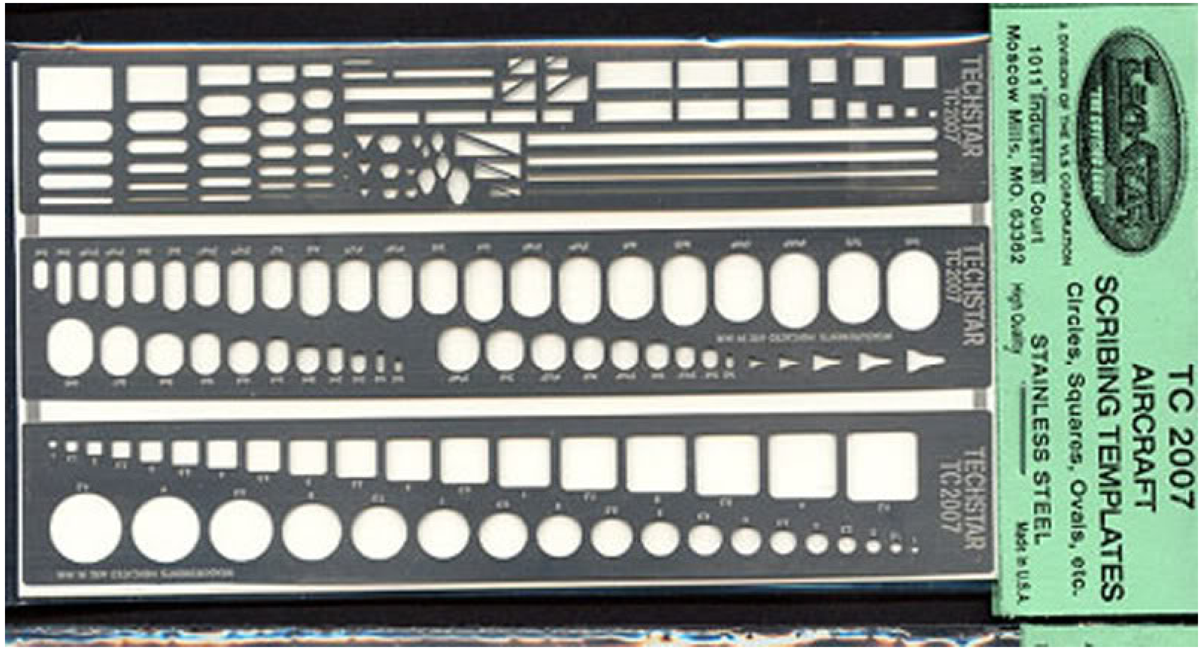
Template

2023

국민대학교 소프트웨어학부

템플릿이란?

- 템플릿(template): 물건을 만들 때 사용되는 틀이나 모형을 의미
- 함수 템플릿(function template): 함수를 찍어내기 위한 형틀



함수 get_max()

```
int get_max(int x, int y)
{
   if( x > y ) return x;
   else return y;
}
```

만약 float 값중
에서 최대값을
구하는 함수가
필요하다면?

함수 get_max()

```
float get_max(float x, float y)
{
   if( x > y ) return x;
   else return y;
}
```

핵심적인 내용은 같고 매개 변수의 타입만 달라진다.

일반화 프로그래밍 (generic programming)

- 일반적인 코드를 작성하고 이 코드를 다양한 타입의 객체에 대하여 재사용하는 프로그래밍 기법

템플릿 함수

```
____ get_max(___x , ___ y)
{
  if( x > y)    return x;
  else return y;
}
```

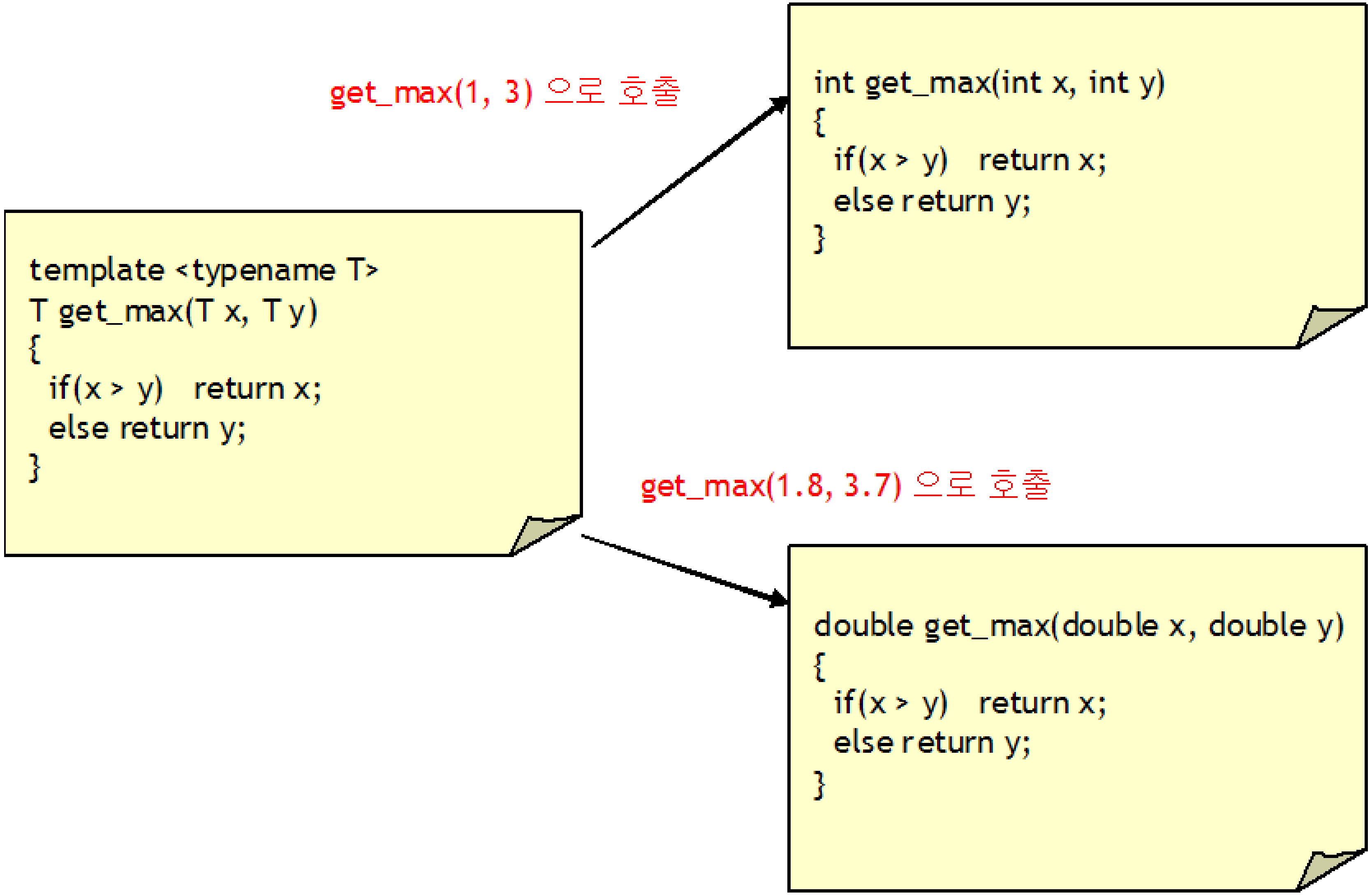
```
int get_max(int  x , int y)
{
  if( x > y)    return x;
  else return y;
}
```

get_max()

```
template<typename T>
T get_max(T x, T y)
{
   if( x > y ) return x;
   else return y;
}
```

자료형이 변수처럼 표기되어 있음을 알 수 있다

템플릿 함수의 사용



```
template <typename T>
T get_max(T x, T y)
{
  if(x > y)   return x;
  else return y;
}
```

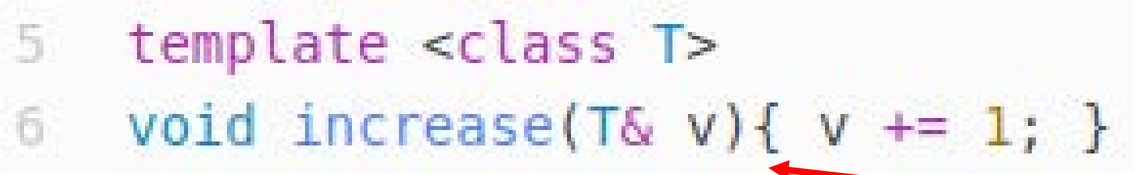
get_max(1, 3) 으로 호출

```
int get_max(int x, int y)
{
  if(x > y)   return x;
  else return y;
}
```

get_max(1.8, 3.7) 으로 호출

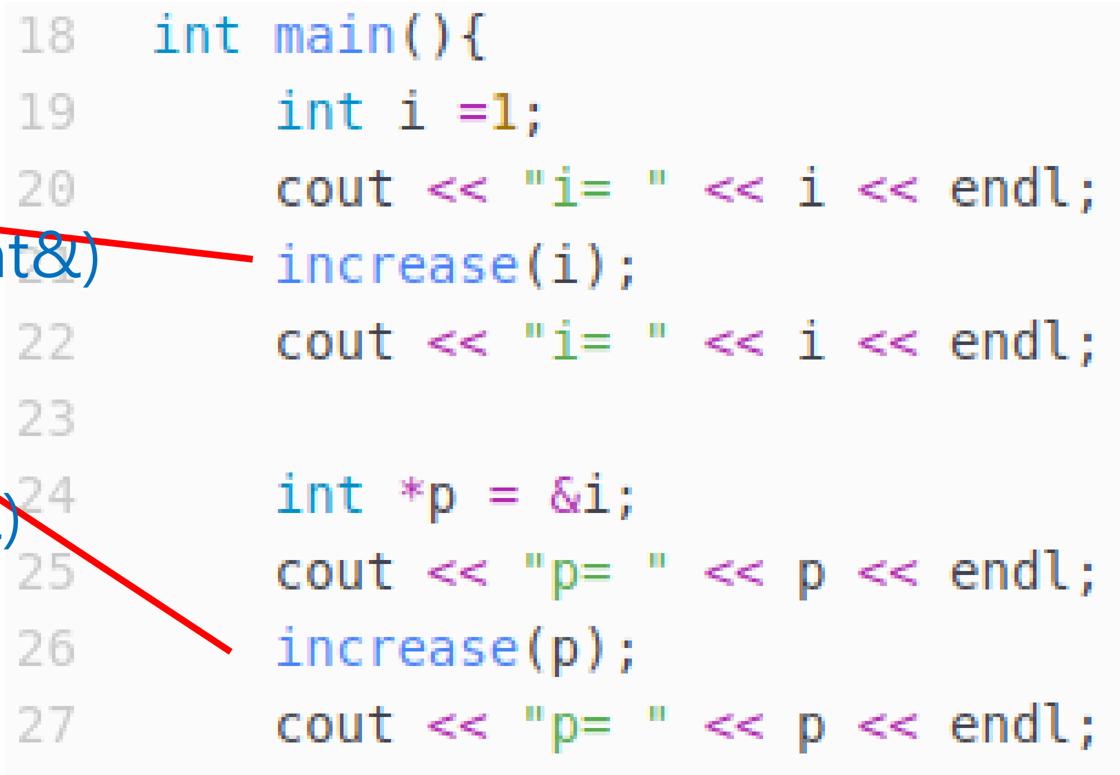
```
double get_max(double x, double y)
{
  if(x > y)   return x;
  else return y;
}
```

템플릿 함수



```
template <class T>
void increase(T& v){ v += 1; }
```

```
void increase(int &v) { v += 1; }
```

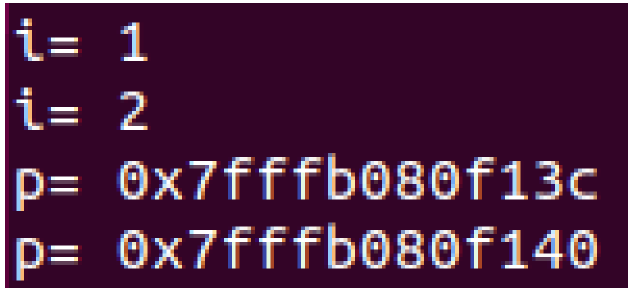


```
int main(){
    int i =1;
    cout << "i= " << i << endl;
    increase(i);
    cout << "i= " << i << endl;

    int *p = &i;
    cout << "p= " << p << endl;
    increase(p);
    cout << "p= " << p << endl;
```

increase(int&)

increase(int *&)

```
void increase(int *&v) { v += 1; }
```



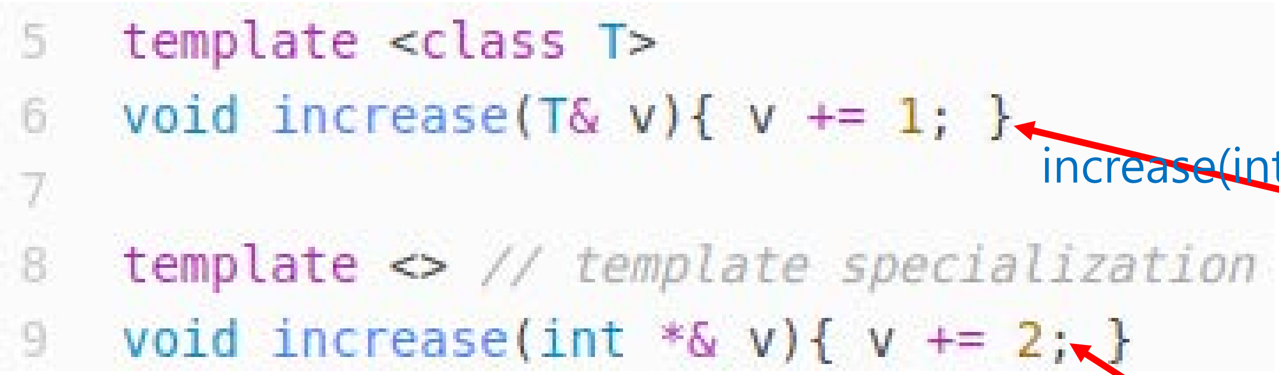
```
i= 1
i= 2
p= 0x7fffb080f13c
p= 0x7fffb080f140
```

+4x1

```
(참고) int &*v   // 컴파일러 오류
```

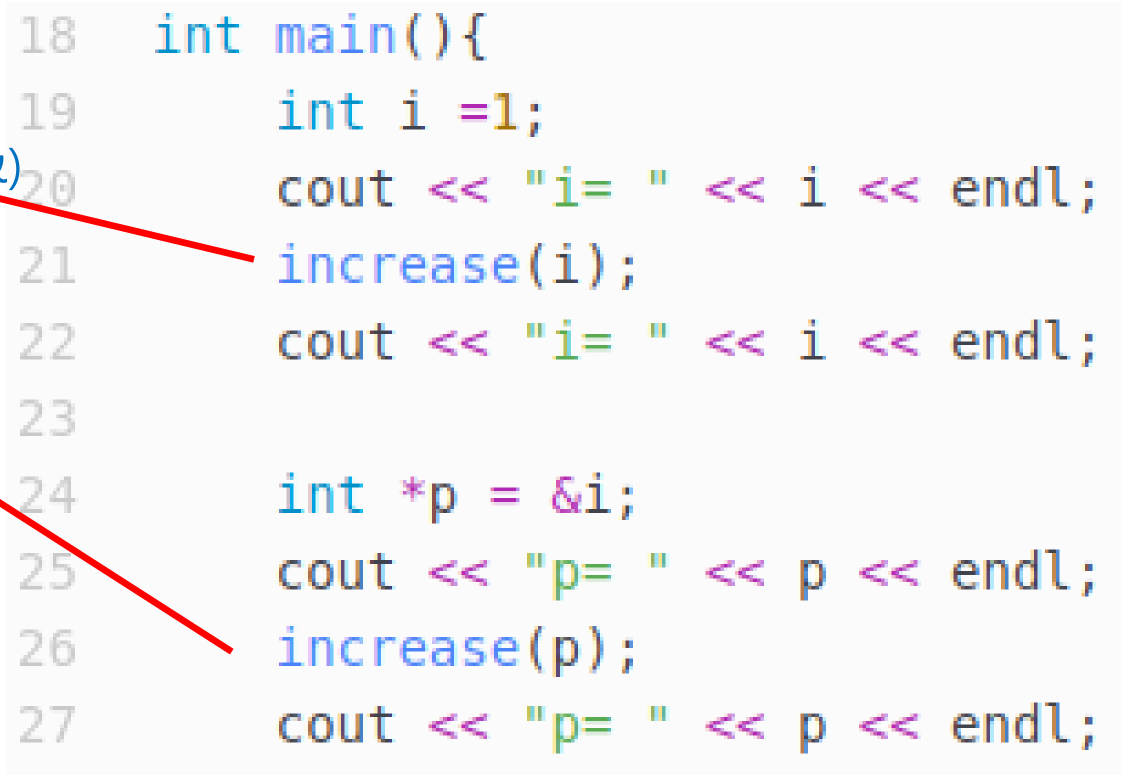
```
int n[4] = { 1, 2, 3, 4 };
int* p = n;

increase(p);
cout << *p;}
```

템플릿 함수의 특수화

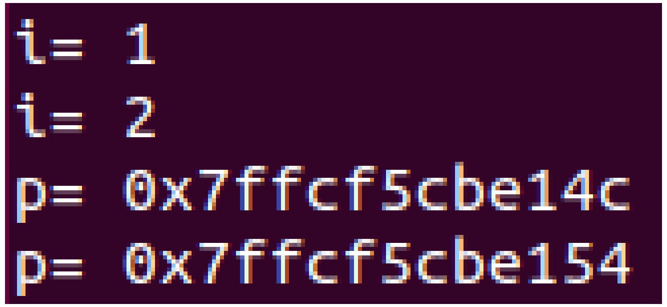


```cpp
template <class T>
void increase(T& v){ v += 1; }

template <> // template specialization
void increase(int *& v){ v += 2; }
```

increase(int&)



