표준 템플릿 라이브러리[1]

- 컨테이너, 알고리즘, 반복자로 구성
  - 컨테이너: std::arrry, std::vector, std::list, std::set, std::map 등
  - 알고리즘: 정렬하거나, 개수를 세거나, 최소, 최대 원소 등을 찾기
  - 이터레이터: 컨테이너(데이터)와 알고리즘(행위)을 연결하는 포인터

[1] 라이너 그림 저 / 류광 역, 핵심 C++ 표준 라이브러리, 인사이트, 2017

컨테이너 공통 연산들

	std::vector 예	설명
생성	std::vector<int> vec1; std::vector<int> vec2(vec1); std::vector<int> vec3 {1, 2, 3};	기본 생성자 복사 생성자 초기화 리스트 생성자
지우기	vec1.clear();	원소(element)를 삭제
크기	vec1.empty(); vec1.size();	컨테이너가 비었는가? 원소(element)의 개수
접근	vec1.begin(), vec1.end() vec1.cbegin(), vec1.cend() vec1.rbegin(), vec1.rend()	처음부터 끝까지 상수 반복자 끝에서부터 처음까지

원소할당, 이동	vec1=vec2;
	vec1=std::move(vec2);
	vec1={10, 20, 30}
		할당(assign)
		이동(move)
		초기화 리스트

비교	vec1 == vec2 vec1 != vec2	비교 연산자

(추가) std::arrary

- 정해진 크기의 배열을 위한 순차 컨테이너
  - 메모리에서 원소들은 고정된 크기의 연속적인 공간에 저장됨
  - 메모리를 자동으로 할당하고 해제함
  - 깊은 비교(deep comparison), 깊은 복사(deep copy) 연산자 지원

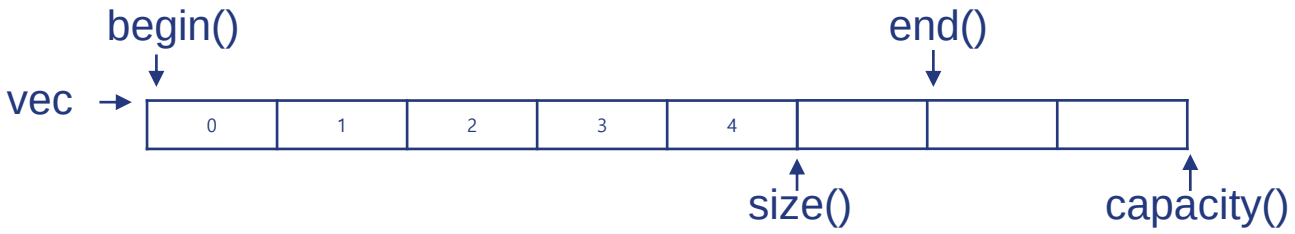
```
std::array<int, 5> arr1 {{5, 4, 3, 2, 1}};  //C++11 prior
std::array<int, 5> arr2 = {1, 2, 3, 4, 5};  //C++11 after revision
std::array arr3 {3.0, 1.0, 4.0};            //C++17

std::cout << arr1.size() << std::endl;
std::cout << arr2.size() << std::endl;
for (const auto& i : arr1)
    std::cout << i << ' '; // 5 4 3 2 1

for ( auto i = arr2.begin() ; i != arr2.end() ; ++i )
    std::cout << *i << '\t' ; // 1, 2, 3, 4, 5
```

https://en.cppreference.com/w/cpp/container/array

std::vector

- 동적 배열을 캡슐화한 순차 컨테이너
  - 메모리에서 원소들은 연속해서 저장
  - 저장공간이 자동으로 확장
  - 추가적인 원소들을 저장하기 위해 크기보다 더 큰 공간 차지

```
vector<int> vec {0, 1, 2, 3};
vec.push_back(4);
std::cout << "size: " << vec.size();                              //5
std::cout << ", capacity: " << vec.capacity() <<std::endl;       //8
for ( auto i = vec.begin() ; i != vec.end() ; ++i )
    cout << *i << '₩t' ; // 0 1 2 3 4
```



https://en.cppreference.com/w/cpp/container/vector

vector<T>의 사용

vector의 생성

```
int intSize = 5 ;
// intSize 크기의 int 타입 vector의 생성
// 각 원소는 int의 기본 값인 0 으로 초기화 됨
vector<int> vInt1(intSize) ;
for ( unsigned int i = 0 ; i < vInt1.size() ; i ++ )
    cout << vInt1[i] << '₩t' ; // 0 0 0 0 0
```

