



C++

K-Digital Class 4

C++ FILES AND STREAMS

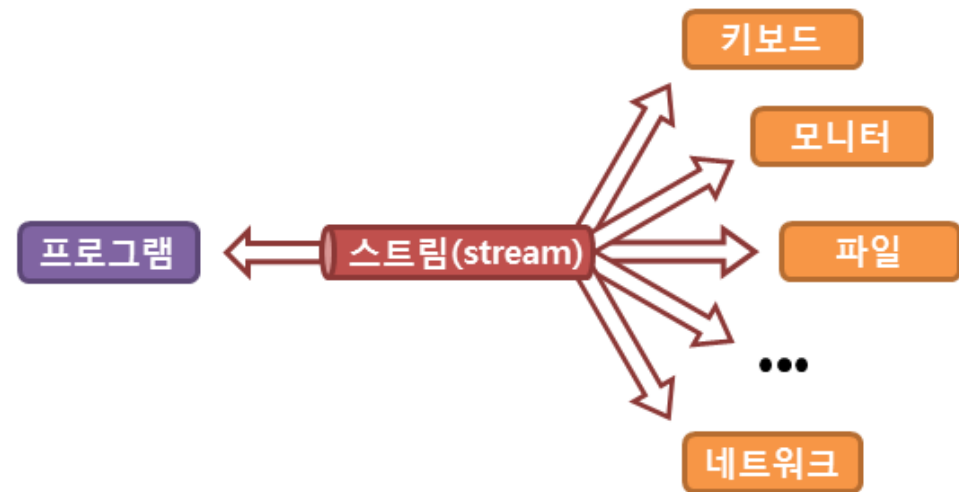
Stream and Buffer

- C++은 C언어와 마찬가지로 입출력에 관한 기능을 언어에서 기본적으로 제공하지 않는다.
- 그 이유는 컴파일러를 만들 때 입출력 기능을 해당 하드웨어에 가장 적합한 형태로 만들 수 있도록 컴파일러가 개발자에게 권한을 주기 위해서다.
- 하지만 대부분의 C++ 컴파일러는 `iostream`과 `fstream` 헤더 파일에 정의되어 있는 클래스 라이브러리를 제공한다.
- `iostream`과 `fstream` 클래스 라이브러리의 중요 개념 중 하나가 바로 스트림(stream)이다.

C++ FILES AND STREAMS

Stream

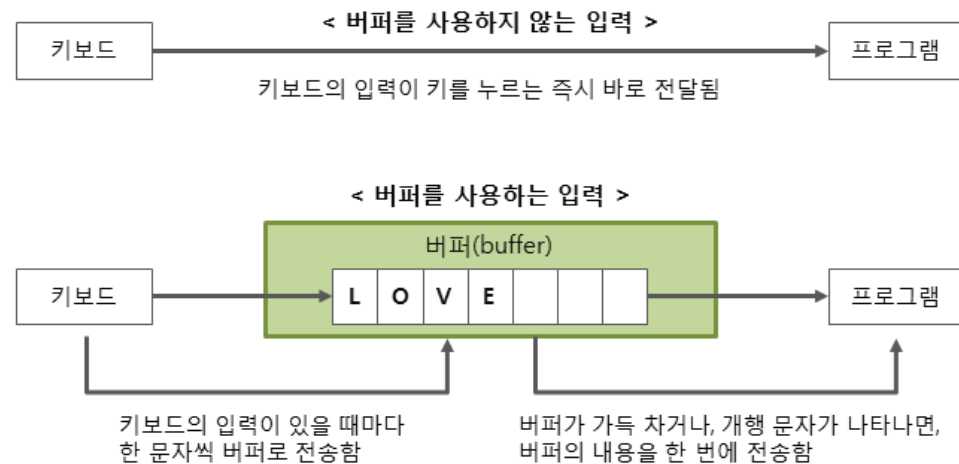
- C++ 프로그램은 파일이나 콘솔의 입출력을 직접 다루지 않고, 스트림(stream)이라는 흐름을 통해 다룬다.
- 스트림(stream)이란 실제의 입력이나 출력이 표현된 데이터의 이상화된 흐름을 의미한다.
- 즉, 스트림은 운영체제에 의해 생성되는 가상의 연결 고리를 의미하며, 중간 매개자 역할을 한다.



C++ FILES AND STREAMS

Buffer

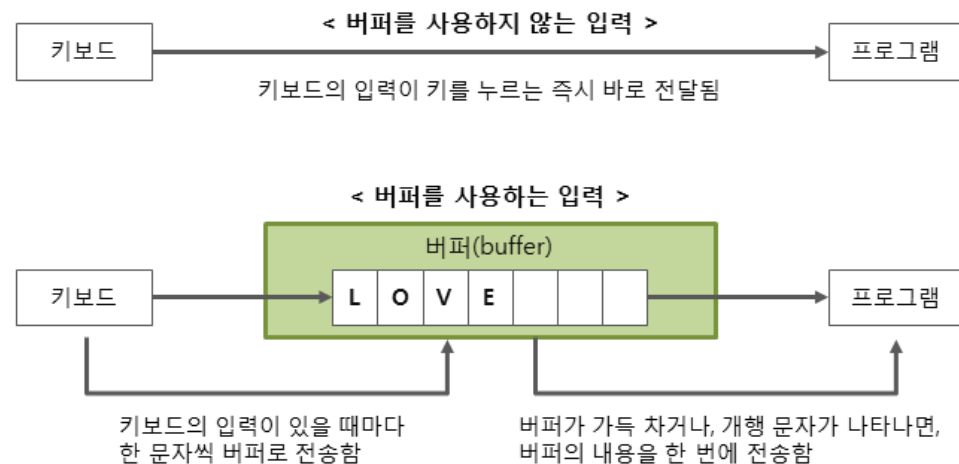
- Stream은 내부에 버퍼(buffer)라는 임시 메모리 공간을 가지고 있다.
- 이러한 버퍼를 이용하면 입력과 출력을 좀 더 효율적으로 처리할 수 있게 된다.



C++ FILES AND STREAMS

Buffer

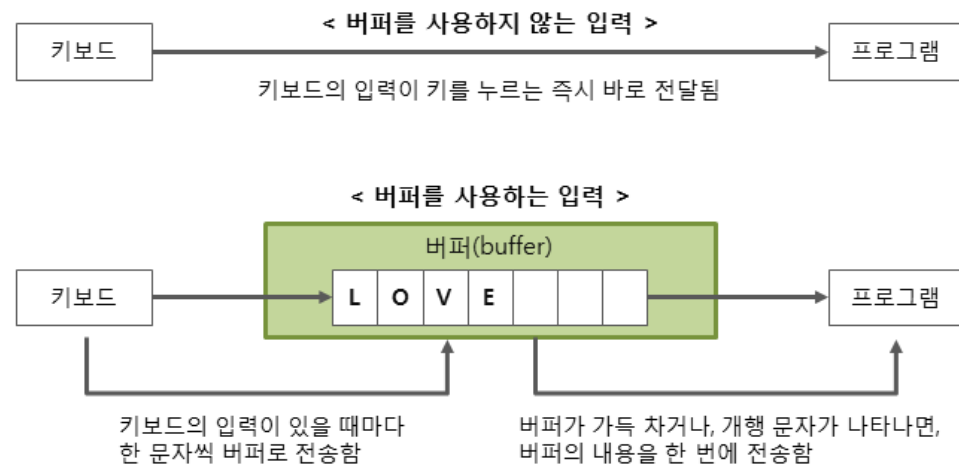
- 버퍼를 사용하면서 얻을 수 있는 장점은 다음과 같다.
 - 문자를 하나씩 전달하는 것이 아닌 묶어서 한 번에 전달하므로, 전송 시간이 적게 걸려 성능이 향상된다.
 - 사용자가 문자를 잘못 입력했을 경우 수정을 할 수가 있다.
- 하지만 입력 작업에 버퍼를 사용하는 것이 반드시 좋은 것만은 아니다.
- 빠른 반응이 요구되는 게임과 같은 프로그램에서는 키를 누르는 즉시 바로 전달 되어야 한다.
- 이렇게 버퍼를 사용하는 입력과 버퍼를 사용하지 않는 입력은 서로 다른 용도로 사용된다.



C++ FILES AND STREAMS

Buffer Synchronization(버퍼 동기화)

- 버퍼가 꽉 찼을 때, 버퍼에 있는 모든 데이터는 물리적인 매체에(만약 출력 스트림이라면) 기록되어지거나(만약 입력 스트림이라면) 단순히 지워집니다. 이러한 처리를 동기화(synchronization)라고 부르고, 다음의 상황에서 일어나게 된다.
- 파일이 닫힐 때 : 모든 버퍼는 읽기 또는 쓰기가 완전히 끝나지 않은 채 닫히려려고 하면 그 이전에 동기화가 이루어진다.
- 버퍼가 꽉 찼을 때 : Buffers 의 크기가 정해져 있고, buffer 가 꽉 찼을 때 자동적으로 동기화가 이루어진다.
- 조작자(manipulators)를 명시했을 때 : 스트림을 동조시키기 위한 명확한 조작자를 사용했을 때 이며, 이들 조작자는 flush 와 endl 이다.
- sync() 함수를 명시했을 때 : 인수없이 sync() 멤버함수를 호출하여 즉시 동기화가 일어나게 할 수 있다. 이 함수는 스트림과 연결된 버퍼가 없거나 오류가 있는 경우에는 -1과 같은 int값을 돌려주게 된다.



C++ FILES AND STREAMS

- 지금까지 standard input(표준 입력)에서 읽고 standard output(표준 출력)으로 읽고 쓰기 위한 cin 및 cout 메서드를 각각 제공하는 iostream 표준 라이브러리(Standard Library)를 사용했다.
- 이번에는 file을 읽고(read) 쓰는(write)하는 방법을 배워보자.

Sr.No	Data Type & Description
1	ofstream This data type represents the output file stream and is used to create files and to write information to files.
2	ifstream This data type represents the input file stream and is used to read information from files.
3	fstream This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

C++ FILES AND STREAMS

- fstream 라이브러리를 사용하려면 표준 <iostream>과 <fstream> 헤더 파일을 모두 포함한다.
- fstream 라이브러리에는 파일을 생성, 쓰기 또는 읽는 데 사용되는 세 가지 클래스가 포함되어 있다.

Example

```
#include <iostream>
#include <fstream>
```


C++ FILES AND STREAMS

Opening a File

- 파일을 읽거나 쓰려면 먼저 파일을 열어(Open)야 한다. Ofstream 또는 fstream 객체는 쓰기 위해 파일을 여는 데 사용할 수 있다. 그리고 ifstream 객체는 읽기 전용으로 파일을 여(Open)는 데 사용된다.
- 다음은 fstream, ifstream 및 ofstream 객체의 멤버인 open() 함수의 표준 구문이다.
- 여기서 첫 번째 매개변수는 열려는 파일의 이름과 위치(file path)를 지정하고 open() 멤버 함수의 두 번째 매개변수는 파일을 열어야 하는 모드(open option)를 정의한다.

```
void open(const char *filename, ios::openmode mode);
```

C++ FILES AND STREAMS

Opening a File

Sr.No	Mode Flag & Description
1	ios::app Append mode. All output to that file to be appended to the end.
2	ios::ate Open a file for output and move the read/write control to the end of the file.
3	ios::in Open a file for reading.
4	ios::out Open a file for writing.
5	ios::trunc If the file already exists, its contents will be truncated before opening the file.

C++ FILES AND STREAMS

Opening a File – C++ use often combination

파일 모드 상수	C언어 파일 모드	설명
<code>ios_base::out ios_base::trunc</code>	"w"	파일을 쓰는 것만이 가능한 모드로 개방함. 파일이 없으면 새 파일을 만들고, 파일이 있으면 해당 파일의 모든 데이터를 지우고 개방함.
<code>ios_base::out ios_base::app</code>	"a"	파일을 쓰는 것만이 가능한 모드로 개방함. 파일이 없으면 새 파일을 만들고, 파일이 있으면 해당 파일의 맨 끝에서부터 데이터를 추가함.
<code>ios_base::in ios_base::out</code>	"r+"	파일을 읽고 쓰는 것이 둘 다 가능한 모드로 개방함.
<code>ios_base::in ios_base::out ios_base::trunc</code>	"w+"	파일을 읽고 쓰는 것이 둘 다 가능한 모드로 개방함. 파일이 없으면 새 파일을 만들고, 파일이 있으면 해당 파일의 모든 데이터를 지우고 개방함.

C++ FILES AND STREAMS

Opening a File

- 이들 값을 OR로 결합하여 둘 이상의 값을 결합할 수 있다. 예를 들어 쓰기 모드에서 파일을 열고 이미 존재하는 파일을 자르고 싶다면 구문은 다음과 같다.
- 비슷한 방법으로 다음과 같이 읽기 및 쓰기 목적으로 파일을 열 수 있다.

```
ofstream outfile;  
outfile.open("file.dat", ios::out | ios::trunc );
```

```
fstream afile;  
afile.open("file.dat", ios::out | ios::in );
```

C++ FILES AND STREAMS

Closing a File

- C++ 프로그램이 종료되면 모든 스트림을 자동으로 플러시하고 할당된 모든 메모리를 해제하고 열려 있는 모든 파일을 닫는다. 그러나 프로그래머는 프로그램이 종료하기 전에 열려 있는 모든 파일을 닫아야 한다.
- 이는 아주 좋은 습관이며 불필요한 메모리 공간을 정리할 수 있다.
- 다음은 `fstream`, `ifstream` 및 `ofstream` 객체의 멤버인 `close()` 함수의 표준 구문입니다.

```
void close();
```

C++ FILES AND STREAMS

Create and Write To a File

- C++ 프로그래밍을 수행하는 동안 화면에 정보를 출력하기 위해 해당 연산자를 사용하는 것처럼 스트림 삽입 연산자(<<)를 사용하여 프로그램에서 파일에 정보를 쓴다.
- 유일한 차이점은 cout 객체 대신 ofstream 또는 fstream 객체를 사용한다는 것이다.

