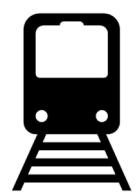


목 차

Template Basic	1
Template 결과 확인	6
MISC	10
Function Template	15
Class Template	19
Dependent Name	37
Template Parameter	40
type deduction	48
Explicit instantiation	59
Template specialization	62
std::conditional	70
Template meta programming	72
Variable template	75
Type traits	78
Variadic template	102
Make tuple	109
Fold expression	115
C RTP	117
SFINAE	123



template basic



핵심 정리

- 함수 오버로딩(function overloading)
 - ⇒ 인자의 형태(타입, 개수)가 다르면 "동일 이름의 함수를 여러 개 만들 수 있다."
 - ⇒ `square(int)`, `square(double)`

- 특징

장점	단점
함수 이름이 동일 하므로 "사용자 입장에서는 하나의 함수처럼 사용" 하게 된다. 사용하기 쉬운 일관된 형태의 라이브러리 구축.	<code>square</code> 함수를 만들 때, 인자 타입과 반환 타입만 다르고 "구현이 동일(유사)한 함수를 여러 개 만들어야" 한다.

- C++ 언어의 해결책

- ⇒ 구현이 동일(유사)한 함수가 여러 개 필요하면 "함수를 만들지 말고 함수를 생성하는 틀(템플릿)"을 만들자.



핵심 정리

- square 함수를 "함수의 틀(square template)" 로 변경하는 방법
 - ① 함수 앞에 "`template<typename T>`" 추가
 - ② "`int` ➔ `T`" 로 변경



핵심 정리

square template(틀) 을 사용해서 컴파일러가 실제 함수를 생성
"template instantiation(템플릿 인스턴스화)" 라고 부름.

개발자가 만든 코드

```
template<typename T>
T square(T a)
{
    return a * a;
}

int main()
{
    square<int>(3);
    square<double>(3.4);
}
```

```
template<>
int square<int>(int a)
{
    return a * a;
}
```

```
template<>
double square<double>(double a)
{
    return a * a;
}
```

square 틀(template)

+

main() 함수

int square(int)

double square(double)

main() 함수



핵심 정리

● 함수 템플릿을 사용하는 방법

템플릿 인자(타입)
을 명시적으로 전달

```
square<int>(3);
```

사용자가 전달한 타입으로 함수를 생성

템플릿 인자(타입)
을 생략

```
square(3);
```

컴파일러가 함수 인자를 보고 타입을 추론(type deduction)

● 템플릿 인스턴스화 결과를 확인 하는 방법

(1) 컴파일 결과로 생성된 "**어셈블리 코드로 확인**"
godbolt.org 사이트 (Compiler Explorer)

(2) **cppinsights.io**

(3) 템플릿 인스턴스화의 결과로 생성된 함수의 이름을 출력

template instantiation 결과 확인



방법 1. 어셈블리 코드로 확인

- **godbolt.org (Compiler Explorer)**
 - ⇒ 다양한 언어의 컴파일 결과를 어셈블리 코드로 확인가능
 - ⇒ **"Using Compiler Explorer"** 영상 참고
- 템플릿 자체는 컴파일시간에 **"컴파일러가 함수를 생성하기 위해서만 사용"**된다.
 - ⇒ 템플릿 자체의 기계어 코드가 생성되지 않는다.
- 함수 템플릿을 만들고 사용하지 않으면
 - ⇒ 인스턴스화(instantiation) 되지 않는다.
 - ⇒ 실제 함수는 생성되지 않는다.
- 코드 폭발(Code Bloat)
 - ⇒ 템플릿이 너무 많은 타입에 대해 **"인스턴스화(instantiation)"** 되어서 코드 메모리가 증가하는 현상.



방법 2. cppinsight.io

- **cppinsight.io**

- ⇒ C++ 코드의 다양한 내부 원리를 보여 주는 사이트
- ⇒ `template`, `range-for`, `virtual function` 등



방법 3. 인스턴스화 된 함수 이름 출력

- 함수 이름을 담은 매크로

<code>__FUNCTION__</code>	C++ 표준 매크로, signature 가 포함되지 않은 함수 이름만 있다.
<code>__FUNCSIG__</code>	비 표준, cl 컴파일러 전용 signature 포함
<code>__PRETTY_FUNCTION__</code>	비 표준, g++, clang 컴파일러 signature 포함

- `std::source_location`

- ⇒ C++20 부터 지원 하는 클래스
- ⇒ 파일 이름, 라인 번호, 함수 이름 등을 구할 수 있다.
- ⇒ 함수 이름에서 signature 포함 여부는 컴파일러마다 다르다.

MISC



핵심 정리

- 함수 템플릿을 만드는 방법

```
class
template<typename T>
T square(T a)
{
    return a * a;
}
```

```
auto square(auto a)
{
    return a * a;
}
```

C++20 부터 지원

- 함수 템플릿을 사용하는 방법

템플릿 인자(타입)
을 명시적으로 전달

square<int>(3);

사용자가 전달한 타입으로 함수를 생성

템플릿 인자(타입)
을 생략

square(3);

컴파일러가 함수 인자를 보고 타입을 추론(type deduction)



핵심 정리

- 함수와 함수 템플릿

square

함수가 아니라 "**함수 템플릿(틀)**"이다.
square 의 주소는 구할 수 없다.

square<**타입**>

함수(함수이름).
함수의 주소는 구할 수 있다.



핵심 정리

```
// square.h  
template<typename T>  
T square(T a);
```

```
// square.cpp  
template<typename T>  
T square(T a)  
{  
    return a * a;  
}
```

```
// using_square.cpp  
#include "square.h"  
  
int main()  
{  
    square<int>(3);  
}
```

이순간 `square<int>` 함수를 생성하려면
"square template 의 구현"을 컴파일러가
알아야 한다.

템플릿은 구현부를 헤더 파일에 작성해야 한다.



핵심 정리

- template 에 대한 다양한 관점
 - ⇒ 함수(클래스)를 생성하는 **틀(template)**
 - ⇒ **"a family of functions(class)"**
 - ⇒ **"generic functions(class)"**
- template 의 종류
 - ① function template
 - ② class template
 - ③ variable template
 - ④ using template (template alias)

function template



핵심 정리

- 동일한 이름의 함수와 함수 템플릿을 만들 수 있다.

square(3)	square(int)
square<>(3) square<int>(3) square(3.4)	square(T)

- square(int) 와 template 으로 부터 생성된 square<int>(int) 는 함수 이름이 다르다.

square(int)	square(int)	_Z6squarei: ?square@@YAHH@Z	(g++) (cl)
square(T)	square<int>(int)	_Z6squareIiET_S0_: ??\$square@H@@YAHH@Z	(g++) (cl)



핵심 정리

- `add` 의 반환 타입을 표기하는 방법
 - ① 사용자가 직접 템플릿 인자로 전달
 - ② `auto`, `decltype` 사용
 - ③ `type_traits` 기술 사용 (`std::common_type`)



핵심 정리

- `decltype(a + b)`

⇒ "`a + b`" 표현식의 결과로 나오는 값의 "타입"

- 모든 변수는 "선언 후에 사용"되어야 한다

```
a = 10;    ← ERROR. 변수를 선언 전에 사용하는 코드
int a;     ← 변수 선언
a = 20;
```

```
template<class T1, class T2>
decltype(a + b) add( T1 a, T2 b )
                     ERROR
                     ↑
                     a, b 변수의 선언
```

```
template<class T1, class T2>
auto add( T1 a, T2 b ) -> decltype(a + b)
                     ↑
                     a, b 변수의 선언
                     OK
```

class template



핵심 정리

- `class` → `class template`
 - ⇒ `class` 앞에 "**template<typename T>**" 추가
 - ⇒ `class` 내부에서 필요한 곳을 "**T**"로 변경
- `Vector` vs `Vector<타입>`

<code>Vector</code>	template 의 이름
<code>Vector<타입></code>	클래스(타입)의 이름



핵심 정리

```
template<typename T>
class Object
{
public:
    void mf1(int n) {}
    void mf2(T n) {}

    template<typename U>
    void mf3(U n) {}
};
```

클래스 템플릿의 멤버 함수
함수 자체는 템플릿이 아님.