배열을 이용한 초기화

```
int intA[3] = {10, 20, 30} ;
vector<int> vInt2(intA, intA+3) ;
//크기 3인 int vector가 {10, 20, 30}으로 초기화
```

vector<T>의 사용

```
#include <vector>     // vector<T>를 활용하기 위해서는 <vector>를 포함해야 함
#include <iostream>
using namespace std ;

int main() {
  cout << "크기를 입력하시오." << endl ;
  int intSize ;
  cin >> intSize ;
  // vector 정의
  vector<int> vInt1(intSize) ;              // intSize 크기의 int 타입 vector의 생성
  for ( unsigned int i = 0 ; i < vInt1.size() ; i ++ ) { // size() 함수를 이용한 크기
    vInt1[i] = i ;                          // [ ] 연산자를 이용한 원소의 접근
    cout << vInt1.at(i) << '₩t' ;           // at() 함수를 이용한 원소의 접근
  }
  cout << endl ;
  // 배열을 이용하여 vector의 생성및 초기화
  int intA[3] = {10, 20, 30} ;
  // 크기 3인 int vector를 {10, 20, 30}으로 초기화
  vector<int> vInt2(intA, intA+3) ;
  for ( unsigned int i = 0 ; i < vInt2.size() ; i ++ )
    cout << vInt2.at(i) << '₩t' ;
  cout << endl ;
}
```

vector<T>의 크기 조정

- 필요하다면 공간을 확장

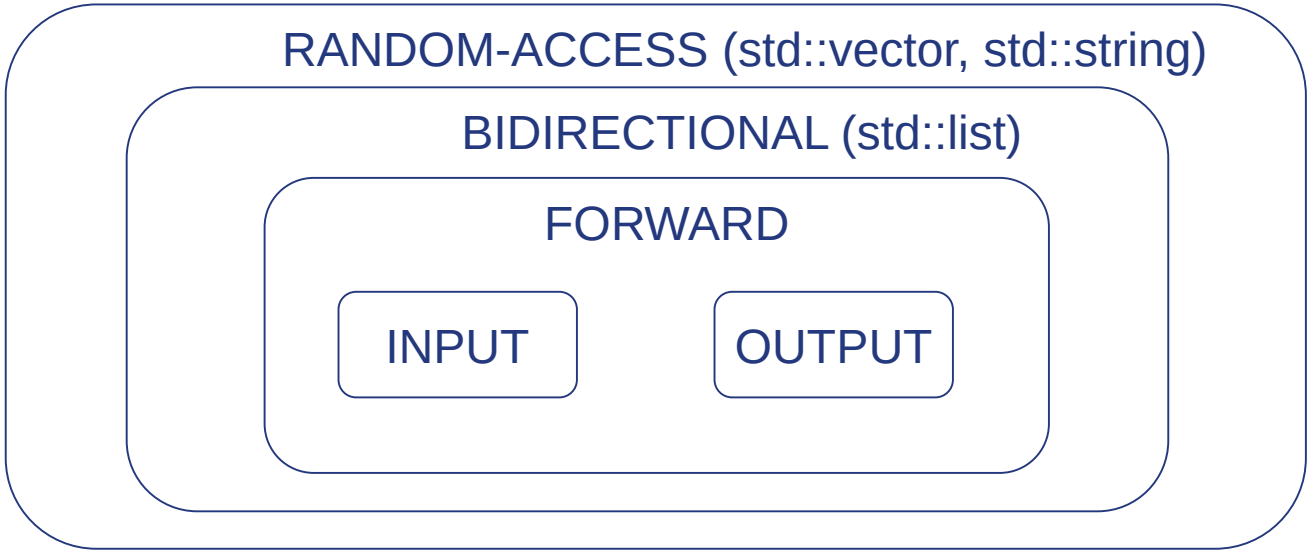
```
#include <iostream>
#include <vector>
using namespace std;
int main () {
  vector<int> vInt ;
  std::cout << vInt.empty() << " " << vInt.size() << " " << vInt.capacity() <<std::endl;//1 0 0
  for ( unsigned i=1; i<10; i++ ) {
    vInt.push_back(i);          // 1 2 3 4 5 6 7 8 9
    std::cout << vInt.size() << " " << vInt.capacity() <<std::endl;//9 16
  }
  vInt.resize(3);               // 1 2 3, size: 3, capacity: 16
  vInt.shrink_to_fit();         // size: 3, capacity: 3
  for ( unsigned i=0; i < vInt.size(); i++ )
    cout << " " << vInt[i]; // 1 2 3
  cout << endl;
}
```