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

int main() {
    // Create and open a text file
    ofstream MyFile("filename.txt");

    // Write to the file
    MyFile << "Files can be tricky, but it is fun enough!";

    // Close the file
    MyFile.close();
}
```

C++ FILES AND STREAMS

Read a File

- 파일에서 읽으려면 ifstream 또는 fstream 클래스와 파일 이름을 사용한다.
- 파일을 한 줄씩 읽고 파일 내용을 인쇄하기 위해 (ifstream 클래스에 속하는) getline() 함수와 함께 while 루프도 사용한다.

```
// Create a text string, which is used to output the text file
string myText;

// Read from the text file
ifstream MyReadFile("filename.txt");

// Use a while loop together with the getline() function to read the file line by line
while (getline (MyReadFile, myText)) {
    // Output the text from the file
    cout << myText;
}

// Close the file
MyReadFile.close();
```

C++ FILES AND STREAMS

Read and Write Example

- 다음은 읽기 및 쓰기 모드에서 파일을 여는 C++ 프로그램이다.
- 사용자가 입력한 정보를 afile.dat라는 파일에 쓴 후 프로그램은 파일에서 정보를 읽어 화면에 출력한다.

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

int main () {
    char data[100];

    // open a file in write mode.
    ofstream outfile;
    outfile.open("afile.dat");

    cout << "Writing to the file" << endl;
    cout << "Enter your name: ";
    cin.getline(data, 100);

    // write inputted data into the file.
    outfile << data << endl;

    cout << "Enter your age: ";
    cin >> data;
    cin.ignore();

    // again write inputted data into the file.
    outfile << data << endl;

    // close the opened file.
    outfile.close();

    // open a file in read mode.
    ifstream infile;
    infile.open("afile.dat");

    cout << "Reading from the file" << endl;
    infile >> data;

    // write the data at the screen.
    cout << data << endl;

    // again read the data from the file and display it.
    infile >> data;
    cout << data << endl;

    // close the opened file.
    infile.close();

    return 0;
}
```


C++ FILES AND STREAMS

File Position Pointers

- ifstream과 ofstream은 모두 파일 위치 포인터를 재 배치하기 위한 멤버 함수를 제공한다.
- 이러한 멤버 함수는 ifstream의 경우 seekg(" seek get ")이고 ofstream의 경우 seekp(" seek put ")이다.
- Seekg 및 seekp에 대한 인수는 일반적으로 long integer입니다. 탐색 방향을 나타내기 위해 두 번째 인수를 지정할 수 있다.
- 탐색 방향은 스트림의 시작 부분을 기준으로 위치 지정을 위한 ios::beg(default)일 수 있다.
- 스트림의 현재 위치를 기준으로 위치를 지정하려면 ios::cur를, 스트림 끝을 기준으로 위치를 지정하려면 ios::end를 사용한다.

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

C++ EXCEPTION HANDLING

- Exception(예외)는 Handler라는 특수 기능에 제어를 전달하여 프로그램의 예외적인 상황(예: 런타임 오류)에 대응하는 방법을 제공한다.
- 예외(Exception)를 잡기(catch) 위해 코드의 일부가 예외 검사를 받는다. 이는 코드의 해당 부분을 try 블록으로 묶음으로써 수행된다. 해당 블록 내에서 예외적인 상황이 발생하면 예외 처리기로 제어를 전송하는 예외가 발생한다. 예외가 발생하지 않으면 코드가 정상적으로 계속되고 모든 예외 처리기가 무시된다.
- try 블록 내부에서 throw 키워드를 사용하면 예외가 발생한다. Exception Handler는 catch 키워드로 선언되며 try 블록 바로 뒤에 위치해야 한다.

C++ EXCEPTION HANDLING

C++ try and catch

- try 문을 사용하면 실행되는 동안 오류를 테스트할 코드 블록을 정의할 수 있다.
- throw 키워드는 문제를 감지하면 예외를 throw하므로 사용자 지정 오류를 생성할 수 있다.
- catch 문을 사용하면 try 블록에서 오류가 발생할 경우 실행할 코드 블록을 정의할 수 있다.
- try 및 catch 키워드는 쌍으로 제공된다.

```
try {  
    // Block of code to try  
    throw exception; // Throw an exception when a problem arise  
}  
catch () {  
    // Block of code to handle errors  
}
```

C++ EXCEPTION HANDLING

Throwing Exceptions

- 예외는 throw 문을 사용하여 코드 블록 내 어디에서나 throw될 수 있다. throw 문의 피연산자는 예외의 유형을 결정하고 모든 표현식이 될 수 있으며 표현식의 결과 유형은 throw된 예외의 유형을 결정한다.
- 다음은 0으로 나누는 조건이 발생한 경우 예외를 throw하는 예이다.

```
double division(int a, int b) {  
    if( b == 0 ) {  
        throw "Division by zero condition!";  
    }  
    return (a/b);  
}
```

C++ EXCEPTION HANDLING

Catching Exceptions

- try 블록 다음에 오는 catch 블록은 모든 예외를 catch한다. catch할 예외 유형을 지정할 수 있으며 이는 catch 키워드 다음에 괄호 안에 나타나는 예외 선언에 의해 결정된다.

```
try {  
    // protected code  
} catch( ExceptionName e ) {  
    // code to handle ExceptionName exception  
}
```

C++ EXCEPTION HANDLING

Handle Any Type of Exceptions (...)

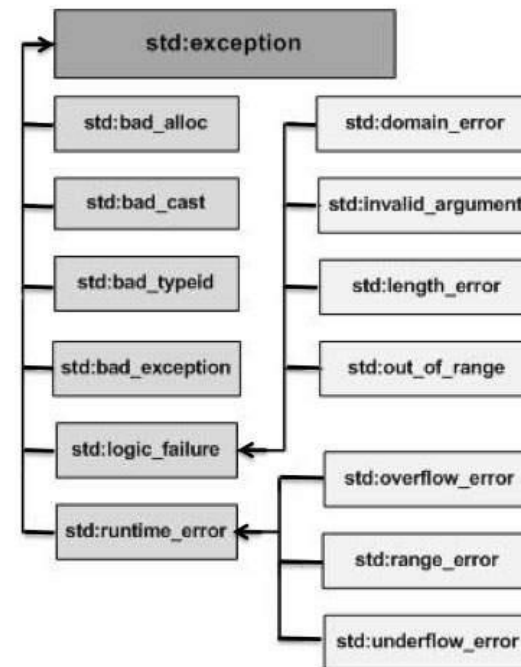
- 앞의 코드는 ExceptionName 유형의 예외를 catch합니다. catch 블록이 try 블록에서 throw된 모든 유형의 예외를 처리하도록 지정하려면 다음과 같이 예외 선언을 묶는 괄호 사이에 줄임표 ...를 넣어야 합니다.

```
try {  
    // protected code  
} catch(...) {  
    // code to handle any exception  
}
```

C++ EXCEPTION HANDLING

C++ Standard Exceptions

- C++는 우리 프로그램에서 사용할 수 있는 <exception>에 정의된 standard exception list를 제공한다. 이들은 그림에 표시된 부모-자식 클래스 계층 구조로 정렬된다.



C++ EXCEPTION HANDLING

C++ Standard Exceptions

Sr.No	Exception & Description
1	<code>std::exception</code> 모든 표준 C++ 예외의 예외 및 상위 클래스이다.
2	<code>std::bad_alloc</code> 이것은 <code>new</code> 에 의해 던질 수 있다
3	<code>std::bad_cast</code> 이것은 <code>dynamic_cast</code> 에 의해 던질 수 있다
4	<code>std::bad_exception</code> 이것은 C++ 프로그램에서 예기치 않은 예외를 처리하는 데 유용한 장치이다.
5	<code>std::bad_typeid</code> 이것은 <code>typeid</code> 에 의해 던질 수 있다.
6	<code>std::logic_error</code> 이론적으로 코드를 읽어 감지할 수 있는 예외이다.
7	<code>std::domain_error</code> 이것은 수학적으로 유효하지 않은 도메인이 사용될 때 발생하는 예외이다.

Sr.No	Exception & Description
8	<code>std::invalid_argument</code> 잘못된 인수로 인해 발생한다.
9	<code>std::length_error</code> 이것은 너무 큰 <code>std::string</code> 이 생성될 때 발생한다.
10	<code>std::out_of_range</code> 이것은 'at' 메서드에 의해 발생할 수 있다 (예: <code>std::vector</code> 및 <code>std::bitset<>::operator[]()</code>).
11	<code>std::runtime_error</code> 이론적으로 코드를 읽어 감지할 수 없는 예외이다.
12	<code>std::overflow_error</code> 수학적 오버플로가 발생하면 throw된다.
13	<code>std::range_error</code> 범위를 벗어난 값을 저장하려고 할 때 발생한다.
14	<code>std::underflow_error</code> 수학적 언더플로가 발생하면 throw된다.

C++ EXCEPTION HANDLING

Define New Exceptions

- exception class 기능을 inheriting 및 overriding하여 고유한 예외를 정의할 수 있다.
- 다음은 std::exception class를 사용하여 표준 방식으로 자신의 예외를 구현하는 예제이다.

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

struct MyException : public exception {
    const char * what () const throw () {
        return "C++ Exception";
    }
};

int main() {
    try {
        throw MyException();
    } catch(MyException& e) {
        std::cout << "MyException caught" << std::endl;
        std::cout << e.what() << std::endl;
    } catch(std::exception& e) {
        //Other errors
    }
}
```

C++ TEMPLATES

Generic programming

- C++이 가지는 프로그래밍 언어로서의 특징 중 하나로 일반화 프로그래밍 (generic programming)을 들 수 있다.
- 일반화 프로그래밍은 데이터를 중시하는 객체 지향 프로그래밍과는 달리 프로그램의 알고리즘에 그 중점을 둔다.
- 이러한 일반화 프로그래밍을 지원하는 C++의 대표적인 기능 중 하나가 바로 템플릿(template)이다.

Template

- 템플릿(template)이란 매개변수의 타입에 따라 함수나 클래스를 생성하는 메커니즘을 의미한다.
- 템플릿은 타입이 매개변수에 의해 표현되므로, 매개변수화 타입 (parameterized type)이라고도 불린다.
- 템플릿을 사용하면 타입마다 별도의 함수나 클래스를 만들지 않고, 여러 타입에서 동작할 수 있는 단 하나의 함수나 클래스를 작성하는 것이 가능하다.

C++ TEMPLATES

Function Template

- C++에서 함수 템플릿(function template)이란 함수의 일반화된 선언을 의미한다.
- 함수 템플릿을 사용하면 같은 알고리즘을 기반으로 하면서, 서로 다른 타입에서 동작하는 함수를 한 번에 정의할 수 있다.
- 임의의 타입으로 작성된 함수에 특정 타입을 매개변수로 전달하면, C++ 컴파일러는 해당 타입에 맞는 함수를 생성해 준다.
- C++에서 함수 템플릿은 다음과 같은 문법으로 정의한다.

```
template <class type> ret-type func-name(parameter list) {  
    // body of function  
}
```

C++ TEMPLATES

함수 템플릿의 인스턴스화

- 함수 템플릿이 각각의 타입에 대해 처음으로 호출될 때, C++ 컴파일러는 해당 타입의 인스턴스를 생성한다.
- 이렇게 생성된 인스턴스는 해당 타입에 대해 특수화된 템플릿 함수이다.
- 이 인스턴스는 함수 템플릿에 해당 타입이 사용될 때마다 호출된다.

C++ TEMPLATES

Class Template

- C++에서 클래스 템플릿(class template)이란 클래스의 일반화(Generic)된 선언을 의미한다.
- 앞서 살펴본 함수 템플릿과 동작은 같으며, 그 대상이 함수가 아닌 클래스라는 점만 다르다.
- 클래스 템플릿을 사용하면, 타입에 따라 다르게 동작하는 클래스 집합을 만들 수 있다.
- 즉, 클래스 템플릿에 전달되는 템플릿 인수(template argument)에 따라 별도의 클래스를 만들 수 있게 된다.
- 이러한 템플릿 인수는 타입이거나 명시된 타입의 상수값일 수 있다.
- C++에서 클래스 템플릿은 다음과 같은 문법으로 정의할 수 있다.

```
template <class type> class class-name {  
    .  
    .  
    .  
}
```

C++ TEMPLATES

Nested class template(중첩 클래스 템플릿)

- C++에서는 클래스나 클래스 템플릿 내에 또 다른 템플릿을 중첩하여 정의할 수 있으며, 이러한 템플릿을 멤버 템플릿(member template)이라고 한다. 멤버 템플릿 중에서도 클래스 템플릿을 중첩 클래스 템플릿(nested class template)이라고 한다.
- 이러한 중첩 클래스 템플릿은 바깥쪽 클래스의 범위 내에서 클래스 템플릿으로 선언되며, 정의는 바깥쪽 클래스의 범위 내에서 뿐만 아니라 범위 밖에서도 가능하다.

