목 차

SECTION 1

C++ 11/14/17

Modern C++

기본 문법

항목 1-01

Hello, Modern C++

주요 학습 내용

> C++ 의 역사
>
> C++ 버전과 컴파일 하는 방법

1. C++의 역사

▌ C++ 탄생 부터 C++98/03

C++은 1979년경에 “ C with Classes”라는 이름으로 Bjarne Stroustrup 에 의해 탄생되었습니다.

C++탄생 이후 표준화 없이 계속 사용되다가 1990년 C++ 표준위원회가 설립되고, 1998년 드디어 1차 표준화를 하게 됩니다. 이때 표준화된 문법을 “C++98” 이라고 부릅니다. C++ 표준 라이브러리인 STL(Standard Template Library)도 이 때 나오게 됩니다.

그후, 2003년에 표준화 문법의 사소한 버그를 수정하고 몇가지를 추가해서 새로운 표준을 발표 합니다. 이 버전을 “C++03”이라고 합니다. 흔히, 2개 버전을 합쳐서 “C++98/03” 이라고 부르기도 합니다.

▌ Modern C++ (C++ 11/14/17/20)

C++93/03이후 2000년대에 들어 C++ 표준 위원회는 새로운 표준을 만들기로 합니다. 2009년 전에는 표준을 완성하겠다는 한자리 연도를 의미하는 “C++0x”라는 이름으로 부르게 됩니다. 하지만, 의지와는 다르게 2자리수의 연도인 2011년에 표준이 완성되고 ISO를 통과 하게 됩니다. 2011년에 표준화된 이 새로운 문법을 “C++11” 이라고 부릅니다. 그 후, 표준화 내용의 사소한 버그를 수정하고 몇가지 기능을 추가해서 2014년 “C++14”를 발표 합니다. “C++14”는 발표 되기 전까지 “C++1y” 라는 이름으로 불리었습니다.

C++11/14 발표 이후 다시 표준위원회는 새로운 표준을 만들기 위해 노력 하게 됩니다. “C++1z”라는 프로젝트 이름으로 출발한 새로운 표준은 2017년 발표되었습니다. 그리고 2020년에 C++20을 발표할 예정입니다.

흔히, C++11 이후의 C++을 "Modern C++” 이라고 부르기도 합니다.

2. 컴파일 하는 방법

C++11/14/17/20 에는 다양한 문법이 추가 되었습니다.

“lambda expression, move semantics, perfect forwarding, uniform initialization, auto, decltype, using alias, structure binding...”

이와 같은 최신 문법을 가진 소스를 컴파일 하려면 컴파일러에 옵션을 제공해야 합니다.

▌ g++ 를 사용해서 컴파일

g++을 사용해서 C++11/14/17 문법을 컴파일 하려면 -std 옵션을 지정해야 합니다.

```
g++ hello.cpp -std=c++11    // C++11 문법으로 컴파일
g++ hello.cpp -std=c++14    // C++14 문법으로 컴파일
g++ hello.cpp -std=c++17    // C++17 문법으로 컴파일
g++ hello.cpp -std=c++2a    // C++20 문법으로 컴파일
```

[참조] g++ 버전에 따라서 별도에 옵션을 지정하지 않고도 컴파일이 가능한경우도 있습니다. 또한, "c++2a" 옵션을 사용하려면 g++ 8.x 버전을 사용해야 합니다.

같은 소스라도 옵션에 따라 빌드 에러가 발생하기도 합니다.

▌ Visual C++를 사용해서 컴파일

MS의 VC++ 컴파일러를 사용할 경우 VC++2017 이상 버전을 사용해야 C++11/14/17 버전을 컴파일 할수 있습니다. cl 컴파일러를 가지고 직접 컴파일 할 경우 아래와 같이 /std:latest 옵션을 사용해서 컴파일 하면 됩니다.

```
cl hello.cpp /std:latest
```

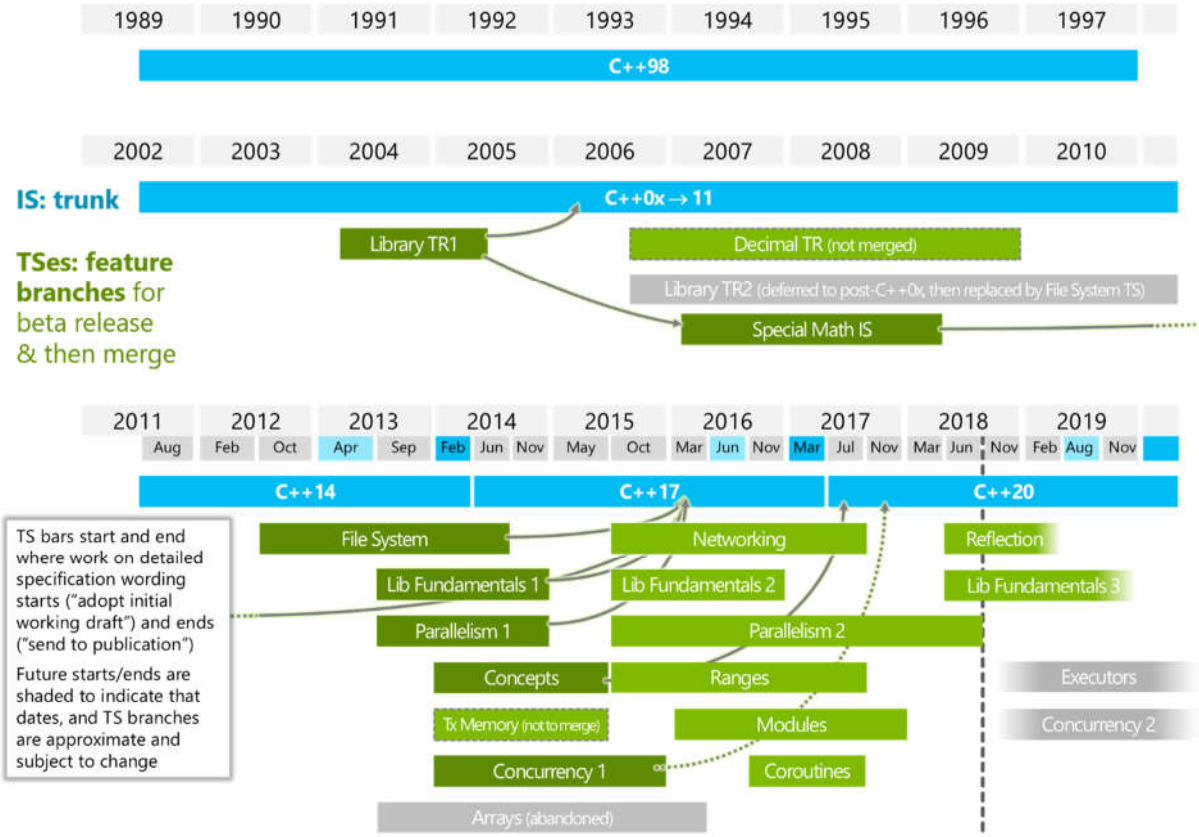
[참조] vc++의 경우 2019년 1월 현재 C++17/20 문법을 완전하게 지원하지는 않습니다.

3. Reference Site

▌ isocpp.org

C++ 표준위원회가 운영하는 C++ 공식 사이트 입니다. C++ 관련 많은 자료를 얻을수 있습니다.

또한, C++ 표준에 대한 최신 정보를 얻을수 있습니다.



"Current ISO C++ status" from isocpp.org

▌ cppreference.com

C++관련 표준 문법과 STL 라이브러리에 대한 자세한 설명과 예제를 제공하는 사이트 입니다.

항목 1-02.

C++11/14 기본 문법

주요 학습 내용

> static_assert
>
> using
>
> noexcept
>
> non member begin/end
>
> range for
>
> delegate constructor
>
> inherit constructor
>
> nullptr
>
> scoped enum
>
> user define literal
>
> raw string
>
> delete function
>
> default function
>
> override
>
> final

1. static_assert

▌ 핵심 개념

컴파일 시간에 특정 표현식이 참이 아닐 경우 컴파일을 멈추도록 하는 문법.

▌ 기본 예제

```
int main()
{
    static_assert(sizeof(void*) == 8, "error, this code can use only 64 bit!");
    static_assert(sizeof(void*) == 8); // C++17 부터 사용가능
}
```

▌ 주의 사항

컴파일 시간의 유효성을 조사하는 것이므로 실행시간 변경되는 변수 값을 조사할 수는 없습니다.

```
int main()
{
    int n = 10;
    static_assert(n == 10, "some message"); // error. n 은 컴파일 시간 상수가 아닙니다.
}
```

▌ assert vs static_assert

C언어 시절부터 사용되던 assert()를 사용하면 실행시간에 특정 변수 값의 유효성을 조사할 수 있습니다. 특히, assert()는 함수 인자의 유효성을 확인 하는 용도로 많이 사용 되었습니다.

```
void foo(int age)
{
    // 함수 인자 값의 유효성을 확인 합니다.
    assert(age > 0);
```

```
}
```

assert() 와 달리 C++11에서 추가된 static_assert()를 사용하면 실행시간이 아닌, 컴파일 시간에 특정 표현식의 유효성을 조사 할 수 있습니다. static_assert()의 사용법은 다음과 같습니다.

```
static_assert( compile-time-boolean-expression , "message"); // C++11 부터
static_assert( compile-time-boolean-expression );           // C++17 부터 사용가능
```

static_assert는

① 1번째 인자로 전달된 표현식이 거짓일 경우 컴파일시에 출력창에 "message"를 출력하고 컴파일을 멈추게 됩니다.

② 1번째 인자로 전달된 표현식은 반드시 컴파일 시간에 연산 가능한 상수식 이어야 합니다.

③ static_assert 는 함수 안에서 사용해도 되고, 함수 외부에서 사용 할 수도 있습니다.

```
// 함수 외부에 있어도 되고
static_assert(sizeof(long) == 8, "error, long size is too small");     // ok

int main()
{
    // 함수 안에 있어도 됩니다.
    static_assert(sizeof(long) == 8, "error, long size is too small"); // ok

    // 반드시 1번째 인자는 컴파일 시간 계산되는 boolean 표현식이어야 합니다.
    int n = 10;
    const int c1 = n;
    const int c2 = 10;
    static_assert(n  == 10, "message"); // error, n은 컴파일 시간에 조사할 수 없습니다.
    static_assert(c1 == 10, "message"); // error, c1은 컴파일 시간에조사할 수 없습니다.
    static_assert(c2 == 10, "message"); // ok, c1 은 컴파일시간에 조사 할 수 있습니다.
}
```

▌ static_assert 활용

static_assert 는 type traits 와 같이 사용하는 경우가 많이 있습니다.

➢ 구조체에 padding이 있는 경우 컴파일을 멈추는 코드

```
#pragma pack(1) // 이 부분을 주석으로 막고 컴파일 해보세요
struct Packet
{
    char m1;
    int  m2;
};
static_assert(  sizeof(Packet) == sizeof(char) + sizeof(int),
                "unexpected padding in Packet");
```

➢ 클래스 템플릿의 인자가 default 생성자가 없을 경우 컴파일을 멈추는 코드

```
template <class T> struct NODE
{
    static_assert(std::is_default_constructible<T>::value,
        "NODE requires default-constructible elements");
    // ……
};
struct Point
{
    Point(int a, int b) {}
};
int main()
{
    NODE<Point> n; // error
}
```

➢ 예외를 던지지 않은 Swap 만들기

```
class Test {};
template <class T> void Swap(T& a, T& b)
{
    static_assert( std::is_nothrow_move_constructible<T>::value
                && std::is_nothrow_move_assignable<T>::value , "Swap may throw");
    auto c = move(b);
    b = move(a);
    a = move(c);
}
int main()
{
    Test t1, t2;
    Swap(t1, t2);
```

```
}
```

[참고] move의 개념과 nothrow는 뒤장에서 자세히 배우게 됩니다.

2. using

▌ 핵심 개념

타입 및 template의 별칭(alias)를 만드는 문법입니다. template alias 라고도 합니다.

▌ 기본 예제

```
template<typename T> using SET = set<T, greater<T>>;

int main()
{
    SET<int>    s1; // set<int,    greater<int>>
    SET<double> s2; // set<double, greater<double>>
}
```

▌ 특징

typedef는 타입의 별칭(alias)만 만들 수 있지만 using을 사용하면 타입 뿐 아니라 템플릿의 별칭도 만들 수 있습니다.

▌ typedef vs using

C++11의 새로운 문법인 using을 사용하면 typedef와 유사하게 기존 타입에 대한 별칭을 만들 수 있습니다.

➢ typedef 를 사용하는 코드

```
typedef int DWORD;
typedef void(*PF)(int);

int main()
{
    DWORD n = 0;
    PF f = 0;
}
```

➢ C++11 의 using 을 사용하는 코드

```
using DWORD = int;
using PF = void(*)(int);

int main()
{
    DWORD n = 0;
    PF f = 0;
}
```

기존의 typedef 과 비교해서 using의 장점은

"typedef는 타입의 별칭만 만들 수 있지만, using은 타입 뿐 아니라 template의 별칭도 만들 수 있다"

는 점입니다.

STL 의 set 사용시 요소 비교를 less가 아닌 greater를 사용하는 SET을 생각해 봅시다. int, double 버전의 set을 별칭(alias)를 만들어서 사용할 경우 typedef를 사용하면 2개의 별칭을 따로 만들어야 합니다.

```
// typedef를 사용할 경우 int, double 버전 각각에 대해 별칭을 따로 제공해야 합니다.
typedef set<int,    greater<int>>    SETI;
typedef set<double, greater<double>> SETD;

int main()
{
    SETI s1;
    SETD s2;
}
```

[참고] greater에 익숙하지 않은 분들은 이 책의 마지막 부분에 있는 "함수객체" 항목을 참고하시기 바랍니다.

하지만, using은 템플릿의 별칭(alias)을 만들 수 있기 때문에 하나의 별칭으로 만들어서 사용 할 수 있습니다.

```
template<typename T> using SET = set<T, greater<T>>;

int main()
{
    SET<int>    s1;
    SET<double> s2;
}
```

▌ 활용 및 장점

using을 활용하면 템플릿 코드에서 typename, ::type, ::value 등의 복잡한 표현식을 제거할 수 있습니다.

C++11 에서는 type traits 사용시 다음과 같이 사용 했습니다.

```
template<typename T> void foo(T a)
{
    bool b = is_pointer<T>::value;      // ::value 가 복잡해 보입니다
    typename remove_pointer<T>::type n; // typename, ::type이 복잡해 보입니다.
}
```

하지만, C++14 에서는 using을 사용해서 다음과 같은 별칭을 제공하고 있습니다.

```
// C++14 에서는 using을 사용해서 다음과 같은 별칭(alias)를 제공하고 있습니다.
template<typename T> using is_pointer_t     = is_pointer<T>::value;
template<typename T> using remove_pointer_t = typename remove_pointer<T>::type;

// 앞에서 살펴본 foo()함수는 다음과 같이 다시 만들 수 있습니다.
template<typename T> void foo(T a)
{
    bool b = is_pointer_t<T>; // ::value 가 없습니다.
    remove_pointer_t<T> n;    // typename, ::type이 필요 없습니다.
}
```

3. noexcept

▌ 핵심 개념

함수가 예외가 발생하지 않음을 컴파일러에게 알려주는 문법입니다.

▌ 기본 예제

```
void foo() throw();  // C++98/03 style
void foo() noexcept; // C++11/14 style
```

▌ 특징

특정 함수가 예외가 없음을 보장할 때, 컴파일러는 더욱 효율적인 최적화를 수행할 수 있습니다. 또한, move_if_noexcept() 등의 함수를 사용해서 보다 효율적으로 동작하는 알고리즘을 작성할 수 있습니다.

▌ 상세 설명

C++98/03 시절에는 특정 함수가 예외가 발생하지 않음을 알려 주기 위해 throw()를 사용했습니다. 또한, 함수가 어떤 종류의 예외가 던지는 가도 throw()를 통해서 알려줄 수 있었습니다.

```
void f1();                  // 예외가 발생 할 수도 있고, 발생하지 않을 수도 있습니다.
void f2() throw();          // 예외가 발생 하지 않습니다
void f3() throw(std::bad_alloc); // bad_alloc 예외가 발생 할 수 있습니다.
```

하지만, C++ 표준 위원회는 예외의 종류 보다도 예외가 있다는 것과 없다는 것을 구별하는 것이 중요하다는 것을 알게 되었고, C++11/14 에서는 함수가 예외가 없을 경우 noexcept를 통해서 예외가 없음을 알려주는 문법을 새로 만들게 되었습니다.

```
void f1();          // 예외가 발생 할 수도 있고, 발생하지 않을 수도 있습니다.
void f2() noexcept; // 예외가 발생 하지 않습니다
```

▌ 활용

C++ 표준 type traits 중 is_nothrow_..... 를 사용하면 특정 타입이 생성자, 소멸자, 복사 생성자, move 생성자 등이 예외를 던지는 지를 조사 할 수 있습니다.

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

class Test
{
public:
    Test() noexcept {}  // noexcept를 제거하고 실행해 보세요.
};

int main()
{
    cout << is_nothrow_default_constructible_v<Test> << endl;
}
```

또한, move_if_noexcept() 함수를 사용하면 move 생성자에 예외가 없을 때만 move 생성자를 사용하고 move 생성자에 예외가 있다면 복사 생성자를 사용하게 할 수 도 있습니다.

```
int main()
{
    Test t1;
    Test t2 = move_if_noexcept(t1); // move 생성자에 예외가 없다면 move로
                                    // 예외가 있다면 복사 생성자로 복사 합니다.
}
```

[참고] 이 예제는 아직 배우지 않은 move 개념을 사용하고 있습니다. move 생성자는 이책의 뒤부분에서 자세히 다루고 있습니다. 지금은 이 예제가 이해 되지 않아도 상관 없습니다.

4. 멤버가 아닌 begin/end 함수

▌ 핵심 개념

STL의 반복자를 꺼낼 때 멤버 함수가 아닌 일반 함수 begin/end 를 사용 하라는 개념.

▌ 기본 예제

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

    // C++98/03 방식의 반복자 꺼내기
    vector<int>::iterator p = v.begin();

    // C++11/14 에서 권장되는 방식. 장점은 배열도 동일방식으로 사용 가능.
    auto p1 = begin(v); // STL 컨테이너
    auto p2 = begin(x); // 일반 배열
}
```

▌ 장점

일반 함수 begin/end를 사용할 경우 STL 컨테이너 뿐 아니라 일반 배열도 사용 가능한 일반화 함수를 만들 수 있습니다.

▌ 기존 방식의 문제점

C++98/03 문법에서는 STL 컨테이너에서 반복자 꺼내기 위해서는 멤버 함수 begin/end를 사용했습니다.

```
vector<int>::iterator p1 = v.begin();
vector<int>::iterator p2 = v.end();
```

이 방식의 문제점은 STL 컨테이너가 아닌 일반 배열도 사용가능한 일반화 함수를 만들기가 복잡해 진다는 점입니다.

다양한 STL 컨테이너의 모든 요소를 출력하는 show() 함수를 생각해 봅시다.

```
template<typename T> void show(T& c)
{
    typename T::iterator p = c.begin();

    while (p != c.end())
    {
        cout << *p << ' ';
        ++p;
    }
    cout << endl;
}
int main()
{
    int x[10] { 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v{ 1,2,3,4,5,6,7,8,9,10 };

    show(v);
    show(x); // error. 배열에는 begin/end 멤버 함수가 없습니다.
}
```

[참고] 위 코드에서 사용하는 vector의 초기화 기술은 이 책의 뒤쪽에 나오는 "객체 초기화" 항목에서 다루게 됩니다.

위 코드 에서 만든 show()는 begin/end 멤버 함수를 가진 STL의 대부분의 컨테이너에 적용될 수 있지만, 일반 배열을 인자로 전달 할 수는 없습니다.

▌ 해결책 1. type_traits 와 함수 오버로딩.

위 코드의 문제점은 is_array<> 와 배열의 크기를 조사하는 extent<>를 사용하면 해결 할 수 있습니다

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

// 배열이 아닌 경우.
template<typename T> void show_imp(T& c, false_type)
{
```

```
    typename T::iterator p = c.begin();

    while (p != c.end())
    {
        cout << *p << ' ';
        ++p;
    }
    cout << endl;
}

// 배열인 경우
template<typename T> void show_imp(T& c, true_type)
{
    auto p = c;
    while (p != c + extent<T, 0>::value )
    {
        cout << *p << ' ';
        ++p;
    }
    cout << endl;
}

template<typename T> void show(T& c)
{
    show_imp(c, is_array<T>());
}
int main()
{
    int x[10]{ 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v{ 1,2,3,4,5,6,7,8,9,10 };
    show(v);
    show(x);
}
```

이 경우의 단점은 STL 컨테이너와 배열을 위해서 각각 함수를 따로 만들어야(show_imp가 2개) 한다는 점입니다.

▌ 해결책 2. 멤버가 아닌 일반 함수 begin/end를 사용.

C++11/14에서는 멤버 함수가 아닌 begin/end 함수를 제공합니다. STL을 컨테이너를 인자로 받는 버전과 배열을 인자로 받는 버전을 이미 C++ 표준에서 제공하고 있습니다.

➢ C++11/14 표준이 제공하는 일반 함수 begin/end

```
// 컨테이너 버전의 begin/end - 멤버 함수 begin/end를 다시 호출해서 반복자를 리턴
template<typename T> inline auto begin(T& c) { return c.begin(); }
template<typename T> inline auto end(T& c)   { return c.begin(); }

// 배열을 인자로 받아서 시작 주소와 마지막 다음 주소를 리턴 하는 begin/end
template<typename T, int N> inline auto begin(T(&ar)[N]) { return ar; }
template<typename T, int N> inline auto end(T(&ar)[N]) { return ar+N; }
```

[참고] 인자의 const reference버전도 별도로 제공 됩니다. C++ 표준 문서를 참고 하세요

멤버가 아닌 일반 함수 begin/end를 사용하면 show 함수를 하나만 만들어도 배열과 STL 컨테이너를 모두 사용할 수 있도록 할 수 있습니다.

➢ 배열과 STL 컨테이너를 모두 사용 할 수 있는 show.

```
template<typename T> void show(T& c)
{
    auto p = begin(c);
    while (p != end(c))
    {
        cout << *p << ' ';
        ++p;
    }
    cout << endl;
}
int main()
{
    int x[10]{ 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v{ 1,2,3,4,5,6,7,8,9,10 };
    show(v);
    show(x);
}
```

해결책 3. range for

컨테이너를 열거 할 때 반복자가 아닌 range-for를 사용하면 훨씬 간결하게 show를 만들 수 있습니다.

```
template<typename T> void show(T& c)
{
    for (auto& n : c)
        cout << c << ' ';

    cout << endl;
}
```

그렇다면, 반복자를 사용한 열거 방식과 range_for를 사용한 열거 방식 중 어떤 방식을 사용하는 것이 좋을 까요 ?

다음 페이지를 참고해 보세요.

5. range-for

▌ 핵심 개념

C++11 에서 새롭게 소개 되는 for문. java, C#의 foreach 문과 유사한 문법.

▌ 기본 예제

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

    for (auto n : x)
        cout << n << ' ';
}
```

▌ 특징 및 장점

배열 뿐 아니라 STL의 대부분의 컨테이너도 사용 가능하며, 컨테이너(배열)의 크기가 변경되어도 코드를 수정할 필요가 없습니다.

▌ 상세 설명

C언어에서 사용하던 전통적인 for 문은 배열(또는 컨테이너)의 크기가 변경될 경우 for 문의 코드가 변경해야 합니다. 하지만, range-for문을 사용하면 크기를 지정할 필요가 없습니다.

➢ 기존의 사용하던 for 문

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

    for (int i = 0; i < 5; i++)
        cout << x[i] << ' ';
}
```

➢ C++11 의 range-for 문

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

    for (auto n : x)
        cout << n << ' ';
}
```

[참고] 컨테이너(배열)의 일부 요소만 출력하려면 range-for 가 아닌 전통적인 for 문을 사용해야 합니다.

range-for 문의 원리

배열 또는 STL의 컨테이너가 아닌 사용자가 만든 컨테이너(List)등에 range-for 문을 사용 하려면 어떻게 해야 할까요 ?

range-for문의 원리는 다음과 같습니다. 아래 코드에서 사용자가 좌측 코드 처럼 만들면 컴파일러는 컴파일시에 우측의 코드로 변경하게 됩니다.

➢ 사용자가 만든 range-for

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

    for (auto n : x)
        cout << n << ' ';
}
```

➢ 컴파일러가 생성한 코드

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

    for (auto p = begin(x); p != end(x); ++p)
    {
        auto n = *p;

        cout << n << ' ';
    }
}
```

결국 사용자가 만든 타입을 range-for문으로 사용할 수 있게 하려면

① **사용자 타입을 인자로 받는 begin/end를 일반 함수로 제공 하거나**

② **사용자 타입안에 멤버 함수로 begin/end를 제공**

하면 됩니다.

➢ range-for를 지원하는 사용자 정의 타입 만들기

```
struct Point3D
{
    int x=1, y=2, z=3;

    int* begin() { return &x; }
    int* end()   { return &z+1; }
};
int main()
{
    Point3D p;

    for (auto n : p)
        cout << n << ' ';
```

```
}
```

6. delegating constructor(위임 생성자)

▌ 핵심 개념

생성자에서 다른 생성자를 호출할 때 사용하는 문법. 생성자의 초기화 리스트 영역에서 다른 생성자를 호출하는 문법.

▌ 기본 예제

```
class Point
{
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) {}

    Point() : Point(0, 0) // delegating constructor
    {
    }
};
```

▌ 참고 사항

C++98/03 문법에서는 특정 생성자에서 다른 생성자를 호출할 수 있는 일반적인 방법이 없습니다. 특히, 임시객체를 만드는 표현을 다른 생성자를 호출하는 것으로 혼동하는 경우가 있으니 주의할 필요가 있습니다.

```
class Point
{
public:
    int x, y;
    Point(int a, int b) : x(a), y(b) {}

    Point()
    {
        Point(0, 0);    // bug, 다른 생성자를 호출하는 표현이 아닙니다.
                        // 임시객체를 만드는 표현입니다.
    }
};
int main()
```

```
{
    Point p;
    cout << p.x << endl; // 0이 아니라 쓰레기값이 출력됩니다.
}
```

기존의 C++98/03 문법에서도 placement new를 사용하면 생성자에서 다른 생성자를 호출 할 수도 있습니다. 하지만 코드 가독성이나 다양한 측면에서 C++11에서 새로 나온 delegating constructor 문법을 사용하는 것이 좋습니다.

```
class Point
{
public:
    int x, y;
    Point(int a, int b) : x(a), y(b) {}

    Point()
    {
        new(this) Point(0, 0);
    }
};
```

7. inherit constructor

▌ 핵심 개념

기반 클래스의 생성자를 파생 클래스에서도 사용 할 수 있도록 상속 받는 문법

▌ 기본 예제

```
struct Base
{
    Base(int a) {}
};
struct Derived : public Base
{
    using Base::Base;  // 기반 클래스의 생성자를 상속 받습니다.
};
int main()
{
    Derived d(5); // ok. 인자가 1개인 생성자를 Base로 부터 상속받았습니다.
}
```

▌ 상세 설명

C++에서는 파생 클래스의 객체를 생성하면 기반 클래스의 생성자도 호출됩니다. 이때, 기반 클래스의 생성자는 기본적으로는 항상 인자가 없는 디폴트 생성자가 사용 됩니다. 만약, 기반 클래스에 디폴트 생성자가 없고, 파생 클래스에서 기반 클래스의 생성자를 명시적으로 호출하지 않으면 파생 클래스의 객체 생성시 에러가 발생합니다.

```
struct Base
{
    Base(int a) {}
};
struct Derived : public Base
{
};
```

```
int main()
{
    Derived d; // error. Base에 디폴트 생성자가 없습니다.
}
```

이 경우의 일반적인 해결책은 파생 클래스에서 인자를 하나 받은 후, 초기화 리스트에서 기반 클래스의 생성자를 명시적으로 호출하면 됩니다.

```
struct Base
{
    Base(int a) {}
};
struct Derived : public Base
{
    Derived(int a) : Base(a) {} // 기반 클래스의 생성자를 명시적으로 호출합니다
};
int main()
{
    Derived d(5); // ok. 결국 5는 Base의 생성자로 전달됩니다.
}
```

이 경우, Derived의 생성자는 인자로 전달 받은 값을 Base로 전달하는 역할만 하고 있습니다. 이 경우는 Derived 의 생성자를 별도로 만들지 말고, Base가 가진 생성자를 Derived로 상속 받으면 됩니다.

주의 할 점은, Base의 생성자를 상속 받으면 컴파일러에 의한 디폴트 생성자는 제공되지 않는 다는 점입니다.

```
struct Base
{
    Base(int a) {}
};
struct Derived : public Base
{
    using Base::Base;  // 기반 클래스의 생성자를 상속 받습니다.
};
int main()
{
    Derived d1(10); // ok
    Derived d2;     // error. 해결책은 사용자가 Derived 또는 Base에
                    // 인자가 없는 생성자를 제공해야 합니다.
}
```

유사한 문법

C++에서는 기반 클래스가 가진 멤버 함수와 동일한 이름의 멤버 함수를 파생 클래스에서 만들 경우, 기반 클래스의 함수는 상속되지 않습니다. 즉, 함수 인자가 달라도 기반 클래스의 함수를 사용할 수 없습니다.

```
struct Base
{
    void foo(double a)     { cout << "double" << endl; }
    void foo(int a, int b) { cout << "int, int" << endl; }
};
struct Derived : public Base
{
    // 기반 클래스의 멤버 함수인 foo는 상속되지 않습니다.
    void foo(int n)        { cout << "int" << endl; }
};
int main()
{
    Derived d;
    d.foo(3.4);  // double 이 아니고 int가 출력됩니다.
    d.foo(1, 2); // error
}
```

이 경우에 using을 사용하면 기반 클래스의 foo 함수를 상속 받을 수 있습니다.

```
struct Derived : public Base
{
    using Base::foo; // 이제, 기반 클래스의 멤버 함수인 foo가 상속 됩니다.

    void foo(int n) { cout << "int" << endl; }
};
int main()
{
    Derived d;
    d.foo(3.4);  // double
    d.foo(1, 2); // ok. int, int 가 출력됩니다.
}
```

8. nullptr

C언어에서의 NULL

C언어 에서는 NULL을 "(void*)0" 로 정의 해서 사용하곤 했습니다. 하지만, C++에서는 void*가 다른타입의 포인터로 암시적 형 변환 될 수 없기 때문에 문제가 될 수 습니다.

```
void foo(int n)   { cout << "foo(int)" << endl; }    // 1
void foo(void* n) { cout << "foo(void*)" << endl; }  // 2

void goo(char* n) { cout << "goo(char*)" << endl; }

int main()
{
    foo(0);         // foo(int)
    foo((void*)0);  // foo(void*)

    // C 방식의 NULL
#define NULL (void*)0

    foo(0);     // 1
    foo(NULL);  // 2
    goo(NULL);  // C에서는 ok..
                // C++ error. "void* => 다른 타입*"으로 암시적 변환 되지 않습니다.
}
```

C++언어에서의 NULL

C++에서는 NULL을 0으로 정의해서 사용합니다.

```
#ifdef __cplusplus
    #define NULL    0       // C++
#else
    #define NULL (void*)0   // C
#endif

void foo(int n)   { cout << "foo(int)" << endl; }
void foo(void* n) { cout << "foo(void*)" << endl; }
void goo(char* n) { cout << "goo(char*)" << endl; }
```

```
int main()
{

    goo(NULL); // C++이라도 아무 문제가 없습니다.
    foo(NULL); // 하지만, 이 경우 void* 버전이 아닌 int 버전이 호출됩니다.
}
```

결국, 기존의 C++에서는 포인터 타입의 0이 없습니다. C++11에서는 포인터 0을 나타내는 nullptr을 추가 했습니다.