std::vector 멤버 함수

	멤버 함수	설명
접근	std::vector<int> vec; vec.front() vec.back() vec[i] vec.at(i) vec.data()	 첫번째 원소(element) 마지막 원소 i^{th} 원소 i^{th} 원소 백터 내부 배열의 포인터
수정	vec.insert(위치, 값) vec.emplace_back(값) vec.push_back(값) vec.erase(위치) vec.pop_back()	원소를 입력 백터 끝에 직접 원소를 생성 백터 끝에 원소 추가 원소를 삭제 백터의 마지막 원소를 삭제

std::iterator

- 포인터를 일반화 한것
  - 컨테이너 안의 한 위치를 나타냄
  - 컨테이너 안의 원소들을 순회 혹은 반복하는데 사용
  - 컨테이너와 알고리즘을 연결하는 중요한 역할
- 반복자 범주(category)



https://www.geeksforgeeks.org/introduction-iterators-c/

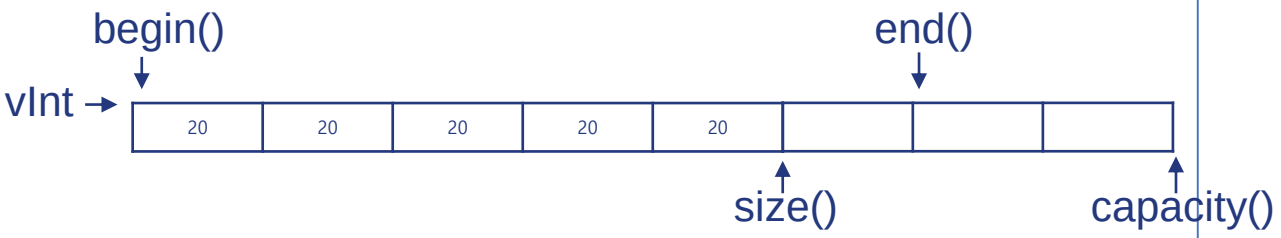
이터레이터 연산들

	input	output	forward	bidirectional	random
access	->		->		->, []
read	=*it		=*it		=*it
write		*it=	*it=		*it=
iterate	++	++	++	++, --	++, -- +=, -= +, -
compare	==, !=		==, !=	==, !=	==, !=, <, > <=, =>

iterator 의 사용

- iterator는 vector, set, list, map 등의 컨테이너에 저장된 각 원소를 접근하기 위한 클래스

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
  vector<int> vInt(5) ; // int 타입 크기 5의 vector 정의
  for ( vector<int>::iterator it = vInt.begin() ; it != vInt.end() ; ++ it ) {
     *it = 20 ; // 각 원소의 값을 지정함
     cout << *it << '₩t' ;  // 20
  }
  cout << endl;
}
```




```
// range-based for since C++11
for ( int &v: vInt ) {
     v = 20;
     cout << v << '₩t';
}
```

상수 iterator

- const_iterator는 원소의 값을 변경할 수가 없음
- 상수를 접근할 때 사용됨

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
  vector<int> vInt2{10, 20, 30, 40, 50};
  // 출력에 사용하므로 const_iterator를 사용함
  for ( vector<int>::const_iterator it = vInt2.cbegin() ; it != vInt2.cend() ; ++ it ) {
      cout << *it << '₩t' ; // 10 20 30 40 50
      // it는 const_iterator이므로 *it의 값을 변경할 수 없다.
      // *it = 10 ; 또는 cin >> *it는 불허
  }
  cout << endl;
}
```

```
for ( const int& v: vInt2 )
        cout << v << '₩t';
```

반복자 관련 함수들

	함수	설명
연산 (operations)	std::next(it) std::prev(it) std::advance(it, n) std::distance(it1, it2)	it의 이전 위치를 가리키는 반복자 it의 다음 위치를 가리키는 반복자 it을 n만큼 증가시킨다 두 반복자 사이의 원소의 개수
접근 (access)	std::begin(컨테이너) std::end(컨테이너) std::rbegin(컨테이너) std::end(컨테이너) std::cbegin(컨테이너) std::cend(컨테이너)	컨테이너의 시작을 가리키는 반복자 컨테이너의 끝을 가리키는 반복자 컨테이너의 역방향 시작을 가리킴 컨테이너의 역방향 끝을 가리킴 컨테이너의 시작 상수 반복자 컨테이너의 끝 상수 반복자