멤버 함수 템플릿
함수 자체가 템플릿

- 멤버 함수 템플릿의 외부 구현 모양

```
template<typename T> template<typename U>
void Object<T>::mf3(U n) {}
```



핵심 정리

<code>Point(const Point& p)</code> <code>Point(const Point<T>& p)</code>	복사 생성자 자신과 동일한 타입만 받을 수 있다.
<code>Point(const Point<int>& p)</code>	생성자 <code>Point<int></code> 타입만 받을 수 있다.
<code>template<typename U></code> <code>Point(const Point<U>& p)</code>	Generic (복사) 생성자 임의 타입의 <code>Point</code> 를 받을 수 있다.

● Coercion by Member Template

⇒ "T가 U로 복사(대입) 가능하다면

`Point<T>` 도 `Point<U>`로 복사(대입) 가능해야
한다."



핵심 정리

generic ctor

```
template<typename U>
Point(const Point<U>& p)
    : x(p.x), y(p.y)
{
}
```



```
template<>
Point(const Point<std::string>& p)
    : x(p.x), y(p.y) {}
```

std::string 객체가 int 로
복사될 수 없으므로 error

generic ctor with requires

```
template<typename U>
requires std::is_convertible_v<U, T>
Point(const Point<U>& p)
    : x(p.x), y(p.y)
{
}
```

"SFINAE",
"enable_if",
"Concept" 강의 참고



핵심 정리

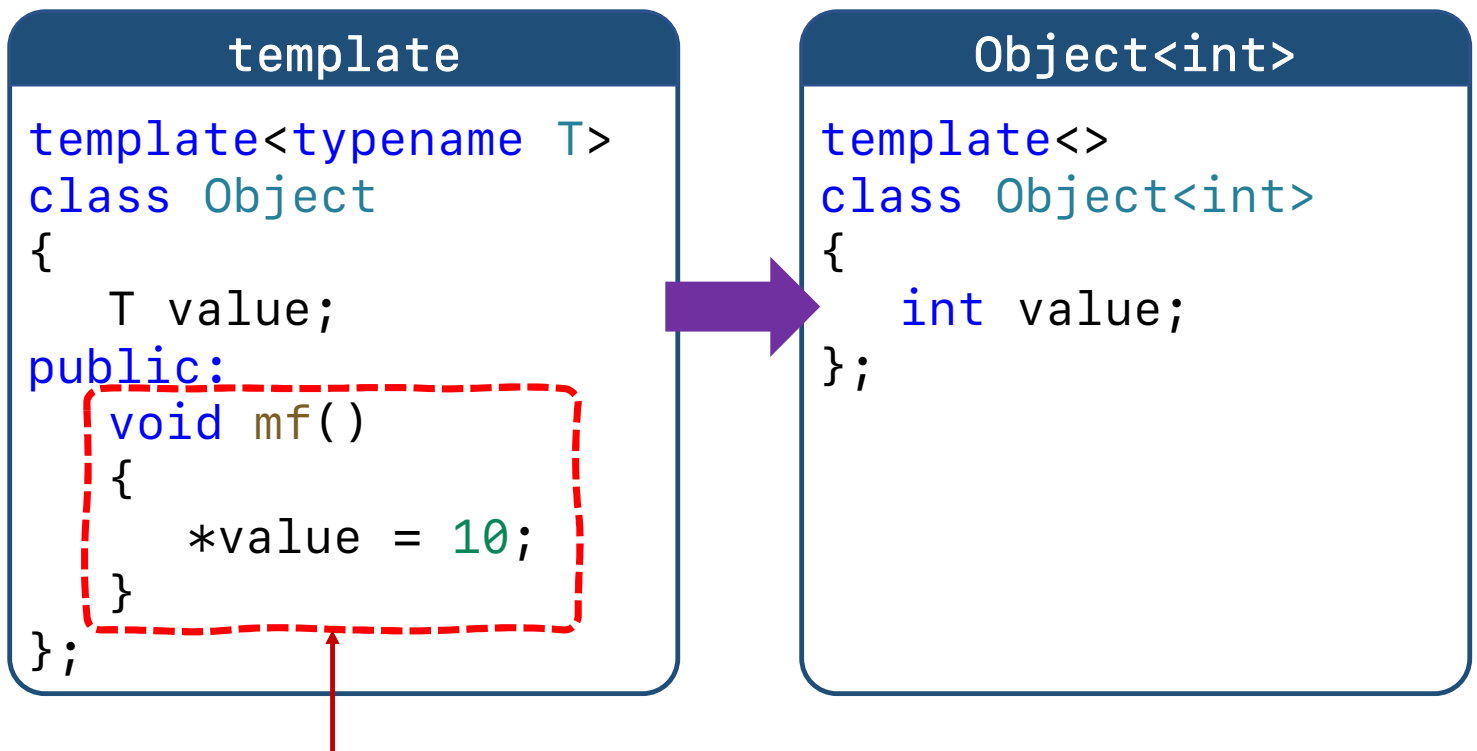
- member function template 기본 모양
- coercion by member template 기술



핵심 정리

● lazy instantiation

⇒ 클래스 템플릿의 멤버함수는 사용된 것만 인스턴스화 된다.



- ① 템플릿 코드에서는 `value` 의 타입이 결정되지 않았으므로 에러가 아니다.
- ② `Object<int>` 로 인스턴스화 되었을 때 "**`mf()` 멤버 함수는 사용된 적인 없으므로 인스턴스화 되지 않는다.**"



핵심 정리

- `std::vector`

- ⇒ `vector` 는 앞쪽으로 삽입/삭제 될 수 없다

- ⇒ `push_front`, `pop_front` 멤버 함수가 없다.



핵심 정리

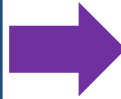
- `class` 의 `static member data`
 - ⇒ 객체를 생성하지 않아도 메모리에 놓인다.
(`static member` 가 `user type` 이면 생성자호출)
- `class template` 의 `static member data`
 - ⇒ `static member` 를 사용하지 않으면 인스턴스화 되지 않는다.



핵심 정리

if

```
template<typename T>
void fn(T value)
{
    if ( false )
        *value = 10;
}
```



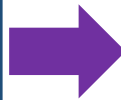
```
template<>
void fn<int>(int value)
{
    if ( false )
        *value = 10;
}
```

Error !

