

# 목 차

SECTION 1	STL Preview	1
	1. STL의 소개	2
	2. string & complex	5
	3. bitset & pair	10
SECTION 2	STL의 설계 원리와 특징	15
	1. 일반화 알고리즘의 개념	16
	2. Container 와 iterator	29
	3. STL 라이브러리의 특징	42
SECTION 3	반복자 ( iterator )	46
	1. 반복자(iterator) 기본	47
	2. iterator category	52
	3. iterator_traits	57
	4. insert iterator	60
	5. stream iterator	68
	6. iterator adapter	72
	7. iterator 보조함수	77
SECTION 4	알고리즘 ( algorithm )	80
	1. algorithm 기본	81
	2. algorithm과 함수	85
	3. algorithm의 변형	93
SECTION 5	컨테이너 ( Container )	103
	1. container 개념	104
	2. sequence container	111
	3. container adapter	118
	4. associative container	122
SECTION 6	utility	133
	1. smart pointer #1	134
	2. smart pointer #2	142

3. chrono	151
4. function & bind	161
<b>SECTION 7 동시성(Concurrency)</b>	<b>170</b>
1. thread	171

Section 1.

# STL Preview

항목 1-1

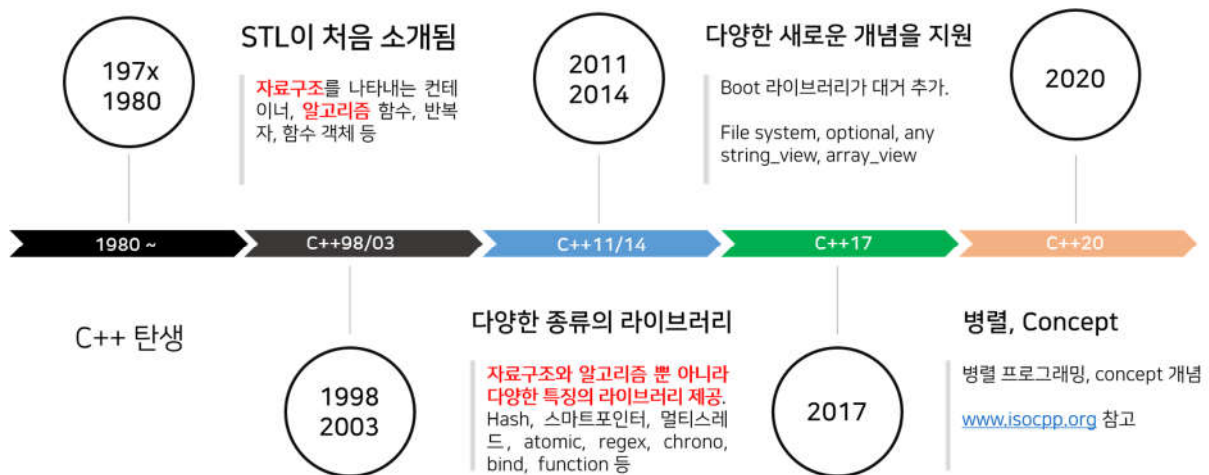
# STL 소개

## 학습 목표

- STL 의 탄생과 역사
- STL 의 주요 요소

# 1. STL 의 역사

## I STL의 탄생과 역사



C++ 의 표준 라이브러리인 STL(Standard Template Library)는 Alexander Stepanov에 의해 처음 소개된 후 1994년 C++ 표준 위원회 회의에서 처음 공식 제안이 이루어 집니다. 그 후, 1998년 C++의 1차 표준안(C++98)에서 공식 표준으로 채택이 되고 C++11, C++14, C++17 을 거치 면서 다양한 요소가 추가 되었습니다.

## I STL 의 주요 요소

1998년 표준화된 최초의 STL 은 자료구조와 알고리즘 위주의 라이브러리만을 제공했습니다. 하지만, 2011년 표준화된 C++11 에서는 자료구조 이외에도 스마트포인터, 스레드 등의 다양한 라이브러리를 추가했습니다. 2017년 표준화된 C++17 에서는 보다 종류의 라이브러리를 제공하고 있습니다. 아래 표는 C++ 버전별 라이브러리의 종류 입니다.

C++ version	Description
C++98	자료구조와 알고리즘 위주의 라이브러리 다양한 알고리즘 함수와 자료구조를 구현한 컨테이너 list, vector, deque, set, map, stack, queue 등
C++11/14	자료구조에 대한 추가(forward_list, tuple, array, hash 기반 컨테이너) 자료구조 이외의 도구들 추가( 스마트 포인터, thread, chrono, function, bind 등)
C++17	any, optional, asio, file system, string_view, array_view, range 병렬 프로그래밍 기반 알고리즘

## I STL의 특징

최초의 STL 버전은 알고리즘, 컨테이너, 함수자 그리고 반복자 라고 불리는 4가지의 구성 요소를 제공합니다. 각 구성요소에 대해서는 다음 항목에서부터 차례대로 배우게 됩니다.

STL은 컨테이너와 연관 배열 같은 C++을 위한 일반 클래스들의 미리 만들어진 집합을 제공하는데, 이것들은 표준 타입 뿐 아니라 사용자 정의 타입 과도 같이 사용될 수 있습니다. STL 알고리즘들은 컨테이너들에 독립적인데, 이것은 라이브러리의 복잡성을 눈에 띄게 줄여주었습니다.

STL은 대부분 템플릿으로 구현되어 있는데, 이 접근법은 전통적인 런타임 다형성에 비해 훨씬 효과적인 컴파일 타임 다형성을 제공합니다. 현대의 C++ 컴파일러들은 STL의 많은 사용에 의해 야기되는 어떤 추상화 단점을 최소화하도록 최적화 되어 있습니다.

항목 1-2

# string & complex

## 학습 목표

- string 사용법
- complex 사용법

# 1. string

## I 핵심 개념

- 문자열 관리를 위한 클래스.
- `<string>` 헤더 필요
- 문자열을 사용할 때 정수형 변수 처럼 사용할 수 있기 때문에 C 언어의 문자열 함수를 사용하는 것보다 직관적으로 문자열을 사용할 수 있습니다.

```
#include <iostream>
#include <string.h> // C헤더
#include <string>   // STL string
using namespace std;

int main()
{
    // char s1[10] = "hello";
    // char s2[10];

    string s1 = "hello";
    string s2;

    s2 = s1;          // strcpy(s2, s1)

    if (s1 == s2)    // strcmp(s1, s2)
    {
    }
    string s3 = s1 + s2;

    cout << s3 << endl;
}
```



## I string 과 변환

string 타입을 C문자열로 변경하려면 `c_str()` 멤버 함수를 사용합니다. `c_str()` 함수를 사용하면 string 타입을 `const char*` 로 형 변환 할 수 있습니다.

또한, `stod()`, `to_string()` 등의 일반 함수를 사용하면 `double` <-> `string` 간의 변환을 할 수 있습니다.

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

int main()
{
    string s1 = "hello";
    char s2[10];

    //strcpy(s2, s1); // error. string => const char* 변환이 필요하다.

    strcpy(s2, s1.c_str()); // ok. s1.data()을 사용해도 된다.

    string s3 = "3.4";

    double d = stod(s3);

    cout << d << endl; // 3.4

    string s4 = to_string(5.4);
    cout << s4 << endl;
}
```

## I string 과 user define literal

이중 따옴표(“)를 사용하는 문자열은 `const char []` 타입이지만 문자열 상수 접미사 `s(“”s)` 를 사용하면 `string` 타입을 나타냅니다. C++11 부터 추가된 user define literal 문법입니다.

```
#include <iostream>
#include <string>
#include <complex>
//using namespace std;
using std::cout;
using std::endl;
using std::string;

//using namespace std::string_literals;
//using namespace std::literals;

void foo(string s)      { cout << "string" << endl;}
void foo(const char* s) { cout << "char*" << endl; }

int main()
{
    // using namespace std::literals;

    foo("hello"); // char*
    foo("hello"s); // string
}
```

## 2. complex

### I 핵심 개념

- `std::complex`는 복소수를 나타내는 템플릿입니다.
- `std::complex`는 일반적으로 `double/float/long double` 타입으로 널리 사용되면, 이외의 타입에 대해서는 unspecified 입니다.
- user define literal “i” 또는 “if”, “il” 를 사용하면 허수부를 쉽게 만들 수 있습니다.

```
#include <iostream>
#include <complex>
using namespace std;
using namespace std::literals;

int main()
{
    complex<double> c1(1, 0);
    cout << c1 << endl;
    cout << c1.real() << endl;
    cout << c1.imag() << endl;

    complex<double> c2 = sin(c1);
    cout << c2 << endl;

    // complex<int> c3(1, 0);
    // complex<int> c4 = sin(c3);

    complex<float> c5(3);           // 3, 0
    complex<float> c6(3, 1);        // 3, 1
    complex<float> c7(3if);         // 0, 3
    cout << c5 << endl;
    cout << c7 << endl;
}
```

항목 1-3

# bitset & pair

## 학습 목표

- bitset 사용법
- pair 사용법

# 1. bitset

## I 핵심 개념

- bit 를 관리하는 클래스
- <bitset> 헤더
- 변환 함수 : to\_string(), to\_ulong(), to\_ullong()
- 비트 접근 함수 : set, reset, [], flip
- 비트 조사 함수 : test, all, none, any, count
- 비트 연산 가능 : &, |, ^

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

int main()
{
    //bitset<8> b1; // 0
    //bitset<8> b1 = 0xf0; // 11110000
    bitset<8> b1 = 0b11110000;

    b1.set();    // 모두 1로
    b1.reset();  // 모두 0으로
    b1.set(1);   // 0000 0010
    b1[2] = 1;   // 0000 0110
    b1[0].flip(); // 0000 0111

    if (b1.test(1) == true) {}
    if (b1[1] == true) {}

    if (b1.none() == true) {}

    int n2 = b1.count();
    cout << n2 << endl; // 3

    bitset<8> b2 = 0b00001111;
    bitset<8> b3 = 0b11110000;
    bitset<8> b4 = b2 | b3;
```

```
cout << b4 << endl;
cout << b1 << endl;
string s      = b1.to_string();
unsigned long n = b1.to_ulong();
cout << s << endl;
cout << n << endl;
}
```

## 2. pair

### ■ 핵심 개념

서로 다른 타입의 데이터 2개(first, second)를 보관하는 template.

```
template<typename T, typename U> struct pair
{
    T first;
    U second;

    pair() {}
    pair(const T& a, const U& b) : first(a), second(b) {}
};
```

### ■ first, second

pair에 각 요소에 접근 할 때는 first, second 를 사용합니다.

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

int main()
{
    pair<int, double> p1(1, 3.4);
    cout << p1.first << endl; // 1
    cout << p1.second << endl; // 3.4
}
```

### ■ make\_pair

make\_pair 함수를 사용하면 pair를 편리하게 만들 수 있습니다.

```
template<typename T> void foo(T arg)
{
}

int main()
```

```

{
    // 1. pair를 만들어서 전달
    pair<int, double> p1(1, 3.4);
    foo(p1);

    // 2. pair의 임시객체를 전달
    foo(pair<int, double>(1, 3.4));

    // 3. make_pair를 사용해서 전달
    foo(make_pair(1, 3.4));

    // 4. C++17 부터는 클래스 템플릿의 암시적 타입 추론이 가능
    foo(pair(1, 3.4)); // C++17 부터 가능

    auto p = pair(1, 3.4); // c++17 부터 가능.
}

```

## I pair의 중첩

템플릿 인자로 템플릿을 사용할 수 있으므로 pair는 2개 뿐 아니라 여러 개를 보관 할 수 도 있습니다.

```

int main()
{
    pair<int, pair<int, int>> p3(1, make_pair(1, 3));
}

```

중첩된 pair는 사용법이 복잡해 보입니다. tuple을 사용하면 중첩된 pair를 편리하게 사용할 수 있습니다.



Section 2.

# STL 의 설계원리와 특징

항목 2-1

## 일반화 알고리즘의 개념

### 학습 목표

- find 알고리즘 만들기

## 2. STL 알고리즘의 설계 방식

### I 선형 검색 알고리즘 - STL find

STL에는 선형 검색 알고리즘을 구현한 `find` 함수가 있습니다. 배열, `list`, `vector` 등의 다양한 자료구조의 주어진 구간에서 특정 값을 검색 할 때 사용합니다. `find` 함수를 사용하려면 `<algorithm>` 헤더를 포함해야 합니다. 다음에 간단한 예제가 있습니다.

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

int main()
{
    int x[10] = { 1,2,3,4,5,6,7,8,9,10 };

    int* p = find(x, x + 10, 5); // x ~ x+10 사이에서 5을 검색합니다.

    cout << *p << endl; // 5
}
```

[참고] `find()` 함수에 대한 자세한 사용법은 뒤에서 배우게 됩니다. 또한, 정확히는 “`find()` 함수”가 아닌 “`find()` 알고리즘”이라는 용어를 사용 하는데, 역시 뒤에서 정확히 다루도록 하겠습니다.

이번 항목에서는 STL의 `find` 함수를 단계별로 직접 구현해 보면서 STL의 설계 철학을 살펴 보도록 하겠습니다.

## I Step1. strchr() 함수 구현하기

C언어의 표준 함수인 `strchr` 함수를 사용하면 문자열에서 특정 문자를 검색 할 수 있습니다. `strchr` 함수는 다음 처럼 간단하게 구현 할 수 있습니다. 표준 함수와의 충돌을 막기 위해 함수 이름을 `estrchr` 로 만들었습니다.

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

char* estrchr(char* s, char c)
{
    while (*s != c && *s != 0) ++s;
    return *s != c ? 0 : s;
}

int main()
{
    char s[] = "abcdefg";
    char* p = estrchr(s, 'c');
}
```

[참고] 정확하게 만들려면 문자열의 상수성도 고려해야 하지만 위 코드에서는 주제에 집중하기 위해 상수성을 고려하지 않았습니다.

`estrchr` 함수는

- 검색에 성공하면 주소를 리턴하고,
- 실패할 경우 0을 리턴 합니다.

C언어의 `strchr()` 함수 도 위의 `estrchr()` 함수와 유사하게 구현 되어 있습니다. 이때, `estrchr()` 함수에는 몇가지의 단점이 있습니다.

- 검색에 포함되는 모든 문자는 NULL 문자로 종료 되어야 합니다.
- 전체 문자열에서만 검색이 가능하고, 부분 문자열로부터 검색할 수가 없습니다.

문자열에서 특정 문자를 검색할 때,

**“전체 문자열 뿐 아니라 부분 문자열(문자열의 특정 구간)에서도 검색”**

할 수 있다면 함수의 활용도를 확장 할 수 있지 않을까요 ?

## I Step2. 검색 구간의 일반화(generic)

전체 문자열 뿐 아니라 문자열의 특정 구간에서도 검색을 가능하게 하려면, 문자열의 시작 주소 뿐 아니라 마지막 주소도 전달해야 합니다. 아래 코드는 검색 구간의 시작과 끝 주소를 전달해서 문자를 검색하도록 `estrchr` 함수를 변경했습니다

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

char* estrchr(char* first, char* last, char value)
{
    while (*first != value && first != last)
        ++first;
    return *first != value ? 0 : first;
}

int main()
{
    char s[] = "abcdefg";

    char* p = estrchr(s, s + 4, 'c'); // "abcde"에서 'c' 검색.
                                     // 단, e는 검색의 대상은 아님

    if (p == 0)
        cout << "fail." << endl;
    else
        cout << *p << endl;
}
```

위의 `estrchr` 함수에서 주의할 점은 검색 구간의 끝인 `last` 는 검색 대상에는 포함되지 않는다는 점입니다. 즉, `s~s+4` 는 “abcde”를 나타내지만 ‘e’ 자체는 검색의 대상이 아닙니다. 따라서, 아래 처럼 사용하면 검색 실패를 나타내는 0이 리턴 됩니다.

```
char* p = estrchr(s, s + 4, 'e');
```

이와 같이 검색 대상에 포함되는 않은 `last`를 “**past the end**” 라고 부르고 아래와 같이 표기 합니다.

[first, last)

대괄호( '[' )는 검색에 포함된다는 의미이고 괄호( ')' ) 는 검색에 포함되지 않는다는 의미 입니다.

이제 `estrchr` 함수는

**“문자열 전체 구간 뿐 아니라 특정 구간 `[first, last)` 에서도 선형 검색을 수행”**

할 수 있게 되었습니다.

하지만, 단점은 `estrchr` 함수는 “문자 배열에서만 선형 검색”을 수행할 수 있습니다. 문자 배열 뿐

**“모든 배열로 부터 선형 검색을 수행 할 수 있다면”**

함수의 활용도가 더욱 넓어지지 않을까요 ?

## I Step3. 검색 대상 타입의 일반화 - 함수 템플릿

`xtrchr` 함수를 일반 함수가 아닌 템플릿으로 만들면 문자 배열 뿐 아니라 모든 타입의 배열에서 선형 검색을 수행할 수 있습니다. 다음 코드는 템플릿으로 만든 `estrchr` 입니다. `char` 대신 'T'를 사용했습니다.

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

template<typename T> T* estrchr(T* first, T* last, T value)
{
    while (*first != value && first != last)
        ++first;
    return *first != value ? 0 : first;
}

int main()
{
    double d[] = { 1,2,3,4,5,6,7,8,9,10 };

    double* p = estrchr(d, d + 4, 5.0); // { 1,2,3,4 } 에서 검색 ( 5는 검색대상아님)

    cout << *p << endl;
}
```

이제, `estrchr()` 함수는 문자열 뿐 아니라

“모든 타입의 배열에 대해 `[first, last)` 구간에서 선형 검색을 수행”

할 수 있게 되었습니다.

하지만, 아직 몇가지의 문제 가 있습니다.

- ① 문자열 뿐 아니라 모든 타입의 배열에서 검색하므로 함수 이름 `estrchr`는 적합하지 않습니다.
- ② `estrchr`의 함수 인자로 `T*` 라고 표시하면 진짜 포인터만 사용할 수 있습니다. 진짜 포인터가 아닌 스마트 포인터와 같은 객체를 사용할 수 없습니다.
- ③ 구간의 타입과 검색 대상 타입이 연관되어 있습니다. 검색 구간의 타입이 `T*` 이고, 검색 대상 타입이 `T` 타입 이므로, `double` 배열에서 `int` 를 검색할 수 없습니다.

문제점에 대한 자세한 설명은 아래 코드를 참고해 보세요.