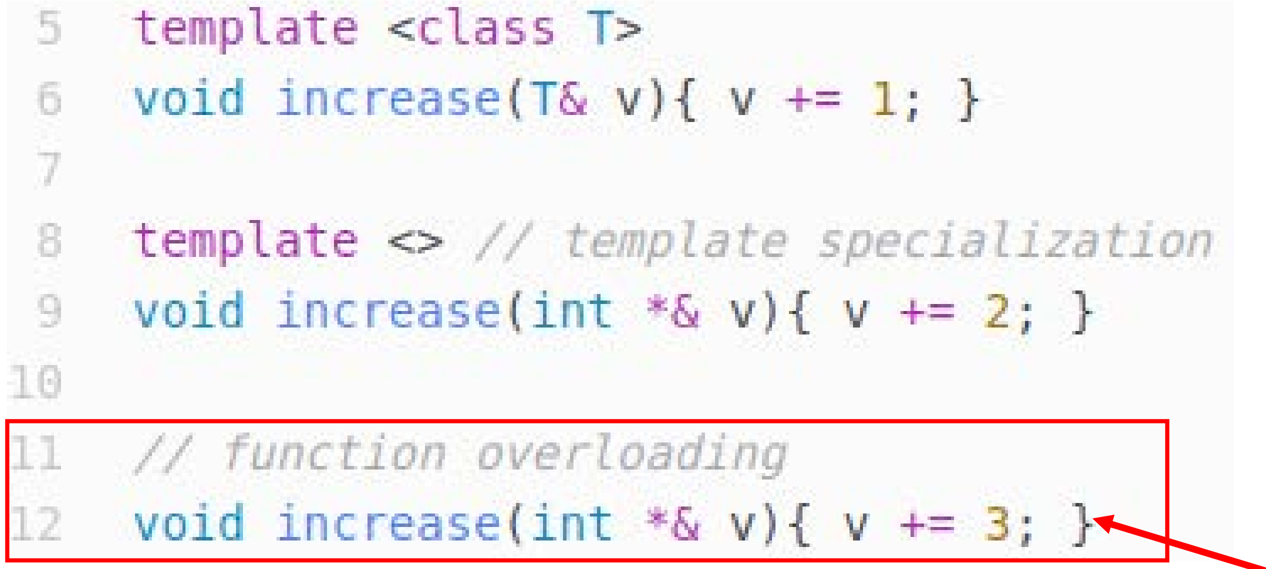
```cpp
int main(){
    int i =1;
    cout << "i= " << i << endl;
    increase(i);
    cout << "i= " << i << endl;

    int *p = &i;
    cout << "p= " << p << endl;
    increase(p);
    cout << "p= " << p << endl;
```



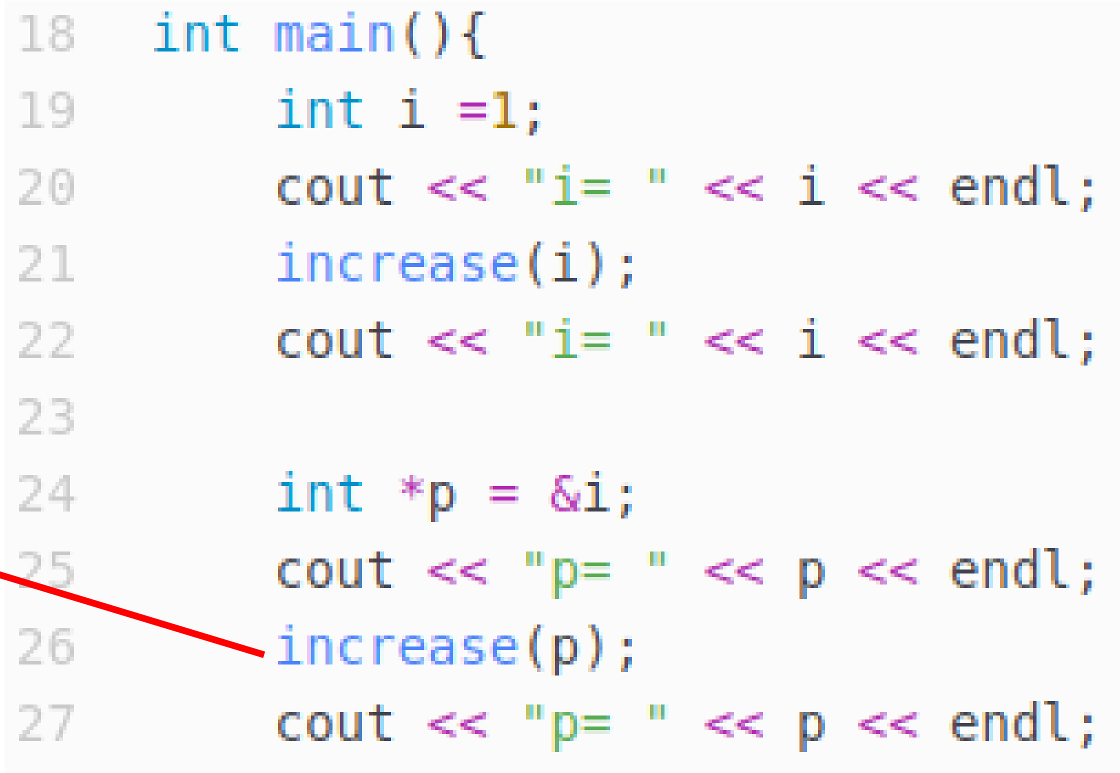
```
i= 1
i= 2
p= 0x7ffcf5cbe14c
p= 0x7ffcf5cbe154
```

+4x2

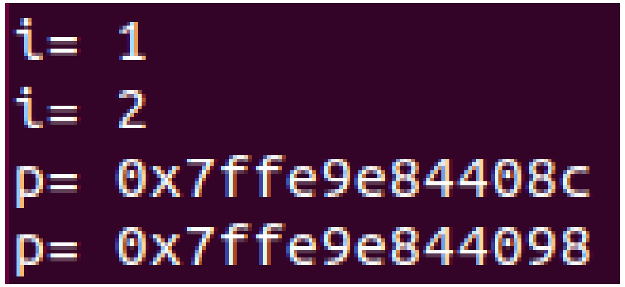
함수 템플릿과 함수 중복



```cpp
template <class T>
void increase(T& v){ v += 1; }

template <> // template specialization
void increase(int *& v){ v += 2; }

// function overloading
void increase(int *& v){ v += 3; }
```



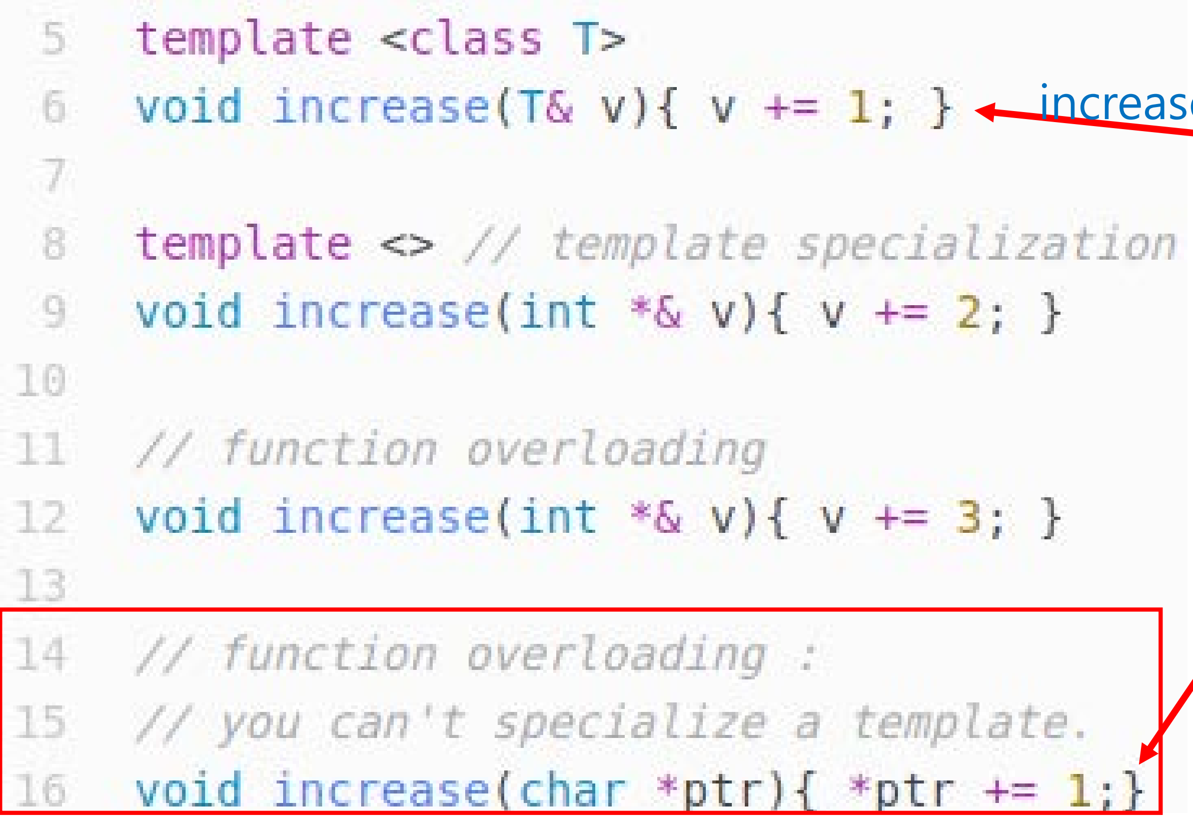
```cpp
int main(){
    int i =1;
    cout << "i= " << i << endl;
    increase(i);
    cout << "i= " << i << endl;

    int *p = &i;
    cout << "p= " << p << endl;
    increase(p);
    cout << "p= " << p << endl;
```



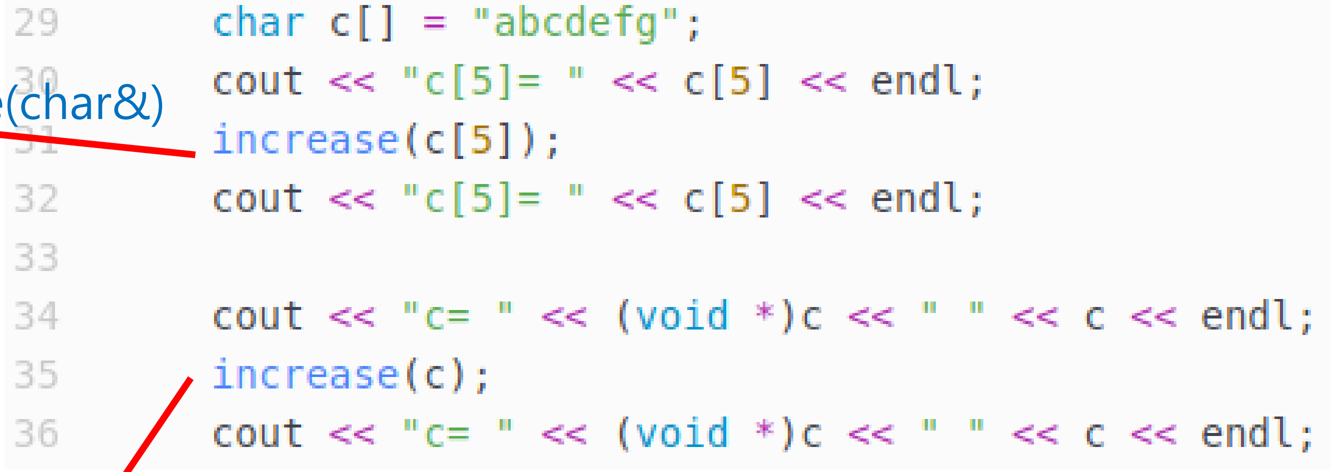
```
i= 1
i= 2
p= 0x7ffe9e84408c
p= 0x7ffe9e844098
```

+4x3

함수 템플릿과 함수 중복(2)



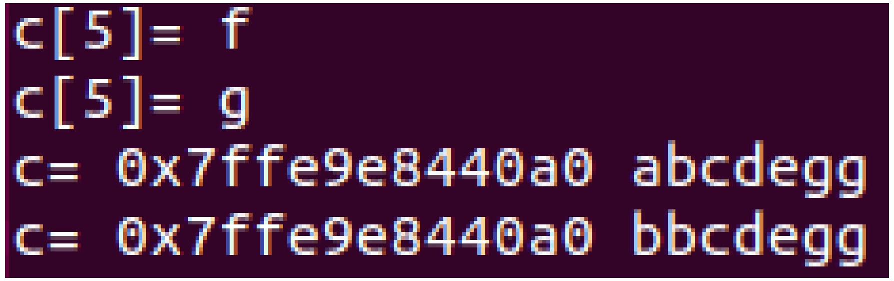
```
template <class T>
void increase(T& v){ v += 1; }

template <> // template specialization
void increase(int *& v){ v += 2; }

// function overloading
void increase(int *& v){ v += 3; }

// function overloading :
// you can't specialize a template.
void increase(char *ptr){ *ptr += 1;}
```



```
    char c[] = "abcdefg";
    cout << "c[5]= " << c[5] << endl;
    increase(c[5]);
    cout << "c[5]= " << c[5] << endl;

    cout << "c= " << (void *)c << " " << c << endl;
    increase(c);
    cout << "c= " << (void *)c << " " << c << endl;
```

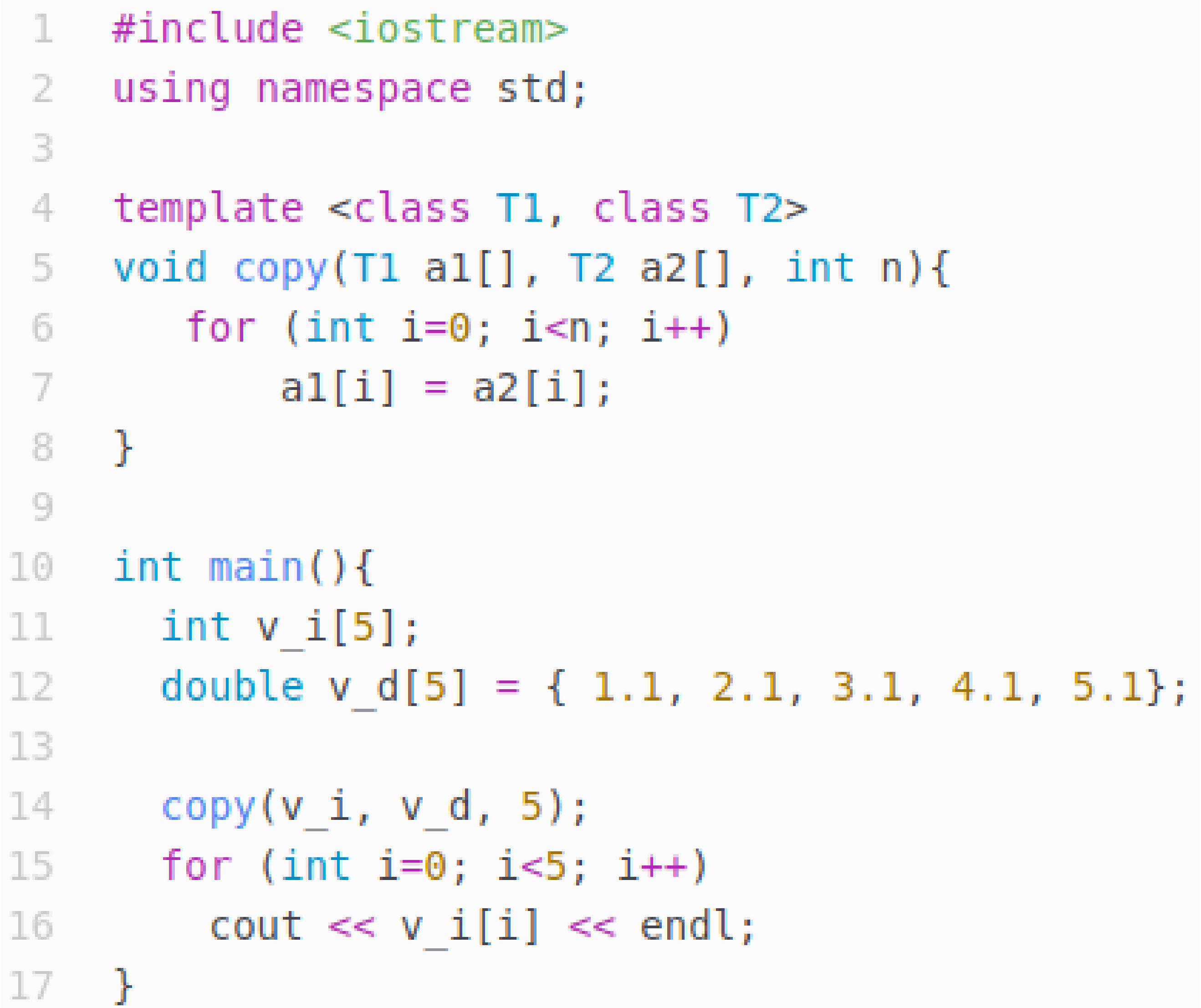
increase(char&)

배열 이름은 변수가 아니므로 reference 에 치환될 수 없음



```
c[5]= f
c[5]= g
c= 0x7ffe9e8440a0 abcdegg
c= 0x7ffe9e8440a0 bbcdegg
```

여러개의 타입 매개 변수를 가지는 템플릿 함수



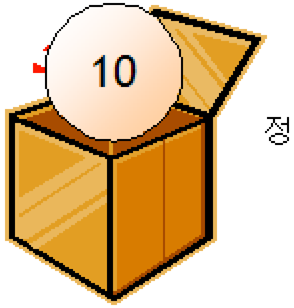
```cpp
#include <iostream>
using namespace std;

template <class T1, class T2>
void copy(T1 a1[], T2 a2[], int n){
   for (int i=0; i<n; i++)
       a1[i] = a2[i];
}

int main(){
  int v_i[5];
  double v_d[5] = { 1.1, 2.1, 3.1, 4.1, 5.1};

  copy(v_i, v_d, 5);
  for (int i=0; i<5; i++)
    cout << v_i[i] << endl;
}
```

클래스 템플릿

- 클래스 템플릿(class template): 클래스를 찍어내는 틀(template)