알고리즘

- 컨테이너의 원소에 동작하는 다양한 기능들이 정의됨
  - 검색(searching), 정렬(sorting), 세기(counting), 변환(transform) 등
  - 범위(range)는 [first, last)로 정의 (예, vecto의 begin(), end() 범위)
- 알고리즘 분류
  - Non-modifying sequence operations
  - Modifying sequence operations
  - Partitioning operations
  - Sorting operations
  - Binary search operations
  - Set operations
  - Minimum/maximum operations
  - Comparison operations
  - Permutation operations

https://en.cppreference.com/w/cpp/algorithm
http://www.fluentcpp.com/getthemap/

반복자 관련 함수들

	함수	설명
원소를 수정하지 않는 알고리즘	find(시작, 끝, 값) count(시작, 끝, 값)	컨테이너의 범위에서 값을 찾는다 범위에서 값의 개수를 센다
원소를 수정하는 알고리즘	replace(시작, 끝, 찾는 값, 수정값) remove(시작, 끝, 값) fill(시작, 끝, 값) reverse(시작, 끝) random_suffle(시작, 끝) unique(시작, 끝) sort(시작 끝) max_element(시작, 끝)	범위에서 찾는값을 수정값으로 변경한다 범위에서 값을 삭제한다 범위를 값으로 채운다 범위에서 값의 순서를 뒤집는다 범위에서 값을 임의 순서로 섞는다 범위에서 중복 값을 제거한다 범위의 값을 정렬한다 범위의 값 중 최댓값을 돌려준다

STL algorithm 함수의 사용

```
#include <iostream>
#include <vector>
#include <algorithm> // sort, max_element, random_shuffle
using namespace std;
int main () {
  vector<int> vInt{1, 2, 3, 4, 5}; // list initialization

  random_shuffle( vInt.begin(), vInt.end() );
  for ( const int& v: vInt )                    // range-based for loop
        cout << v << '₩t' ;
  cout << endl ;

  sort( vInt.begin(), vInt.end() );

  // auto의 경우 const를 위해서 cbegin(), cend()를 사용함
   for ( auto it=vInt.cbegin(); it != vInt.cend(); ++it ) // range-based for loop
         cout << *it << '₩t' ;
  cout << endl;

  auto largest = max_element( vInt.cbegin(), vInt.cend()-1 );
  cout << "최대값: " << *largest << endl ; // 4
  cout << "위치: " << largest - vInt.cbegin() << endl ; // 3
}
```

std::string[1]

- 문자들의 순차열(sequence of characters)을 표현하는 클래스
  - 문자열을 분석하거나 변경하는 다양한 수단을 제공
  - 문자형식, 문자 특질(trait), 할당자를 템플릿 매개변수로 받음

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>,
    class Allocator = std::allocator<CharT>
> class basic_string;
typedef std::basic_string<char> string;
```

[1] 라이너 그림 저 / 류광 역, 핵심 C++ 표준 라이브러리, 인사이트, 2017

std::string 멤버 함수

	std::string 예	설명
생성	std::string str1; std::string str2(str1);	기본 생성자 복사 생성

변환	str1.data(); str1.c_str();
	std::to_string();
	std::stoi(str); std::stod();
		C 문자열을 돌려줌
		숫자를 문자열로 변환
		문자열을 숫자로 변환

크기	str1.empty();
	str1.size(); str1.length();
	str1.capacity();
		빈 문자열인가?
		문자열 길이
		재할당 없이 저장 가능한 문자열 길이

원소할당	str1 = "hello"; str2 = "hello"s;	C 문자열 할당 C++ 문자열 할당
원소접근	임의접근(random-access) 반복자	
비교	==, !=, <, >, <=, >=	비교 연산자
연결	std::string("a") + "b"; **"a" + "b";**	C++ 문자열과 C 문자열 연결 가능 **C 문자열과 C 문자열 연결은 에러**
찾기	str1.find(검색문자(열)); str1.rfind(검색문자(열));	검색 문자(열)이 str1에 처음 나타난 위치 검색 문자(열)이 str1에 마지막 나타난 위치
수정	str1.push_back() str.pop_back()	str1의 마지막에 문자를 추가함 str1의 마지막 문자를 제거함

STL string 클래스