```

// 아래 템플릿의 모양을 잘 생각하면서 main 함수의 주석을 생각해 보세요
// 검색 구간이 T* 일때 검색 대상 타입은 T이어야 합니다.
template<typename T> T* estrchr(T* first, T* last, T value)
{
    //.....
}
int main()
{
    double d[] = { 1,2,3,4,5,6,7,8,9,10 };

    double* p = estrchr(d, d + 4, 5);    // error. 이순간, first, last의 타입은 double*
                                         // 로 결정됩니다.
                                         // 하지만, 5는 double 이 아니고 int 타입입니다.

    smart_pointer<double> p1(d);
    smart_pointer<double> p2(d+4);
    estrchr(p1, p2, 5.0);    // error. estrchr()함수는 구간의 타입이 T* 이므로
                             // 진짜 포인터만 사용해야 합니다.
                             // 스마트 포인터를 사용할 수 없습니다.
}

```

[참고] 물론 스마트 포인터는 자원 삭제와 연관성이 크므로 위 코드 처럼 동적 할당 되지 않은 배열의 주소를 담은 것은 바람직하지 않습니다. 위 코드에서는 estrchr 함수의 인자로 객체를 전달할 수 없다는 관점으로만 생각하시기 바랍니다. 뒤에 반복자를 다룰 때 보다 명확하게 설명됩니다

다음 단계에서, step3의 문제점을 모두 해결해 보도록 하겠습니다.



## I Step4. more generic

이전 단계에서의 문제점을 해결 하기 위해서는 `estrchr()` 함수를 다음 처럼 변경해야 합니다.

- ① 함수이름을 `efind` 로 변경합니다.
- ② 구간의 타입과 검색하는 요소의 타입의 연관성을 없애기 위해 템플릿 인자를 2개 사용합니다.
- ③ 구간의 타입은 `T*` 로 할 경우 포인터만 사용 가능 하므로 `*`를 표기하지 않습니다.

완성된 코드는 다음과 같습니다.

```
// 1. 구간의 타입은 T이고 검색 대상 타입은 U 이므로 서로 별개의 타입입니다.
// 2. 구간의 타입을 표시할때 * 를 표기하지 않았으므로 구간을 나타내는 것은
//   진짜 포인터 뿐 아니라 포인터를 흉내낸 객체도 사용 가능합니다.
template<typename T, typename U> T efind(T first, T last, U value)
{
    while (*first != value && first != last)
        ++first;
    return *first != value ? 0 : first;
}

int main()
{
    double d[] = { 1,2,3,4,5,6,7,8,9,10 };

    double* p = efind(d, d + 4, 5); // ok. double 배열에서 int를 검색 할 수 있습니다.

    cout << *p << endl;
}
```

이제, `efind` 함수 템플릿은 `double`의 배열에서 `int` 값을 검색 할 수도 있게 되었습니다. 이제 마지막으로 `efind` 의 리턴 값에 대해서 생각해 봅시다.

## I Step5. 검색 실패시 0 대신 last 리턴

efind 함수는 [first, last) 의 구간을 검색하는데, last는 검색 대상에는 포함 되지 않습니다. 따라서, 검색에 성공할 경우는 last 가 리턴 되지는 않습니다. 그렇다면, 검색에 실패 했을 때 0이 아닌 last를 리턴 하면 어떨까요 ?

0대신 last를 리턴 하면 다음과 같은 장점이 있습니다.

- 코드 구현이 쉬워 지고 함수가 약간 빨라 지게 됩니다.
- 리턴 값에 대한 활용도를 좀더 높아집니다.

아래의 2개 코드를 비교해 보세요. 두 함수의 마지막 줄을 비교해 보세요.

검색 실패시 0을 리턴 하는 경우	검색 실패시 last를 리턴 하는 경우
<pre>template&lt;typename T, typename U&gt; T efind(T first, T last, U value) {     while (*first!=value &amp;&amp; first != last)         ++first;     return *first != value ? 0 : first; }</pre>	<pre>template&lt;typename T, typename U&gt; T efind(T first, T last, U value) {     while (*first!=value &amp;&amp; first != last)         ++first;     return first; // 검색에 실패 하면                 // first 는 last위치가 됩니다. }</pre>

또한, 검색 실패시 last를 리턴 하면 리턴 값의 활용도가 높아 지게 됩니다.

```
// 다음의 efind()는 검색 실패시 last 위치를 리턴 합니다.
template<typename T, typename U> T efind(T first, T last, U value)
{
    while (*first != value && first != last)
        ++first;
    return first;
}
int main()
{
    double d[] = { 1,2,3,4,5,6,7,8,9,10 };

    double* ret = efind(d, d + 4, 6);
```

```
// 검색 실패시 0이 아닌 last인 d+4를 리턴 하게 됩니다.  
if (ret == d + 4)  
{  
    // 검색 실패 시 다시 다음 구간을 검색하려고 합니다.  
    // ret의 의미는 다음 구간의 시작이 됩니다.  
    ret = efind(ret, ret + 4, 5);  
}  
}
```

결국, 임의 구간을 검색할 때 검색 실패시 리턴값은 다음 구간의 시작을 나타내게 됩니다. 따라서, 다음 구간을 검색할 때 편리하게 활용될 수 있습니다.

## I Step6. efind 정리

완성된 efind 함수 템플릿의 모양은 다음과 같습니다.

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

template<typename T, typename U> T efind(T first, T last, U value)
{
    while (*first != value && first != last)
        ++first;
    return first;
}

int main()
{
    double d[] = { 1,2,3,4,5,6,7,8,9,10 };
    double* p = efind(d, d + 4, 5);

    // 검색 실패에 대한 조사는 리턴값과 last를 비교해야 합니다.
    if (p == d + 4)
        cout << "fail." << endl;
}
```

완성된 efind 함수 템플릿과 C언어의 strchr 함수와 비교하면 다음과 같은 특징(장점)이 있습니다.

	strchr()	efind()
검색 대상 타입	문자 배열	모든 타입의 배열
검색 구간	전체 문자열	[first, last) 의 임의의 구간
구간의 표현	char* 로 표현	모든 타입의 포인터, 포인터 역할의 객체
검색 실패시 리턴값	0	last

[참고] 반복자를 도입하면 efind()는 모든 타입의 “배열” 이 아닌 모든 타입의 “선형 자료구조”로 확장 됩니다. 자세한 이야기는 “반복자(iterator)”를 이야기 할 때 다루게 됩니다.

## I 용어 정리

`efind()` 함수 처럼 함수 또는 클래스를 만들 때 특정 타입이 아닌 모든 타입에 적용할 수 있도록 만드는 것을

### **“일반화 프로그래밍 ( Generic Programming )”**

이라고 합니다.

앞에서 만든 `efind` 는 결국 STL의 `find` 와 동일합니다. 또한, STL에서는 `find`와 같은 일반화 함수를 “함수” 라고 부르지 않고 “알고리즘” 이라고 부르고 있습니다.

### **“find 함수” => “find 알고리즘”**

이 책에서도, 이제부터 는 “find 알고리즘” 이라고 부르도록 하겠습니다.

또한, 검색 구간을 전달할 때 `last`는 마지막 요소가 아닌 마지막 다음 요소를 가리키게 됩니다. 이것을 흔히

### **“past the end”**

라고 부릅니다.

## I 연습문제

C 표준에는 문자열 복사할 때 `strcpy()` 함수를 사용합니다. 하지만, 이 함수는 다음과 같은 단점이 있습니다.

- 전체 문자열 복사만 가능합니다. 부분 문자열 복사가 안됩니다.
- 문자 배열만 복사 가능합니다. `int`, `double` 등의 배열을 복사 할 때는 사용할 수 없습니다.

### 실습 1.

`strcpy` 함수를 `find` 와 같이 일반화 함수로 만들어 보세요. `efind`를 만들 때 처럼 단계별로 만들어 보세요. 함수 이름은 `xcopy` 로 만들어 보세요.

### 실습 2.

STL에는 “**copy** 알고리즘”이 있습니다. 인터넷에서 `copy()` 의 구현 소스를 구한 후, 여러분이 만든 `xcopy` 와 비교해 보세요

[참고] 모든 타입의 배열을 복사하기 위해 `memcpy()` 함수를 사용할 수 도 있지만, `memcpy()` 함수는 다음과 같은 단점이 있습니다.

- ① 배열 요소의 타입이 객체 타입인 경우 각 요소는 대입 연산자를 통해서 복사가 되어야 합니다. `memcpy()`는 대입연산자를 호출 할 수 없습니다.
- ② `memcpy()`는 배열과 같은 연속된 메모리만 사용 가능합니다. `list`와 같이 연속되지 않은 자료구조에는 사용할 수 없습니다. 하지만, `copy()`알고리즘은 `list`도 사용 가능 합니다. (뒷장에서 반복자(iterator)를 배울 때 설명됩니다.)

항목 2-2

# Container & Iterator

## 학습 목표

- slist 만들기
- iterator의 개념

# 1. STL 컨테이너

## I Single Linked List

array, list, tree 등의 자료 구조는 data를 저장하는 용도로 사용됩니다. STL에서는 이와 같이 data를 저장하는 자료 구조를 “컨테이너(Container)”라고 합니다.

이번에는 간단한 Single Linked List 를 만들어 보도록 하겠습니다. 주제에 집중하기 위해 최대한 간단하게 만들어 보도록 하겠습니다.

```
template<typename T> struct Node
{
    T data;
    Node* next;
    Node(T a, Node* n) : data(a), next(n) {}
};

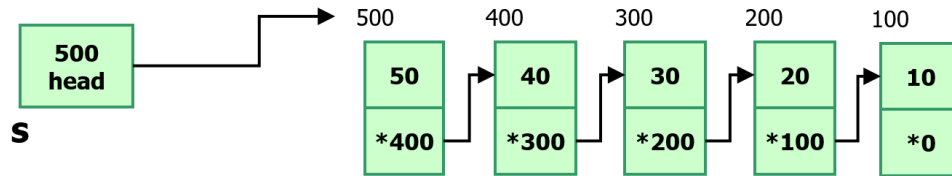
template<typename T> class slist
{
public:
    Node<T>* head = 0;
    void push_front(const T& a) { head = new Node<T>(a, head); }
};

int main()
{
    slist<int> s;
    s.push_front(10);
    s.push_front(20);
    s.push_front(30);
    s.push_front(40);
    s.push_front(50);
}
```

[참고] 주제에 집중하기 위해 pop\_xxx() 함수와 소멸자 등은 생략했습니다. 또한, 설명을 위해서 멤버 데이터 head를 public 영역에 놓았습니다



위 코드에서 `main` 함수가 실행 되었을 때 메모리 구조는 다음과 같습니다. `push_front` 함수, `Node` 생성자의 코드를 참고 하면 됩니다.



## I 싱글 리스트와 `efind` 알고리즘의 결합

우리는 이전의 항목에서 모든 타입의 배열의 `[first, last)` 구간에서 선형 검색을 수행하는 `efind` 알고리즘을 만들었습니다. 활용성을 최대한 높이기 위해서 다양한 기법을 추가 했습니다. 그렇다면

**“`efind` 알고리즘을 사용해서 `slist` 에서 선형 검색을 수행할 수 있을까요 ?”**

안됩니다.

그 이유는 다음과 같습니다.

배열의 요소를 가리키는 포인터를 `p1`, `slist`의 요소를 가리키는 포인터를 `p2`라고 할 때 각각 배열과 리스트는 구간을 이동하는 방식과 요소에 접근하는 방식이 다릅니다.

	배열	리스트
다음 요소로 이동	<code>++p1</code>	<code>p2 = p2-&gt;next</code>
요소의 값에 접근	<code>*p1</code>	<code>p2-&gt;data</code>

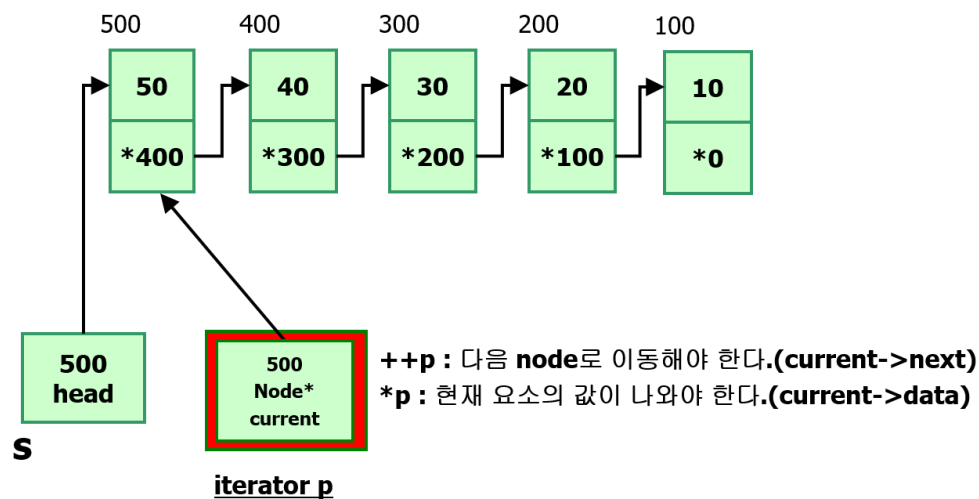
따라서, `efind` 알고리즘을 사용해서는 `slist`에서는 값을 검색 할 수 없습니다. 그런데, `efind` 를 사용해서 배열 뿐 아니라 `slist`에서도 값을 검색할 수 있다면 좋지 않을까요 ?

## 2. STL 반복자

### I 반복자(iterator) 개념

efind 알고리즘은 구간을 이동할 때 ++연산을 사용하고 요소를 접근 할때는 \* 연산자를 사용합니다. 따라서, efind를 사용해서 slist를 검색하려면 slist 의 요소를 이동할 때 p->next 를 방식으로 이동하지 말고 ++p로 이동 할 수 있어야 합니다.

slist의 요소를 가리킬 수 있는 포인터 역할의 객체를 만들면 어떨까요 ? 아래 그림을 생각해 봅시다.



위 그림에서 iterator p 는 진짜 포인터가 아닌 객체입니다. 그런데, 멤버 데이터로 Node\* current 를 가지고 있고, 500번지로 초기화 되어 있다면 p는 slist의 첫번째 요소를 가리킬 수 있게 됩니다. 즉, 객체이지만 포인터 처럼 slist 안에 있는 node를 가리킬 수 있게 됩니다. 또한, p는 객체이므로 ++, \* 등의 연산자를 재정의 할 수 있습니다. 그렇다면 ++ 연산자를 재정의해서 current = current->next 코드를 제공한다면 slist의 구간을 이동할 때 ++로 이동할 수 있지 않을까요 ?

이와 같이 어떤 객체가 포인터처럼 요소를 가리키고, ++로 이동하고, \*로 값을 접근할수 있게 만드는 것을 STL 에서는 반복자(iterator) 라고 합니다.

또한, efind 알고리즘을 만들 때 구간을 나타내는 first, last의 타입을 T\* 가 아닌 T로 만들었기때문에 진짜 포인터 뿐 아니라, 객체(반복자)도 전달할 수 있습니다.

efind()의 인자로 전달되기 위한 조건

```
// first, last의 타입이 T* 가 아닌 T 이므로 진짜 포인터가 아닌 객체도 전달할 수
// 있습니다. 하지만, T가 가져야 하는 조건이 있습니다.
template<typename T, typename U> T efind(T first, T last, U value)
```

```
{
    while (*first != value && first != last) // 조건1. *, != 연산이 가능해야 합니다.
        ++first;                          // 조건2. ++ 연산이 가능해야 합니다.
    return first;
}
```

결국, `efind` 알고리즘에 전달될 수 있게 하려면 `iterator` 는 다음과 같은 요구조건을 만족해야 합니다.

- ① `slist` 의 각 요소(`node`)를 가리킬 수 있어야 합니다.( 멤버 데이터로 `Node*`가 있어야 합니다.)
- ② ++연산으로 다음 요소로 이동할 수 있어야 합니다.( `operator++()` 함수가 필요)
- ③ \*연산으로 요소가 가진 값을 얻을 수 있어야 한다.( `operator*`() 함수가 필요)
- ④ 2개의 `iterator` 객체에 != 연산을 할 수 있어야 한다.(`operator !=()` 함수가 필요)

아래 코드는 완성된 `iterator` 클래스 입니다.

```
// slist의 요소를 가리키는 포인터 역할의 객체입니다.
// 반복자(iterator) 라고 부릅니다.
template<typename T> class slist_iterator
{
    Node<T>* current;
public:
    slist_iterator(Node<T>* p = 0) : current(p) {}

    // efind()알고리즘에 전달되기 위해서는 ++, *, != 가 필요합니다.
    // efind()에서 사용하지는 않지만, == 도 제공하는 것이 좋습니다.
    slist_iterator& operator++()
    {
        current = current->next;
        return *this;
    }
    T& operator*() { return current->data; }
    bool operator==(const slist_iterator& it) const { return current == it.current; }
    bool operator!=(const slist_iterator& it) const { return current != it.current; }
};

int main()
{
    slist<int> s;
    s.push_front(10);
    s.push_front(20);
```

```
s.push_front(30);

slist_iterator<int> p(s.head); // s의 첫번째 요소를 가리키는 객체입니다.

// p는 객체이지만 포인터와 유사하게 사용할 수 있습니다.
cout << *p << endl;
++p;
cout << *p << endl;
}
```

slist\_iterator 덕분에 slist에 있는 요소를 열거하거나 접근할 때 실제 포인터와 동일하게 ++, \* 연산을 사용할 수 있게 되었습니다.

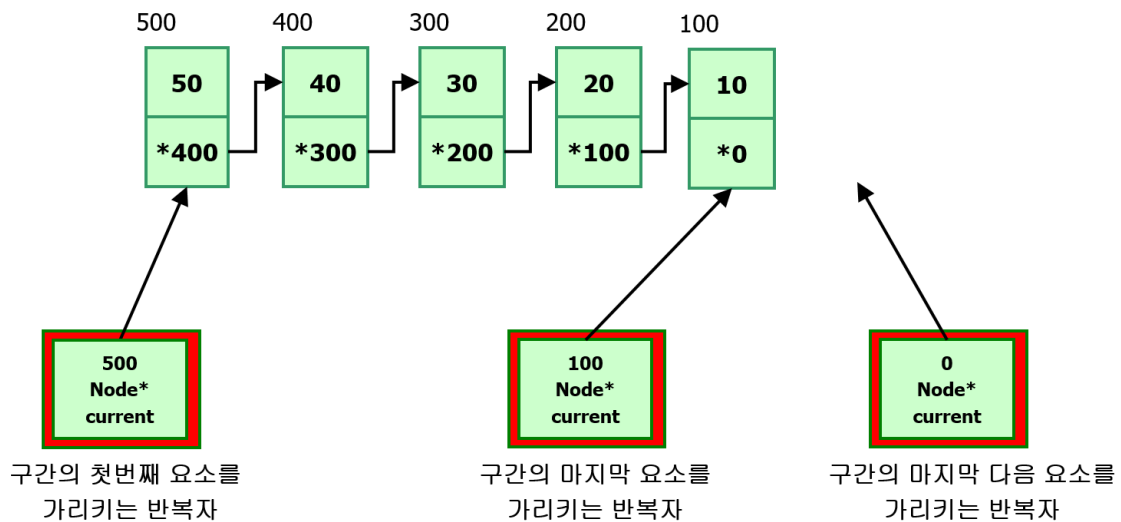
이처럼

“자료구조의 내부 구조(연속된 메모리이거나 연속되지 않은 메모리 이거나)에 상관 없이 동일한 방식으로 열거 할 수 있게 하는 디자인 기법을 GoF's 디자인 패턴에서 “반복자(iterator)” 패턴” 이라고 합니다.