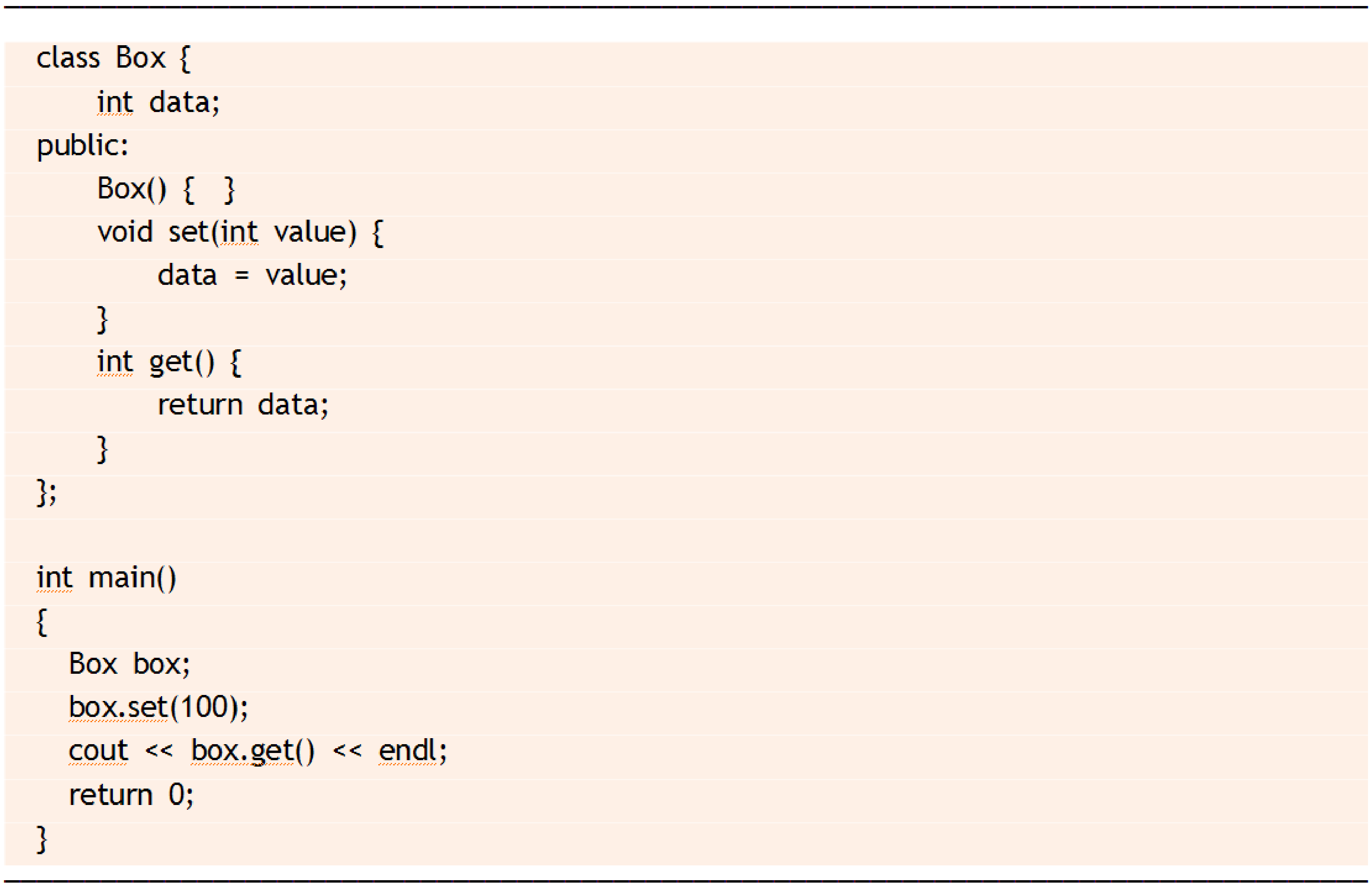
```
template <typename 타입이름, ...>
class 클래스이름
{
...

}
```

- 예제: 하나의 값을 저장하고 있는 Box class



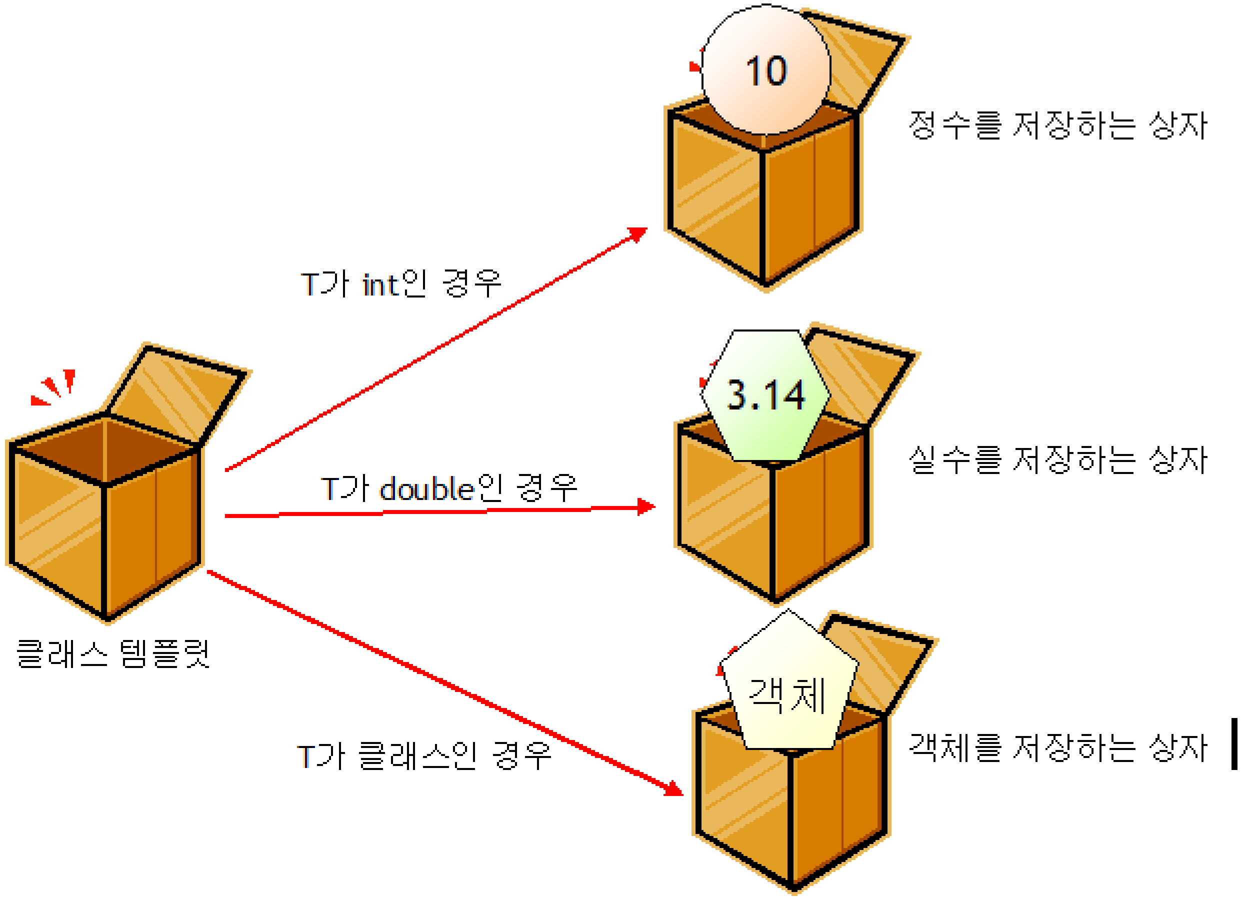
정수를 저장하는 상자

예제



```
class Box {
    int data;
public:
    Box() {  }
    void set(int value) {
        data = value;
    }
    int get() {
        return data;
    }
};

int main()
{
  Box box;
  box.set(100);
  cout << box.get() << endl;
  return 0;
}
```

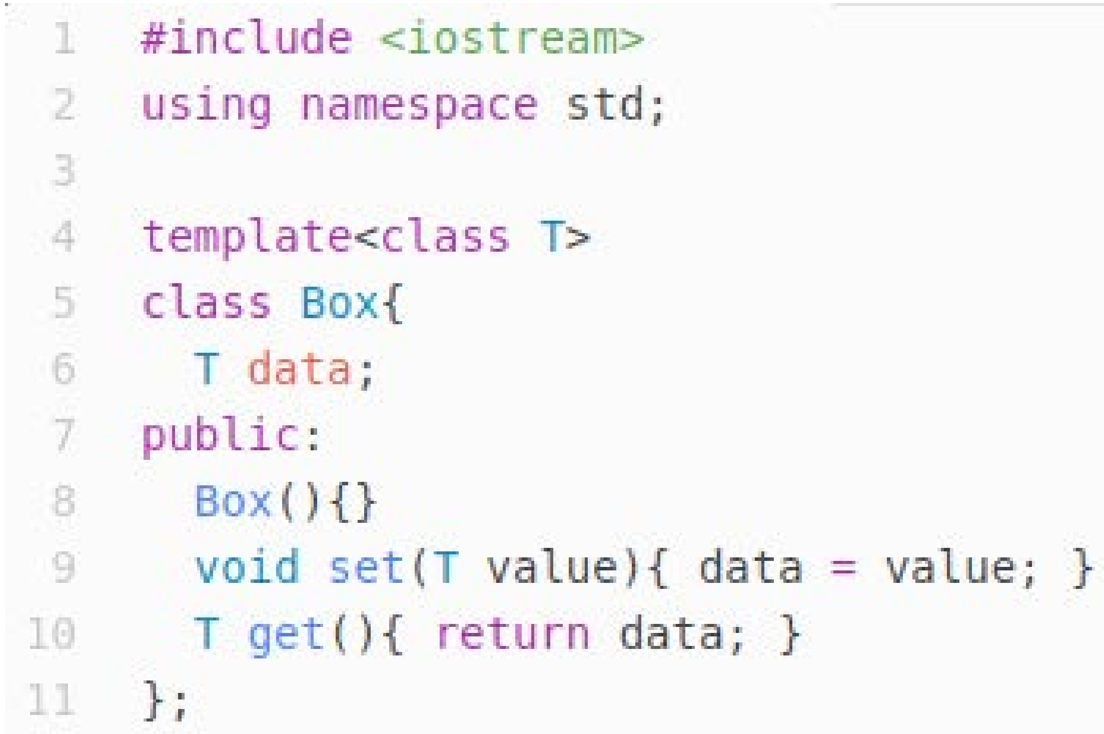
클래스 템플릿 버전


10
정수를 저장하는 상자
T가 int인 경우
3.14
실수를 저장하는 상자
T가 double인 경우
클래스 템플릿
객체
객체를 저장하는 상자
T가 클래스인 경우

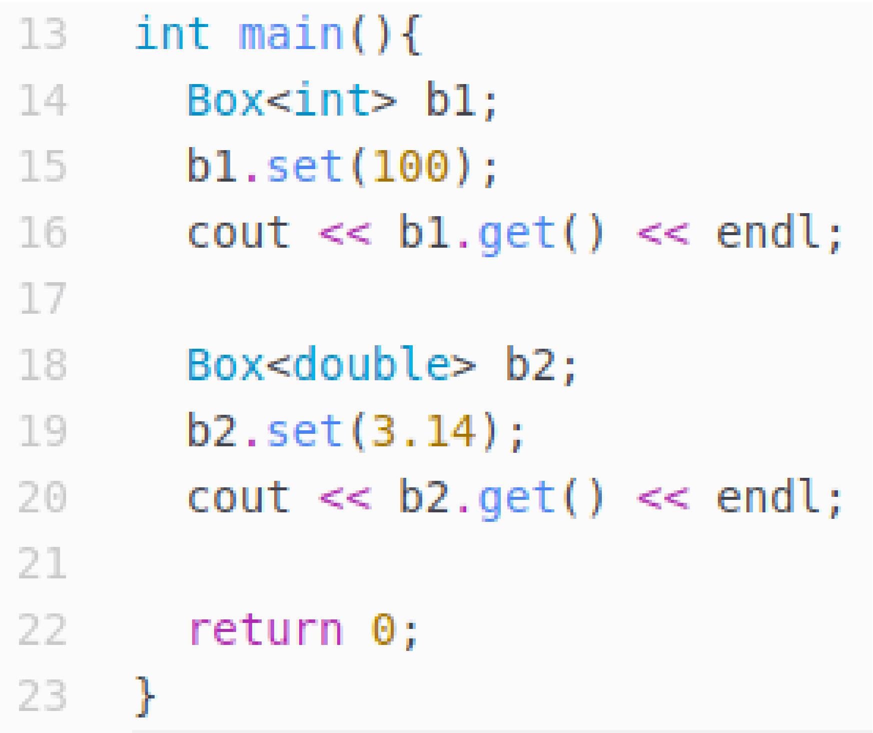
클래스 템플릿 정의

```
template <typename T>
class 클래스이름
{
...// T를 어디서든지 사용할 수 있다.
}
```

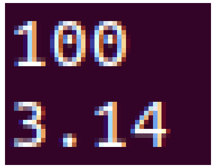
템플릿 클래스의 사용



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

template<class T>
class Box{
  T data;
public:
  Box(){}
  void set(T value){ data = value; }
  T get(){ return data; }
};
```



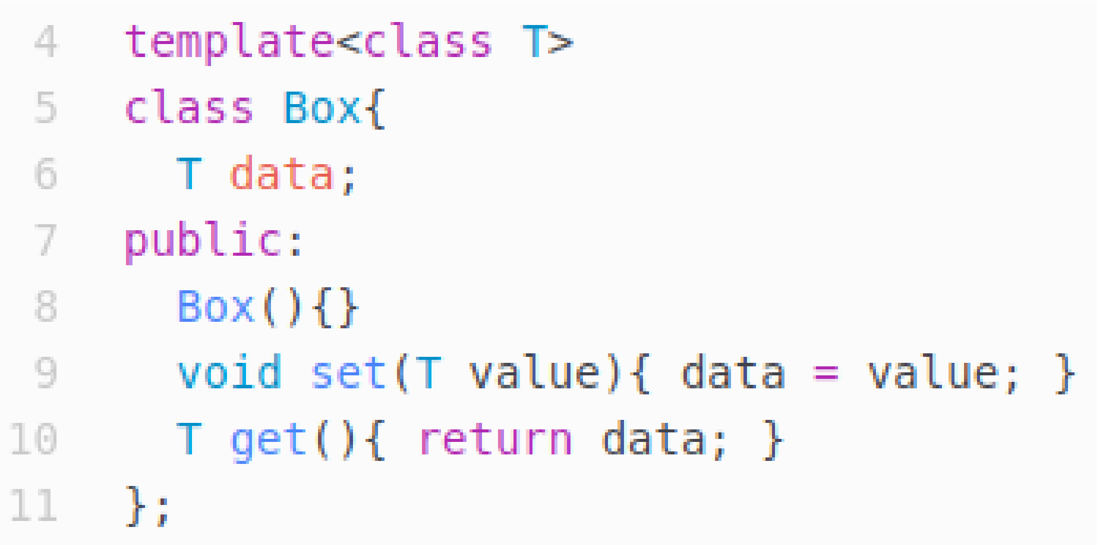
```
int main(){
  Box<int> b1;
  b1.set(100);
  cout << b1.get() << endl;

  Box<double> b2;
  b2.set(3.14);
  cout << b2.get() << endl;