```
template <typename T>
class X
{
    template <typename U>
    class Y
    {
        ...
    }
    ...
}

int main(void)
{
    ...
}

template <typename T>
template <typename U>
X<T>::Y<U>::멤버함수이름()
{
    ...
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

- C++이 가지는 프로그래밍 언어로서의 특징 중 하나로 일반화 프로그래밍(generic programming)을 들 수 있다.
- 이러한 일반화 프로그래밍은 데이터를 중시하는 객체 지향 프로그래밍과는 달리 프로그램의 알고리즘에 그 중점을 둔다.
- C++ 표준 템플릿 라이브러리인 STL도 이러한 일반화 프로그래밍 패러다임의 한 축을 담당하고 있다.
- STL은 알고리즘을 일반화한 표현을 제공하여, 데이터의 추상화와 코드를 재활용할 수 있게 한다.
- STL은 1994년 휴렛팩커드연구소의 알렉스 스테파노프(Alex Stepanov)와 멩 리(Meng Lee)가 처음으로 그 구현을 발표한다.
- 그 후 STL은 ISO/ANSI C++ 표준 위원회에 의해 C++ 표준 템플릿 라이브러리로 포함되게 된다.

C++ STL(STANDARD TEMPLATE LIBRARY)

STL Components

- 컨테이너(container)
 - 컨테이너는 특정 종류의 객체 컬렉션(Object collection)을 관리하는 데 사용된다.
 - Deque, list, vector, map 등과 같은 여러 유형의 컨테이너가 있다.
- 반복자(iterator)
 - 반복자는 객체 컬렉션(Object collection)의 구성요소를 단계별로 실행하는 데 사용된다.
 - 이러한 컬렉션은 컨테이너 또는 컨테이너의 하위 집합일 수 있다.
- 알고리즘(algorithm)
 - 알고리즘은 컨테이너에서 작동한다. 반복자를 이용해서 컨테이너 내용의 초기화, 정렬, 검색 및 변환을 수행하는 수단을 제공한다.

C++ STL(STANDARD TEMPLATE LIBRARY)

Container

- STL에서 컨테이너(container)는 같은 타입의 여러 객체를 저장하는 일종의 집합이라 할 수 있다.
- 컨테이너는 클래스 템플릿으로, 컨테이너 변수를 선언할 때 컨테이너에 포함할 요소의 타입을 명시할 수 있다.
- C++ STL 에서 컨테이너는 크게 두 가지 종류가 있다. 먼저 배열 처럼 객체들을 순차적으로 보관하는 시퀀스 컨테이너 (sequence container) 와 키(key)를 바탕으로 대응되는 값을 찾아주는 연관 컨테이너 (associative container)가 있다.

C++ STL(STANDARD TEMPLATE LIBRARY)

Sequence container

- Sequence container에는 vector, list, deque 이렇게 3개가 정의되어 있다. 먼저 벡터(vector)의 경우, 쉽게 생각하면 가변길이 배열이라 생각하면 된다
- 벡터에는 원소들이 메모리 상에서 실제로 순차적으로 저장되어 있고, 따라서 임의의 위치에 있는 원소를 접근하는 것을 매우 빠르게 수행할 수 있다.
- vector의 임의의 원소에 접근하는 것은 배열처럼 []를 이용하거나, at 함수를 이용하면 된다. 또한 맨 뒤에 원소를 추가하거나 제거하기 위해서는 push_back 혹은 pop_back 함수를 사용하면 된다.
- 벡터의 크기를 return하는 함수인 size의 경우, return하는 값의 타입은 size_type 멤버 타입으로 정의되어 있다.

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

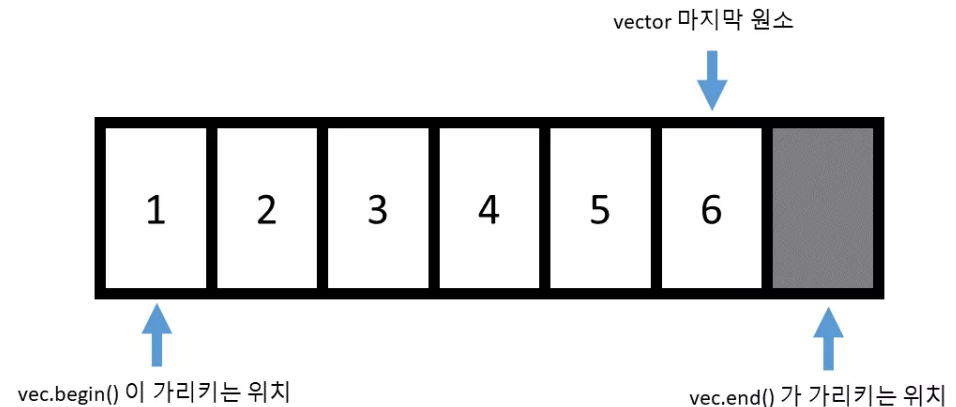
int main() {
    std::vector<int> vec;
    vec.push_back(10); // 맨 뒤에 10 추가
    vec.push_back(20); // 맨 뒤에 20 추가
    vec.push_back(30); // 맨 뒤에 30 추가
    vec.push_back(40); // 맨 뒤에 40 추가

    for (std::vector<int>::size_type i = 0; i < vec.size(); i++) {
        std::cout << "vec 의 " << i + 1 << " 번째 원소 :: " << vec[i] << std::endl;
    }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- iterator는 container의 원소에 접근할 수 있는 포인터와 같은 객체라 할 수 있다. 물론 vector의 경우 `[]`를 이용해서 정수형 변수로 마치 배열처럼 임의의 위치에 접근할 수 있지만 반복자를 사용해서도 마찬가지로 작업을 수행할 수 있다. 특히 algorithm 라이브러리의 경우 대부분이 반복자를 인자로 받아서 algorithm을 수행한다.
- 반복자는 컨테이너에 iterator 멤버 타입으로 정의되어 있다. vector의 경우 반복자를 얻기 위해서는 `begin()` 함수와 `end()` 함수를 사용할 수 있는데 이는 다음과 같은 위치를 반환한다.



C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- begin() 함수는 예상했던 대로, vector 의 첫 번째 원소를 가리키는 반복자를 반환한다. 그런데, 흥미롭게도 end() 의 경우 vector 의 마지막 원소 한 칸 뒤를 가리키는 반복자를 반환하게 된다. 왜 end() 의 경우 vector 의 마지막 원소를 가리키는 것이 아니라, 마지막 원소의 뒤를 가리키는 반복자를 반환할까?
- 이에 여러가지 이유가 있겠지만, 가장 중요한 점이 이를 통해 빈 벡터를 표현할 수 있다는 점이다. 만일 begin() == end() 라면 원소가 없는 벡터를 의미 하게 된다. 만약에 vec.end() 가 마지막 원소를 가리킨다면 비어 있는 벡터를 표현할 수 없게 된다.

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

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);

    // 전체 벡터를 출력하기
    for (std::vector<int>::iterator itr = vec.begin(); itr != vec.end(); ++itr) {
        std::cout << *itr << std::endl;
    }

    // int arr[4] = {10, 20, 30, 40}
    // *(arr + 2) == arr[2] == 30;
    // *(itr + 2) == vec[2] == 30;

    std::vector<int>::iterator itr = vec.begin() + 2;
    std::cout << "3 번째 원소 :: " << *itr << std::endl;
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- vector 의 반복자의 타입은 `std::vector<>::iterator` 멤버 타입으로 정의되어 있고, `vec.begin()` 이나 `vec.end()` 함수가 이를 반환한다. `End()` 가 vector 의 마지막 원소 바로 뒤를 가리키기 때문에 for 문에서 vector 전체 원소를 보고 싶다면 `vec.end()` 가 아닐 때 까지 반복하면 된다.
- 반복자를 마치 포인터처럼 사용한다고 하였는데, 실제로 현재 반복자가 가리키는 원소의 값을 보고 싶다면...

```
std::cout << *itr << std::endl;
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 포인터로 * 를 해서 가리키는 주소 값의 값을 보았던 것처럼, * 연산자를 이용해서 itr 이 가리키는 원소를 볼 수 있다.
- 물론 itr 은 실제 포인터가 아니고 * 연산자를 오버로딩해서 마치 포인터 처럼 동작하게 만든 것이다.
- * 연산자는 itr 이 가리키는 원소의 레퍼런스를 반환한다.

```
std::vector<int>::iterator itr = vec.begin() + 2;  
std::cout << "3 번째 원소 :: " << *itr << std::endl;
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 반복자 역시 + 연산자를 통해서 그만큼 떨어져 있는 원소를 가리키게 할 수도 있다. (그냥 배열을 가리키는 포인터와 정확히 똑같이 동작한다)
- 반복자를 이용하면 예제와 같이 insert 와 erase 함수도 사용할 수 있다.

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

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
        ++itr) {
        std::cout << *itr << std::endl;
    }
}

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);

    std::cout << "처음 벡터 상태" << std::endl;
    print_vector(vec);
    std::cout << "-----" << std::endl;

    // vec[2] 앞에 15 추가
    vec.insert(vec.begin() + 2, 15);
    print_vector(vec);

    std::cout << "-----" << std::endl;
    // vec[3] 제거
    vec.erase(vec.begin() + 3);
    print_vector(vec);
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- vector 에서 반복자로 erase 나 insert 함수를 사용할 때 주의해야할 점이 있다.

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

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    std::cout << "[ ";
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
        ++itr) {
        std::cout << *itr << " ";
    }
    std::cout << "]";
}

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);
    vec.push_back(20);

    std::cout << "처음 벡터 상태" << std::endl;
    print_vector(vec);

    std::vector<int>::iterator itr = vec.begin();
    std::vector<int>::iterator end_itr = vec.end();

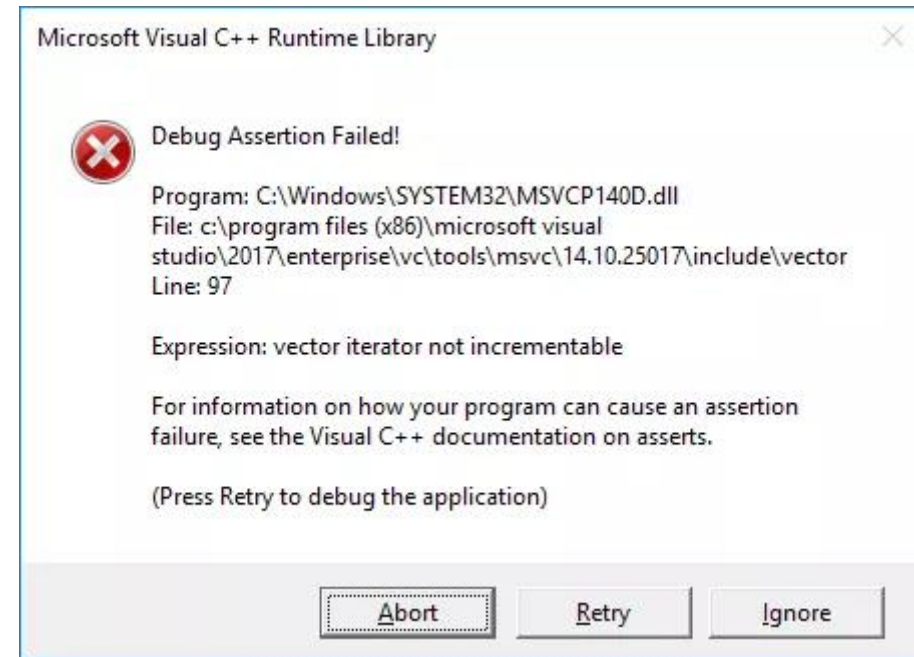
    for (; itr != end_itr; ++itr) {
        if (*itr == 20) {
            vec.erase(itr);
        }
    }

    std::cout << "값이 20 인 원소를 지운다!" << std::endl;
    print_vector(vec);
}
```


C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 컴파일 후 실행하였다면 그림과 같은 오류가 발생한다.



C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 왜 이런 오류가 발생하는 것일까?
- 문제는 바로 위 코드에서 발생한다. 컨테이너에 원소를 추가하거나 제거하게 되면 기존에 사용하였던 모든 반복자들을 사용할 수 없게 된다. 다시 말해 위 경우 `vec.erase(itr)` 을 수행하게 되면 더 이상 `itr` 은 유효한 반복자가 아니게 되는 것이다. 또한 `end_itr` 역시 무효화 된다.
- 따라서 `itr != end_itr` 이 영원히 성립되며 무한 루프에 빠져 위와 같은 오류가 발생한다.
- 결과적으로 코드를 제대로 고치려면 다음과 같이 해야 한다.

