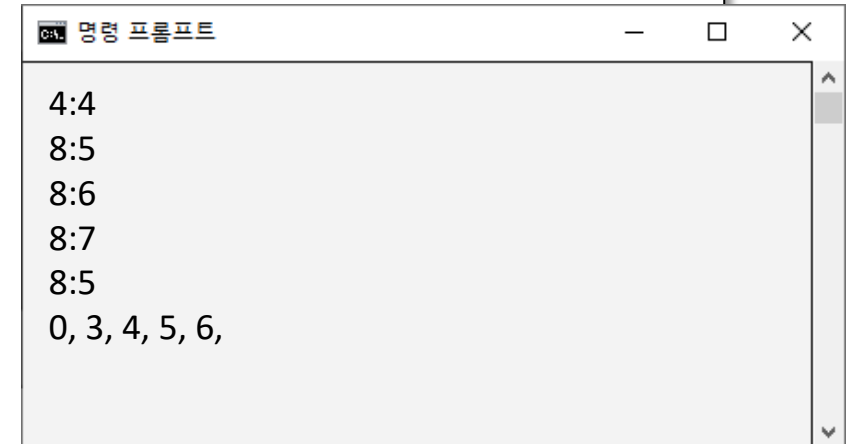
    vec1.push_back(5);
    cout << vec1.capacity() << ":" << vec1.size() << endl;

    vec1.push_back(6);
    cout << vec1.capacity() << ":" << vec1.size() << endl;

    vec1.insert(vec1.begin(), 0);
    cout << vec1.capacity() << ":" << vec1.size() << endl;

    vec1.erase(vec1.begin() + 1, vec1.begin() + 3);
    cout << vec1.capacity() << ":" << vec1.size() << endl;

    for (int i = 0; i < vec1.size(); i++)
        cout << vec1[i] << ", ";
}
```



명령 프롬프트

4:4  
8:5  
8:6  
8:7  
8:5  
0, 3, 4, 5, 6,



# std::vector 사용법과 동작 방식

## ■ std::vector 주요 멤버 함수와 기능

멤버 함수	설명	시간 복잡도
operator []	특정 위치 원소의 참조를 반환	O(1)
front()	첫 번째 원소의 참조를 반환	O(1)
back()	마지막 원소의 참조를 반환	O(1)
push_back()	마지막에 원소를 추가	O(1)
emplace_back()	push_back()과 같지만 객체의 복제나 이동이 없어서 효율적	O(1)
pop_back()	마지막 원소를 삭제 (마지막 원소를 반환하지 않음)	O(1)
insert()	특정 위치에 원소를 삽입	O(N)
erase()	특정 위치의 원소를 삭제	O(N)
clear()	모든 원소를 삭제	O(1)
size()	원소의 개수를 반환	O(1)
empty()	벡터가 비어 있으면 true를 반환	O(1)

## 2. 배열: C에서 C++로

### 4) 주요 모던 C++ 문법

# 주요 모던 C++ 문법

---

- auto
- using
- 범위 기반 for 문
- 람다 표현식
- chrono 라이브러리

# auto

## ■ 타입 추론 auto 키워드

- 변수 선언 시 타입 자리에 auto 키워드를 지정하면 컴파일 시간에 자동으로 타입을 추론하여 결정
- 복잡한 타입을 대체하거나 또는 범용적인 코드 작성 시 유용
- const 또는 레퍼런스(&) 속성을 사용해 auto를 한정할 수 있음

```
vector<int> odds()  
{  
    return { 1, 3, 5, 7, 9 };  
}  
  
auto a1 = 10;           // int  
auto a2 = 3.14f;        // float  
auto a3 = "hello";      // const char*  
auto a4 = "hello"s;     // std::string  
auto a5 = sqrt(2.0);    // double  
auto a6 = odds();       // vector<int>  
auto a7 = a6.begin();   // vector<int>::iterator  
auto lambda = [](int x) { return x * 2; };
```

# using

## ■ 타입 정의(aliasing)

- C언어 또는 C++11 이전이라면 typedef 를 사용하고, C++11 이후라면 using 사용을 권장
- using 문법이 읽기 더 쉽고, 템플릿 별칭(template alias)도 지원함

```
// C언어 또는 C++03
typedef unsigned char uchar;
typedef pair<int, string> pis;
typedef double da10[10];
typedef void (*func)(int);
```

```
void my_function(int n)
{
    //
}
```

```
// C++11 이후
using uchar = unsigned char;
using pis = pair<int, string>;
using da10 = double[10];
using func = void(*)(int);
```

```
template <typename T>
using matrix1d = vector<T>;
```

```
// 사용 예제
da10 arr {};
matrix1d<float> vec(3);
func fp = &my_function;
```

# 범위 기반 for 문

## ■ 범위 기반 for 문 (range-based for)

- 배열 또는 STL 컨테이너에 들어있는 모든 원소를 순차적으로 접근하는 C++11의 새로운 반복문
- 사용자 정의 타입에 대해 범위 기반 for 문을 지원하려면 `std::begin()`, `std::end()` 함수를 지원해야 함

```
vector<int> numbers {10, 20, 30};
```

```
for (int n : numbers)  
    cout << n << endl;
```

```
for (auto& n : numbers)  
    cout << n << endl;
```

```
string strs[] = {"I", "love", "you"};
```

```
for (const auto& s : strs)  
    cout << s << " ";  
cout << endl;
```

```
for (auto iter = begin(numbers); iter < end(numbers); ++iter) {  
    cout << *iter << endl;  
}
```

# 람다 표현식

- 람다 표현식(lambda expression)
  - C++11에서 지원하는 이름 없는 함수 객체
  - 함수의 포인터 또는 함수 객체(function object)를 대체

```
auto square = [](double a) { return a * a; };  
cout << "square(1.414) = " << square(1.414) << endl;
```

```
vector<Person> students;  
students.push_back({"Kim", 20});  
students.push_back({"Lee", 30});  
students.push_back({"Park", 24});  
students.push_back({"Choi", 40});
```

```
struct Person  
{  
    string name;  
    int age;  
};
```

```
sort(students.begin(), students.end(), [](const Person& p1, const Person& p2) {  
    return p1.name < p2.name;  
});
```

```
for (const auto& p : students)  
    cout << p.name << " : " << p.age << endl;
```

# chrono 라이브러리

## ■ chrono 라이브러리

- OS 독립적으로 정밀한 시간 측정 가능. 나노초(nano second) 단위까지 측정 가능
- 특정 연산 전후로 time\_point를 측정하고, time\_point 차이를 이용하여 실제 연산 시간을 계산  
→ 전체 타입을 모두 명시하여 코드를 작성하면 복잡하므로 `auto` 사용 예제 코드를 참고!
- 프로그램 동작 시간을 제대로 측정하려면 g++ 사용 시에는 `-O2` 옵션을 사용하고, Visual Studio 사용 시에는 `Release` 모드로 빌드해야 함
- <chrono>에 정의되어 있음

```
auto start = chrono::system_clock::now();

// Do something!

auto end = chrono::system_clock::now();
auto msec = chrono::duration<double>(end - start).count() * 1000;
cout << "Elapsed time: " << msec << "ms." << endl;
```