  return 0;
}
```



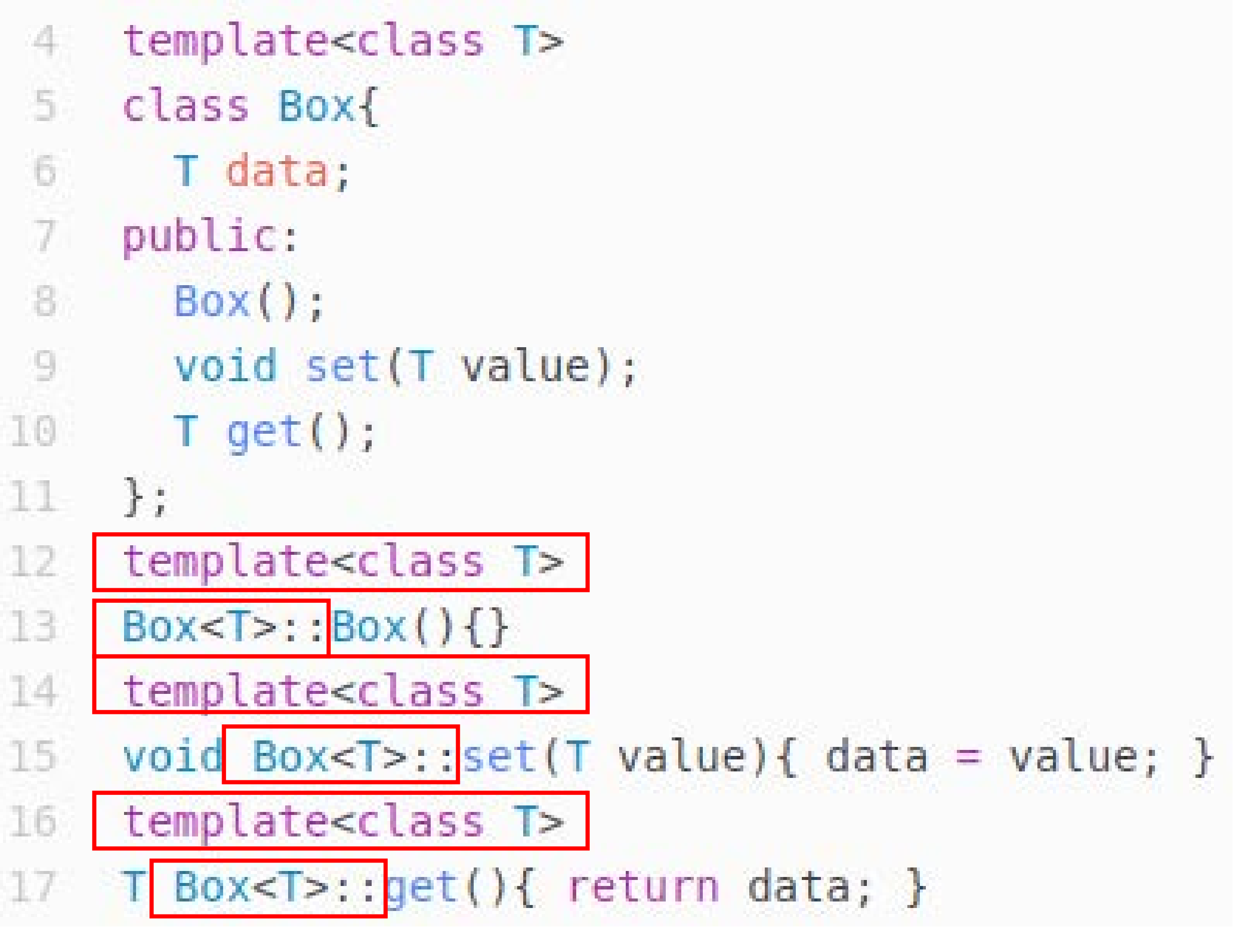
```
100
3.14
```

클래스를 사용하는 코드(main() 함수)에 따라서 실제 클래스 (Box<int> 와 Box<double>) 를 만들어야 하기 때문에 템플릿 클래스(template <class T>class Box) 의 선언과 정의는 그 클래스를 사용하는 소스 코드에 포함되어 함께 compile 되어야 한다.
(따로 컴파일한 다음 link 만 해서는 안 됨, 왜냐하면 box.o 를 compile 할 때는 Box<int> 를 만들지 않기 때문이다.)

클래스 외부에 멤버 함수를 정의할 때



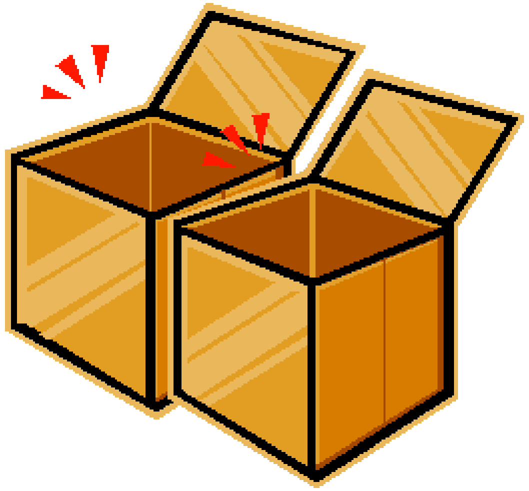
```cpp
template<class T>
class Box{
  T data;
public:
  Box(){}
  void set(T value){ data = value; }
  T get(){ return data; }
};
```



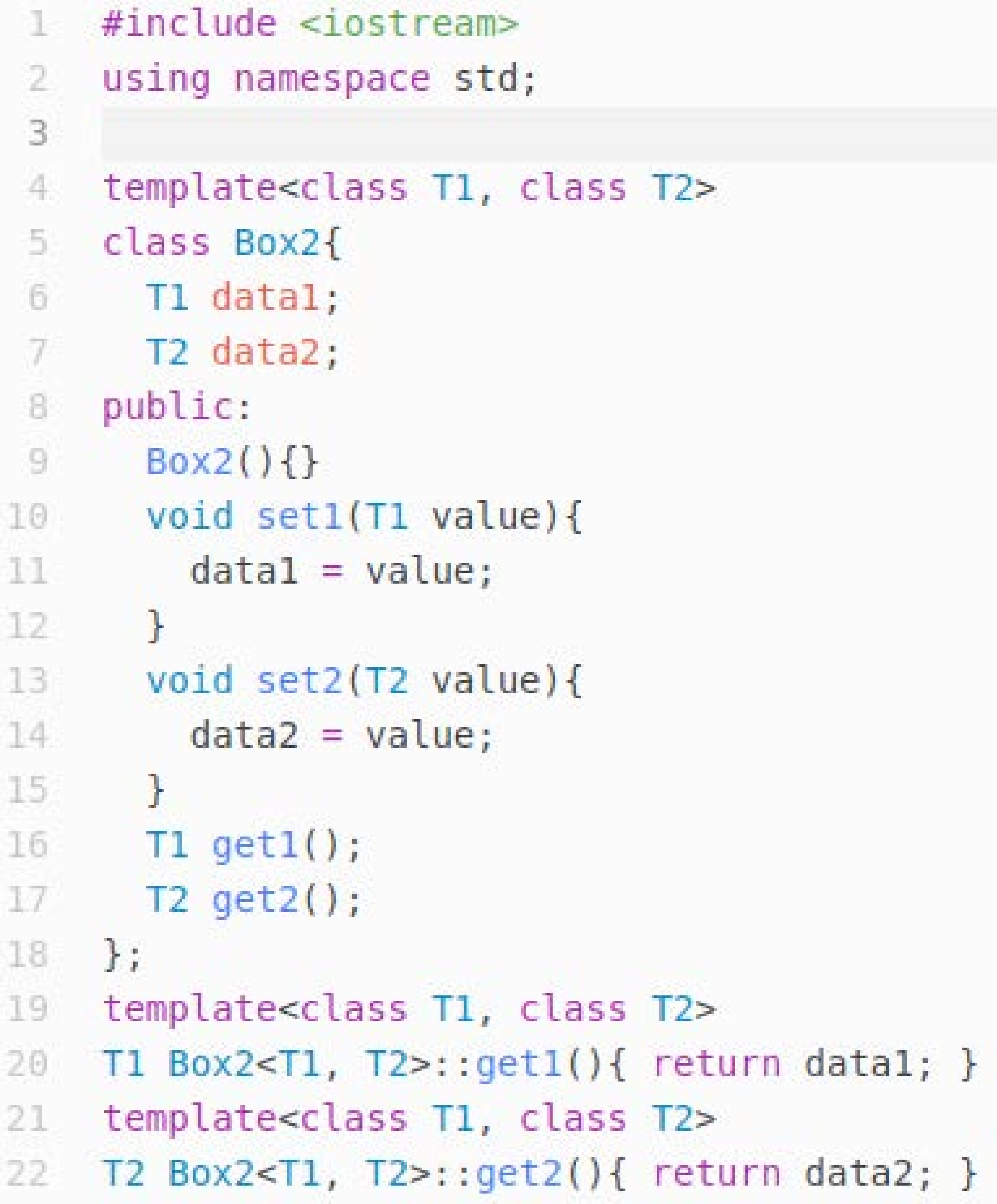
```cpp
template<class T>
class Box{
  T data;
public:
  Box();
  void set(T value);
  T get();
};
template<class T>
Box<T>::Box(){}
template<class T>
void Box<T>::set(T value){ data = value; }
template<class T>
T Box<T>::get(){ return data; }
```

두개의 타입 매개 변수

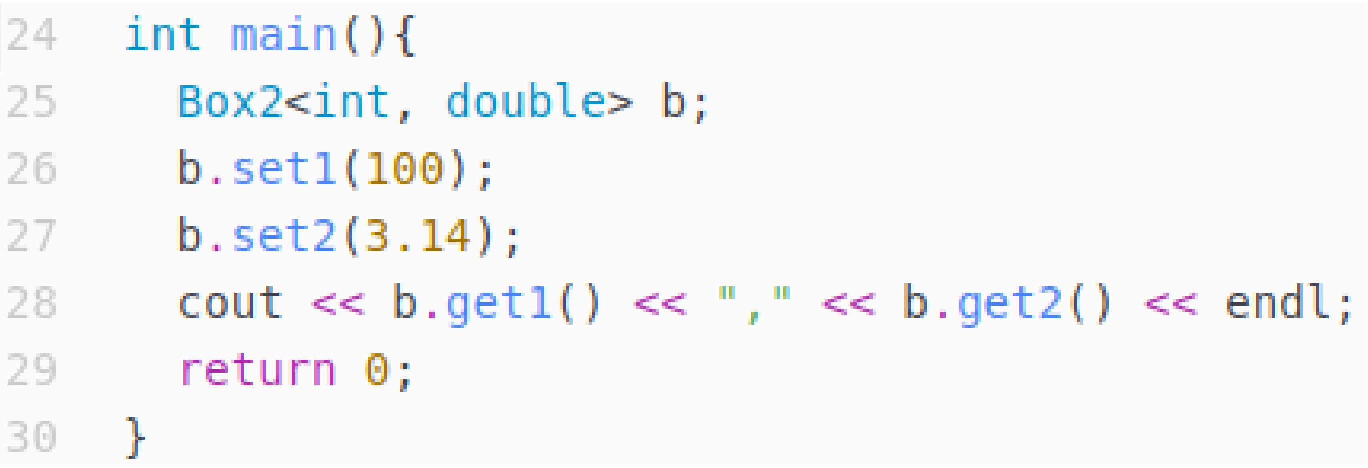
- 두 개의 데이터를 저장하는 클래스 Box2



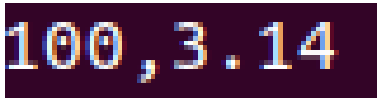
Box2 클래스 템플릿



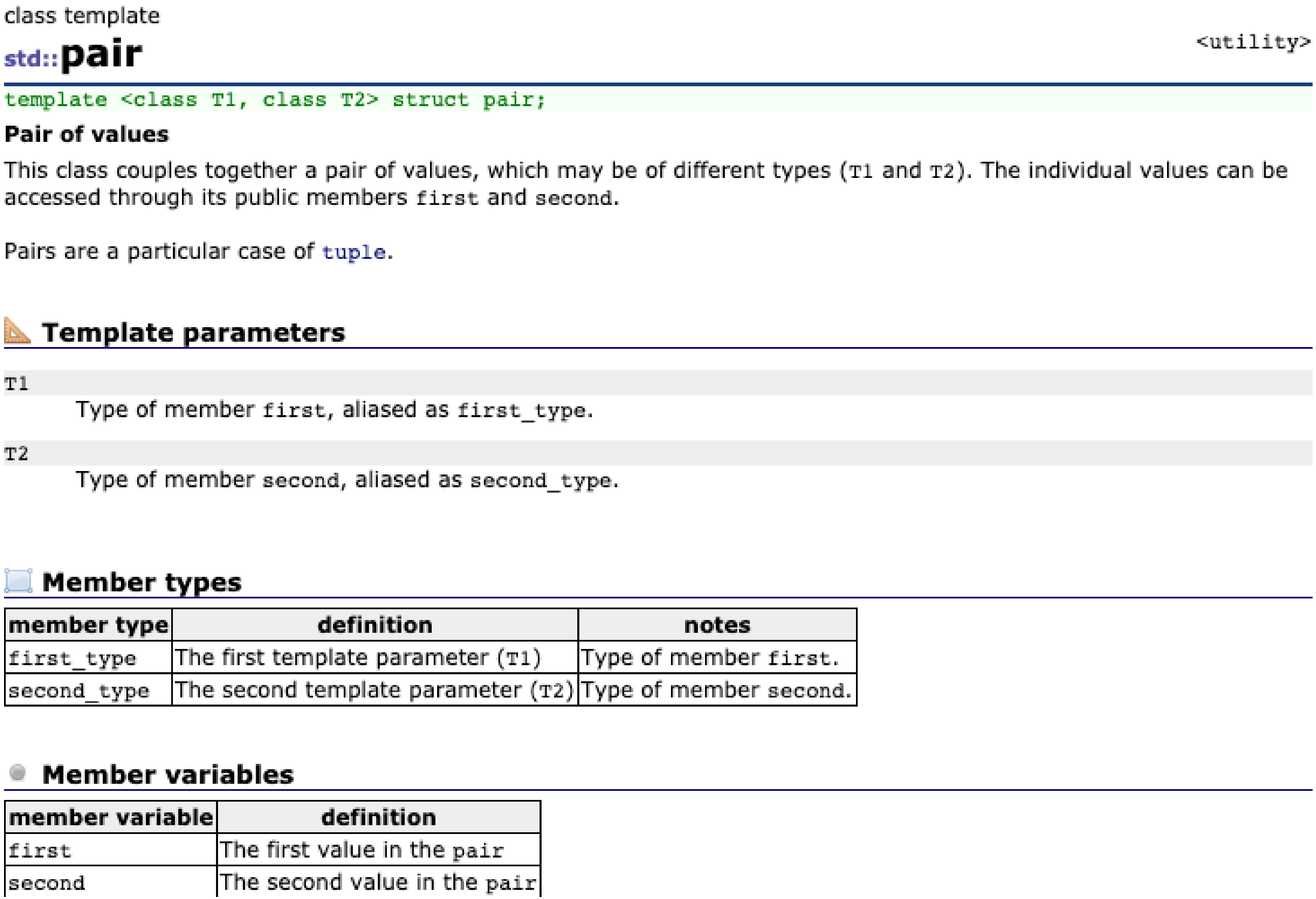
```cpp
#include <iostream>
using namespace std;

template<class T1, class T2>
class Box2{
  T1 data1;
  T2 data2;
public:
  Box2(){}
  void set1(T1 value){
    data1 = value;
  }
  void set2(T2 value){
    data2 = value;
  }
  T1 get1();
  T2 get2();
};
template<class T1, class T2>
T1 Box2<T1, T2>::get1(){ return data1; }
template<class T1, class T2>
T2 Box2<T1, T2>::get2(){ return data2; }
```



```cpp
int main(){
  Box2<int, double> b;
  b.set1(100);
  b.set2(3.14);
  cout << b.get1() << "," << b.get2() << endl;
  return 0;
}
```



```
100,3.14
```



class template

std::pair

<utility>

```
template <class T1, class T2> struct pair;
```

Pair of values

This class couples together a pair of values, which may be of different types (`T1` and `T2`). The individual values can be accessed through its public members `first` and `second`.

Pairs are a particular case of `tuple`.

Template parameters

`T1`

Type of member `first`, aliased as `first_type`.

`T2`

Type of member `second`, aliased as `second_type`.

Member types

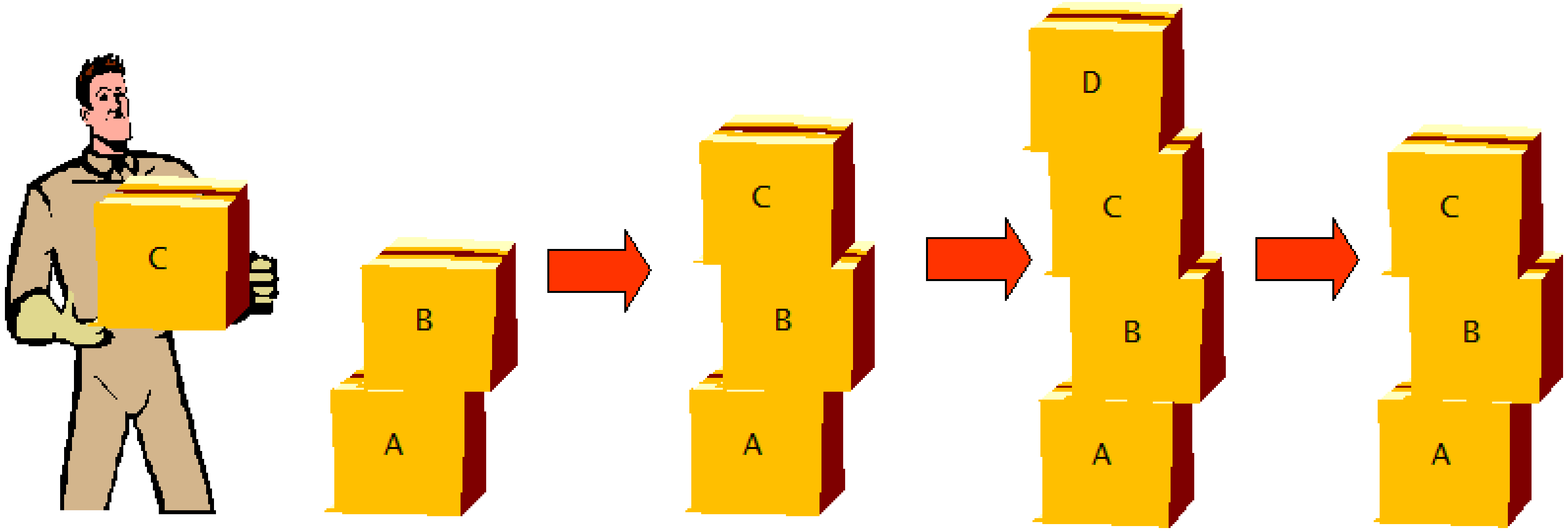
member type	definition	notes
`first_type`	The first template parameter (`T1`)	Type of member `first`.
`second_type`	The second template parameter (`T2`)	Type of member `second`.

Member variables

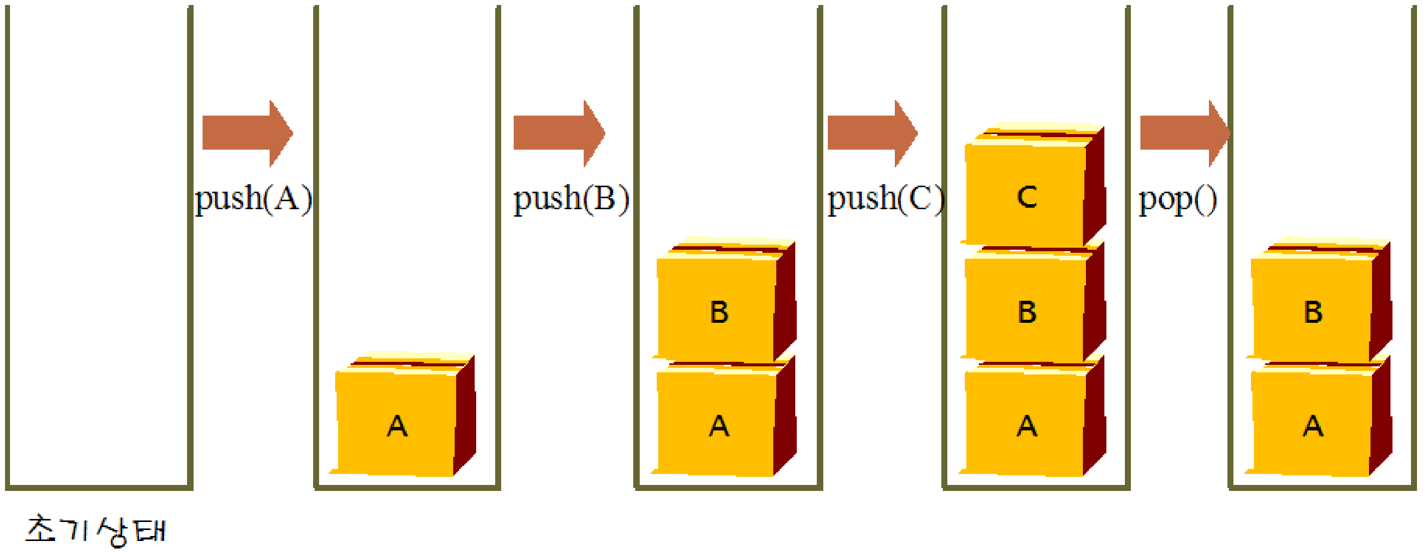
member variable	definition
`first`	The first value in the `pair`
`second`	The second value in the `pair`

예제: 스택

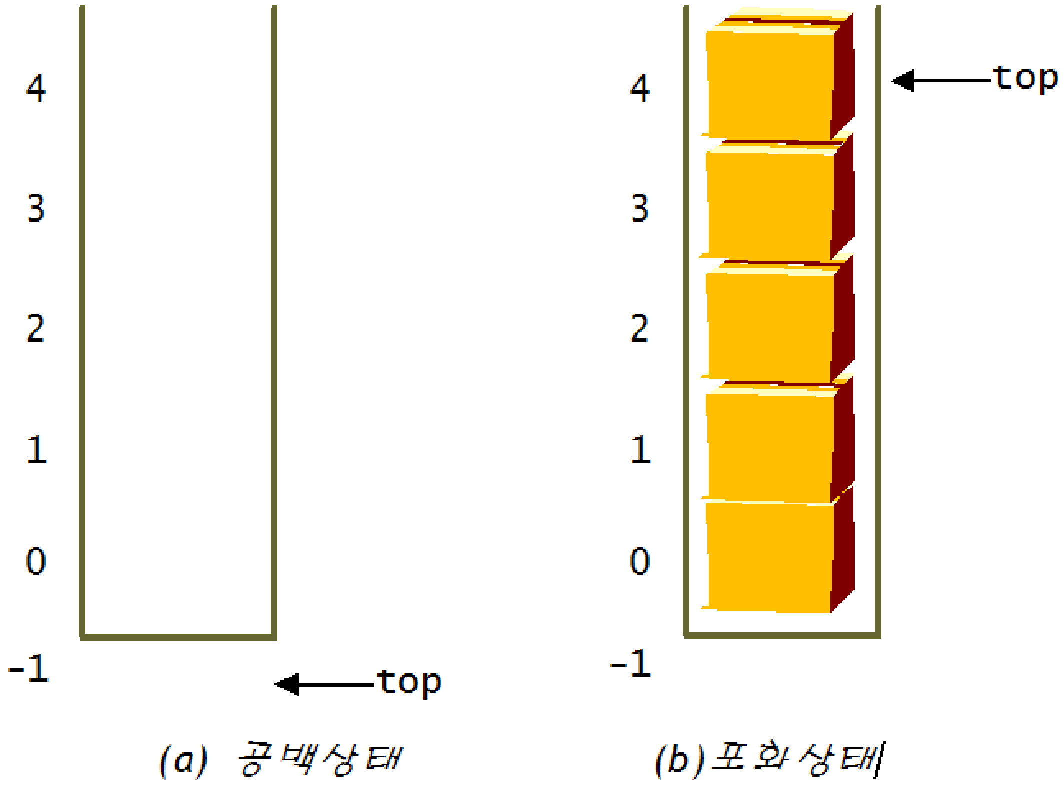
- 스택(stack): 후입 선출(LIFO:Last-In First-Out) 자료 구조

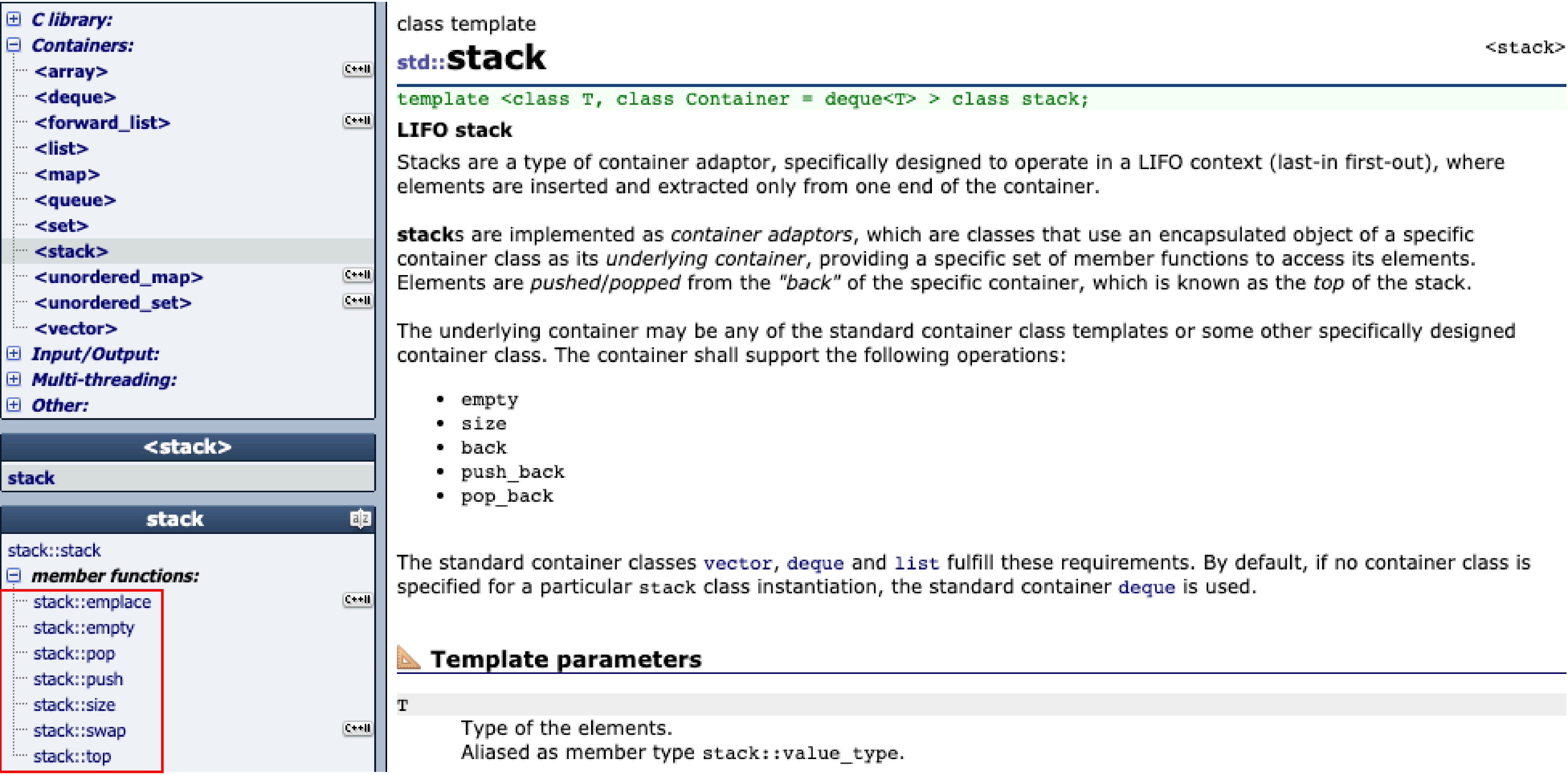


스택의 연산들



스택의 공백 상태와 포화 상태





class template

<stack>

std::stack