▌ C++11 nullptr

C++11 에서는 포인터 0을 나타내는 nullptr 을 새롭게 도입했습니다.

nullptr 은 모든 타입의 포인터 0을 의미 합니다.

int로의 암시적변환은 허용하지 않지만, bool로의 direct initialization은 허용합니다.

```
int main()
{
    int* p = nullptr; // ok

    // 정수로 변환될수 없습니다.
    int  n1(nullptr);  // error
    int  n2 = nullptr; // error

    // bool로 암시적 변환을 허용합니다. ( direct initialization 만 허용됩니다.)
    bool b1(nullptr); // ok
    bool b2 = nullptr;// error
}
```

▌ nullptr_t

nullptr의 타입은 nullptr_t 입니다.

```
int main()
```

```
{
    cout << typeid(nullptr).name() << endl;
}
```

▌ nullptr 만들기

template과 변환 연산자를 사용하면 nullptr과 유사하게 만들 수 있습니다.

```
struct nullptr_t
{
    template<typename T> operator T*() { return 0; }
};
nullptr_t nullptr;

int main()
{
    int*  p1 = nullptr; // ok
    char* p2 = nullptr; // ok

    int n = nullptr; // error
}
```

9. scoped enum

▌ 핵심 개념 .

기존의 enum은 타입 이름 없이 사용 가능 합니다.(unscoped enum), 하지만 새로운 enum class 문법은 반드시 타입이름을 표기해야 합니다.

▌ 기본 예제

```
// C++98/03 enum - unscoped enum
enum Color { red = 1, blue = 2};

// C++11/14 enum - unscoped enum
enum class Shape { rect = 1, circle = 2};

int main()
{
    // C++98/03 unscoped enum
    int n1 = Color::red;// ok.
    int n2 = red;       // ok. Color 없이 사용가능합니다.
    int red = 10;       // ok. 하지만 혼란스럽습니다.

    // C++11/14 scoped enum
    int n3 = rect;        // error. Shape가 필요 합니다.
    int n4 = Shape::rect; // error. int 타입으로 암시적 형변환 될 수 없습니다
    Shape s = Shape::rect;// ok
}
```

▌ enum class와 underlying type

C++98/03의 enum은 요소의 타입을 지정할 수 없지만 C++11의 enum class를 사용하면 타입을 지정할 수 있습니다. 또한, underlying_type◇ traits를 사용하면 타입을 조사할 수 있습니다.

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

enum class Color1 : int  { red = 1, blue = 2 };
```

```
enum class Color2 : char { red = 1, blue = 2 };

int main()
{
    cout << typeid(underlying_type_t<Color1>).name() << endl;
    cout << typeid(underlying_type_t<Color2>).name() << endl;
}
```

10. user define literal

▌ 핵심 개념

literal에 접미사를 만드는 문법. 일반 개발자가 만들 때는 _로 시작되어야 합니다.

▌ 기본 예제

```
class Meter
{
    int value;
public:
    explicit Meter(int n) : value(n) {}
};

Meter operator""_m(unsigned long long n)
{
    return Meter(static_cast<int>(n));
}

int main()
{
    Meter m = 3_m;
}
```

▌ 함수 인자

literal의 종류에 따라 operator""() 함수의 함수 인자 타입이 정해져 있습니다.

정수 리터럴	리터럴 연산자에서 unsigned long long 인수나 const char* 인수로 받음.
부동소수점 리터럴	리터럴 연산자에서 long double 인수나 const char* 인수로 받음.
문자열 리터럴	리터럴 연산자에서 (const char*, size_t) 쌍을 인수로 받음.

문자 리터럴	리터럴 연산자에서 char 인수로 받음.

predefine literal

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

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

int main()
{
    foo("hello"s);

    complex<int> c1 = 2i; // 0 + 2i
    cout << c1 << endl;

    seconds s = 3h + 3min + 20s;
    cout << s.count() << endl;
}
```

11. raw string

▌ 핵심 개념 .

back slash(“₩”)를 특수문자로 처리하지 않고 일반 문자로 처리하는 문자열

R”(문자열)” 의 형태로 사용합니다.

▌ 기본 예제

```
int main()
{
    string s1 = "hello\tworld";
    string s2 = R"(hello\nworld)";

    cout << s1 << endl; // "hello   world"
    cout << s2 << endl; // "hello\nworld"
}
```

▌ 사용자 정의 구분자 추가

“와 (사이에 사용자 정의 구문자를 추가 할 수 있고, encoding prefix를 추가할 수 있습니다.

```
// 문자열 안에 " 를 표현하고 싶을때
string s3 = R"("quoted" "string")"; // 그냥 적으면 됩니다.

// 문자열 안에 )" 를 표현하고 싶을때
string s4 = R"(foo"()")";           // error.

//  사용자 정의 구분자를 추가하면 됩니다. :  "**" 사용
string s5 = R"**(foo"()")**";   // ok

// encoding prefix를 추가 할 수 있습니다.
string s6 = u8R"(fdfdfa)";
```

12. delete function

▌ 핵심 개념

함수 또는 멤버 함수는 삭제 하는 문법. 삭제된 함수를 사용하려고 하면 컴파일 시간에 에러 발생.

▌ 기본 예제

```
class People
{
public:
    People() {}
    People(const People& p) = delete; // 디폴트 복사 생성자를 삭제 합니다
};
int main()
{
    People p1;
    People p2 = p1; // error. 복사 생성자가 삭제 되었습니다.
}
```

▌ 복사 및 대입 금지 기술

C++에서는 사용자가 복사 생성자/대입연산자를 제공하지 않아도 컴파일러가 제공해 줍니다.

```
class People
{
public:
    // 복사 생성자와 대입연산자가 없지만 컴파일러가 제공해 줍니다.
};
int main()
{
    People p1;
    People p2 = p1; // ok. 디폴트 복사 생성자가 사용됩니다.
    p2 = p1;        // ok. 디폴트 대입 연산자가 사용됩니다.
}
```

Singleton 등의 기법을 사용 할 때는 복사와 대입을 막아야 합니다. C++98/03 시절에는 private 영역에 복사 생성자(대입연산자)의 선언부만 제공해서 외부에서의 복사와 대입을 막았습니다. 하지만, C++11 부터는 delete function 을 사용하면 됩니다.

➢ C++98/03 에서의 복사 금지 기술

```
class People
{
public:
    People() {}

private:
    // 복사 생성자와 대입연산자를
    // private 영역에 선언만 합니다.
    People(const People& p);
    void operator=(const People& p);
};
```

➢ C++11 delete function 을 사용한 복사 금지

```
class People
{
public:
    People() {}

    // 복사 생성자와 대입연산자를
    // 삭제 합니다.
    People(const People& p) = delete;
    void operator=(const People& p) = delete;
};
```

▌ 함수를 제공하지 않은 것, 선언만 제공하는 것, 함수를 삭제 하는 것

C/C++언어는 전통적으로 표준 타입은 서로 암시적 형변환이 가능합니다. 아래 코드를 생각해 봅시다.

```
// 2개 정수의 최대 공약수를 구하는 함수 입니다
int gcd(int a, int b)
{
    return b != 0 ? gcd(b, a % b) : a;
}
int main()
{
    gcd(10, 4);     // ok
    gcd(10.2, 4.3); // ok. double 버전의 함수는 없지만 double은 int로
                    // 암시적 형 변환 되므로 gcd(int, int) 함수가 호출됩니다.
}
```

결국 위 코드의 핵심은 **"인자의 타입이 double 인 함수를 제공하지 않았지만, 인자가 암시적 형 변환 되어서 호출 가능한 동일 이름의 함수가 있으므로 int 버전의 함수가 사용된다"** 는 점입니다.

하지만, 이경우 오히려 버그의 원인이 되므로 호출을 막는 것이 좋습니다. 아래 코드를 생각해 봅시다.

```
int gcd(int a, int b)
{
    return b != 0 ? gcd(b, a % b) : a;
}
double gcd(double, double); // double 버전의 함수를 선언만 제공합니다.

int main()
{
    gcd(10, 4);     // ok
    gcd(10.2, 4.3); // double 버전의 함수선언이 있으므로 컴파일은 가능합니다.
                    // 하지만, 구현부가 없으므로 link error가 발생합니다.
}
```

위 코드는 double 버전 함수의 선언이 있으므로 int 버전을 사용하지는 않습니다. 하지만, double 버전의 구현이 없으므로 링크 시간에 에러가 발생합니다.

즉, 암시적 형변환에 호출을 막는 것은 성공했습니다. 하지만, 문제는 에러가 컴파일 시간이 아니라 링크 시간에 발생한다는 점입니다. 링크하지 않고, object 파일만 만들 때는 에러는 발견할 수가 없게 됩니다. 즉, 위 코드를 아래 처럼 컴파일하면 에러가 발생하지 않습니다.

```
g++ delfunc.cpp -c    // g++ 사용시
cl delfunc.cpp  /c    // cl 사용시
```

double버전 사용시 링크 시간 에러가 아닌 컴파일 에러가 나오게 하려면 함수를 삭제 하면 됩니다.

```
int gcd(int a, int b)
{
    return b != 0 ? gcd(b, a % b) : a;
}
double gcd(double, double) = delete;
int main()
{
    gcd(10, 4);     // ok
    gcd(10.2, 4.3); // compile-error. 삭제된 함수를 호출하려고 합니다.
}
```

▌ 핵심 정리

결국 정리를 하면,

double 버전 함수를 제공하지 않은 경우	암시적 형변환을 통해 호출가능한 동일 이름의 다른 함수가 있으면 호출됩니다.
double 버전 함수를 선언만 제공한 경우	링크 시간에 에러가 발생합니다.
double 버전 함수를 삭제한 경우	컴파일 시간 에러가 발생합니다.

13. default function

▌ 핵심 개념 .

컴파일러에게 특정 함수의 default 구현을 제공해 달라는 문법.

▌ 기본 예제

```
class Point
{
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) {}

    // 각 함수의 기본 구현을 제공해 달라는 문법
    Point() = default;
    Point(const Point&) = default;
    Point(Point&&) = default;       // move 생성자
    Point& operator=(const Point& p) = default;
    Point& operator=(Point&& p) = default;
};
```

[참고] move 생성자는 이 책의 뒷부분에서 자세히 배우게 됩니다.

▌ 상세 설명

클래스를 만들 때 사용자가 생성자를 하나도 제공하지 않으면 컴파일러는 인자 없는 생성자(default 생성자)를 하나 제공합니다. 하지만 사용자가 생성자를 하나라도 제공하면 컴파일러는 디폴트 생성자를 제공하지 않습니다.

```
class Point
{
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) {}
};

int main()
{
    Point p1;       // error. 디폴트 생성자가 필요 합니다.
```

```
    Point p2(0, 0); // ok.
}
```

이 경우 디폴트 생성자를 제공할 때, 사용자가 직접 만들어도 되지만, "default function"문법을 사용해서 컴파일러가 제공하도록 할 수 도 있습니다

➢ 사용자가 직접 생성자를 제공한 경우

```
class Point
{
    int x, y;
public:
    Point(int a, int b):x(a),y(b){}
    Point() {}
};
```

➢ C++11 default function 문법을 사용한 경우.

```
class Point
{
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) {}
    Point() = default;
};
```

사용자가 직접 아무 일도 하지 않은 생성자를 만드는 것고, default function 문법을 사용해서 디폴트 생성자를 제공하는 것에는 약간의 차이가 있습니다.

객체를 만들 때 value initialization을 사용 할 경우, 사용자가 직접 생성자는 제공한 경우 초기화 되지 않은 멤버는 쓰레기 값을 가지게 되지만, default function을 사용해서 생성자를 제공한 경우에는 모든 멤버가 자동으로 0으로 초기화 됩니다.

```
struct Point1
{
    int x, y;
    Point1() = default;
};
struct Point2
{
    int x, y;
    Point2() {}
};
int main()
{
    // {}로 초기화 하는 것을 value initialization 이라고 합니다.
    Point1 p1{}; // 모든 멤버가 0으로 초기화 됩니다.
    Point2 p2{}; // 모든 멤버는 초기화 되지 않습니다.

    cout << p1.x << endl; // 0
```

```
    cout << p2.x << endl; // 쓰레기 값
}
```

value initialization 에 대해서는 “객체 초기화” 항목에서 자세히 배우게 됩니다.

14. override

▌ 핵심 개념

가상 함수 재 정의 시에 버그를 줄이기 위해 override를 붙이는 문법입니다.

▌ 기본 예제

```
class Animal
{
public:
    virtual void Cry() {}
};
class Dog : public Animal
{
public:
    virtual void Cry() override {}
};
```

▌ 장점

가상 함수 재정의시 오타 등으로 인한 버그를 줄이기 위해서 만든 문법

▌ 가상 함수 재정의 시의 문제점

C++에서는 가상 함수 재정의시에 인자 타입이 다르거나, 함수 이름이 잘못된 경우 등 다양한 실수를 했을 때 컴파일 시간에 에러가 나지 않기 때문에 버그가 발생할 확률이 높습니다.

```
class Base
{
public:
    virtual void foo(double d) {}
    virtual void goo() const {}
    virtual void hoo() {}
    void koo() {}
};
class Derived : public Base
```

```
{
public:
    // 아래와 같은 실수를 했을 때, 컴파일러는 에러를 발생하지 않고
    // 새로운 함수를 만든 것으로 생각하게 됩니다.
    virtual void foo(int d) {}  // 인자의 타입이 잘못되었습니다.
    virtual void goo() {}       // 기반 클래스의 goo는 상수 함수 입니다.
    virtual void hooo() {}      // 함수 이름이 잘못되었습니다.
    virtual void koo() {}       // 기반 클래스의 koo 함수는 가상 함수가 아닙니다.
};
```

이와 같은 문제점을 해결하기 위해 가상 함수를 재정의 할 때는 함수 () 뒤에 override 를 붙이면, 기반 클래스의 함수 모양과 다를 경우 컴파일 시간에 에러를 발생하게 됩니다.

```
class Derived : public Base
{
public:
    // 컴파일 하면 아래 4개의 함수는 모두 error가 발생합니다.
    virtual void foo(int d) override {}
    virtual void goo()  override {}
    virtual void hooo() override {}
    virtual void koo()  override {}
};
```

▌ 가상함수 재정의시 override를 표시해도 되고 표시하지 않아도 됩니다.

C++11/14 에서 새로운 문법이 많이 추가 되었지만, 모든 C++ 컴파일러는 C++98/03 시절의 문법도 전부 지원 해야 합니다. C++98/03 에서는 override 문법이 없으므로, 가상 함수 재정의시에 override 키워드를 붙이지 않았습니다. 최신의 C++ 컴파일러를 사용해도 과거의 문법을 지원해야 하므로 가상 함수를 재정의 할 때 override를 붙여도 되고 붙이지 않아도 됩니다. 하지만, override를 사용하는 것이 버그를 줄이고 안전한 코드를 만들 수 있습니다.

```
class Derived : public Base
{
public:
    virtual void foo(int) override {}   // override를 붙여도 되고.
    virtual void goo()  const {}        // override를 붙이지 않아도 됩니다.
};
```

▌ 생각해 볼 문제 - 가상함수 재정의의 어려움

코드가 복잡한 경우 가상함수를 재정의시에 실수 하는 경우가 있습니다. 아래 코드의 실행결과가 “Derived foo” 가 출력 되도록 가상함수를 foo를 재정의해 해보세요. 단, override 키워드를 사용하지 말고 해보세요.

```
template<typename T> class Base
{
public:
    virtual void foo(const T a)
    {
        cout << "Base foo" << endl;
    }
};
class Derived : public Base<int*>
{
public:
    // foo 를 재정의해서 "Derived foo"라고 출력해 보세요.
};

int main()
{
    Base<int*>* p = new Derived;
    p->foo(0); // "Derived foo" 가 나와야 합니다.
}
```

override 키워드가 없을 경우 가상함수의 잘못되어도 컴파일 시간에 에러가 발생되지 않기 때문에 발견하기 어려운 버그가 발생할 수 있습니다.

15. final

▌ 핵심 개념

하위 클래스가 더 이상 가상함수를 재정의하지 못하게 하기 위한 문법.

클래스(구조체)뒤에 사용할 경우 해당 타입의 파생 클래스를 만들 수 없다.

▌ 기본 예제

```
// 가상함수 뒤에 final을 사용하는 경우
struct Base
{
    virtual void foo() {}
};
struct Derived : public Base
{
    virtual void foo() override final {}
};
struct Derived2 : public Derived
{
    virtual void foo() override {} // error
};

// 클래스 이름 뒤에 final을 사용하는 경우
class Test final    // Test 클래스의 파생 클래스를 만들 수 없습니다.
{
};
class DerivedTest : public Test // error.
{
};
```

▌ 참고

std::is_final 을 사용하면 특정 클래스가 final로 선언되어 있는지 조사할 수 있습니다.

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

```
class A {};
class B final {};

int main()
{
    cout << is_final<A>::value << endl;
    cout << is_final<B>::value << endl;
}
```

항목 1-03

C++17 기본 문법

주요 학습 내용

> if init
>
> if constexpr
>
> structure binding

1. if init

▌ 핵심 개념

C++17 부터는 if 또는 switch 문에 초기화 구문을 추가할 수 있습니다.

▌ 기본 예제

```
int foo()
{
    return 0;
}
int main()
{
    // 예전 스타일.
    int ret = foo();

    if (ret == 0)
    {
    }
    // C++17 스타일
    // if ( init 구문; 조건문 )
    if (int ret = foo(); ret == 0)
    {
        cout << "ret is 0" << endl;
    }

    // switch 문에도 초기화 구문을 추가할수 있습니다.
    switch (int n = foo(); n)
    {
    case 0: break;
    case 1: break;
    }
}
```

2. if constexpr (static if)

▌ 핵심 개념

실행 시간이 아닌 컴파일시간에 조건을 조사하는 if 문.

조건문 안에는 반드시 컴파일 시간에 조사 가능한 식이 놓여야 합니다.

▌ 기본 예제

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

template<typename T> void printv(T v)
{
    if constexpr (is_pointer<T>::value)
        cout << v << " : " << *v << endl;
    else
        cout << v << endl;
}
int main()
{
    int n = 10;
    printv(n);
    printv(&n);
}
```

3. structure binding

▌ 핵심 개념

구조체, 배열, pair, tuple 등에서 요소의 값에 접근하기 위한 문법.

▌ 기본 예제

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

struct Point
{
    int x;
    int y;
};
int main()
{
    Point pt = { 1,2 };

    auto  [a,  b] = pt;
    auto& [rx, ry] = pt;

    int x[2] = { 1,2 };
    auto[e1, e2] = x;

    pair<int, double> p(1, 3.4);
    auto[n, d] = p;

    tuple<int, short, double> t3(1, 2, 3.4);
    auto[a1, a2, a3] = t3;
}
```

항목 1-04

object initialization

주요 학습 내용

> C++98/03 초기화 기술의 문제점
>
> member initializer
>
> brace-init, brace-init-list
>
> direct vs copy initialization
>
> default vs value initialization
>
> initializer_list
>
> aggregate initialization

1. C++98/03 초기화 기술의 문제점

C++11 이전의 C++ 문법에서는 객체를 초기화하는데 몇가지 문제점이 있었습니다.

① 객체의 종류에 따라 초기화 방법이 다릅니다.

② 클래스안에 non static 멤버로 배열이 있을 경우 초기화 방법이 없습니다.

③ 배열을 동적으로 만들 때 초기화 할 수 없습니다

④ STL의 컨테이너를 초기화 하는 편리한 방법이 없습니다.

```
// 1. 객체의 종류(일반변수, 배열, 구조체등)에 따라 초기화 방법이 다르다.
int n1 = 0;
int n2(0);
int ar[3] = { 1,2,3 };
struct Point p = { 1,2 };

// 2. 클래스 안에 배열 멤버를 초기화 할수 없다.
class Test
{
    int x[10];
};

// 3. 동적으로 할당하는 배열을 초기화 할수 없다.
int* p = new int[3];

// 4. STL의 컨테이너를 초기화 할수 있는 편리한 방법이 없다.
vector<int> v;
v.push_back(1);
```

그래서, C++11에서는 몇가지 초기화 문법을 추가했습니다.

2. member initializer

Java 나 C# 등의 다른 언어처럼 C++도 이제 non-static member data를 선언할 때 초기값을 지정 할 수 있습니다.

쉬어 보이는 문법 이지만 주의 할점이 있습니다. 아래 코드를 실행해서 결과를 확인해 보세요.

```
int x = 0;

class Test
{
public:
    int data2 = ++x;    // member initializer

    Test() {}
    Test(int b) : data(b) {}
};

int main()
{
    Test t1;    // member initializer가 실행됩니다.
    Test t2(3); // member initializer가 실행되지 않습니다.

    cout << x << endl; // 1

    cout << t1.data << endl; // 1
    cout << t2.data << endl; // 3
}
```

3. brace-init, brace-init-list

▌ 핵심 개념 .

중괄호({ }) 를 사용해서 객체를 초기화 할수 있습니다. 일반 변수, 배열, 구조체, 클래스등 모든 종류의 객체를 하나의 방식으로 초기화 할 수 있습니다.

▌ 기본 예제

```
int main()
{
    // 일반 변수, 배열, 구조체, 클래스 를 모두 동일한 방식으로 초기화 합니다.
    int n{ 0 };
    int x[3]{ 1,2,3 };
    vector<int> v{ 1,2,3 };
}
```

▌ preventing narrow

중괄호 초기화를 사용하는 경우, data 손실이 발생하는 코드는 암시적 변환이 허용되지 않습니다.

```
int main()
{
    int n1 = 3.4;   // ok
    int n2( 3.4 );  // ok
    int n3{ 3.4 };  // error.
    char c{ 300 };  // error. 300은 1바이트에 담을수 없습니다.
}
```

▌ braced-init-list

중괄호 안에 한 개이상의 값을 가지고 있는 것을 braced-init-list 라고 합니다. braced-init-list로 구조체, 배열, 클래스 등을 초기화 할 수 있습니다.

```
struct Point
```

```
{
    int x, y;
};
class Complex
{
    int re, im;
public:
    Complex(int r, int i) : re(r), im(i) {}
};
int main()
{
    int x[2]{ 1,2 };
    Point p{ 1,2 };
    Complex c{ 1,2 };
}
```

4. direct initialization vs copy initialization

▌ 핵심 개념

direct initialization : 객체를 초기화 할 때 = 없이 () 또는 {}를 사용해서 초기화 하는 것.

copy initialization : 객체를 초기화 할 때 = 를 사용해서 초기화 하는 것

▌ 기본 코드

```
int main()
{
    int n1(0);      // 직접 초기화(direct)
    int n2 = 0;     // 복사 초기화(copy)

    int n3{ 0 };    // 직접 초기화
    int n4 = { 0 }; // 복사 초기화
}
```

▌ explicit 와 초기화 방법.

생성자 인자가 2개 이상 있어도 복사 초기화를 사용할 수 있습니다.

```
class Point
{
    int x, y;
public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}
};
int main()
{
    Point p1(0, 0);     // ok. 직접 초기화(direct initialization)
    Point p2 = (0, 0);  // error.
    Point p3{ 0,0 };    // ok. 직접 초기화(direct initialization)
    Point p4 = { 0,0 }; // ok. 복사 초기화(copy initialization)
}
```

생성자를 explicit로 만들면 복사 초기화를 사용할 수 없습니다. C++11 부터는 인자가 2개 이상인 생

성자도 explicit 로 만들 수 있습니다.

```
class Point
{
    int x, y;
public:
             Point(int a)        : x(a), y(0) {}
    explicit Point(int a, int b) : x(a), y(b) {}
};

int main()
{
    Point p1(0);         // ok. direct
    Point p2 = 0;        // ok. copy
    Point p3{ 0 };       // ok. direct
    Point p4 = { 0 };    // ok. copy

    Point p5{ 0,0 };     // ok. direct
    Point p6 = { 0, 0 };// error. copy. explicit 생성자
}
```

5. default initialization vs value initialization

▌ 핵심 개념

default initialization	초기화 방법을 사용하지 않고 객체를 생성하는 것
value initialization	비어 있는 brace-init을 사용해서 객체를 생성하는 것

▌ 기본 예제

```
int main()
{
    int n1{ 0 };    // direct initialization
    int n2 = { 0 }; // copy initialization

    int n3;         // default initialization.
    int n4{};       // value initialization

    cout << n3 << endl; // 쓰레기값
    cout << n4 << endl; // 0
}
```

▌ 생성자와 value 초기화

생성자의 제공 여부에 따라서 value initialization의 결과는 달라 집니다.

```
struct NoCtor
{
    int x, y;
};
struct DefaultCtor
{
    int x, y;
    DefaultCtor() = default;
};
struct UserCtor
```

```
{
    int x, y;
    UserCtor() {}
};

int main()
{
    NoCtor c1{};
    DefaultCtor c2{};
    UserCtor c3{};

    cout << c1.x << endl;   // 0
    cout << c2.x << endl;   // 0
    cout << c3.x << endl;   // 쓰레기값
}
```

6. initializer_list

▌ 핵심 개념

braced-init-list의 요소를 가르 키는 포인터의 쌍(pair).

포인터 2개 또는 포인터 하나와 개수로 구현되어 있습니다. begin(), end(), size()의 3가지 멤버함수를 제공합니다.

begin()/end()로 얻는 반복자는 상수 반복자 입니다.

▌ 기본 예제

```
int main()
{
    // VC++ : int *first, *last
    // g++  : int *first, count
    initializer_list<int> s = { 1,2,3,4,5,6,7,8,9,10 };

    auto p1 = s.begin(); // begin(s)
    auto p2 = s.end();   // end(s)

    // 반복자는 상수입니다.
    *p1 = 10; // error.
}
```

▌ 함수 인자와 initializer_list

함수 인자로 initializer_list를 사용 할 수 있습니다.

```
void foo( initializer_list<int> e )
{
    for (auto n : e)
        cout << n << " ";
    cout << endl;
}
int main()
{
    foo({ 1,2,3,4 });
```

```
    foo({ 1,2,3,4,5,6,7 });
    foo({});
}
```

▌ 생성자 인자와 initializer_list

객체 생성시 braced-init-list를 사용하면 initializer_list를 인자로 가지는 생성자가 우선시 됩니다.

```
class Point
{
    int x, y;
public:
    Point(int a)                   { cout << "int" << endl; }              // 1
    Point(int a, int b)            { cout << "int, int" << endl; }         // 2
    Point(initializer_list<int> s) { cout << "initializer_list" << endl; } // 3
};
int main()
{
    Point p1(1);        // 1 없으면 error
    Point p2{ 1 };      // 3 없으면 1

    Point p3(1, 1);     // 2  없으면 error
    Point p4({ 1,1 });  // 3 없으면  error가 나와야 하지만 {1,1} 가 Point 로
                        // 형변환으로 임시객체가 생성되어 p4로 복사 됩니다.
    Point p5{ 1,1 };    // 3 없으면 2

    Point p6 = { 1,1 }; // 3 없으면   2
    Point p7(1, 2, 3);  // error.

    Point p8{ 1, 2, 3 }; // ok. 3
    Point p9 = { 1,2,3 }; // ok. 3
}
```

▌ STL 컨테이너와 initializer_list

C++11 부터는 STL의 다양한 컨테이너를 만들 때 아래 처럼 만들어져 있습니다.

```
template<typename T, typename Ax = allocator<T> > class vector
```

```
{
    T* buff;
public:
    vector(size_t sz, T initValue = T()) {}  // 1
    vector(initializer_list<T> s)        {}  // 2
};

int main()
{
    vector<int> v1(10, 3); // 1번 생성자를 사용합니다. 10개를 3으로 초기화
    vector<int> v2{10, 3}; // 2번 생성자를 사용합니다. 2개의 요소, 10, 3

    // 아래 처럼 사용가능 합니다.
    vector<int> v3 { 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v4 = { 1,2,3,4,5,6,7,8,9,10 };
}
```

7. aggregate initialization

▌ 핵심 개념 .

생성자가 없어도 braced-init-list로 초기화 가능한 타입을 aggregate라고 합니다. 배열, 구조체 등이 있습니다.

▌ 기본 예제

```
struct Point
{
    int x;
    int y;
    Point() = default;
};
int main()
{
    Point p1;           // ok
    Point p2 = { 1,1 }; // ok. 인자가 2개인 생성자가 없지만 에러가 발생하지 않습니다.
}
```

▌ 객체 초기화와 생성자

모든 객체를 생성될 때 적절한 모양의 생성자를 호출해야 합니다.

```
struct Point
{
    int x;
    int y;
    Point(int a, int b) {}
};
int main()
{
    Point p1;           // error. 디폴트 생성자가 없습니다.
    Point p2 = { 1,1 }; // ok.
}
```

위 코드에서 p1 객체를 생성하려면 디폴트 생성자가 있어야 하는데 디폴트 생성자가 없으므로 error가

발생합니다. 그렇다면, 만약에 인자가 2개인 생성자도 없으면 어떻게 될까요 ?

```
struct Point
{
    int x;
    int y;
};
int main()
{
    // C언어 시절 부터 구조체는 아래 {}로 초기화 할 수 있었습니다.
    Point p1;           // ok.
    Point p2 = { 1,1 }; // ok.
}
```

C언어 시절부터 구조체는 중괄호(braced-init-list)로 초기화 할 수 있었기 때문에 위 코드에서 p1, p2를 만드는 것은 전혀 문제가 없습니다. 이 처럼 생성자가 없어도 braced-init-list로 초기화 될 수 있는 타입을 aggregate라고 합니다. 대표적인 적인 배열과 구조체 입니다.

그렇다면, 사용자가 인자 없는 생성자를 명시적으로 제공하면 어떻게 될까요 ? 또는, default function 문법으로 생성자를 제공하면 어떻게 될까요 ?

```
struct Point1
{
    int x, y;
    Point1() {}
};

struct Point2
{
    int x, y;
    Point2() = default;
};

int main()
{
    Point1 p1 = { 1,1 }; // error. Point1 는 aggregate가 아닙니다.
    Point2 p2 = { 1,1 }; // ok.    Point2 는 aggregate입니다.
}
```

또한, 가상 함수가 있으면 aggregate가 아니지만, 일반 멤버 함수는 있어도 됩니다.

```
struct Point1
{
    int x, y;
    virtual void foo() {}
};
struct Point2
{
    int x, y;
    void foo() {}
};
int main()
{
    Point1 p1 = { 1,1 }; // error. Point1 는 aggregate가 아닙니다.
    Point2 p2 = { 1,1 }; // ok.    Point2 는 aggregate입니다.
}
```

그 외의 aggregate의 조건은 아래 사이트를 참고 하시면 됩니다.

http://en.cppreference.com/w/cpp/language/aggregate_initialization

항목 1-05

auto / decltype

주요 학습 내용

> auto type deduction
>
> decltype type deduction
>
> suffix return type

1. auto type deduction

▌ 핵심 개념

auto가 우변의 표현식으로 부터 타입을 추론하는 규칙

auto a = exp	exp가 가진 const, volatile, reference 속성을 제거한후 auto 타입 결정 exp가 배열이라면 auto는 포인터
auto& a = exp	exp가 가진 reference 속성을 제거한후 auto 타입 결정, const와 volatile 속성은 유지 exp가 배열이라면 auto도 배열
auto&& a = exp	exp 가 lvalue 라면 auto는 참조 타입, exp가 rvalue 라면 auto는 값 타입. 책 후반부의 "완벽한 전달자" 항목 참고

```
int main()
{
    int n = 10;
    int& r = n;
    const int c = 10;
    const int& cr = c;

    // 규칙 1. 값 타입 일때
    // 우변 수식의 reference, const, volatile 속성을 제거하고 auto의 타입 결정
    auto a1 = n;  // int
    auto a2 = r;  // int
    auto a3 = c;  // int
    auto a4 = cr; // int

    // 규칙 2. 참조 타입 일때
    // 우변 수식의 레퍼런스 속성을 제거하고 auto결정. const, volatile 속성은 유지
    auto& a5 = n;  // auto : int     a5 : int&
    auto& a6 = r;  // auto : int     a6 : int&
    auto& a7 = c;  // auto : const int  a7 : const int&
    auto& a8 = cr; // auto : const int  a8 : const int&

    // 규칙 3. forwarding reference - "perfect forwarding" 항목 참고
    auto&& a9  = n;     // auto : int&  a9  : int&
    auto&& a10 = 10;    // auto : int   a10 : int&&
```

```
    // 배열과 auto
    int x[3] = { 1,2,3 };
    auto  a11 = x;  // int* a11      = x;
    auto& a12 = x;  // int (&a12)[3] = x;
}
```

auto의 추론 규칙은 템플릿 추론 규칙과 동일합니다.