`if (false)` 이므로 실행은 안되지만 "인스턴스화에는 포함" 된다.

if constexpr

```
template<typename T>
void fn(T value)
{
    if constexpr (false)
        *value = 10;
}
```



```
template<>
void fn<int>(int value)
{
}
```



핵심 정리

- 동일한 이름이 함수(템플릿)이 여러 개 있을 때
 - ⇒ 어느 함수를 사용할지는 컴파일 시간에 결정한다.
 - ⇒ 선택되지 않은 함수가 템플릿 인 경우 인스턴스화 되지 않는다.



핵심 정리

- 동일 이름의 함수와 함수 템플릿이 있을 때
 - ⇒ 함수가 우선적으로 선택 된다.
- 함수가 선언만 있다면
 - ⇒ 템플릿을 사용하는 것이 아니라,
 - ⇒ 함수의 구현부가 없으므로 링크 에러 발생



Point<int>

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

    friend std::ostream&
    operator<<(std::ostream& os, const Point<int>& pt);
}
```

함수 템플릿이 아닌 "일반 함수 operator<<()" 의 선언



핵심 정리

Point<int>

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

    template<typename U>
    friend std::ostream&
    operator<<(std::ostream& os, const Point<U>& pt);
};
```

friend

```
operator<<(Point<int>)
{
}
```

friend

```
operator<<(Point<double>)
{
}
```

friend

```
operator<<(Point<long>)
{
}
```

- Point<int> 는 operator<<(Point<int>) 만 friend 관계로 만들면 된다.



Point<int>

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

    friend std::ostream&
    operator<<(std::ostream& os, const Point<int>& pt);
}
```

함수 템플릿이 아닌 "일반 함수 operator<<()" 의 선언



Point<int>

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

    friend std::ostream&
    operator<<(std::ostream& os, const Point<int>& pt)
    {
        std::cout << pt.x << ", " << pt.y << std::endl;
        return os;
    }
};
```



핵심 정리

- C++ template 의 종류
 - ① function template
 - ② class template
 - ③ **variable template**
 - ④ using template (template alias)
- variable template은 왜, 언제 사용하는가 ?
 - ⇒ 주로 "**template specialization 문법**"과 같이 사용.
 - ⇒ "**type_traits**" 구현의 핵심 문법
 - ⇒ 실제 STL 의 구현에서 다양하게 활용되고 있다.



핵심 정리

- 타입에 대한 별칭(alias) 를 만드는 방법

C style

```
typedef std::unordered_set<int> SET;
```

C++11 using

```
using SET = std::unordered_set<int>;
```

- 타입이 아닌 템플릿에 대한 별칭을 만들 수 있을까 ?

타입 alias

```
std::unordered_set<int> → SET
```

템플릿 alias

```
std::unordered_set → SET
```

```
SET<int> → std::unordered_set<int>
```

- C++ template 의 종류

① function template

② class template

③ variable template

④ using template (template alias)



Dependent Name



핵심 정리

- **dependent name**

⇒ a name that depends on a template parameter

⇒ 3가지 종류

non-type	
type	typename 을 붙여야 한다.
template	template 을 붙여야 한다.



핵심 정리

- STL 과 `value_type` 멤버

⇒ STL 의 모든 컨테이너에는 "`value_type`" 이라는 멤버 타입이 있다.

⇒ 컨테이너가 저장하는 타입이 필요할 때 사용.

<code>std::list<int>::value_type</code>	
<code>typename std::list<T>::value_type</code>	dependent name
<code>typename T::value_type</code>	dependent name



TEMPLATE PARAMETER



핵심 정리

- `template parameter` 의 종류
 - ⇒ `type`
 - ⇒ `non-type`
 - ⇒ `template`



핵심 정리

- `std::stack`

- ⇒ 선형 컨테이너(`vector`, `list`, `deque`)의 멤버 함수이름을 `stack` 처럼 사용할 수 있게 변경한 것
- ⇒ "**container adapter**"

```
stack<int, std::vector<int>>
```



```
stack<int, std::vector>
```



핵심 정리

- `std::vector`

⇒ template parameter 가 2개인 template

```
template<class T, class Ax = std::allocator<T> >  
class vector;
```



핵심 정리

- STL 의 sequence container
 - ⇒ 인자의 갯수가 2개인 template

```
template<class T, class Ax = std::allocator<T> >  
class vector;
```

```
template<class T, class Ax = std::allocator<T> >  
class list;
```

```
template<class T, class Ax = std::allocator<T> >  
class deque;
```



핵심 정리

- template parameter 의 종류 - 3가지

- ① type

- ② template - 이전 강좌 참고

- ③ **non-type** (타입이 아닌 것들)

- NTTP (Non-Type Template Parameter)

정수형 상수	컴파일 시간 상수만 가능 변수는 안됨
실수형 상수	C++20 부터 지원
enum 상수	enum 또는 enum class
포인터, 함수 포인터	static storage 만 가능 지역변수 주소 안됨.
auto	C++17 부터 가능



핵심 정리

- template parameter 의 auto

```
template<int N, double D, auto A>
```

```
struct Triple
```

```
{
```

```
};
```

int, double, 포인터 등

모든 non-type parameter

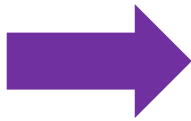
전달가능

- function parameter 의 auto (C++20)

```
void foo(auto a)
```

```
{
```

```
}
```



```
template<typename T>
```

```
void foo(T a)
```

```
{
```

```
}
```




핵심 정리

● `raw array` vs `std::vector` vs `std::array`

- ⇒ 모든 요소를 연속된 메모리에 보관하고, `[]` 연산자를 사용해서 요소 접근
- ⇒ 지역 변수로 생성할 경우

	요소저장공간	멤버 함수	크기 변경
<code>raw array</code>	stack	X	X
<code>std::vector</code>	heap	0	0
<code>std::array</code>	stack	0	X

type deduction



핵심 정리

- `template type deduction (inference)`
 - ⇒ 사용자가 `type parameter` 를 생략할 경우
컴파일러는 "**함수 인자를 보고 타입을 결정**"한다.
 - ⇒ 타입 추론(연역)
- 3가지의 규칙
- 추론된 타입을 확인하는 방법
 - ① **`cppinsights.io`**
 - ② **`godbolt.org` (어셈블리 코드로 확인)**
 - ③ 인스턴스화 된 함수 이름 출력



핵심 정리

- type deduction 규칙

⇒ 함수 인자의 모양에 따라 3가지 규칙

T


함수 인자가 가진 "**const, volatile, reference 속성을 제거**"하고 T 타입을 결정

T&

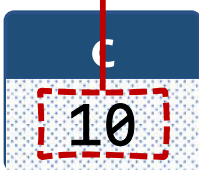
함수 인자가 가진 "**reference 속성만 제거**"하고 T 타입을 결정. const, volatile 은 유지

T&&

```
void f1(T arg)
{
    arg
    10
}
```



```
void f2(T& arg)
{
    }
}
```



const 객체를 가리키려면
const & 가 되어야 한다.



핵심 정리

- type deduction 규칙

⇒ 함수 인자의 모양에 따라 3가지 규칙

T

함수 인자가 가진 "**const, volatile, reference 속성을 제거**"하고 T 타입을 결정

T&

함수 인자가 가진 "**reference 속성만 제거**"하고 T 타입을 결정. const, volatile 은 유지

T&&

- forwarding reference (T&&)

⇒ **lvalue** 와 **rvalue** 를 모두 받을 수 있는 템플릿

3 (rvalue)

T

int

T&&

int&&

n (lvalue)

int&

int&

C++ Intermediate 과정의 "**perfect forwarding**" 강좌 참고



핵심 정리

● auto type deduction

⇒ template type deduction 규칙과 동일하다.