```
template <class T, class Container = deque<T> > class stack;
```

LIFO stack

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.

stacks are implemented as *container adaptors*, which are classes that use an encapsulated object of a specific container class as its *underlying container*, providing a specific set of member functions to access its elements. Elements are *pushed/popped* from the *"back"* of the specific container, which is known as the *top* of the stack.

The underlying container may be any of the standard container class templates or some other specifically designed container class. The container shall support the following operations:

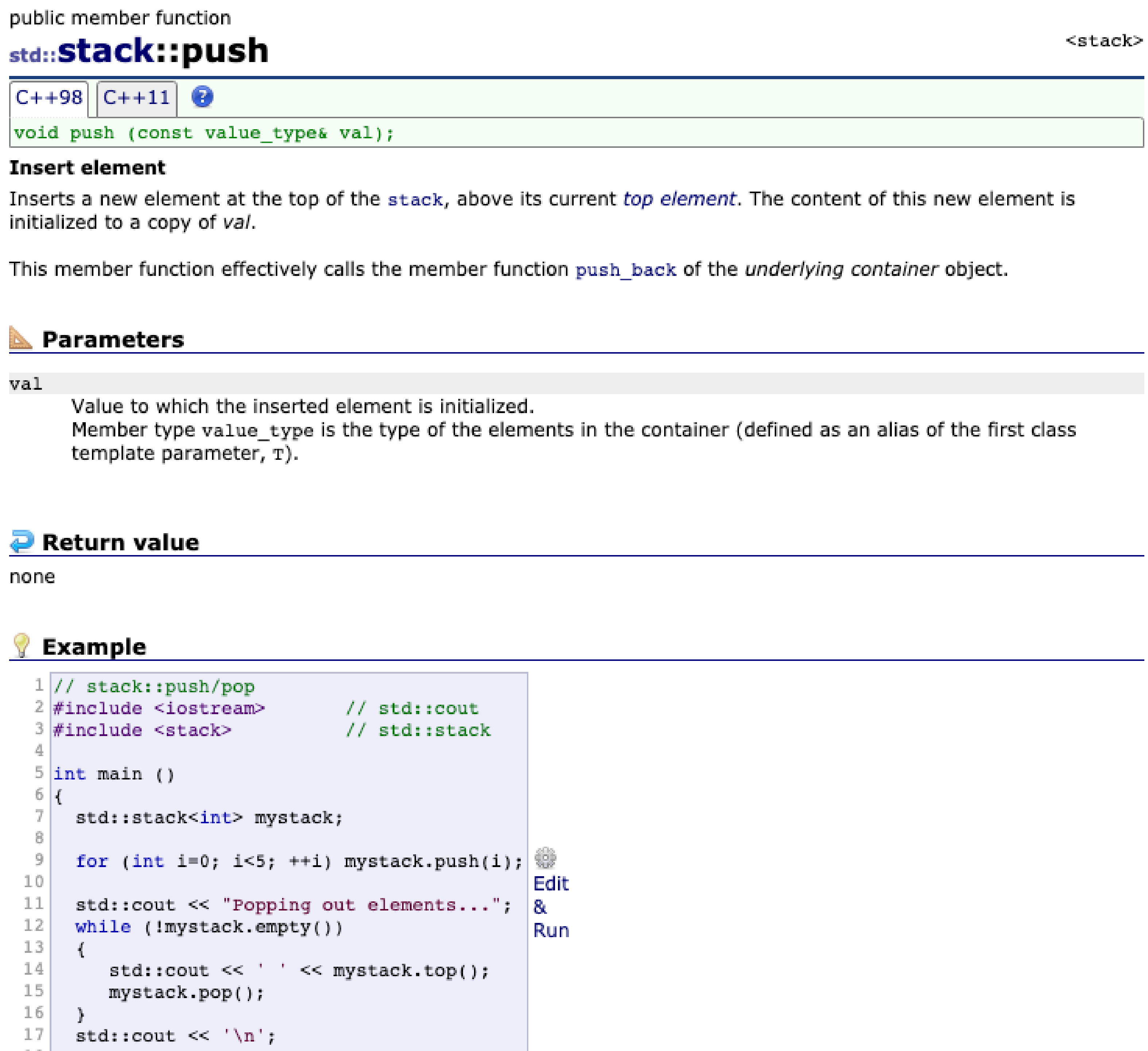
- `empty`
- `size`
- `back`
- `push_back`
- `pop_back`

The standard container classes `vector`, `deque` and `list` fulfill these requirements. By default, if no container class is specified for a particular `stack` class instantiation, the standard container `deque` is used.

Template parameters

`T`

Type of the elements.
Aliased as member type `stack::value_type`.



public member function

std::stack::push

<stack>

C++98 C++11 ?

```
void push (const value_type& val);
```

Insert element

Inserts a new element at the top of the `stack`, above its current *top element*. The content of this new element is initialized to a copy of *val*.

This member function effectively calls the member function `push_back` of the *underlying container* object.

Parameters

`val`

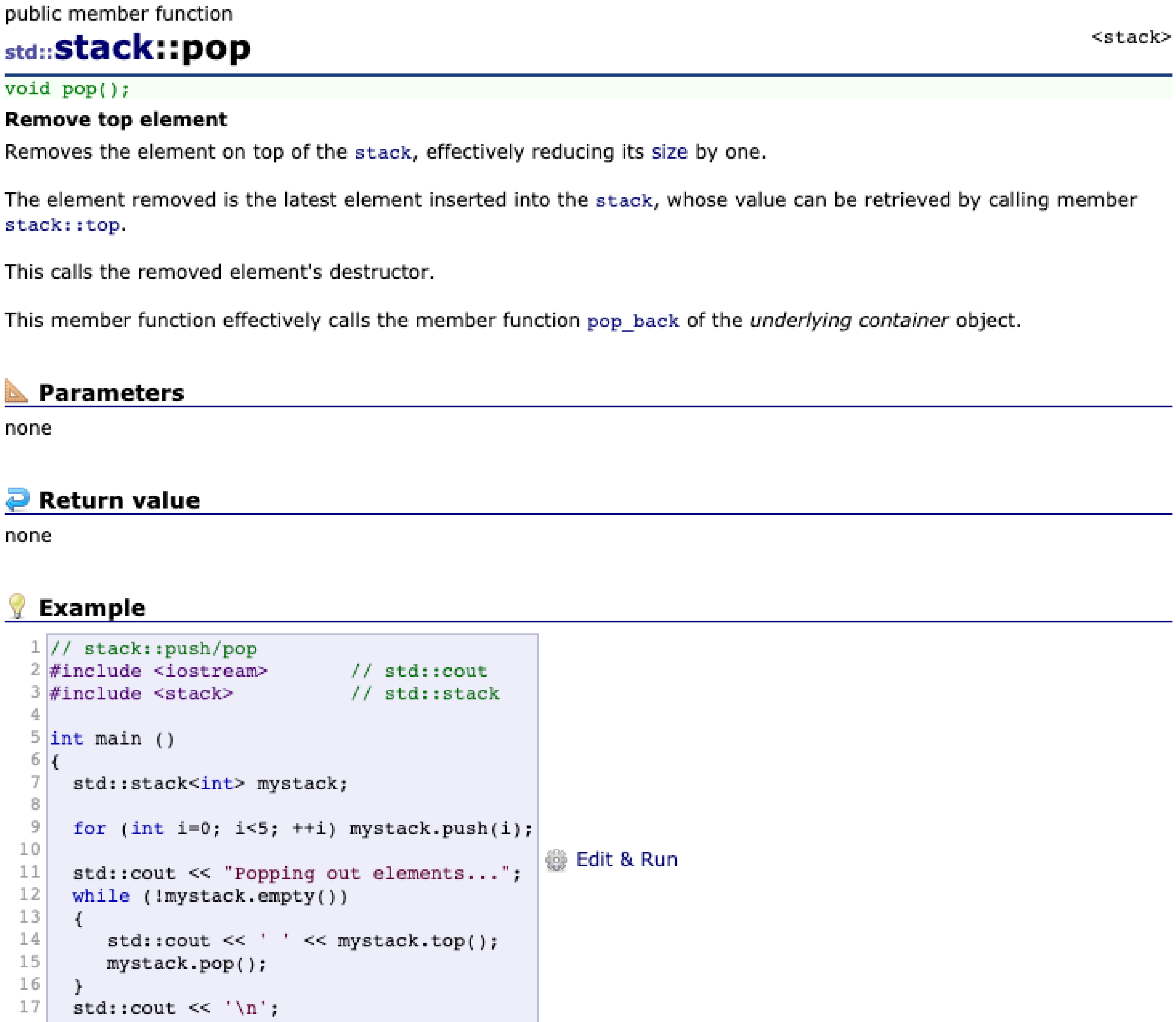
Value to which the inserted element is initialized.
Member type `value_type` is the type of the elements in the container (defined as an alias of the first class template parameter, `T`).

Return value

none

Example

```
// stack::push/pop
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main ()
{
  std::stack<int> mystack;

  for (int i=0; i<5; ++i) mystack.push(i);

  std::cout << "Popping out elements...";
  while (!mystack.empty())
  {
     std::cout << ' ' << mystack.top();
     mystack.pop();
  }
  std::cout << '\n';
```



public member function

<stack>

std::stack::pop

```
void pop();
```

Remove top element

Removes the element on top of the `stack`, effectively reducing its size by one.

The element removed is the latest element inserted into the `stack`, whose value can be retrieved by calling member `stack::top`.

This calls the removed element's destructor.

This member function effectively calls the member function `pop_back` of the *underlying container* object.

Parameters

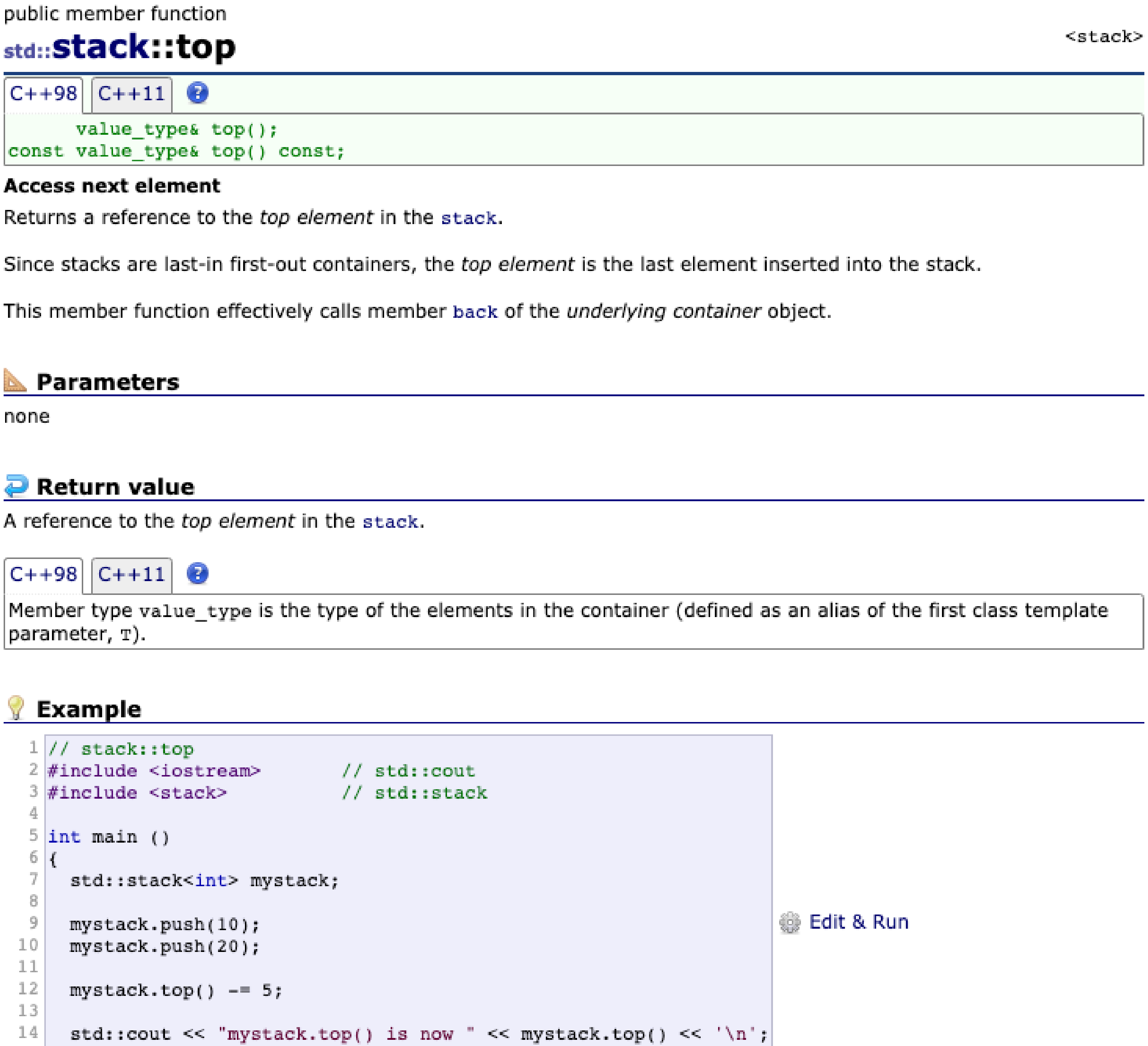
none

Return value

none

Example

```
// stack::push/pop
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main ()
{
  std::stack<int> mystack;

  for (int i=0; i<5; ++i) mystack.push(i);

  std::cout << "Popping out elements...";
  while (!mystack.empty())
  {
     std::cout << ' ' << mystack.top();
     mystack.pop();
  }
  std::cout << '\n';
```



public member function

std::stack::top

<stack>

C++98 C++11 ?