2. decltype deduction

▌ 핵심 개념 .

"decltype(exp) n" 은 exp와 동일한 타입의 변수 n을 선언하는 코드 입니다.

이때, exp가 심볼이라면 심볼의 선언을 보고 동일한 타입을 결정합니다.

그외에 exp에 연산자 등이 포함되어 있으면 exp가 좌변에 올 수 있으면 참조, 그렇지 않으면 값 타입으로 결정됩니다.

▌ 기본 예제

```
int main()
{
    int n = 10;
    const int& r = n;
    decltype(r) r2 = r; // const int& r2;

    int* p = &n;

    decltype(p)    d1; // int*
    decltype(*p)   d2; // int&

    int x[3] = { 1,2,3 };
    decltype(x[0]) d3; // x[0]은 lvalue가 될 수 있으므로   int&

    decltype(n + n) d4; // "n+n = 20" 은 될수 없으므로  int
    decltype(++n)   d5; // "++n = 20" 은 될수 있으므로  int&
    decltype(n++)   d6; // "n++ = 20" 은 될수 없으므로  int
}
```

3. decltype(auto)

▌ 핵심 개념 .

우변을 보고 타입을 결정하지만 decltype의 규칙 적용.

▌ 기본 예제

```
int x = 0;

int& foo() { return x; }

int main()
{
    auto            r1 = foo(); // int r1 = foo();

    decltype(foo()) r2 = foo(); // int& r2 = foo();

    decltype(auto)  r3 = foo(); // int& r3 = foo();
}
```

4. suffix return type, trailing return

▌ 핵심 개념 .

함수의 리턴 타입을 함수의 () 뒤쪽에 표현하는 방식, 함수 템플릿이나 람다 표현식에서 주로 사용

▌ 기본 예제

```
auto square(int a) -> int
{
    return a * a;
}
int main()
{
    int n = square(3);
}
```

▌ 후위 반환 타입이 필요한 경우

서로 다른 2개의 임의의 타입의 변수의 곱을 구하는 함수 템플릿 Mul() 을 생각해 봅시다. Mul의 리턴 타입은 어떻게 적어야 할까요 ?

```
template<typename T, typename U> ? Mul(T a, U b)
{
    return a * b;
}
```

결국 T타입과 U 타입의 변수를 곱할 때 나오는 타입이므로 decltype(a*b)로 하면 됩니다. 하지만, 리턴 타입을 함수 이름 앞에 표기하게 되면, a, b변수를 선언하기 이전에 사용하는 모양이 되기 때문에 컴파일 에러가 발생하게 됩니다.

```
template<typename T, typename U>
decltype(a*b) Mul(T a, U b) // error. a, b 변수를 선언하기 전에 사용했습니다.
{
    return a * b;
}
```

이 경우 함수의 리턴 타입을 함수()뒤에 표기하는 후위형 반환 타입 구문을 사용하면 해결할 수 있습니다.

```
template<typename T, typename U>
auto Mul(T a, U b) -> decltype(a*b) // ok
{
    return a * b;
}
```

또한, C++14 부터는 return 문장이 하나인 경우는 후위 리턴 표기법을 생략 할 수도 있습니다.

```
template<typename T, typename U> auto Mul(T a, U b)  // C++14 부터는 ok
{
    return a * b;
}
```

SECTION 2

rvalue 와 move semantics

항목 2-01

Temporary Object

주요 학습 내용

> 임시 객체의 생성과 수명
>
> 임시 객체의 특징
>
> 함수 인자와 임시 객체
>
> 함수 리턴 값과 임시 객체
>
> 값 리턴과 참조 리턴, RVO, NRVO
>
> 임시 객체가 생성되는 다양한 경우

1. 임시 객체의 생성과 수명

▌ 핵심 개념

"클래스 이름()"를 사용하면 임시 객체를 만들 수 있습니다.

임시 객체는 자신을 선언한 문장의 끝에서 파괴 됩니다.

▌ 기본 예제

```
class Point
{
public:
    int x, y;

    Point(int a = 0, int b = 0) : x(a), y(b) { std::cout << "Point()"  << std::endl; }
    ~Point()                                 { std::cout << "~Point()" << std::endl; }
};

int main()
{
    // Point p1(1, 2);  // named object.
                        // p1 이라는 이름이 있습니다.
                        // main 함수의 블록({})을 벗어날때 파괴 됩니다.

    // Point(1, 2);     // unnamed object. temporary.
                        // 이름이 없습니다.
                        // 문장의 끝에서 파괴 됩니다.

    Point(1, 2), std::cout << "X" << std::endl;

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

임시 객체의 수명

C++에서는 “클래스 이름()” 를 사용하면 임시 객체를 생성할 수 있습니다. 임시 객체는 이름이 없고, 자신을 선언한 문장의 끝에서 파괴 됩니다.

```
int main()
{
    Point p;    // 일반 객체, p라는 이름이 있습니다.
                // 자신을 선언한 블럭을 벗어 날 때 파괴 됩니다.

    Point();    // 임시 객체, 이름이 없습니다.
                // 자신을 생성한 문장의 끝에서 파괴 됩니다.

    cout << "A" << endl;
}
```

2. 임시 객체의 특징

임시 객체는 단일 문장에서만 사용 되므로 다음과 같은 특징이 있습니다.

- 임시 객체는 등호의 왼쪽에 올 수 없습니다.
- 임시 객체는 주소를 구할 수 없고, 참조로 가리 킬 수 없습니다.
- 임시 객체는 const 참조나, C++11의 rvalue reference로 가리 킬 수 있습니다. 이 경우, 임시 객체의 수명은 참조 변수의 수명으로 연장됩니다.

```
int main()
{
    Point* p = &Point();    // error
    Point& r = Point();     // error

    const Point& cr = Point();  // ok
    Point&& rr = Point();       // ok

    Point().x = 10; // x가 public 영역에 있어도 error입니다.
                    // 임시객체는 lvalue가 될 수 없습니다.
}
```

▌ 임시 객체와 lvalue

임시 객체는 자신을 생성한 문장에서만 사용되고, 문장의 끝나면 즉시 파괴 됩니다. 따라서, 임시 객체에 어떤 값을 넣는 것은 허용되지 않습니다. 또한, 임시 객체는 주소 연산자로 주소를 구할 수 도 없습니다.

```
int main()
{
    Point().x = 10; // error
    cout << &Point() << endl; // error
}
```

▌ 임시 객체와 참조 타입

임시 객체와 참조 타입에 대한 문법은 약간 복잡합니다. 아래와 같은 규칙이 있습니다.

[참고] C++11 에서 rvalue reference 문법이 추가 되었으므로, C++11이전에 사용되던 전통적인 참조 문법은 lvalue reference 라고 부르도록 하겠습니다. rvalue reference에 대한 자세한 내용은 뒤쪽에서 배우게 됩니다.

① 임시 객체는 lvalue reference 변수로 가리킬 수 없습니다.

② 임시 객체는 const lvalue reference 변수로 가리킬 수 있습니다. 이경우, 임시 객체의 수명은 참조 변수의 수명과 동일합니다. 하지만, const 참조 변수 이므로, 참조 변수를 통해서는 값을 변경할 수 없습니다.

③ 임시 객체는 C++11의 rvalue 참조 변수로 가리킬 수 있습니다.

```
int main()
{
    // 1. C++98/03 참조
    Point& r1 = Point(); // error

    // 2. C++98/03 const 참조
    const Point& r2 = Point(); // ok.
    r2.x = 10;  // error. x가 public에 있어도, r2는 const 입니다.

    // 3. C++11 rvalue 참조
    Point&& r3 = Point(); // ok
    r3.x = 10; // ok.
}
```

3. 함수 인자와 임시 객체

어떤 객체가 함수의 인자로만 사용 된다면 객체를 만든 후에 인자로 전달하는 것 보다는 임시객체를 사용해서 전달하는 것이 좋습니다.

```
int main()
{
    Point p(0, 0);  // 객체 p를 만들고
    foo(p);         // 함수 인자로 전달합니다.
                    // p 는 더이상 필요 없지만 즉시, 파괴되지 않고 블럭을 벗어날 때
                    // 파괴 됩니다.
    foo(Point(0, 0));   // 임시객체를 사용한 인자 전달
                        // 함수 호출이 종료 되면 인자로 사용된 임시객체는
                        // 즉시 파괴 됩니다.
}
```

이 경우 주의 할 점은 함수 인자로 임시객체를 사용할 수 있게 하려면 함수를 만들 때는 반드시 const 참조나 혹은 값 타입으로 만들어야 합니다.(또는, 뒤에서 나오는 rvalue reference 타입으로 받을수도 있습니다.)

```
void foo(Point& p) {}
int main()
{
    foo( Point(0, 0) ); // error. 비 상수 참조로는 임시객체를 가리킬 수 없습니다.
}
```

STL의 알고리즘 다양한 알고리즘 함수를 사용 할 때 임시객체를 사용하는 경우가 많이 있습니다.

```
int main()
{
    vector<int> v{ 1,3,5,7,9,2,4,6,8,10 };
    sort(begin(v), end(v), greater<int>());
}
```

4.함수 리턴 값과 임시 객체

▌ 핵심 개념

함수가 값 타입으로 리턴 하면 임시객체가 생성됩니다.

▌ 기본 예제

```
struct Point
{
    int x, y;

    Point() { cout << "Point()" << endl; }
    ~Point() { cout << "~Point()" << endl; }
};
Point foo()
{
    Point p;
    return p;   // 이 순간 p의 복사본(임시객체)를 만들어서 리턴
                // return Point(p);
}
int main()
{
    foo();
}
```

▌ 값 리턴과 리턴용 임시객체

어떤 함수가 값 타입으로 객체를 리턴 하면 리턴용 임시객체가 생성됩니다. 리턴용 임시객체는 함수 호출 문장의 끝에서 파괴 됩니다. 아래 코드 에서 Point 객체는 foo()안에서 만든 p밖에 없지만 리턴용 임시객체가 때문에 소멸자가 2번 호출되게 됩니다.

```
struct Point
{
    int x, y;

    Point() { cout << "Point()" << endl; }
    ~Point() { cout << "~Point()" << endl; }
```

```
    Point(const Point&) { cout << "Point(const Point&)" << endl; }
};
Point foo()
{
    Point p;
    return p;   // 이 순간 p의 복사본(임시객체)를 만들어서 리턴
                // return Point(p);
}
int main()
{
    foo(), cout << "A" << endl;;
    cout << "B" << endl;
}
```

[참고] g++ 사용시 -fno-elide-constructors 을 사용해서 컴파일 해야 임시객체가 생성되는 것을 볼 수 있습니다.

▌ RVO, NRVO (Named RVO)

함수가 값타입으로 리턴 할 경우 만든 후 리턴 하지 말고 만들면서 리턴 하면 보다 효율적인 코드를 작성할 수 있습니다.

```
Point foo()
{
    // Point p;     // 만들 때 생성자,
    // return p;    // 리턴할 때 임시객체를 만들기 위한 복사 생성자 호출

    return Point(); // 리턴용 임시객체만 생성.
}
```

위와 같은 기법을 “RVO (Return Value Optimization)” 이라고 합니다. 하지만, 요즘 컴파일러들은 이름 있는 객체를 만든 후에 리턴 해도 최적화 과정을 통해서 임시객체를 제거하는 경우가 많이 있습니다. 이와 같은 현상을 “이름있는 객체의 RVO”라는 의미로 “NRVO”라고 합니다.

5. 값 리턴과 참조 리턴, RVO, NRVO

① 함수가 값을 리턴 하면 임시객체가 생성되지만, 참조를 리턴 하면 임시객체가 생성되지 않습니다.

② 값을 리턴 하는 함수는 lvalue가 될 수 없지만(리턴 값이 임시객체 이므로) 참조를 리턴 하는 함수는 lvalue가 될 수 있습니다

③ 지역변수의 참조를 리턴 하면 안됩니다.

```
struct Point
{
    int x;
    int y;
};
Point p = { 1, 2 };

Point  foo() { return p; }
Point& goo() { return p; }

int main()
{
    foo().x = 10;   // error.

    goo().x = 10;   // ok
    cout << p.x << endl;
}
```

6. 임시 객체가 생성되는 다양한 경우

▌ 캐스팅 과 임시객체

값에 의한 캐스팅은 임시 객체가 생성되지만 참조에 의한 캐스팅은 임시객체가 생성되지 않습니다.

▌ 멤버 이름 충돌

기반 클래스의 멤버 data 이름과 파생 클래스의 멤버 data 이름이 동일한 경우, 기반 클래스의 멤버에 접근하려면 객체를 기반 클래스 타입으로 캐스팅 하면 됩니다.

```
struct Base
{
    int value = 10;
};
struct Derived : public Base
{
    int value = 20;
};
int main()
{
    Derived d;
    cout << d.value << endl; // 20
    cout << static_cast<Base>(d).value << endl; // 10
}
```

하지만, 이경우 값에 의한 캐스팅은 Base의 임시객체를 만드는 표현이 되므로 lvalue에 놓을 수가 없습니다. 참조에 의한 캐스팅을 사용해야 합니다.

```
Derived d;
// static_cast<Base>(d) : d를 복사 해서 Base의 임시객체를 만드는 표현입니다.
static_cast<Base>(d).value = 30; // error. 임시객체는 lvalue가 될수 없습니다.
static_cast<Base&>(d).value = 30; // ok
```

항목 2-02

rvalue vs lvalue

주요 학습 내용

> rvalue vs lvalue
>
> rvalue reference
>
> reference 와 overloading
>
> reference collapse

1. rvalue vs lvalue

▌ 핵심 개념

lvalue	= 의 좌측 또는 우측에 놓을 수 있는 식(expression) 메모리에 놓이고 &연산자로 주소를 얻을 수 있다. 단일 식을 넘어 지속되는 개체 이름을 가진다 참조를 리턴 하는 함수 임시객체
rvalue	= 의 우측에만 놓을 수 있는 식(expression) &연산자로 주소를 얻을 수 없다. rvalue를 사용하는 식 외에는 유지 되지 않은 임시값 이름을 가지지 않는다. 값을 리턴 하는 함수 literal

▌ 기본 예제

```
int a = 42, b = 43;

a = 10; // a는 lvalue, 10은 rvalue
10 = b; // error. rvalue는 등호의 왼쪽에 올수 없습니다.
b = a;  // ok. lvalue는 등호의 왼쪽과 오른쪽에 모두 올수 있습니다.
// lvalue는 이름(a) 과 값(10)이 모두 있지만 rvalue는 이름이 없고 값만 있습니다.

int* p1 = &a;  // ok. lvalue는 주소연산자로 주소를 구할수 있습니다.
int* p2 = &10; // error. rvalue는 주소를 구할수 없습니다

const int c = 10;
c = 20; // error.   c는 rvalue일까요 ? lvalue일까요 ?
        // c는 이름이 있고, 주소 연산자로 주소를 구할수 있고,
        // 자신을 선언한 블럭을 벗어날때 까지 사용가능합니다. - lvalue입니다.
        // 수정 불가능한(immutable) lvalue 입니다.
```

2. expression

▌ expression 과 value categories

lvalue 와 rvalue 의 개념은 객체가 아닌 expression 에 적용되는 개념입니다.

expression 은 하나의 값으로 계산 될수 있는 식입니다.

```
int a = 42;
a = 10;     // ok. a는 lvalue
a + 1 = 10; // error. a + 1 은 lvalue 가 될 수 없음.
```

위 코드에서 a, a+1 은 모두 하나의 식(expression) 입니다. a 는 lvalue 가 될 수 있지만 a+1 은 lvalue 가 될 수 없습니다.

3. lvalue reference vs rvalue reference

▌ 핵심 개념

타입&	C++98/03 문법. lvalue 만 참조 할 수 있습니다. lvalue reference 라고 부릅니다.
const 타입&	C++98/03 문법. lvalue 와 rvalue 를 모두 참조 할 수 있습니다.
타입&&	C++11/14 문법. rvalue 만 참조 할 수 있습니다. rvalue reference 라고 부릅니다.

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

    // 규칙1. lvalue reference는 lvalue 만 참조 할 수 있습니다.
    int& lr1 = n;   // ok
    int& lr2 = 10;  // error. 10 is rvalue

    // 규칙 2. const lvalue reference는 lvalue와 rvalue를 모두 참조 할 수 있습니다.
    const int& clr1 = n;    // ok
    const int& clr2 = 10;   // ok

    // 규칙 3. rvalue reference 는 rvalue만 참조 할 수 있습니다
    int&& rr1 = n;  // error. n is lvalue
    int&& rr2 = 10; // ok
}
```

4. reference 와 함수 오버로딩

▌ 핵심 개념

① &, const&, && 로 함수 오버로딩 할 수 있습니다.

② lvalue, rvalue에 따라 함수가 선택되는 순서가 중요합니다.

③ rvalue reference type의 이름 있는 변수는 lvalue 입니다.

```
void foo(int& a)       { cout << "int&" << endl; }       // 1
void foo(const int& a) { cout << "const int&" << endl; } // 2
void foo(int&& a)      { cout << "int&&" << endl; }      // 3

int main()
{
    int n = 10;

    foo(n);  // int&,  없으면 const int&
    foo(10); // int&&, 없으면 const int&&

    int& r1 = n;
    foo(r1); // int&, 없으면 const int&

    int&& r2 = 10;  // 10은 rvalue 이지만 10을 가르키는 r2 는 lvalue이다.
                    // 이름이 있고, 단일식 벗아나도 존재한다.
                    // "이름이 있는 rvalue reference 는 lvalue 이다."

    foo(r2);        // int&, 없으면 const int&

    foo(static_cast<int&&>(r2));    // int&&, 없으면 const int&
                                    // r2를 rvalue로 캐스팅하는 코드
}
```

5. reference collapse

▌ 핵심 개념

template 이나 typedef (또는 using)을 사용할 때 참조과 충돌 날 때의 규칙

& &	&
& &&	&
&& &	&
&& &&	&&

▌ 기본 코드

```
using LR = int&;
using RR = int&&;

int n = 10;

LR r1 = n;  // int&  r1 = n
RR r2 = 10; // int&& r2 = 10

// reference collapsing 이라고 합니다. && &&의 경우만 && 입니다.
LR& r3 = n ;    // int& &   => int&
RR& r4 = n ;    // int&& &  => int&
LR&& r5 = n ;   // int&& &  => int&
RR&& r6 = 10 ;  // int&& && => int&&

template<typename T> void foo(T& a) {}

foo<int&>(n); // T는 int& 이므로 a 는 int& & 입니다. 결국 a 는 int&


int& & r7 = n; // error. 사용자가 직접 참조 충돌을 사용하면 안된다.
```

항목 2-03

forwarding reference

주요 학습 내용

> 함수 인자와 레퍼런스
>
> forwarding reference
>
> forwarding reference 활용.

1. 함수 인자와 레퍼런스

함수 인자로 lvalue reference를 사용할 경우 인자로 lvalue 만 사용 할 수 있고, rvalue reference를 사용할 경우 rvalue 만 사용 할 수 있습니다.

```
// 함수 인자로 lvalue reference를 사용합니다. lvalue 만 전달 할 수 있습니다.
void foo(int& a) {}

// 함수 인자로 rvalue reference를 사용합니다. rvalue 만 전달 할 수 있습니다.
void goo(int&& a){}

int main()
{
    int n = 10;
    foo(n);     // ok
    foo(10);    // error

    goo(n);     // error
    goo(10);    // ok
}
```

▌ 함수 템플릿과 레퍼런스 인자

아래의 함수 템플릿을 생각해 봅시다. 이 함수의 인자는 lvalue reference 일까요? rvalue reference 일까요 ? 아니면 lvalue, rvalue 모두 가능할까요 ?

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

단순히, "&" 가 하나만 있다고 함수 인자가 lvalue reference 라고 생각할 수는 없습니다. 만약, T의 타입이 int& 였다면, T& 는 결국 "int& &" 가 되므로 reference collapse 규칙을 적용한 후 타입이 결정되고 함수 코드가 생성되게 됩니다. 좀더 자세히 살펴 보도록 하겠습니다.

2. 함수 템플릿과 레퍼런스

▌ 함수 템플릿과 레퍼런스 인자

템플릿은 T의 타입이 결정되면 함수가 생성됩니다. 또한, 사용자가 함수 템플릿의 타입을 명시적으로 지정할 수도 있습니다. 설명을 단순화 하기 위해 정수(int) 타입 만을 생각해 보겠습니다.

사용자 T의 타입을 각각 “int, int&, int&&” 타입으로 지정 했을 때 컴파일러에 의해서 생성되는 함수를 생각해 봅시다. 아래 코드의 주석을 잘 생각해 보세요.

```
template<typename T> void foo(T& a)
{
}
int main()
{
    int n = 10;

    // 사용자가 템플릿의 인자를 명시적으로 전달하고 있습니다.
    // int, int&, int&& 중 어떤 것을 전달해도 함수 인자는 int& 로 결정됩니다.
    // 함수 인자로는 lvalue 만 전달할 수 있습니다.
    foo<int>(n);    // T : int   T& : int&    결국 생성되는 함수는 foo(int& a)
    foo<int&>(n);   // T : int&  T& : int& &  결국 생성되는 함수는 foo(int& a)
    foo<int&&>(n);  // T : int&& T& : int&& & 결국 생성되는 함수는 foo(int& a)
}
```

결국, 위 함수 템플릿의 경우 T가 “값 타입, lvalue reference, rvalue reference” 중 어느 것으로 전달되어도 최종적으로는 foo 함수의 인자는 lvalue reference 타입으로 결정됩니다.

따라서, 이 생성된 함수에는 lvalue 만 전달할 수 있습니다.

▌ 템플릿 인자를 명시적으로 전달하지 않은 경우

이번에는 템플릿 인자를 명시적으로 전달하지 않은 경우를 생각해 봅시다.

```
template<typename T> void foo(T& a)
{
}
int main()
{
    int n = 10;
```

```
    foo(10);    // error.
    foo(n);     // ok. T는 int,  T&는 int& 로 결정됩니다.
}
```

먼저, 10의 경우를 생각해 보면, 컴파일러는 최대한 노력해서 10을 전달 받을 수 있도록 T의 타입을 결정해야 합니다. 하지만, T를 "int, int&, int&&" 중 어떠한 것으로 결정해도 최종적으로 T& 는 결국 int& 타입이 되므로 lvalue 만 전달 할 수 있습니다. 따라서, rvalue 인 10을 전달 받을 수 있도록 하는 T의 타입을 결정할 수 없기 때문에 이 코드는 에러가 발생합니다.

반면, n을 전달할 경우, T의 타입을 "int, int&, int&&" 중 어떠한 것으로 결정해도, T&는 int& 타입이 되므로 모두 가능합니다. 이 경우 컴파일러는 T를 int 로 결정하게 됩니다.

결국, T& 의 경우는 모든 타입에 대해서 lvalue reference 을 인자로 가지는 함수만 생성할 수 있습니다.

이때 중요한 것은 단순히 &가 하나이기 때문이 아니라, T의 타입이 int, int&, int&& 중 어떠한 것으로 결정되어도 결국 T& 는 int& 타입이 됩니다.

3. forwarding reference

이번에는 함수 템플릿의 인자가 T&& 일 경우는 생각해 봅시다.

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

이 경우도 결국 함수 템플릿 이므로 T의 타입이 결정되면 실제 C++ 함수가 생성되게 됩니다. T의 타입에 따라 어떤 모양의 함수가 생성되는지를 자세히 살펴 보겠습니다.

▌ 사용자가 템플릿 인자를 명시적으로 전달하는 경우

앞의 예제와 마찬 가지로 사용자가 명시적으로 타입을 지정하는 경우는 생각해 봅시다. 그리고 각각의 경우 함수 인자로 "n , 10" 중 어느 것을 전달해야 할지도 생각해 봅시다.

```
template<typename T> void foo(T&& a)
{
}
int main()
{
    int n = 10;

    // 사용자가 템플릿의 인자를 명시적으로 전달하고 있습니다.
    // T의 타입이 int 또는 int&& 로 결정되면 int&&를 함수 인자로 가지는 함수가 생성되고
    // T의 타입이 int& 로 결정되면 함수 인자로 int& 를 가지는 함수가 생성됩니다.
    foo<int>(10);   // T : int   T&& : int&&    결국 생성되는 함수는 foo(int&& a)
    foo<int&>(n);   // T : int&  T&& : int& &&  결국 생성되는 함수는 foo(int&  a)
    foo<int&&>(10); // T : int&& T&& : int&& && 결국 생성되는 함수는 foo(int&& a)
}
```

결국, 함수 템플릿의 인자가 T&& 일 경우, T가 어떻게 결정 되느냐에 따라서 lvalue reference, 또는 rvalue reference를 인자로 가지는 함수가 생성되게 됩니다.

▌ 사용자가 템플릿 인자를 명시적으로 전달하지 않은 경우

이번에는 사용자가 함수 템플릿의 인자를 명시적으로 전달하지 않은 경우를 생각해 봅시다.

이 경우에는 사용자가 타입을 전달하지 않았으므로 컴파일러는 함수의 인자를 보고 T의 타입을 결정해야 합니다.

먼저, 함수 인자로 n을 전달하는 경우는 생각해 보면, n은 lvalue 이므로 n을 전달 받기 위해서는 lvalue reference 를 인자로 가지는 함수가 생성되어야 합니다. 앞의 예제에서 T가 int& 로 결정되면 T&& 는 int& 로 결정될 수 있으므로 컴파일러는 T의 타입을 int& 로 결정하게 됩니다.

반면에, 함수 인자로 10을 전달하면 컴파일러는 T를 int로 결정하게 되고 결국 함수 인자로 int&&를 가지는 함수가 생성되게 됩니다.

▌ forwarding reference 의 의미

결국 함수 인자를 정리해 보면 아래와 같습니다.

함수 모양	
f(int&)	함수 인자로 int 타입의 lvalue 만 전달 할 수 있습니다.
f(int&&)	함수 인자로 int 타입의 rvalue 만 전달 할 수 있습니다.
f(T&)	함수 인자로 임의의 타입의 lvalue 만 전달 할 수 있습니다.
f(T&&)	함수 인자로 임의의 타입의 lvalue 와 rvalue 를 모두 전달할 수 있습니다.*

[참고] 정확히는 사용자가 lvalue전달하면 lvalue reference 를 인자로 가지는 함수를 생성하고, rvalue를 전달하면 rvalue reference 를 인자로 가지는 함수를 생성하게 됩니다.

▌ forwarding reference 의 활용

forwarding reference 는 C++11 의 핵심 기술인 "perfect forwarding" 에서 아주 중요하게 사용됩니다.

자세한 내용은 뒤 부분에 나오는 "perfect forwarding" 항목을 참고하시면 됩니다.

항목 2-04

move semantics

주요 학습 내용

> move constructor
>
> std::move
>
> copy vs move
>
> move implementation
>
> move 와 noexcept
>
> move 와 const object
>
> 복사와 이동을 모두 지원하는 멤버 함수 만들기.

1. 복사(copy) 생성자와 이동(move) 생성자

▌ 핵심 개념

복사 생성자 (Copy constructor)	Cat(const Cat&)
이동 생성자 (Move constructor)	Cat(Cat&&)

▌ 기본 예제

```
class Cat
{
    char* name;
    int   age;
public:
    Cat(const char* s, int a) : age(a) {
        name = new char[strlen(s) + 1];
        strcpy(name, s);
    }
    // copy constructor
    Cat(const Cat& c) : age(c.age) {
        name = new char[strlen(c.name) + 1];
        strcpy(name, c.name);
    }
    // move constructor
    Cat(Cat& c) : name(c.name), age(c.age)  {
        c.name = 0;
        c.age = 0;
    }
    ~Cat() { delete[] name; }
};
int main()
{
    Cat c1("nabi", 2);
    Cat c2 = c1; // copy
    Cat c3 = static_cast<Cat&&>(c1); // move
}
```

2. std::move

▌ 핵심 개념

lvalue 또는 rvalue를 인자로 받아서 rvalue로 캐스팅 하는 함수. move 생성자를 호출하기 위해 사용

▌ 기본 예제

```
int main()
{
    Cat c1("nabi", 2);
    Cat c2 = c1;         // copy
    Cat c3 = move(c1);   // move
                         // static_cast<Cat&&>(c1);
}
```

▌ std::move 의 원리

lvalue또는 rvalue를 모두 전달 받기 위해서 forwarding reference(T&&)를 사용합니다.

```
template<typename T>
typename remove_reference<T>::type&& move(T&& arg)
{
    return static_cast<typename remove_reference<T>::type&&>(arg);
}
```

3. copy 와 move

▌ 핵심 개념

rvalue는 const lvalue reference에 전달 될 수 있습니다. 이동(move) 생성자를 제공하지 않으면 복사(copy) 생성자를 사용하게 됩니다.

▌ 기본 예제

```
class Test
{
public:
    Test() = default;
    Test(const Test& t) { cout << "copy" << endl; }
    Test(Test&& t)      { cout << "move" << endl; }
};
int main()
{
    Test t1;
    Test t2 = move(t1); // static_cast<Test&&>(t1) 입니다
                        // 1. move 생성자가 있으면 move 생성자를 사용합니다.
                        // 2. move 생성자가 없으면 copy 생성자를 사용합니다.
}
```

▌ std::swap

STL의 swap은 move를 사용해서 만들어져 있습니다.

```
template<typename T> void swap(T& a, T& b)
{
    // T 타입에 move 생성자가 있으면 move 생성자를 사용합니다.
    // T 타입에 move 생성자가 없으면 copy 생성자를 사용합니다.
    T tmp = move(a);
    a = move(b);
    b = move(tmp);
}
```

4. move 지원하기

▌ 핵심 개념

① move 생성자와 move 대입연산자를 모두 지원해야 합니다.

② move 계열함수 에서는 모든 멤버를 move 하고, 복사 계열 함수에서는 모든 멤버를 복사 합니다.

▌ 기본 예제

```
class Data
{
    string str;
public:
    Data(string s = "AAA") : str(s) {}
    // 복사 계열에서는 복사를
    Data(const Data& s) : str(s.str) { cout << "copy" << endl; }
    Data& operator=(const Data& s)
    {
        cout << "copy=" << endl;
        str = s.str;
        return *this;
    }
    // Move 계열함수 : 모든 멤버를 move 로 옮겨야 한다.
    Data(Data&& s) : str(move(s.str)) { cout << "move" << endl; }
    Data& operator=(Data&& s)
    {
        cout << "move=" << endl;
        str = move(s.str);
        return *this;
    }
};
int main()
{
    Data d1("AAA");
    Data d2 = d1;       // copy
    Data d3 = move(d1); // move
}
```

5. move 와 noexcept

▌ 핵심 개념

① buffer의 요소를 복사 하는 경우는 예외 안정성의 강력 보장을 지원 합니다.

② buffer의 요소를 이동 하는 경우는 예외 안정성의 강력 보장을 지원 하지 못합니다

③ buffer이동시는 move 계열함수가 예외가 없을 때만 move를 사용하는 것이 좋습니다.

