템플릿 메타 프로그래밍에 대한 이해

2024. 09. 26

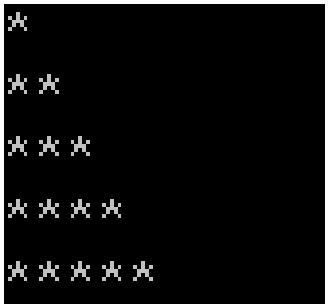
지능형 선박연구본부 연성모 책임연구원

CONTENTS

개요

- 별을 출력하는 c 프로그램
  - 인라인 코드

```
printf(“*\n”);
Printf(“**\n”);
Printf(“***\n”);
Printf(“****\n”);
printf(“*****\n”);
```



  - 루프문 사용

```
for(auto i=0; i<5; i++)
{
   for(auto j=0; j<=i; j++)
   {
      printf(“*”);
   }
   printf(“\n”);
}
```

개요

- 코드 이곳저곳에서 별을 출력해야 하는 경우
  - 인라인 코드 블럭을 코드 내에 복사/붙여넣거나
  - 루프문 코드 블록을 코드 내에 복사/붙여넣거나
    - 만일 코드 블럭에 에러가 있는 경우?
      - 답이 없음

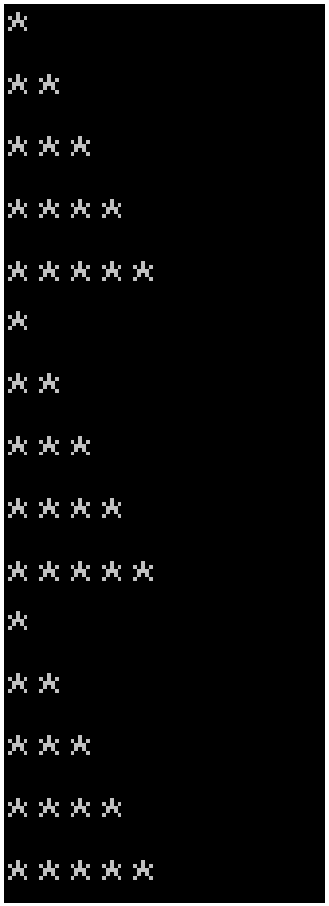
- 별의 크기/모양을 상황에 따라 바꾸고 싶은 경우?
  - 알고리즘은 동일한데? → 코드 중복 → 중복 코드 자동생성

함수, 객체 매크로, 템플릿

구조적 프로그래밍 패러다임
(procedural, structural programming)

data와 data manipulation을 통합

객체지향 프로그래밍 패러다임
(Object Oriented Programming)



개요

- 함수형 프로그래밍
  - 값을 다루는 알고리즘의 **일반화**

```
void star(int max_num, char shape)
{
   for(auto i=0; i<max_num; i++)
   {
      for(auto j=0; j<=i; j++)
      {
         printf("%c", shape);
      }
      printf("\n");
   }
}
```

생각의 확장

- 값이 아닌 자료형에 대한 **일반화**?