```
      value_type& top();
const value_type& top() const;
```

Access next element

Returns a reference to the *top element* in the `stack`.

Since stacks are last-in first-out containers, the *top element* is the last element inserted into the stack.

This member function effectively calls member `back` of the *underlying container* object.

Parameters

none

Return value

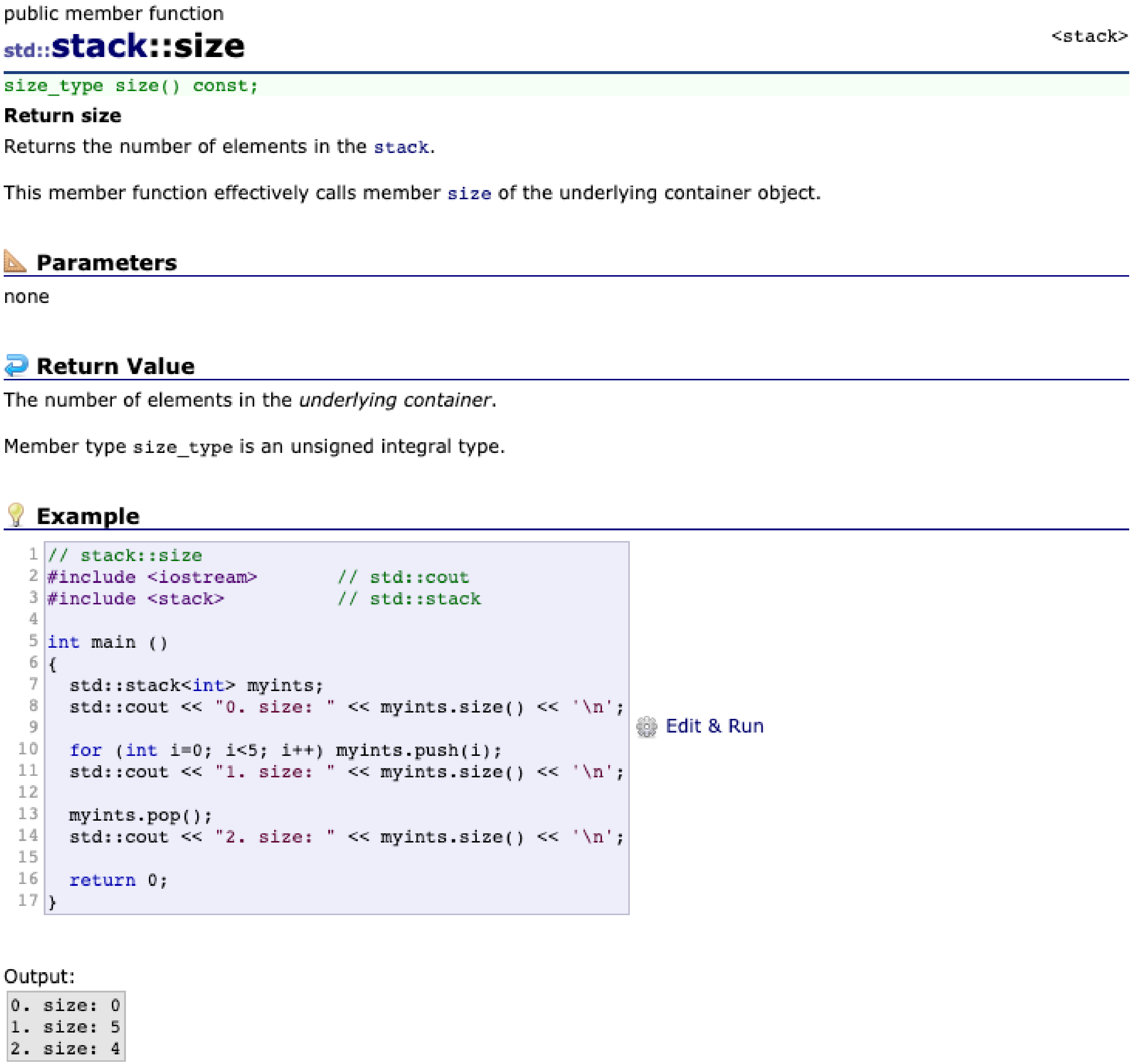
A reference to the *top element* in the `stack`.

C++98 C++11 ?

Member type `value_type` is the type of the elements in the container (defined as an alias of the first class template parameter, `T`).

Example

```
// stack::top
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main ()
{
  std::stack<int> mystack;

  mystack.push(10);
  mystack.push(20);

  mystack.top() -= 5;

  std::cout << "mystack.top() is now " << mystack.top() << '\n';
```



public member function

std::stack::size

<stack>

```
size_type size() const;
```

Return size

Returns the number of elements in the stack.

This member function effectively calls member size of the underlying container object.

Parameters

none

Return Value

The number of elements in the *underlying container*.

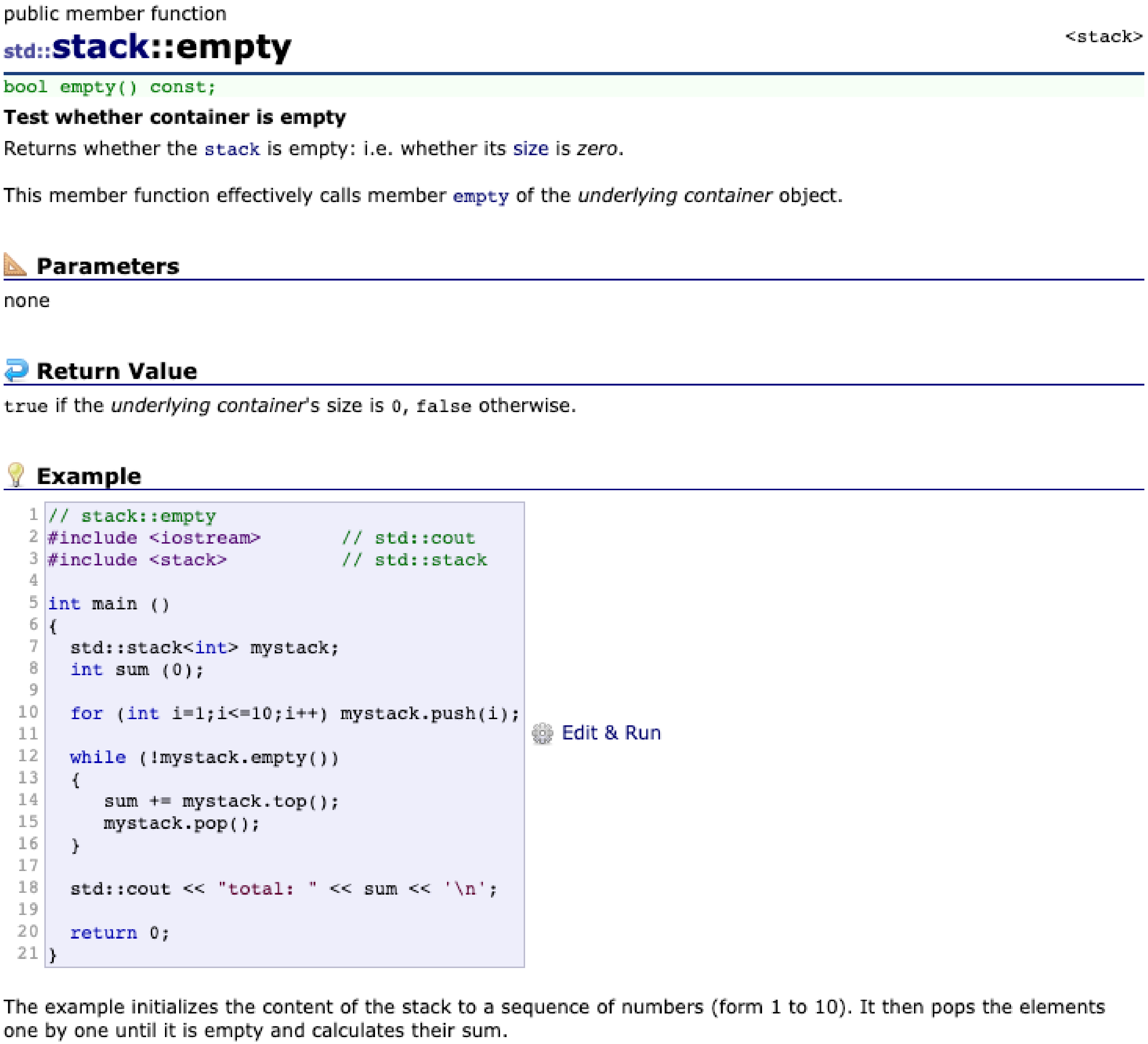
Member type size_type is an unsigned integral type.

Example

```
// stack::size
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main ()
{
  std::stack<int> myints;
  std::cout << "0. size: " << myints.size() << '\n';

  for (int i=0; i<5; i++) myints.push(i);
  std::cout << "1. size: " << myints.size() << '\n';

  myints.pop();
  std::cout << "2. size: " << myints.size() << '\n';

  return 0;
}
```

Output:

```
0. size: 0
1. size: 5
2. size: 4
```



public member function

std::stack::empty

<stack>

`bool empty() const;`

Test whether container is empty

Returns whether the `stack` is empty: i.e. whether its size is *zero*.

This member function effectively calls member `empty` of the *underlying container* object.

Parameters

none

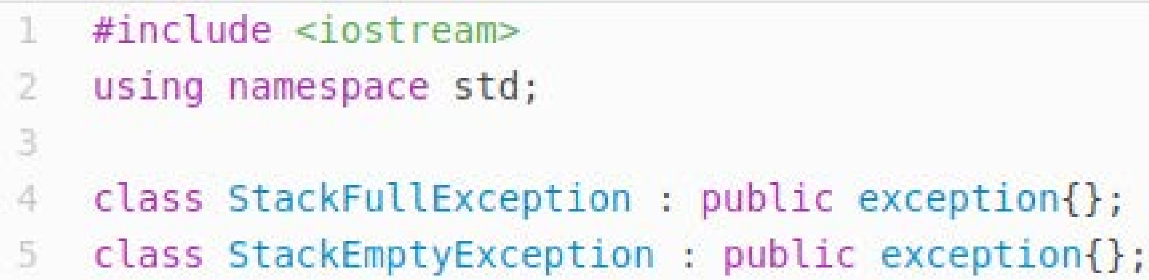
Return Value

`true` if the *underlying container*'s size is `0`, `false` otherwise.

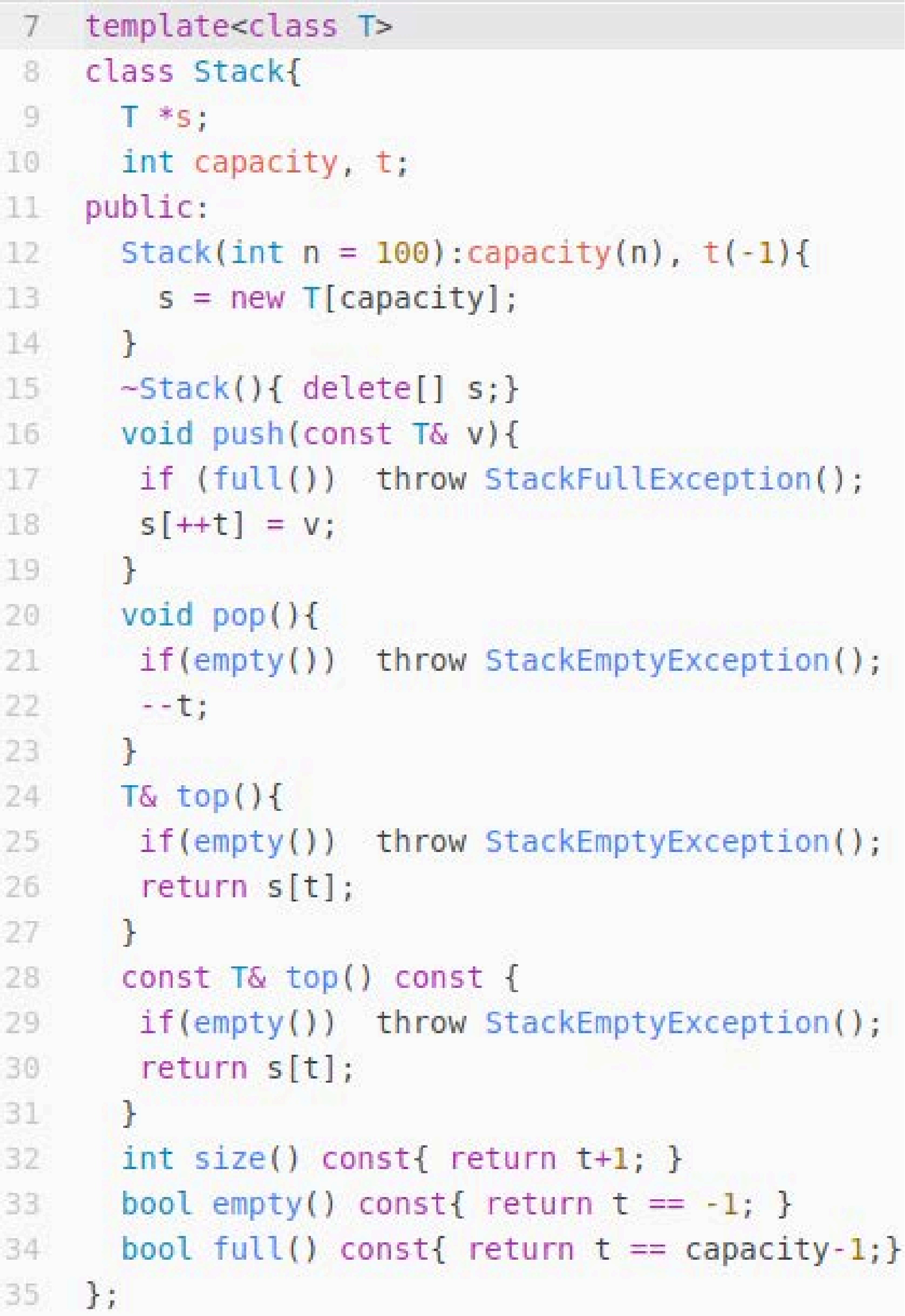
Example

```cpp
// stack::empty
#include <iostream>       // std::cout
#include <stack>          // std::stack

int main ()
{
  std::stack<int> mystack;
  int sum (0);

  for (int i=1;i<=10;i++) mystack.push(i);

  while (!mystack.empty())
  {
     sum += mystack.top();
     mystack.pop();
  }

  std::cout << "total: " << sum << '\n';

  return 0;
}
```

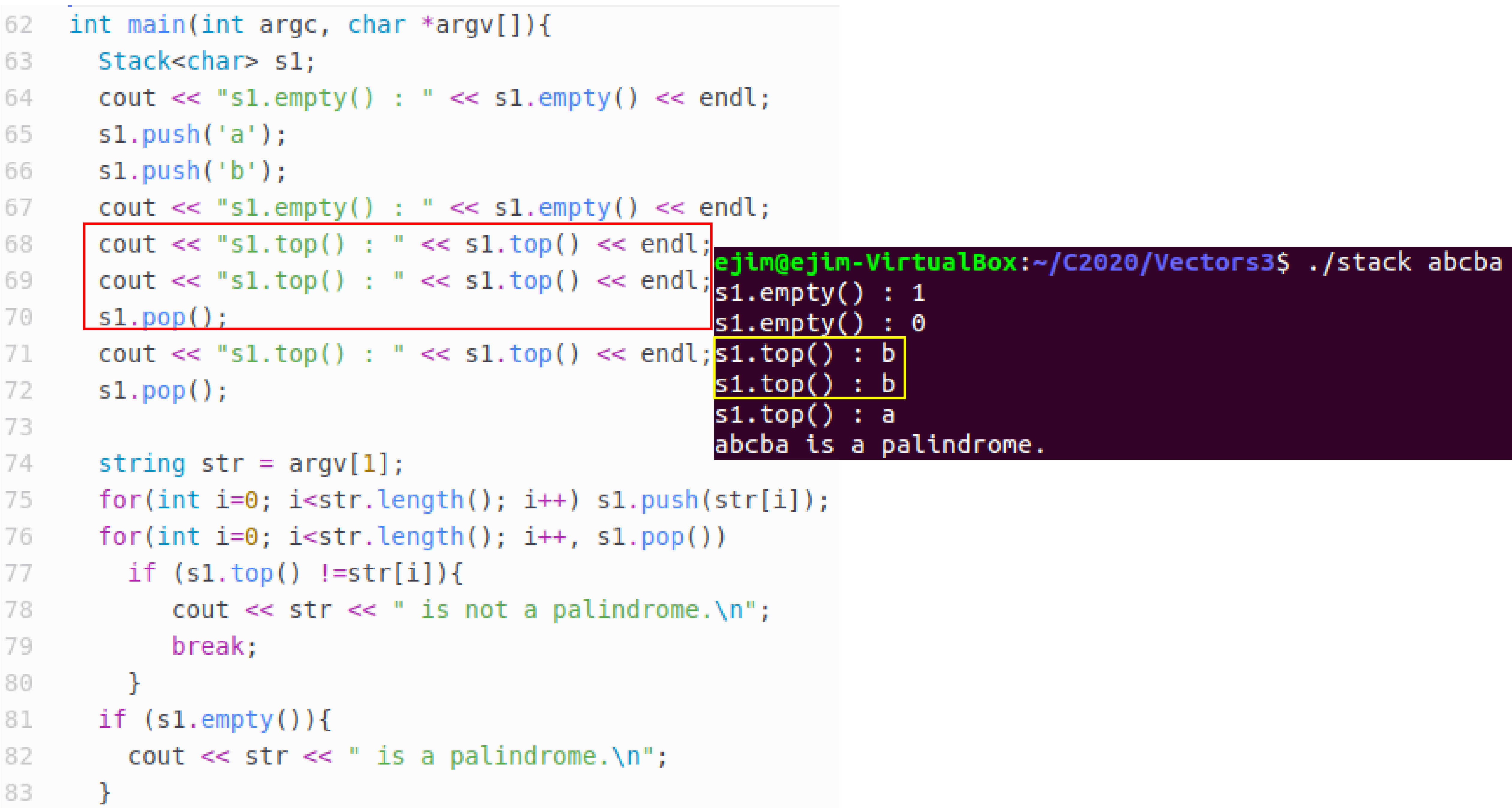
The example initializes the content of the stack to a sequence of numbers (form 1 to 10). It then pops the elements one by one until it is empty and calculates their sum.



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

class StackFullException : public exception{};
class StackEmptyException : public exception{};
```