## I begin(), end()

efind() 알고리즘은 [first, last) 구간에서 선형 검색을 수행하는데, last를 검색의 대상이 아닙니다. 따라서, slist의 전체를 검색하려면 마지막 요소가 아닌 마지막 다음 요소를 가리키는 반복자를 전달해야 합니다.

결국 아래 그림에서 마지막 요소의 next 에서는 0이 들어 있으므로 마지막 다음을 가리키는 반복자는 current 멤버의 값을 0으로 초기화 하면 됩니다.



이제 slist 안에 자신의 시작과 마지막 다음 요소를 가리키는 반복자를 리턴 하는 begin(), end() 멤버 함수를 추가 합니다.

```
template<typename T> class slist
{
    Node<T>* head = 0;
public:
    void push_front(const T& a) { head = new Node<T>(a, head); }

    // 자신이 가진 시작요소와 마지막 다음 요소를 가리키는 반복자는 리턴하는 함수.
    slist_iterator<T> begin() { return slist_iterator<T>(head); }
    slist_iterator<T> end()   { return slist_iterator<T>(0); } // past the end iterator
};

int main()
{
    slist<int> s;
    s.push_front(10);
    s.push_front(20);
}
```

```
s.push_front(30);
slist_iterator<int> p1 = s.begin();
slist_iterator<int> p2 = s.end(); // past the end

while (p1 != p2)
{
    cout << *p1 << endl;
    ++p1;
}
// slist에 efind() 알고리즘을 사용 할 수 있습니다
slist_iterator<int> ret = efind(s.begin(), s.end(), 20);
}
```

위 코드에서 반복자 p1, p2는 실제 포인터는 아니지만 연산자 재정의 덕분에 포인터 처럼 사용할수 있습니다. 또한, efind() 알고리즘으로 slist의 선형검색도 가능하게 되었습니다.

“반복자 덕분에 efind()는 검색 대상 구간, 검색 대상 타입 뿐 아니라,  
검색 대상 자료구조에도 독립적으로 일반화(Generic) 되었습니다.”

## I typedef

앞에서 만든 slist의 반복자 타입은 slist\_iterator 입니다. 만약 더블 리스트가 있다면 더블 리스트도 자신만의 반복자 타입을 가지게 될 것입니다. 모든 컨테이너의 반복자 타입의 이름을 일관된 형태로 사용할 수 있게 한다면 편리하지 않을까요 ?

STL에는 모든 컨테이너는

**“자신의 반복자 이름을 “iterator”라는 이름으로 외부에 알리기로 약속”**

되어 있습니다. 따라서, slist 안에 아래와 같은 typedef를 추가 합니다

반복자의 이름을 "iterator"로 외부에 알려줍니다.

```
template<typename T> class slist
{
    // .....
    typedef slist_iterator<T> iterator;
    // .....
};
int main()
{
    slist<int> s;
    s.push_front(10);

    // slist_iterator<int> p = s.begin(); // 이렇게 사용하지 말고

    slist<int>::iterator p = s.begin(); // 이렇게 사용합니다.
}
```

결국, 컨테이너에서 반복자를 꺼낼 때는 다음과 같이 사용하면 됩니다.

**“컨테이너이름<요소타입>::iterator p = s.begin()”**

## I SOURCE CODE

다음 코드는 `efind()`, `slist`, `slist_iterator`에 대한 전체 소스입니다.

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

template<typename T, typename U> T efind(T first, T last, U value)
{
    while (*first != value && first != last)
        ++first;
    return first;
}

template<typename T> struct Node
{
    T data;
    Node* next;
    Node(T a, Node* n) : data(a), next(n) {}
};

template<typename T> class slist_iterator
{
    Node<T>* current;
public:
    inline slist_iterator(Node<T>* p = 0) : current(p) {}

    inline slist_iterator& operator++()
    {
        current = current->next;
        return *this;
    }
    inline T& operator*() { return current->data; }
    inline bool operator==(const slist_iterator& it) const
    { return current == it.current; }
    inline bool operator!=(const slist_iterator& it) const
    { return current != it.current; }
};

template<typename T> class slist
{
    Node<T>* head = 0;
public:
    void push_front(const T& a) { head = new Node<T>(a, head); }
    typedef slist_iterator<T> iterator;
    iterator begin() { return iterator(head); }
```



```
    iterator end()    { return iterator(0); }
};
int main()
{
    slist<int> s;
    s.push_front(10);
    s.push_front(20);
    s.push_front(30);
    s.push_front(40);
    slist<int>::iterator p = efind(s.begin(), s.end(), 20);

    if (p == s.end())
        cout << "fail" << endl;
    else
        cout << *p << endl;
}
```

### 3. STL 라이브러리의 구조

#### Ⅰ 자료 구조와 알고리즘의 분리

대부분의 프로그램을 작성할 때는 동적 배열, list, tree, hash 등의 자료구조가 필요 합니다. 이런 자료구조 들은 대부분 여러 개의 데이터를 보관하고 있게 됩니다. 그래서 STL에는 이런 자료구조를 나타내는 클래스를 “컨테이너(Container)” 라고 부릅니다. [참고] 다른 언어에서는 “컬렉션(collection)” 이라고 부르는 경우도 있습니다.

일반적으로 어떤 자료구조가 있다면 검색, 정렬, 삭제, 변경 등의 다양한 알고리즘이 필요합니다. “객체 지향의 캡슐화 개념”에서는 이런 함수를 멤버 함수로 제공하는 것이 원칙입니다.

그렇다면, 다음의 경우를 생각해 봅시다.

- vector, list, string 의 3개의 컨테이너가 필요 합니다.
- 선형검색(find), 뒤집기(reverse), 요소 변경(replace) 의 3개의 알고리즘이 필요합니다.
- int, double, char\* 타입에 대해서 동작해야 합니다.

먼저, 템플릿을 고려하지 않으면 3개의 클래스에 3개의 멤버 함수를 3개의 타입으로 제공해야 하므로

**“3개 클래스 \* 3개 멤버 함수 \* 3개 타입 = 27개의 함수가 필요합니다.”**

타입 독립적으로 만들기 위해 각 멤버 함수를 템플릿으로 제공한다고 해도

**“3개 클래스 \* 3개 멤버 함수 템플릿 = 9개의 함수(템플릿)가 필요합니다.”**

하지만, 3개의 알고리즘 함수를 “멤버 함수가 아닌 일반 함수”로 제공하면

**“모든 자료구조에 사용 가능한 3개의 일반 함수 템플릿만 있으면 충분”**

하게 됩니다.

10개의 클래스(자료구조, 컨테이너)에 10개의 알고리즘을 10개의 타입으로 제공한다면 1000개의 멤버 함수가 필요하고, 함수 템플릿을 타입에 독립적으로 사용 한다 해도 100개의 함수 템플릿이 필요하지만, 멤버 함수가 아닌 일반 함수 템플릿으로 제공하면 단, 10개의 함수 템플릿만 있으면 됩니다.

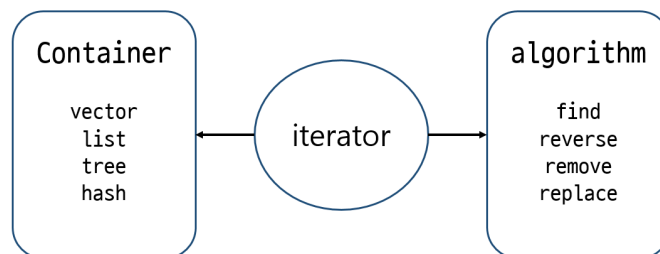
그런데, 문제는 `find` 알고리즘을 만들려면 하나의 요소에서 다음 요소로 이동 할 수 있어야 하는데, 자료구조가 연속된 메모리인 `vector` 와 연속되지 않은 `list` 는 이동 방식이 다르다는 점입니다.

그래서,

“자료구조의 내부구조에 상관없이 동일한 방식으로 이동 할 수 있는 개념”

이 필요하고, 그것이 바로 “반복자(iterator)” 라는 개념입니다.

그래서, 흔히 STL 의 3개 요소를 “컨테이너(Container)”, “알고리즘(Algorithm)”, “반복자(iterator)” 라고 합니다.



이와 같은 구조 때문에, 흔히 STL을

“자료구조와 알고리즘을 분리한 라이브러리”

라고도 합니다.

항목 2-3

## STL 라이브러리의 특징

### 학습 목표

- Member Type
- C++17 과 STL
- Concept

## 1. Member Type

### | value\_type

- STL 에는 멤버 데이터, 멤버 함수 이외에도 “멤버 타입“ 이라는 개념 제공.
- 템플릿 의존 타입(template dependent type) 의 경우 typename을 표기 해야 합니다.
- value\_type은 컨테이너가 저장하는 타입, 반복자가 가리키는 타입을 나타냅니다.

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

template<typename T>
void print(T& c)
{
    // T : list<double>
    // T::value_type => list<double>::value_type
    //                      ==> double
    // typename : value_type을 타입의 이름으로 해석.
    typename T::value_type n = c.front(); // double

    cout << n << endl;
}

int main()
{
    list<double> v = { 1,2,3 };
    print(v);
}
```

## 2. C++17과 STL

### ■ class template type deduction

C++17 부터는 클래스 템플릿도 템플릿 인자를 명시적으로 지정하지 않아도 컴파일러가 추론할 수 있습니다.

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

template<typename T> class List
{
public:
    List(int sz, T initValue) {}
};

int main()
{
    List<int> s1(10, 0);
    List      s2(10, 0); // C++17 부터 ok.

    list<int> s3 = { 1,2,3 };
    list      s4 = { 1,2,3 }; // C++17 style
}
```

## 3. Concept

### I Concept - C++20

- 템플릿은 “모든 타입”에 적용되는 것이 아니라 “조건을 만족”하는 타입에 대해서 동작한다.
- 특정 함수(또는 클래스)를 사용하기 위해 타입이 가져야 하는 조건.
- min 함수에 전달되는 인자는 LessThanComparable을 만족해야 한다.
- C++20 부터는 코드를 사용해서 Concept을 정의 할 수 이다.

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

struct Point
{
    int x, y;
};

int main()
{
    Point p1, p2;

    min(p1, p2); // < 연산을 지원해야 한다. LessThanComparable Concept을 만족해야 한다.

    int n1 = min(1, 2);
}
```

## Section 3.

# iterator



항목 3-1

# iterator basic

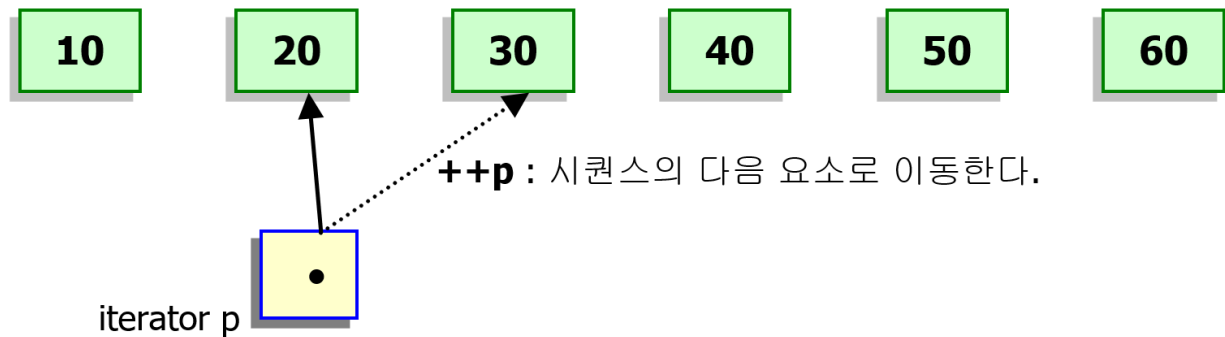
## 학습 목표

- 반복자 기본 개념
- 반복자 구간
- Past the end
- 무효화된 반복자
- 반복자의 종류

## 1. 반복자 기본 개념

### ■ 반복자(iterator) 란 ?

반복자(iterator)는 포인터와 유사한 객체로서 STL의 알고리즘 함수가 컨테이너에 저장된 객체들의 시퀀스를 순회 할 때 사용됩니다. 컨테이너와 알고리즘 함수의 사이를 연결해주는 중간자로서의 역할을 담당하고, 반복자 덕분에 저장방식(자료구조)에 관계 없이 동일한 알고리즘 함수를 구현 할 수 있습니다.



### ■ 반복자 구간

시퀀스의 시작을 가리키는 first, 마지막 다음을 가리키는 last 의 반복자 한 쌍을 “반복자 구간”이라고 합니다. 반복자 구간은 “[first, last)” 처럼 표기합니다. 또한 first에서 시작해서 operator++ 연산으로 도달 가능한 경우 “유효(Valid, 도달가능-reachable)한 구간”이라고 하고, 모든 STL의 알고리즘은 넘겨 받은 구간이 유효하다는 가정하에 작업을 수행합니다. 또한 아무 요소도 없는 구간은 “first == last” 가 되는데 이를 “빈 구간(empty)”라고 합니다.

## I past the end

Container의 `end()` 함수를 통해서 얻게 되는 iterator는 마지막 요소가 아니라 마지막 다음을 가리키는데 이를 보통 `past-the-end iterator`라고 부릅니다. 이 iterator를 참조하는 것은 오류가 됩니다.

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

int main()
{
    vector<int> v(5, 0);
    vector<int>::iterator p = v.end();
    cout << *p << endl; // bug. undefined
}
```

## I 무효화된 반복자

반복자는 무효화 될 수 있는데 무효화된 반복자를 사용하는 것은 오류가 됩니다.

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

void main()
{
    vector<int> v(5, 0);
    vector<int>::iterator p = v.begin();
    cout << *p << endl;
    v.resize(10);
    cout << *p << endl; // 오류. 무효화된 반복자
}
```

## I 컨테이너의 반복자 타입과 반복자 꺼내기

특정 컨테이너의 반복자 타입은 다음과 같습니다.

“컨테이너이름<요소타입>::iterator”

하지만, C++11 의 auto를 사용하면 보다 편리하게 사용할 수 있습니다.

```
vector<int>::iterator p = v.begin(); // 정확한 타입의 이름을 사용하는 코드
```

```
auto p = v.begin(); // auto를 사용하면 쉽게 사용할 수 있습니다.
```

또한, STL 컨테이너에서 반복자를 꺼낼 때는 다음의 2가지 방법이 있습니다.

① 멤버 함수 begin()/end()를 사용하는 방법

② 멤버 함수가 아닌 begin()/end()를 사용하는 방법 - C++11 부터 가능.

두가지 방법 중 멤버 함수를 사용하는 방법 보다 멤버가 아닌 함수를 사용하는 것이 좋은데, 멤버가 아닌 일반 함수를 사용하면 STL 컨테이너 뿐 아니라 일반 배열을 사용하는 코드로 쉽게 변경이 가능합니다. 다음 코드를 참고 하세요.

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

int main()
{
    //int v[10] = { 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };

    // 반복자 꺼내기 1. 멤버 함수 begin()/end()사용
    auto p = find(v.begin(), v.end(), 5); // v가 배열인 경우 error가 발생합니다.

    // 반복자 꺼내기 2. 멤버가 아닌 일반 함수 begin()/end()사용
    auto p = find(begin(v), end(v), 5); // v가 STL 컨테이너 뿐 아니라 배열이라도
    // 아무 문제 없습니다.
}
```

## I 반복자의 종류

반복자를 나타내는 설명들 중에서 재미있는 설명이 있습니다.

“반복자 처럼 동작하는 것은 모두 반복자이다.”

이 말의 의미는 ++연산으로 다음 요소로 이동 가능하고, \*로 값에 접근할 수 있는 모든 것은 반복자라는 의미 입니다. 따라서, 일반 포인터도 반복자입니다. 반복자에는 다음과 같은 것들이 있습니다.

- ① 일반 포인터
- ② STL의 각 컨테이너에서 begin()/end() 함수로 얻는 반복자
- ③ 스트림 반복자 ( stream iterator )
- ④ 삽입 반복자 ( insert iterator )

그 외에 C++17 부터 새롭게 등장하는 asio, coroutine, filesystem등 일반 자료 구조가 아닌 다른 분야의 라이브러리에도 반복자의 개념을 가지고 사용하게 됩니다. C++에서 반복자는 아주 넓은 의미로 활용됩니다.

## 항목 3-2

# iterator category

### 학습 목표

- STL과 error 메시지
- Iterator\_category
- 알고리즘과 반복자 카테고리
- advance() 알고리즘 만들기

## 1. iterator category

### ■ STL과 error 메시지

STL을 잘못 사용하면 이해 하기 힘든 어려운 에러가 발생합니다.

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

int main()
{
    list<int> s{ 1,3,5,7,9,2,4,6,8,10 };

    sort(s.begin(), s.end());    // error.
}
```

### ■ iterator\_category

수행 가능한 연산에 따른 반복자의 분류.

알고리즘에 전달되는 반복자의 종류를 구별하기 위해 사용.

	increment read	increment read (multiple passes)	write	decrement	random access +,-,[]	continuous storage
input iterator	0					
output iterator			0			
forward iterator	0	0				
bidirectional iterator	0	0		0		
random access iterator	0	0		0	0	
contiguous iterator(C++17)	0	0		0	0	0

[참고] 쓰기 가능한 반복자를 mutable iterator 라고 합니다.