▌ 기본 예제

```
int main()
{
    Test t1;
    Test t2 = t1;                   // 항상 복사 생성자를 사용합니다.
    Test t3 = move(t1);             // 항상 move 생성자를 사용합니다.
    Test t4 = move_if_noexcept(t3); // move 생성자에 예외가 없을 때만
                                    // move 생성자를 사용합니다.
}
```

▌ 버퍼 이동 하기

작은 버퍼를 큰 버퍼로 이동할 때, 복사 보다는 이동이 효율적입니다. 하지만, N개 이동을 성공하고 N+1 번째를 이동하다가 예외가 발생한 경우 버퍼를 원상 복구 할 수 없습니다. (복구를 위해 다시 이동할때 예외가 또 발생할 수 있습니다.) 따라서, 안전하게 버퍼의 내용을 옮기려면 이동 생성자에 예외가 없을 때만 이동 생성자를 사용하고 예외 가능성이 있으면 복사로 옮기는 것 좋습니다.

```
class Test
{
public:
    Test() = default;

    Test(const Test& s) { cout << "copy" << endl; }

    // move 생성자 : noexcept가 있을 때와 없을때를 각각 결과를 확인해 보세요
    Test(Test&& s) noexcept { cout << "move" << endl; }
```

```
    // copy 대입, move 대입도 제공해야 합니다.
};
int main()
{
    // 10개의 객체를 생성합니다.
    Test* p1 = new Test[10];

    // 20개의 버퍼를 할당합니다.
    Test* p2 = static_cast<Test*>(operator new(sizeof(Test) * 20));

    for (int i = 0; i < 10; i++)
    {
        new(&p2[i]) Test(move_if_noexcept(p1[i]));  // 에외가 없을 때만
                                                    // move 생성자를 사용합니다.
    }
    // 11~20 객체의 디폴트 생성자를 호출합니다.
    for (int i = 10; i < 20; i++)
        new (&p2[i]) Test();

    // 기존 버퍼를 제거합니다.
    delete[] p1;
}
```

▌ STL과 move

STL의 vector와 같은 컨테이너는 resize() 함수에서 버퍼를 재할당 할 때 move 생성자가 예외가 없을 때만 move 생성자를 사용합니다.

```
int main()
{
    vector<Test> v(10);
    v.resize(20);   // 에외가 없을 때만 move 생성자를 사용합니다.

}
```

[참고] Microsoft 의 visual C++ 의 경우 2017년 하반기 이전 버전의 경우는 예외가 있어도 move를 사용합니다. 2017년 하반기에 나온 버전부터 예외가 없을 때만 move를 사용합니다.

6. const object 와 move

▌ 핵심 개념

const object 는 move 될 수 없습니다.

```
class Test
{
public:
    Test() = default;

    Test(const Test& s) { cout << "copy" << endl; }
    Test(Test&& s)      { cout << "move" << endl; }
};

int main()
{
    const Test t;

    Test t1 = t;       // copy
    Test t2 = move(t); // static_cast<const Test&&>(t) 입니다. copy 가 호출됩니다.
}
```

7. 복사와 이동을 모두 지원 하는 setter 함수 만들기

▌ 핵심 개념

클래스의 setter 함수를 만들 때 복사와 move를 모두 지원하게 하는 기술.

▌ 기본 예제

```
class Test
{
    string data;
public:
    // 아래 처럼 만들면 무조건 이동 됩니다.
    void setData(const string&  d) { data = move(d); }

    // 아래 처럼 만들면 무조건 복사 됩니다.
    void setData(const string&  d) { data = d; }
};

int main()
{
    string d = "data";

    Test t;
    // setData를 아래 처럼 사용하고 싶습니다.
    t.setData(d);       // 복사로 전달하고 합니다.
                        // 함수 호출 뒤에도 d를 사용할 수 있어야 합니다.
    t.setData(move(d)); // move로 전달합니다.함수 호출 뒤에는 d를 사용 할 수 없습니다.
}
```

▌ 해결책 1. 2개의 setter를 제공합니다.

lvalue reference와 rvalue reference를 각각 가지는 setData를 만들면 복사와 이동을 구별할 수 있습니다.

```
class Test
{
    string data;
```

```
public:
    void setData(string&  d) { data = d; }       // 또는, const string&도 가능합니다.
    void setData(string&& d) { data = move(d); }
};

int main()
{
    string d = "data";

    Test t;
    t.setData(d);       // ok. 복사 됩니다.
                        // d가 멤버 데이터로 전달될 때까지 비용은 copy 1회 입니다.
    t.setData(move(d)); // ok. 이동 됩니다.
                        // d가 멤버 데이터로 전달될 때까지 비용은 move 1회 입니다.
}
```

이 방식의 장점은 복사와 이동의 경우 성능이 최적화 되어 있다는 점입니다. 하지만, 단점은 setter 하나 만들 때 멤버 함수 2개를 만들어야 합니다.

▌ 해결책 2. forwarding reference로 함수 자동 생성 하기.

forwarding reference를 사용 하면 lvalue 참조와 rvalue 참조 버전을 자동으로 생성되게 할 수 있습니다.

```
class Test
{
    string data;
public:
    template<typename T> void setData(T&& d)
    {
        // 주의 move 아니라 forward를 사용해야 합니다.
        data = std::forward<T>(d);
    }
};

int main()
{
    string d = "data";

    Test t;
    t.setData(d); // setData(string& d) { data = static_cast<string&>(d); } 생성.
```

```
                    // d가 멤버 데이터로 전달될 때까지 비용은 copy 1회 입니다.

    t.setData(move(d)); // setData( string& d) { data=static_cast<string&&>(d); }
                        // d가 멤버 데이터로 전달될 때까지 비용은 move 1회 입니다.
}
```

이 방식의 장점은 필요한 함수를 자동으로 생성해 준다는 점입니다. 하지만, 단점은 template 이므로 인자가 string 아닌 다른 타입으로 사용 될 수도 있다는 점입니다.

▌ 해결책 3. call by value 사용하기

또, 다른 해결책은 setter를 만들 때 call by value를 사용하는 방법 입니다.

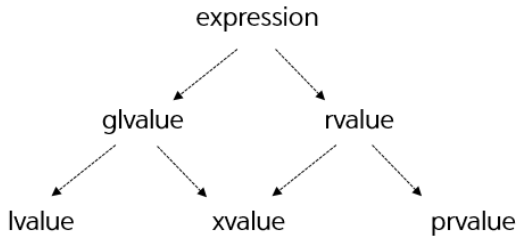
```
class Test
{
    string data;
public:
    void setData(string d)  // call by value
    {
        data = move(d);
    }
};

int main()
{
    string d = "data";

    Test t;
    t.setData(d); // ok. 복사 됩니다.
                  // d가 멤버 데이터로 전달될 때까지 비용은 copy 1회, move 1회 입니다.
    t.setData(move(d)); // ok. 이동 됩니다.
                        // d가 멤버 데이터로 전달될 때까지 비용은 move 2회 입니다.
}
```

이 방식의 장점은 setter를 한 개만 만들면 된다는 점입니다. 단점은 함수를 2개 제공 했을 때 보다 약간의 성능저하(move 1회)가 발생합니다. 하지만, move의 성능저하가 크지 않은 경우 나쁘지 않은 구현입니다.

8. value categories

▌ 핵심 개념

C++ 의 각 expression 의 분류.



lvalue	An *lvalue* (so called, historically, because lvalues could appear on the left-hand side of an assignment expression) designates a function or an object. [*Example:* If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue. —*end example*]
xvalue	An *xvalue* (an "eXpiring" value) also refers to an object, usually near the end of its lifetime (so that its resources may be moved, for example). Certain kinds of expressions involving rvalue references yield xvalues. [*Example:* The result of calling a function whose return type is an rvalue reference to an object type is an xvalue (5.2.2). —*end example*]
prvalue	A *prvalue* ("pure" rvalue) is an rvalue that is not an xvalue. [*Example:* The result of calling a function whose return type is not a reference. literal.

glvalue	A *glvalue expression* is either lvalue or xvalue. Properties: ▪ A glvalue may be implicitly converted to a prvalue with lvalue-to-rvalue, array-to-pointer, or function-to-pointer implicit conversion. ▪ A glvalue may be polymorphic: the dynamic type of the object it identifies is not necessarily the static type of the expression. ▪ A glvalue can have incomplete type, where permitted by the expression.
rvalue	An *rvalue expression* is either prvalue or xvalue. Properties: ▪ Address of an rvalue may not be taken: `&int()`, `&i++`[3], `&42`, and `&std::move(x)` are invalid. ▪ An rvalue can't be used as the left-hand operand of the built-in assignment or compound assignment operators.

	▪ An rvalue may be used to initialize a const lvalue reference, in which case the lifetime of the object identified by the rvalue is extended until the scope of the reference ends.

참고 : http://en.cppreference.com/w/cpp/language/value_category

SECTION 3

Perfect forwarding

항목 3-01

perfect forwarding

주요 학습 내용

> perfect forwarding 개념
>
> perfect forwarding 해결책
>
> forwarding reference
>
> std::forward
>
> perfect forwarding 활용
>
> perfect forwarding 주의 사항

1. perfect forwarding 개념

함수와 함수로 전달될 인자를 받아서 함수의 수행 시간을 측정하는 chronometry()이라는 함수 템플릿을 생각해 봅시다.

함수의 기본 코드는 다음과 같습니다.

```
void foo(int& a) { a = 20; }

// 특정 함수의 성능을 측정해주는 래퍼 함수
template<typename F, typename T> void chronometry(F f, T arg)
{
    // 시간 기록
    f(arg);
    // 함수 수행해 걸린 시간 출력
}
int main()
{
    int n = 10;
    chronometry(foo, n);
    cout << n << endl; // 10
}
```

이때 chronometry 의 인자로 전달받은 arg를 원본 함수인 foo 에게 완벽하게 전달하는 것을

"완벽한 전달(Perfect forwarding)"

이라고 합니다.

그런데, 위 코드의 문제는 foo 함수는 인자를 참조로 받고 있는데, chronometry 에서 arg로 전달 받을 때 값으로 받고 있기 때문에 n이 foo 함수 까지 완벽하게 전달되지 않습니다. arg라는 복사본이 foo 함수로 전달됩니다.

이 문제를 해결하려면 chronometry 의 2번째 인자를 참조로 만들어야 합니다. 하지만 이경우도 문제가 될수 있습니다.

```
void foo(int& a) { a = 20;}
void goo(int a)  {}

template<typename F, typename T> void chronometry(F f, T& a)
{
    f(a);
```

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

    chronometry(foo, n);    // ok.. 이제는 n을 foo 까지 참조로 보낼수 있습니다.
    cout << n << endl;  // 20

    goo(10);            // goo에 10을 보낼수 있습니다. chronometry(goo, 10) 도 되야
합니다.
    chronometry(goo, 10);   // error. rvalue를 lvalue reference로 받을수 없습니다.
}
```

foo()와 goo() 함수에 대해서 인자를 완벽하게 전달하려면 chronometry 은 어떻게 만들어야 할까요 ?

2. perfect forwarding 해결책

▌ step 1. overloading chronometry

chronometry 함수에 전달된 인자를 원본 함수에게 완벽하게 전달하려면 chronometry 함수 자체를 lvalue reference 와 rvalue reference 버전으로 각각 따로 만들어야 합니다.

설명을 위해 인자를 타입을 int로 놓고 코드를 작성했습니다.

```
void foo(int  n) { n = 10; }
void goo(int& n) { n = 10; }

template<typename F> void chronometry(F f, int& arg) { f(arg); } // 1
template<typename F> void chronometry(F f, int&& arg){ f(arg); } // 2

int main()
{
    int x = 0;

    chronometry(goo, x); // ok. 1번 chronometry 을 사용합니다.
    chronometry(foo, 10);// ok. 2번 chronometry 을 사용합니다.

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

[참조] const int& 와 const int&& 버전, volatile 버전도 별도로 제공해야 합니다.

이제, chronometry 함수는 인자로 10 과 x 를 모두 원본 함수 goo 와 foo 에게 전달할수 있습니다.

▌ step 2. rvalue reference 버전의 문제점 - rvalue 캐스팅

이전에 만든 chronometry은 아직 한가지의 문제 가 있습니다. 다음 코드를 생각해 보세요.

코드의 주석 부분을 주의 깊게 생각해 보세요.

```
void hoo(int&& n) {}

template<typename F> void chronometry(F f, int& arg) { f(arg); } // 1
template<typename F> void chronometry(F f, int&& arg) { f(arg); } // 2

int main()
{
```

```
    hoo(10); // 아무 문제 없습니다.
    chronometry(hoo, 10);   // hoo(10)이 문제 없으므로 이 코드도 아무 문제가 없어야
                            // 합니다. 하지만 error 발생합니다. 에러의 원인은,
                            // 이 경우 2번 chronometry을 사용하는데, 10은 rvalue 지만
                            // int&& arg = 10  에서 arg는 lvalue 입니다.
                            // 따라서, hoo(arg)는 결국 hoo(int&)를 찾게 됩니다.
}
```

결국 "이름을 가지는 rvalue reference 는 lvalue 이다." 이는 규칙 때문입니다. 해결책은 int&& 버전의 chronometry에서는 원본 함수에 arg를 보낼 때 int&& 로 캐스팅해서 보내야 합니다.

▌ 완성된 코드

결국, chronometry 함수에서 전달받은 인자를 원본 함수에게 완벽하게 전달하려면

① **인자를 받을 때 int&, int&&, 그리고 const, volatile 버전을 각각 만들어야 합니다.**

② **int&& 버전에서는 원본 함수를 호출할 때 인자를 rvalue 로 캐스팅해서 전달해야 합니다.**

완성된 코드는 다음과 같습니다.

```
void foo(int  n) { n = 10; }
void goo(int& n) { n = 10; }
void hoo(int&& n) {}

template<typename F> void chronometry(F f, int& arg)  { f(arg); }

template<typename F> void chronometry(F f, int&& arg)
{
    f(static_cast<T&&>(arg)); // arg를 rvalue로 캐스팅 합니다.
}
int main()
{
    int n = 10;
    chronometry(foo, 10);
    chronometry(goo, n);
    chronometry(hoo, 10);
}
```

3. forwarding reference

Section 3에서 살펴본 바와 같이 T&&를 forwarding reference 라고 합니다.

"forwarding reference" 를 사용하면 모든 타입의 lvalue, rvalue 를 전달 받을 수 있습니다.

정확히는 "forwarding reference" 를 사용하면 lvalue reference 버전과 rvalue reference 버전의 함수를 자동 생성 할 수 있습니다.

```
void f1(int&  a) {} // int 타입의 lvalue 만 전달 받을수 있습니다.
void f2(int&& a) {} // int 타입의 rvalue 만 전달 받을수 있습니다.

template<typename T>
void f3(T& a) {}    // 모든 타입의 lvalue 만 전달 받을수 있습니다.

// T&& 를 사용하면 모든 타입의 lvalue 와 rvalue를 받을수 있습니다.
// int 타입의 lvalue가 전달되면 : T => int&, T&& => int& && 즉, int& 가 됩니다.
// int 타입의 rvalue가 전달되면 : T => int,  T&& => int&& 가 됩니다.
template<typename T> void f4(T&& a) {}

int main()
{
    int n = 10;

    f4(n);  // f4(int&) 함수가 생성됩니다.
    f4(10); // f4(int&&) 함수가 생성됩니다.
}
```

4. forwarding reference 를 사용한 perfect forwarding

"forwarding reference"를 사용하면 완벽한 전달(perfect forwarding)에 필요한 함수를 자동 생성되게 할 수 있습니다.

```
void goo(int& n) { n = 10; }
void hoo(int&& n) {}

template<typename F, typename T> void chronometry(F f, T&& arg)
{
    f(static_cast<T&&>(arg));
}

int main()
{
    chronometry(goo, x);    // chronometry( ..., int& )  함수가 생성됩니다.
    chronometry(hoo, 10);   // chronometry( ..., int&& ) 함수가 생성됩니다.
}
```

▌ 컴파일러가 생성해 주는 코드

결국 위 코드를 컴파일 하면 아래의 코드가 생성됩니다.

```
// 컴파일러가 생성한 chronometry
void chronometry( void(*f)(int&), int& arg)
{
    // lvalue인 arg를 다시 lvalue로 캐스팅합니다. - 없어도 되는 코드입니다.
    // 하지만 캐스팅이 있어도, 문제 되지 않습니다.
    f(static_cast<int&>(arg));
}

void chronometry(void(*f)(int&&), int&& arg)
{
    // arg는 lvalue 이므로 rvalue로 캐스팅해야 합니다. - 반드시 필요합니다.
    f(static_cast<int&&>(arg));
}

int main()
{
```

```
    chronometry(goo, x);
    chronometry(hoo, 10);
}
```

5. std::forward()

chronometry 함수 안에서 원래의 함수로 인자를 전달할 때는 반드시 캐스팅 후에 전달해야 합니다. 이때, 캐스팅하는 코드는 다음과 같은 의미를 가지게 됩니다.

```
template<typename F, typename T> void chronometry(F f, T&& arg)
{
    // chronometry(..., 10) 처럼 rvalue를 전달하면 arg는 lvalue인데,
    // 이것을 다시 rvalue로 캐스팅합니다.

    // chronometry(..., n)  처럼 lvalue를 전달하면 arg는 lvalue인데,
    // 이것을 다시 lvalue로 캐스팅합니다.

    f(static_cast<T&&>(arg));
}
```

▌ std::forward

C++ 표준의 std::forward() 함수는 내부적으로 static_cast<T&&> 를 하는 함수 입니다.

```
template<typename F, typename T> void chronometry(F f, T&& arg)
{
    f(std::forward<T>(arg)); // f(static_cast<T&&>(arg)) 와 동일합니다.
}
```

6. 가변인자 템플릿

지금 까지는 원본 함수의 인자가 한 개인 경우만 다루었습니다.

원본 함수의 인자가 한 개 이상인 함수를 "perfect forwarding" 하려면 chronometry을 가변 인자 템플릿으로 만들어야 합니다.

```
template<typename F, typename ... Types> void chronometry(F f, Types&& ... args)
{
    f(std::forward<Types>(args)...);
}
```

▌ callable object

C++에는 함수 뿐 아니라 함수 객체, 람다 표현식등 함수와 유사하게 ()를 사용해서 호출가능한 다양한 요소가 있습니다.

()를 사용해서 함수 처럼 사용한 것들을 callable object 라고 합니다.(Section 5 참고)

chronometry 가 함수 뿐 아니라 함수 객체(lvalue, rvalue)를 모두 받게 하려면 인자 뿐 아니라 함수도 "forwarding reference" 를 사용해야 합니다.

```
template<typename F, typename ... Types> void chronometry(F&& f, Types&& ... args)
{
    (std::forward<F>(f))(std::forward<Types>(args)...);
}
```

7. perfect forwarding 활용

▌ STL 과 perfect forwarding

C++ 표준 라이브러리인 STL에 있는 많은 요소에서 "Perfect forwarding" 기술을 사용하고 있습니다.

① STL 컨테이너의 emplace_xxx()계열 함수.

② shared_ptr<>의 객체를 만드는 make_shared.

▌ emplace_xxx()

사용자 정의 타입을 저장하는 vector에 요소를 추가하는 것을 생각해 봅시다. push_back() 함수를 사용하는 것 보다 emplace_back() 함수를 사용하는 것이 훨씬 효율적입니다.

```
class Point
{
    int x, y;
public:
    Point(int a = 0, int b = 0) : x(a), y(b) { cout << "Point()" << endl; }
    ~Point() { cout << "~Point()" << endl; }
};
int main()
{
    vector<Point> v;

    // 아래 코드는 결국 Point 객체를 2개 만들게 됩니다.
    Point p(1, 2);  // 1. Point 객체를 생성합니다.
    v.push_back(p); // 2. p를 복사한 복사본을 생성해서 힙 에 있는 버퍼에 추가 됩니다.

    // Point를 생성 할 때 필요한 생성자의 인자만 보내서 vector가 Point를 만들게 합니다.
    // 아래 코드는 Point 객체를 한개만 생성하게 됩니다.
    v.emplace_back(1, 2);  // 1,2를 사용해서 Point객체를 힙에 직접 생성합니다.
}
```

결국, vector에 요소를 추가하기 위해 객체를 외부에서 만든 후에 vector에 전달하지 말고, vector가 바로 객체를 만들게 하면 좋지 않을까요 ? 그런데, 이때 객체를 만들기 위한 생성자의 인자는 완벽하게 전달되어야 합니다.

```
vector<Point> v;
```

```
int a = 1, b = 2;
v.emplace_back(a, b); // a, b는 Point의 생성자로 완벽하게 전달되어야 합니다.
```

make_shared

C++ 표준 스마트 포인터인 shared_ptr◇은 참조 계수 기반으로 객체 관리하게 됩니다. 따라서 shared_ptr◇을 생성하면 참조계수를 관리하는 관리객체가 생성되게 됩니다.

```
int main()
{
    int a = 0, b = 0;
    // 아래 코드는 2번의 메모리 할당이 발생합니다.
    shared_ptr<Point> p(new Point(a, b));   // 1. Point 객체를 생성합니다.
                                            // 2. 참조계수를 관리하기 위해 관리객체를
                                            // 생성합니다.
}
```

이때, make_shared를 사용하면 "객체 크기 + 관리객체 크기" 를 한번에 메모리 할당하기 때문에 메모리를 효율적으로 사용 할 수 있습니다.

```
int main()
{
    // 1. "sizeof(Point) + sizeof(관리객체)" 크기를 한번에 할당합니다.
    // 2. Point(a,b) 생성자를 호출합니다. - placement new 사용
    shared_ptr<Point> p = make_shared<Point>(a, b);
}
```

결국 make_shared 안에서 Point 객체를 만들고 생성자를 호출하게 되는데, 이때 생성자에 전달되는 인자는 완벽하게 전달되어야 합니다. 그래서 make_shared는 "perfect forwarding"기술을 사용하고 있습니다.

```
shared_ptr<Point> p = make_shared<Point>(a, b); // a, b 는 Point 생성자로 완벽하게
                                                // 전달 되어야 합니다.
```

8. perfect forwarding 주의 사항

"perfect forwarding" 기술을 사용할 때는 몇가지 주의할 점이 있습니다.

① 오버로딩된 함수를 전달 할 때는 캐스팅이 필요합니다.

② 포인터 0을 전달하려면 nullptr 을 사용해야 합니다.

```
void foo(int n)    { }
void foo(double d) { }
void goo(int*)     { }

template<typename F, typename T> void chronometry(F f, T&& arg)
{
    f(std::forward<T>(arg));
}

int main()
{
    foo(10);            // ok.. foo(int) 가 호출됩니다.
    chronometry(foo, 10);   // error. 어떤 foo 함수 인지를 알 수 없습니다.
    chronometry(static_cast<void(*)(int)>(foo), 10); // ok

    goo(0);             // ok, 0은 int*로 암시적 변환 됩니다.
    chronometry(goo, 0);    // error. 0은 int이므로 arg는 int 타입입니다.
                        // int 타입은 int* 로 암시적 변환 될수 없습니다

    chronometry(goo, nullptr); // ok.
}
```

SECTION 4

Functor 와 Lambda Expression

항목 4-01

Lambda Expression & Functor 소개

주요 학습 내용

Lambda expression 개념 소개

Function object 개념 소개

1. Lambda Expression 소개

▌ 핵심 개념

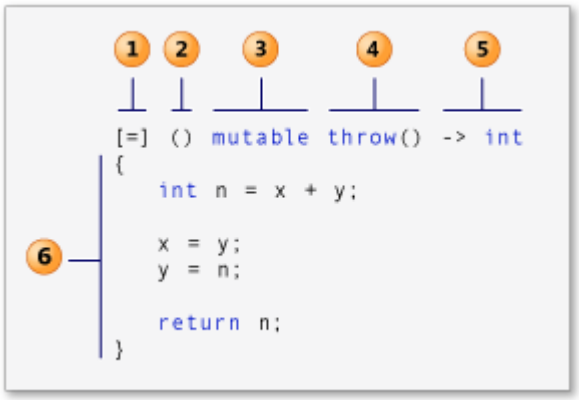
익명의 함수 객체를 만드는 표현. auto 변수에 담아서 함수 처 럼 사용할 수 있다.

▌ 기본 예제

```
int main()
{
    auto f = [](int a, int b) { return a + b; };    // 결국 이 표현은 f라는 함수를
                                                    // 만든 것과 유사합니다.
    cout << f(1, 2) << endl;
}
```

▌ 보충 설명

람다 표현식(Lambda Expression)은 익명의 함수 객체를 만드는 코드 입니다. 람다 표현식을 만드는 방법은 다음과 같습니다.



1. *capture clause* (Also known as the *lambda-introducer* in the C++ specification.)
2. *parameter list* Optional. (Also known as the *lambda declarator*)
3. *mutable specification* Optional.
4. *exception-specification* Optional.
5. *trailing-return-type* Optional.
6. *lambda body*)

또한, 람다 표현식이 시작됨을 알리는 "[]" 를 lambda introducer 라고 합니다

▌ 람다 표현식을 정확히 이해 하려면

람다 표현식을 정확히 이해 하려면, 함수 객체를 이해 해야 합니다. 또한, 함수 객체를 정확히 이해 하려면 인라인 함수를 정확히 이해 해야 합니다.

이어지는 장에서 함수 객체와 인라인 함수에 대해서 먼저 살펴 본 후에, 람다 표현식에 대해서 자세히 살펴 보도록 하겠습니다.

2. 함수 객체(Function Object, Functor) 개념 소개

▌ 핵심 개념

() 연산자를 재정의해서 함수 처럼 사용가능한 객체.

▌ 기본 예제

```
template<typename T> struct plus
{
    T operator()( const T& lhs, const T& rhs ) const
    {
        return lhs + rhs;
    }
};
int main()
{
    plus<int> p;
    int n1 = p(1,2);    // p는 객체지만 함수처럼 사용할 수 있습니다.
                // p.operator()(1,2) 처 럼 해석 됩니다.
    int n2 = p.operator ()(1,2); // 이렇게 사용 할 수도 있습니다.

    cout << n1 << " " << n2 << endl;
}
```

▌ Function Object(Functor) 란 ?

함수 객체(Function Object)는 엄격히 말하면 () 연산자를 사용해서 함수 처 럼 호출 가능한 모든 것을 함수 객체라고 합니다.

"함수, 멤버 함수, 함수 포인터, 멤버 함수 포인터, 함수에 대한 참조, 멤버 함수에 대한 참조, ()연산자를 재정의한 클래스(구조체)"

등이 모두 함수 객체(Function Object) 입니다.

하지만, 좁은 의미로는 보통 () 연산자를 재정의한 클래스(구조체)를 "Function Object 또는 Functor" 라 고 합니다. 이 책에서는 함수 객체를 () 연산자를 재정의 한 클래스(구조체)로 나타내는 의미로 사용하도록 하겠습니다.

▌ Function Object 장점.

일반 함수를 사용하면 되는데, 왜 함수 객체를 사용 할까요 ?

함수 객체는 일반 함수와 비교해서 다음과 같은 장점이 있습니다.

① **상태를 가질 수 있다.**

② **일반 함수는 자신만의 타입이 없다.** signature 가 동일한 모든 함수는 같은 타입이지만, **함수 객체는** signature**가 동일 해도 다른 타입이다.**

③ **상황에 따라 함수 객체는 일반 함수 보다 빠르다.(인라인 치환이 가능하다는 의미)**

이와 같은 장점이 어떤 의미 인지, 다음 항목에서 자세히 살펴 보도록 하겠습니다.

항목 4-02

function object 의 장점

주요 학습 내용

> 상태를 가지는 함수
>
> 함수 객체와 인라인 치환

1. 상태를 가지는 함수

▌ 핵심 개념

함수 객체는 클래스(구조체)이므로 ()연산자 뿐 아니라, 멤버 데이터, 생성자, 복사 생성자 등을 추가로 만들어 활용 할 수 있습니다.

▌ 기본 예제

```
struct Compare
{
    int policy;     // 멤버 데이타 => 함수객체의 상태를 표현

    Compare(int p) : policy(p) {}   // 생성자 => 함수 객체를 초기화

    inline bool operator()(int a, int b)
    {
        return policy ? a < b : a > b;
    }
};
int main()
{
    Compare cp1(0), cp2(1);

    cout << cp1(1, 2) << endl;
    cout << cp2(1, 2) << endl;
}
```

▌ URandom 만들기

함수 객체는 객체이므로 일반 함수와 다르게 멤버 데이터를 가질 수 있고 생성자를 활용 할 수 있습니다. 간단한 예제를 가지고 함수 객체의 장점을 생각해 보도록 하겠습니다.

3개의 서로 다른 임의의 난수를 구하는 예제를 생각해 봅시다. C언어에서 난수를 구하려면 rand() 함수를 사용하면 됩니다. rand() 함수를 사용하기 전에는 반드시 srand() 함수로 난수 초기화를 해야 합니다. 함수 객체는 결국 객체이므로 생성자를 사용하면 초기화 코드를 수행 할 수 있습니다.

```
#include <iostream>
#include <ctime>
```

```
using namespace std;

class URandom
{
public:
    URandom() {  srand(time(0)); } // 함수 객체는 생성자를 활용할 수 있습니다.
    int operator()() {  return rand(); }
};

int main()
{
    URandom r;
    for (int i = 0; i < 10; i++)
        cout << r() % 10 << ' ';
    cout << endl;
}
```

URandom 객체를 생성하면 자동으로 생성자가 호출되므로 자동으로 난수를 초기화 할 수 있습니다. 그런데, 아직 이 함수 객체는 중복된 숫자가 나올 수도 있으므로 항상 중복을 조사해야 합니다.

만약, () 연산자 안에서 난수를 구한 후 바로 리턴 하지 말고, 어딘 가에 보관 해 두면, 다음 난수를 구할 때 중복을 확인할 수 있지 않을 까요 ? 객체는 멤버 데이터가 있으므로 상태를 가질 수 있습니다.

STL bitset

STL의 bitset을 사용하면 데이터를 비트 별로 관리 할 수 있습니다.

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

int main()
{
    bitset<8> bs; // 8 bit를 관리 합니다.

    bs.reset(); // 0000'0000
    bs.set(1);  // 0000,0010
    bs.flip(3); // 0000,1010

    if (bs.test(3))
        cout << "3번째 비트가 set 되었습니다." << endl;
```

```
    string s = bs.to_string();
    cout << s << endl;      // 00001010

    unsigned long n = bs.to_ullong();
    cout << n << endl;      // 10
}
```

▌ URandom 과 bitset

URandom 에 bitset를 멤버 data로 추가하면, () 연산자에서 난수를 구한 후 난수의 정보를 bitset에 기록할 수 있습니다. 그리고, 다음 번에 난수를 구할 때 bitset의 정보를 조사하면 이전에 발생되지 않은 새로운 난수를 구할 수 있습니다. 또한, 더 이상 새로운 난수가 없을 때는

① **더 이상 난수가 없음을 알려주기 위해 -1을 리턴 하거나**

② **bitset을 초기화 해서 다시 사용**

할 수 있습니다. 이 정책을 관리하는 멤버 데이터가 recycle 멤버 입니다.

완성된 코드는 다음과 같습니다.

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

template<int N> class URandom
{
public:
    URandom(bool b = false) : recycle(b)
    {
        srand(time(0));
        random_map.reset(); // bitset의모든비트를1로초기화한다
    }
    int operator()()
    {
        if (random_map.all())
        {
            if (!recycle)
                return -1; // 더 이상 새로운 난수가 없음
            random_map.reset();
```

```
        }
        int n = -1;
        while (1)
        {
            n = rand() % N;
            if (!random_map.test(n))
                continue;
            random_map.set(n);
            break;
        }
        return n;
    }
private:
    bitset<N> random_map;
    bool recycle;
};
int main()
{
    URandom<10> r1; // 0~9 사이의 중복을 허용하지 않는 난수를 구하는 함수객체
                    // 10개의 모든 난수를 사용하면 -1리턴.
    for (int i = 0; i < 15; ++i)
        cout << r1() << " ";
    cout << endl;

    URandom<10> r2(true);
    for (int i = 0; i < 15; ++i)
        cout << r2() << " ";
    cout << endl;
}
```

2. 인라인 치환과 함수 객체

▌ 핵심 개념

일반 함수는 다른 함수의 인자로 사용될 때 인라인 치환 될 수 없지만, 함수 객체는 인라인 치환 될 수 있습니다.