```
std::vector<int>::iterator itr = vec.begin();

for (; itr != vec.end(); ++itr) {
    if (*itr == 20) {
        vec.erase(itr);
        itr = vec.begin();
    }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 사실 생각해 보면 위 바뀐 코드는 꽤나 비효율적임을 알 수 있다. 왜냐하면 20 인 원소를 지우고, 다시 처음으로 돌아가서 원소들을 찾고 있기 때문이다. 그냥 20 인 원소 바로 다음 위치 부터 찾으면 될 것이기 때문이다.
- 그렇다면 아예 위처럼 굳이 반복자를 쓰지 않고 erase 함수에만 반복자를 바로 만들어서 전달하면 된다.

```
for (std::vector<int>::size_type i = 0; i != vec.size(); i++) {  
    if (vec[i] == 20) {  
        vec.erase(vec.begin() + i);  
        i--;  
    }  
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 사실 생각해 보면 위 바뀐 코드는 꽤나 비효율적임을 알 수 있다. 왜냐하면 20 인 원소를 지우고, 다시 처음으로 돌아가서 원소들을 찾고 있기 때문이다. 그냥 20 인 원소 바로 다음 위치 부터 찾으려 할 것이기 때문이다.
- 그렇다면 아예 위처럼 굳이 반복자를 쓰지 않고 erase 함수에만 반복자를 바로 만들어서 전달하면 된다.
- 사실 이 방법은 그리 권장하는 방법은 아니다. 기껏 원소에 접근하는 방식은 반복자를 사용하는 것으로 통일하였는데, 위 방법은 이를 모두 깨 버리고 그냥 기존의 배열 처럼 정수형 변수 i 로 원소에 접근하는 것이기 때문이다.
- Algorithm library에서 다시 하도록 한다.

```
for (std::vector<int>::size_type i = 0; i != vec.size(); i++) {  
    if (vec[i] == 20) {  
        vec.erase(vec.begin() + i);  
        i--;  
    }  
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- vector 에서 지원하는 반복자로 const_iterator 가 있다. 이는 마치 const pointer를 생각하면 된다.
- 즉, const_iterator 의 경우 가리키고 있는 원소의 값을 바꿀 수 없다.

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

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
        ++itr) {
        std::cout << *itr << std::endl;
    }
}

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);

    std::cout << "초기 vec 상태" << std::endl;
    print_vector(vec);

    // itr 은 vec[2] 를 가리킨다.
    std::vector<int>::iterator itr = vec.begin() + 2;

    // vec[2] 의 값을 50으로 바꾼다.
    *itr = 50;

    std::cout << "-----" << std::endl;
    print_vector(vec);

    std::vector<int>::const_iterator citr = vec.cbegin() + 2;

    // 상수 반복자가 가리키는 값을 바꿀 수 없다. 불가능!
    *citr = 30;
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- vector 에서 지원하는 반복자로 const_iterator 가 있다. 이는 마치 const pointer를 생각하면 된다.
- 즉, const_iterator 의 경우 가리키고 있는 원소의 값을 바꿀 수 없다.

▲ 컴파일 오류

'citr': you cannot assign to a variable that is const

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

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
        ++itr) {
        std::cout << *itr << std::endl;
    }
}

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);

    std::cout << "초기 vec 상태" << std::endl;
    print_vector(vec);

    // itr 은 vec[2] 를 가리킨다.
    std::vector<int>::iterator itr = vec.begin() + 2;

    // vec[2] 의 값을 50으로 바꾼다.
    *itr = 50;

    std::cout << "-----" << std::endl;
    print_vector(vec);

    std::vector<int>::const_iterator citr = vec.cbegin() + 2;

    // 상수 반복자가 가리키는 값을 바꿀 수 없다. 불가능!
    *citr = 30;
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- const 반복자가 가리키고 있는 값을 바꿀 수 없다고 오류가 발생한다. 주의할 점은, const 반복자의 경우 cbegin() 과 cend() 함수를 이용하여 얻을 수 있다.
- 많은 경우 반복자의 값을 바꾸지 않고 참조만 하는 경우가 많으므로, const iterator 를 적절히 이용하는 것이 좋다.

```
std::vector<int>::const_iterator citr = vec.cbegin() + 2;
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- vector 에서 지원하는 반복자 중 마지막 종류로 역반복자 (reverse iterator) 가 있다.
- 이는 반복자와 똑같지만 벡터 뒤에서 부터 앞으로 거꾸로 간다는 특징이 있다.

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

template <typename T>
void print_vector(std::vector<T>& vec) {
    // 전체 벡터를 출력하기
    for (typename std::vector<T>::iterator itr = vec.begin(); itr != vec.end();
        ++itr) {
        std::cout << *itr << std::endl;
    }
}

int main() {
    std::vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    vec.push_back(40);

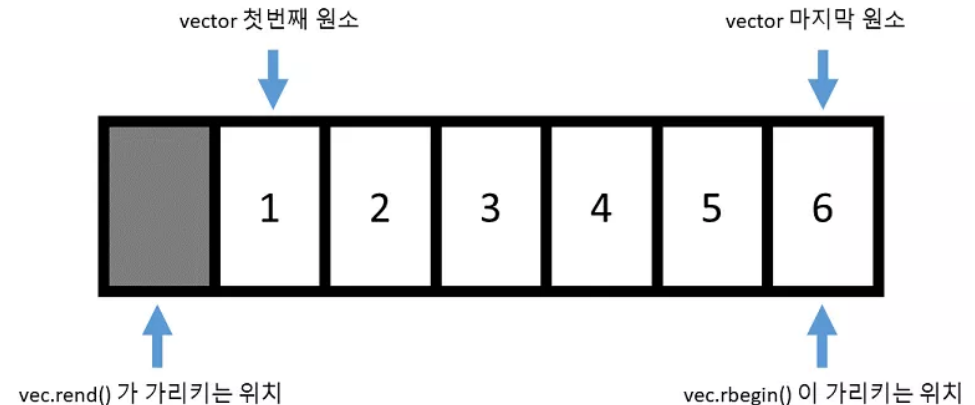
    std::cout << "초기 vec 상태" << std::endl;
    print_vector(vec);

    std::cout << "역으로 vec 출력하기!" << std::endl;
    // itr 은 vec[2] 를 가리킨다.
    std::vector<int>::reverse_iterator r_iter = vec.rbegin();
    for (; r_iter != vec.rend(); r_iter++) {
        std::cout << *r_iter << std::endl;
    }
}
```


C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 앞에서 반복자의 `end()` 가 맨 마지막 원소의 바로 뒤를 가리켰던 것처럼, 역반복자의 `rend()` 역시 맨 앞 원소의 바로 앞을 가리키게 된다. 또한 반복자의 경우 값이 증가하면 뒤쪽 원소로 가는 것처럼, 역반복자의 경우 값이 증가하면 앞쪽 원소로 가게 된다.
- 또 반복자가 상수 반복자가 있는 것처럼 역반복자 역시 상수 역반복자가 있다. 그 타입은 `const_reverse_iterator` 타입이고, `crbegin()`, `crend()` 로 얻을 수 있다.



C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 역반복자를 사용하는 것은 매우 중요하다.

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

int main() {
    std::vector<int> vec;
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);

    // 끝에서 부터 출력하기
    for (std::vector<int>::size_type i = vec.size() - 1; i >= 0; i--) {
        std::cout << vec[i] << std::endl;
    }

    return 0;
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Iterator(반복자)

- 그림과 오류가 발생하게 된다. 맨 뒤의 원소 부터 제대로 출력하는 코드 같은데 왜 이런 문제가 발생하였을까? 그 이유는 vector 의 index 를 담당하는 타입이 부호 없는 정수 이기 때문이다. 따라서 1 가 0 일 때 1 - 를 하게 된다면 -1 이 되는 것이 아니라, 해당 타입에서 가장 큰 정수가 되는 것이다.
- 따라서 for 문이 영원히 종료할 수 없게 되므로, 이 문제를 해결하기 위해서는 부호 있는 정수로 선언해야 하는데, 이 경우 vector 의 index 타입과 일치하지 않아서 타입 캐스팅을 해야 한다는 문제가 발생하게 된다.
- 따라서 가장 현명한 선택으로는 역으로 원소를 참조하고 싶다면, 역반복자를 사용하는 것이다.

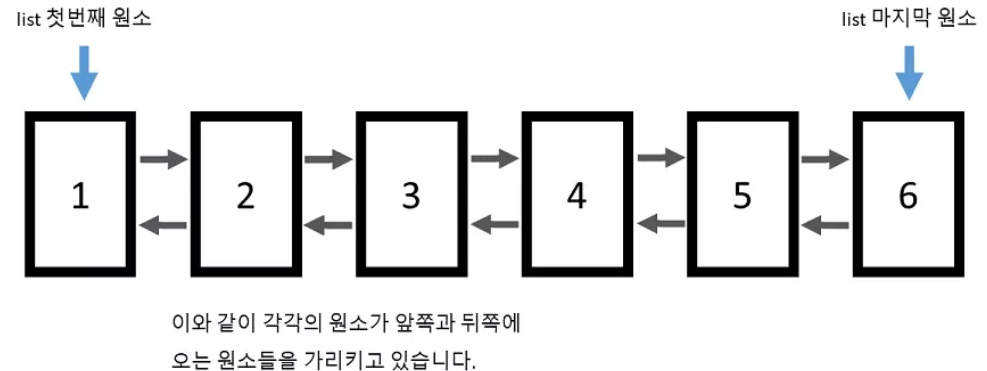
실행 결과

```
3
2
1
// ... (생략) ...
0
0
0
0
1
0
593
0
0
[1] 22180 segmentation fault (core dumped) ./test
```

C++ STL(STANDARD TEMPLATE LIBRARY)

list

- 리스트(list) 의 경우 양방향 연결 구조를 가진 자료형이라 볼 수 있다.
- vector 와는 달리 임의의 위치에 있는 원소에 접근을 바로 할 수 없다. list 컨테이너 자체에서는 시작 원소와 마지막 원소의 위치만을 기억하기 때문에, 임의의 위치에 있는 원소에 접근하기 위해서는 하나씩 링크를 따라가야 한다.
- 그래서 리스트에는 아예 [] 나 at 함수가 아예 정의되어 있지 않다.



C++ STL(STANDARD TEMPLATE LIBRARY)

list

- 리스트(list) 의 경우 양방향 연결 구조를 가진 자료형이라 볼 수 있다.
- vector 와는 달리 임의의 위치에 있는 원소에 접근을 바로 할 수 없다. list 컨테이너 자체에서는 시작 원소와 마지막 원소의 위치만을 기억하기 때문에, 임의의 위치에 있는 원소에 접근하기 위해서는 하나씩 링크를 따라가야 한다.
- 그래서 리스트에는 아예 [] 나 at 함수가 아예 정의되어 있지 않다.

```
#include <iostream>
#include <list>

int main() {
    std::list<int> lst;

    lst.push_back(10);
    lst.push_back(20);
    lst.push_back(30);
    lst.push_back(40);

    for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
        std::cout << *itr << std::endl;
    }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

list

- 리스트의 반복자의 경우 다음과 같은 연산밖에 수행할 수 없다.
 - `itr++`, `itr--`
 - `itr * 5` 같은 연산은 불가능하다. 즉 임의의 위치에 있는 원소를 가리킬 수 없다. 반복자는 오직 하나씩 밖에 움직일 수 없다.
 - 이와 같은 이유는 `list`의 구조를 생각해보면 알 수 있습니다. 앞서 말했듯이 리스트는 왼쪽 혹은 오른쪽 쪽을 가리키고 있는 원소들의 모임으로 이루어져 있기 때문에, 한 번에 한 칸 씩 밖에 이동할 수 없다. 즉, 메모리 상에서 원소들이 연속적으로 존재하지 않을 수 있다는 뜻이다.
 - 반면에 벡터의 경우 메모리 상에서 연속적으로 존재하기 때문에 쉽게 임의의 위치에 있는 원소를 참조할 수 있다.