## ■ 일반화 알고리즘과 알고리즘이 요구하는 반복자 카테고리

STL의 다양한 알고리즘에는 인자로 전달되는 반복자가 요구되는 조건이 있습니다. find() 알고리즘은 input iterator 면 충분 하지만, reverse() 알고리즘을 사용하려면 반드시 bidirectional iterator를 전달해야 합니다.

generic algorithm	iterator categories
find()	입력 반복자
reverse()	양방향 반복자
sort()	임의접근 반복자

```
#include <iostream>
#include <algorithm>
#include <forward_list>
#include <list>
#include <vector>
using namespace std;

int main()
{
    // single linked list - forward iterator
    forward_list<int> fs;
    find(fs.begin(), fs.end(), 20); // ok
    reverse(fs.begin(), fs.end()); // error
    // double linked list - bidirectional iterator
    list<int> ds;
    reverse(ds.begin(), ds.end()); // ok
    sort(ds.begin(), ds.end());    // error
    ds.sort(); // ok
    // vector - random access iterator
    vector<int> v;
    sort(v.begin(), v.end());      // ok
    v.sort();                      // error
}
```



## I advance() 알고리즘 만들기

- ① iterator\_category 를 타입화 합니다.
- ② 모든 반복자 설계자는 자신의 반복자의 종류를 iterator\_category 라는 이름으로 외부에 알려줍니다.
- ③ Iterator category 에 따라 함수 오버로딩으로 xadvance()를 구현해야 합니다.

```
// 반복자 카테고리는 타입화 합니다.
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag :input_iterator_tag {};
struct bidirectional_iterator_tag :forward_iterator_tag {};
struct random_access_iterator_tag :bidirectional_iterator_tag {};

// 반복자 설계시 자신의 category를 알려준다.
template<typename T> class list_iterator
{
public:
    typedef bidirectional_iterator_tag iterator_category;
};

template<typename T> class vector_iterator
{
public:
    typedef random_access_iterator_tag iterator_category;
};

template<typename T> inline void xadvanceImp(T& p, int n, random_access_iterator_tag)
{
    cout << "임의접근" << endl; p = p + n;
}

template<typename T> inline void xadvanceImp(T& p, int n, input_iterator_tag)
{
    cout << "임의접근이 아닐때" << endl; while (n-->0) ++p;
}

// xadvance() 구현
template<typename T> inline void xadvance(T& p, int n)
{
    xadvanceImp(p, n, typename T::iterator_category());
}
```

```
int main()
{
    int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v(x, x + 10);

    vector<int>::iterator p = v.begin();
    xadvance(p, 5);
    cout << *p << endl; // 6
}
```

항목 3-3

# iterator\_traits

## 학습 목표

- iterator\_traits 개념
- iterator\_traits 필요성

## 1. iterator\_traits 개념

### I 핵심 개념

객체형 반복자와 실제 포인터의 차이점 때문에 생기는 문제를 해결하기 위한 도구.

iterator\_traits template. defined in <iterator>

```
template<typename T> struct iterator_traits
{
    typedef typename T::iterator_category iterator_category;
    typedef typename T::value_type value_type;
    typedef ptrdiff_t difference_type;

    typedef typename T::pointer pointer;
    typedef typename T::reference reference;
};

template<typename T> struct iterator_traits<T*>
{
    // get traits from pointer
    typedef random_access_iterator_tag iterator_category;
    typedef T value_type;
    typedef ptrdiff_t difference_type;

    typedef T *pointer;
    typedef T& reference;
};
```

## ■ iterator\_traits 의 필요성

앞의 항목에서 만든 xadvance()는 모든 반복자를 N 만큼 전진 시키는 일반화된 알고리즘입니다. 포인터도 반복자의 일종이므로 xadvance()로 전진 가능해야 합니다. 하지만, 포인터 안에는 iterator\_category 가 없기 때문에 error가 발생합니다.

```
template<typename T> inline void xadvance(T& p, int n)
{
    xadvanceImp(p, n, typename T::iterator_category()); // error. T는 int* 타입인데
                                                         // int* 안에는 iterator_category가 없습니다.
}
int main()
{
    int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
    int* p = x;

    xadvance(p, 5);
}
```

## ■ iterator\_traits를 사용한 해결책

iterator\_traits를 사용하면 반복자 타입 T가 객체가 아닌 실제 포인터 라도 아무 문제없이 동작하는 알고리즘을 만들수 있습니다.

```
template<typename T> inline void xadvance(T& p, int n)
{
    xadvanceImp(p, n, typename iterator_traits<T>::iterator_category()); // ok.
}
int main()
{
    int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
    int* p = x;

    xadvance(p, 5);
}
```

항목 3-4

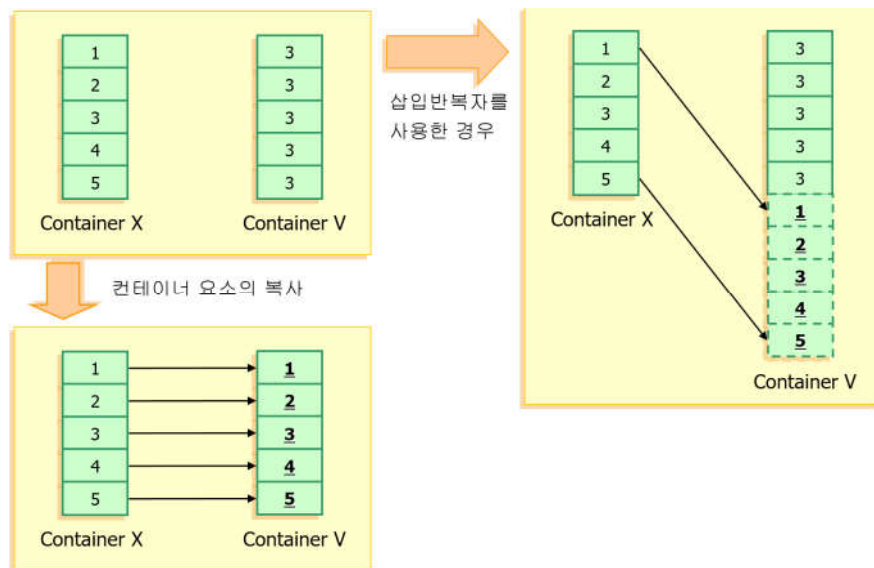
# insert iterator

## 학습 목표

- 삽입 반복자 개념
- `front_insert_iterator<>`, `front_inserter()`
- `back_insert_iterator<>`, `back_inserter()`
- `insert_iterator<>`, `inserter()`

## 1. insert iterator 개념

### 복사와 삽입



### 삽입반복자의 종류

	클래스 버전	함수 버전
전방 삽입 반복자	<code>template &lt;typename Container&gt;</code> <code>class front_insert_iterator;</code>	<code>front_inserter()</code>
후방 삽입 반복자	<code>template &lt;typename Container&gt;</code> <code>class back_insert_iterator;</code>	<code>back_inserter()</code>
임의 삽입 반복자	<code>template &lt;typename Container&gt;</code> <code>class insert_iterator;</code>	<code>inserter()</code>

## 2. 전방 삽입 반복자

### Ⅰ 전방 삽입 반복자 개념

전방 삽입 iterator는 container의 제일 앞쪽에 요소를 삽입 할 때 사용합니다.

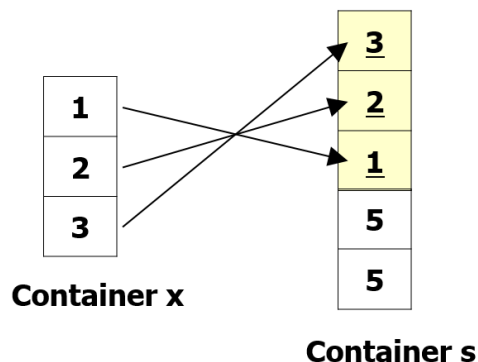
▪ `template <typename Container> class front_insert_iterator`

템플릿 인자로 요소를 삽입하고자 하는 Container의 Type을 지정하고 생성자에 해당 Container 객체를 전달 합니다. 예를 들어 `list<int>` type의 객체 `s` 에 요소를 삽입하고자 한다면

```
front_insert_iterator<list<int> > fins(s);
```

처럼 사용하면 됩니다.

전방 삽입 연산자에서 주의할 점은 **앞쪽에 하나 하나씩 차례대로 삽입 되기 때문에 삽입된 요소의 순서가 원본과는 반대가 된다는 점**입니다. 예를 들어 아래의 그림에서 container `x`에 있는 1,2,3을 5,5 가 들어 있는 Container `s`의 제일 앞쪽에 삽입하면, 1이 먼저 `s`의 제일위에 삽입되고, 다음에 2, 다음에 3이 제일 위에 삽입됩니다. 결국 `s` 는 3,2,1,5,5 을 가지게 됩니다.



다음 예제는 전방 삽입 반복자를 사용하는 예제 입니다.

```
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

int main()
{
    int x[] = { 1, 2, 3 };
    list<int> s(2, 5);
```



```
front_insert_iterator< list<int> > fins(s);
copy(x, x + 3, fins);
copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "));
}
```

## I 함수 버전

STL에는 전방 삽입 반복자를 편리하게 사용할 수 있도록 다음과 같은 함수 버전도 제공합니다.

```
template<typename Container>
front_insert_iterator<Container> front_inserter(Container& _Cont);
```

다음은 이전의 예제를 함수 버전의 삽입 반복자를 사용해서 만든 예제입니다.

```
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

int main()
{
    int x[] = { 1, 2, 3 };
    list<int> s(2, 5);

    copy(x, x + 3, front_inserter(s));
    copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "));
}
```

### 3. 후방 삽입 반복자

Container의 마지막에 요소를 추가하는 후방 삽입 반복자도 제공됩니다.

```
template <typename Container> class back_insert_iterator
```

또한, 후방 삽입 반복자의 함수 버전도 제공됩니다.

```
template <typename Container>  
back_insert_iterator<Container> back_inserter(Container& _Cont);
```

다음 예제는 후방 삽입반복자를 사용하는 예제 입니다.

```
#include <iostream>  
#include <algorithm>  
#include <list>  
#include <iterator>  
using namespace std;  
  
int main()  
{  
    int x[] = { 1, 2, 3 };  
    list<int> s(2, 5);  
  
    // 클래스 버전을 사용한 방식  
    back_insert_iterator< list<int> > bins(s);  
    copy(x, x + 3, bins);  
  
    // 함수 버전을 사용하는 방식  
    copy(x, x + 3, back_inserter(s));  
  
    copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "));  
}
```

## 4. 임의 삽입 반복자

Container의 시작과 끝이 아닌 중간의 임의의 구간에 요소를 삽입 하는 반복자도 제공됩니다.

```
template <typename Container> class insert_iterator;
```

템플릿 인자로써는 요소를 삽입할 Container의 type을 지정하고 생성자로 Container 객체와 삽입할 곳의 반복자를 전달합니다. 즉, `list<int>` 의 객체 `s` 의 2번째 위치에 새로운 요소를 삽입하려면

```
list<int>::iterator p = s.begin();
++p;
```

```
insert_iterator<list<int> > ins(s, p);
copy(x, x + 3, ins);
```

또한, 임의 삽입반복자의 함수 버전도 제공됩니다.

```
template<typename Container, typename Iterator>
insert_iterator<Container> inserter(Container& _Cont, Iterator _It);
```

다음 예제는 임의 삽입반복자를 사용하는 예제 입니다.

```
#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

int main()
{
    int x[] = { 1, 2, 3 };
    list<int> s(2, 5);

    list<int>::iterator p = s.begin();
    ++p;

    insert_iterator< list<int> > ins(s, p);

    copy(x, x + 3, ins);
```

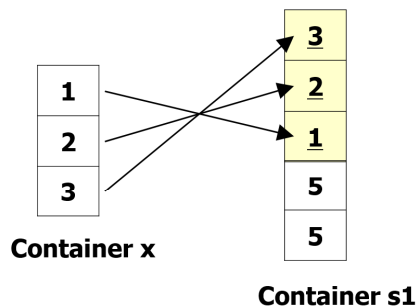
```

    copy(s.begin(), s.end(), ostream_iterator<int>(cout, " "));
}

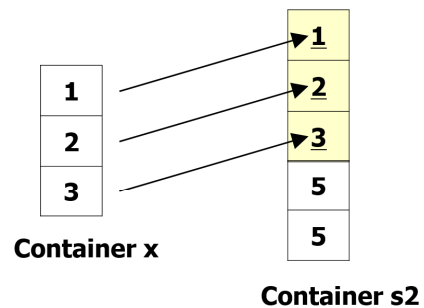
```

## 임의 삽입을 사용한 컨테이너의 앞쪽에 삽입.

임의 삽입 반복자를 사용해서도 Container의 제일 앞쪽에 요소를 삽입 할 수 있습니다. 하지만 이때는 전방 삽입과는 달리 원래 요소의 순서를 유지 하면서 삽입됩니다.



`front_inserter( s1 )` 를 사용한 경우



`inserter( s2, s2.begin() )` 를 사용한 경우

```

#include <iostream>
#include <algorithm>
#include <list>
#include <iterator>
using namespace std;

int main()
{
    int x[] = { 1, 2, 3 };
    list<int> s1(2, 5);
    list<int> s2(2, 5);

    copy(x, x + 3, front_inserter(s1));
    copy(x, x + 3, inserter(s2, s2.begin()));

    cout << "Container s1 : ";
    copy(s1.begin(), s1.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Container s2 : ";
    copy(s2.begin(), s2.end(), ostream_iterator<int>(cout, " "));
}

```

```
    cout << endl;  
}
```

항목 3-5

# stream iterator

## 학습 목표

- ostream\_iterator
- istream\_iterator
- ostreambuf\_iterator
- istreambuf\_iterator

# 1. stream iterator

## I 핵심 개념

입출력 스트림과 연결된 반복자.

- `ostream_iterator`
- `ostreambuf_iterator`
- `istream_iterator`
- `istreambuf_iterator`

stream iterator. defined in <iterator>

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

int main()
{
    ostream_iterator<int> p1(cout);
    istream_iterator<int> p2(cin);
    *p1 = *p2;
}
```

## I copy 알고리즘과 stream\_iterator

ostream\_iterator를 사용하면 반복자를 사용해서 출력을 할 수 있습니다

```
int main()
{
    int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
    ostream_iterator<int> p(cout, " ");

    copy(x, x + 10, p);
}
```

## I file 입출력과 stream\_iterator

스트림 반복자와 파일 객체를 사용하면 파일 입출력을 간단하게 할 수 있습니다. 다음 코드는 배열 x의 모든 요소를 "a.txt" 파일로 출력하는 예제 입니다.

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

int main()
{
    ofstream f("a.txt");
    ostream_iterator<int> p(f, " ");

    int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
    copy(x, x + 10, p);
}
```

또한, 입력 스트림 반복자를 사용하면 간단한 코드로 파일의 내용을 화면에 출력할 수 있습니다.

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



```
int main()
{
    ifstream f("stream5.cpp");
    istreambuf_iterator<char> p1(f), p2;
    ostream_iterator<char> p3(cout, "");

    copy(p1, p2, p3);
}
```

다음 예제는 파일 복사의 예제입니다.

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

int main()
{
    ifstream f1("stream6.cpp");
    ofstream f2("a.txt");

    istreambuf_iterator<char> p1(f1), p2;
    ostream_iterator<char> p3(f2, "");

    copy(p1, p2, p3);
}
```

항목 3-6

# iterator adapter

## 학습 목표

- reverse\_iterator
- const\_iterator
- move\_iterator

## 1. reverse\_iterator

### I 핵심 개념

- 기존 반복자의 역방향으로 동작하는 반복자
- 기존 알고리즘을 역방향으로 동작하게 할 수 있습니다.

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

int main()
{
    list<int> s = { 1,2,3,4,5,6,7,3,9,10};
    auto p1 = begin(s);
    auto p2 = end(s);

    reverse_iterator< list<int>::iterator> p3( p2 ); // p3와 p2는 다른 객체
    reverse_iterator< list<int>::iterator> p4( p1 );

    auto ret1 = find(p1, p2, 3); //
    auto ret2 = find(p3, p4, 3); //

    //++p3; // --p2 처럼 동작
    cout << *p3 << endl; // 10
    ++p3;
    cout << *p3 << endl; // 9
    ++p3;
    cout << *p3 << endl; // 3
    --p2;
    cout << *p2 << endl; // 10
}
```

## reverse\_iterator 만드는 방법.

- 기존 반복자를 가지고 생성
  - ✓ reverse\_iterator 템플릿을 직접 사용
  - ✓ make\_reverse\_iterator() 도움 함수 사용
- 컨테이너에서 직접 꺼내기 - rbegin()/rend() 멤버 함수 또는 일반 함수

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

int main()
{
    list<int> s = { 1,2,3,4,5,6,7,3,9,10};
    auto ret = find( rbegin(s), rend(s), 3);
    auto p1 = begin(s);
    auto p2 = end(s);

    // 방법 1.
    reverse_iterator< list<int>::iterator > r1(p2);

    // 방법 2.
    auto r2 = make_reverse_iterator(p2);

    // 방법 3.
    auto r3 = s.rbegin();
    auto r4 = rbegin(s);

    cout << *r1 << endl;
    cout << *r2 << endl;
    cout << *r3 << endl;
    cout << *r4 << endl;
}
```

## 2. Iterator의 종류

### I 컨테이너에서 꺼낼 수 있는 iterator의 4가지 종류

- iterator
- reverse\_iterator
- const\_iterator
- const\_reverse\_iterator

```
#include <iostream>
#include <list>
#include <forward_list>
#include <algorithm>
using namespace std;

int main()
{
    list<int> s = { 1,2,3,4,5,6,7,8,9,10};
    forward_list<int> s2 = { 1,2,3,4,5,6,7,8,9,10};

    //auto p5 = rbegin(s2);
    //auto p1 = begin(s);

    // list<int>::iterator p1 = begin(s);
    // list<int>::reverse_iterator p2 = rbegin(s);
    // list<int>::const_iterator p3 = cbegin(s);
    // *p3 = 10; // error
    // list<int>::const_reverse_iterator p4 = crbegin(s);

    auto p1 = begin(s);
    auto p2 = rbegin(s);
    auto p3 = cbegin(s);
    // *p3 = 10; // error

    auto p4 = crbegin(s);
}
```

### 3. move iterator – C++11

#### I 핵심 개념

- 컨테이너의 요소를 복사가 아닌 move 로 접근하는 반복자
- move iterator를 생성하는 방법.
  - ✓ move\_iterator <> 클래스 템플릿 사용
  - ✓ make\_move\_iterator() 함수 사용.

```
#include <iostream>
#include <iterator>
#include <vector>
#include "People.h"
using namespace std;

int main()
{
    vector<People> v;
    v.push_back(People("A"));
    v.push_back(People("B"));
    v.push_back(People("C"));
    v.push_back(People("D"));

    cout << "-----" << endl;

    //vector<People> v2(begin(v), end(v));
    vector<People> v2(make_move_iterator(begin(v)),
                     make_move_iterator(end(v)));
}
```

항목 3-7

## iterator 보조함수

### 학습 목표

- next()/prev()
- advance()/distance()
- iter\_swap()

## 1. iterator 보조 함수

### ▮ raw pointer와 증가 연산자

- Raw Pointer를 값으로 리턴 하는 경우 증가 연산자를 사용할 수 없습니다.
- next() 알고리즘을 사용하면 증가 연산의 수행 한 것과 동일한 효과를 볼 수 있습니다.

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

int* foo()
{
    static int x[10] = {1,2,3,4,5,6,7,8,9,10};
    return x;
}

int main()
{
    //auto p1 = ++foo();    // error
    auto p2 = next(foo()); // ok

    cout << *p2 << endl;
}
```



## ■ iterator 보조 함수

- <iterator> 헤더에는 반복자 관련 몇가지 보조 함수를 제공합니다.
- next/prev/advance/distance/iter\_swap

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

int main()
{
    int x[10] = { 1,2,3,4,5,6,7,8,9,10};

    forward_list<int> s = { 1,2,3,4,5,6,7,8,9,10};

    auto p1 = next(begin(s));

    advance(p1, 3); // p1 + 3;

    cout << *p1 << endl; // 5

    cout << distance( begin(s), p1) << endl; // p1 - begin(s)

    iter_swap(p1, begin(s));

    cout << *p1 << endl; // 1
}
```

Section 4.

# algorithm

항목 4-1

# algorithm 개념

## 학습 목표

- Algorithm 특징
- Algorithm 은 컨테이너를 알지 못한다.
- Algorithm 보다 멤버 함수가 좋은 경우가 있다.

## 1. algorithm 기본 개념

### ■ 멤버 함수가 아닌 일반 함수이다.

- STL 알고리즘은 특정한 컨테이너가 아닌 다양한 컨테이너에 대해서 사용할 수 있는 일반 함수로 되어 있습니다.
- STL 알고리즘을 사용하려면 `<algorithm>`, `<numeric>`, `<memory>` 등의 헤더가 필요합니다.

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

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

    auto p = find( begin(v), end(v), 3);

    return 0;
}
```

## ❑ 알고리즘은 컨테이너를 알지 못한다.

- 알고리즘은 인자로 전달된 반복자가 어떤 컨테이너의 반복자 인지 알 수 없습니다.
- `remove` 알고리즘은 컨테이너의 크기를 줄이지 않습니다.
- 컨테이너의 크기를 줄이려면 컨테이너 자체의 `erase()` 멤버 함수를 사용해야 합니다.