▌ 기본 예제

```
inline bool cmp1(int a, int b) { return a < b; }
inline bool cmp2(int a, int b) { return a > b; }
struct Less    { inline bool operator()(int a, int b) { return a < b; } };
struct Greater { inline bool operator()(int a, int b) { return a > b; } };

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

    // 1. 비교 정책으로 일반함수를 사용할 때
    // 장점 : 정책을 여러번 교체해도 sort()함수가 하나이다.
    // 단점 : 정책의 인라인 치환이 안된다.
    sort(x, x + 10, cmp1);// sort(int*, int*, bool(*)(int, int)) 인 함수 생성
    sort(x, x + 10, cmp2);// sort(int*, int*, bool(*)(int, int)) 인 함수 생성

    // 2. 비교 정책으로 함수객체를 사용 할 때
    // 장점 : 정책의 인라인 치환이 가능.
    // 단점 : 비교정책을 교체한 횟수만큼 sort() 함수 생성
    Less    f1;
    Greater f2;
    sort(x, x + 10, f1); // sort(int*, int*, Less) 인 함수 생성
    sort(x, x + 10, f2); // sort(int*, int*, Greater) 인 함수 생성
}
```

▌ 인라인 함수와 함수 포인터

인라인 함수 문법은 컴파일 시간에 동작하는 문법 입니다. 따라서, 함수 포인터 변수 에 인라인 함수의 주소를 담아서 사용하면 인라인 치환 될 수 없습니다.

```
        int Add1(int a, int b) { return a + b; }
inline  int Add2(int a, int b) { return a + b; }

int main()
{
    Add1(1, 2); // 호출
    Add2(1, 2); // 치환

    int(*f)(int, int) = &Add2;

    // f는 변수 이므로 실행시간에 언제라도 변경 가능 합니다.
    int n;
    cin >> n;

    if (n == 1) f = &Add1;

    f(1, 2); // 호출 됩니다.
}
```

▌ 알고리즘의 정책 함수

특정 함수가 사용하는 알고리즘의 정책을 변경 하기 위해서 함수 인자로 함수 포인터를 사용하는 경우가 있습니다. 이 경우, 정책으로 사용된 함수가 인라인 치환이 될 수 없기 때문에 성능저하가 발생할 수 있습니다.

```
void Sort(int* x, int sz, bool(*cmp)(int, int) )
{
    for (int i = 0; i < sz - 1; i++)
    {
        for (int j = i + 1; j < sz; j++)
        {
            if ( cmp(x[i], x[j]) )  // 인자로 전달된 cmp1이 인라인
                        //함수 지만, 인라인 치환 될 수 없습니다
                swap( x[i], x[j]);
        }
    }
}
//--------------------
inline bool cmp1(int a, int b) { return a < b; }
inline bool cmp2(int a, int b) { return a > b; }
```

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

    Sort(x, 10, cmp1);
}
```

▌ 정책 함수로서의 함수 객체

템플릿과 함수객체를 활용하면 정책 변경이 가능하고 정책 함수의 인라인 치환이 가능한 알고리즘을 만들 수 습니다.

```
struct Less    { inline bool operator()(int a, int b) const { return a < b; }};
struct Greater { inline bool operator()(int a, int b) const { return a > b; }};

// 정책 변경이 가능하고, 정책의 인라인 치환이 되는 Sort
template<typename T> void Sort(int* x, int sz, T cmp)
{
    for (int i = 0; i < sz - 1; i++)
    {
        for (int j = i + 1; j < sz; j++)
        {
            if (cmp(x[i], x[j]))
                swap(x[i], x[j]);
        }
    }
}
int main()
{
    int x[10]{ 1,3,5,7,9,2,4,6,8,10 };

    Less f1;    Sort(x, 10, f1);
    Greater f2; Sort(x, 10, f2);
}
```

항목 4-03

Lambda Expression

주요 학습 내용

Closure Object

Lambda Expression 과 Type

Lambda Expression 과 인라인 치환

Lambda Expression 과 return type

Capturing local variable

Capturing member data

generic lambda expression

nullary lambda

Lambda expression & conversion

1. Closure Object

▌ 핵심 개념

람다 표현식을 사용하면 컴파일러는 자동으로 함수 객체를 생성하게 됩니다. 이 함수 객체를 "클로져 객체(Closure Object)" 라고 합니다.

▌ 기본 예제

```
int main()
{
    // 사용자가 아래처럼 람다 표현식을 사용하면
    sort(x, x + 10, [](int a, int b) { return a < b;  });

    // 컴파일러는 아래 처럼 코드를 생성합니다.
    class Closure
    {
    public:
        inline bool operator()(int a, int b) const
        {
            return a < b;
        }
    };
    sort(x, x + 10, Closure());;
}
```

▌ 람다 표현식은 함수 객체를 만드는 표현식.

사용자가 C++ 코드에서 람다 표현식을 사용하면 컴파일러는 람다 표현식의 구현 코드와 동일한 모양의 ()연산자 함수를 가지는 함수 객체를 생성하게 됩니다.

```
int main()
{
    auto f = [](int a, int b) { return a + b; };

    // 위코드는 아래 코드와 동일합니다.
    class Closure
    {
```

```
    public:
        inline int operator()(int a, int b) const
        {
            return a + b;
        }
    };
    auto f = Closure();
}
```

결국 람다 표현식은 컴파일러에게 함수 객체 코드를 생성하게 하는 문법입니다.

2. Lambda Expression 과 타입

▌ 핵심 개념

구현이 동일 해도 모든 람다 표현식은 다른 타입 입니다.

▌ 기본 예제

```
int main()
{
    auto f1 = [](int a, int b) { return a + b; };
    auto f2 = [](int a, int b) { return a + b; };
    cout << typeid(f1).name() << endl;
    cout << typeid(f2).name() << endl;
}
```

▌ 모든 람다 표현식은 다른 타입이다.

람다 표현식은 함수 객체를 만드는 표현식 입니다. 람다 표현식에 의해 생성된 함수 객체의 클래스 이름은 컴파일러가 임의로 만들게 됩니다. 이때, 구현이 동일한 람다 표현식도 다른 이름을 가진 클래스를 생성하게 되므로 모든 람다 표현식은 다른 타입입니다.

따라서, 람다 표현식으로 초기화된 auto 형 변수에 다른 람다 표현식을 대입 할수 없습니다.

```
int main()
{
    auto f1 = [](int a, int b) { return a + b; };
    f1 = [](int a, int b) { return a + b; };// error. f1의 타입과 우변의
                        // 람다 표현식은 다른 타입입니다.
}
```

3. Lambda Expression 과 inline 치환

▌ 핵심 개념

람다 표현식은 auto 변수, 함수 포인터, function 등의 담을 수 있습니다. 이중, auto 를 사용한 경우만 인라인 치환이 됩니다.

▌ 기본 예제

```
int main()
{
    auto f1           = [](int a, int b) { return a + b; };
    int(*f2)(int, int)    = [](int a, int b) { return a + b; };
    function<int(int,int)> f3 = [](int a, int b) { return a + b; };

    f1 = [](int a, int b) { return a + b; }; // error
    f2 = [](int a, int b) { return a + b; }; // ok
    f3 = [](int a, int b) { return a + b; }; // ok

    f1(1, 2); //  인라인 치환 됨.
    f2(1, 2); //  치환 안됨.
    f3(1, 2); //  치환 안됨.
}
```

▌ 람다 표현식을 담는 변수

람다 표현식은 함수 포인터로 암시적 형 변환 됩니다. 하지만, 이경우, 함수 포인터형의 변수는 “변수” 이기 때문에 실행 시간에 값을 변경 할 수 있습니다. 따라서, 인라인 치환될 수 없습니다. 하지만, auto 에 담은 람다 표현식의 경우, auto 로 선언했던 변수에 다른 람다 표현식을 담을 수 없으므로 인라인 치환이 가능합니다.

▌ 람다 표현식과 함수 인자

람다 표현식을 함수 인자로 전달한 경우, 함수 포인터나 function을 사용해서 받을 수 있습니다. 하지만 이경우 인라인 치환이 되지 않습니다.

```
void foo( int(*f)(int, int) ) // ok. 하지만 f를 사용하면 인라인 치환이 되지는 않습니다.
```

```
{
}

int main()
{
    foo( [](int a, int b) { return a + b; } );
    foo( [](int a, int b) { return a - b; } );
}
```

람다 표현식을 함수 인자로 받을 때, 인라인 치환이 되게 하려면 템플릿을 사용해야 합니다.

```
template<typename T> void foo(T f)
{
    f(1, 2);  // ok. 인라인 치환됨.
}
int main()
{
    foo( [](int a, int b) { return a + b; } );
    foo( [](int a, int b) { return a - b; } );
}
```

참고로 auto 는 함수 인자로 사용될 수 없습니다.

```
void foo( auto f)  // error. auto는 함수 인자로 사용 될 수 없습니다.
{
}
```

4. Lambda Expression 과 return type

▌ 핵심 개념

람다 표현식 작성시 return 문이 2개 이상 있고, 서로 다른 타입을 리턴 하는 경우 후위형 반환 타입(suffix return type) 구문을 사용해서 리턴 타입을 표현해야 합니다.

▌ 기본 예제

```
auto f = [](int a, double b)-> double
{
    if (a == 1)
        return a;
    else
        return b;
};
```

▌ 람다 표현식의 리턴 타입

람다 표현식은 결국 함수 객체의 () 연산자 함수를 만드는 모양입니다. 모든 함수는 리턴 타입이 있듯이 람다 표현식도 리턴 타입을 표기해야 합니다. 후위형 반환 타입 구문(suffix return type syntax)을 사용해서 리턴 타입을 표현 해야 합니다.

```
auto f1 = [](int a, int b) -> int { return a + b; };
```

하지만,

① **리턴 문장이 하나 이거나**

② **2개이상의 리턴 문장이 있지만, 모두 같은 타입을 리턴 한다면**

리턴 타입 표기를 생략할 수 있습니다.

5. Capturing Local Variable

▌ 핵심 개념

람다 표현식에서는 전역 변수는 접근 할 수 있지만, 지역변수는 캡쳐(Capture)해야만 사용할 수 있습니다.

▌ 기본 예제

```
int g = 0;

int main()
{
    int v1 = 10;
    int v2 = 10;

    auto f1 = [=, &v2](int a) mutable { g = a; v1 = a; v2 = a; };

    f1(20);

    cout << g  << endl; // 20
    cout << v1 << endl; // 10
    cout << v2 << endl; // 20
}
```

▌ 값 캡쳐 (default capture by value)

“[=]” 를 사용하면 모든 지역변수를 값으로 캡쳐를 할 수 있습니다. 값으로 지역변수를 캡쳐 한 경우 값을 변경할 수는 없습니다.

```
int main()
{
    int v1 = 10, v2 = 10;

    // "[=]" 모든 지역변수를 값으로 캡쳐 합니다.
    auto f1 = [=](int a) { return a + v1 + v2; };

    auto f2 = [=](int a) { v1 = a; };  // error. v1, v2를 변경할 수 없습니다.
```

```
}
```

결국 람다 표현식은 컴파일러에게 함수 객체 코드를 생성하게 하는 문법입니다.

▌ 값 캡쳐 (default capture by value) 의 원리

값으로 지역변수를 캡쳐한 경우 "클로져 타입" 안에는 지역변수의 복사본을 보관할 멤버 데이터가 생성됩니다. 또한, () 연산자 함수는 상수 함수로 만들어 지기 때문에 멤버 데이터를 변경할 수 없기 때문에 에러가 발생합니다.

```
int main()
{
    int v1 = 10, v2 = 10;

    auto f1 = [=](int a) { v1 = a; };  // error. v1, v2를 변경할 수 없습니다.

    // 위 코드의 원리는 다음과 같습니다.
    class Closure
    {
        int v1;
        int v2;
    public:
        Closure(int a, int b) : v1(a), v2(b) {}

        inline void operator()(int a) const
        {
            v1 = a; // error. 상수 멤버함수 안에서 멤버 데이터의 값을
                    //변경 할 수 없습니다.
        }
    };
    auto f1 = Closure(v1, v2);
}
```

이 경우 에러가 발생하지 않게 하려면 () 연산자 함수를 비 상수 함수로 만들면 됩니다.

▌ mutable 람다.

람다 표현식을 mutable 로 만들면 () 연산자 함수를 비 상수 함수로 만들 수 있습니다. 따라서, 람다 표현시에서 캡쳐 된 지역변수의 값을 변경할 수 있습니다. 하지만, 이경우 복사된 멤버 데이터를 변경하게 되므로 원본 지역변수는 변경 되지 않습니다.

```
int main()
{
    int v1 = 10;

    auto f1 = [=](int a) mutable { v1 = a; }; // ok. 하지만 v1의 복사본이 변경됩니다.

    f1(20);

    cout << v1 << endl; // 10
}
```

▌ 참조 캡쳐 (default capture by reference)

"[&]" 를 사용하면 지역 변수를 참조 로 캡쳐 할 수 있습니다.지역 변수를 참조로 캡쳐 할 경우 지역 변수의 복사본이 아닌 원본을 변경할 수 있습니다.

```
int main()
{
    int v1 = 10, v2 = 10;

    auto f1 = [&](int a) { v1 = a; };   // 지역변수 v1이 변경됩니다.

    f1(20);

    cout << v1 << endl; // 20
}
```

지역변수를 참조로 캡쳐 할 경우 클로져는 지역변수를 참조로 보관하고 있게 됩니다.

```
int main()
{
    int v1 = 10, v2 = 10;

    class Closure
    {
        int& v1;
        int& v2;
    public:
        Closure(int& a, int& b) : v1(a), v2(b) {}
```

```
        inline void operator()(int a) const
        {
            v1 = a; // ok. 상수 멤버 함수는 멤버의 값을 변경할 수 없지만
                    //     멤버가 가리키는 곳의 값은 변경 할 수 있습니다.
        }
    };
    auto f1 = Closure(v1, v2);
    f1(20);
    cout << v1 << endl;
}
```

특정 변수만 캡쳐 하는 방법

모든 지역 변수가 아닌 특정 지역 변수만 캡쳐 하려면 변수 이름을 사용하면 됩니다.

```
int main()
{
    int v1 = 10, v2 = 10;

    [=]() {}; // 모든 지역변수를 값으로 캡쳐 합니다.
    [&]() {}; // 모든 지역변수를 참조로 캡쳐 합니다.

    [v1]() {};  // 지역변수 v1을 값으로 캡쳐 합니다.
    [&v1]() {}; // 지역변수 v1을 참조로 캡쳐 합니다.

    [v1, &v2]() {};  // v1은 값으로 v2는 참조로 캡쳐 합니다.
    [=,  &v2]() {};  // 모든 지역변수는 값으로 v2만 참조로 캡쳐 합니다.
    [&,   v1]() {};  // 모든 지역변수는 참조로 v1만 값으로 캡쳐 합니다.

    [&, &v1]() {}; // error,

    [&a = v1]() { a = 20; };// v1을 참조로 캡쳐하는데, 멤버 변수이름을 a로
                // 만듭니다. 결국, v1이 20이 됩니다.
}
```

6. Capturing Member Data

멤버 데이터를 캡쳐 할 때는 "[this]" 또는 "[=]" 를 사용하면 됩니다. 객체의 복사본이 아닌 주소를 캡쳐 하게 되므로 멤버 데이터의 값을 변경 할 수 있습니다.

```
class Test
{
    int data;
public:
    void foo()
    {
        auto f = [this](int a) { data = a; };
        f(20);
        cout << data << endl; // 20
    }
};
int main()
{
    Test t;
    t.foo();
}
```

멤버 함수 안에서 람다 표현식을 만들 때 멤버 변수에 접근하기 위해서는 다음 중 하나를 사용합니다.

① [this] : **객체의 주소를 캡쳐 합니다.**

② [=] : **모든 지역변수를 캡쳐 합니다. this도 결국은 멤버 함수의 지역변수 이므로 캡쳐 됩니다.**(3장 thiscall 참고)

[this] 또는 [=] 를 사용하는 경우, 결국은 포인터를 캡쳐 하는 것이므로 멤버 데이터의 값을 변경 할 수 있게 됩니다.

▌ [*this] 캡쳐 - C++17

C++17 부터는 this를 값으로 캡쳐 할 수 도 있습니다. [*this] 를 사용하면 됩니다.

7. Generic Lambda Expression

▌ 핵심 개념

auto는 함수 인자로 사용 될 수 없지만 람다 표현식에서는 인자로 auto를 사용할 수 있습니다.

▌ 기본 예제

```
int main()
{
    auto f = [](auto a, auto b) { return a + b; };

    cout << f(1, 2.5) << endl;
}
```

▌ 원리

람다 표현식의 인자로 auto를 사용할 경우 클로져의 ()연산자는 템플릿으로 만들어 지게 됩니다.

```
class Closure
{
public:
    template<typename T, typename U>
    inline auto operator()(T a, T b) -> decltype(a + b)
    {
        return a + b;
    }
};
```

8. nullary lambda

▌ 핵심 개념

람다 표현식에 인자가 없을 경우, ()를 생략할 수 있습니다.

```
int main()
{
    auto f1 = [](int a) { return a; } // 인자가 한 개인 람다 표현식
    auto f2 = []()      { return 0; } // 인자가 없는 람다 표현식
    auto f3 = []        { return 0; } // 인자가 없는 경우 ()를 생략할 수 있습니다.
}
```

9. multi interface lambda expression

▌ 핵심 개념

동일한 이름의 서로 다른 인자의 개수를 가지는 람다 표현식을 만드는 테크닉

① 모든 람다 표현식은 closure 라 불리는 함수 객체를 생성한다.

② C++ 17부터는 class template 도 생성자 인자를 통해서 컴파일러가 type deduction 을 수행할 수 있다.

는 문법을 가지고 만든 테크닉

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

template<typename ... Types> class mi : public Types... // public lambda1, public
lambda2
{
public:
    mi(Types&& ... args) : Types(args)... {}

    // 기반 클래스의 특정함수를 사용할수 있게..
    using Types::operator()...;
};

int main()
{
    mi f([](int a, int b) { return a + b; } // class lambda1{}; lambda1();
    [](int a, int b, int c) { return a + b + c; }); // class lambda2{}; lambda2();

    cout << f(1, 2) << endl; // 3
    cout << f(1, 2, 3) << endl; // 6
}
```

Section 5

Template Specialization

항목 5-01

Specialization

▌ 이번 장에서 배우게 되는 내용

특수화 (Specialization)

부분 특수화 (Partial specialization)

부분 특수화의 다양한 모양

특수화 와 디폴트 파라미터

1. Specialization의 개념과 기본 모양

클래스 템플릿은 결국 클래스를 만드는 틀입니다. 컴파일 시간에 전달된 타입을 가지고 클래스를 생성하게 됩니다. 이때, 특정 타입의 대해 다른 틀(템플릿)을 사용해서 클래스를 만들려면 "특수화(Specialization)" 또는 "부분 특수화(Partial Specialization)"을 사용하면 됩니다.

[참고] Specialization은 "특수화" 또는 "전문화"로 번역됩니다.

다음 코드를 생각해 봅시다.

```
// primary template : 기본 적으로는 아래의 템플릿을 사용해서 코드를 생성합니다.
template<typename T> class Stack
{
    T data;
public:
    void push(T a) { cout << "T" << endl; }
};

// 부분 특수화(Partial Specialization)
// T가 포인터 타입인 경우는 아래 템플릿을 사용해서 코드를 생성합니다.
template<typename T> class Stack<T*>
{
    T data;
public:
    void push(T a) { cout << "T*" << endl; }
};

// 특수화(Specialization)
// T가 char* 일 때 아래 템플릿을 사용해서 코드를 생성합니다.
template<> class Stack<char*>
{
    char* data;
public:
    void push(char* a) { cout << "char*" << endl; }
};
int main()
{
    Stack<int>    s1; // primary template을 사용해서 코드를 생성.
    Stack<int*>   s2; // partial specialization 버전을 용해서 코드 생성
    Stack<char*>  s3; // specilization 버전을 사용해서 코드 생성.

    s1.push(0);
```

```
    s2.push(0);
    s3.push(0);
}
```

한 줄씩 코드를 자세히 살펴 보도록 하겠습니다.

Primary template

```
template<typename T> class Stack
{
    T data;
public:
    void push(T a) { cout << "T" << endl; }
};
```

기본 모양의 템플릿 입니다. "특수화(부분 특수화)" 되지 않은 모든 타입은 이 템플릿을 사용해서 코드를 생성합니다.

다음의 코드는 위의 템플릿을 사용해서 코드를 생성합니다.

```
Stack<int>    s1; // primary template 을 사용해서 코드를 생성.
```

부분 특수화 (Partial Specialization)

```
template<typename T> class Stack<T*>
{
    T data;
public:
    void push(T a) { cout << "T*" << endl; }
};
```

템플릿 인자가 포인터 일 경우에는 위 템플릿을 사용해서 코드를 생성합니다. 다음의 코드는 위의 템플릿을 사용해서 코드를 생성합니다.

```
Stack<int*>   s2; // partial specialization 버전을 사용해서 코드 생성
```

▌ 특수화 (Specialization)

```
template<> class Stack<char*> // 특징, "typename T" 없습니다.
{
    char* data;
public:
    void push(char* a) { cout << "char*" << endl; }
};
```

템플릿 인자가 "char*"의 경우에는 위 템플릿을 사용해서 코드를 생성합니다. 이경우 임의의 타입이 아닌 특정 타입으로 확정 되었으므로 "typename T"를 표기하지 말아야 합니다. 다음의 코드는 위의 템플릿을 사용해서 코드를 생성합니다.

```
Stack<char*>   s2; // specialization 버전을 사용해서 코드 생성
```

▌ Specialization 과 코드 메모리

위 코드에서 Stack 템플릿은 결국 3개의 틀(Primary template, Partial Specialization, Specialization)을 제공하고 있습니다. 틀(템플릿)이 여러 개 제공된다고 해서 생성되는 목적 코드 (실행파일등)의 크기가 커지는 것은 아닙니다. 결국 템플릿은 코드를 생성하는 "틀(template)"일 뿐이므로, 여러 개의 템플릿을 제공해도 사용자가 한 가지 타입으로만 사용했다면 생성되는 실제 클래스는 하나만 만들어질 뿐입니다.

```
template<typename T> class Stack        { ..... };
template<typename T> class Stack<T*>    { ..... };
template<>           class Stack<char*> { ..... };

int main()
{
    // 3개의 stack template 이 있지만 한가지 타입으로만 사용한다면
    // 실제 생성된 코드는 int 버전의 stack 클래스 하나만 있을 뿐입니다.
    Stack<int>    s1;
}
```

▌ 멤버 함수와 Specialization

특정 타입에 대해서 클래스 전체가 아닌 일부 멤버 함수만 Specialization을 사용할 수 있습니다. 다음 코드에서 멤버 함수 pop() 은 모든 타입에 대해서 동일 하게 사용하지만 push() 멤버 함수는 char* 에 대해서 다른 구현을 사용할 수 있습니다.

```
template<typename T> class Stack
{
    T data;
public:
    void pop() {}
    void push(T a);
};
template<typename T> void Stack<T>::push(T a)
{
    cout << "T" << endl;
}
// char* 타입에 대해서 push()함수만 Specialization 합니다.
template<> void Stack<char*>::push(char* a)
{
    cout << "char*" << endl;
}
int main()
{
    Stack<int>    s1; s1.push(0);
    Stack<char*>  s2; s2.push(0);   // primary template을 사용하지만
                                    // push 함수만 특수화(Specialiazation) 버전을 사용.
}
```

하지만, 특정 멤버 함수에 대해서 "Specialization" 이 아닌, "Partial Specialization"을 사용할 수는 없습니다. "Partial Specialization"을 사용하려면 클래스 템플릿 전체를 새롭게 만들어야 합니다.

```
template<typename T> void Stack<T*>::push(T a) // error
{
}
```

2. 다양한 모양의 Specialization

이제, 간단한 예제를 가지고 "Specialization"에 대해서 좀더 자세히 살펴 보도록 하겠습니다. 다음 코드에서 Object 클래스의 foo() 를 호출하면 화면에 "T, U" 로 출력 하게 됩니다.

```
template<typename T, typename U> struct Object
{
    void foo() { cout << "T, U" << endl; }
};
```

이제 Partial Specialization과 Specialization을 사용해서 아래 처럼 실행될 수 있도록 만들어 보겠습니다

```
int main()
{
    Object<int, double>     ob1; ob1.foo(); // T,  U
    Object<int*, double*>   ob2; ob2.foo(); // T*, U*
    Object<int*, double>    ob3; ob3.foo(); // T*, U
    Object<int, int>        ob4; ob4.foo(); // T, T
    Object<int, short>      ob5; ob5.foo(); // T, short
    Object<short, short>    ob6; ob6.foo(); // short, short

    Object<int, Object<double, short>> ob7; ob7.foo(); // A, Object<B, C>
}
```

▌ T*, U*

템플릿 인자 2개가 모두 포인터일 때 "T*, U*"로 출력이 되도록 하려면 아래 처럼 "부분 특수화 (Partial Specialization)" 을 해야 합니다.

```
// template parameter 2개가 모두 포인터 일 때
template<typename T, typename U> struct Object<T*, U*>
{
    void foo() { cout << "T*, U*" << endl; }
};
Object<int*, double*> ob2;
ob2.foo(); // T*, U*
```

T*, U

이 경우는 템플릿의 1번째 인자는 포인터 이고 2번째 인자는 임의의 타입일 때 입니다. 다음 처럼 만들면 됩니다.

```
// template parameter의 1번째만 포인터 일 때
template<typename T, typename U> struct Object<T*, U>
{
    void foo() { cout << "T*, U" << endl; }
};
Object<int*, double> ob3;
ob3.foo(); // T*, U
```

T, T

이 경우는 2개의 템플릿의 인자가 같은 경우 입니다.

```
// 1번째, 2번째 template parameter가 동일한 타입일 때
template<typename T> struct Object<T, T>
{
    void foo() { cout << "T, T" << endl; }
};
Object<int, int> ob4;
ob4.foo();  // T, T
```

여기서 중요한 점은 primary template 은 템플릿 인자가 2개이지만 Partial Specialization 할 때는 템플릿 인자의 개수를 다르게 할 수 있다는 점입니다.

```
// primary template 은 템플릿 인자가 2개 입니다.
template<typename T, typename U> struct Object {};

// Partial Specialization할때는 템플릿 인자의 갯수를 다르게 할 수 있습니다.
template<typename T> struct Object<T, T> // 하지만, 이 부분은 반드시 인자가 2개로
{                                        // 표기해야 합니다.
};
```

T, short

이 경우는 템플릿의 1번째 인자는 임의의 타입이고, 2번째 인자는 short 인 경우 입니다. 역시 임의의 타입이 한 개 있으므로 템플릿 인자를 1개 받으면 됩니다.

```
// 임의의 타입은 한 개만 있으면 됩니다.
template<typename T> struct Object<T, short>
{
    void foo() { cout << "T, short" << endl; }
};

Object<int, short> ob5;
ob5.foo(); // T, short
```

여기서, 또 하나 중요한 점은 아래 처럼 2개의 Partial Specialization 버전이 있을 때 입니다.

```
template<typename T> struct Object<T, T>     {}; // A
template<typename T> struct Object<T, short> {}; // B

Object<short, short> ob6; // error, A 를 사용할수도 있고, B 를 사용할수도 있습니다.
```

Object<short, short>의 경우 <T,T> 도 만족하고 <T,short> 도 만족합니다. 따라서, 컴파일러는 "Ambiguos" 에러를 발생하게 됩니다.

이 경우 해결책은, <short, short>를 위한 Specialization 버전을 제공하면 됩니다.

short, short

이 경우는 2개의 인자가 모두 short 인 경우 입니다. 임의의 타입은 없고, 2개의 타입이 모두 특정 타입으로 결정 되었으므로 "Specialization"을 사용합니다.

```
template<> struct Object<short, short>
{
    void foo() { cout << "short, short" << endl; }
};
Object<short, short>    ob6;
ob6.foo(); // short, short
```

A, Object<B,C>

약간 복잡한 경우 입니다.

아래와 같이 Object가 2번째 템플릿 인자로 다시 Object를 받는 모양입니다.

```
Object<int, Object<double, short>> ob7;
```

결국 이 표현은

```
Object<임의의 타입, Object< 임의의 타입 ,  임의의 타입 >> ob7;
```

이므로 템플릿 인자는 3개가 되어야 합니다.

```
// primary template 은 parameter가 2개 이지만
// Partial Specialization 버전은 parameter의 개수를 다르게 할 수 있습니다.
template<typename A, typename B, typename C> struct Object<A, Object<B, C>>
{
    void foo() { cout << "A, Object<B, C>" << endl; }
};

Object<int, Object<double, short>> ob7;
ob7.foo(); // A, Object<B, C>
```

완성된 코드

다음은 완성된 전체 코드 입니다.

```
template<typename T, typename U> struct Object
{
    void foo() { cout << "T, U" << endl; }
};
template<typename T, typename U> struct Object<T*, U*>
{
    void foo() { cout << "T*, U*" << endl; }
};
template<typename T, typename U> struct Object<T*, U>
```

```
{
    void foo() { cout << "T*, U" << endl; }
};
template<typename T> struct Object<T, T>
{
    void foo() { cout << "T, T" << endl; }
};
template<typename T> struct Object<T, short>
{
    void foo() { cout << "T, short" << endl; }
};
template<> struct Object<short, short>
{
    void foo() { cout << "short, short" << endl; }
};
template<typename A, typename B, typename C> struct Object<A, Object<B, C>>
{
    void foo() { cout << "A, Object<B, C>" << endl; }
};
int main()
{
    Object<int, double>     ob1; ob1.foo(); // T,  U
    Object<int*, double*>   ob2; ob2.foo(); // T*, U*
    Object<int*, double>    ob3; ob3.foo(); // T*, U
    Object<int, int>        ob4; ob4.foo(); // T, T
    Object<int, short>      ob5; ob5.foo(); // T, short
    Object<short, short>    ob6; ob6.foo(); // short, short
    Object<int, Object<int, short>> ob7; ob7.foo(); // A, Object<B, C>
}
```

3. Specialization 과 default parameter

템플릿이 default parameter를 가질 때 default 값의 표현은 primary template 에만 지정해야 합니다. partial specialization 버전에서는 디폴트 값을 표시 하지 않습니다.

```
// primary template 에는 디폴트 값을 표시합니다.
template<typename T, int N = 10 > class Stack {};

// 아래 코드는 error. 부분 특수화 버전은 디폴트 값을 표시하면 안됩니다.
// template<typename T, int N = 10>  class Stack<T*, N> {};

// ok.  디폴트 값을 표시하지 않았지만 인자를 전달하지 않으면 N = 10 이 됩니다.
template<typename T, int N>  class Stack<T*, N> {};

int main()
{
    Stack<int*> st; // N = 10 이 됩니다.
}
```

C++11 표준인 unique_ptr<> 은 다음과 같이 선언되어 있습니다.

```
template<typename T, typename D = default_delete<T>> class unique_ptr {};
template<typename T, typename D>                     class unique_ptr<T[], D> {};
```

핵심 개념 정리

- ☑ Primary template, Specialization, Partial Specialization 기본 모양
- ☑ 부분 특수화(Partial Specialization)의 다양한 모양
- ☑ Primary template 의 파라미터가 2 개 라도, 부분 특수화 버전의 파라미터의 버전을 다르게 만들 수 있습니다.
- ☑ 부분 특수화 버전을 만들 때 임의의 타입이 몇 개 필요한지를 결정하는 것이 중요합니다.
- ☑ Template 에 default parameter 가 있을 경우, primary template 에만 표시 하면 됩니다. 특수화, 부분 특수화 버전에는 default parameter 를 표시 하지 않습니다.

항목 5-02

Specialization 활용

▌ 이번 장에서 배우게 되는 내용들.