<code>T arg = 함수인자</code>	함수의 인자를 보고 T의 타입 결정
---------------------------	---------------------

<code>auto a = 우변</code>	우변을 보고 auto 의 타입 결정
--------------------------	---------------------

● type deduction 규칙

⇒ 함수 인자의 모양에 따라 3가지 규칙

T	함수 인자가 가진 " const, volatile, reference 속성을 제거"하고 T 타입을 결정
---	--

T&	함수 인자가 가진 " reference 속성만 제거"하고 T 타입을 결정. <code>const, volatile</code> 은 유지
----	--

T&&	인자가 3(rvalue) 라면 <code>T = int, T&& = int&&</code> 인자가 n(lvalue) 라면 <code>T = int&, T&& = int&</code>
-----	--



핵심 정리

- int x[3] 일 때
⇒ x의 정확한 타입은 int[3]

auto a1 = x;	int a1[3] = x;	compile error
	int* a1 = x;	compile ok
auto& a2 = x;	int(&a2)[3] = x;	compile ok



argument decay

배열 전달시 포인터로 받게 되는 현상



f1(T arg)	T = int* f1(int* arg)
f2(T& arg)	T = int[3], arg = int(&)[3] f2(int(&arg)[3])



핵심 정리

- 문자열의 정확한 타입

⇒ **char 배열 (char* 아님)**

"banana"	char[7]
"apple"	char[6]



핵심 정리

- template type deduction

function
template

type parameter 를 전달하지 않으면
함수 인자를 통해서 타입 추론.

class
template

~ C++14 까지는 타입 추론 안됨

C++17 부터 타입 추론 가능



핵심 정리

● `class template`

~ C++14

반드시 타입 인자를 전달해야 한다.

C++17 ~

타입 인자 생략이 가능하다.

● `Object Generator`

⇒ 클래스 템플릿의 타입 인자가 복잡한 경우 사용했던 기술.



● Object Generator

- ⇒ 복잡한 타입의 객체를 쉽게 생성하기 위한 함수를 제공한다.
- ⇒ "클래스 템플릿은 타입 추론 될 수 없지만(C++17이전), 함수 템플릿은 타입 추론 될 수 있다." 는 문법을 활용한 기술



핵심 정리

- `std::pair`, `std::tuple` 등의 객체를 생성하는 방법

클래스 이름을
직접 사용

```
std::pair<int, double> p1(3, 3.4)
```

```
std::pair p2(3, 3.4)
```

C++17

`make_xxx`
함수 사용

```
std::make_pair(3, 3.4);
```

Explicit Instantiation



핵심 정리

● Template Instantiation

⇒ 템플릿 으로부터 "실제 함수/클래스를 생성"하는 과정

implicit
instantiation

```
fn<int>(3);  
fn(3);
```

explicit
instantiation

```
template void fn<int>(int);  
template void fn<>(int);  
template void fn(int);  
  
template class Type<int>;  
template void Type<double>::mf1();
```



핵심 정리

square.h

```
template<typename T>
T square(T a);
```

square.cpp

```
template<typename T>
T square(T a)
{
    return a * a;
}
```

main.cpp

```
#include "square.h"

int main()
{
    square(3);
    square(3.3);
}
```

컴파일러 시간에

int square(int)

double square(double)

함수를 생성해야 한다.

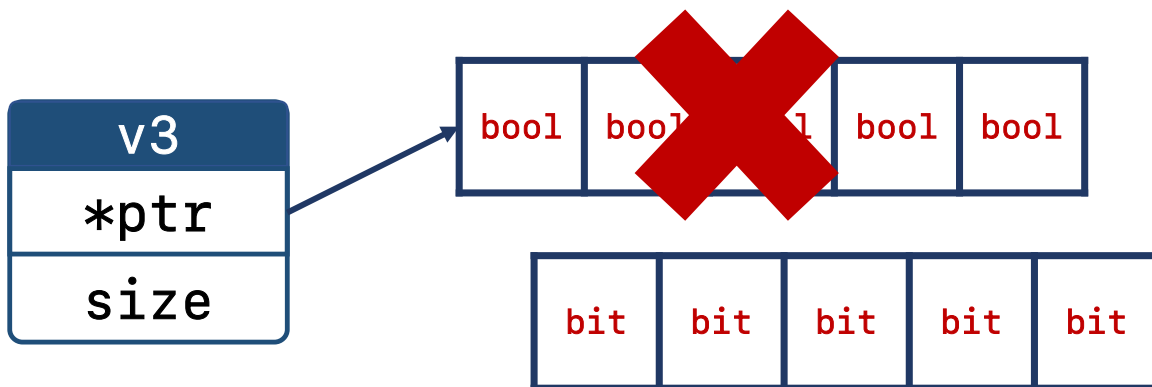
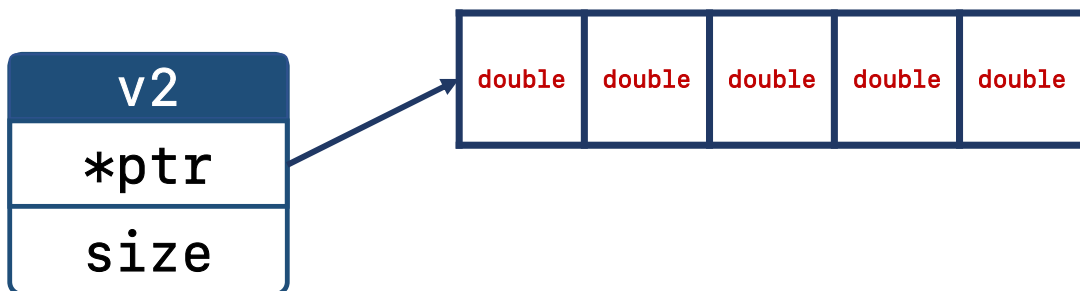
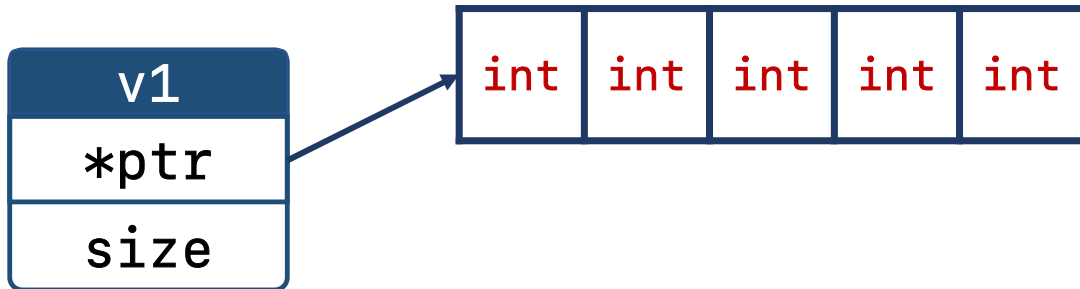
- Template 의 일반적인 코딩 관례

⇒ 함수 템플릿의 **"구현부를 헤더 파일에 포함"**.