```
#include <iostream>
#include <list>

int main() {
    std::list<int> lst;

    lst.push_back(10);
    lst.push_back(20);
    lst.push_back(30);
    lst.push_back(40);

    for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
        std::cout << *itr << std::endl;
    }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

list

- list 에서 정의되는 반복자의 타입은 BidirectionalIterator 타입이다. 이름에서도 알 수 있듯이 양방향으로 이동할 수 있되, 한 칸 씩 밖에 이동할 수 없다. 반면에 벡터에서 정의되는 반복자의 타입은 RandomAccessIterator 타입이다.
- 즉, 임의의 위치에 접근할 수 있는 반복자 이다. (참고로 RandomAccessIterator 는 BidirectionalIterator 를 상속 받고 있다)
- 리스트의 반복자는 BidirectionalIterator 이기 때문에 ++ 과 -- 연산만 사용 가능하다. 따라서 for 문으로 하나, 하나 원소를 확인 해보는 것은 가능하다.
- 마찬가지로 erase 함수를 이용하여 원하는 위치에 있는 원소를 지울 수 도 있다. 리스트의 경우는 벡터와는 다르게, 원소를 지워도 반복자가 무효화 되지 않는다. 왜냐하면, 각 원소들의 주소 값들은 바뀌지 않기 때문이다.

```
#include <iostream>
#include <list>

template <typename T>
void print_list(std::list<T>& lst) {
    std::cout << "[ ";
    // 전체 리스트를 출력하기 (이 역시 범위 기반 for 문을 쓸 수 있습니다)
    for (const auto& elem : lst) {
        std::cout << elem << " ";
    }
    std::cout << "]" << std::endl;
}

int main() {
    std::list<int> lst;

    lst.push_back(10);
    lst.push_back(20);
    lst.push_back(30);
    lst.push_back(40);

    std::cout << "처음 리스트의 상태 " << std::endl;
    print_list(lst);

    for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
        // 만일 현재 원소가 20 이라면
        // 그 앞에 50 을 삽입한다.
        if (*itr == 20) {
            lst.insert(itr, 50);
        }
    }

    std::cout << "값이 20 인 원소 앞에 50 을 추가 " << std::endl;
    print_list(lst);

    for (std::list<int>::iterator itr = lst.begin(); itr != lst.end(); ++itr) {
        // 값이 30 인 원소를 삭제한다.
        if (*itr == 30) {
            lst.erase(itr);
            break;
        }
    }

    std::cout << "값이 30 인 원소를 제거한다" << std::endl;
    print_list(lst);
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

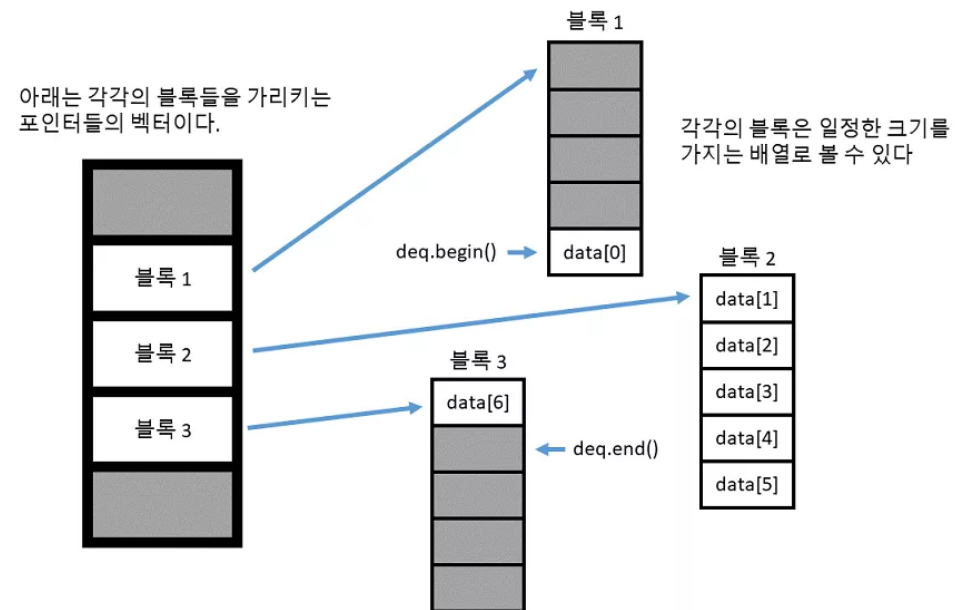
deque - double ended queue

- deque는 벡터와 비슷하게 임의의 위치의 원소에 접근할 수 있으며 맨 뒤에 원소를 추가/제거 하는 작업도 수행할 수 있다. 또한 벡터와는 다르게 맨 앞에 원소를 추가/제거 하는 작업 까지도 수행 가능하다.
- 벡터와는 다르게 deque의 경우 원소들이 실제로 메모리 상에서 연속적으로 존재하지는 않는다. 이 때문에 원소들이 어디에 저장되어 있는지에 대한 정보를 보관하기 위해 추가적인 메모리가 더 필요로 한다.
- 실제 예로, 64 비트 libc++ 라이브러리의 경우 1 개의 원소를 보관하는 deque은 그 원소 크기에 비해 8 배나 더 많은 메모리를 필요로 한다.
- 즉 deque 은 실행 속도를 위해 메모리를 (많이) 희생하는 컨테이너이다.

C++ STL(STANDARD TEMPLATE LIBRARY)

deque - double ended queue

- 그림은 deque의 구조이다. 일단, 벡터와는 다르게 원소들이 메모리에 연속되어 존재하는 것이 아니라 일정 크기로 잘라서 각각의 블록 속에 존재한다. 따라서 이 블록들이 메모리 상에 어느 곳에 위치하여 있는지 저장하기 위해서 각각의 블록들의 주소를 저장하는 벡터가 필요로 한다.
- 참고로 이 벡터는 기존의 벡터와는 조금 다르게, 새로 할당 시에 앞쪽 및 뒤쪽 모두에 공간을 남겨놓게 된다. (벡터의 경우 뒤쪽에만 공간이 남았다) 따라서 이를 통해 맨 앞과 맨 뒤에 매우 빠른($O(1)$)의 속도로 insert 및 erase를 수행할 수 있는 것이다.



C++ STL(STANDARD TEMPLATE LIBRARY)

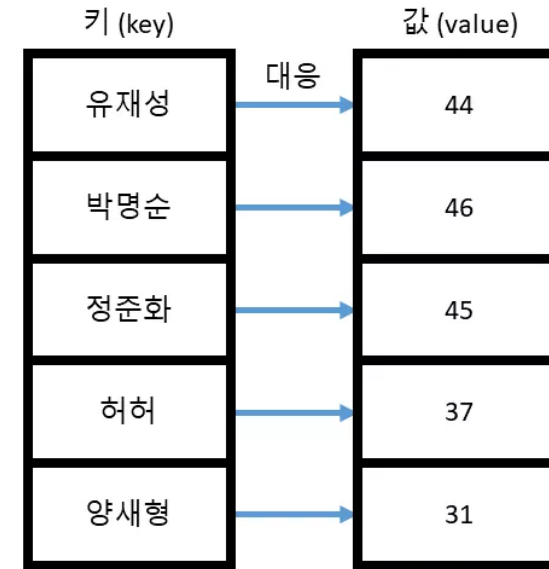
What choose container ?

- 어떠한 컨테이너를 사용할지는 전적으로 이 컨테이너를 가지고 어떠한 작업들을 많이 하는가에 달려있다.
- 일반적인 상황에서는 그냥 벡터를 사용한다 (거의 만능이다!)
- 만약에 맨 끝이 아닌 중간에 원소들을 추가하거나 제거하는 일을 많이 하고, 원소들을 순차적으로만 접근 한다면 리스트를 사용한다.
- 만약에 맨 처음과 끝 모두에 원소들을 추가하는 작업을 많이 하면 deque를 사용한다.

C++ STL(STANDARD TEMPLATE LIBRARY)

Associative container

- 연관 컨테이너는 시퀀스 컨테이너와는 다르게 키(key) - 값(value) 구조를 가진다. 다시 말해 특정한 키를 넣으면 이에 대응되는 값을 돌려준다는 것이다.
- 물론 템플릿 라이브러리이기 때문이 키와 값 모두 임의의 타입의 객체가 될 수 있다.

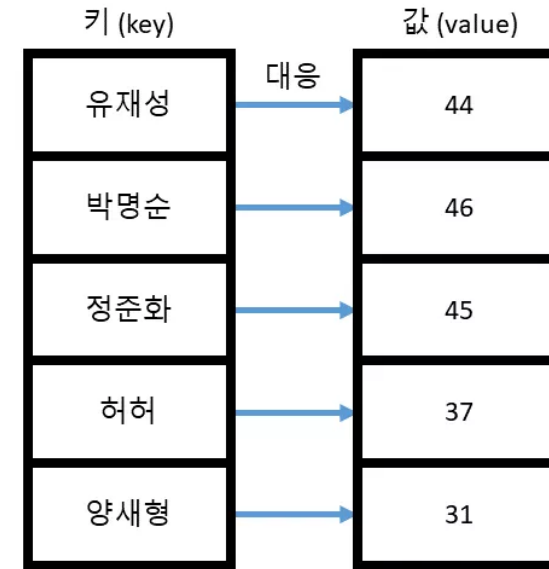


이름(key)을 바탕으로 나이(value)를 얻을 수 있다.

C++ STL(STANDARD TEMPLATE LIBRARY)

Associative container

- 연관 컨테이너는 세트(set)와 멀티세트(multiset), 맵(map), 멀티맵(multimap)을 지원한다. 물론 맵과 멀티맵을 셋처럼 사용할 수 있다. 왜냐하면 해당하는 키가 맵에 존재하지 않으면 당연히 대응되는 값을 가져올 수 없기 때문이다.
- 하지만 맵의 경우 셋 보다 사용하는 메모리가 크기 때문에 키의 존재 유무만 궁금하다면 세트를 사용하는 것이 좋다.



이름(key)을 바탕으로 나이(value)를 얻을 수 있다.

C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- Set에 원소를 추가하기 위해서는 Sequence container 처럼 insert 함수를 사용하면 된다. 한 가지 다른 점은, Sequence container 처럼 '어디에' 추가할 지에 대한 정보가 없다는 점이다.
- Sequence container가 상자 하나에 원소를 한 개씩 담고, 각 상자에 번호를 매긴 것이라면, 세트는 그냥 큰 상자 안에 모든 원소들을 쑤셔 넣은 것이라 보면 된다. 그 상자 안에 원소가 어디에 있는지는 중요한 것이 아니고, 그 상자 안에 원소가 '있냐/없냐' 만이 중요한 정보이다.

```
#include <iostream>
#include <set>

template <typename T>
void print_set(std::set<T>& s) {
    // 셋의 모든 원소들을 출력하기
    std::cout << "[ ";
    for (typename std::set<T>::iterator itr = s.begin(); itr != s.end(); ++itr) {
        std::cout << *itr << " ";
    }
    std::cout << "]" << std::endl;
}

int main() {
    std::set<int> s;
    s.insert(10);
    s.insert(50);
    s.insert(20);
    s.insert(40);
    s.insert(30);

    std::cout << "순서대로 정렬되어 나온다" << std::endl;
    print_set(s);

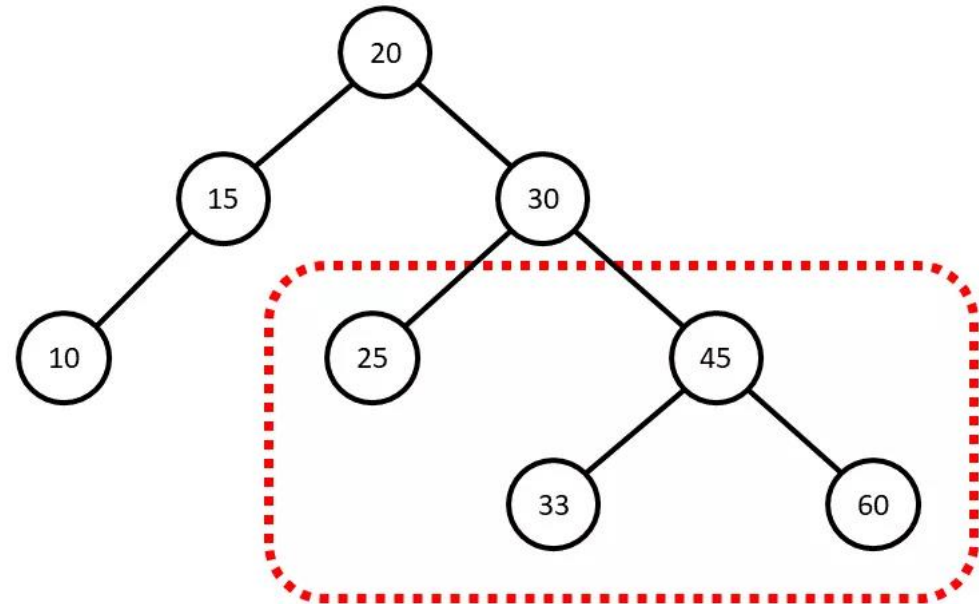
    std::cout << "20 이 s 의 원소인가? :: ";
    auto itr = s.find(20);
    if (itr != s.end()) {
        std::cout << "Yes" << std::endl;
    } else {
        std::cout << "No" << std::endl;
    }

    std::cout << "25 가 s 의 원소인가? :: ";
    itr = s.find(25);
    if (itr != s.end()) {
        std::cout << "Yes" << std::endl;
    } else {
        std::cout << "No" << std::endl;
    }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- set에 원소를 넣었을 때 10 -> 50 -> 20 -> 40 -> 30으로 넣었지만 실제로 반복자로 원소들을 모두 출력했을 때 나온 순서는 10 -> 20 -> 30 -> 40 -> 50 순으로 나왔다는 점에 유의 한다. set의 경우 내부에 원소를 추가할 때 정렬된 상태를 유지하며 추가한다.
- 20 을 찾았을 때 Yes 가 나오고 세트에 없는 원소인 25 를 찾는다면 No 가 출력된다.
 - 세트가 이러한 방식으로 작업을 수행할 수 있는 이유는 바로 내부적으로 트리 구조로 구성되어 있기 때문이다.
- 그림은 흔히 볼 수 있는 트리 구조를 나타낸다. 각각의 원소들은 트리의 각 노드들에 저장되어 있고, 다음과 같은 규칙을 지키고 있다.
 - 왼쪽에 오는 모든 노드들은 나보다 작다
 - 오른쪽에 있는 모든 노드들은 나보다 크다



C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- 세트의 중요한 특징으로 바로 세트 안에는 중복된 원소들이 없다는 점이다.

```
#include <iostream>
#include <set>

template <typename T>
void print_set(std::set<T>& s) {
    // 세트의 모든 원소들을 출력하기
    std::cout << "[ ";
    for (const auto& elem : s) {
        std::cout << elem << " ";
    }
    std::cout << " ] " << std::endl;
}

int main() {
    std::set<int> s;
    s.insert(10);
    s.insert(20);
    s.insert(30);
    s.insert(20);
    s.insert(10);

    print_set(s);
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- 클래스 객체를 세트에 넣고 싶을 때
- 컴파일 하였다면 아래와 같은 오류가 발생합니다.

binary '<': no operator found which takes a left-hand operand of type 'const Todo' (or there is no acceptable conversion)

```
#include <iostream>
#include <set>
#include <string>

template <typename T>
void print_set(std::set<T>& s) {
    // 셋의 모든 원소들을 출력하기
    std::cout << "[ ";
    for (const auto& elem : s) {
        std::cout << elem << " " << std::endl;
    }
    std::cout << "]" << std::endl;
}

class Todo {
    int priority; // 중요도. 높을 수록 급한것!
    std::string job_desc;

public:
    Todo(int priority, std::string job_desc)
        : priority(priority), job_desc(job_desc) {}
};

int main() {
    std::set<Todo> todos;

    todos.insert(Todo(1, "농구 하기"));
    todos.insert(Todo(2, "수학 숙제 하기"));
    todos.insert(Todo(1, "프로그래밍 프로젝트"));
    todos.insert(Todo(3, "친구 만나기"));
    todos.insert(Todo(2, "영화 보기"));
}
```


C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- 앞서 세트는 원소들을 저장할 때 내부적으로 정렬된 상태를 유지한다고 했다. 즉 정렬을 하기 위해서는 반드시 원소 간의 비교를 수행해야 한다. 하지만, 우리의 Todo 클래스에는 operator< 가 정의되어 있지 않다. 따라서 컴파일러는 < 연산자를 찾을 수 없기에 위와 같은 오류를 낸다.
- 해결점은 직접 Todo 클래스에 operator< 를 만들어주는 수 밖에 없다.