IfThenElse

couple template

tuple 만들기

1. IfThenElse

이번 장에서는 "Specialization" 이나 "Partial Specialization"을 활용한 다양한 기법들을 살펴 보도록 하겠습니다. 먼저 IfThenElse template 입니다

▌ Bit<> 구조체

N 비트를 관리하기 위한 Bit 라는 구조체를 생각해 봅시다.

```
template<int N> struct Bit
{
    using data_type = unsigned int;
    data_type data;

    // N bit를 관리를 하기 위한 다양한 멤버 함수들..
};
int main()
{
    Bit<8>  b1; // 8bit를 관리하기 위한 객체
    Bit<32> b2; // 32bit를 관리하기 위한 객체
}
```

위 코드는 N bit의 data를 관리하기 위해 아래의 data type 을 사용하고 있습니다. 즉, 4바이트(32bit)를 사용하고 있습니다.

```
using data_type = unsigned int;
```

하지만, Bit<8> 처럼 8비트만을 관리할 때는 4바이트가 아닌 1바이트만 있어도 충분합니다. 결국, Bit<> 를 사용할 때 템플릿 인자로 전달된 N의 크기가 1~8 일 때는 char, 9~16일때는 unsigned short, 17~32 일 때는 unsigned int를 사용하는 것이 가장 좋습니다.

이처럼 조건에 따라 다른 타입을 선택하고 싶을 때가 있습니다. 이럴 때에는 IfThenElse 라는 간단한 template을 사용하면 해결 할 수 있습니다.

▌ IfThenElse

다음 코드를 생각해 봅시다.

```
template<bool, typename T, typename U> struct IfThenElse
{
    using type = T;
};
// 1번째 인자가 false일때는 위한 부분 전문화
template<typename T, typename U> struct IfThenElse<false, T, U>
{
    using type = U;
};
int main()
{
    IfThenElse<true,  int, double>::type t1; // int, true이면 type은 int가 됩니다.
    IfThenElse<false, int, double>::type t2; // double, false이면 type은 double.
}
```

먼저 IfThenElse<> 템플릿은 bool 값과 T, U 라는 2개의 임의의 타입을 인자로 가지는 템플릿입니다. IfThenElse 안에 있는 type은 기본적으로 T 와 동일한 타입 입니다. 하지만, 부분 전문화(Partial Specialization) 버전에 따라서 1번째 인자가 false일때는 type은 U 와 동일한 타입이 됩니다.

따라서, 아래와 같이 사용될 수 있습니다.

```
IfThenElse<true,  int, double>::type t1; // int,    true이면 type은 int가 됩니다.
IfThenElse<false, int, double>::type t2; // double, false 이면 type 은 double 이 됩니다.
```

결국 IfThenElse<> 는 다음과 같은 의미를 가지게 됩니다.

```
IfThenElse<조건, 타입1, 타입2>::type t3;    // 조건이 참이면 type은 "타입1",
                                            // 거짓이면 "타입 2"가 됩니다.
```

▌ Bit<> 와 IfThenElse<>

IfThenElse<>를 사용하면 Bit 구조체는 아래 처럼 변경 할 수 있습니다.

```
template<int N> struct Bit
{
    using data_type = typename IfThenElse<(N <= 8), char, unsigned int>::type;
    data_type data;
};
```

IfThenElse<>는 다양한 C++기반 오픈소스에서 널리 사용하고 있습니다. 일부 오픈소스에서는 Select<> 이름으로 만들어져서 사용되기도 합니다. 안드로이드 프레임워크 소스 내에도 Select<>라는 이름으로 사용되고 있습니다.

C++11 – std::conditional

IfThenElse<>의 개념은 C++11 에서 conditional<> 이라는 이름으로 표준에 추가 되었습니다.

따라서, 사용자가 만들 필요 없이 <type_traits> 헤더를 포함하면 사용할 수 있습니다.

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

int main()
{
    typedef conditional<true, int, double>::type Type1;     // int
    typedef conditional<false, int, double>::type Type2;    // double
    typedef conditional<sizeof(int) >= sizeof(double), int, double>::type Type3;

    cout << typeid(Type1).name() << '\n';
    cout << typeid(Type2).name() << '\n';
    cout << typeid(Type3).name() << '\n';
}
```

또한, C++14 부터는 conditional을 간결하게 사용할 수 있는 conditional_t 를 제공하고 있습니다.

```
template< bool B, class T, class F >
using conditional_t = typename conditional<B,T,F>::type;
```

연습문제

앞에서 만든 Bit<>는 조건에 따라 char 또는 unsigned int를 사용하고 있습니다. 코드를 수정해서 조건에 따라 3개의 타입(char, unsigned short, unsigned int) 를 사용할 수 있게 코드를 수정해 보세요.

IfThenElse<> 대신 C++11 표준인 std::conditional 을 사용해 보세요.

2. couple

부분 특수화를 사용하는 또다른 예제를 살펴 보도록 하겠습니다.

couple template

아래의 couple template은 임의의 타입 객체를 2개 보관할 수 있습니다. STL의 pair와 유사한 template 입니다. couple 안에 있는 N은 couple이 몇 개의 객체를 보관하고 있는지를 나 타냅니다.

printN() 함수 템플릿은 couple 객체가 몇 개의 객체를 보관하고 있는지 N을 출력하는 함수 입니다.

```
template<typename T> void printN(const T& c)  { cout << T::N << endl; }

template<typename T, typename U> struct couple
{
    T v1;
    U v2;
    static constexpr int N = 2;
};
int main()
{
    couple<int, double> c2;

    printN(c2); // 2
}
```

couple은 서로 다른 타입의 객체를 2개를 보관할 수 있습니다. 그런데, C++에서는 템플릿의 인자로 자기 자신도 넣을 수 있기 때문에 couple을 아래 처럼 사용하면 2개 뿐 아니라 원하는 만큼 보관할 수가 있습니다.

```
int main()
{
    couple<int, couple<int, int>> c3;
    couple<int, couple<int, couple<int, int>>> c4;

    printN(c3);
    printN(c4);
}
```

그런데, 문제는 c3의 경우 3개의 값을 보관하므로 3, c4는 4가 출력되어야 하는데 아직까지는 N = 2 로

되어 있으므로 2가 출력되게 됩니다.

이제 부분 특수화를 사용해서 개수를 정확히 출력 될 수 있도록 해야 합니다.

▌ 2번째 인자가 자기 자신인 경우를 위한 부분 특수화

아래 코드를 모양을 보고 부분 특수화를 어떻게 해야 할지 생각해 봅시다.

```
// 아래 코드에서 int 자리는 int 뿐 아니라 다른 타입도 가능합니다.
couple<int, couple<int, int>> c3;
```

결국 위 코드의 모양은 couple<"타입1", couple<"타입2", "타입3"> > 의 모양일 경우 이므로 임의의 타입은 3개를 전달 받도록 만들어야 합니다.

```
template<typename A, typename B, typename C> struct couple<A, couple<B, C>>
{
    A           v1;
    couple<B,C> v2;

    static constexpr int N = couple<B, C>::N + 1;   // 이 부분이 핵심 입니다.
                                                    // N = 3 이라고 하면 안됩니다.
};
couple<int, couple<int, int>> c3;
couple<int, couple<int, couple<int, int>>> c4;
```

위 코드에서 핵심은 N = 3이 아니라는 점입니다. c3의 경우는 3개지만, c4의 경우는 4개 이므로 반드시 아래 처럼 표현해야 합니다.

```
static constexpr int N = couple<B, C>::N + 1;
```

▌ 연습 문제

1) couple 의 1 번째 인자가 자기 자신일 경우에 N 의 값이 올바르게 출력 되도록 부분 특수화를 추가해 보세요.

```
int main()
{
    couple<couple<int, int>, int> c3;
    couple<couple<couple<int, int>, int>, int> c4;

    printN(c3); // 3
    printN(c4); // 4
}
```

2) couple 의 1 번째, 2 번째 인자가 모두 자기 자신일 경우에 올바르게 출력 되도록 부분 특수화를 추가해 보세요.

```
int main()
{
    couple<couple<int, int>, couple<int, int>> c4;
    printN(c4); // 4
}
```

3. tuplue

couple template은 2개 뿐 아니라 사용자가 원하는 만큼의 data를 보관할 수 있습니다. 하지만, 사용하기에는 좀 복잡해 보입니다.

```
couple<couple<int, int>, couple<int,  couple<int, int>>> c5;
```

상속과 부분 특수화 기술을 사용하면 여러 개의 객체를 보관하는 couple을 쉽게 만들수 수 있습니다.

tuple

C++ 표준의 tuple<> 은 서로 다른 타입의 객체를 임의의 개수 만큼 보관할 수 있습니다. couple<> 과 상속, 특수화 기술을 사용하면 tuple<> 과 유사한 클래스를 쉽게 만들 수 있습니다.

완성된 코드는 아래와 같습니다.

```
template<typename T, typename U> struct couple
{
    T v1;
    U v2;
    static constexpr int N = 2;
};

struct Null {}; // empty class

// primary template.
template<typename P1,
         typename P2,
         typename P3 = Null,
         typename P4 = Null,
         typename P5 = Null>
class xtuple : public couple<P1, xtuple<P2, P3, P4, P5, Null>>
{
};
// 템플릿 인자가 2개일때를 위한 partial specialization
template<typename P1, typename P2> class xtuple<P1, P2> : public couple<P1, P2>
{
};
int main()
{
```

```
    xtuple<int, char, short, long, double> t5;
}
```

먼저 xtuple의 primary template의 template 인자를 살펴보면 디폴트 인자가 적용되어 있으므로 2~5개까지의 템플릿 인자를 사용할 수 있습니다.

```
xtuple<int, char> t2;
xtuple<int, char, short, long, double> t5;
```

또한, xtuple<>은 내부적으로 별도의 멤버 data가 없으므로 xtuple<> 은 자신의 기본 클래스와 동일한 메모리 구조를 가지게 됩니다.

```
// xtuple은 자신의 멤버가 없으므로 기본 클래스와 동일한 메모리 구조를 가지게 됩니다.
class xtuple : public couple<P1, xtuple<P2, P3, P4, P5, Null>>
{
};
```

결국, xtuple<int, char, short, long, double>에 메모리 구조를 알기 위해서는 기본 클래스의 모양을 알아야 합니다.

```
// Base class : copule<int, xtuple<char, short, long, double, Null>>
xtuple<int, char, short, long, double> t5;
```

그런데, 기본 클래스에 다시 xtuple<>이 있으므로 역시 동일한 방법으로 기본 클래스의 모양을 알아야 합니다. 결국, 정리해 보면 다음과 같이 됩니다.

```
//                          couple<short, xtuple<long, double>>
//             couple<char, xtuple<short, long, double>>
// couple<int, xtuple<char, short, long, double>>
xtuple<int, char, short, long, double> t5;
```

최종적으로, xtuple<long, double>이 나오게 됩니다. 그런데, 이 경우는 인자가 2개인 부분 특수화 버전이 아래처럼 되어 있습니다.

```
template<typename P1, typename P2> class xtuple<P1, P2> : public couple<P1, P2>
```

결국, 총정리 하면

```
//                                        couple<long, double>
//                            couple<short, xtuple<long, double>>
//              couple<char, xtuple<short, long, double>>
// couple<int, xtuple<char, short, long, double>>
xtuple<int, char, short, long, double> t5;
```

입니다.

따라서, xtuple<int, char, short, long, double>은 다음과 동일한 메모리 모양을 가지게 됩니다.

```
couple<int, couple<char, couple<short, couple<long, double>>>> t5;
```

이번 절에서 만든 xtuple은 2~5개의 data를 보관할 수 있습니다. C++11에서 추가된 "가변인자 템플릿(Variadic template)문법"을 사용하면 인자 개수에 제한이 없는 tuple을 만들 수 있습니다.

이 책의 뒷장에서 가변 인자 템플릿을 다룰 때, 인자의 개수 제한이 없는 tuple을 만들어 보도록 하겠습니다.

▌ 연습문제

1) 앞에서 만든 xtuple<> 에 생성자를 추가해서 다음 처럼 사용할 수 있게 만들어 보세요

```
xtuple<int, double, char> t3(1, 3.4, 'A'); // 생성자로 초기값을 지정할 수 있게 해보세요
```

2) xtuple<>에서 값을 꺼낼 때 사용할 수 있는 get<>을 만들어 보세요.

```
int main()
{
    xtuple<int, double, char> t3(1, 3.4, 'A');

    double d = xget<1>(t3);

    cout << d << endl; // 3.4
}
```

▌ 핵심 개념 정리

- ☑ IfThenElse 는 "bool 기반의 타입 선택(Type Selection)"을 하는 도구 입니다. Select<>라는 이름으로 만들어서 사용되는 경우도 있습니다. C++11 에서는 conditional<> 이라는 이름으로 표준화 되었습니다.
- ☑ template parameter 로 자기 자신을 사용할 수 있습니다.
- ☑ couple template 의 부분 특수화 기술.
- ☑ xtuple 만들기

항목 5-03

template meta programming

배우게 되는 내용

template meta programming

constexpr

1. template meta programming

"템플릿 특수화(Specialization)" 기술을 사용하면 컴파일 시간에 연산을 수행하는 메타 함수를 만들 수 있습니다. 아래 코드를 생각해 봅시다.

```
template<int N> struct Factorial
{
    enum { value = N * Factorial<N-1>::value };
};
// 재귀를 종료하기 위해 Specialization 버전을 사용합니다.
template<> struct Factorial<1>
{
    enum { value = 1};
};
int main()
{
    int n = Factorial<5>::value;
}
```

위 코드에서 "Factorial<5>::value"를 생각해 보면 아래처럼 재귀적으로 코드가 반복되게 됩니다.

```
int main()
{
    int n = Factorial<5>::value;
            // 5 * Factorial<4>::value
            //     4 * Factorial<3>::value
            //         3 * Factorial<2>::value
            //             2 * Factorial<1>::value // Factorial<1>는 특수화 버전
            //                 1                   // 을 사용하게 됩니다.

            // 5 * 4 * 3 * 2 * 1  => 컴파일 시간에 연산 가능 하므로
            // 컴파일을 마치면 최종적으로 120 이 남게 됩니다.
}
```

결국 "Factorial<5>::value" 는 "5*4*3*2*1" 의 코드를 생성하게 됩니다. 이처럼, 컴파일 시간에 어떤 연산을 수행하게 하는 기술을 "Template Meta Programming" 이라고 합니다.

이와 같은 Template Meta Programming 은 대부분 재귀를 사용하게 되는데, 재귀의 종료를 위해서는 반드시 특정 값을 위한 특수화(Specialization) 버전을 제공해야 합니다.

여기서 주의 할 점 은, 템플릿 인자로는 상수만 사용할 수 있습니다.

따라서, 아래 처럼 사용할 수는 없습니다.

```
int a = 5;
int n = Factorial<a>::value; // error, a 는 상수가 아닌 변수 입니다.
```

2. C++11/14/17

C++11/14 - constexprt function

C++11/14 에서 등장한 constexpr 함수 를 사용하면 컴파일 시간에 연산이 가능한 메타 함수를 쉽게 만들 수 있게 되었습니다. Factorial을 C++11/14 스타일의 constexpr 함수로 만들면 동일한 효과를 볼 수 있습니다.

```
constexpr int Factorial(int n)
{
    return n == 1 ? 1 : n * Factorial(n - 1);
}
int main()
{
    int x[Factorial(5)]; // int x[120]
}
```

C++17 - constexprt lambda

C++17 부터는 람다 표현식도 컴파일 시간에 연산이 가능한 메타 함수로 만들 수 있습니다.

```
int main()
{
    constexpr auto f =  [](int a, int b) { return a + b; };

    int y[f(1,2)];
}
```

연습 문제

binary 메타 함수를 만들어보세요. (0 또는 1 외의 숫자 입력시 컴파일 에러가 나오게 만들어 보세요)

```
int main()
{
    int n = binary<101>::value;
```

```
    cout << n << endl; // 5
}
```

▌ 핵심 개념 정리

- 컴파일 시간의 재귀를 사용한 "Template Meta Programming" 기술
- constexpr function (C++11/14), constexpr lambda (C++17)

Section 6

Type traits

항목 6-1

type traits 개념

배우게 되는 내용

Type traits 개념

Is_pointer

1. type_traits 개념

▌ printv 함수

다음의 코드에서 printv() 템플릿은 인자로 전달된 변수의 값을 출력하는 함수 입니다.

템플릿으로 만들어져 있기 때문에 다양한 타입의 변수 값을 출력할 수 있습니다.

```
template<typename T> void printv(const T& v)
{
    cout << v << endl;
}
int main()
{
    int    n = 1;
    double d = 3.4;
    printv(n);
    printv(d);
    printv(&d); // 포인터 타입도 전달 가능합니다.
}
```

printv() 는 템플릿 이므로 다음 처럼 변수의 주소도 전달 가능합니다.

```
printv(&d); // 포인터 타입도 전달 가능합니다.
```

그런데, 일반적으로 변수의 주소를 출력할 때는 변수의 값도 알고 싶을 때가 많습니다. 그렇다면 printv() 를 개선해서 변수 주소가 전달 될 경우 주소 뿐 아니라 값도 출력 가능 하도록 변경해 봅시다. 아마, 코드는 다음 처럼 작성될 것입니다.

```
template<typename T> void printv(const T& v)
{
    if ( T 가 포인터 타입이면 )
        cout << v << " : " << *v << endl;
    else
        cout << v << endl;
}
```

이제, 위 코드를 완성하려면 T가 포인터 타입 인지 아닌지를 조사 할 수 있어야 합니다.

이처럼 함수 또는 클래스 템플릿을 만들 때, 템플릿 파라미터 T의 다양한 특성(Traits)을 조사하는 기술을 "Type Traits" 라 고 합니다.

C++11 표준에서 제공되는 is_pointer<>를 사용하면 다음 처럼 T가 포인터 인지를 조사할 수 있습니다.

```
if ( is_pointer<T>::value ) {}
```

이번 장에서는 is_pointer<>와 같은 traits의 사용법 뿐 아니라, 만드는 법, 그 밖에 traits와 관련된 다양한 주제에 대해서 살펴 보도록 하겠습니다.

2. is_pointer

▌ is_pointer<> 만들기

is_pointer<>는 템플릿 부분 특수화(partial specialization) 문법을 사용하면 아주 쉽게 만들 수 있습니다. C++ 표준과의 이름 충돌을 피하기 위해 xis_pointer<>로 만들겠습니다.

```
// C++ 표준과의 충돌을 막기 위해 접두에 'x'를 붙여 만들었습니다.
template<typename T> struct xis_pointer
{
    enum { value = false };
};
// 템플릿 부분 전문화 문법에 따라 T가 포인터인 경우 아래 템플릿을 사용하게 됩니다.
template<typename T> struct xis_pointer<T*>
{
    enum { value = true };
};
template<typename T> void foo(const T& v)
{
    if (xis_pointer<T>::value)
        cout << "포인터" << endl;
    else
        cout << "포인터 아님" << endl;
}
int main()
{
    int n = 0;
    foo(n);
    foo(&n);
}
```

위 코드의 원리는 간단합니다. xis_pointer<T>::value 에서

- **T가 포인터 라면 primary template 을 사용하므로 value는 false가 되고**
- **T가 포인터가 아니라면 부분 특수화 버전이 사용되므로 value 는 true 가 됩니다.**

일반적으로 간단한 traits를 만들 때는 아래 처럼 만들면 됩니다.

- **primary template을 만들고 value = false 로 하고**

- **조건에 따른 부분 특수화(Partial Specialization) 버전을 만들고 value = true 로 합니다.**

enum 상수 vs constexpr

C++11/14 이전의 C++98/03 문법에서는 구조체(또는 클래스)를 만들 때 일반 멤버 변수를 바로 초기화 할 수 없었습니다.

```
template<typename T> struct xis_pointer
{
    bool value = false; // C++98/03 문법에서는 에러, C++11/14 부터는 가능
    enum { value = false }; // C++98/03 문법에서도 허용.
};
```

그래서, 예전부터 traits를 만들 때 enum 상수를 즐겨 사용했습니다.

하지만, C++11/14 부터는 constexpr 상수를 사용하는 것이 좋습니다. 그래서, xis_pointer<>를 최신 방식으로 만들면 다음과 같습니다. 또한, const와 volatile을 고려하면 아래 처럼 만들어야 합니다.

```
template<typename T> struct xis_pointer
{
    static constexpr bool value = false;
};
template<typename T> struct xis_pointer<T*>
{
    static constexpr bool value = true;
};
// const, volatile
template<typename T> struct xis_pointer<T* const>
{
    static constexpr bool value = true;
};
template<typename T> struct xis_pointer<T* volatile>
{
    static constexpr bool value = true;
};
template<typename T> struct xis_pointer<T* const volatile>
{
    static constexpr bool value = true;
```

```
};
```

▌ 메타 함수 (meta function)

xis_pointer<>를 사용하는 코드를 보면 마치 함수를 호출하는 것처럼 생각할 수 있습니다. 괄호를 호출하는 진짜 함수가 아니라, 의미상으로 유사하다는 말입니다.

```
bool b = xis_pointer<T>::value;
```

위 코드에서 함수 이름을 xis_pointer, T를 함수 인자, value를 리턴 값으로 생각해 볼 수 있습니다.

그런데, 템플릿은 실행 시간이 아닌 컴파일시간에 코드를 생성하고, 컴파일 시간 상수인 value를 꺼내게 되므로 컴파일 시간에 true/false 여부가 결정됩니다. 그래서 이런 것을 흔히, 컴파일 시간에 사용하는 함수 라는 의미로 “메타 함수” 라고 부르기도 합니다.

3. is_array

is_array<> 만들기

이번에는 템플릿의 인자 T가 배열 인지를 조사하는 xis_array를 만들어 보도록 하겠습니다. 기본 코드는 다음 처럼 하면 됩니다.

```
template<typename T> struct xis_array
{
    static constexpr bool value = false;
};
template<typename T> struct  xis_array< ? >
{
    static constexpr bool value = true;
};
```

이제 위 코드에서 ? 부분을 채우면 됩니다. ?를 채우기 위해서는 배열의 타입에 대해서 정확히 알아야 합니다. 일반적으로 변수 선언문에서 변수의 이름을 제외하면 타입만 남게 됩니다.

```
int a;  // 변수 이름은 a입니다. a의 타입은 int 입니다.
int* p; // 변수 이름은 p 입니다. p 의 타입은 int* 입니다.
```

동일한 개념을 배열에 적용하면 배열의 타입은 다음과 같습니다.

```
int x[3];       // 변수 이름은 x, 타입은 int[3] 입니다.
int y[3][2];    // 변수 이름은 y, 타입은 int[3][2]
int z[2][3][2]; // 변수 이름은 z, 타입은 int[2][3][2]
```

또한, 2차원 배열의 경우도 1차 배열의 배열입니다. 결국 임의의 타입 T의 임의의 크기 N의 배열 타입의 공통적인 모양은 다음과 같습니다.

```
T[N];
```

결국, xis_array<> 는 다음과 같습니다.

```
template<typename T> struct xis_array
{
    static constexpr bool value = false;
};
template<typename T, int N> struct xis_array<T[N]>
{
    static constexpr bool value = true;
};

template<typename T> void foo(const T& v)
{
    if (xis_array<T>::value)
        cout << "배열 입니다." << endl;
    else
        cout << "배열이 아닙니다." << endl;
}

int main()
{
    int x[3] = { 0 };
    int y[3][2] = { 0 };
    foo(x);
    foo(y);
}
```

여기서, 핵심은 primary template 은 인자를 T 하나만 받지만, 부분 특수화 버전은 다음 처럼 T, N으로 만들게 된다는 점입니다.

```
template<typename T> struct xis_array       // primary template
template<typename T, int N> struct xis_array<T[N]>
```

또한, 배열 중에 "크기를 알수 없는 배열(unknown size array)"도 있으므로 아래 처럼 부분 전문화 버전을 하나 더 제공하는 것이 일반 적입니다.

```
template<typename T> struct xis_array<T[]>
{
    static constexpr bool value = true;
};
```

마지막으로 위 코드를 테스트 할 때는 반드시 foo() 가 인자를 참조(T&)로 받아야 합니다. 값(T)로 전달 받을 경우 "Array to Pointer Conversion" 현상(Decay) 때문에 배열이 아닌 포인터로 전달되게 됩니다.

```
template<typename T> void foo(T v)  // bug, 배열을 값으로 전달 받으면 T는 포인터
{                                   // 타입이 됩니다.
    if (xis_array<T>::value)
        cout << "배열 입니다." << endl;
    else
        cout << "배열이 아닙니다." << endl;
}
```

[참고] argument decay 항목을 참고하시면 됩니다.

▌ 배열의 크기 구하기

이번에는 배열의 크기를 구하는 문제를 생각해 봅시다.

다음 코드를 생각해 보세요

```
template<typename T, int N> struct xis_array<T[N]>
{
    static constexpr bool value = true;
};
template<typename T> void foo(const T& v)
{
    bool b = xis_array<T>::value;
}
int main()
{
    int x[3] = { 0 };
    foo(x);
}
```

위 코드는 main 함수에서 int [3] 타입인 x 를 foo()에 전달 했으므로 xis_array<> 안에서 T는 int 로 결정되고, N은 3으로 결정됩니다. 즉, xis_array<> 안에서는 배열의 크기가 3이라는 사실을 알 수 있다는 점입니다.

따라서, xis_array<> 에서 다음과 같은 코드를 추가하면 배열의 크기도 구할 수 있습니다.

```
template<typename T> struct xis_array
{
    static constexpr size_t size = -1;  // 배열이 아닌 경우는 -1로 처리합니다.
                                        // 또는, size 자체를 제공하지 않아도 됩니다.
    static constexpr bool value = false;
};
template<typename T, int N> struct xis_array<T[N]>
{
    static constexpr size_t size = N;   // 핵심입니다. 배열의 크기
    static constexpr bool value = true;
};
template<typename T> void foo(const T& v)
{
    cout << xis_array<T>::size << endl; // 3
}
int main()
{
    int x[3] = { 0 };
    foo(x);
}
```

위의 코드의 단점은 2차 이상의 배열의 경우 각 차원의 크기를 개별적으로 구할 수가 없습니다. C++ 표준의 extent 를 사용하면 2차 이상의 배열도 각 차원 별로 크기를 모두 구할 수 있습니다.

▌ C++ 표준 is_array<>, extent<>

C++11 표준에서는 배열의 크기를 구하는 extent<>를 제공하고 있습니다.

C++11 표준 traits 를 사용하려면 <type_traits> 헤더를 포함해야 합니다.

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

template<typename T> void foo(const T& v)
{
    if (is_array<T>::value)
    {
```

```
        cout << extent<T, 0>::value << endl; // 3
        cout << extent<T, 1>::value << endl; // 2
        cout << extent<T, 2>::value << endl; // 1
    }
}
int main()
{
    int x[3][2][1] = { 0 };
    foo(x);
}
```

▌ 핵심 개념 정리

- 템플릿 파라미터 T 의 다양한 특성을 조사하는 기술을 type traits 라고 합니다.
- 만드는 방법은 대부분 primary template 에서는 false 를, 부분 특수화 버전에서는 true 를 리턴(value = true)해 줍니다.
- C++11 표준에서 배열의 크기를 구하는 traits 을 extent<> 입니다.

항목 6-2

type traits – integral_constant

배우게 되는 내용

int2type

integral_constant

true_type, false_type

1. int2type

이번에는 앞에서 만든 xis_pointer<>를 사용해서 printv()를 완성해 보도록 하겠습니다.

printv() 를 완성하면서 발생하는 여러가지 문제점을 살펴 보고, int2type의 개념과 C++11/14에서 추가된 integral_constant 개념에 대해서 살펴 보도록 하겠습니다. 내용이 좀 어렵기 때문에 주의 깊게 보셔야 합니다.

먼저, 본격적인 주제에 들어가기 앞서, 간단한 도구인 int2type에 대해서 먼저 살펴 보도록 하겠습니다.

함수 오버로딩

C++에서는 함수 이름은 동일 해도 인자의 타입이나 인자의 개수가 다르면 동일 이름을 여러 개 만들 수 있습니다.

```
void foo(int a)        { cout << "int" << endl; }
void foo(double d)     { cout << "double" << endl; }
void foo(int a, int b) { cout << "int, int" << endl; }
```

함수를 사용할 때, 컴파일러는 인자의 개수나 인자의 타입에 보고 적합한 함수를 선택하게 됩니다.

```
foo(1);     // foo(int) 호출
foo(3.4);   // foo(double) 호출
foo(1, 2);  // foo(int, int) 호출
```

그렇다면, 인자의 개수나 타입이 아닌, 인자의 값에 따라 다른 함수가 호출되게 할수 있을까요 ?

즉, 인자가 0일 때, 1일 때 각각 다름 함수가 호출되게 할 수 있을까요 ?

```
// 0, 1은 모두 같은 타입(int) 이므로 아래 2줄은 같은 함수를 호출합니다.
// 0, 1을 가지고 다른 함수를 호출 하도록 할 수 없을까요 ?
foo(0);
foo(1);
```

int2type 이라는 훌륭한 도구를 사용하면 타입이 아닌 값을 가지고 다른 함수를 호출하게 만들

수 있습니다.

int2type

다음 코드를 잘 생각해 보세요.

```
template<int N> struct int2type
{
    enum { value = N};
};
void foo(int a)         { cout << "int"         << endl; }
void foo(int2type<0> a) { cout << "int2type<0>" << endl; }
void foo(int2type<1> a) { cout << "int2type<1>" << endl; }

int main()
{
    // 0과 1은 같은 타입입니다. 아래 2줄은 모두 동일 함수를 호출합니다.
    foo(0);
    foo(1);

    // 템플릿은 인자가 결정되면 클래스(구조체)를 생성합니다.
    // 따라서, 아래 코드에서 t0, t1은 다른 타입입니다.
    int2type<0> t0;
    int2type<1> t1;

    // t0, t1이 다른 타입이므로, 아래 2줄은 다른 함수를 호출합니다.
    foo(t0);
    foo(t1);
}
```

결국,

"int2type<>은 정수형 상수를 독립된 타입으로 만드는 도구"

입니다.

77, 78 은 모두 같은 타입(int) 이지만, int2type<77>, int2type<78> 은 각각 독립된 별개의 타입이 됩니다. 그래서 int를 type으로 만든 다는 의미로 이름이 "int2type" 입니다.

int2type을 사용시 주의할 점은 컴파일 시간에 결정된 정수형 상수만 사용 가능하다는 점입니다.

```
int n = 0;
int2type<n>   t0; // error. 변수를 사용할 수 없습니다.
int2type<3.4> t1; // error. 실수를 사용할 수 없습니다.
```

int2type 덕분에 컴파일 시간에 결정된 모든 상수는 타입화 될 수 있고, 각각을 인자로 해서 다른 함수를 호출할 수 있게 됩니다.

언제 사용할까? 라는 의문이 드시는 분은 이제 다음에 나오는 주제를 끝까지 읽어 보시면 마지막에 int2type이 등장하게 됩니다.

이제, 본격적으로 이장에서 해야할 주제로 들어가 보겠습니다.

2. printv()와 is_pointer<>의 결합

문제점

이제, 우리가 만든 xis_pointer<>를 사용해서 printv() 함수를 완성해 보겠습니다. 아래 코드를 작성하고 컴파일 해보세요

```
template<typename T> struct xis_pointer     { static constexpr bool value = false;};
template<typename T> struct xis_pointer<T*> { static constexpr bool value = true;};

template<typename T> void printv(const T& v)
{
    if ( xis_pointer<T>::value )
        cout << v << " : " << *v << endl;
    else
        cout << v << endl;
}

int main()
{
    int    n = 1;
    printv(n);  // A. error
    printv(&n); // B
}
```

그런데, 컴파일 하면 A 코드 때문에 컴파일 에러가 발생합니다. A를 주석 처리하면 에러가 발생하지 않습니다. 왜 에러 일까요 ?