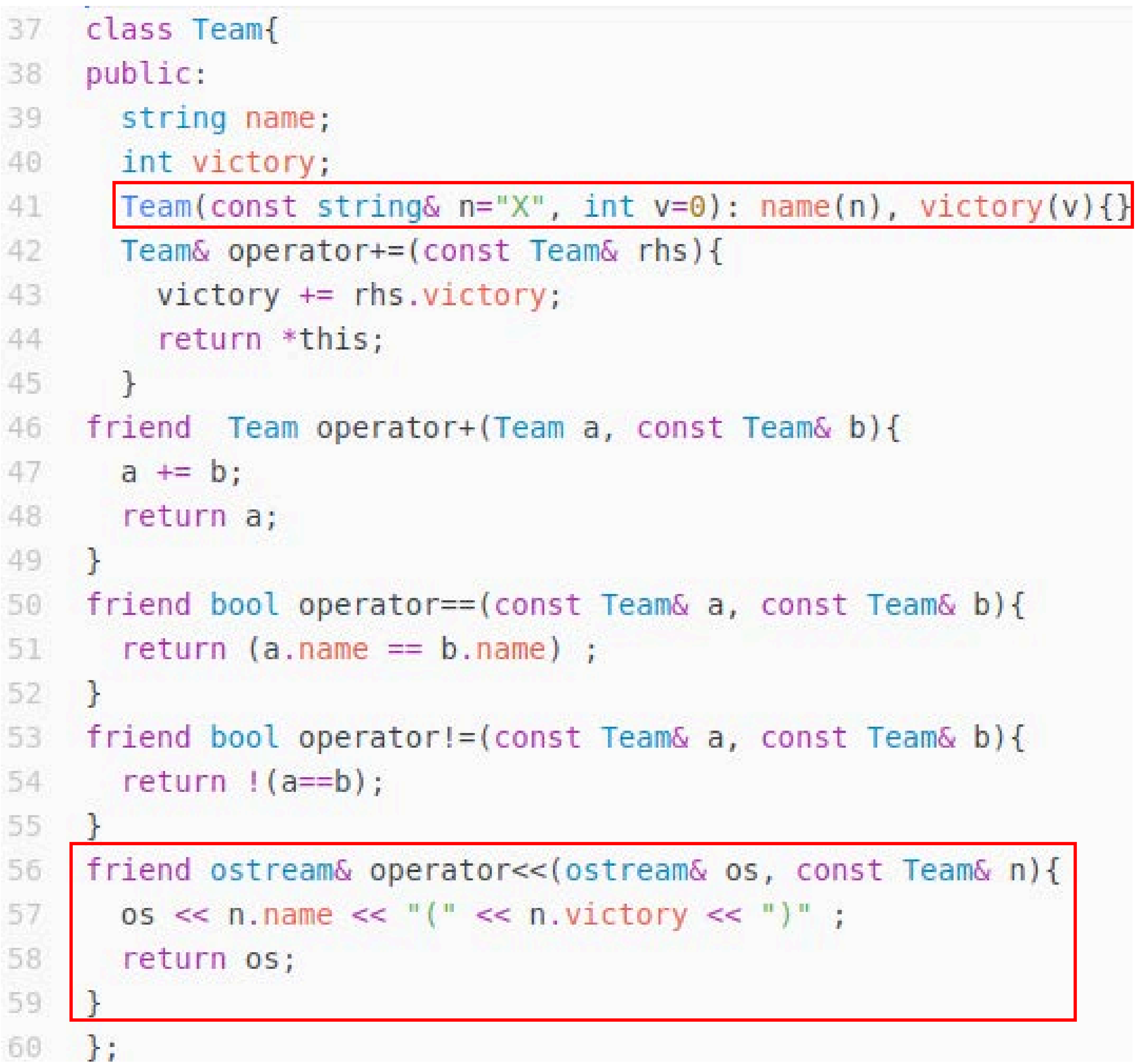


```cpp
template<class T>
class Stack{
  T *s;
  int capacity, t;
public:
  Stack(int n = 100):capacity(n), t(-1){
    s = new T[capacity];
  }
  ~Stack(){ delete[] s;}
  void push(const T& v){
   if (full())  throw StackFullException();
   s[++t] = v;
  }
  void pop(){
   if(empty())  throw StackEmptyException();
   --t;
  }
  T& top(){
   if(empty())  throw StackEmptyException();
   return s[t];
  }
  const T& top() const {
   if(empty())  throw StackEmptyException();
   return s[t];
  }
  int size() const{ return t+1; }
  bool empty() const{ return t == -1; }
  bool full() const{ return t == capacity-1;}
};
```

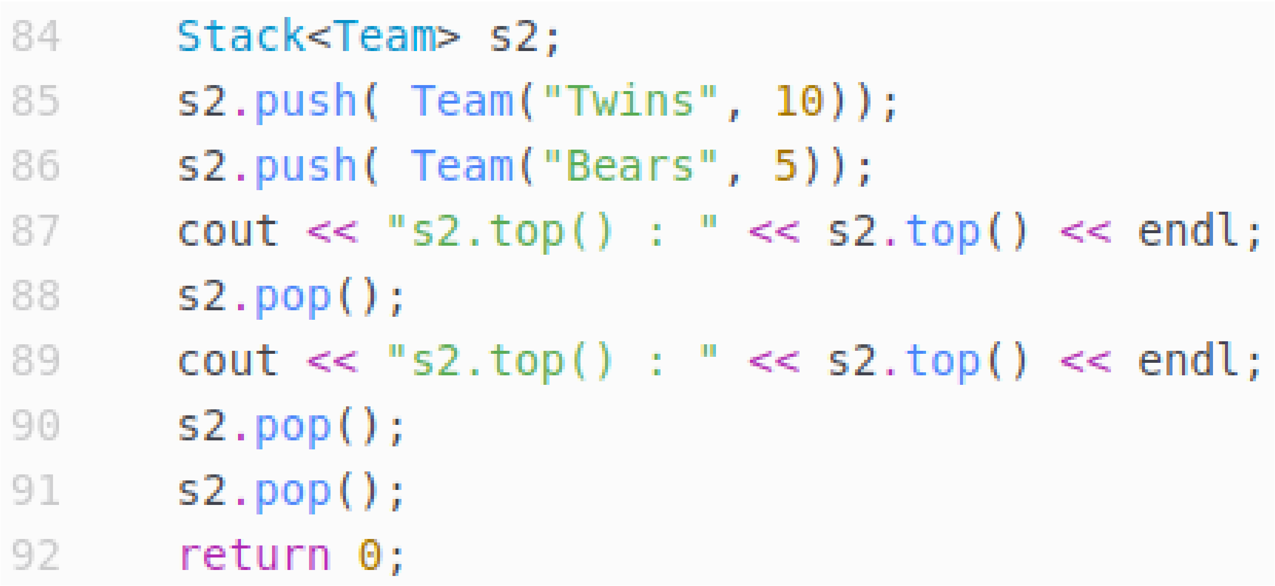


```cpp
int main(int argc, char *argv[]){
  Stack<char> s1;
  cout << "s1.empty() : " << s1.empty() << endl;
  s1.push('a');
  s1.push('b');
  cout << "s1.empty() : " << s1.empty() << endl;
  cout << "s1.top() : " << s1.top() << endl;
  cout << "s1.top() : " << s1.top() << endl;
  s1.pop();
  cout << "s1.top() : " << s1.top() << endl;
  s1.pop();

  string str = argv[1];
  for(int i=0; i<str.length(); i++) s1.push(str[i]);
  for(int i=0; i<str.length(); i++, s1.pop())
    if (s1.top() !=str[i]){
       cout << str << " is not a palindrome.\n";
       break;
    }
  if (s1.empty()){
    cout << str << " is a palindrome.\n";
  }
```

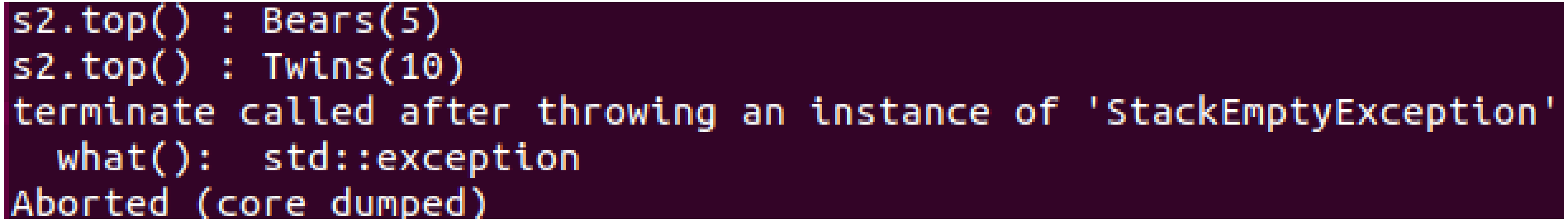
```
ejim@ejim-VirtualBox:~/C2020/Vectors3$ ./stack abcba
s1.empty() : 1
s1.empty() : 0
s1.top() : b
s1.top() : b
s1.top() : a
abcba is a palindrome.
```



```
class Team{
public:
  string name;
  int victory;
  Team(const string& n="X", int v=0): name(n), victory(v){}
  Team& operator+=(const Team& rhs){
    victory += rhs.victory;
    return *this;
  }
friend  Team operator+(Team a, const Team& b){
  a += b;
  return a;
}
friend bool operator==(const Team& a, const Team& b){
  return (a.name == b.name) ;
}
friend bool operator!=(const Team& a, const Team& b){
  return !(a==b);
}
friend ostream& operator<<(ostream& os, const Team& n){
  os << n.name << "(" << n.victory << ")" ;
  return os;
}
};
```



```
    Stack<Team> s2;
    s2.push( Team("Twins", 10));
    s2.push( Team("Bears", 5));
    cout << "s2.top() : " << s2.top() << endl;
    s2.pop();
    cout << "s2.top() : " << s2.top() << endl;
    s2.pop();
    s2.pop();
    return 0;
```

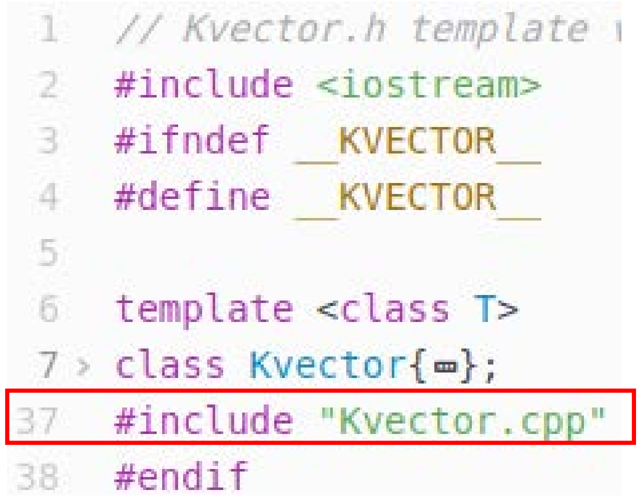


```
s2.top() : Bears(5)
s2.top() : Twins(10)
terminate called after throwing an instance of 'StackEmptyException'
  what():  std::exception
Aborted (core dumped)
```

실습

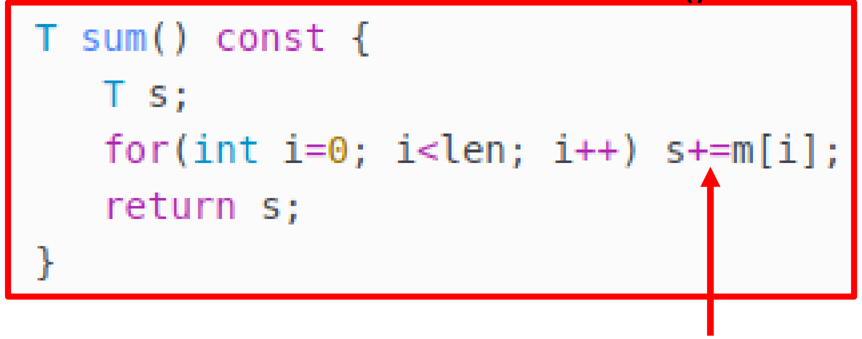
Kvector 를 template class 로 만들어 m 이 정수 배열이 아닌 임의의 타입의 배열이 되도록 수정하고 멤버 함수 sum() 을 추가하여 주어진 main() 함수에 대하여 다음과 같은 출력이 되도록 하여라.

Kvector class 가 template 으로 사용되려면 Kvector.cpp 가 main() 함수와 같은 file 에 포함되어야 한다.



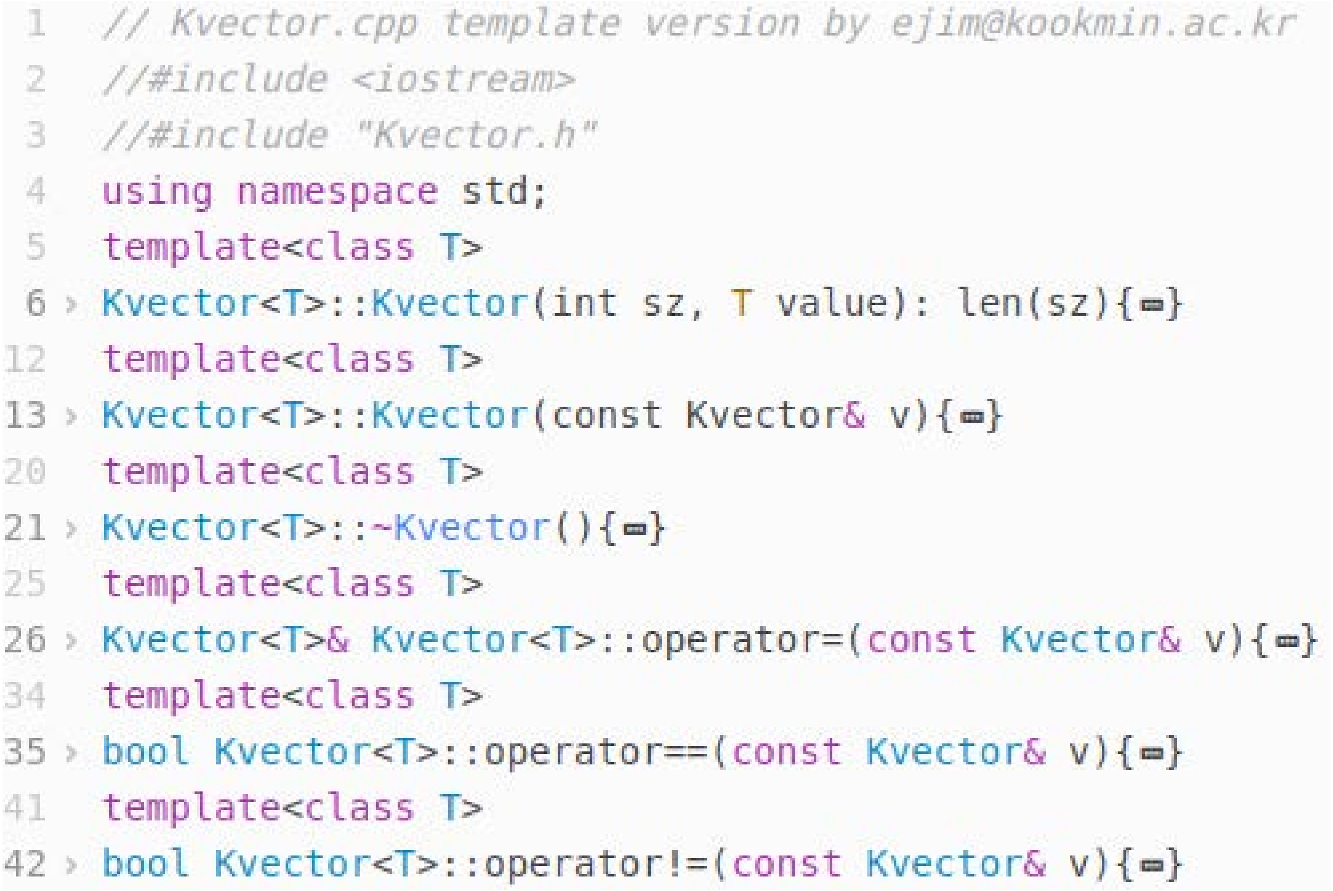
```
// Kvector.h template
#include <iostream>
#ifndef __KVECTOR__
#define __KVECTOR__

template <class T>
class Kvector{⋯};
#include "Kvector.cpp"
#endif
```

Kvector 의 member function sum() 을 추가하라.