```
#include <iostream>
#include <set>
#include <string>

template <typename T>
void print_set(std::set<T>& s) {
    // 셋의 모든 원소들을 출력하기
    for (const auto& elem : s) {
        std::cout << elem << " " << std::endl;
    }
}

class Todo {
    int priority;
    std::string job_desc;

public:
    Todo(int priority, std::string job_desc)
        : priority(priority), job_desc(job_desc) {}

    bool operator<(const Todo& t) const {
        if (priority == t.priority) {
            return job_desc < t.job_desc;
        }
        return priority > t.priority;
    }

    friend std::ostream& operator<<(std::ostream& o, const Todo& td);
};

std::ostream& operator<<(std::ostream& o, const Todo& td) {
    o << "T 중요도: " << td.priority << " " << td.job_desc;
    return o;
}

int main() {
    std::set<Todo> todos;

    todos.insert(Todo(1, "농구 하기"));
    todos.insert(Todo(2, "수학 숙제 하기"));
    todos.insert(Todo(1, "프로젝트의 프로젝트"));
    todos.insert(Todo(3, "친구 만나기"));
    todos.insert(Todo(2, "영화 보기"));

    print_set(todos);

    std::cout << "-----" << std::endl;
    std::cout << "숙제를 끝냈다면!" << std::endl;
    todos.erase(todos.find(Todo(2, "수학 숙제 하기")));
    print_set(todos);
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- 세트에서 < 를 사용하기 위해서는 반드시 예제와 같은 형태로 함수를 작성해야 한다. 즉 const Todo 를 레퍼런스로 받는 const 함수로 말이다.
- 이를 지켜야 하는 이유는 세트 내부적으로 정렬 시에 상수 반복자를 사용하기 때문이다. (상수 반복자는 상수 함수만을 호출할 수 있다 .

```
bool operator<(const Todo& t) const {  
    if (priority == t.priority) {  
        return job_desc < t.job_desc;  
    }  
    return priority > t.priority;  
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- Todo < 연산자는 중요도(priority)가 다르면
 - return priority > t.priority;
 - 로 해서 중요도 값이 높은 것이 위로 가게 하였다.
- 만약 중요도가 같다면
 - return job_desc < t.job_desc;
 - 로 비교해서 job_desc 가 사전상에서 먼저 오는 것이 먼저 나오게 된다.

```
bool operator<(const Todo& t) const {  
    if (priority == t.priority) {  
        return job_desc < t.job_desc;  
    }  
    return priority > t.priority;  
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- 클래스 자체에 operator< 를 두지 않더라도 세트를 사용하는 방법.

```
#include <iostream>
#include <set>
#include <string>

template <typename T, typename C>
void print_set(std::set<T, C> s) {
    // 모든 요소를 출력하기
    for (const auto& elem : s) {
        std::cout << elem << " " << std::endl;
    }
}

class Todo {
    int priority;
    std::string job_desc;

public:
    Todo(int priority, std::string job_desc)
        : priority(priority), job_desc(job_desc) {}

    friend struct TodoCmp;
};

friend std::ostream& operator<<(std::ostream& o, const Todo& td);

struct TodoCmp {
    bool operator()(const Todo& t1, const Todo& t2) const {
        if (t1.priority == t2.priority) {
            return t1.job_desc < t2.job_desc;
        }
        return t1.priority > t2.priority;
    }
};

std::ostream& operator<<(std::ostream& o, const Todo& td) {
    o << "종료코드: " << td.priority << " " << td.job_desc;
    return o;
}

int main() {
    std::set<Todo, TodoCmp> todos;

    todos.insert(Todo(1, "물류 하기"));
    todos.insert(Todo(2, "수박 숙제 하기"));
    todos.insert(Todo(1, "프로그래밍 프로젝트"));
    todos.insert(Todo(3, "친구 만나기"));
    todos.insert(Todo(2, "영양 보기"));

    print_set(todos);

    std::cout << "-----" << std::endl;
    std::cout << "숙제할 날짜" << std::endl;
    todos.erase(todos.find(Todo(2, "수박 숙제 하기")));
    print_set(todos);
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- Todo 클래스에서 operator< 가 삭제되었다. 하지만 세트를 사용하기 위해 반드시 Todo 객체간의 비교를 수행해야 하기 때문에 다음과 같은 클래스를 만들었다.

```
struct TodoCmp {  
    bool operator()(const Todo& t1, const Todo& t2) const {  
        if (t1.priority == t2.priority) {  
            return t1.job_desc < t2.job_desc;  
        }  
        return t1.priority > t2.priority;  
    }  
};
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Set(세트)

- 예제는 정확히 함수 객체를 나타내고 있다. 이 TodoCmp 타입을 `std::set<Todo, TodoCmp> todos;` 정의 하고 set에 두번째 인자로 넘겨주게 되면 세트는 이를 받아서 TodoCmp 클래스에 정의된 함수 객체를 바탕으로 모든 비교를 수행하게 된다.

```
struct TodoCmp {  
    bool operator()(const Todo& t1, const Todo& t2) const {  
        if (t1.priority == t2.priority) {  
            return t1.job_desc < t2.job_desc;  
        }  
        return t1.priority > t2.priority;  
    }  
};
```

C++ STL(STANDARD TEMPLATE LIBRARY)

map

- 맵은 셋과 거의 똑같은 자료 구조이다.
- 다만 세트의 경우 키만 보관했지만, 맵의 경우 키에 대응되는 값(value) 까지도 같이 보관하게 된다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(std::map<K, V>& m) {
    // 맵의 모든 원소들을 출력하기
    for (auto itr = m.begin(); itr != m.end(); ++itr) {
        std::cout << itr->first << " " << itr->second << std::endl;
    }
}

int main() {
    std::map<std::string, double> pitcher_list;

    // 참고로 2017년 7월 4일 현재 투수 방어율 순위입니다.

    // 맵의 insert 함수는 pair 객체를 인자로 받습니다.
    pitcher_list.insert(std::pair<std::string, double>("박세웅", 2.23));
    pitcher_list.insert(std::pair<std::string, double>("해커", 2.93));

    pitcher_list.insert(std::pair<std::string, double>("피어밴드", 2.95));

    // 타입을 지정하지 않아도 간단히 std::make_pair 함수로
    // std::pair 객체를 만들 수 있습니다.
    pitcher_list.insert(std::make_pair("차우찬", 3.04));
    pitcher_list.insert(std::make_pair("장원준", 3.05));
    pitcher_list.insert(std::make_pair("헉터", 3.09));

    // 혹은 insert 를 안쓰더라도 [] 로 바로
    // 원소를 추가할 수 있습니다.
    pitcher_list["니퍼트"] = 3.56;
    pitcher_list["박종훈"] = 3.76;
    pitcher_list["필리"] = 3.90;

    print_map(pitcher_list);

    std::cout << "박세웅 방어율은? :: " << pitcher_list["박세웅"] << std::endl;
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

map

- 맵의 경우 템플릿 인자로 2 개를 가지는데, 첫번째는 키의 타입이고, 두번째는 값의 타입입니다.
- 맵에 원소를 넣기 위해서는 반드시 `std::pair` 객체를 전달해야 합니다.

```
template <class T1, class T2>
struct std::pair {
    T1 first;
    T2 second;
};
```


C++ STL(STANDARD TEMPLATE LIBRARY)

map

- 맵의 경우 operator[] 를 이용해서 해당하는 키가 맵에 없다면 새로운 원소를 추가할 수 도 있다.
- 만일 키가 이미 존재하고 있다면 값이 대체될 것이다.

```
// 혹은 insert 들 안쓰더라도 [] 로 바로  
// 원소들 추가할 수 있습니다.  
pitcher_list["니퍼트"] = 3.56;  
pitcher_list["박종훈"] = 3.76;  
pitcher_list["켈리"] = 3.90;
```

C++ STL(STANDARD TEMPLATE LIBRARY)

map

- 맵의 경우도 세트와 마찬가지로 반복자를 이용해서 순차적으로 맵에 저장되어 있는 원소들을 탐색할 수 있다. 세트의 경우 *itr 가 저장된 원소를 바로 가리켰는데, 맵의 경우 반복자가 맵에 저장되어 있는 std::pair 객체를 가리키게 된다.
- 따라서 itr->first 를 하면 해당 원소의 키를, itr->second 를 하면 해당 원소의 값을 알 수 있다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(const std::map<K, V>& m) {
    // kv 에는 맵의 key 와 value 가 std::pair 로 들어갑니다.
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
    }
}

int main() {
    std::map<std::string, double> pitcher_list;

    pitcher_list["오승환"] = 3.58;
    std::cout << "류현진 방어율은? :: " << pitcher_list["류현진"] << std::endl;

    std::cout << "-----" << std::endl;
    print_map(pitcher_list);
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

map

- double 의 디폴트 생성자의 경우 그냥 변수를 0 으로 초기화 해버린다.
- 따라서 되도록이면 find 함수로 원소의 키가 존재하는지 먼저 확인 후에, 값을 참조하는 것이 좋다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(const std::map<K, V>& m) {
    // kv 에는 맵의 key 와 value 가 std::pair 로 들어갑니다.
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
    }
}

template <typename K, typename V>
void search_and_print(std::map<K, V>& m, K key) {
    auto itr = m.find(key);
    if (itr != m.end()) {
        std::cout << key << " --> " << itr->second << std::endl;
    } else {
        std::cout << key << "은(는) 목록에 없습니다" << std::endl;
    }
}

int main() {
    std::map<std::string, double> pitcher_list;

    pitcher_list["오승환"] = 3.58;

    print_map(pitcher_list);
    std::cout << "-----" << std::endl;

    search_and_print(pitcher_list, std::string("오승환"));
    search_and_print(pitcher_list, std::string("류현진"));
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

map

- find 함수는 맵에서 해당하는 키를 찾아서 이를 가리키는 반복자를 반환한다. 만약에, 키가 존재하지 않는다면 end()를 반환한다.
- 주의할 점은 맵 역시 세트 처럼 중복된 원소를 허용하지 않는다는 점이다. 이미, 같은 키가 원소로 들어 있다면 나중에 오는 insert 는 무시된다.
- 만약에, 원소에 대응되는 값을 바꾸고 싶다면 insert 를 하지 말고, [] 연산자로 대응되는 값을 바꿔주면 된다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(const std::map<K, V>& m) {
    // kv 에는 맵의 key 와 value 가 std::pair 로 들어갑니다.
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
    }
}

int main() {
    std::map<std::string, double> pitcher_list;

    // 맵의 insert 함수는 std::pair 객체를 인자로 받습니다.
    pitcher_list.insert(std::pair<std::string, double>("박세웅", 2.23));
    pitcher_list.insert(std::pair<std::string, double>("박세웅", 2.93));

    print_map(pitcher_list);

    // 2.23 이 나올까 2.93 이 나올까?
    std::cout << "박세웅 방어율은? :: " << pitcher_list["박세웅"] << std::endl;
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

멀티세트(multiset)과 멀티맵(multimap)

- 세트와 맵 모두 중복된 원소를 허락하지 않는다. 만일, 이미 원소가 존재하고 있는데 insert 를 하였으면 무시가 되었다.
- 하지만 멀티셋과 멀티맵은 중복된 원소를 허락한다.

```
#include <iostream>
#include <set>
#include <string>

template <typename K>
void print_set(const std::multiset<K>& s) {
    // 셋의 모든 원소들을 출력하기
    for (const auto& elem : s) {
        std::cout << elem << std::endl;
    }
}

int main() {
    std::multiset<std::string> s;

    s.insert("a");
    s.insert("b");
    s.insert("a");
    s.insert("c");
    s.insert("d");
    s.insert("c");

    print_set(s);
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

멀티세트(multiset)과 멀티맵(multimap)

- 맵과는 다르게, 한 개의 키에 여러 개의 값이 대응될 수 있다는 것은 알 수 있다.
- 하지만 이 때문에 [] 연산자를 멀티맵의 경우 사용할 수 없다.
- m[1] 같이 접근한다면 어떤 값에 접근 할지 모르기 때문에 멀티맵의 경우 아예 [] 연산자를 제공하지 않는다.

C++ STL(STANDARD TEMPLATE LIBRARY)

멀티세트(multiset)과 멀티맵(multimap)

- find 함수를 사용했을 때 무엇을 리턴할까? 일단 해당하는 키가 없으면 m.end() 를 리턴한다. 그렇다면 위 경우 1 이라는 키에 3 개의 문자열이 대응되어 있는데 어떤 것을 반환해야 할까? 제일 먼저 insert 한 것? 아니면 문자열 중에서 사전 순으로 가장 먼저 오는 것?
- 사실 C++ 표준을 읽어보면 무엇을 반환하라고 정해 놓지 않았다. 즉, 해당되는 값들 중 아무거나 반환해도 상관 없다는 뜻이다. 위 경우 hello 가 나왔지만, 다른 라이브러리를 쓰는 경우 hi 가 나올 수도 있고, ahihi 가 나올 수도 있다.
- 그렇다면 1 에 대응되는 값들이 뭐가 있는지 어떻게 알까? 이를 위해 멀티맵은 다음과 같은 함수 equal_range를 제공하고 있다.