에러의 원인을 파악하기 위해 컴파일러가 생성하는 코드를 생각해 봅시다. B는 문제가 없으니 A의 경우만 생각해 봅시다.

```
// printv(n) 으로 사용했을 때 의 컴파일러가 생성한 코드
void printv(const int& v)
{
    if ( false )  // n 은 포인터가 아니므로 false 입니다.
        cout << v << " : " << *v << endl;
    else
        cout << v << endl;
}
```

결국 위 코드는

- **n은 정수 이므로 T는 int로 결정됩니다.**
- **T가 포인터가 아니므로 xis_pointer<T>::value 는 컴파일 시간엔 false로 결정됩니다.**
- **컴파일 시간에 false로 결정 되었으므로 if 문은 실행되지는 않습니다.**

그런데, if 절이 false 가 되어서, 실행되지 않더라도 컴파일은 해야 하지 않을까요 ? 그런데 아래 코드에서 v는 포인터가 아닌 값(정수) 이므로 '*' 연산자를 사용해서 *v 연산을 수행할 수가 없습니다.

```
cout << v << " : " << *v << endl;  // *v 표현 때문에 컴파일 시간 에러입니다.
```

핵심은, printv()가 포인터가 아닌 값을 전달 받으면 if 조건문은 컴파일 시간에 false 로 결정됩니다. if 절이 컴파일 시간에 false로 결정되어도 실행 시간에 실행은 안되지만 컴파일 과정이 생략되는 것은 아니라는 점입니다.

간단히, 정리하면

```
void printv(int v)
{
    if ( false )
    {
        // 이부분은 실행시간에 절대 실행되지 않습니다.
        // 하지만 컴파일에서 제외되는 것은 아닙니다.
        *v = 1;
    }
}
```

해결책은, 컴파일 시간에 false로 결정 되었다면 실행에서 제외되는 것 뿐 아니라 컴파일 자체를 제외 할 수 있는 기술이 필요합니다.

[참고] C++17의 "if constexpr" 문법을 사용하면 if의 조건 결과에 따라 컴파일에서 제외 시킬수 있습니다.

```
template<typename T> void printv(T v)
{
    if constexpr (false)    // C++17 문법 입니다.
```

```
        *v = 10;        // 에러가 발생하지 않습니다.
}
int main()
{
    printv(0);
}
```

하지만, C++11/14 에서는 "if constexpr" 이 지원되지 않으므로 다른 방식의 해결책을 사용합니다.

▌ Lazy Instantiation 과 int2type

이 문제를 해결하기 위해서는 "함수 템플릿은 사용되지 않으면 코드를 생성하지 않는다"는 Lazy Instantiation 개념과 컴파일 시간 정수를 타입화 하는 int2type 의 개념을 사용해야 합니다.

앞에서 배운 지연된 인스턴스화의 개념을 다시 살펴 봅시다. 아래 코드에서 코드를 생각해 봅시다.

```
template<typename T> class B
{
    T data;
public:
    void foo(T a) {*a = 0;} // T가 int로 결정되어도 b.foo() 함수를 사용한적이 없으면
                            // 실제 C++ 함수를 생성하지 않으므로 에러가 나지 않습니다.
};
int main()
{
    B<int> b;
    // b.foo() 함수를 사용한적이 없습니다.
}
```

결국 사용되지 않은 함수 템플릿은 C++ 코드도 생성하지 않기 때문에 컴파일 되지 않습니다.

▌ if 문에 의한 함수 분배 (dispatch)

이제, 지연된 인스턴스화의 개념을 잘 기억해 놓고 다음 코드를 생각해 봅시다.

이 코드의 의도는 pointer( )와 not_pointer()가 진짜 함수가 아니고 템플릿 이므로 항상 컴파일

되는 것이 아니라 "사용되지 않으면 실제 함수가 만들어 지지 않는다" 는 생각으로 만든 코드입니다.

```
// T가 포인터가 아닐 때 아래 함수 템플릿이 사용되면 컴파일 error 입니다.
template<typename T> void pointer(const T& v)    {  *v = 10;}
template<typename T> void not_pointer(const T& v){  }

void printv(const int& v)
{
    if ( false )   // 컴파일 시간에 false이므로
        pointer(v);// "이 함수 템플릿은 사용되지 않을 것이다"라는 의도 입니다.
    else
        not_pointer(v);
}
```

즉, printv() 안에 있는 if 문은 "컴파일 시간에 false로 결정되었으므로 pointer()는 사용되지 않을 것이다. 따라서 pointer()함수는 int 타입의 함수로 인스턴스화 되지 않기 때문에 문제가 없다." 라는 의도 입니다.

될까요 ?

안됩니다.!

C++ 에서 if 문은 실행시간 조건문 입니다. 실행시간에 조건을 조사하기 위해 만든 문법입니다. 따라서, 참/거짓 여부가 컴파일 시간에 결정되어도 컴파일 시간에 컴파일러는 if 절과 else 절 모두가 사용된다고 생각하고 코드를 생성합니다.

```
void printv(const int& v)
{
    if ( false )    // 비록, 컴파일 시간에 false이지만 컴파일러는 아래 2개의 함수가
        pointer(v); // 모두 사용된다고 판단해서 모두 인스턴스화를 합니다.
    else
        not_pointer(v);
}
// pointer(), not_pointer() 함수 템플릿이 모두 사용된다고 판단해서 2개 모두
// C++ 함수를 생성(인스턴스화) 합니다.
void pointer(const int& v)    { *v = 10;} // error.
void not_pointer(const int& v){  }
```

결국, 우리가 필요한 것은 조건이 false일 때는 특정 코드(*v)가 컴파일 되지 않도록 하고 싶습니다. 그런데, if 조건문을 사용한 함수 호출 분배(dispatch)는 컴파일 시간이 아닌 실행 시간이므로

"사용되지 않으면 코드를 생성하지 않는다"라는 개념을 적용할 수 없습니다.

어떻게 해야 할까요 ?

▌ 함수 오버로딩에 의한 함수 분배 (dispatch)

동일 이름의 함수가 여러 개 있을 때 어떤 함수를 호출할 것인가를 결정하는 것은 컴파일 시간에 이루어 집니다.

아래 코드를 생각해 봅시다.

```
// 이템플릿은 T를 int 타입에 대해 코드를 생성하면 컴파일 에러가 발생합니다.
// 하지만, main 함수에서 이 함수 템플릿을 사용한 적이 없으므로 실제 함수가 생성되지
// 않습니다.
template<typename T> void printv_imp(const T& v, int n)
{
    cout << v << " : " << *v << endl;
}
template<typename T> void printv_imp(const T& v, double n)
{
    cout << v << endl;
}
int main()
{
    int n = 10;

    printv_imp(n, 3.4); // 어떤 printv_imp()를 호출할지를 컴파일 시간에 결정합니다.
                        // 사용되지 않은 함수 템플릿은 인스턴스화 되지 않습니다.
}
```

위 코드에는 2개의 printv_imp() 함수 템플릿이 있습니다. 이중 printv(T, int) 버전인 경우 내부적으로 *v 표현을 사용하므로 T가 값 타입 일 때는 절대 사용되면 안됩니다.

main() 함수에서 2번째 인자로 3.4를 전달했으므로 printv(T, double) 버전의 printv_imp()가 사용됩니다. 이때, 어느 함수를 사용할 것인가는 컴파일 시간에 결정됩니다. 즉, 사용되지 않은 printv_imp(T, int) 은 실제 C++ 함수로 생성되지 않으므로 컴파일 되지 않습니다.

즉, 함수 인자를 보고 동일 이름의 함수중 어느 함수를 선택할 것인가를 결정하는 것은 실행시간 함수 분배(dynamic call dispatch)가 아닌 컴파일 시간 함수 분배(static call dispatch)입니다.

결국, 포인터 일 때 와 아닐 때를 위한 함수를 따로 만들고, 함수의 선택을 함수 오버로딩에 의해

서 한다면 문제가 해결됩니다.

이제 printv()를 아래 처럼 생각해 봅시다.

```
template<typename T> void printv(const T& v)
{
    // 포인터 여부에 따라 다른 함수가 선택되게 합니다.
    // 함수 오버로딩에 의한 함수 선택은 컴파일 시간에 이루어집니다.
    printv_imp(v, xis_pointer<T>::value);   // T가 포인터라면 true(1),
                                            // 포인터가 아니면 false(0) 입니다.
}
```

그런데, 문제는 xis_pointer<T>::value 는 T가 포인터 이거나 아니거나 결국 true(1) 또는 false(0) 입니다. 0 또는 1 은 모두 같은 타입(int)으로는 함수 오버로딩으로 사용할 수 없다는 점 입니다.

이때, 앞에서 배운 int2type을 적용하면 문제를 해결할 수 있습니다. 결국 완성된 코드는 다음과 같습니다.

```
template<int N> struct int2type
{
    enum { value = N };
};

template<typename T> void printv_imp(const T& v, int2type<1>)
{
    cout << v << " : " << *v << endl;
}
template<typename T> void printv_imp(const T& v, int2type<0>)
{
    cout << v << endl;
}
template<typename T> void printv(const T& v)
{
    // 0, 1을 서로 다른 타입으로 만들어서 함수 오버로딩에 사용합니다.
    printv_imp(v, int2type<xis_pointer<T>::value>() );
}
int main()
{
    int n = 10;
    printv(n);
}
```

3. C++11 표준 - integral_constant<>

integral_constant

int2type은 int 형 상수를 타입으로 만들 수 있습니다. 그런데, int 뿐 아니라 short2type, char2type도 있으면 좋지 않을 까요 ? (실수는 템플릿 인자가 될 수 없으므로 double2type 은 만들 수 없습니다.)

C++ 표준위원회는 int2type을 좀더 발전시킨 integral_constant 라는 템플릿을 C++11 표준에 추가했습니다. int 뿐 아니라 char, short, long 등 정수 계열의 모든 상수를 타입으로 만들 수 있는 템플릿 입니다.

```
template<typename T, T N> struct integral_constant
{
    static constexpr T value = N;
    typedef T type;
};
// 아래 3개의 변수는 모두 다른 타입입니다.
integral_constant<int, 0>   n0;
integral_constant<int, 1>   n1;
integral_constant<short, 1> s1;
```

true_type, false_type

true, false는 참, 거짓을 나타내는 값 입니다. 둘 다 bool 이라는 같은 타입 입니다. 하지만, integral_constant<>를 사용하면 참을 나타내는 타입, 거짓을 나타내는 타입을 만들 수 있습니다.

```
// 0, 1 은 같은 타입(int)이지만
// integral_constant<int, 0> 과 integral_constant<int, 1>은 다른 타입입니다.
integral_constant<int, 0>   n0;
integral_constant<int, 1>   n1;

// true와 false는 같은 타입(bool) 이지만, true_type과 false_type은 다른 타입입니다.
typedef integral_constant<bool, true>  true_type;
typedef integral_constant<bool, 1>     false_type;
```

또한, true_type, false_type은 결국 integral_constant<> 이므로 내부적으로 value 멤버를 가지고 있습니다.

```
bool b1 = true_type::value;  // true 입니다.
bool b2 = false_type::value; // false 입니다.
```

C++11의 is_pointer<>의 구현 방식

C++11 의 공식 표준에서는 is_pointer<>등의 traits 를 만들 때 상속의 개념을 사용하고 있습니다.

```
template<typename T>
struct is_pointer : public false_type {};// false_type 안에 value = false 가 있습니다.

template<typename T>
struct is_pointer<T*>:public true_type {};// true_type  안에 value=true  가 있습니다.
```

핵심은 T 가 포인터 인지 아닌 지에 따라 기본 클래스가 달라진다는 점입니다.

is_pointer<T>에서 T 가 포인터가 아닌 경우 기본 클래스는 false_type 이고, T 가 포인터 인 경우 기본 클래스는 true_type 입니다.

상속의 개념 덕분에 traits 를 사용할 때 보다 가독성이 높게 사용할 수 있게 되었습니다. 앞 절에서 만든 printv() 함수는 아래 처럼 다시 만들 수 있습니다. 주석으로 이전에 만든 코드를 표시 했으므로 2 개를 비교해 보시면 됩니다.

```
template<typename T>
void printv_imp(const T& v, true_type)  // int2type<1> 대신 true_type 사용 합니다.
{
    cout << v << " : " << *v << endl;
}
template<typename T>
void printv_imp(const T& v, false_type) // int2type<0> 대신 false_type 사용 합니다.
{
    cout << v << endl;
}
template<typename T>
void printv(const T& v)
```

```
{
    printv_imp(v, is_pointer<T>()); // int2type<is_pointer<T>::value>() 대신
                                    // is_pointer<>객체를 생성합니다.
                                    // T가 포인터 이면 printv_imp( v, true_type)이
                                    // 호출됩니다.
}
```

핵심은 int2type 을 사용하는 대신 is_pointer<T>() 를 사용해서 is_pointer<T>() 를 사용해서 임시객체를 만들고 있습니다. 그런데, T가 포인터 라면 is_pointer의 기반 클래스는 true_type가 되고, 포인터가 아니면 false_type 이 기반 클래스 입니다. C++은 어떤 객체를 인자로 전달할 때 기반 클래스 타입으로 전달 받을수 있으므로, T 가 포인터 라면 printv_imp(v, true_type)이 호출되게 됩니다.

▌C++ 표준의 is_pointer<>를 사용하는 방법

앞에서 살펴본 것처럼 템플릿 안에서 T 가 포인터 인지를 조사하려면 is_pointer<> 를 사용하면 됩니다.

이때, is_poitner<>를 사용해서 포인터 여부에 따라서 다른 코드를 작성하는 방법은 2 가지 방법이 있습니다.

(1) is_pointer<T>::value 로 조사하는 방법

(2) true_type, false_type 을 사용해서 함수 오버로딩을 이용하는 방법

- **is_pointer<T>::value 로 조사하는 방법**

```
template<typename T> void foo(T a)
{
    if (is_pointer<T>::value)
    {
        // T가 포인터 일때의 코드를 작성합니다.
        // 단, '*a' 와 같은 코드는 작성 할 수 없습니다.
    }
    else
    {
    }
}
```

::value 를 조사하는 방식에서 주의 해야 할 점은 T 가 포인터로 결정이 되어도 *a 라는 표현은 사용할 수 없다는 점입니다.

- true_type, false_type 을 사용해서 함수 오버로딩을 이용하는 방법

```
template<typename T> void foo_imp(T a, true_type)  {} //'*a' 표현식을 사용 가능합니다
template<typename T> void foo_imp(T a, false_type) {}

template<typename T> void foo(T a)
{
    foo_imp(a, is_pointer<T>());
}
```

▌ 핵심 개념 정리

- if 문에 의한 함수 호출 분배는 실행시간 분배 이지만 함수 오버로딩에 의한 함수 호출 분배는 컴파일 시간에 이루어 집니다.
- Int2type 을 사용하면 컴파일 시간에 결정된 정수형 상수를 타입화 할 수 있습니다. C++11 표준에는 integral_constant<>가 있습니다.
- is_pointer<T> 의 기반 클래스는 는 T 가 포인터 라면 true_type, T 가 포인터가 아니면 false_type 이 됩니다.
- C++17 의 if constexpr 을 사용하면 컴파일 시간 조건문을 만들수 있습니다.

항목 6-3

type traits - type modification

배우게 되는 내용

remove_pointer

recursive

1. type modification

C++ 표준의 type traits에는 타입의 특성을 조사할 수도 있지만 변형된(또는 연관된)타입을 얻을 수도 있습니다.

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

template<typename T> void foo(const T& a)
{
    // T가 포인터 인지 조사
    bool b = is_pointer<T>::value;

    // T에서 포인터를 제거한 타입 얻기
    typename remove_pointer<T>::type n; // n은 int 타입.
}
int main()
{
    int n = 0;
    foo(&n);
}
```

remove_pointer<T>::type 은 T에서 포인터를 제거한 타입(위 코드에서는int)을 얻을 수 있습니다.

이번 장에서는 remove_pointer<> 등을 만드는 방법에 대해서 살펴보도록 하겠습니다.

▌ remove_pointer<> 만들기

remove_pointer<>는 is_pointer<>와 마찬가지로 "템플릿 부분 특수화 (template partial specialization)" 을 사용하면 쉽게 만들 수 있습니다.

remove_pointer<> 와 같은 traits를 만들려면 다음의 단계로 만들면 됩니다.

① primary template을 제공하고 typedef를 제공합니다.

② 템플릿 부분 특수화을 사용해서 원하는 타입을 얻을 수 타입을 분할합니다.

③ 필요한 경우 재귀(Recursive)의 개념을 추가합니다.

이제, 자세히 살펴 보도록 하겠습니다. C++표준에 remove_pointer가 있으므로 여기서는 xremove_pointer<>라는 이름으로 만들겠습니다.

① primary template을 제공하고 typedef 를 제공.

```
template<typename T> struct xremove_pointer
{
    typedef T type;
};
int main()
{
    xremove_pointer<int*>::type n; // T가 int* 이므로 ::type 역시 int* 입니다.
}
```

위처럼 primary template을 만들고 사용하면

- "remove_pointer<int*>::type" 라고 표현할 때 T가 int* 가 되므로 결국 "::type" 도 int*가 됩니다.
- 우리가 원하는 것은 int* 가 아닌 int 이므로 int* 에서 int와 * 를 분리해야 합니다.
- 부분 특수화를 통해서 int*를 우리가 원하는 타입인 int와 *로 분리해야 합니다.

② 템플릿 부분 특수화를 사용해서 원하는 타입을 얻을 수 있도록 타입 분할.

int* 에서 우리는 원하는 것은 int 입니다. Int 를 얻기 위해서 부분 특수화를 통해 int 와 *를 분리해야 합니다.

```
template<typename T> struct  xremove_pointer
{
    typedef T type;
};
// xremove_pointer<int*>로 사용하면 부분 특수화 버전이 사용되고
// int* => T* 로 매칭 되므로 T는 int가 됩니다.
template<typename T> struct xremove_pointer<T*>  // T(int)와 *로 분할
{
    typedef T type;
```

```
};
int main()
{
    xremove_pointer<int*>::type n; // 이제는 이순간 부분 특수화 버전이 사용됩니다.
}
```

핵심은, 템플릿 부분 특수화를 통해서 우리가 원하는 타입을 얻을 수 있도록 분할한다는 점입니다.

③ 필요한 경우 재귀(Recursive)의 개념을 추가합니다.

아래 코드의 결과는 어떻게 될까요 ?

```
xremove_pointer<int***>::type n;
```

우리가 만든 xremove_pointer<>는 포인터를 한개 를 제거하게 되므로 ::type은 int** 가 됩니다. C++ 표준의 remove_pointer<> 동작 방식 역시 동일합니다.

그런데, 만약 모든 포인터를 제거하고 싶다면 어떻게 해야 할까요 ?

이경우, 부분 특수화 버전에 재귀의 개념을 넣으면 쉽게 해결 할 수 있습니다.

```
template<typename T> struct  xremove_pointer
{
    typedef T type;
};
template<typename T> struct xremove_pointer<T*>
{
    typedef typename xremove_pointer<T>::type type;
};
xremove_pointer<int***>::type n;
```

위 코드는 ::type 의 타입은 아래 처럼 계산됩니다.

1. int** 와 *로 분리되므로 xremove_pointer<int**>::type 를 구해야 합니다.
2. int* 와 *로 분리되므로 xremove_pointer<int*>::type을 구해야 합니다.
3. int 와 *로 분리되므로 xremove_pointer<int>::type을 구해야 하는데, int가 포인터가 아니므로 primary template을 사용하게 됩니다. 결국 ::type은 int가 됩니다.

이와 같은 재귀의 개념을 사용하는 기술은 아주 널리 사용되므로 반드시 이해 해야 합니다.

[참고] C++표준의 remove_pointer<>의 경우는 재귀개념을 사용하지 않습니다. 따라서, remove_pointer<int>::type의 경우 int* 가 됩니다.**

2. 함수의 리턴 타입과 인자 타입 알아내기

이번에는 템플릿 인자 T가 함수일 때, 함수의 리턴 타입과 인자 타입을 구하는 traits를 만들어 보도록 하겠습니다. 함수는 인자가 2개라고 가정하고 해보겠습니다. 먼저, 사용하는 방법은 다음과 같습니다.

```
char f(int n, double d)
{
    return 'A';
}
template<typename T> void foo(T& t)
{
    typename result<T>::type      ret;  // char
    typename argument<0, T>::type arg0; // int
    typename argument<1, T>::type arg1; // double

    cout << typeid(ret).name() << endl;
    cout << typeid(arg0).name() << endl;
    cout << typeid(arg1).name() << endl;
}
int main()
{
    foo(f);
}
```

result<>와 argument<>를 만들어 보도록 하겠습니다. 먼저 result<> 입니다.

result<> 만들기

먼저 primary template 입니다.

```
template<typename T> struct result
{
    typedef T type;
};
result<char(int, double)>::type ret;
```

primary template<> 만 있다면. "result<T>::type" 은 함수 타입(char(int,double)) 이 됩니다.

그런데, 우리가 원하는 것은 리턴 타입인 char 를 구해야 합니다.

결국 핵심은,

"부분 특수화를 사용해서 T를 "리턴 타입(인자타입1, 인자타입2)"의 형태로 분할"

해야 합니다.

함수 모양으로 부분 특수화

인자가 2개인 함수라고 가정 했으므로 T를 인자 2개 모양의 함수으로 부분 특수화 하려면 템플릿의 인자는 3개로(리턴타입, 두개의 인자 타입) 해야 합니다.

```
template<typename R, typename A1, typename A2> struct result<R(A1, A2)>
{
    typedef R type;
};
```

역시 핵심은 리턴 타입을 구하기 위해 부분 특수화를 통해서 primary template 의 인자 T를 R(A1, A2) 으로 분리한다는 점입니다.

그런데, 만약 result<>를 아래 처럼 사용하게 된다면 어떻게 될까요 ?

```
// primary template
template<typename T> struct result
{
    typedef T type;
};
int main()
{
    result<int>::type n;    // int는 함수가 아니므로 primary template 이 사용됩니다.
                            // n은 int 입니다.
}
```

result<>에는 반드시 함수 타입을 보내야 하는데, 사용자가 잘못 사용한 경우 입니다. 이때는 primary template 이 사용되므로 결국 n은 int로 결정됩니다.

result<>를 사용할 때 함수가 아닌 다른 타입이 전달될 때 에러를 발생하게 하려면 아래처럼 만

들면 됩니다.

```
template<typename T> struct result
{
    // 의도적으로 typedef를 제공하지 않습니다.
};
int main()
{
    result<int>::type n;    // primary template 안에는 ::type이 없으므로 error.
}
```

또 다른 방법은 부분 특수화 버전만 사용하고 primary 버전은 사용될 필요가 없으면 primary template을 선언만 해도 됩니다.

```
template<typename T> struct result; // 의도적으로 전방 선언만 제공합니다.

int main()
{
    result<int>::type n;    // primary template은 구조체의 완전한 definition 이
                            // 없으므로 error입니다.
}
```

이경우 primary template 이 사용되지 않더라도 부분 특수화를 버전을 만들기 위해서는 반드시 전방 선언은 필요합니다.

▌ argument<> 만들기

이번에는 argument template 을 만들어 봅시다. 역시, primary template 입니다.

```
template<int N, typename T> struct argument
{
    typedef T type;
};
argument<0, char(int, double)>::type ret;
```

argument는 몇 번째 인자의 타입을 구하는지를 전달 받아야 하므로 템플릿의 첫번째 인자로

int를 전달 받고 있습니다. primary template 만 있다면 "::type"은 역시 char(int, double) 이 됩니다. 첫번째 인자 타입인 int를 구하기 위해서는 역시 부분 특수화를 사용해야 합니다

먼저 부분특수화를 다음 처럼 한다고 생각해 봅시다.

```
template<int N, typename R, typename A1, typename A2> struct argument<R(A1, A2)>
{
    typedef ? type;
};
```

이 경우 몇 번째 인자의 타입을 요구하는지 알 수 없으므로 ? 를 채울 수가 없습니다.

따라서, argument<> 의 부분 특수화는 N이 0일때와 1일때로 나누어서 만들어야 합니다.

```
template<typename R, typename A1, typename A2> struct argument<0, R(A1, A2)>
{
    typedef A1 type;
};
template<typename R, typename A1, typename A2> struct argument<1, R(A1, A2)>
{
    typedef A2 type;
};
argument<0, char(int, double)>::type ret; // int
argument<1, char(int, double)>::type ret; // double
```

최종적으로 완성된 전체 코드는 다음과 같습니다.

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

template<typename T> struct result
{
    typedef T type;
};
template<typename R, typename A1, typename A2> struct result<R(A1, A2)>
{
    typedef R type;
};
```

```
template<int N, typename T> struct argument
{
    typedef T type;
};
template<typename R, typename A1, typename A2> struct argument<0, R(A1, A2)>
{
    typedef A1 type;
};
template<typename R, typename A1, typename A2> struct argument<1, R(A1, A2)>
{
    typedef A2 type;
};
char f(int n, double d)
{
    return 'A';
}
template<typename T> void foo(T& t)
{
    typename result<T>::type      ret;  // char
    typename argument<0, T>::type arg0; // int
    typename argument<1, T>::type arg1; // double

    cout << typeid(ret).name() << endl;
    cout << typeid(arg0).name() << endl;
    cout << typeid(arg1).name() << endl;
}
```

지금 까지의 코드는 함수의 인자가 2개일 때만 생각한 코드 입니다. 임의의 개수를 인자로 가지는 함수라면 어떻게 해야 할까요 ?

뒷 부분에 나오는 "가변인자 템플릿(Variadic Template)" 항목에서 살펴 보도록 하겠습니다.

using

C++14 에서는 remove_pointer<>를 보다 편리하게 사용할 수 있도록 using 을 사용해서 alias 를 제공하고 있습니다.

```
// C++14 부터 지원 됩니다.
template<typename T>
```

```
using remove_pointer_t = typename remove_pointer<T>::type;

template<typename T> void foo(T a)
{
    typename remove_pointer<T>::type n1;  // 코드가 좀 복잡해 보입니다.

    remove_pointer_t<T> n2; // 위 코드와 동일한 의미 입니다.
                            // typename 이나 ::type 등을 표시하지 않아도 됩니다.
}
```

▌ 핵심 개념 정리

- 변형된(연관된) 타입을 얻는 traits 를 만들려면 primary template 을 만든 후에, 원하는 타입을 얻을 수 있도록 부분 특수화를 해야 합니다.
- 재귀적(recursive)인 개념을 사용해서 type 을 구하는 기법도 널리 사용됩니다.
- using 을 사용하면 typename, ::type 등을 표시 하지 않고도 traits 를 사용할 수 있습니다.

항목 6-4

C++ 표준 type traits

배우게 되는 내용

<type_traits>

1. C++ 표준 type_traits

C++ 표준(C++11/14)이 다양한 traits를 사용하려면 <type_traits> 헤더가 포함되어야 합니다. <type_traits> 헤더 안에는 다양한 traits 를 제공하고 있습니다.

이와 같은 traits는 두 가지의 형태로 분류해 볼 수 있습니다.

- **is_pointer<>와 같이 템플릿 인자 T의 다양한 속성을 조사하는 traits**
- **remove_pointer<>와 같이 템플릿 인자 T의 변형(연관)된 타입을 얻는 traits**

타입 조사

is_pointer<>등의 traits는 다음의 3가지 방법으로 사용 할 수 있습니다.

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

template<typename T> void foo_imp(T& a, true_type)  {}
template<typename T> void foo_imp(T& a, false_type) {}

template<typename T> void foo(const T& a)
{
    bool b1 = is_pointer<T>::value; // 방법 1. ::value 를 통해서 확인
    bool b2 = is_pointer_v<T>;      // 방법 2. value에 대한 using 단축 표현 사용
    foo_imp(a, is_pointer<T>());    // 방법 3. 포인터 여부에 따라 call dispatch
}
int main()
{
    int n = 0;
    foo(&n);
}
```

변환 타입 얻기

remove_pointer<>등의 traits는 다음의 2가지 방법으로 사용 할 수 있습니다

```
template<typename T> void foo(const T& a)
{
    typename remove_pointer<T>::type n1;    // 1. ::type 으로 변형된 타입 얻기
    remove_pointer_t<T> n2;                 // 2. ::type에 대한 using 단축 표현 사용
}
```

C++ 표준은 제공하는 다양한 traits의 종류와 사용법은 "C++ 표준 문서"를 참고 하면 됩니다.

Section 7

Variadic Template

항목 7-1

variadic template

배우게 되는 내용

Variadic template 기본 모양

Parameter pack

sizeof...

pack expansion

1. 가변 인자 템플릿의 기본 모양

C++11 부터는 가변 인자 함수, 가변 인자 매크로와 유사하게 템플릿도 가변 인자로 만들 수 있습니다. 다음 코드를 생각해 봅시다.

```
template<typename T1, typename T2> class xtuple
{
};
int main()
{
    xtuple<int> t1;             // error
    xtuple<int, int> t2;        // ok
    xtuple<int, int, int> t3;   // error
}
```

xtuple<> 템플릿은 템플릿 인자가 2개이므로 반드시 타입을 2개 전달해야만 인스턴스화 할 수 있습니다.

하지만, C++11에서 추가된 가변 인자 템플릿을 사용하면 템플릿 인자의 개수에 제한이 없이 사용할 수 있습니다.

```
template<typename ... T > class xtuple
{
};
int main()
{
    xtuple<>    t0;             // ok. 템플릿 인자가 없어도 됩니다.
    xtuple<int> t1;             // ok.
    xtuple<int, int> t2;        // ok.
    xtuple<int, int, int> t3;   // ok.
}
```

함수 템플릿도 가변인자 템플릿으로 만들 수 있습니다. "..." 의 위치를 주의해서 보세요.

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

```
int main()
{
    foo();              // T :                    arg :
    foo(1);             // T : int,               arg : 1
    foo(1, 3.4, 'c');   // T : int, double, char   arg : 1, 3.4, 'c'
}
```

위 코드의 마지막 줄에서는 함수 템플릿 foo()에 인자를 3개 전달 했으므로 T안에는 3개의 타입이, arg안에는 3개의 값을 가지고 있습니다. 즉 T는 하나의 타입이 아닌 여러 개의 타입을 가지고 있게 됩니다. 그래서, 가변 인자 템플릿을 만들 때의 일반적인 코딩 관례는 템플릿 인자 이름을 T 대신 복수형으로 표현하는 경우가 많이 있습니다. 따라서, 일반적인 코딩 관례의 가변 인자 템플릿의 기본 모양은 다음과 같습니다.

```
// 가변인자 클래스 템플릿
template<typename ... Types> class xtuple
{
};

// 가변인자 함수 템플릿
template<typename ... Types> void foo( Types ... args )
{
}
```

2. parameter pack

가변 인자 함수 템플릿을 사용하면 0개 이상의 인자를 전달할 수 있습니다.

```
template<typename ... Types> void foo(Types ... args)
{
}
int main()
{
    foo();              // void foo() 모양이 함수가 생성.
    foo(1);             // void foo(int ) 모양이 함수가 생성.
    foo(1, 3.4, 'A');   // void foo(int, double, 'A') 모양의 함수 생성
}
```

이때, 사용자가 아래 처럼 사용하면

```
foo(1, 3.4, 'A'); // Types : int, double, char    args : 1, 3.4, 'A'
```

Types 안에는 int, double, char 의 3개의 타입을, args 안에는 1, 3.4, 'A' 의 3개의 값을 저장하고 있게 됩니다. 이때 args를 **"Parameter Pack"** 이라고 합니다.

sizeof...

Parameter pack 안에 있는 파라미터의 개수를 구하려면 아래 처럼 하면 됩니다.

```
template<typename ... Types> void foo(Types ... args)
{
    cout << sizeof...(args) << endl; // 3
    cout << sizeof...(Types) << endl; // 3
}
int main()
{
    foo(1, 3.4, 'A');
}
```

3. pack expansion

▌ args...

Parameter Pack의 이름 뒤에 "..." 을 붙이면 , Parameter Pack의 이름을 pack이 가진 각각의 요소로 치환 할 수 있습니다.