Template Specialization



핵심 정리





```
template<typename T> class Vector
{
    // .....
};
```

primary template

```
template<> class Vector<bool>
{
    // .....
};
```

specialization
(특수화, 전문화)

```
template<typename T> class Vector<T*>
{
    // .....
};
```

partial specialization
(부분 특수화, 부분 전문화)



핵심 정리

```
template< typename T, typename U >
```

```
struct Object
```

```
{
```

```
};
```

primary template 의 인자가 2개 일 때

partial specialization 버전의 인자의 갯수는
다를 수 있다.

```
template< typename A, typename B, typename C >
```

```
struct Object< A, Object<B, C> >
```

```
{
```

```
};
```

이부분은 반드시 2개를 표기해야 한다.



핵심 정리

```
template<typename T> struct remove_pointer
{
    // ...
};

template<typename T> struct remove_pointer<T*>
{
    // ...
};

int main()
{
    remove_pointer< int* >::print();
}
```

primary template 버전을
사용한다면 $T = \text{int}^*$ 로 결정

partial specialization 버전을
사용한다면 $T = \text{int}$ 로 결정



핵심 정리

- specialization, partial specialization 을 만들 때는 "**default parameter** 표기 하지 않는다."
- 표기하지 않아도 "**primary template** 의 **default** 값을 **사용**" 하게 된다.



핵심 정리

- 특수화 또는 부분 특수화 버전만 사용하고 `primary template` 은 사용하지 않으려면
⇒ `primary template` 을 선언만 제공한다.



핵심 정리

- template parameter 의 종류 - 3가지
 - ① type
 - ② template - 이전 강좌 참고
 - ③ **non-type** (타입이 아닌 것들)
- NTTP (Non-Type Template Parameter)

정수형 상수

컴파일 시간 상수만 가능
변수는 안됨

실수형 상수

C++20 부터 지원

enum 상수

enum 또는 enum class

포인터, 함수 포인터

static storage 만 가능
지역변수 주소 안됨.

auto

C++17 부터 가능

`std::conditional`



핵심 정리

- `std::conditional<bool, Type1, Type2>::type`
 - ⇒ 조건에 따라 타입을 선택하는 템플릿
 - ⇒ `<type_traits>` 헤더
 - ⇒ 템플릿의 1번째 인자(`bool`) 에 따라 타입 선택

true	<code>type = Type1;</code>
false	<code>type = Type2;</code>



Template meta programming



● Template Meta Programming

- ⇒ 실행시간이 아닌 컴파일 시간에 연산을 수행하는 코드
- ⇒ 재귀의 종료를 위한 "**template specialization**" 기술 사용
- ⇒ C++11 이 발표되기 전에 사용하던 기술
- ⇒ modern C++(C++11) 이후에는 "**constexpr 함수**" 사용



핵심 정리

- `std::conditional<bool, Type1, Type2>::type`
 - ⇒ 조건에 따라 타입을 선택하는 템플릿
 - ⇒ `<type_traits>` 헤더
 - ⇒ 템플릿의 1번째 인자(`bool`) 에 따라 타입 선택

true	<code>type = Type1;</code>
false	<code>type = Type2;</code>

variable template

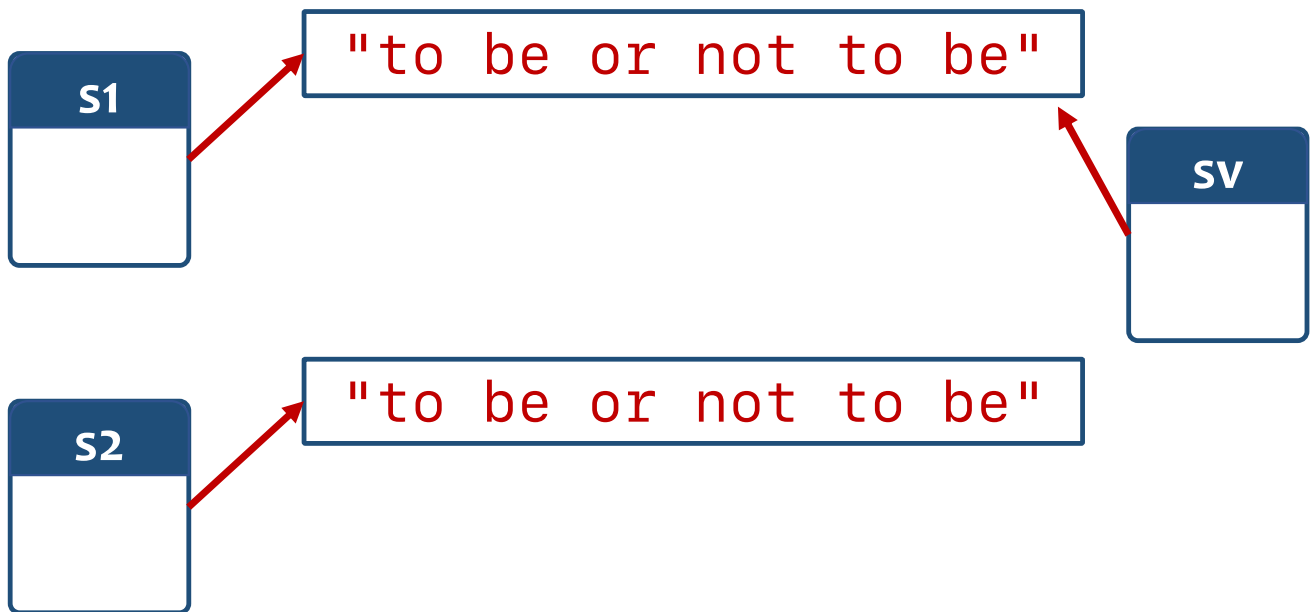


핵심 정리

update 가 자주 발생하는 프로그램을 작성하는데,
“클래스가 추가된 연도를 관리” 하고 싶다.



핵심 정리



- **borrowed range (빌린 범위)**

- ⇒ C++20 에서 소개된 용어

- ⇒ 자원을 소유하지 않고, 다른 `range(container)` 가 소유한 자원을 사용하는 `range`

- ⇒ "**`std::ranges::enable_borrowed_range`**" 라는 variable template 으로 조사 가능.

- `<ranges>`** 헤더

type traits



템플릿 코드 작성시 타입 인자 "T 가 포인터 인 경우와 그렇지 않은 경우 각각 다르게 코드를 작성" 하고 싶다.

- type traits 라이브러리

- ① 타입에 대한 다양한 속성을 조사 하거나
(**query the properties of types**)

- ② 변형(transformation)된 타입을 구할 때 사용
(**Type modifications**)

- ⇒ C++11 에서 표준에 추가됨

- ⇒ **<type_traits>** 헤더



핵심 정리

- T 가 포인터 인지 조사하는 방법

C++11	<code>std::is_pointer<T>::value</code>
-------	--

C++17	<code>std::is_pointer_v<T></code>
-------	---

- T 에서 포인터를 제거한 타입 구하기

C++11	<code>typename std::remove_pointer<T>::type</code>
-------	--

C++14	<code>std::remove_pointer_t<T></code>
-------	---



핵심 정리

- type traits 역사

- ⇒ 1990년대 말 부터 사용되기 시작
- ⇒ boost 라이브러리에서 제공(2000년)
- ⇒ C++11 에서 C++ 표준으로 도입

- 강의에서 다루는 것

- ⇒ 구현 원리 / 주의 사항 / 활용
- ⇒ 초창기 구현 방식 부터 최신 구현 방법까지



핵심 정리

● is_pointer 구현 방법

- ① is_pointer 라는 이름의 구조체 템플릿을 만들고
- ② "enum { value = false }" 를 멤버로 추가
- ③ T* 인 경우의 partial specialization 버전을 만들고 "enum { value = true }" 로 변경
- ④ const T*, volatile T*, const volatile T* 도 필요

함수 인자

is_pointer<T>::value

함수 이름 함수 반환값

● meta function

- ⇒ 컴파일러가 컴파일 시간에 사용하는 함수
- ⇒ 컴파일 시간에 "true/false" 가 결정된다.