```
#include <string>          // <string>을 반드시 포함해야 한다.
#include <iostream>
using namespace std ;
int main() {
    // 초기화
    string strGreeting = "Hello, C++ World !" ;
    // 출력: cout를 이용
    cout << strGreeting << endl ;
    // 길이 조회: length() 멤버 함수, 문자 접근: [ ] 연산자
    for ( unsigned int i = 0 ; i < strGreeting.length() ; i ++ )
        cout << strGreeting[i] ;
    cout << endl ;
    // 입력: cin을 이용
    string strVal1, strVal2 ;
    cin >> strVal1 >> strVal2 ;
    // 비교 vs compare() 멤버 함수, strVal1.compare(strVal2) == 0
    if ( strVal1 == strVal2 )
        strGreeting += " Same." ; // 문자열 병합: += 연산자
    else
        strGreeting += " Different." ;
    cout << strGreeting  << endl ;
}
```

```
#include <vector>
#include <string>       // string을 사용하기 위함
#include <sstream>      // stringstream을 사용하기 위함
#include <iostream>
using namespace std ;

int main() {
  cout << "크기를 입력하시오." << endl ;
  int intSize ;
  cin >> intSize ;

  vector<string> vString(intSize) ; // string의 vector 생성
  for ( unsigned int i = 0 ; i < vString.size() ; i ++ ) {
    stringstream intStringStream ;
    intStringStream << i ; // 정수 i
    string val ;
    intStringStream >> val ; // 정수 i의 값이 문자열로 변환되어 val에 저장됨

    vString[i] = val ; // vString[i] 는 string 임
    cout << vString[i] << '₩t' ;
  }
  cout << endl ;
}
```

(추가) std::pair

- pair는 두 값을 그룹으로 묶는 클래스 템플릿 (세개 이상은 std::tuple 사용)
- 두 값은 서로 다르게 지정 가능하며, first, second로 접근 가능

```
#include <utility>
#include <string>
#include <iostream>
int main(){
    std::pair<std::string, int> pair1 {"hello, world", 12};
    std::cout << myPair.first << ", " << myPair.second << std::endl;

    auto pair2 = std::make_pair("hello, world", 12);
    std::cout << pair2.first << ", " << pair2.second << std::endl;

    auto pair3 = std::pair {"hello, world", 12};
    std::cout << pair3.first << ", " << pair3.second << std::endl;
}
```

(추가) std::map

- 키(key)와 값(value)의 쌍으로 데이터를 저장
  - 키를 기준으로 데이터를 추가, 조회, 삭제 등을 수행함
  - 키 값을 기준으로 저장된 데이터를 정렬된 상태로 유지