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include "ecourse_stl.hpp"
using namespace std;

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

    auto p = remove(begin(v), end(v), 3);

    show(v); // 1,2,1,2,1,2,1, 2,3,1

    v.erase(p, end(v));

    show(v); // 1,2,1,2,1,2,1

    return 0;
}
```

## ❑ 알고리즘 보다 멤버 함수가 좋은 경우가 있다.

- list 에서 요소를 제거할 때는 앞으로 당기는 것 보다는 메모리를 직접 제거하는 것이 효율적입니다.
- remove 알고리즘을 사용하는 것보다는 remove 멤버 함수를 사용하는 것이 좋습니다.

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include "ecourse_stl.hpp"
using namespace std;

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

    list<int> v = { 1,2,3,1,2,3,1,2,3,1 };

    v.remove(3); // 당기는 것이 아니라 메모리 제거

    show(v); // 1,2,1,2,1,2,1,    2,3,1
    /*
    // 아래 처럼 해도 되지만 성능이 좋지 않습니다.
    auto p = remove(begin(v), end(v), 3);

    show(v); // 1,2,1,2,1,2,1, 2,3,1

    v.erase(p, end(v));

    show(v); // 1,2,1,2,1,2,1
    */
    return 0;
}
```

## 항목 4-2

# algorithm 과 함수

### 학습 목표

- Algorithm 과 function
- Functor
- Predicate
- `std::bind`

## 2. algorithm & function

### Ⅰ 단항 함수와 이항 함수

STL의 알고리즘 중에는 함수는 인자로 받는 경우가 많이 있습니다. 이때, 전달되는 함수는 아래의 2가지로 구분 됩니다.

단항 함수	인자가 1개인 함수
이항 함수	인자가 2개인 함수

또한, 알고리즘에 전달되는 함수는 일반 함수 뿐 아니라 ()를 사용해서 호출가능한 모든 callable object(함수, 함수객체, 람다표현식등)를 사용할 수 있습니다.

### Ⅰ for\_each

for\_each 알고리즘은 첫번째와 두번째 인자로 구간을, 그리고 3번째 인자로 단항 함수를 전달 받습니다. 주어진 구간에 있는 모든 요소를 차례대로 3번째 인자로 전달된 단항 함수로 전달합니다.

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

void show(int n)
{
    cout << n << endl;
}

int main()
{
    int x[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    for_each(x, x + 10, show); // using function
    for_each(x, x + 10, [](int a) { cout << a << endl; }); // using lambda expression
}
```



## I transform

transform 알고리즘은 2가지의 버전이 있습니다.

첫번째 버전은 다음 처럼 4개의 인자를 가지고 있습니다.

```
template<class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(InputIterator _First1, InputIterator _Last1,
                        OutputIterator _Result, UnaryFunction _Func);
```

첫번째 파라미터와 2번째 파라미터로 전달된 구간의 각 요소를 마지막 인자로 전달된 **단항함수**로 전달합니다. 그리고 단항 함수의 리턴 값으로 3번째 인자로 전달된 구간에 넣습니다.

아래 예제는 x배열의 각 요소의 절대값을 y배열에 복사하는 프로그램입니다.

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

int main()
{
    int x[] = { 1, -2, 3, -4, 5, -6, 7, -8, 9, -10 };
    int y[10];

    transform(x, x + 10, y, [](int a) { return abs(a); });

    for_each(y, y + 10, [](int a) { cout << a << endl; });
}
```

transform 알고리즘의 2번째 버전은 아래와 같이 5개의 인자를 가지고 있습니다.

```
template<class InputIterator1, class InputIterator2,
        class OutputIterator, class BinaryFunction>
OutputIterator transform(InputIterator1 _First1, InputIterator1 _Last1,
                        InputIterator2 _First2, OutputIterator _Result,
                        BinaryFunction _Func);
```

2개의 구간에 있는 각 요소를 마지막 인자로 전달된 **이항 함수**로 전달합니다. 그리고 이항 함수의 리턴 값으로 4번째 인자로 전달된 구간에 쓰게 됩니다.

아래 예제는 x, y 의 2개 배열에 있는 요소의 절대값의 합을 z배열에 넣는 프로그램 입니다.

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

int main()
{
    int x[5] = { 1, 2, 3, 4, 5 };
    int y[5] = { -1, -2, -3, -4, -5 };
    int z[5];

    transform(x, x + 5, y, z, [](int a, int b) { return abs(a) + abs(b); });

    for_each(y, y + 10, [](int a) { cout << a << endl; });
}
```

### 3. functor

#### I <functional> 헤더

STL에는 몇 가지의 미리 정의된 함수 객체를 제공합니다. STL의 함수객체를 사용하려면 <functional> 헤더를 포함해야 합니다.

산술연산	비교연산	논리연산
<code>plus&lt;T&gt;</code> <code>minus&lt;T&gt;</code> <code>multiplies&lt;T&gt;</code> <code>divides&lt;T&gt;</code> <code>modulus&lt;T&gt;</code> <code>negate&lt;T&gt;</code>	<code>equal_to&lt;T&gt;</code> <code>not_equal_to&lt;T&gt;</code> <code>greater&lt;T&gt;</code> <code>less&lt;T&gt;</code> <code>greater_equal&lt;T&gt;</code> <code>less_equal&lt;T&gt;</code>	<code>logical_and&lt;T&gt;</code> <code>logical_or&lt;T&gt;</code> <code>logical_not&lt;T&gt;</code>

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

void main()
{
    int x[5] = { 1, 2, 3, 4, 5 };
    int y[5] = { 6, 7, 8, 9, 10 };
    int s1[5], s2[5];
    transform(x, x + 5, y, s1, plus<int>());
    transform(x, x + 5, y, s2, minus<int>());

    for (auto n : s1)
        cout << n << endl;
    cout << endl;

    for (auto n : s2)
        cout << n << endl;
    cout << endl;
}
```

## 4. Predicate

### ■ remove vs remove\_if

참 또는 거짓을 리턴 하는 함수(또는 함수객체)를 조건자(Predicate)라고 합니다. STL의 알고리즘 중에서 \_if로 끝나는 알고리즘은 조건자를 인자로 가지고 있습니다.

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

void main()
{
    int x[10] = { 1, 2, 3, 4, 3, 8, 3, 2, 5, 7 };
    int y[10] = { 1, 2, 3, 4, 3, 8, 3, 2, 5, 7 };

    // 상수 제거 버전
    int* px = remove(x, x + 10, 3); // remove 3

    // 조건자 버전
    int* py = remove_if(y, y + 10, [](int a) { return a % 2 == 0; });

    copy(x, px, ostream_iterator<int>(cout, " "));
    cout << endl;

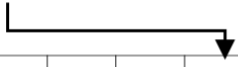
    copy(y, py, ostream_iterator<int>(cout, " "));
    cout << endl;
};
```

### ■ remove 알고리즘은 컨테이너를 줄이지 않는다.

위의 코드에서 y배열에 있는 홀수는 제거가 되고 홀수가 아닌 요소가 앞으로 당겨집니다. 배열의 뒷부분은 이전에 들어 있던 요소의 값이 그대로 들어 있게 됩니다. 이때 리턴 값은 remove 연산의 결과로 유효한 요소의 다음을 가리키는(past the end) iterator(이 경우 포인터)가 리턴 됩니다. 즉,

1	2	3	4	3	8	3	2	5	7
---	---	---	---	---	---	---	---	---	---

```
int* py = remove_if( y, y+10, IsOdd() );
```



<b><u>2</u></b>	<b><u>4</u></b>	<b><u>8</u></b>	<b><u>2</u></b>	3	8	3	2	5	7
-----------------	-----------------	-----------------	-----------------	---	---	---	---	---	---

## 5. std::bind 를 사용한 인자 고정

### I std::bind

std::bind()를 사용하면 함수의 M 항 함수를 N항 함수로 고정할 수 있습니다. modulus 함수 객체는 인자가 2개인 이항 함수 객체 입니다. 하지만 remove\_if 알고리즘은 인자로 단항 함수를 요구합니다. std::bind를 사용하면 이항 함수객체를 단항 함수 객체로 변경할 수 있습니다.

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <functional>
using namespace std;
using namespace std::placeholders;

void main()
{
    int x[10] = { 1, 2, 3, 4, 3, 8, 3, 2, 5, 7 };

    // 주어진 구간에서 홀수를 제거하는 코드.
    int* p = remove_if(x, x + 10, bind(modulus<int>(), _1, 2));

    copy(x, p, ostream_iterator<int>(cout, " "));
};
```

std::bind()에 대해서는 뒤 장의 “function & bind” 항목에서 좀더 자세히 다루게 됩니다.

항목 4-3

## algorithm 의 변형

### 학습 목표

- inplace vs copy
- constant vs predicator

## 1. algorithm 의 변형

■ 하나의 알고리즘의 4가지 형태로 존재할 수 있습니다.

하나의 알고리즘 함수는 4가지

in place version	각 요소에 대한 연산의 결과를 컨테이너 자신에 적용하는 방식.	remove
copy version	각 요소에 대한 연산의 결과를 다른 컨테이너에 넣는 방식.	remove_copy
predicate version	조건자를 사용하는 버전	remove_if
cop & predicate version	조건자, 복사 버전	remove_copy_if

■ inplace vs copy

일부 알고리즘은 In-place 버전과 Copy 버전이 모두 존재 합니다. In-Place 버전은 알고리즘의 출력 결과로 얻어지는 Sequence 가 수행 대상이 되는 Container 내부에 놓이게 됩니다.

반면 Copy 버전은 알고리즘의 출력 결과로 얻어지는 Sequence가 다른 Contrainer에 놓이게 됩니다. ( 어떤 경우는 동일 Container의 다른 구간에 넣기도 합니다.)

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

void main()
{
    int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
    int y[10] = { 1,2,3,4,5,6,7,8,9,10 };
    int z[10] = { 0 };

    // inplace version
    int* p = remove(x, x + 10, 5);
    copy(x, p, ostream_iterator<int>(cout, " "));
    cout << endl;
```



```
// copy version
int* p2 = remove_copy(y, y + 10, z, 5);

copy(z, p2, ostream_iterator<int>(cout, " "));
}
```

하지만 sort 의 복사 버전인 sort\_copy()는 STL에 존재 하지 않습니다. sort 에 사용되는 비용이 copy 보다 훨씬 커서 copy를 먼저 수행하고 sort를 호출하는 것이 더 바람직하기 때문입니다. 하지만 복사(copy)와 제거(remove)는 한번에 하면 copy를 먼저 한 후 remove를 수행한 것보다 두 배 가까이 비용을 절약 할 수 있기 때문에 별도의 복사 버전(remove\_copy)를 제공합니다.

## ■ 조건자(predicator) 버전

일부 알고리즘 중에는 함수를 인자로 가지는 경우가 있습니다. 대부분 조건자 함수를 가지는 경우입니다. 함수 또는 함수객체, 람다 표현식등을 조건자로 전달할 수 있습니다. 알고리즘의 조건자 버전은 주로 \_if 로 끝나게 됩니다.

그런데, sort 같은 알고리즘은 뒤에 if가 붙지 않는데, 그 이유는 조건자 버전은 인자가 3개이고 조건자를 사용하지 않은 버전은 인자가 2개 이기 때문에 조건자 버전과 그렇지 않은 버전을 명확히 구분할수 있으므로 \_if 가 붙지 않습니다.

## Appendix

# Algorithm Catalog

### 📖 학습 목표

- 수정 불가 시퀀스 알고리즘
- 수정 가능 시퀀스 알고리즘
- 분할 알고리즘
- 정렬 관련 알고리즘
- 범용 수치 알고리즘

## 수정 불가능 시퀀스 알고리즘 ( non-modifying sequence algorithm )

defined in <algorithm>	
all_of (C++11) any_of (C++11) none_of(C++11)	checks if a predicate is true for all, any or none of the elements in a range
for_each	applies a function to a range of elements
for_each_n (C++17)	applies a function object to the first n elements of a sequence
count count_if	returns the number of elements satisfying specific criteria
mismatch	finds the first position where two ranges differ
equal	determines if two sets of elements are the same
find find_if find_if_not (C++11)	finds the first element satisfying specific criteria
find_end	finds the last sequence of elements in a certain range
find_first_of	searches for any one of a set of elements
adjacent_find	finds the first two adjacent items that are equal (or satisfy a given predicate)
search	searches for a range of elements
search_n	searches for a number consecutive copies of an element in a range

## 수정 가능 시퀀스 알고리즘 ( modifying sequence algorithm )

defined in <algorithm>	
copy copy_if (C++11)	copies a range of elements to a new location
copy_n (C++11)	copies a number of elements to a new location
copy_backward	copies a range of elements in backwards order
move (C++11)	moves a range of elements to a new location
move_backward (C++11)	moves a range of elements to a new location in backwards order
fill	copy - assigns the given value to every element in a range
fill_n	copy - assigns the given value to N elements in a range
transform	applies a function to a range of elements

generate	assigns the results of successive function calls to every element in a range
generate_n	assigns the results of successive function calls to N elements in a range
remove remove_if	removes elements satisfying specific criteria
remove_copy remove_copy_if	copies a range of elements omitting those that satisfy specific criteria
replace replace_if	replaces all values satisfying specific criteria with another value
replace_copy replace_copy_if	copies a range, replacing elements satisfying specific criteria with another value
swap	swaps the values of two objects
swap_ranges	swaps two ranges of elements
iter_swap	swaps the elements pointed to by two iterators
reverse	reverses the order of elements in a range
reverse_copy	creates a copy of a range that is reversed
rotate	rotates the order of elements in a range
rotate_copy	copies and rotate a range of elements
shuffle	randomly re - orders elements in a range
unique	removes consecutive duplicate elements in a range
unique_copy	creates a copy of some range of elements that contains no consecutive duplicates
sample (C++17)	selects n random elements from a sequence

## Ⅰ 분할 알고리즘 ( partitioning operations algorithm )

defined in <algorithm>	
is_partitioned(C++11)	determines if the range is partitioned by the given predicate
partition	copies a number of elements to a new location
partition_copy(C++11)	copies a range dividing the elements into two groups
stable_partition	divides elements into two groups while preserving their relative order

<code>partition_point(C++11)</code>	locates the partition point of a partitioned range
-------------------------------------	--

## 정렬 알고리즘 ( sorting operations algorithm )

defined in <algorithm>	
<code>is_sorted (C++11)</code>	checks whether a range is sorted into ascending order
<code>is_sorted_until(C++11)</code>	finds the largest sorted subrange
<code>sort</code>	sorts a range into ascending order
<code>partial_sort</code>	sorts the first N elements of a range
<code>partial_sort_copy</code>	copies and partially sorts a range of elements
<code>stable_sort</code>	sorts a range of elements <b>while</b> preserving order between equal elements
<code>nth_element</code>	partially sorts the given range making sure that it is partitioned by the given element

## 이진 검색 알고리즘 ( binary search operations algorithm on sorted ranges)

defined in <algorithm>	
<code>lower_bound</code>	returns an iterator to the first element not less than the given value
<code>upper_bound</code>	returns an iterator to the first element greater than a certain value
<code>binary_search</code>	determines <b>if</b> an element exists in a certain range
<code>equal_range</code>	returns range of elements matching a specific key

## 집합 연산 알고리즘 ( set operations algorithm on sorted ranges)

defined in <algorithm>	
<code>merge</code>	merges two sorted ranges
<code>inplace_merge</code>	merges two ordered ranges in - place
<code>includes</code>	returns <b>true</b> <b>if</b> one set is a subset of another

set_difference	computes the difference between two sets
set_intersection	computes the intersection of two sets
set_symmetric_difference	computes the symmetric difference between two sets
set_union	computes the <a href="#">union</a> of two sets

## ■ 힙 연산 알고리즘 ( heap operations algorithm )

defined in <algorithm>	
is_heap (C++11)	checks <a href="#">if</a> the given range is a max heap
is_heap_until (C++11)	finds the largest subrange that is a max heap
make_heap	creates a max heap out of a range of elements
push_heap	adds an element to a max heap
pop_heap	removes the largest element from a max heap
sort_heap	turns a max heap into a range of elements sorted in ascending order

## ■ 최대/최소 연산 알고리즘 ( maximum/minimum operations algorithm )

defined in <algorithm>	
max	returns the greater of the given values
max_element	returns the largest element in a range
min	returns the smaller of the given values
min_element	returns the smallest element in a range
minmax (C++11)	returns the smaller and larger of two elements
minmax_element (C++11)	returns the smallest and the largest elements in a range
clamp (C++17)	clamps a value between a pair of boundary values
lexicographical_compare	returns <a href="#">true if</a> one range is lexicographically less than another
is_permutation (C++11)	determines <a href="#">if</a> a sequence is a permutation of another sequence

next_permutation	generates the next greater lexicographic permutation of a range of elements
prev_permutation	generates the next smaller lexicographic permutation of a range of elements

## 범용 수치 알고리즘 ( numeric operations algorithm )

defined in <numeric>	
iota (C++11)	fills a range with successive increments of the starting value
accumulate	sums up a range of elements
inner_product	computes the inner product of two ranges of elements
adjacent_difference	computes the differences between adjacent elements in a range
partial_sum	computes the partial sum of a range of elements
reduce (C++17)	similar to std::accumulate, except out of order
exclusive_scan (C++17)	similar to std::partial_sum, excludes the ith input element from the ith sum
inclusive_scan (C++17)	similar to std::partial_sum, includes the ith input element in the ith sum
transform_reduce (C++17)	applies a functor, then reduces out of order
transform_exclusive_scan(C++17)	applies a functor, then calculates exclusive scan
transform_inclusive_scan(C++17)	applies a functor, then calculates inclusive scan

## 메모리 연산 알고리즘 ( operations on uninitialized memory )

defined in <memory>	
uninitialized_copy	copies a range of objects to an uninitialized area of memory
uninitialized_copy_n (C++11)	copies a number of objects to an uninitialized area of memory
uninitialized_fill	copies an object to an uninitialized area of memory, defined by a range
uninitialized_fill_n	copies an object to an uninitialized area of memory, defined by a start and a count
uninitialized_move (C++17)	moves a range of objects to an uninitialized area of memory
uninitialized_move_n (C++17)	moves a number of objects to an uninitialized area of memory

<code>uninitialized_default_construct</code> (C++17)	constructs objects by <a href="#">default</a> - initialization in an uninitialized area of memory, defined by a range
<code>uninitialized_default_construct_n</code> (C++17)	constructs objects by <a href="#">default</a> - initialization in an uninitialized area of memory, defined by a start and a count
<code>uninitialized_value_construct</code> (C++17)	constructs objects by value - initialization in an uninitialized area of memory, defined by a range
<code>uninitialized_value_construct_n</code> (C++17)	constructs objects by value - initialization in an uninitialized area of memory, defined by a start and a count
<code>destroy_at</code> (C++17)	destroys an object at a given address
<code>destroy</code> (C++17)	destroys a range of objects
<code>destroy_n</code> (C++17)	destroys a number of objects in a range



## Section 5.

# Container

항목 5-1

# Container basic

## 학습 목표

- Container 종류
- Member Type
- Policy Base Design
- Allocator

# 1. Container 의 종류

## I 핵심 개념

STL 에는 다음과 같은 컨테이너를 제공하고 있습니다.

categories	name		description
sequence containers	array	C++11	static contiguous array
	vector		dynamic contiguous array
	deque		double-ended queue
	forward_list	C++11	singly-linked list
	list		doubly-linked list
associative containers	set		collection of unique keys, sorted by keys
	map		collection of key-value pairs, sorted by keys, keys are unique
	multi_set		collection of keys, sorted by keys
	multi_map		collection of key-value pairs, sorted by keys
unordered associative containers	unordered_set	C++11	collection of unique keys, hashed by keys
	unordered_map	C++11	collection of key-value pairs, hashed by keys, keys are unique
	unordered_multi_set	C++11	collection of keys, hashed by keys
	unordered_multi_map	C++11	collection of key-value pairs, hashed by keys
container adaptors	stack		adapts a container to provide stack (LIFO data structure)
	queue		adapts a container to provide queue (FIFO data structure)
	priority_queue		adapts a container to provide priority queue

## 2. Container 의 특징

### Container의 공통적인 특징

- ① 모든 컨테이너는 “레퍼런스” 보다는 “값”을 보관합니다. 즉, 모든 삽입은 내부적으로 복사본을 생성합니다.
- ② 일반적으로 모든 원소들은 순서를 가지고 있습니다. 원소를 순회할 수 있는 반복자가 있습니다.
- ③ STL 자체는 예외를 발생시키지 않습니다.

### 리턴과 삭제를 동시에 하지 않는다.

STL의 대부분의 컨테이너는 요소를 리턴하는 함수와 요소를 삭제 하는 함수 가 분리되어 있습니다. 이런 설계를 한 이유는

- 참조 리턴을 통해서 임시객체를 막고
- 예외 안전성의 강력 보장을 지원하기 위해서

입니다.

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

int main()
{
    stack<int> st;
    st.push(10);
    st.push(20);
    st.push(30);
    cout << st.top() << endl; // 30. 리턴만하고 제거하지 않습니다.
    cout << st.top() << endl; // 30.

    st.pop(); // 제거만 하고 리턴하지 않습니다.
    cout << st.top() << endl; // 20
}
```