핵심 정리

- 왜 enum 을 사용하는가 ?

- ⇒ 구조체 안에서 초기화 코드를 작성하고,
- ⇒ 컴파일 시간에 값을 알 수 있어야 한다.

C++11 이전 스타일

```
template<typename T>
struct is_pointer
{
    enum { value = false };
};
```

C++11 이후 스타일

```
template<typename T>
struct is_pointer
{
    static constexpr bool value = false;
};
```



핵심 정리

- `is_pointer` 구현을 참고해서 아래 2개를 만들어 보세요
 - ⇒ `is_const`
 - ⇒ `is_array`



핵심 정리

- `printf` 함수 템플릿
 - ⇒ 인자로 전달된 변수의 값을 출력하는 함수(디버깅용)
 - ⇒ 인자의 타입이 포인터라면 "**변수 값(메모리 주소)와 메모리에 있는 값(*value)**"도 출력



핵심 정리

template 원형

```
template<typename T>
void printv(const T& value)
{
    if ( is_pointer<T>::value )
        std::cout << value << " : " << *value << std::endl;
    else
        std::cout << value << std::endl;
}
```

T = int* 인 경우

```
template<>
void printv(const int*& value)
{
    if ( true )
        std::cout << value << " : " << *value << std::endl;
    else
        std::cout << value << std::endl;
}
```

OK!

if (false) 이므로 이부분은 실행되지는 않지만
컴파일을 해야 한다.

int 변수를 dereference(*) 할 수 없으므로 에러.

```
template<>
void printv(const int& value)
{
    if ( false )
        std::cout << value << " : " << *value << std::endl;
    else
        std::cout << value << std::endl;
}
```




핵심 정리

- `printf` 함수 템플릿 문제를 해결하는 방법

(1)	<code>std::integral_constant (int2type)</code>	C++11
-----	--	-------

(2)	<code>std::enable_if</code>	C++11
-----	-----------------------------	-------

(3)	<code>if constexpr</code>	C++17
-----	---------------------------	-------

(4)	<code>concept</code>	C++20
-----	----------------------	-------



핵심 정리

T = int, if 사용시

```
template<>
void printv<int>(const int& value)
{
    if ( false )
        std::cout << value << " : " << *value << std::endl;
    else
        std::cout << value << std::endl;
}
```

컴파일 시간에 false 로 결정되었지만 이부분의 코드가 인스턴스화 된 C++ 함수에 포함됨.

● if constexpr

⇒ 조건이 false 인 경우 코드가 인스턴스화 된 함수에 포함되지 않음.

T = int, if constexpr 사용시

```
template<>
void printv<int>(const int& value)
{
    if constexpr ( false )
    else
        std::cout << value << std::endl;
}
```



방법 1.

- 포인터인 경우와 포인터가 아닌 경우를 별도의 함수 템플릿으로 분리.
 - ⇒ "사용되지 않은 함수 템플릿은 인스턴스화 되지 않는다"는 점을 활용
- 하지만,
 - ⇒ if 문 사용시 컴파일 시간에 false 로 결정되어도 "모두 사용되는 것으로 간주해서 "pointer(), not_pointer()" 를 모두 인스턴스화" 한다.
 - ⇒ 실패!



핵심 정리

- 함수 오버로딩 (function overloading)
 - ⇒ 동일한 이름의 함수가 여러 개 있을 때 "어떤 함수를 호출할지 결정하는 것은 컴파일 시간에 결정" 한다.

어떤 함수(템플릿)을 사용할지
컴파일 시간에 결정



```
template<typename T>
void printv_imp(T a, int)
{
}
```

printv_imp(value, 3);

```
template<typename T>
void printv_imp(T a, double)
{
}
```

사용되지 않았으므로 인스턴스화 되지 않는다.



핵심 정리

- 0, 1 은 같은 타입, int 이다.

⇒ fn(0) 과 fn(1) 은 "같은 함수, fn(int)" 호출

int2type<0>

```
template<>
struct int2type<0>
{
    enum { value = 0 };
};
```

int2type<1>

```
template<>
struct int2type<1>
{
    enum { value = 1 };
};
```



다른 타입 !!



핵심 정리

● `int2type`

⇒ 컴파일 시간에 결정된 "**정수형 상수를 타입으로 만드는 템플릿**"

⇒ 77은 값이지만, `int2type<77>` 은 타입이다.

`0, 1`

같은 타입(`int`) 객체

`int2type<0>()`

`int2type<1>()`

다른 타입 객체(임시객체)

`fn(0)`

`fn(1)`

같은 함수 호출. `fn(int)`

`fn(int2type<0>())`

`fn(int2type<1>())`

다른 함수 호출



- **int2type**
 - ⇒ To treat an integral constant as a type at compile-time
 - ⇒ To achieve static call dispatch based on constant integral values
 - ⇒ 2000년대 초반 Andrei Alexandrescu 에 의해 소개됨
 - ⇒ C++11 을 만들 때 "**integral_constant**" 로 발전됨.



핵심 정리

- `int2type`
 - ⇒ "**int**" 타입의 상수를 타입으로 만드는 도구
 - ⇒ `int` 뿐 아니라 다른 "**정수형 타입의 상수(bool, char, short, long 등)도 타입**"으로 만들면 좋지 않을까 ?
- `std::integral_constant`
 - ⇒ C++11 표준
 - ⇒ `int2type` 의 발전된 형태

<code>int2type</code>	int 타입 상수만 타입화
<code>integral_constant</code>	"모든 정수형 타입의 상수"를 타입화



핵심 정리

- `std::true_type`, `std::false_type`
⇒ `true` 와 `false` 를 가지고 만든 타입

`true`
`false`

참 거짓을 나타내는 값
같은 타입(`bool`)

`std::true_type`
`std::false_type`

참 거짓을 나타내는 타입
다른 타입



핵심 정리

true_type

```
template<>
struct integral_constant<bool, true>
{
    static constexpr bool value = true;
};
```

false_type

```
template<>
struct integral_constant<bool, true>
{
    static constexpr bool value = false;
};
```

T가 포인터인 경우

T가 포인터가 아닌 경우

`is_pointer<T>`

- C++ 표준의 `std::is_pointer` 구현 원리
 - ⇒ T의 포인터 여부에 따라 "**`std::true_type`** 또는 **`std::false_type`**으로 부터 상속"



핵심 정리

1. class(struct) template 을 먼저 만들고

```
template<typename T> struct is_pointer : std::false_type {};  
template<typename T> struct is_pointer<T*> : std::true_type {};  
template<typename T> struct is_pointer<T* const> : std::true_type {};  
template<typename T> struct is_pointer<T* volatile> : std::true_type {};  
template<typename T> struct is_pointer<T* const volatile>  
: std::true_type {};
```

```
template<typename T>  
constexpr bool is_pointer_v = std::is_pointer<T>::value;
```