```
#include <map>
int main() {
    std::map<std::string, int> map1 = {{"kim", 4}, {"lee", 2}};
    map1.insert({"park", 1});
    auto result = map1.insert({"choi", 3});
    if(result.second) std::cout << "The data is added successfully!" << std::endl;
    result = map1.insert_or_assign("choi", 0);
    if(result.second) std::cout << "The data is updated or added successfully!" << std::endl;
    for (auto iter = map1.cbegin(); iter != map1.cend(); ++iter)
        std::cout << iter->first << ", " << iter->second << std::endl;
    for (const auto& it: map1)
        std::cout << it.first << ", " << it.second << std::endl;
}
```

(추가) std::list

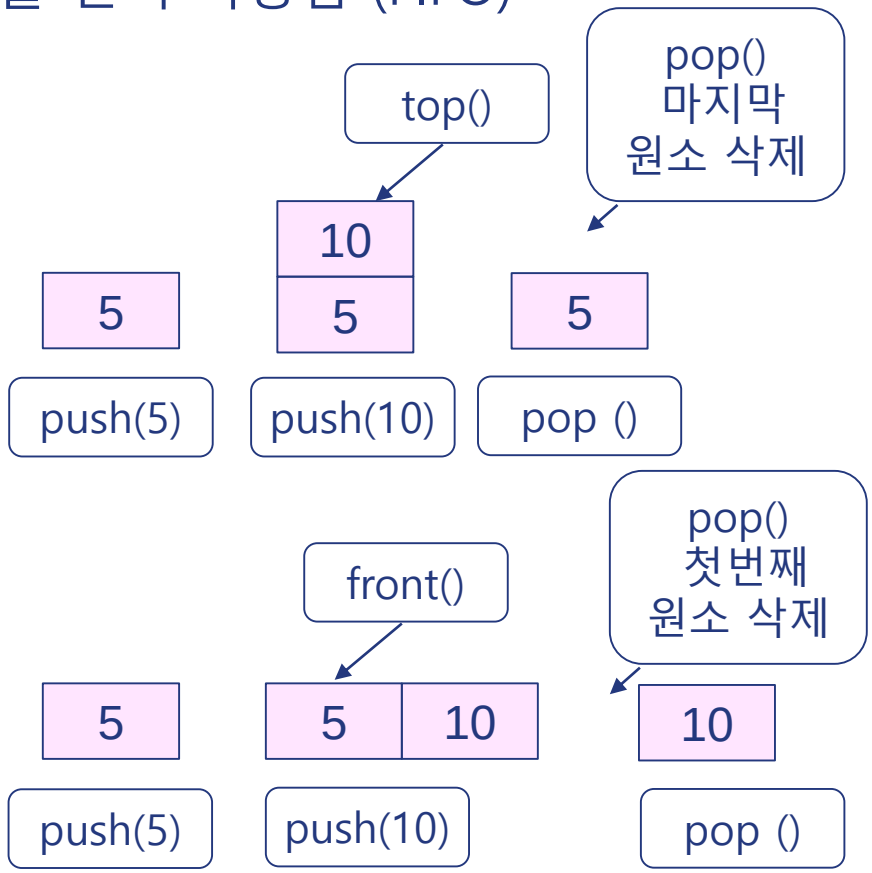
- 컨테이너의 어느 위치에 상수 시간에 원소를 추가하거나 삭제할 수 있음
- 내부적으로 이중 연결 리스트 구조로 이뤄져 있음
- 리스트의 끝에 원소를 추가, 이터레이터를 이용한 전방/역방향 이동, 리스트 크기 등의 기능을 제공함

```
#include <list>
int main() {
    std::list<int> lst = {1, 2, 3};
    lst.push_back(4);
    lst.insert(std::next(lst.begin()), 0);
    lst.insert(lst.end(), 5);

    for(const auto& it: lst)
        std::cout << it << std::endl;
}
```

(추가) Container Adapter

- 기존 컨테이너 클래스를 이용해서 만든 컨테이너들
  - std::stack (스택) : 가장 마지막에 저장된 원소를 먼저 사용함 (LIFO)
  - std::queue(큐) : 먼저 저장된 원소를 먼저 사용함 (FIFO)

```
#include <stack>
#include <queue>
int main() {
    std::stack<int> stk;
    stk.push(5); stk.push(10); //top에 원소 입력
    int a = stk.top(); stk.pop();
    std::cout << a << std::endl; //10 출력

    std::queue<int> q;
    q.push(5); q.push(10); //끝에 원소 입력
    int b = q.front(); q.pop();
    std::cout << b << std::endl; //5 출력
}
```



(추가) std::optional (C++17)

- 어떤 값이 존재하거나 그렇지 않음을 표현
- 함수의 반환값이 없다는 것을 nullptr, -1 등과 같은 특수한 값으로 표현하지 않아도 됨

```
#include <optional>
#include <string>
#include <iostream>
std::optional<int> getData(int i){
    if (i<0) return {};  //return std::nullopt;    이것도 가능
    return i;
}
int main(){
    auto data1 = getData(1);
    auto data2 = getData(-1);

    if(data1) std::cout << "optional<int>: " << data1.value() << std::endl;
    if(data2) std::cout << "optional<int>: " << data2.value() << std::endl;
    //std::cout << data2.value(); //bad optional access
    std::cout << data2.value_or(0) << std::endl;
}
```

(추가) std::variant (C++17)

- 타입에 안전한 공용체(union)
- 개발자가 지정한 타입 집합 중에서 하나의 타입 값을 가짐

```
#include <variant>
struct Visitor {
    void operator()(int i) {std::cout<<"int";}
    void operator()(std::pair<int, int>) {std::cout<<"pair";}
    void operator()(const std::string& s) {std::cout<<"string";}
};
int main(){
    std::variant<int, std::pair<int, int>, std::string> v; //세가지 타입 지정
    v = 3;
    int i1 = std::get<int>(v);
    int i2 = std::get<0>(v);
    //auto i3 = std::get<float>(v);         //compile error
    //auto i4 = std::get<std::string>(v);   //bad_variant_access
    std::visit(Visitor(), v);
    v = std::pair {4, 5};
    v = "hello";
}
```