## I Member Type

STL의 각 컨테이너에는 다음과 같은 타입이 정의 되어 있습니다.

member type	definition
value_type	T
size_type	std::size_t
difference_type	std::ptrdiff_t
reference	value_type&
const_reference	const value_type&
pointer	value_type*
const_pointer	const value_type*
iterator	
const_iterator	
reverse_iterator	std::reverse_iterator<iterator>
const_reverse_iterator	std::reverse_iterator<const_iterator>

## I 단위 전략 디자인 ( policy base design)

STL 의 다양한 컨테이너는 다양한 연산에 대해서 성능저하 없이 정책을 변경할 수 있도록 설계 되었습니다. 사용자는 함수객체를 템플릿 인자로 전달해서 컨테이너가 사용하는 다양한 정책을 변경할 수 있습니다.

part4\container\container2.cpp

```
// string 클래스의 비교 전략을 담은 클래스를 설계 합니다.
struct my_traits : char_traits<char>
{
    static bool lt( char a, char b ) { return toupper(a) < toupper(b); }
    static bool gt( char a, char b ) { return toupper(a) > toupper(b); }

    static int compare( const char* a, const char* b, int sz)
    {
        return strcmpi( a, b );
    }
}
```

```
};
typedef basic_string<char, my_traits> my_string;

int main()
{
    my_string s1 = "abcd";
    my_string s2 = "ABCD";

    if ( s1 == s2 )
        cout << "same" << endl;
    else
        cout << "not same" << endl;
}
```

## Ⅰ 메모리 할당기 ( Allocator )

STL 의 컨테이너는 내부적으로 메모리 할당을 위해 단위전략 디자인 패턴을 사용한 메모리 할당기를 사용하고 있습니다. 컨테이너 사용시 메모리 할당 방식을 변경하려면 “사용자 정의 메모리 할당기(allocator)”를 컨테이너에 template 인자로 전달하면 됩니다.

다음 예제는 vector의 메모리 할당 정책을 변경하기 위해, 사용자 정의 allocator를 사용하는 코드입니다.

part4\container\container3.cpp

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

// user define allocator
template <class T> class MyAlloc
{
public:
    typedef T          value_type;
    typedef T*         pointer;
    typedef const T*   const_pointer;
    typedef T&         reference;
    typedef const T&   const_reference;
```

```

typedef std::size_t    size_type;
typedef std::ptrdiff_t difference_type;

template <class U> struct rebind
{
    typedef MyAlloc<U> other;
};

pointer address (reference value) const
{
    return &value;
}

const_pointer address (const_reference value) const
{
    return &value;
}

MyAlloc() throw() { }
MyAlloc(const MyAlloc&) throw() { }
~MyAlloc() throw() { }
template <class U> MyAlloc (const MyAlloc<U>&) throw() {}

size_type max_size () const throw()
{
    return std::numeric_limits<std::size_t>::max() / sizeof(T);
}

pointer allocate (size_type num, const void* = 0)
{
    std::cerr << "allocate " << num << " element(s)"
                << " of size " << sizeof(T) << std::endl;
    pointer ret = (pointer)(::operator new(num*sizeof(T)));
    std::cerr << " allocated at: " << (void*)ret << std::endl;
    return ret;
}

void construct (pointer p, const T& value)
{
    new((void*)p)T(value);
}

void destroy (pointer p)
{
    p->~T();
}

void deallocate (pointer p, size_type num)
{
    std::cerr << "deallocate " << num << " element(s)"

```

```
        << " of size " << sizeof(T)
        << " at: " << (void*)p << std::endl;
        ::operator delete((void*)p);
    }
};

template <class T1, class T2>
bool operator== (const MyAlloc<T1>&, const MyAlloc<T2>&) throw() {
    return true;
}

template <class T1, class T2>
bool operator!= (const MyAlloc<T1>&, const MyAlloc<T2>&) throw() {
    return false;
}

int main()
{
    vector<int, MyAlloc<int> > v;
    v.push_back( 10 );
}
```



항목 5-2

# Sequence container

## 학습 목표

- Sequence Container 의 공통된 특징
- array, vector, deque
- forward\_list, list

## 항목 13

# Sequence Containers

### 배우게 되는 내용

- ① sequence container의 공통된 특징
- ② array, vector, deque
- ③ forward\_list, list

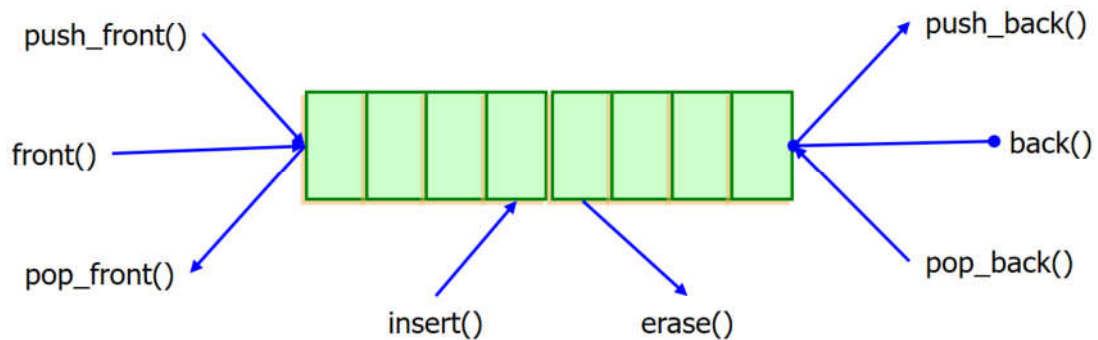
# 1. sequence container

## ■ 핵심 개념

각 요소를 순차적으로 접근 할 수 있는 할 수 Container.

sequence containers	array	C++11	static contiguous array
	vector		dynamic contiguous array
	deque		double-ended queue
	forward_list	C++11	singly-linked list
	list		doubly-linked list

## ■ sequence container의 요소 접근



	전방 삽입/삭제	임의 삽입/삭제	후방 삽입/삭제
array	X	X	X
forward_list	0	0	X
list	0	0	0
deque	0	0	0
vector	X	0	0

## I template argument

각 sequence container의 template argument는 다음과 같습니다.

```
template<typename T, typename Allocator = allocator<T> > class vector;  
template<typename T, typename Allocator = allocator<T> > class deque;  
template<typename T, typename Allocator = allocator<T> > class list;  
template<typename T, typename Allocator = allocator<T> > class forward_list;  
template<typename T, size_t N> struct array;
```

## I 컨테이너별 반복자 카테고리

array	random access iterator
forward_list	forward iterator
list	bidirectional iterator
deque	random access iterator
vector	random access iterator

## I container 생성

sequence container를 생성할 때는 괄호()를 사용하는 경우와 중괄호{}를 사용하는 경우를 주의해야 합니다.

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

int main()
{
    vector<int> v1(10, 3); // 10개의 요소를 3으로 초기화 합니다.
    vector<int> v2{ 10,3 }; // 2개의 요소를 10, 3으로 초기화 합니다.
    cout << v1.size() << endl;
    cout << v1.capacity() << endl;
    cout << v2.size() << endl;
    cout << v2.capacity() << endl;
}
```

## I std::array

std::array 는 C의 배열과 동일한 메모리 구조를 사용합니다. C의 배열과 동일 성능을 가지고 몇가지 멤버함수가 제공되므로 C의 배열보다 편리하게 사용할 수 있습니다.

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

int main()
{
    array<int, 10> ar{ 1,2,3,4,5,6,7,8,9,10 };

    ar[0] = 10;
    ar.at(1) = 10;

    for (auto n : ar)
        cout << n << endl;
}
```

## I std::vector

std::array 는 stack 에 메모리를 가지고 있지만 std::vector는 동적으로 할당된 배열을 사용해서 메모리를 관리합니다. capacity()개념을 가지고 요소를 관리하게 됩니다.

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct FindCharacter
{
    string data;
public:
    FindCharacter( string s ) : data(s) {}

    inline bool operator()( char c )
    {
        return find( data.begin(), data.end(), c ) != data.end();
    }
};

int main()
{
    vector<string> v;

    ifstream f("컨테이너3.cpp");
    string s;
    while( getline(f, s) )
        v.push_back(s);

    FindCharacter fc("0123456789");

    for ( int i =0; i < v.size(); i++ )
    {
        replace_if( v[i].begin(), v[i].end(), fc, ' ');
    }
    for ( int i = 0; i < v.size(); i++)
        cout << v[i] << endl;
}
```

## I std::forward\_list

std::forward\_list 는 싱글 링크드 리스트를 구현한 컨테이너 입니다.

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

int main()
{
    array<int, 10> ar{ 1,2,3,4,5,6,7,8,9,10 };

    ar[0] = 10;
    ar.at(1) = 10;

    for (auto n : ar)
        cout << n << endl;
}
```

항목 5-3

# Container adapter

## 학습 목표

- Container adapter 개념
- Stack, queue, priority\_queue



## 1. Container adapter 개념

### I 핵심 개념

container adapter는 sequence 컨테이너의 인터페이스를 변경해서 다른 자료구조처럼 동작하게 만든 것으로, adapter 패턴을 Generic 버전으로 구현한 클래스 템플릿입니다.

“stack, queue, priority\_queue” 등 3가지 adapter가 제공됩니다.

### I 기본 코드

```
template<typename T, typename C = deque<T> > class stack
{
    C st;
public:
    void push(const T& a) { st.push_back(a); }
    void pop() { st.pop_back(); }
    T& top() { return st.back(); }
    int size() const { return st.size(); }
    bool empty() const { return size() == 0; }
};
```

## sequence container 와 container adapter

stack의 경우는 한쪽으로만 입출력이 발생하므로 내부 자료 구조로서, vector, deque, list 등을 모두 사용할 수 있습니다. 하지만, queue 는 양쪽으로 입출력이 발생하므로 deque, list는 사용 할 수 있지만, vector를 사용할 수는 없습니다.

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

int main()
{
    stack<int, vector<int>> s; // ok
    queue<int, vector<int>> q; // 문제가 됩니다.
    q.push(10);
    q.pop();    // error
}
```

## priority\_queue의 정책 변경

priority\_queue는 값을 꺼낼 때 우선 순위가 가장 높은 요소가 나오게 됩니다. 이때 함수 객체를 사용하면 우선 순위를 결정하는 정책을 변경할 수 있습니다.

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

int main()
{
    priority_queue<int, vector<int>, greater<int>> pq;
    pq.push(10);
    pq.push(20);
    pq.push(15);
    cout << pq.top() << endl; // 10
}
```

## I allocator 변경

container adapter는 자체적으로는 메모리를 관리 하지 않습니다. 따라서, allocator등을 변경하려면 container adapter의 템플릿 인자가 아닌, 내부 자료구조를 나타내는 container의 allocator를 변경해야 합니다.

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

int main()
{
    stack<int, vector<int, MyAlloc<int>>> st;
}
```

항목 5-4

## associative container

### 학습 목표

- set, multiset
- map, multimap
- unordered\_set
- unordered\_map

## 1. associative container

### ■ 연관 컨테이너

연관 컨테이너는 키에 기반해서 요소(값)의 효율적인 조회를 지원하는 가변 크기 컨테이너입니다.

연관 컨테이너도 요소들의 삽입과 삭제를 지원하나, 순차열과는 달리 특정 위치에서의 삽입, 삭제를 위한 메커니즘은 제공하지 않습니다. 그런 메커니즘을 지원하지 않는 이유는, 일반적으로 연관 컨테이너 안에서 요소들이 배치되는 방식이 클래스의 불변식 중 하나이기 때문입니다. 예를 들어 정렬된 연관 컨테이너(tree)의 요소들은 항상 오름차순으로 저장되며, 해시된 연관 컨테이너의 요소들은 항상 해시 함수에 의거해서 저장됩니다.

STL에는 4가지 종류의 연관 컨테이너를 제공합니다.

set, multi_set	균형 잡힌 tree
map, multi_map	pair를 저장하는 set
unordered_set, unordered_multiset	hash table
unordered_map, unordered_multimap	pair를 저장하는 hash

## 2. set, multiset

### I set

set은 균형 잡힌 tree( RB Tree 또는 AVL Tree)로 구현된 컨테이너 입니다. set에 요소를 삽입할때는 insert() 함수를 사용합니다.

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

int main()
{
    set<int> s;

    s.insert(10);
    s.insert(30);
    s.insert(15);
    s.insert(20);

    for (auto n : s)
        cout << n << " ";
    cout << endl;
}
```

### I set, multiset

set은 요소의 중복을 허용하지 않지만, multiset는 중복을 허용합니다. 또한, set 의 insert() 함수는 pair를 리턴하는데, pair의 second값을 통해서 성공/실패를 조사할 수 있습니다.

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

int main()
{
    set<int> s;
```

```

s.insert(10);

pair<set<int>::iterator, bool> ret = s.insert(10);

if (ret.second == false )
    cout << "fail, insert set" << endl;
}

```

## I set과 단위 전략

set은 요소가 삽입 될 때 less 객체를 사용해서 요소의 크기를 비교합니다. 단위전략을 사용해서 비교 정책을 교체 할 수 있습니다.

```

#include <iostream>
#include <set>
#include <functional>
using namespace std;

int main()
{
    set<int, greater<int>> s;

    s.insert(10);
    s.insert(30);
    s.insert(15);
    s.insert(20);

    for (auto n : s)
        cout << n << " ";
    cout << endl;
}

```

## I set과 algorithm

set은 이진 tree로 구현되어 있기 때문에 요소를 검색 할때는 선형 검색 보다는 이진 검색을 사용하는 것이 좋습니다.

```

#include <iostream>

```

```
#include <set>
#include <functional>
using namespace std;

int main()
{
    set<int> s{ 10, 30, 15, 20 };

    auto p1 = find(s.begin(), s.end(), 15); // 에러는 없지만 선형검색입니다.
    auto p2 = s.find(15); // 이진 검색 입니다.

    cout << *p1 << endl;
    cout << *p2 << endl;
}
```



### 3. map, multimap

#### I map

map을 사용하면 키 값을 가지고 data를 저장할 수 있습니다. map은 pair를 저장하는 set입니다. map에 요소를 넣을 때는 pair를 생성해 넣거나, [] 연산자를 사용할 수도 있습니다.

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

int main()
{
    map<string, string> m;

    // map 에 data 넣기1.
    pair<string, string> p("mon", "월요일");
    m.insert( p );

    // 방법 2.
    m.insert( make_pair("tue", "화요일"));

    // 방법 3. [] 연산자
    m["wed"] = "수요일";

    cout << m["wed"] << endl; // "수요일"
    cout << m["sun"] << endl; // sun을 만들고 data는 디폴트 생성자로 초기화
}
```

#### I map 과 반복자

map의 반복자는 pair를 가리키게 됩니다. first로 키값을 second로 data에 접근할 수 있습니다.

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

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

int main()
{
    map<string, string> m{ { "mon", "월요일" }, { "tue", "화요일" }, { "wed", "수요일" } };

    auto ret = m.find("tue");

    cout << ret->first << endl;
    cout << ret->second << endl;
}
```

## I map example

set은 요소가 삽입 될 때 less 객체를 사용해서 요소의 크기를 비교합니다. 단위전략을 사용해서 비교 정책을 교체 할 수 있습니다.

```
#include <iostream>
#include <list>
#include <string>
#include <map>
#include <sstream>
#include <fstream>
#include <algorithm>
using namespace std;

int main()
{
    typedef map<string, list<int>> MAP;

    MAP m;
    ifstream f("readme.txt");
    int no = 0;
    string s;

    while( getline( f, s ) )
    {
        ++no;
        istringstream iss( s );
        string word;
```

```
    while( iss >> word )
    {
        m[word].push_back(no);
    }
}

MAP::iterator p = m.begin();

while ( p != m.end() )
{
    cout << p->first << " : ";
    for (auto n : p->second)
        cout << n << ", ";
    cout << endl;
    ++p;
}
}
```

## 4. unordered\_set, unordered\_map

### ■ unordered\_set, unordered\_map

unordered\_xxx 컨테이너는 hash 기반의 컨테이너입니다. 사용법은 std::set, std::map 과 유사하지만 hash 가 가지는 특징인 hash function 등을 단위 전략으로 지정할수 있습니다.