2. struct template 을 사용해서 variable template 구현



핵심 정리

- T 가 포인터 인지 조사하는 방법

C++11	<code>std::is_pointer<T>::value</code>
-------	--

C++17	<code>std::is_pointer_v<T></code>
-------	---

- T의 포인터 여부에 따라 다른 구현을 작성하려면

(1)	<code>if constexpr</code> 로 조사후 작성	C++17
-----	------------------------------------	-------

(2)	<code>std::true_type</code> , <code>std::false_type</code> 으로 함수 오버로딩 사용	C++11
-----	---	-------

(3)	<code>std::enable_if</code>	C++11
-----	-----------------------------	-------

(4)	<code>concept</code>	C++20
-----	----------------------	-------



- `type traits` 라이브러리

- ① 타입에 대한 다양한 속성을 조사 하거나
(`query the properties of types`)
- ② 변형(`transformation`)된 타입을 구할 때 사용
(`Type modifications`)



핵심 정리

- `remove_pointer` 구현 방법

- ① `remove_pointer` 라는 이름의 구조체 템플릿을 만들고
- ② "`using type = T`" 멤버로 추가
- ③ 원하는 타입(포인터를 제거한 타입)을 얻을 수 있도록 부분 특수화 버전을 제공.



핵심 정리

- T 가 포인터 인지 조사하는 방법

C++11	<code>std::is_pointer<T>::value</code>
C++17	<code>std::is_pointer_v<T></code>

```
template<typename T>  
constexpr bool is_pointer_v = std::is_pointer<T>::value;
```

variable template

- T 에서 포인터를 제거한 타입 구하기

C++11	<code>typename std::remove_pointer<T>::type</code>
C++14	<code>std::remove_pointer_t<T></code>

```
template<typename T>  
using remove_pointer_t = typename std::remove_pointer<T>::type;
```

using template

variadic template



핵심 정리

- 가변인자 템플릿

- ⇒ C++11 부터 지원

- ⇒ 템플릿 파라미터에 "... 을 사용하는 기술"

- ⇒ 템플릿 사용시 "템플릿의 타입 인자의 갯수에 제한이 없다."

T1, T2	template parameter
Ts	template parameter pack

Template

```
template<typename...Ts>
class tuple
{
};
```

```
class tuple<>
{
};
```

```
class tuple<int>
{
};
```

```
class tuple<int, double>
{
};
```

▶ cppinsights.io



핵심 정리

- 가변 인자 함수 템플릿

Template

```
template<typename...Ts>  
void f2(Ts ... args)  
{  
}
```

```
void f2(int)  
{  
}
```

```
f2(3);
```

```
f2(3, 3.4, 'A');
```

```
void f2(int, double, char)  
{  
}
```

```
void f3(...)  
{  
}
```

```
f3(3);
```

```
f3(3, 3.4, 'A');
```



핵심 정리

● Parameter Pack

Ts	int, double, char	template parameter pack
args	1, 3.4, 'A'	function parameter pack

● sizeof...

⇒ parameter pack 안에 있는 요소의 갯수를 구하는 연산자

⇒ 주의! 반드시 () 를 사용해야 한다.

sizeof	sizeof(n)	ok
	sizeof n	ok
sizeof...	sizeof...(args)	ok
	sizeof... args	error



핵심 정리

- Pack Expansion

- ⇒ parameter pack 안에 있는 "모든 요소를 콤마(,)를 사용해서 순서대로 나열".
- ⇒ pack 이름 뿐 아니라 "pack 이름을 사용하는 패턴" 에도 사용가능.

"pack 이름을 사용하는 패턴..."

➔ 패턴 **e1**, 패턴 **e2**, 패턴 **e3** ...



핵심 정리

- Pack Expansion loci

- ⇒ 모든 문맥(Context) 에서 pack expansion 이 가능한 것은 아님.
- ⇒ "함수 인자를 전달하는 괄호 안" 또는 "{ }로 초기화 되는 문맥" 에서 pack expansion 이 가능.



- Pack Expansion 이 발생하는 Context

- ① Brace-enclosed initializers
- ② Function argument lists
- ③ Parenthesized initializers
- ④ Template argument lists
- ⑤ Lambda captures
- ⑥ Template parameter list
- ⑦ Base specifiers and member initializer lists

make tuple



- **std::tuple**

- ⇒ 임의 갯수의 "서로 다른 타입의 객체를 저장"할 수 있는 타입.
- ⇒ C++11 추가

- **std::get**

- ⇒ std::tuple 에 있는 각 요소에 접근할 때 사용하는 함수 템플릿.



핵심 정리

- **step 1. 가변인자 템플릿 사용**

- ⇒ tuple 을 가변인자 클래스 템플릿으로 제공.

- ⇒ 저장하는 요소의 갯수를 관리하는 static 멤버 "N" 제공



핵심 정리

- **step 2. 값 한 개 보관하기**
 - ⇒ 템플릿 인자가 한 개 이상 있는 경우를 위한
"**partial specialization**" 제공
 - ⇒ 생성자 추가



핵심 정리

- step 3. N개 값 보관하기

`tuple<>`

이 경우는 `primary template`을 사용. 기반 클래스 없음.



`tuple<char>`
`char value = 'A';`



`tuple<double, char>`
`double value = 3.4;`



`tuple<int, double, char>`
`int value = 3;`

`tuple<int, double, char>`
의 기반 클래스는
`tuple<double, char>`



핵심 정리

- **step 4. 생성자 변경**

- ⇒ 자신은 한 개를 보관하고, 나머지는 기반 클래스에 전달
- ⇒ `std::move` 를 지원하기 위해서는 "**forwarding reference**" 사용



Fold Expression



핵심 정리

● Fold Expression (C++17)

⇒ Parameter pack 안의 "모든 요소에 대해서 이항 연산을 수행 하는 표현식"

⇒ 4 가지 형태로 제공

unary right fold	$(\textit{pack op} \dots)$
	$(\textcolor{red}{args} + \dots) \rightarrow (1 + (2 + (3 + 4)))$
unary left fold	$(\dots \textit{op pack})$
	$(\textcolor{red}{args} + \dots) \rightarrow (1 + (2 + (3 + 4)))$
binary right fold	$(\textit{pack op} \dots \textit{op init})$
	$(args + \dots + \textcolor{red}{0}) \rightarrow (1 + (2 + (3 + (4 + \textcolor{red}{0}))))$
binary left fold	$(\textit{init op} \dots \textit{op pack})$
	$(\textcolor{red}{args} + \dots) \rightarrow (1 + (2 + (3 + 4)))$

C RTP



핵심 정리

- CRTP(Curiously Recurring Template Pattern)
 - ⇒ "기반 클래스에서 파생 클래스의 클래스 이름을 사용"할 수 있게 하는 기술
- CRTP 핵심
 - ① "기반 클래스를 템플릿으로" 만들고
 - ② 파생 클래스를 만들 때 "자신의 클래스 이름을 기반 클래스의 템플릿 인자로 전달".