```
#include <iostream>
#include <map>
#include <string>

template <typename K, typename V>
void print_map(const std::multimap<K, V>& m) {
    // 맵의 모든 원소들을 출력하기
    for (const auto& kv : m) {
        std::cout << kv.first << " " << kv.second << std::endl;
    }
}

int main() {
    std::multimap<int, std::string> m;
    m.insert(std::make_pair(1, "hello"));
    m.insert(std::make_pair(1, "hi"));
    m.insert(std::make_pair(1, "ahihi"));
    m.insert(std::make_pair(2, "bye"));
    m.insert(std::make_pair(2, "baba"));

    print_map(m);

    std::cout << "-----" << std::endl;

    // 1 을 키로 가지는 반복자들의 시작과 끝을
    // std::pair 로 만들어서 리턴한다.
    auto range = m.equal_range(1);
    for (auto itr = range.first; itr != range.second; ++itr) {
        std::cout << itr->first << " : " << itr->second << " " << std::endl;
    }
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

멀티세트(multiset)과 멀티맵(multimap)

- equal_range 함수의 경우 인자로 멀티맵의 키를 받은 뒤에, 이 키에 대응되는 원소들의 반복자들 중에서 시작과 끝 바로 다음을 가리키는 반복자를 std::pair 객체로 만들어서 반환한다.
- 즉, begin() 과 end() 를 std::pair 로 만들어서 세트로 리턴 한다. 다만, first 로 시작점을, second 로 끝점 바로 뒤를 알 수 있다. 왜 끝점 바로 뒤를 가리키는 반복자를 리턴 하는지는 굳이 설명 안 해도 알 수 있다.
- 예제처럼 1 에 대응되는 모든 원소들을 볼 수 있게 된다.

```
for (auto itr = range.first; itr != range.second; ++itr) {  
    std::cout << itr->first << " : " << itr->second << " " << std::endl;  
}
```


C++ STL(STANDARD TEMPLATE LIBRARY)

algorithm

- C++ 표준 라이브러리는 앞에서 이야기 했던 대로, 컨테이너에 반복자들을 가지고 이런 저런 작업을 쉽게 수행할 수 있도록 도와주는 라이브러리 이다.
- 여기서 말하는 이런 저런 작업이란, 정렬이나 검색과 같이 단순한 작업들 말고도, ' 이런 조건이 만족하면 컨테이너에서 지워줘 ' 나 ' 이런 조건이 만족하면 1 을 더해 ' 와 같은 복잡한 명령의 작업들도 알고리즘 라이브러리를 통해 수행할 수 있다.
- 알고리즘에 정의되어 있는 여러가지 함수들로 작업을 수행하게 된다.

C++ STL(STANDARD TEMPLATE LIBRARY)

정렬 (sort, stable_sort, partial_sort)

- Algorithm library에서 지원하는 정렬(sort)은 한 가지 밖에 없을 것 같은데 정렬 알고리즘에서는 무려 3 가지 종류의 함수를 지원하고 있다.
- sort : 일반적인 정렬 함수.
- stable_sort : 정렬을 하되 원소들 간의 순서를 보존한다. 만약에 벡터에 [a, b] 순으로 있었는데, a 와 b 가 크기가 같다면 정렬을 [a,b] 혹은 [b,a] 로 할 수 있다. sort 의 경우 그 순서가 랜덤으로 정해진다. 하지만 stable_sort 의 경우 그 순서를 반드시 보존한다. 즉 컨테이너 상에서 [a,b] 순으로 있었다면 정렬 시에도 (크기가 같다면) [a,b] 순으로 나오게 된다. 이 때문에 sort 보다 좀 더 느리다.
- partial_sort : 배열의 일부분만 정렬한다.

C++ STL(STANDARD TEMPLATE LIBRARY)

sort

- sort 함수는 예제와 같이 정렬할 원소의 시작 위치와, 마지막 위치 바로 뒤를 반복자로 받는다.
- 참고로 sort 에 들어가는 반복자의 경우 반드시 임의접근 반복자(RandomAccessIterator) 타입을 만족해야 하므로, 우리가 보왔던 컨테이너들 중에서는 벡터와 deque만 가능하고 나머지 컨테이너는 sort 함수를 적용할 수 없다. (예를 들어 리스트의 경우 반복자 타입이 양방향 반복자(BidirectionalIterator) 이므로 안된다)

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

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << *begin << " ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(6);
    vec.push_back(4);
    vec.push_back(7);
    vec.push_back(2);

    std::cout << "정렬 전 ----" << std::endl;
    print(vec.begin(), vec.end());
    std::sort(vec.begin(), vec.end());

    std::cout << "정렬 후 ----" << std::endl;
    print(vec.begin(), vec.end());
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

sort

- sort 함수는 기본적으로 오름차순으로 정렬을 해준다. 그렇다면 만약에 내림차순으로 정렬하고 싶다면 어떻게 할까? 만약에 여러분이 직접 만든 타입 이었다면 단순히 operator< 를 반대로 바꿔준다면 오름차순에서 내림차순이 되었겠지만, 이 경우 int 이기 때문에 이는 불가능 하다.
- 하지만 앞서 대부분의 알고리즘은 3 번째 인자로 특정한 조커를 전달한다고 하였는데, 여기에 우리가 비교를 어떻게 수행할 것인지에 대해 알려주면 된다.
- 예제와 같이 함수 객체를 위한 구조체를 정의해주고, 그 안에 operator() 함수를 만들어주면 함수 객체 준비는 완료된다.

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

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << *begin << " ";
        begin++;
    }
    std::cout << std::endl;
}

struct int_compare {
    bool operator()(const int& a, const int& b) const { return a > b; }
};

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(6);
    vec.push_back(4);
    vec.push_back(7);
    vec.push_back(2);

    std::cout << "정렬 전 ----" << std::endl;
    print(vec.begin(), vec.end());
    std::sort(vec.begin(), vec.end(), int_compare());

    std::cout << "정렬 후 ----" << std::endl;
    print(vec.begin(), vec.end());
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

partial sort

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

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << *begin << " ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(6);
    vec.push_back(4);
    vec.push_back(7);
    vec.push_back(2);

    std::cout << "정렬 전 ----" << std::endl;
    print(vec.begin(), vec.end());
    std::partial_sort(vec.begin(), vec.begin() + 3, vec.end());

    std::cout << "정렬 후 ----" << std::endl;
    print(vec.begin(), vec.end());
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

partial sort

- `partial_sort` 함수는 일부만 정렬하는 함수라고 하였다.
- `partial_sort` 는 인자를 다음과 같이 3 개를 기본으로 받는다.
- 이 때 정렬을 `[start, end)` 전체 원소들 중에서 `[start, middle)` 까지 원소들이 전체 원소들 중에서 제일 작은 원소 순으로 정렬 시킨다.

```
std::partial_sort(start, middle, end)
```

C++ STL(STANDARD TEMPLATE LIBRARY)

partial sort

- 예제와 같이 `vec.begin()` 부터 `vec.end()` 까지 (즉 벡터 전체에서) 원소들 중에서, `vec.begin()` 부터 `vec.begin() + 3` 까지에 전체에서 가장 작은 원소들만 순서대로 저장하고 나머지 위치는 상관 없는 것이다.

```
std::partial_sort(vec.begin(), vec.begin() + 3, vec.end());
```

5 3 1 6 4 7 2

에서 가장 작은 3개 원소인 1, 2, 3 만이 정렬되어서

1 2 3 6 5 7 4

C++ STL(STANDARD TEMPLATE LIBRARY)

partial sort

- 만약에 우리가 전체 배열을 정렬할 필요가 없을 경우, 예를 들어서 100 명의 학생 중에서 상위 10 명의 학생의 성적순을 보고 싶을 경우
- 이런 식이면 굳이 sort 로 전체를 정렬 할 필요 없이 partial_sort 로 10 개만 정렬 하는 것이 더 빠르게 된다.

C++ STL(STANDARD TEMPLATE LIBRARY)

stable sort

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

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << " ";
        begin++;
    }
    std::cout << std::endl;
}

struct User {
    std::string name;
    int age;

    User(std::string name, int age) : name(name), age(age) {}
};

bool operator<(const User& u) const { return age < u.age; }

std::ostream& operator<<(std::ostream& o, const User& u) {
    o << u.name << " ", << u.age;
    return o;
}

int main() {
    std::vector<User> vec;
    for (int i = 0; i < 100; i++) {
        std::string name = "";
        name.push_back('a' + i / 26);
        name.push_back('a' + i % 26);
        vec.push_back(User(name, static_cast<int>(rand() % 10)));
    }

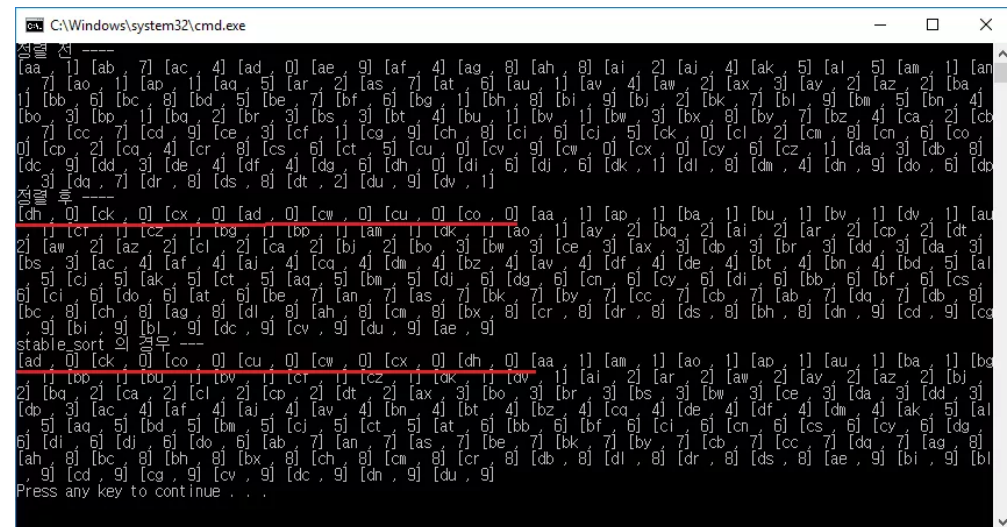
    std::vector<User> vec2 = vec;

    std::cout << "정렬 전 ----" << std::endl;
    print(vec.begin(), vec.end());

    std::sort(vec.begin(), vec.end());

    std::cout << "정렬 후 ----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "stable_sort 의 경우 ----" << std::endl;
    std::stable_sort(vec2.begin(), vec2.end());
    print(vec2.begin(), vec2.end());
}
```