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

int main()
{
    unordered_set<int> s;

    s.insert(10);
    s.insert(20);

    unordered_set<int>::iterator p = s.find(20);

    cout << *p << endl;
}
```

### ■ hash function

STL에는 표준 타입들에 대한 hash 함수를 함수 객체 형태로 제공합니다.

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

int main()
{
    hash<int> hi;
    hash<double> hd;
    hash<string> hs;
```

```

cout << hi(100) << endl;
cout << hd(3.14) << endl;
cout << hs("hello"s) << endl;
}

```

## Ⅰ 사용자 정의 타입과 hash

사용자 정의 타입을 unordered 컨테이너에 넣으려면 hash function 을 제공해야 합니다.

템플릿 인자로 전달하는 방식과 기존의 hash 템플릿의 부분 전문화를 사용하는 방식등 2가지 형태로 구현 가능합니다.

```

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

struct People
{
    string name;
    int age;
};

struct PeopleHash
{
    int operator()(const People& p) const
    {
        return hash<string>()(p.name) + hash<int>()(p.age);
    }
};

struct PeopleEqual
{
    bool operator()(const People& p1, const People& p2) const
    {
        return p1.name == p2.name && p1.age == p2.age;
    }
};

int main()
{
    unordered_set<People, PeopleHash, PeopleEqual> s{ People{"kim", 10},
                                                       People{"park", 20} };
}

```

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

struct People
{
    string name;
    int age;
};

template<> struct hash<People>
{
    int operator()(const People& p) const
    {
        return hash<string>()(p.name) + hash<int>()(p.age);
    }
};

template<> struct equal_to<People>
{
    bool operator()(const People& p1, const People& p2) const
    {
        return p1.name == p2.name && p1.age == p2.age;
    }
};

int main()
{
    unordered_set<People> s{ People{ "kim", 10 }, People{ "park", 20 } };
}
```

## Section 6.

# utility

항목 6-1

# smart pointer #1

## 학습 목표

- smart pointer 기본 개념
- shared\_ptr
- make\_shared
- enable\_shared\_from\_this



## 1. C++ 표준 스마트 포인터

### I 핵심 개념

C++ 표준에는 다음과 같은 스마트 포인터를 제공합니다.

name		description
auto_ptr		소유권 이전 방식의 스마트 포인터 ( deprecated in C++17)
unique_ptr	C++11	독점적 소유권 방식
shared_ptr	C++11	소유권 공유 방식
weak_ptr	C++11	소유권에 참여하지 않음
observer_ptr	C++17	소유권 없음(no ownership)

## 2. shared\_ptr

### I 핵심 개념

- 소유권 공유의 개념을 구현한 스마트 포인터, 참조 계수 방식으로 객체의 수명을 관리.
- Raw Pointer의 2개의 크기를 가지게 됩니다.
- shared\_ptr 생성시 참조계수등을 관리하는 제어 블록이 생성됩니다.
- 참조계수의 증가/감소는 원자적 연산으로 수행됩니다.

### I 기본 코드

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

class Car
{
    int color;
public:
    Car() { cout << "Car()" << endl; }
    ~Car() { cout << "~Car()" << endl; }
    void Go() { cout << "Car Go" << endl; }
};

int main()
{
    shared_ptr<Car> p1(new Car);
    p1->Go();
    shared_ptr<Car> p2 = p1;
    cout << p2.use_count() << endl;
    p1.reset();
    cout << p2.use_count() << endl;
    cout << p2 << endl;
    cout << p2.get() << endl;
}
```

## I . 연산과 -> 연산

shared\_ptr 사용시 . 연산을 사용하면 shared\_ptr 자체의 멤버에 접근할 수 있습니다.

-> 연산을 사용하면 대상 객체의 멤버에 접근할 수 있습니다.

```
#include <iostream>
#include <memory>
#include "Car.h"
using namespace std;

int main()
{
    shared_ptr<Car> p1( new Car ); //1

    p1->Go(); // Car 의 멤버 접근.

    Car* p = p1.get();
    cout << p << endl;

    shared_ptr<Car> p2 = p1; // 2
    cout << p1.use_count() << endl; // 2

    //p1 = new Car; // error
    p1.reset( new Car ); // ok
    p1.reset();
    p1.swap(p2);
}
```

## ■ 삭제자(delete) 변경하기

shared\_ptr<T>은 기본적으로 소멸자에서 delete를 사용해서 객체를 파괴 합니다. 함수, 함수객체, 람다표현식 등을 사용해서 객체의 파괴방식을 변경할 수 있습니다.

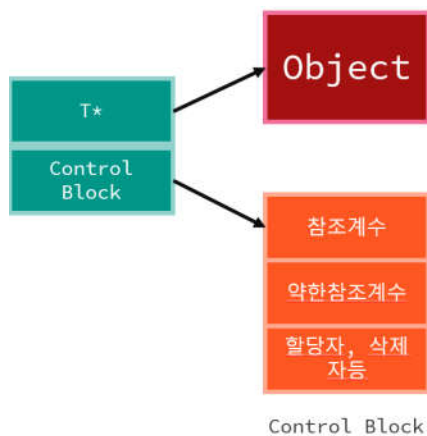
```
#include <iostream>
#include <memory>
using namespace std;

void del_arr(int* p)
{
    delete[] p;
}

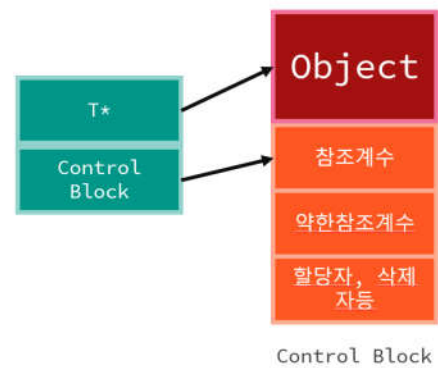
int main()
{
    shared_ptr<int> p1(new int);           // delete
    shared_ptr<int> p2(new int[10], del_arr);
    shared_ptr<int> p3(new int[10], [](int* p) { delete[] p; });
}
```

## ■ make\_shared

shared\_ptr<T>은 참조계수 기반으로 객체를 관리 하기 때문에 shared\_ptr<T>의 객체를 생성하면 참조계수 등을 관리하기 제어블럭이 같이 생성되게 됩니다. 이때, make\_shared 를 사용하면 객체와 제어블럭을 묶어서 메모리를 할당 할 수 있기 때문에 보다 효율적으로 메모리를 사용할 수 있습니다.



make\_shared\_를 사용하지 않은 경우



make\_shared\_를 사용하는 경우

operator new() 함수를 재정의해서 테스트해 볼 수 있습니다.

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

class Car
{
    int color;
public:
    Car() { cout << "Car()" << endl; }
    ~Car() { cout << "~Car()" << endl; }
};

void* operator new(size_t sz)
{
    cout << "operator new : " << sz << endl;
    return malloc(sz);
}

int main()
```

```
{
    shared_ptr<Car> p1(new Car);    // 1. Car 객체 생성
                                   // 2. control block 생성
    shared_ptr<Car> p2 = make_shared<Car>(); // sizeof(Car) + sizeof(control block)을
한번에 생성
}
```

## ■ enable\_shared\_from\_this

enable\_shared\_from\_this 를 사용하면 객체 안에서 this를 사용해서 객체 자신의 참조계수를 증가할 수 있습니다.

```
#include <iostream>
#include <memory>
#include <thread>
using namespace std;
class Worker : public enable_shared_from_this<Worker>
{
    int data = 0;
    shared_ptr<Worker> holdme;
public:
    ~Worker() { cout << "~Worker()" << endl; }
    void run()
    {
        holdme = shared_from_this();
        thread t(&Worker::main, this);
        t.detach();
    }
    void main()
    {
        cout << "Worker main" << endl;
        this_thread::sleep_for(1s);
        data = 100;
        cout << "Worker end" << endl;
        holdme.reset();
    }
};
int main()
{
    {
        shared_ptr<Worker> p(new Worker);
        p->run();
    }
    cout << "main end" << endl;
    getchar();
}
```

항목 6-2

## smart pointer #2

### 학습 목표

- 상호 참고와 weak\_ptr
- unique\_ptr



## 1. 스마트 포인터의 상호 참조 문제

### I 핵심 개념

shared\_ptr<>을 사용할 때 상호 참조가 발생하면 메모리 누수가 발생할 수 있습니다.

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

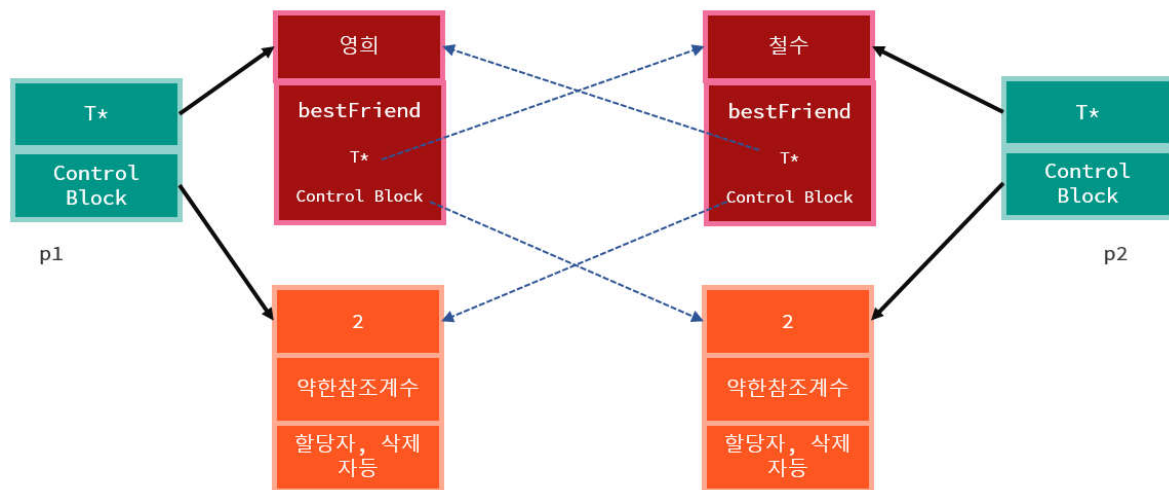
class People
{
public:
    People(string s) : name(s) {}
    ~People() { cout << name << " 파괴" << endl; }

    shared_ptr<People> bestFriend;
private:
    string name;
};

int main()
{
    shared_ptr<People> p1(new People("영희"));
    shared_ptr<People> p2(new People("철수"));

    p1->bestFriend = p2;
    p2->bestFriend = p1;
}
```

위 코드 실행시 메모리 그림이 다음과 같습니다.



이 문제를 해결하려면 참조 계수가 증가 하지 않은 스마트 포인터가 필요합니다. 다음 항목을 참고하세요

## 2. weak\_ptr

### I 핵심 개념

객체의 수명에 관련된 참조 계수를 증가하지 않은 스마트 포인터.

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

class Car
{
    int color;
public:
    Car() { cout << "Car()" << endl; }
    ~Car() { cout << "~Car()" << endl; }
};

int main()
{
    shared_ptr<Car> sp1 = make_shared<Car>();
    cout << sp1.use_count() << endl; // 1

    shared_ptr<Car> sp2 = sp1;
    cout << sp1.use_count() << endl; // 2

    weak_ptr<Car> wp1 = sp1;
    cout << sp1.use_count() << endl; // 2

    sp2.reset();
    cout << sp1.use_count() << endl; // 1
}
```

## I expired

`weak_ptr<>`의 `expired()` 멤버 함수를 사용하면 객체가 파괴되었는지를 조사할 수 있습니다.

```
int main()
{
    shared_ptr<Car> sp1 = make_shared<Car>();
    weak_ptr<Car> wp1 = sp1;

    sp1.reset();

    if (wp1.expired())
        cout << "객체가 파괴되었습니다." << endl;
    else
    {
        cout << "객체가 파괴 되지 않았습니다." << endl;

        wp->Go(); // error, weak_ptr<>은 -> 연산자를 제공하지 않습니다.
    }
}
```

하지만, `weak_ptr<>`은 `->` 연산자를 제공하지 않기 때문에 객체에 접근할 수는 없습니다. 따라서, `weak_ptr`을 사용해서 객체에 접근하려면 다시 `shared_ptr<>`을 만들어서 사용해야 합니다.

## weak\_ptr<>로 객체에 접근 하는 방법.

weak\_ptr<>로 객체를 가리킬 때 객체에 접근하려면 weak\_ptr<>의 lock() 함수를 사용해서 shared\_ptr<>의 객체를 생성해야 합니다.

```
int main()
{
    shared_ptr<Car> sp1 = make_shared<Car>();
    weak_ptr<Car> wp1 = sp1;

    shared_ptr<Car> sp2 = wp1.lock();
    if (sp2)
        sp2->Go();
}
```

## weak\_ptr<>을 사용한 상호 참조 문제의 해결

weak\_ptr<>를 사용하면 shared\_ptr<>의 상호 문제를 해결 할 수 있습니다. 상호 참조가 발생하는 경우 shared\_ptr<>대신 weak\_ptr<>을 사용하면 됩니다.

```
class People
{
public:
    People(string s) : name(s) {}
    ~People() { cout << name << " 파괴" << endl; }

    void setBestFriend(weak_ptr<People> wp) { bestFriend = wp; }
    void print_bestFriend()
    {
        shared_ptr<People> sp = bestFriend.lock();

        if (sp)
            cout << sp->name << endl;
    }
private:
    string name;
    weak_ptr<People> bestFriend;
};

int main()
```

```
{
    shared_ptr<People> p1(new People("영희"));
    shared_ptr<People> p2(new People("철수"));

    p1->setBestFriend(p2);
    p2->setBestFriend(p1);
    p1->print_bestFriend();
}
```

### 3. unique\_ptr

#### ■ 핵심 개념

자원에 대한 독점적 소유권을 가진 스마트 포인터. 복사는 불가능하지만, 이동은 가능합니다.

#### ■ 기본 코드

weak\_ptr<>를 사용하면 shared\_ptr<>의 상호 문제를 해결 할 수 있습니다. 상호 참조가 발생하는 경우 shared\_ptr<>대신 weak\_ptr<>을 사용하면 됩니다.

```
int main()
{
    unique_ptr<Car> up1 = make_unique<Car>();
    unique_ptr<Car> up2 = up1;      // error
    unique_ptr<Car> up3 = move(up1); // ok
}
```

#### ■ unique\_ptr<> 과 배열

배열 형식으로 메모리를 할당할 경우 unique\_ptr<>의 타입인자는 배열형식으로 전달해야 합니다. 또한, 배열 타입의 unique\_ptr<>은 [] 연산은 가능하지만 \* 연산은 사용할 수 없습니다.

```
int main()
{
    unique_ptr<int> up1(new int);
    unique_ptr<int> up2(new int[10]); // bug.
    unique_ptr<int[]> up3(new int[10]); // ok.
                                // int 타입의 unique_ptr

    *up1 = 10; // ok
    up1[0] = 10; // error, [] 연산자는 배열타입의 unique_ptr만 가능합니다.
                // 배열 타입의 unique_ptr
    *up3 = 10; // error. 배열타입의 unique_ptr의 경우 *을 사용할수 없습니다.
    up3[0] = 10; // ok.
}
```

## I 삭제자 변경

`unique_ptr`의 경우, 삭제자를 변경하려면 삭제자의 타입을 템플릿 인자로 전달하면 됩니다.

```
struct Freer
{
    void operator()(void* p) { free(p); }
};

int main()
{
    unique_ptr<int, Freer> up1(new malloc(100));

    // unique_ptr의 삭제자로 람다 표현식을 사용한 경우
    auto freer = [](void* p) { free(p); };
    unique_ptr<int, decltype(freer)> up2(new malloc(100), freer);
}
```



항목 6-3

# chrono

## 학습 목표

- ratio
- duration
- using chrono
- clock

## 1. std::ratio

### ■ 핵심 개념

- defined in header <ratio>
- 컴파일 시간 상수 값을 가지는 유리수의 표현(분자와 분모)과 유리수 연산을 지원하는 클래스 템플릿.

ratio template. defined in header <ratio>

```
template< std::intmax_t Num, std::intmax_t Denom = 1 > struct ratio
{
    static constexpr intmax_t num = Num;
    static constexpr intmax_t den = Denom;
    .....
};
```

[참고] 실제 구현은 compile-time 약분을 수행 해서 값을 넣게 됩니다.

### ■ ratio를 사용한 컴파일 시간 유리수 ( rational number )의 표현

ratio 템플릿은 컴파일 시간 상수를 가지고 분자와 분모로 구성된 유리수 하나를 표현할 때 사용합니다. ratio 템플릿 안에는 각각 분자와 분모를 나타내는 상수 값을 static 멤버 데이터로 보관하고 있습니다.

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

int main()
{
    ratio<2, 10> r1; // 2/10 을 나타 냅니다.
                    // 컴파일 시간 연산을 통해서 1/5로 저장 됩니다.

    // ratio 안의 분자, 분모 값은 static 멤버 이므로 객체 이름 또는 클래스 이름으로
    // 접근할 수 있습니다.
    // 1. 객체 이름을 사용한 분자/분모 접근
    cout << r1.num << endl;
    cout << r1.den << endl;
```

```
// 2. 타입 이름을 사용한 분자/분모 접근
cout << ratio<2, 10>::num << endl;
cout << ratio<2, 10>::den << endl;
}
```

다음의 간단한 pr() 함수 템플릿으로 ratio의 값을 출력해 볼 수 있습니다. pr()함수 템플릿은 2가지 방식으로 사용할 수 있습니다.

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

template<typename T> void pr(const T& r1 = T())
{
    cout << T::num << "/" << T::den << endl;
}

int main()
{
    ratio<1, 5> r1;
    pr(r1); // 1. pr 함수에 객체를 전달하는 방식
    pr<ratio<3, 10>>(); // 2. pr 함수에 타입을 전달하는 방식
}
```

## I 컴파일 시간(compile-time) 유리수 연산

C++ 표준에는 유리수에 대해 컴파일 시간 사칙 연산을 수행하는 4개의 메타 함수와 6개의 비교함수를 제공합니다.

compile-time rational arithmetic	ratio_add ratio_subtract ratio_multiply ratio_divide
compile-time rational comparison	ratio_equal ratio_not_equal ratio_less ratio_less_equal ratio_greater ratio_greater_equal

다음 코드는 수치 연산에 대한 간단한 예제 입니다.

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

template<typename T> void pr(const T& r1 = T())
{
    cout << T::num << "/" << T::den << endl;
}

int main()
{
    // compile-time rational arithmetic
    ratio_add<    ratio<1, 10>, ratio<3, 5>> r1; pr(r1); // 7/10
    ratio_subtract<ratio<1, 10>, ratio<3, 5>> r2; pr(r2); // -1/2
    ratio_multiply<ratio<1, 10>, ratio<3, 5>> r3; pr(r3); // 3/50
    ratio_divide <ratio<1, 10>, ratio<3, 5>> r4; pr(r4); // 1/6
}
```

ratio\_equal 등의 비교 연산을 사용할 때는 다음의 2가지 방식으로 사용이 가능합니다.