핵심 정리

- GUI event 를 가상함수 기반으로 처리하는 경우
 - ⇒ GUI event 는 아주 많은 종류가 있다.
 - ⇒ "가상함수 테이블의 크기에 대한 메모리 오버헤드"가 있다.
- CRTP 를 사용한 event 처리
 - ⇒ "가상 함수가 아닌 함수를 가상함수 처럼 동작"



핵심 정리

- CRTP 를 사용하는 경우 주의 사항
 - ⇒ 기반 클래스가 템플릿이므로 파생 클래스의 갯수가 많아
지만 "**Code Bloat 현상**"이 있을 수 있다.
- Thin Template
 - ⇒ "**To reduce object code duplication**" when a
class template is instantiated for many
types.
 - ⇒ 클래스 템플릿을 만들 때 "**템플릿 인자를 사용하지 않은
멤버는 기반 클래스(템플릿이 아닌)를 만들어 제공**".

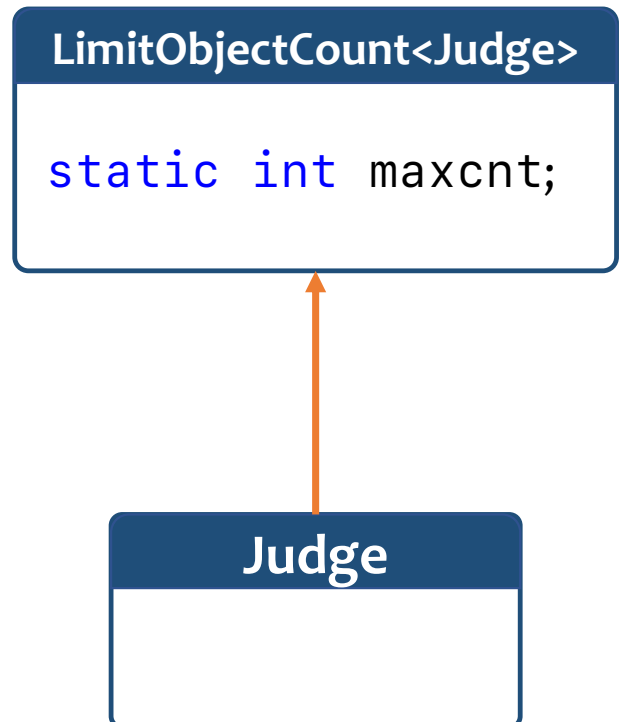
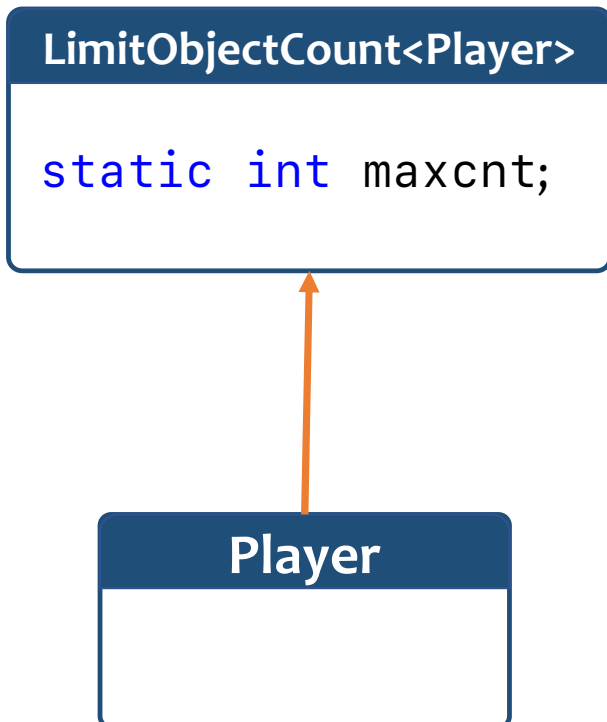
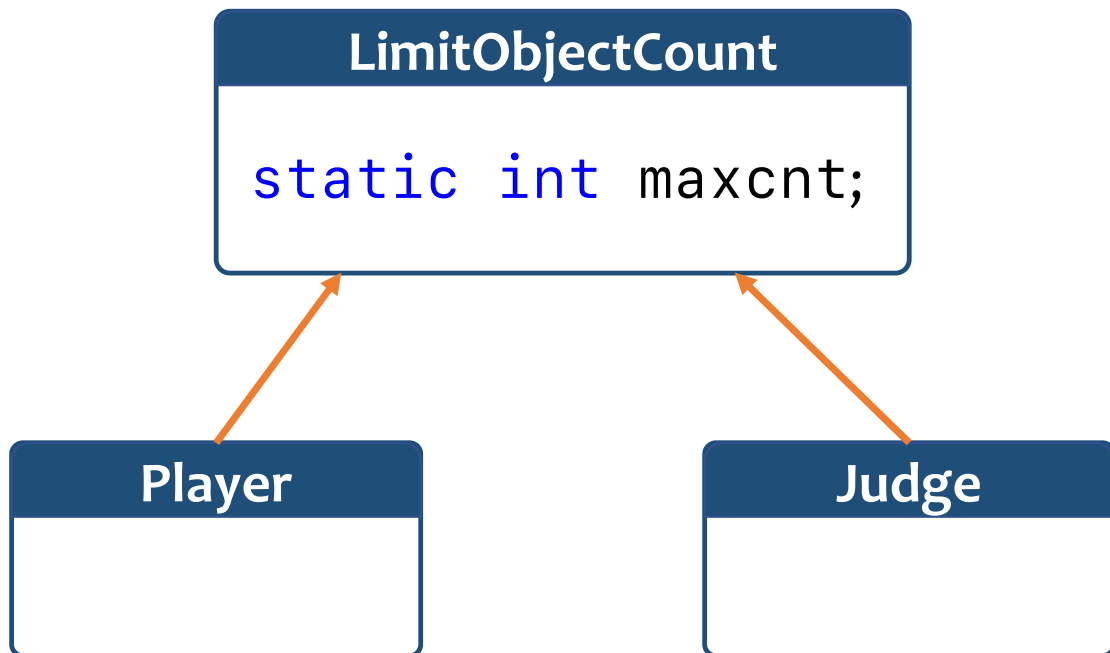


핵심 정리

- `std::view_interface`
 - ⇒ C++20 에서 추가된 "**range library**" 가 제공하는 클래스 템플릿
 - ⇒ CRTP 기술로 구현.
 - ⇒ 파생 클래스의 `begin()`, `end()` 에 의존해서 다양한 멤버함수를 구현



핵심 정리



SFINAE



핵심 정리

- Overloading Resolution

⇒ 동일한 이름의 함수가 여러 개 있을 때 "함수 찾는 순서"

① exactly matching

② template

③ ...



핵심 정리

overloading resolution 에 따라 "함수 템플릿을 사용하기로 결정"이 되었는데, 템플릿 인자 T의 타입을 결정하고 "인스턴스화를 할 때 실패" 했다면 ?

① 컴파일 에러가 발생할 것이다.

② **V** 에러는 아니고, 함수 생성에 실패 했으므로, 동일이름의 함수인 "**fn(...)**" 이 사용한다.

치환 실패는 에러가 아니다.

Subscription **F**ailure **I**s **N**ot **A**n **E**rror

호출 가능한 "후보 군에서 제외" 되고,
"동일 이름의 다른 함수가 있으면 사용" 된다.

● SFINAE 활용 기술

⇒ enable_if

⇒ member detect

⇒ 그 외에 다양한 기술에 활용

C++20 부터

Concept 기술로 대체.



핵심 정리

- SFINAE 가 적용되는 3가지 경우
 - ⇒ function return type
 - ⇒ function parameter
 - ⇒ template parameter