```
void goo(int a, int b, int c) {}

template<typename ... Types> void foo(Types ... args)
{
    goo(args);      // error. pack expansion 이 필요합니다.
    goo(args...);   // ok. pack의 이름을 pack의 요소(1,2,3)로 치환 합니다.
                    // goo(1,2,3);
}
int main()
{
    foo(1,2,3);
}
```

이 처럼 Parameter pack의 이름 뒤에 "..." 을 붙여서, pack의 이름을 pack이 가진 각각의 요소로 치환하는 것을 **"팩 확장(pack expansion)"** 이라고 합니다.

▌ pack expansion

팩 확장(Pack expansion)에서 "..." 은 pack의 이름 뿐 아니라 pack을 사용하는 다양한 패턴 뒤에 붙일 수 있습니다

"(pack 이름을 사용하는 패턴)..." => "패턴(요소1), 패턴(요소1), 패턴(요소3)"

다음 코드에서 다양한 확장의 예를 볼 수 있습니다. "..." 의 위치에 따라 어떻게 확장되는지를 주의 깊게 살펴 보시면 됩니다.

```
template<typename ... Types> void foo(Types ... args)
{
    // args 안에 있는 요소를 E1, E2 라고 할때
    goo( args... );     // goo( E1, E2)
    goo( &args...);     // goo( &E1, &E2)
    goo( ++args...);    // goo( ++E1, ++E2)
    goo(static_cast<int>(args)...);
                        // goo( static_cast<void*>(E1), static_cast<void*>(E2));

    goo( hoo(args)...); // goo( hoo(E1), hoo(E2))

    int x[] = { args... };  // int a[] = { E1, E2 };

    // Types 안의 있는 타입을 각각 T1, T2라 할때
    pair<Types...> p1;          // pair<T1, T2>
    tuple<Types...> t1;         // tuple<T1, T2>
    tuple<pair<Types...>> t2;   // tuple<pari<T1, T2>>
    pair<tuple<Types>...> p2;   // pair<tuple<T1>, tuple<T2> >
    tuple<pair<int, Types>...> t3; // tuple<pair<int, T1>, pair<int, T2>>
}
```

pack expansion 위치

이번에, 인자로 전달된 모든 요소를 화면에 출력하기 위해서 아래 처 럼 만들었다고 생각해 봅시다.

```
template<typename T> int print(const T& v)
{
    cout << v << endl;
    return v;
}
template<typename ... Types> void foo(Types ... args)
{
    print(args)...; // 의도는 print(1), print(2), print(3)
                    // 하지만 error.
}
int main()
{
    foo(1, 2, 3);
```

```
}
```

Parameter pack은 아무 곳에서나 확장 되지는 않습니다. 함수 인자, template 인자, 배열 초기화, barace-init-list 등 list 형태로 사용되는 곳에서만 확장이 가능 합니다. 따라서, 위코드는 컴파일하면 "parameter pack must be expanded in this context" 의 컴파일 에러가 발생합니다.

배열의 초기화 식에서는 팩 확장이 가능하므로, 코드를 아래 처럼 수정하면 됩니다.

```
template<typename ... Types> void foo(Types ... args)
{
    int dummy[] = { print(args)... }; // int dummy[] = { print(1), print(2), print(3) };
}

foo(1, 2, 3);
```

일단, 위 코드는 문제 없이 잘 컴파일 되고 실행 됩니다.

하지만, 위 코드도 몇가지 문제가 있습니다.

먼저, "foo()"의 처럼 인자가 없을 때 입니다. 이경우 args... 에 는 아무 요소도 없게 되므로 아래 처럼 컴파일 됩니다.

```
template<typename ... Types> void foo(Types ... args)
{
    int dummy[] = { print(args)... }; // int dummy[] = { };  이므로 error
}
foo();
```

해결책은 배열 초기화 앞에 요소 하나를 추가하면 됩니다.

```
int dummy[] = { 0, print(args)... }; // int dummy[] = {0, };  이므로 ok.
```

다음으로 print() 함수의 리턴 값이 void 인 경우입니다.

```
template<typename T> void print(const T& v) {} // void 리턴

template<typename ... Types> void foo(Types ... args)
```

```
{
    int dummy[] = { 0, print(args)... }; // error
}
foo(1,2,3);
```

해결책은 C의 , 연산자를 사용하면 됩니다.

```
template<typename ... Types> void foo(Types ... args)
{
    int dummy[] = { 0, ((void)print(args), 0)... }; // print리턴값은 무시되고,
                                                    // 0으로 배열을 채우게 됩니다.
}
foo(1,2,3);
```

마지막으로 print()대신, cout을 사용하고, call by value 대신 C++11/14의 "Perfect forwarding" 기술을 사용하도록 수정한 코드입니다.

```
template<typename ... Types> void foo(Types&& ... args)
{
    int dummy[] = { 0, (cout << std::forward<Types>(args) << " ", 0)... };
}
int main()
{
    foo(1, 2, 3);
}
```

4. parameter pack 에서 요소 꺼내기

이번에는 parameter pack 안에 있는 각각의 요소를 꺼내는 방법을 살펴 보도록 하겠습니다.

▌ Pack expansion 사용하기

전달된 모든 인자의 타입이 동일하다면 pack expansion을 사용해서 배열에 담아 사용 할 수 있습니다.

```
template<typename ... Types> void foo(Types ... args)
{
    int x[] = { args... };

    for (int n : x)
        cout << n << endl;
}
int main()
{
    foo(1, 2, 3);
}
```

위 코드는 인자가 하나도 없다면 문제가 될 수 있습니다. C++11 의 initializer_list<>를 사용하면 인자가 없는 경우도 해결할 수 있습니다.

```
template<typename ... Types> void foo(Types ... args)
{
    int x[] = { args... };    // error. 크기가 0인 배열은 만들수 없습니다.
    int y[] = { 0, args... }; // ok.

    initializer_list<int> e = { args... };  // ok. 요소가 없는 initializer_list는
                                            // 만들수 있습니다.
}
int main()
{
    foo();
}
```

각 요소의 타입이 다르다면 tuple에 담아서 사용할 수 있습니다.

```
template<typename ... Types> void foo(Types ... args)
{
    tuple<Types...> tp(args...);

    cout << get<0>(tp) << endl;
    cout << get<1>(tp) << endl;
    cout << get<2>(tp) << endl;
}
int main()
{
    foo(1, 3.4, 'A');
}
```

▌ recursive call 을 사용해서 요소 꺼내기

재귀 호출과 유사한 개념을 사용해서도 parameter pack에서 모든 요소를 꺼낼 수 있습니다. 단, 이때는 함수 템플릿의 모든 인자를 가변 인자로 하면 안되고, 1번째 인자는 일반 템플릿으로 전달 받아야 합니다.

코드를 다음과 같습니다. 주석을 주의 깊게 보시면 됩니다.

```
template<typename T, typename ... Types> void foo(T value, Types ... args)
{
    cout << value << endl;

    foo(args...); // foo( 2, 3)
                  // foo( 3 )
                  // foo();
}
// 재귀를 종료하기 위해 인자가 없는 경우의 함수를 제공합니다.
void foo() {}

int main()
{
    foo(1, 2, 3);   // value : 1,  args : 2, 3
}
```

foo(1,2,3)를 사용해서 호출하면 1은 value 로 전달되고, 나머지 2,3은 Parameter pack인 args에 전달됩니다. 이때, Pack expansion을 사용해서 다시 foo(args...) 으로 전달 하면 foo(2,3)이 되므로 value에는 2, args 에는 3이 전달됩니다. 이 과정을 반복하면 최종적으로는 인자가 없는 foo()가 호출되는데, 재귀의 종료를 위해 인자가 없는 foo()함수를 별도로 제공하고 있습니다.

주의 할 점은, 이런 방식은 진짜로 재귀 호출이 아니라는 점입니다.

실제, C++ 컴파일러에 의해 생성된 foo 함수는 하나가 아니라 인자가 1개, 2개, 3개 짜리 foo()를 생성하게 됩니다. 즉, 컴파일러는 다음과 같은 C++ 코드를 생성하게 됩니다.

```
void foo() {}
// 가변인자 템플릿으로 부터 컴파일러가 생성한 코드
void foo(int value)
{
    foo();
}
void foo(int value, int arg0)
{
    foo(arg0);
}
void foo(int value, int arg0, int arg1 )
{
    foo(arg0, arg1);
}
int main()
{
    foo(1, 2, 3);
}
```

따라서, foo()함수의 크기가 큰 경우 code bloat 를 고려해야 합니다. 그래서, 이런 방식으로 만들어 지는 함수는 대부분 작은 크기의 함수를 inline 함수로 만드는 것이 좋습니다.

또한, 아래 처럼 static 지역변수를 사용할 때 주의해야 합니다.

```
template<typename T, typename ... Types> inline void foo(T value, Types ... args)
{
    static int n = 0;   // 컴파일러에 여러개의 foo 함수를 생성하게 되므로
    ++n;                // 모든 함수는 각각 다른 n을 가지게 됩니다.

    foo(args...);
}
```

핵심 개념 정리

- 가변 인자 템플릿 기본 모양
- Parameter pack, pack expansion

항목 7-2

Fold Expression – C++17

배우게 되는 내용

Fold Expression 개념

Parameter pack 안에 있는 요소 출력하기.

1. Fold Expression

▌ Fold Expression 개념

C++17 부터 제공되는 "Fold Expression" 은 parameter pack 이 가진 모든 요소에 대한 이항 연산을 수행하는 단축 표현입니다.

Fold Expression 은 기본적으로 다음과 같은 4가지 형태가 가능합니다. 아래 표에서 "I" 는 초기값을 의미 합니다.

이름	Fold Expression	의미
Unary right fold	(pack op ...)	E1 op (E2 op (E3 op (E4 op E5)))
Unary left fold	(... op pack)	(((E1 op E2) op E3) op E4) op E5
binary right fold	(pack op ... op I)	E1 op (E2 op (E3 op (E4 op (E5 op I))))
Unary right fold	(I op ... op pack)	(((I op E1) op E2) op E3) op E4) op E5

[참고] Pack 안에 요소가 5개(E1, E2, E3, E4, E5) 있다고 가정한 코드 입니다.

▌ 예제

Fold Expression 을 사용하면 parameter pack 안에 있는 모든 요소에 대해서 다양한 이항 연산을 수행할 수 있습니다.

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

template<typename ... Types> void foo(Types ... args)
{
    // unary fold expression : 초기값이 없는 경우
    int n1 = (args + ...);      // 1 + (2 + (3 + (4 + 5)))
    int n2 = (... + args);      // (((1 + 2) + 3) + 4) + 5

    // binary fold expression : 초기값이 있는 경우
    int n3 = (args + ... + 10); // 1 + (2 + (3 + (4 + (5 + 10)))
    int n4 = (10 + ... + args); // ((((10 + 1) + 2) + 3) + 4) + 5
}
```

```
int main()
{
    foo(1, 2, 3, 4, 5);
}
```

[참고] 위 코드를 컴파일 하려면 컴파일러가 C++17 의 "Fold Expression" 문법을 지원해야 합니다. g++의 경우 7.x 이상 버전을 사용해야 합니다. VC++의 경우 2017년 6월 현재 이 문법을 지원하지 못합니다.

Pack 안의 모든 요소를 출력.

Fold Expression 을 사용하면 간단한 방법으로 parameter pack 안에 있는 모든 요소를 화면 출력할수 있습니다.

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

template<typename ... Types> void foo(Types ... args)
{
    // 아래 표현식에서 cout 은 초기값(I)이 됩니다.
    (cout << ... << args) << endl; // ((((cout << 1) << 2) << 3) << 4) << 5 << endl;
}
int main()
{
    foo(1, 2, 3, 4, 5);
}
```

항목 7-3

variadic template 활용

배우게 되는 내용

result<>, argument<>

tuple<>

get<>

1. result<>, argument<>

이번에는 앞에서(“항목15. 부분 특수화 활용”) 만들었던, 함수의 리턴 타입과 인자의 타입을 구하는 result<>, argument<> 템플릿을 다시 만들어 보도록 하겠습니다.

앞에서는 함수의 인자가 2개인 경우만 생각하고 만들었는데, 이번에는 인자의 개수에 제한을 두지 않고 만들어 보도록 하겠습니다.

result<> 만들기

앞에서 만들었던 result 모양은 다음과 같습니다.

```
template<typename T> struct result
{
    typedef T type;
};
template<typename R, typename A1, typename A2> struct result<R(A1, A2)>
{
    typedef R type;
};
int main()
{
    result<int(char, double)>::type ret; // int
}
```

위 코드의 문제점은 부분 특수화 모양에서 함수의 인자가 2개인 함수만 가능하다는 점입니다. 인자에 개수에 제한을 두지 않으려면 부분 특수화 버전에서 가변 인자 템플릿을 사용하면 됩니다.

```
// 함수 인자를 가변 인자 템플릿으로 만듭니다.
template<typename R, typename ... Types> struct result<R(Types...)>
{
    typedef R type;
};
int main()
{
    result<int(char, double, long, int)>::type ret; // int
}
```

▌ argument<> 만들기

다음은 함수 인자의 타입을 구하는 argument<> 템플릿 입니다.

먼저, primary template 입니다.

```
template<int N, typename T> struct argument
{
    typedef T type;
};
argument<0, int(char, long, int)>::type ret; // char 가 나와야 합니다.
```

0번째 인자의 타입을 구하기 위한 부분 특수화

이제, 함수 타입(int(char, int, long)) 으로부터 0번째 인자의 타입인 char를 구할 수 있도록 부분 특수화를 통해서 타입 분할을 해야 합니다.

먼저, 다음 코드 처럼 부분 특수화를 하면 어떨까요 ?

```
template<int N, typename R, typename ... Types> struct argument<N, R(Types...)>
{
    typedef ? type;
};
```

이경우, 2가지의 문제점이 있습니다.

- **부분 특수화에서 N을 사용하면 몇번째 인자 인지 알 수 없으므로 인자의 타입을 구할 수 없습니다. '0번째', '1번째' 등으로 특수화 해야 합니다.**
- **모든 인자를 가변 인자 템플릿(Types...) 으로 하면 인자의 타입을 알 수 없습니다. 첫번째 인자는 가변 인자 템플릿이 아닌 독립된 타입으로 해야 합니다.**

결국, 아래 처럼 만들어야 합니다.

```
// 0번째 인자의 타입을 구하기 위한 부분 특수화
// 핵심 : 0번째 인자 타입은 가변인자 템플릿이 아닌 독립된 타입(T)로
//        부분 특수화 합니다.
template<typename R, typename T, typename ... Types> struct argument<0, R(T, Types...)>
{
    typedef T type;
};
```

```
argument<0, int(char, double, long, int)>::type ret; // char
```

핵심은, 부분 특수화를 할 때 모든 인자를 가변 인자 템플릿으로 하면 안되고, 0번째 인자의 타입은 별도의 템플릿 T로 전달 받는다는 점입니다.

N번째 인자의 타입을 구하는 부분 특수화 - Recursive 활용

마지막으로, 0번째가 아닌 N번째 인자의 타입을 구하는 부분 특수화를 생각해 봅시다.

핵심은 다음 개념입니다.

```
argument<1, R(A1, A2, A3)>::type // argument<0, R(A2, A3)>::type 과 동일한 타입니다.
```

즉, F라는 함수의 N번째 인자의 타입은 "F함수에서 0번째 인자를 제거한 함수의 N-1번째 인자 타입"과 동일합니다.

결국, 템플릿 인자 N이 0이 될 때 까지 재귀적으로 수행하게 되면 결국 N은 0이 되고 0번째 인자의 타입을 구하는 부분 특수화 버전을 사용하게 됩니다.

```
template<int N, typename R, typename T, typename ... Types>
struct argument<N, R(T, Types...)>
{
    // 함수의 0번째 인자(T)를 제거한 타입에서 N-1 번째 인자의 타입을 구합니다.
    // 결국, N이 0이 되면 "0일때의 부분 특수화 버전"을 사용하게 됩니다.
    typedef typename argument<N-1, R(Types...)>::type type;
};
```

이와 같은 재귀(Recursive) 기술은 아주 널리 사용되는 기법이므로 반드시 정확하게 이해 해야 합니다.

마지막으로, 인자가 없는 함수를 위한 부분 특수화가 필요 합니다.

```
template<int N, typename R> struct argument<N, R(void)>
{
    typedef void type;
};
```

또는, primary template 다음과 같이 수정해도 됩니다.

```
// 함수 타입이 아니거나,
// 인자가 없는 함수의 경우는 다음의 primary template을 사용하게 됩니다.
template<int N, typename T> struct argument
{
    typedef void type;
};
// 아래 2개의 부분 특수화 버전은 인자가 최소 1개 이상일 경우만 사용됩니다.
template<typename R, typename T, typename ... Types>
struct argument<0, R(T, Types...)> { …… }

template<int N, typename R, typename T, typename ... Types>
struct argument<N, R(T, Types...)> { …… };
```

완성된 코드입니다.

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

// result template
template<typename T> struct result
{
    typedef T type;
};
template<typename R, typename ... Types> struct result<R(Types...)>
{
    typedef R type;
};

// argument template
template<int N, typename T> struct argument
{
    typedef void type;
};
template<typename R, typename T, typename ... Types>
struct argument<0, R(T, Types...)>
{
    typedef T type;
};
template<int N, typename R, typename T, typename ... Types>
struct argument<N, R(T, Types...)>
```

```
{
    typedef typename argument<N - 1, R(Types...)>::type type;
};

// 테스트 코드입니다.
template<typename T> void foo(T& t)
{
    typename result<T>::type ret;       // int
    typename argument<0, T>::type arg1; // int
    typename argument<1, T>::type arg2; // char**

    cout << typeid(ret).name() << endl;
    cout << typeid(arg1).name() << endl;
    cout << typeid(arg2).name() << endl;
}

int main(int argc, char** argv)
{
    foo(main);
}
```

2. tuple<> 만들기

C++ 표준에는 서로 다른 타입의 객체를 N개 저장 할 수 있는 tuple 이 있습니다. 또한, tuple에서 값을 꺼내기 위해서는 get<>을 사용합니다. 다음 코드는 tuple을 사용하는 간단한 예제입니다.

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

int main()
{
    tuple<int, double, char> t3(1, 3.4, 'A');

    cout << get<0>(t3) << endl;
    cout << get<1>(t3) << endl;
    cout << get<2>(t3) << endl;
}
```

이제부터 tuple을 직접 만들어 보도록 하겠습니다. tuple과 get는 이미 C++ 표준에 있으므로 이름을 xtuple, xget 으로 만들도록 하겠습니다.

Step 1. 기본 모양

xtuple 은 0~N 개 의 data를 보관할 수 있어야 합니다. 결국, template 인자를 가변 인자 템플릿으로 만들어야 합니다.

```
template<typename ... Types> struct xtuple
{
    static constexpr int N = 0;
};
int main()
{
    xtuple<>            t0;
    xtuple<int>         t1;
    xtuple<int, double> t2;
}
```

main 함수에서 볼 수 있듯이 xtuple<>은 가변 인자 템플릿이므로 0~N개의 템플릿 인자를 사용할 수 있습니다. 하지만, 아직까지는 아무 data도 보관하고 있지 않습니다. 보관하는 data가 없다는 의미로 N을 0으로 초기화 했습니다.

Step 2. Data 한 개를 보관하는 xtuple

이제, xtuple이 값을 보관할 수 있도록 만들어 보겠습니다. 일단, 모든 인자를 가변 인자로 받으면 전달된 타입을 알 수 없습니다, 그래서, 첫번째 인자 타입은 가변 인자 템플릿이 아닌 일반 템플릿으로 전달 받을 수 있도록 부분 특수화를 해야 합니다.

```
// primary template
template<typename ... Types> struct xtuple
{
    static constexpr int N = 0;
};

// 템플릿 인자가 한 개 이상 전달 되면 아래의 부분 특수화 버전이 사용됩니다
template<typename T, typename ... Types> struct xtuple<T, Types...>
{
    T value;

    xtuple() = default;
    xtuple(const T& v, const Types& ... args) : value(v) {}

    static constexpr int N = 1;
};

int main()
{
    xtuple<>    t0; // 템플릿 인자가 없으므로 primary template 버전을 사용합니다.

    // 아래 2줄의 경우는 부분 특수화 버전이 사용됩니다.
    xtuple<int>         t1(1);      // ok. 이제 1을 value에 보관합니다.
    xtuple<int, double> t2(1, 3.4); // 1을 value에 보관하지만 3.4는 보관되지 않습니다.
}
```

완성된 xtuple은 1개의 data를 보관 할 수 있습니다. 하지만, 위의 구현은 아래 처럼 xtuple객체를 만들 수 는 있지만 내부적으로는 3.4를 보관할 수 없습니다.

다음 단계에서 xtuple이 여러 개의 data를 보관하도록 코드를 변경해 보겠습니다.

Step 3. 여러 개의 Data를 저장하는 xtuple - recursive 와 상속

Step1에서 만든 xtuple<>은 생성자로 전달된 값 중에서 첫번째 요소만 보관할 수 있습니다. xtuple<>을 수정하기 전에 아래 코드를 다시 한번 정확히 정리해 놓기 바랍니다.

```
int main()
{
    xtuple<int, double, char> t3(1, 3.4, 'A');  // 첫번째 요소인 1만 보관합니다.
    xtuple<     double, char> t2(   3.4, 'A');  // 첫번째 요소인 3.4만 보관합니다.
    xtuple<             char> t1(        'A');  // 첫번째 요소인 'A' 만 보관합니다.
}
```

상속과 재귀 개념을 도입해서 여러 개를 보관할 수 있도록 xtuple<>을 변경해 보겠습니다. 다음 사항을 주의 깊게 봐야 합니다.

- 상속의 모양, 생성자 모양, static 변수 N 의 변경

```
// primary template
template<typename ... Types> struct xtuple
{
    static constexpr int N = 0;
};
// 부분 특수화 버전에 상속 개념이 추가 되었습니다.
// Base 클래스의 모양을 잘 생각해 보세요
template<typename T, typename ... Types>
struct xtuple<T, Types...> : public xtuple<Types...>
{
    T value;
    xtuple() = default;

    // 1번째 인자는 자신이 보관하고 나머지 인자는 기본 클래스에 전달합니다.
    // 기본 클래스 모양은 "xtuple<Types...>" 입니다.
    xtuple(const T& v, const Types& ... args) : value(v), xtuple<Types...>(args...) {}

    static constexpr int N = xtuple<Types...>::N + 1;
};
int main()
{
    xtuple<int, double, char> t3(1, 3.4, 'A'); // 1만 보관합니다.
                            // T : int, Types... : double, char 이므로
                            // Base 클래스 모양은 xtuple<double, char> 입니다.
}
```

main 함수에서 xtuple<int, double, char> 을 전달 했으므로 T는 int, Types... 는 double, char 입니다. 최종적으로 인자가 없는 xtuple<>은 primary template을 사용하게 되므로 재귀가 끝나게 됩니다. 재귀적으로 동작하게 되므로 아래와 같은 상속의 모양을 가지는 클래스가 생성됩니다.

```
struct xtuple<>
{
    // primary template 이므로 아무 것도 보관하지 않습니다.
};
struct xtuple<char> : xtuple<>
{
    int value;   // 'A' 를 보관합니다.
};
struct xtuple<double, char> : public xtuple<char>
{
    int value;   // 3.4를 보관합니다.
};
struct xtuple<int, double, char> : public xtuple<double, char>
{
    int value;   // 1을 보관합니다.
};
int main()
{
    xtuple<int, double, char> t3(1, 3.4, 'A');
}
```

Xtuple<>은 0개 이상 임의의 개수의 값을 보관할 수 있습니다. 남은 일은 xtuple<> 저장되어 있는 data에 접근할 수 있어야 합니다.

xtuple<>에 저장된 값을 접근하는 xget<>을 만들어 봅시다.

3. xget<> 만들기

xtuple<>에 저장된 값을 접근하는 xget<>을 만들어 봅시다.

▌ Step 1. 이름이 동일한 멤버 data

어떤 클래스의 멤버 data의 이름이 자신의 기본 클래스에 있는 멤버 data와 이름이 동일한 경우를 생각해 봅시다.

```
struct Base
{
    int value = 10;
};
struct Derived : public Base
{
    // 기본 클래스에 있는 이름과 동일한 이름의 멤버 data가 있습니다.
    int value = 20;
};
int main()
{
    Derived d;
    cout << d.value << endl; // 20. 자신의 멤버에 접근합니다.

    // 기반 클래스에 멤버에 접근하려면 객체를 기반 클래스타입으로 캐스팅 하면 됩니다.
    cout << static_cast<Base&>(d).value << endl; // 10
}
```

여기서 주의 할 점은 기반 클래스로 캐스팅 할 때 는 반드시 참조로 캐스팅 해야 합니다. 값으로 캐스팅하면 임시 객체가 생성되어서 lvalue 로 사용할 수 없게 됩니다.

```
static_cast<Base>(d).value = 0; // error. 값 캐스팅은 임시객체를 생성합니다.
                                // 임시객체는 lvalue가 될 수 없습니다.
static_cast<Base&>(d).value = 0;// ok. 참조 캐스팅은 lvalue 가 될 수 있습니다.
```

[참고] 임시객체와 lvalue에 대한 자세한 이야기는 C++11/14 과정을 참고 하시기 바랍니다.

xtuple에 값을 여러개 저장하면 자신은 첫번째 요소만 저장하고 나머지는 Base 클래스들이 각각

하나씩 저장하게 됩니다. 그런데, 모두 value라는 이름으로 저장하고 있습니다. 그래서, 각각의 멤버에 접근하려면 기반 클래스 타입으로 캐스팅하면 됩니다.

```
int main()
{
    // 3번째 기반 클래스인 xtuple<>             // 아무것도 보관하지 않습니다.
    // 2번째 기반 클래스인 xtuple<char>         // 'A' 보관 합니다.
    // 1번째 기반 클래스인 xtuple<double, char> // 3.4 보관 합니다.
    xtuple<int, double, char> t3(1, 3.4, 'A');  // 1   보관합니다.

    // 각 요소를 출력하려면 기반 클래스 타입으로 캐스팅 하면 됩니다.
    cout << t3.value << endl; // 1
    cout << static_cast<xtuple<double, char>&>(t3).value << endl;   // 3.4
    cout << static_cast<xtuple<char>&>(t3).value << endl;           // 'A'
}
```

결국, xtuple<>이 보관하는 N번째 요소의 타입을 구하는 xget<> 함수 템플릿은 다음의 모양을 가지게 됩니다. xget<>의 리턴 타입과 캐스팅부분을 자세히 보시면 됩니다.

```
template<int N, typename T>
튜플 T의 N번째 요소의 타입& xget(T& tp)
{
    return static_cast<튜플 T의 N번째 기반 클래스 타입&>(tp).value;
}
int main()
{
    xtuple<int, double, char> t3(1, 3.4, 'A');

    double d = xget<1>(t3);
}
```

결국, 필요한 것은 xtuple<> N번째 요소의 타입과 N번째 요소를 보관하는 기반 클래스 타입을 구할 수 있으면 됩니다.

▌ Step 2. xtuple<> 의 N 번째 타입 요소 구하기

xtuple의 N번째 요소의 타입을 구하는 xtuple_element_type<>을 생각해 봅시다.

만드는 방법은 앞 부분에서 만든 함수의 인자의 타입을 구하는 argument<>와 거의 유사합니다.

primary template 입니다.

```
// primary template
template<int N, typename T> struct xtuple_element_type
{
};
xtuple_element_type<0, xtuple<int, char>>::type n;  // ::type 이 없으므로 error입니다.
                                                    // 결국, primary template을 사용하
                                                    // 지 않고, 부분 특수화 버전을
                                                    // 사용하게 됩니다.
```

primary template은 실제 사용되지 않고 부분 특수화 버전만 사용할 것이기 때문에 내부적으로는 어떠한 typedef도 넣지 않았습니다. 또한, 이 처럼 부분 특수화만 사용되만 primary 버전이 사용되지 않은 경우는 구현부를 생략하고 선언만 제공해도 됩니다.

0번째 요소의 타입을 구하기 위한 부분 특수화 버전.

```
// primary template, 사용되지 않으므로 선언만 제공합니다.
template<int N, typename T> struct xtuple_element_type;

// 튜플 T의 0번째 요소의 타입을 구하기 위한 부분 특수화 입니다.
template<typename T, typename ... Types>
struct xtuple_element_type<0, xtuple<T, Types...>>
{
    // xtuple<T, Types..>의 0번째 요소의 타입은 T입니다.
    typedef T type;

    // 0번째 요소를 저장하는 튜플 타입은 자기 자신입니다.
    typedef xtuple<T, Types...> tupleType;
};
int main()
{
    xtuple_element_type<0, xtuple<int, char>>::type n;      // int 타입입니다.
    xtuple_element_type<0, xtuple<int, char>>::tupleType t; // 0번째 요소(int)를
                                                            // 저장하는 튜플의 타입
                                                // 자기 자신 타입(xtuple<int,char>) 입니다.
}
```

::type은 0번째 요소의 타입이고, ::tupleType은 0번째 요소를 저장하는 튜플의 타입니다. 임의의 xtuple의 0번째 요소는 Base 클래스가 아닌 자기 자신이 보관하고 있습니다.

N번째 요소의 타입을 구하기 위한 부분 특수화 버전.

xtuple<T, Types...> 의 1번째 요소의 타입은 xtuple<Types...> 의 0번째 요소의 타입입니다.

즉, N을 하나 줄이고 T를 제거하는 과정을 N == 0 될 때 까지 재귀적으로 만들면 됩니다.

```
// N > 0 일때의 부분 특수화 입니다.
template<int N, typename T, typename ... Types>
struct xtuple_element_type<N, xtuple<T, Types...>>
{
    typedef typename xtuple_element_type<N - 1, xtuple<Types...>>::type      type;
    typedef typename xtuple_element_type<N - 1, xtuple<Types...>>::tupleType tupleType;
};
```

결국 특정 튜플의 N 번째 요소의 타입은 xtuple_element_type<N, 튜플>::type이 되고, N번째 요소를 저장하는 튜플의 타입은 xtuple_element_type<N, 튜플>::tupleType 입니다.

최종적으로 완성된 xtuple<>과 xget<> 코드 입니다.

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

// xtuple
template<typename ... Types> struct xtuple
{
    static constexpr int N = 0;
};

template<typename T, typename ... Types>
struct xtuple<T, Types...> : public xtuple<Types...>
{
    T value;
    xtuple() = default;
    xtuple(const T& v, const Types& ... args) : value(v), xtuple<Types...>(args...) {}
    static constexpr int N = xtuple<Types...>::N + 1;
};
```

```
// xtuple_element_type
template<int N, typename T> struct xtuple_element_type;

template<typename T, typename ... Types>
struct xtuple_element_type<0, xtuple<T, Types...>>
{
    typedef T type;
    typedef xtuple<T, Types...> tupleType;
};

template<int N, typename T, typename ... Types>
struct xtuple_element_type<N, xtuple<T, Types...>>
{
    typedef typename xtuple_element_type<N - 1, xtuple<Types...>>::type      type;
    typedef typename xtuple_element_type<N - 1, xtuple<Types...>>::tupleType tupleType;
};

// get
template<int N, typename T> typename xtuple_element_type<N, T>::type& xget(T& tp)
{
    return static_cast<typename xtuple_element_type<N, T>::tupleType&>(tp).value;
}

int main()
{
    xtuple<int, double, char> t3(1, 3.4, 'A');

    xget<0>(t3) = 10;

    cout << xget<0>(t3) << endl; // 10
    cout << xget<1>(t3) << endl; // 3.4
    cout << xget<2>(t3) << endl; // 'A'
}
```

핵심 개념 정리

- result<>, argument<> 만들기
- 상속을 사용한 xtuple<> 만들기
- xget<> 만들기