- ① ::value 를 꺼내서 if 문으로 확인하는 방식.
- ② true\_type, false\_type을 사용한 함수 오버로딩을 사용하는 방식.

```

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

void pr_cmp(true_type) { cout << "pr_cmp : true" << endl; }
void pr_cmp(false_type) { cout << "pr_cmp : false" << endl; }

int main()
{
    // 1. value를 사용하는 방식
    if ( ratio_equal<ratio<1, 5>, ratio<2, 10>>::value )
        cout << "Same" << endl;

    // 2. 함수 오버로딩을 사용하는 방식
    pr_cmp(ratio_equal<ratio<1, 5>, ratio<2, 10>>());
}

```

## Ⅰ using을 사용한 SI 단위

일반적으로 kilo는 1000/1을 의미하고 centi는 1/100, milli 는 1/1000을 의미 합니다. typedef (또는 using) 와 ratio를 사용하면 간단하게 국제 표준 단위(SI)를 만들 수 있습니다. C++ 표준에는 아래와 같은 표준 SI ratios를 제공하고 있습니다.

C++ 표준이 제공하는 SI typedefs

```

typedef ratio<1, 1000000000000000000LL> atto;
typedef ratio<1, 1000000000000000LL> femto;
typedef ratio<1, 1000000000000LL> pico;
typedef ratio<1, 1000000000> nano;
typedef ratio<1, 1000000> micro;
typedef ratio<1, 1000> milli;
typedef ratio<1, 100> centi;
typedef ratio<1, 10> deci;
typedef ratio<10, 1> deca;
typedef ratio<100, 1> hecto;
typedef ratio<1000, 1> kilo;
typedef ratio<1000000, 1> mega;
typedef ratio<1000000000, 1> giga;
typedef ratio<1000000000000LL, 1> tera;
typedef ratio<1000000000000000LL, 1> peta;
typedef ratio<1000000000000000000LL, 1> exa;

```

따라서, 100분의 1이 필요한 경우 `ratio<1,100>` 대신 `centi` 를 사용할 수 있습니다.

chrono\ratio5.cpp

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

template<typename T> void pr(const T& r1 = T())
{
    cout << T::num << "/" << T::den << endl;
}

int main()
{
    kilo k;
    pr(k);           // 1000/1
    pr<mega>();       // 1000000/1

    pr<centi>();      // 1/100
    pr<micro>();      // 1/1000000
}
```

## 2. std::chrono::duration

### ■ 핵심 개념

- defined in header <chrono>, std::chrono namespace
- 특정 단위(주기, period)에 대한 하나의 값(tick)을 보관하는 클래스 템플릿.
- 시간의 간격을 나타내는 클래스.

duration template. defined in header <chrono>

```
template<typename Rep,                // 값의 타입
        typename Period = std::ratio<1, 1>> // 값의 단위(주기)
class duration;
```

### ■ duration 의 개념

duration 템플릿은 ratio 로 표현되는 단위(주기)에 대한 하나의 값을 보관하는 클래스 입니다. duration이 보관하는 것은 오직 하나의 값입니다. duration을 사용하면 단위에 맞게 자동으로 연산 되도록 만들수 있습니다.

다음은 duration을 사용하는 예제 입니다.

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

int main()
{
    // 1. duration 사용
    duration<int, ratio<1, 100>> d1(2); // 1/100 * 2
    duration<int, ratio<1, 1000>> d2(d1);
    cout << d2.count() << endl; // 20

    // 2. using predefined SI
    using Meter      = duration<int, ratio<1, 1>>;
    using KiloMeter  = duration<int, kilo>;
    using CentiMeter = duration<int, centi>;
```

```

KiloMeter km(1);

CentiMeter cm(km);
cout << cm.count() << endl; // 100,000

CentiMeter c2(150);
//Meter m2 = c2;           // error. data 손실의 가능성이 있으면 암시적 변환 안됨.

Meter m2 = duration_cast<Meter>(c2);
cout << m2.count() << endl;

// 3. casting.
Meter me2(560);
cout << duration_cast<KiloMeter>(me2).count() << endl;
cout << floor<KiloMeter>(me2).count() << endl; // 버림
cout << ceil<KiloMeter>(me2).count() << endl; // 올림
cout << round<KiloMeter>(me2).count() << endl; // 반올림
cout << abs(me2).count() << endl;
}

```

## I duration 과 시간

STL에는 duration을 사용하여 시간관련 라이브러리를 제공합니다.

```

#include <iostream>
#include <chrono>
#include <thread>
using namespace std;
using namespace std::chrono;

int main()
{
    // 1. predefined typedef
    hours      h(1); // typedef duration<int, ratio<3600,1>> hours;
    minutes    m = h;
    seconds     s = h; // typedef duration<int, ratio<1,1>> seconds;
    milliseconds ms = h;
    microseconds us = h;
    nanoseconds ns = h;
}

```



```
cout << h.count() << endl;
cout << m.count() << endl;
cout << s.count() << endl;
cout << ms.count() << endl;
cout << us.count() << endl;
cout << ns.count() << endl;

// 2. literals( h, min, s, ms, us, ns )
seconds sec = 1min;

sec = 1min + 3s;
sec += 40s;
cout << sec.count() << endl;

seconds sec2 = 1min + 3s;
cout << sec2.count() << endl;
}
```

### 3. std::chrono::system\_clock

#### I system\_clock & time\_point

- system\_clock : 현재 시간을 얻을 때 사용하는 클래스
- time\_point : 시간의 시작점과 duration< >을 보관하는 클래스

```
#include <iostream>
#include <chrono>
#include <ctime>
#include <string>
using namespace std;
using namespace std::chrono;

int main()
{
    // 1. 현재 시간을 얻는 방법
    system_clock::time_point tp = system_clock::now(); // 정적함수 호출

    // 2. time_point=>duration 얻어내기. 정밀도가 낮아지는 경우 명시적 캐스팅 필요.
    nanoseconds nanosec = tp.time_since_epoch(); // 1970년 1월1일 이후
                                                // 단위로 리턴.
    seconds sec = duration_cast<seconds>(tp.time_since_epoch());

    cout << nanosec.count() << endl;
    cout << sec.count() << endl;

    using days = duration<int, ratio<60*60*24, 1>>;
    cout << (duration_cast<days>(tp.time_since_epoch())).count() << endl;

    // 3. time_point => 칼린더 형태의 날짜로 변경하기
    time_t t = system_clock::to_time_t(tp); // 1. time_point => struct time_t
    string s = ctime(&t);                  // 2. struct time_t => char*
    cout << s << endl;
}
```

항목 6-4

# function & bind

## 학습 목표

- `std::bind`
- `std::placeholders` namespace
- `std::function`

## 1. std::bind

### ■ 핵심 개념

callable object의 인자 또는 객체를 고정한 forwarding call wrapper 를 생성하는 함수 템플릿.

기본 모양

```
template<class F, class ... Args>          /*unspecified*/ bind(F&& f, Args&& ... args);
template<class R, class F, class ... Args> /*unspecified*/ bind(F&& f, Args&& ... args);
```

### ■ callable object

invoke 연산을 사용할 수 있는 객체로서, std::function<>, std::bind(), std::thread::thread(), std::invoke()등에 사용 할 수 있습니다. 다음과 같은 종류가 있습니다.

- function object - function, reference to function, pointer to function, member function, pointer to member function, ()연산자를 재정의한 클래스, lambda expression
- pointer to member data

주의할 점은 멤버 함수 포인터 뿐 아니라 멤버 데이터를 가리키는 포인터도 callable object 입니다.

## I bind 기본 예제

```
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;

void foo(int a, int b)
{
    cout << "foo : " << a << ", " << b << endl;
}

int main()
{
    // foo(1,2) 인자 2개를 1, 2로 고정한 인자 없는 callable object 생성
    bind(&foo, 1, 2)();

    // foo(1,3) 2번째 인자를 3으로 고정한 인자가 한개인 callable object 생성
    bind(&foo, _1, 3)(1);
    bind(&foo, _2, _1)(1, 3); // foo(3,1)
}
```

## I 인자를 참조 타입으로 바인딩 하기

bind() 사용시 인자를 참조로 binding 하려면 reference\_wrapper< > 또는 ref() 를 사용해야 합니다.

```
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;

void foo(int a, int& b)
{
    b = 30;
}

int main()
{
    int n = 10;
```

```

    bind(&foo, 10, reference_wrapper<int>(n))(); // foo( 10, n);

    cout << n << endl; // 30

    n = 10;
    bind(&foo, 10, ref(n))();
    cout << n << endl; // 30
}

```

## ■ 멤버 함수 와 객체 binding

멤버 함수는 다음의 2가지 형태로 바인딩 할 수 있습니다.

- ① 객체 자체를 포함해서 바인딩.
- ② 객체를 포함 시키지 않고, 함수 호출 시 객체를 인자로 전달하도록 바인딩.

```

#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;

class Car
{
    int color;
public:
    void Go(int speed) { cout << "Car Go " << this << endl; }
};

int main()
{
    Car c;

    // 1. 객체 자체를 binding 하는 방식
    bind(&Car::Go, &c, 10)();
    bind(&Car::Go, &c, _1)(10);

    // 2. 객체를 함수 인자로 전달하는 방식
    bind(&Car::Go, _1, 10)(c);
    bind(&Car::Go, _1, 10>(&c);
    bind(&Car::Go, _1, _2)(c, 10);
    bind(&Car::Go, _1, _2>(&c, 10);

```

```
}
```

## ■ 멤버 데이터 binding

static 이 아닌 멤버 data(public) 도 바인딩 될 수 있습니다.

```
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;

class Car
{
public:
    int color = 10;
};

int main()
{
    Car c;
    cout << bind(&Car::color, &c)() << endl;    // 10

    bind(&Car::color, &c)() = 20;
    cout << c.color << endl;    // 20

    bind(&Car::color, _1)(c) = 30;

    cout << c.color << endl; // 30
}
```

## 2. std::function

### I 핵심 개념

함수 포인터의 일반화된 개념으로서 일반화된 함수 래퍼 입니다. callable object 를 저장 했다가 호출(involve) 할 때 사용합니다.

<functional>

```
template< class > class function; /* undefined */
template< class R, class... Args > class function<R(Args...)>;
```

### I 기본 예제

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

void foo0()          { cout << "f0" << endl; }
void foo1(int a)      { cout << "f1 : " << a << endl; }
int  foo2(int a, double d) { cout << "f2 : " << a << ", " << d << endl; return a; }

int main()
{
    // 인자가 0~2개인 함수를 보관할 function 만들기
    function<void()>      f0 = foo0;
    function<void(int)>    f1 = foo1;
    function<int(int, double)> f2 = foo2;

    f0();
    f1(5);

    int ret = f2(10, 3.4);
    cout << ret << endl;
}
```



## ■ function 과 bind

std::bind를 사용하면 함수의 인자의 개수를 변경해서 function 담을 수도 있습니다

```
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;

void foo1(int a)
{
    cout << "f1 : " << a << endl;
}
void foo2(int a, double d)
{
    cout << "f2 : " << a << ", " << d << endl;
}
int main()
{
    function<void(int)> f = foo1;
    f(10); // foo1(10);

    f = bind(&foo2, _1, 3);
    f(10); // foo2(10, 3);
}
```

## ■ function 에는 다양한 callable object를 담을 수 있습니다.

function 에는 일반 함수 뿐 아니라, 함수 객체, 람다 표현식, 멤버 함수, 멤버 data 등도 저장했다가 호출할 수 있습니다.

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

int main()
{
    function<int(int, int)> f1 = plus<int>();
    function<int(int, int)> f2 = [](int a, int b) { return a + b; };

    cout << f1(1, 2) << endl;
}
```

```

    cout << f2(1, 2) << endl;
}

```

## ■ 멤버 함수 와 function

function 에는 멤버 함수도 저장할 수 있습니다. 이 경우에는 2가지 형태로 사용이 가능합니다.

- ① function 에 인자로 객체를 전달하는 방식
- ② bind를 사용해서 객체를 저장하는 방식

bind로 객체를 binding을 할 때도 값, 참조, 주소로 바인딩 할 수 있습니다.

```

class Car
{
    int color;
public:
    void Go(int speed) { cout << "Car Go " << this << endl; }
};

int main()
{
    Car c;
    cout << &c << endl;

    // 1. bind를 사용하지 않은 경우
    function<void(Car&, int)> f1 = &Car::Go;
    f1(c, 10);

    // 2. bind를 사용하는 경우 - 값 binding
    function<void(int)> f2 = bind(&Car::Go, c, _1);
    f2(10);

    // 주소 binding
    function<void(int)> f3 = bind(&Car::Go, &c, _1);
    f3(10);

    // 참조 binding
    function<void(int)> f4 = bind(&Car::Go, ref(c), _1);
    f4(10);
}

```

## ■ 멤버 데이터와 function

std::bind 와 유사하게 function 에도 멤버 데이터의 주소를 담을 수 있습니다. 이 경우도 2가지 방법이 있습니다.

- ① function 에 인자로 객체를 전달하는 방식
- ② bind를 사용해서 객체를 저장하는 방식

또한, function 을 사용한 호출을 lvalue에 놓으려면 참조 타입으로 담아야 합니다.

```
struct Test
{
    int data = 10;
};

int main()
{
    Test t;

    // 1. bind를 사용하지 않은 경우.
    function<int(Test&)> f1 = &Test::data;

    cout << f1(t) << endl;

    // 2. bind를 사용해서 객체를 고정하는 경우
    function<int()> f2 = bind(&Test::data, &t);
    cout << f2() << endl;

    // f1(t) = 20; // error
    // f2() = 20; // error
    // 3. lvalue에 오게 하려면 참조 타입으로 저장해야 합니다.
    function<int&(Test&)> f3 = &Test::data;
    function<int&()> f4 = bind(&Test::data, &t);
    f3(t) = 20;
    f4() = 30;
    cout << t.data << endl; // 30
}
```

Section 7.

# Concurrency

항목 7-1

# thread

## 학습 목표

- thread
- mutex & lock\_guard
- thread\_local
- promise & future
- packaged\_task
- async

## 1. thread

### I thread 생성

C++에서 스레드를 생성하려면 thread 객체를 생성하면 됩니다 또한, 스레드를 생성한 후에는 join() 하거나 detach() 해야 합니다.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
using namespace std::chrono;

// Step1. thread

int foo(int n)
{
    cout << "start foo, thread id : " << this_thread::get_id() << endl;
    this_thread::sleep_for(2s);
    cout << "end foo" << endl;
    return 100;
}

int main()
{
    thread t(foo, 10);

    cout << "main id : " << this_thread::get_id() << endl;
    t.join();
}
```

### I thread 와 callable object

다양한 종류의 callable object를 스레드 함수로 사용할 수 있습니다.

```
#include <iostream>
#include <thread>
#include <chrono>
using namespace std;
using namespace std::chrono;
```

```

void f1() {}

struct F00
{
    void operator()(int a) {}
};
class Test
{
public:
    void foo() {}
};
int main()
{
    thread t1(f1);           // 일반 함수
    thread t2(F00(), 10);    // 함수 객체

    Test test;
    thread t3(&Test::foo, &test); // 멤버 함수
    thread t4([](int a) {}, 10); // 람다 표현식

    t1.detach();
    t2.detach();
    t3.detach();
    t4.detach();
}

```

## ❏ 동기화 객체와 lock\_guard

무텍스, 세마 포어, 조건변수 등 대부분의 동기화 요소 역시 제공됩니다. 또한, 무텍스 등을 사용할 때는 lock\_guard 등을 사용하는 것이 좋습니다.

```

#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

mutex m;

void foo()
{

```

```
    lock_guard<mutex> g(m);

    // .....
}
int main()
{
    thread t1(foo);
    thread t2(foo);
    t1.join();
    t2.join();
}
```

## ■ thread local storage ( thread specific storage)

스레드별 메모리 공간을 할당 하려면 `thread_local` 키워드를 사용합니다.

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

int next3times()
{
    thread_local static int n = 0;
    n = n + 3;
    return n;
}
void foo(string name)
{
    cout << name << " : " << next3times() << endl;
    cout << name << " : " << next3times() << endl;
    cout << name << " : " << next3times() << endl;
}

int main()
{
    thread t1(foo, "A"s);
    thread t2(foo, "B"s);

    t1.join();
    t2.join();
}
```



```
}

```

## I promise & future

주 스레드에서 새로운 스레드의 결과 값을 꺼내 올 때는 promise 객체와 future 객체를 활용합니다.

```
#include <iostream>
#include <future>
#include <chrono>
using namespace std;
using namespace std::chrono;

int foo(promise<int>& p)
{
    cout << "start foo, thread id : " << this_thread::get_id() << endl;
    this_thread::sleep_for(2s);
    cout << "end foo" << endl;
    p.set_value(100);
    return 100;
}

int main()
{
    promise<int> p;
    thread t(foo, ref(p));

    future<int> ft = p.get_future();
    cout << "main id : " << this_thread::get_id() << endl;

    int n = ft.get();
    cout << "value : " << n << endl;
    t.join();
}
```

## I packaged\_task

packaged\_task 객체를 사용하면 thread 의 리턴 값을 promise 객체를 사용해서 전달할 수 있습니다.

```

#include <iostream>
#include <future>
#include <chrono>
#include <string>
using namespace std;
using namespace std::chrono;

string foo()
{
    cout << "start packaged test : " << this_thread::get_id() << endl;
    this_thread::sleep_for(2s);
    cout << "end packaged test" << endl;

    return "hello"s;    // p.set_value("hello"s)
}

int main()
{
    packaged_task<string> p_task(foo); // promise를 포함

    auto future = p_task.get_future();
    cout << "main : " << this_thread::get_id() << endl;

    thread t1(move(p_task));
    auto ret = future.get();
    cout << "result : " << ret << endl;

    t1.join();
}

```

## I async

async() 함수를 사용하면 스레드를 쉽게 생성하고 결과값을 가져올 수 있습니다.

```

#include <iostream>
#include <string>
#include <future>
#include <chrono>
using namespace std;
using namespace std::chrono;

string foo()

```

```
{
    cout << "start foo : " << this_thread::get_id() << endl;
    this_thread::sleep_for(2s);
    cout << "end foo" << endl;

    return "hello"s;
}
int main()
{
    cout << "main thread : " << this_thread::get_id() << endl;

    // 1. 리턴값을 받지 않은 경우
    //async(foo);

    // 2. 리턴값을 받는 경우
    future<string> ft = async(foo);

    // 3. launch option 사용
    //future<string> ft = async(launch::deferred, foo);

    cout << "main end" << endl;

    auto r = ft.get();
    cout << "value : " << r << endl;
}
```