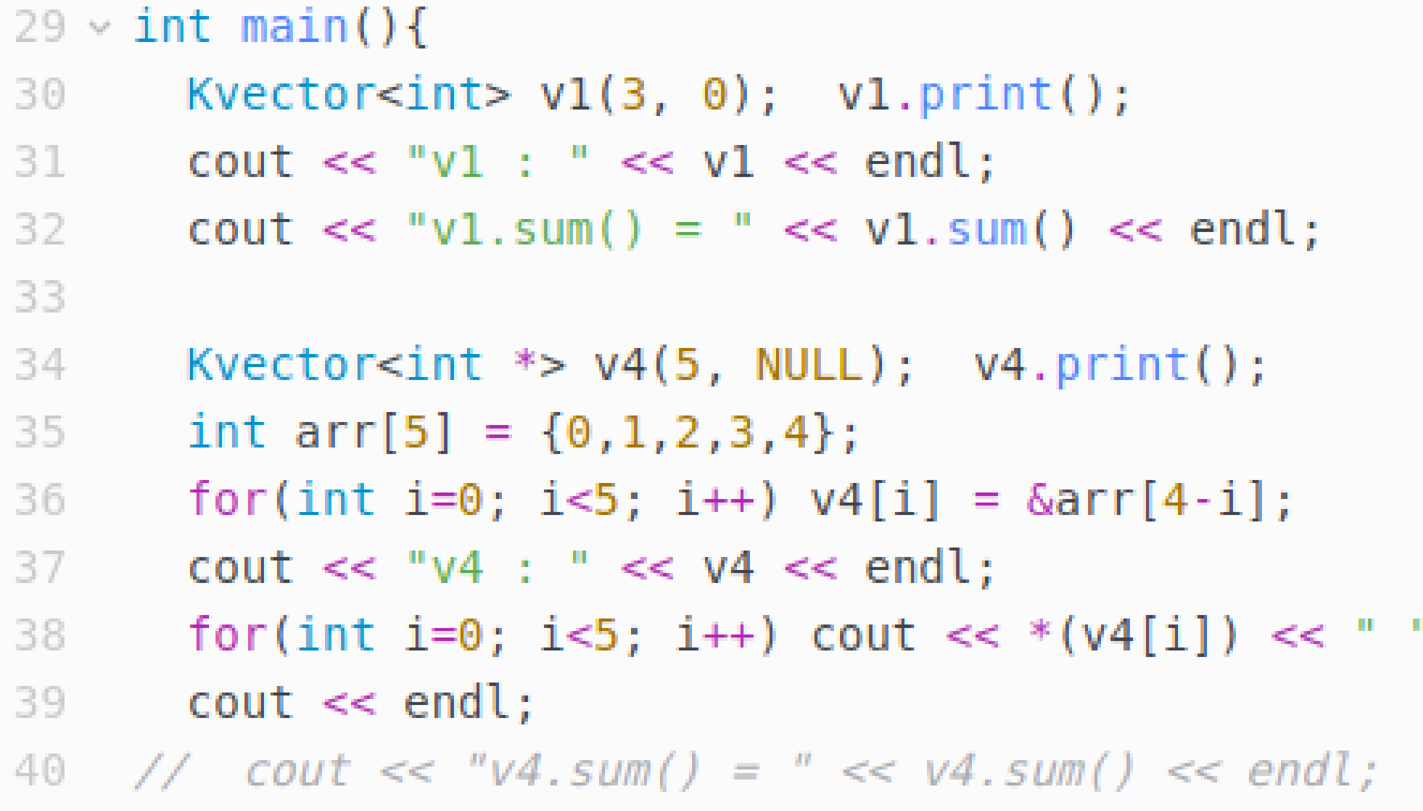


```
T sum() const {
   T s;
   for(int i=0; i<len; i++) s+=m[i];
   return s;
}
```

type T 에 대하여 += 연산자가 정의되어 있어야 한다.

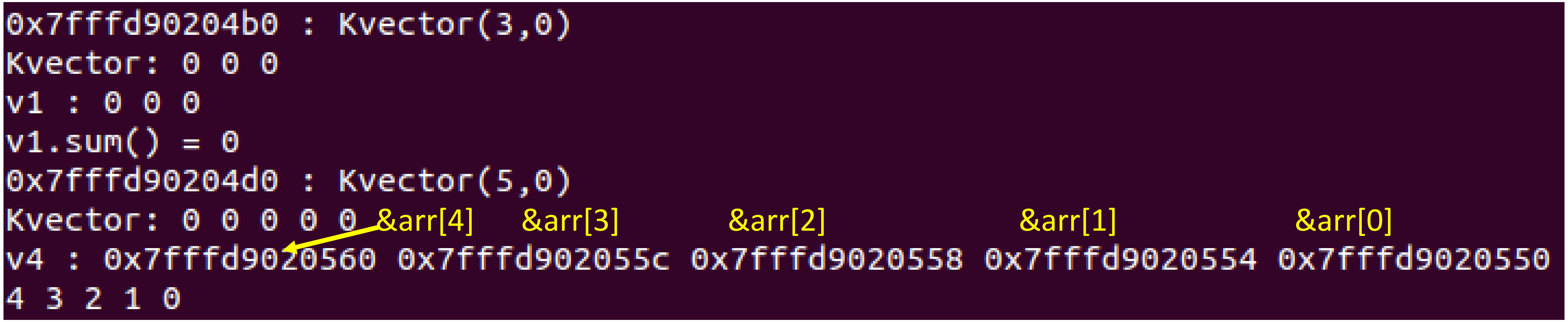


```
// Kvector.cpp template version by ejim@kookmin.ac.kr
//#include <iostream>
//#include "Kvector.h"
using namespace std;
template<class T>
Kvector<T>::Kvector(int sz, T value): len(sz){⋯}
template<class T>
Kvector<T>::Kvector(const Kvector& v){⋯}
template<class T>
Kvector<T>::~Kvector(){⋯}
template<class T>
Kvector<T>& Kvector<T>::operator=(const Kvector& v){⋯}
template<class T>
bool Kvector<T>::operator==(const Kvector& v){⋯}
template<class T>
bool Kvector<T>::operator!=(const Kvector& v){⋯}
```



```
int main(){
  Kvector<int> v1(3, 0);  v1.print();
  cout << "v1 : " << v1 << endl;
  cout << "v1.sum() = " << v1.sum() << endl;

  Kvector<int *> v4(5, NULL);  v4.print();
  int arr[5] = {0,1,2,3,4};
  for(int i=0; i<5; i++) v4[i] = &arr[4-i];
  cout << "v4 : " << v4 << endl;
  for(int i=0; i<5; i++) cout << *(v4[i]) << " ";
  cout << endl;
//  cout << "v4.sum() = " << v4.sum() << endl;
```

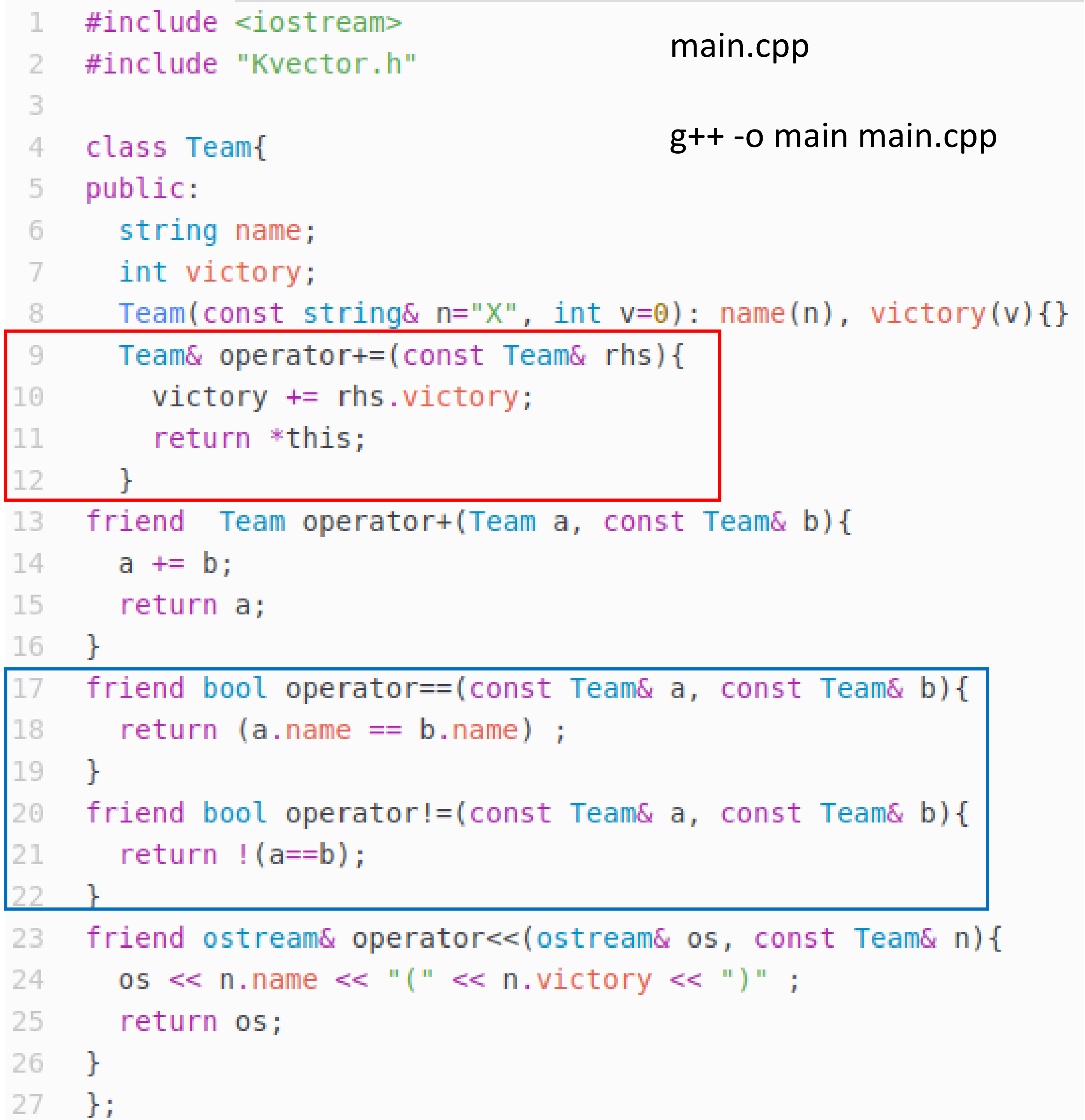
← compile error : pointer 끼리 덧셈 불가



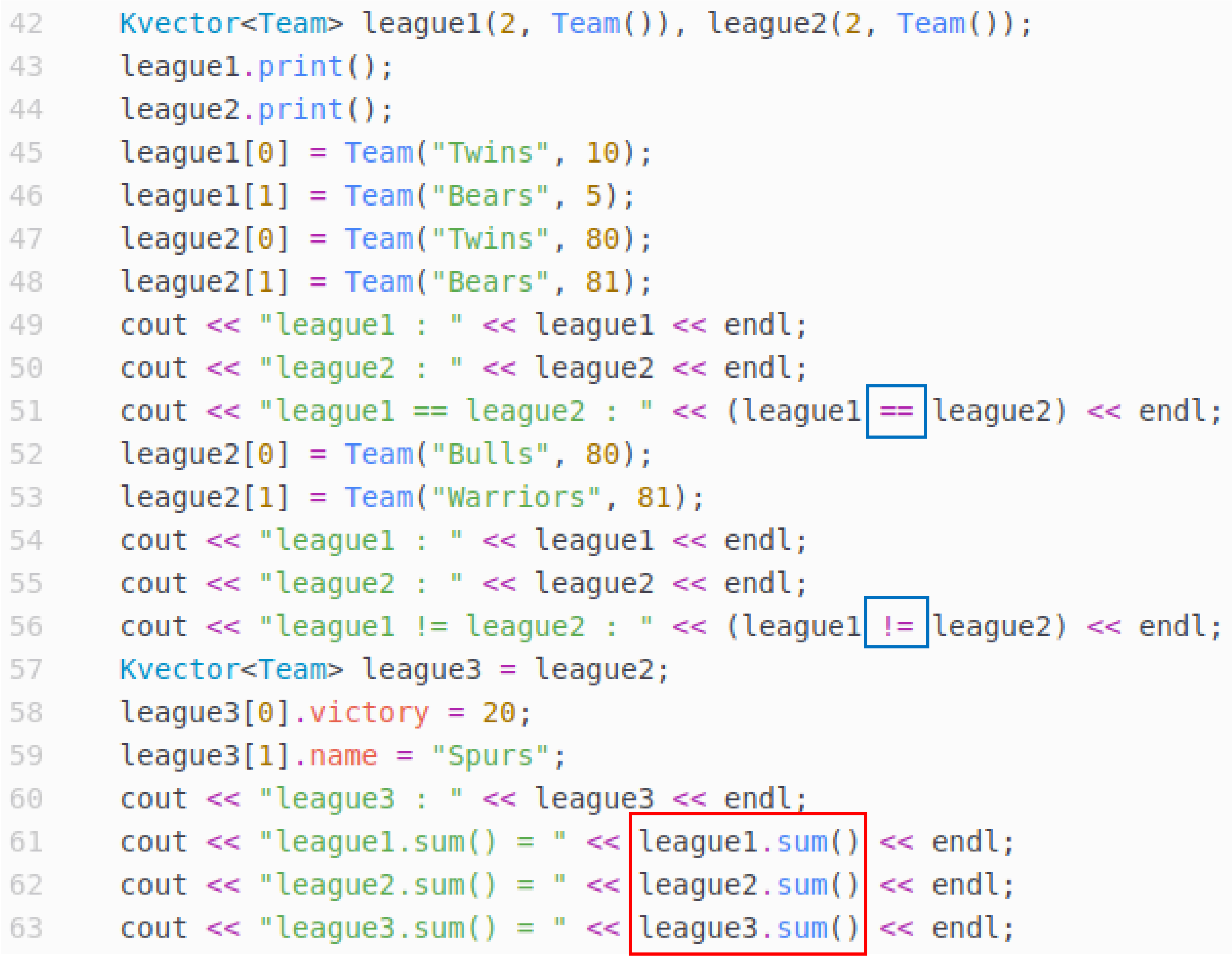
```
0x7fffd90204b0 : Kvector(3,0)
Kvector: 0 0 0
v1 : 0 0 0
v1.sum() = 0
0x7fffd90204d0 : Kvector(5,0)
Kvector: 0 0 0 0 0
v4 : 0x7fffd9020560 0x7fffd902055c 0x7fffd9020558 0x7fffd9020554 0x7fffd9020550
4 3 2 1 0
```

main.cpp

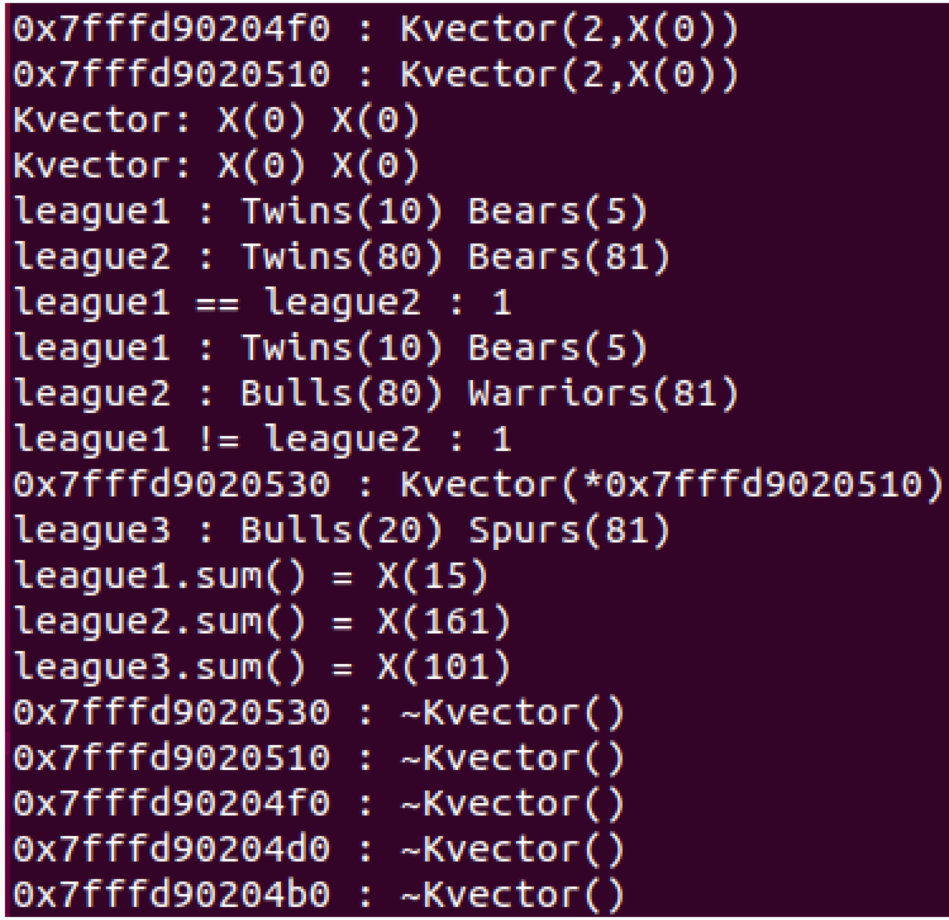
g++ -o main main.cpp



```cpp
#include <iostream>
#include "Kvector.h"

class Team{
public:
  string name;
  int victory;
  Team(const string& n="X", int v=0): name(n), victory(v){}
  Team& operator+=(const Team& rhs){
    victory += rhs.victory;
    return *this;
  }
friend  Team operator+(Team a, const Team& b){
  a += b;
  return a;
}
friend bool operator==(const Team& a, const Team& b){
  return (a.name == b.name) ;
}
friend bool operator!=(const Team& a, const Team& b){
  return !(a==b);
}
friend ostream& operator<<(ostream& os, const Team& n){
  os << n.name << "(" << n.victory << ")" ;
  return os;
}
};
```



```cpp
    Kvector<Team> league1(2, Team()), league2(2, Team());
    league1.print();
    league2.print();
    league1[0] = Team("Twins", 10);
    league1[1] = Team("Bears", 5);
    league2[0] = Team("Twins", 80);
    league2[1] = Team("Bears", 81);
    cout << "league1 : " << league1 << endl;
    cout << "league2 : " << league2 << endl;
    cout << "league1 == league2 : " << (league1 == league2) << endl;
    league2[0] = Team("Bulls", 80);
    league2[1] = Team("Warriors", 81);
    cout << "league1 : " << league1 << endl;
    cout << "league2 : " << league2 << endl;
    cout << "league1 != league2 : " << (league1 != league2) << endl;
    Kvector<Team> league3 = league2;
    league3[0].victory = 20;
    league3[1].name = "Spurs";
    cout << "league3 : " << league3 << endl;
    cout << "league1.sum() = " << league1.sum() << endl;
    cout << "league2.sum() = " << league2.sum() << endl;
    cout << "league3.sum() = " << league3.sum() << endl;
```



```
0x7fffd90204f0 : Kvector(2,X(0))
0x7fffd9020510 : Kvector(2,X(0))
Kvector: X(0) X(0)
Kvector: X(0) X(0)
league1 : Twins(10) Bears(5)
league2 : Twins(80) Bears(81)
league1 == league2 : 1
league1 : Twins(10) Bears(5)
league2 : Bulls(80) Warriors(81)
league1 != league2 : 1
0x7fffd9020530 : Kvector(*0x7fffd9020510)
league3 : Bulls(20) Spurs(81)
league1.sum() = X(15)
league2.sum() = X(161)
league3.sum() = X(101)
0x7fffd9020530 : ~Kvector()
0x7fffd9020510 : ~Kvector()
0x7fffd90204f0 : ~Kvector()
0x7fffd90204d0 : ~Kvector()
0x7fffd90204b0 : ~Kvector()
```