```
C:\Windows\system32\cmd.exe
정렬 전 ----
[aa, 1] [ab, 7] [ac, 4] [ad, 0] [ae, 9] [af, 4] [ag, 8] [ah, 8] [ai, 2] [aj, 4] [ak, 5] [al, 5] [am, 1] [an, 1] [ao, 1] [ap, 1] [aq, 5] [ar, 2] [as, 7] [at, 6] [au, 1] [av, 4] [aw, 2] [ax, 3] [ay, 2] [az, 2] [ba, 1] [bb, 6] [bc, 8] [bd, 5] [be, 7] [bf, 6] [bg, 1] [bh, 8] [bi, 9] [bj, 2] [bk, 7] [bl, 9] [bm, 5] [bn, 4] [bo, 3] [bp, 1] [bq, 2] [br, 3] [bs, 3] [bt, 4] [bu, 1] [bv, 1] [bw, 3] [bx, 8] [by, 7] [bz, 4] [ca, 2] [cb, 7] [cc, 7] [cd, 9] [ce, 3] [cf, 1] [cg, 9] [ch, 8] [ci, 6] [cj, 5] [ck, 0] [cl, 2] [cm, 8] [cn, 6] [co, 0] [cp, 2] [cq, 4] [cr, 8] [cs, 6] [ct, 5] [cu, 0] [cv, 9] [cw, 0] [cx, 0] [cy, 6] [cz, 1] [da, 3] [db, 8] [dc, 9] [dd, 3] [de, 4] [df, 4] [dg, 6] [dh, 0] [di, 6] [dj, 6] [dk, 1] [dl, 8] [dm, 4] [dn, 9] [do, 6] [dp, 3] [dq, 7] [dr, 8] [ds, 8] [dt, 2] [du, 9] [dv, 1]
정렬 후 ----
[ad, 0] [ck, 0] [cx, 0] [ad, 0] [cw, 0] [cu, 0] [co, 0] [aa, 1] [ap, 1] [ba, 1] [bu, 1] [bv, 1] [dv, 1] [au, 1] [cr, 1] [cz, 1] [bg, 1] [bp, 1] [am, 1] [dk, 1] [ao, 1] [ay, 2] [bq, 2] [ai, 2] [ar, 2] [cp, 2] [dt, 2] [aw, 2] [az, 2] [cl, 2] [ca, 2] [bj, 2] [bo, 3] [bw, 3] [ce, 3] [ax, 3] [do, 3] [br, 3] [bs, 3] [dd, 3] [da, 3] [ls, 3] [ac, 4] [af, 4] [aj, 4] [cq, 4] [dm, 4] [bz, 4] [av, 4] [df, 4] [de, 4] [bt, 4] [bn, 4] [bd, 5] [cs, 5] [c], 5] [ak, 5] [ct, 5] [aq, 5] [bm, 5] [di, 6] [dg, 6] [cn, 6] [cy, 6] [di, 6] [bb, 6] [bf, 6] [cs, 6] [ci, 6] [do, 6] [at, 6] [be, 7] [an, 7] [as, 7] [bk, 7] [by, 7] [cc, 7] [cb, 7] [ab, 7] [dq, 7] [db, 8] [bc, 8] [ch, 8] [ag, 8] [dl, 8] [ah, 8] [cm, 8] [bx, 8] [cr, 8] [dr, 8] [ds, 8] [bh, 8] [dn, 9] [cd, 9] [cg, 9] [bi, 9] [bl, 9] [dc, 9] [cv, 9] [du, 9] [ae, 9]
stable_sort 의 경우 ----
[ad, 0] [ck, 0] [co, 0] [cu, 0] [cw, 0] [cx, 0] [dh, 0] [aa, 1] [am, 1] [ao, 1] [ap, 1] [au, 1] [ba, 1] [bg, 1] [bp, 1] [bu, 1] [bv, 1] [cr, 1] [cz, 1] [dk, 1] [dv, 1] [ai, 2] [ar, 2] [aw, 2] [ay, 2] [az, 2] [bj, 2] [bq, 2] [ca, 2] [cl, 2] [cp, 2] [dt, 2] [ax, 3] [bo, 3] [br, 3] [bs, 3] [bw, 3] [ce, 3] [da, 3] [dd, 3] [db, 3] [ac, 4] [af, 4] [aj, 4] [av, 4] [bn, 4] [bt, 4] [bz, 4] [cq, 4] [de, 4] [df, 4] [dm, 4] [ak, 5] [al, 5] [aq, 5] [bd, 5] [bm, 5] [ci, 5] [ct, 5] [at, 6] [bb, 6] [bf, 6] [ci, 6] [cn, 6] [cs, 6] [cy, 6] [dg, 6] [di, 6] [dj, 6] [do, 6] [ab, 7] [an, 7] [as, 7] [be, 7] [bk, 7] [by, 7] [cb, 7] [cc, 7] [da, 7] [ag, 8] [ah, 8] [bc, 8] [bh, 8] [bx, 8] [ch, 8] [cm, 8] [cr, 8] [db, 8] [dl, 8] [dr, 8] [ds, 8] [ae, 9] [bi, 9] [bl, 9] [cd, 9] [cg, 9] [cv, 9] [du, 9] [dc, 9] [dn, 9] [du, 9]
Press any key to continue . . .
```

C++ STL(STANDARD TEMPLATE LIBRARY)

stable sort

- `stable_sort` 는 원소가 삽입되어 있는 순서를 보존하는 정렬 방식이라고 하였다. `stable_sort` 가 확실히 어떻게 `sort` 와 다른 지 보여주기 위해서 다음과 같은 클래스를 만들어보자.
- 이 `User` 클래스는 `name` 과 `age` 를 멤버로 갖는데, 크기 비교는 이름과 관계 없이 모두 `age` 로 하게 된다. 즉 `age` 가 같다면 크기가 같다고 볼 수 있다.

```
struct User {  
    std::string name;  
    int age;  
  
    User(std::string name, int age) : name(name), age(age) {}  
  
    bool operator<(const User& u) const { return age < u.age; }  
};
```

C++ STL(STANDARD TEMPLATE LIBRARY)

stable sort

- 처음에 벡터에 원소들을 삽입하는 부분인데, 이름은 aa, ab, ac, ... 순으로 하되 age 의 경우 0 부터 10 사이의 랜덤 값을 부여하였다. 즉 name 의 경우 string 순서대로 되어있고, age 의 경우 랜덤 순서로 되어 있다.
- stable_sort 는 삽입되어 있던 원소들 간의 순서를 보존 한다고 했다. 따라서 같은 age 라면 반드시 삽입된 순서, 즉 name 순으로 나올 것이다. (왜냐하면 애초에 name 순으로 넣었기 때문이다.)

```
for (int i = 0; i < 100; i++) {  
    std::string name = "";  
    name.push_back('a' + i / 26);  
    name.push_back('a' + i % 26);  
    vec.push_back(User(name, static_cast<int>(rand() % 10)));  
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

stable sort

그 결과를 살펴보면 확연히 다름을 알 수 있습니다. 먼저 `sort` 의 경우

```
dh, ck, cx, ad, cw, cu, co
```

순으로 나와 있고 (age 가 0 일 때) `stable_sort` 의 경우 `age` 가 0 일 때

```
ad, ck, co, cu, cw, cx, dh
```

순으로 나오게 됩니다. 다시 말해 `sort` 함수의 경우 정렬 과정에서 원소들 간의 상대적 위치를 랜덤하게 바꿔버리지만 `stable_sort` 의 경우 그 순서를 처음에 넣었던 상태 그대로 유지함을 알 수 있습니다.

당연히도 이러한 제약 조건 때문에 `stable_sort` 는 그냥 `sort` 보다 좀 더 오래걸립니다. C++ 표준에 따르면 `sort` 함수는 최악의 경우에서도 $O(n \log n)$ 이 보장되지만 `stable_sort` 의 경우 최악의 경우에서 $O(n (\log n)^2)$ 으로 작동하게 됩니다. 조금 더 느린 편이지요.

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

- 람다 함수는 C++ 에서는 C++ 11 에서 처음으로 도입되었다. 람다 함수를 통해 쉽게 이름이 없는 함수 객체를 만들 수 있게 되었다.
- 익명의 함수 객체를 말한다.

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

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << " ] ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "처음 vec 상태 ----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "벡터에서 홀수인 원소 제거 ----" << std::endl;
    vec.erase(std::remove_if(vec.begin(), vec.end(),
                             [](int i) -> bool { return i % 2 == 1; } ),
              vec.end());

    print(vec.begin(), vec.end());
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

- 람다 함수의 정의.

```
[](int i) -> bool { return i % 2 == 1; }
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

- 일반적인 정의를 보면 보기와 같다.
- `capture_list` 가 뭔지는 아래에서 설명하도록 하고, 함수 정의를 보자면 인자로 `int i` 를 받고, `bool` 을 반환하는 람다 함수를 정의한 것이다. 반환 타입을 생략한다면 컴파일러가 알아서 함수 본체에서 `return` 문을 보고 리턴 타입을 추측해준다. (만약에 `return` 경로가 여러 군데여서 추측할 수 없다면 컴파일 오류가 발생한다)

[capture list] (받는 인자) -> 리턴 타입 { 함수 본체 }

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

- 리턴 타입을 생략할 경우.

[capture list] (받는 인자) {함수 본체}

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

- 하지만 람다 함수도 말 그대로 함수이기 때문에 자기 자신만의 영역(scope)를 가진다. 따라서 일반적인 상황이라면 함수 외부에서 정의된 변수들을 사용할 수 없다.
- 예를 들어서 최대 2 개 원소만 지우고 싶은 경우 예제와 같이 람다 함수 외부에 몇 개를 지웠는지 변수를 정의한 뒤에 사용해야만 하는데 (함수 안에 정의하면 함수 호출될 때 마다 새로 생성된다)
- 문제는 그 변수에 접근할 수 없다는 점이다. 하지만 놀랍게도 람다 함수의 경우 그 변수에 접근할 수 있다. 바로 캡처 목록(capture list)을 사용하는 것이다.

```
std::cout << "벡터에서 홀수인 원소 최대 2 개 제거 ---" << std::endl;
int num_erased = 0;
vec.erase(std::remove_if(vec.begin(), vec.end(),
    [](int i) {
        if (num_erased >= 2)
            return false;
        else if (i % 2 == 1) {
            num_erased++;
            return true;
        }
        return false;
    }),
    vec.end());
print(vec.begin(), vec.end());
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

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

template <typename Iter>
void print(Iter begin, Iter end) {
    while (begin != end) {
        std::cout << "[" << *begin << " ";
        begin++;
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> vec;
    vec.push_back(5);
    vec.push_back(3);
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);

    std::cout << "처음 vec 상태 ----" << std::endl;
    print(vec.begin(), vec.end());

    std::cout << "벡터에서 홀수인 원소 ----" << std::endl;
    int num_erased = 0;
    vec.erase(std::remove_if(vec.begin(), vec.end(),
        [&num_erased](int i) {
            if (num_erased >= 2)
                return false;
            else if (i % 2 == 1) {
                num_erased++;
                return true;
            }
            return false;
        }),
        vec.end());
    print(vec.begin(), vec.end());
}
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

- 예제와 같이 캡처 목록에는 어떤 변수를 캡처 할 지 써주면 된다. 위 경우 num_erased 를 캡처 하였다. 즉 람다 함수 내에서 num_erased 를 마치 같은 scope 안에 있는 것 처럼 사용할 수 있게 된다.
- num_erased 앞에 & 가 붙어있는데 이는 실제 num_erased 의 레퍼런스를 캡처 한다는 의미이다. 즉 함수 내부에서 num_erased 의 값을 바꿀 수 있게 될수 있게 된다.

```
[&num_erased](int i) {  
    if (num_erased >= 2)  
        return false;  
    else if (i % 2 == 1) {  
        num_erased++;  
        return true;  
    }  
    return false;  
})
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

- & 를 앞에 붙이지 않는다면 num_erased 의 복사본을 얻게 되는데, 그 복사본의 형태는 const 이다. 따라서 예제 처럼 함수 내부에서 num_erased 의 값을 바꿀 수 없게 된. 그렇다면 클래스의 멤버 함수 안에서 람다를 사용할 때 멤버 변수들을 참조하려면 어떻게 해야 할까?

```
[num_erased](int i){  
    if (num_erased >= 2)  
        return false;  
    else if (i % 2 == 1) {  
        num_erased++;  
        return true;  
    }  
    return false;  
})
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

- 예를 들어 위 같은 예제를 생각해보자. 쉽게 생각해보면 그냥 똑같이 num_erased 를 & 로 캡처 해서 람다 함수 안에서 사용할 수 있을 것 같지만 실제로는 컴파일 되지 않는다.
- 왜냐하면 num_erased 가 일반 변수가 아니라 객체에 종속되어 있는 멤버 변수이기 때문이다. 즉 람다 함수는 num_erased 를 캡처하려고 하면 이 num_erased 가 이 객체의 멤버 변수가 아니라 그냥 일반 변수라고 생각하게 된다.

```
class SomeClass {
    std::vector<int> vec;

    int num_erased;

public:
    SomeClass() {
        vec.push_back(5);
        vec.push_back(3);
        vec.push_back(1);
        vec.push_back(2);
        vec.push_back(3);
        vec.push_back(4);

        num_erased = 1;

        vec.erase(std::remove_if(vec.begin(), vec.end(),
                                  [&num_erased](int i) {
                                      if (num_erased >= 2)
                                          return false;
                                      else if (i % 2 == 1) {
                                          num_erased++;
                                          return true;
                                      }
                                      return false;
                                  }),
                  vec.end());
    }
};
```

C++ STL(STANDARD TEMPLATE LIBRARY)

Lambda function

- 이것을 해결하기 위해선 직접 멤버 변수를 전달하기 보다는 this 를 전달해주면 된다.
- 예제 같이 this 를 복사본으로 전달해서 (참고로 this 는 레퍼런스로 전달할 수 없다) 함수 안에서 this 를 이용해서 멤버 변수들을 참조해서 사용하면 된다.
- 예제에 설명한 경우 말고도 캡처 리스트의 사용 방법은 꽤나 많은데 아래 간단히 정리해보도록 하겠다.
 - [] : 아무것도 캡처 안함
 - [&a, b] : a 는 레퍼런스로 캡처 하고 b 는 (변경 불가능한) 복사본으로 캡처
 - [&] : 외부의 모든 변수들을 레퍼런스로 캡처
 - [=] : 외부의 모든 변수들을 복사본으로 캡처

```
num_erased = 0;

vec.erase(std::remove_if(vec.begin(), vec.end(),
    [this](int i) {
        if (this->num_erased >= 2)
            return false;
        else if (i % 2 == 1) {
            this->num_erased++;
            return true;
        }
        return false;
    })),
    vec.end());
```