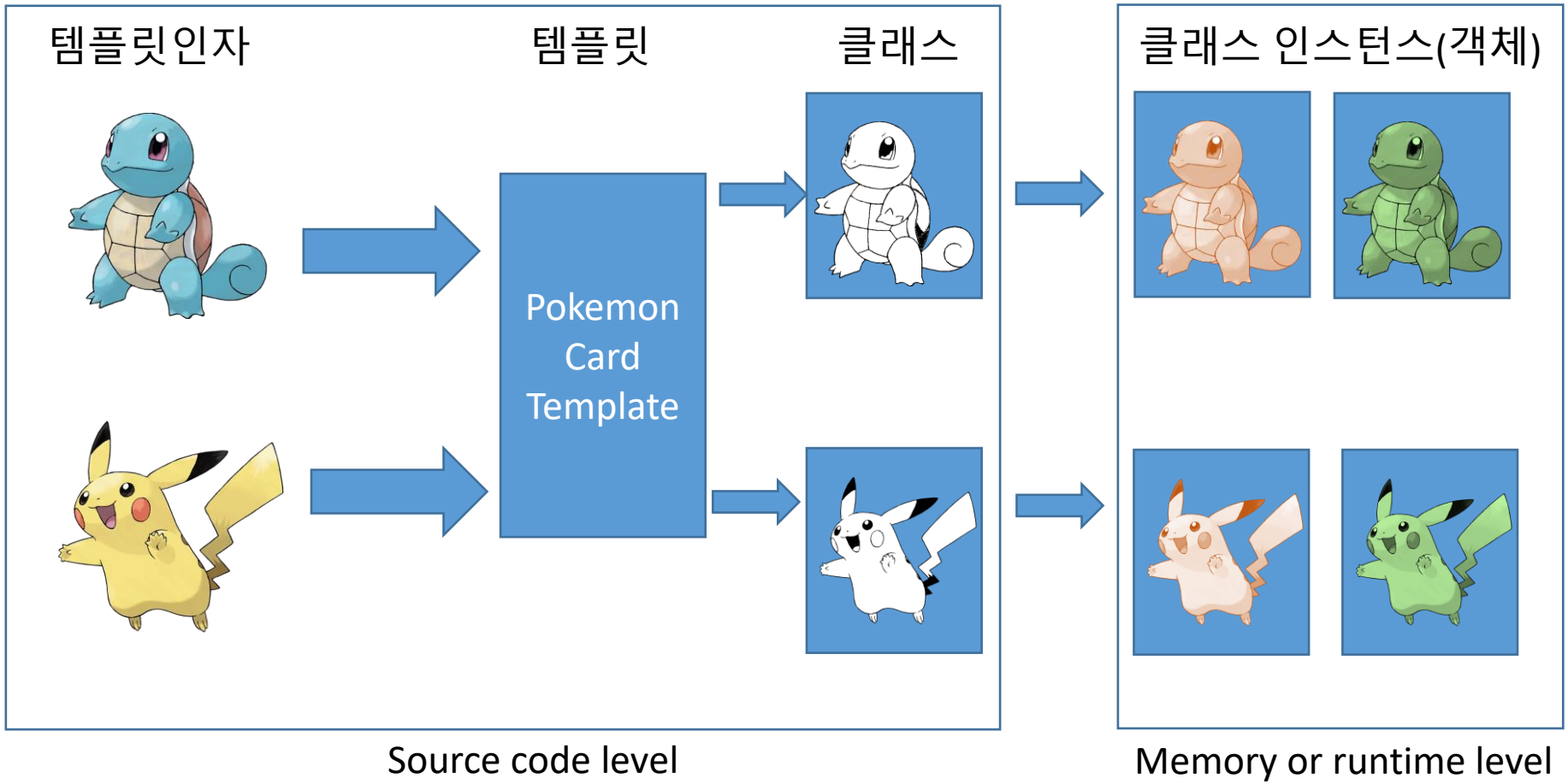
개요

- 템플릿(Template)이란
  - 사전적 의미: 어떤 것을 만들 때 안내 역할을 하는 틀을 의미
  - C++ 용어: **자료형**을 입력으로 하여 동일한 알고리즘을 사용하는 코드(클래스/함수)를 생성하도록 하는 도구

- 템플릿 프로그래밍(Template Programming)이란
  - 템플릿을 활용한 프로그래밍 또는 **일반화 프로그래밍**(Generic programming)을 의미
    - 템플릿 == Generics
      - Java, .Net, Rust 계열에서는 템플릿 대신 **Generics** 이라는 용어를 사용함

개요

- 클래스(class)와 템플릿(template)의 차이
  - 클래스는 특정 자료형에 대한 기술 (사용자 정의 자료형)
    - 내부 멤버의 자료형은 고정
  - 템플릿은 클래스 표현에서 주요 자료형을 외부 입력으로 처리
    - 일종의 **클래스/함수 생성기**



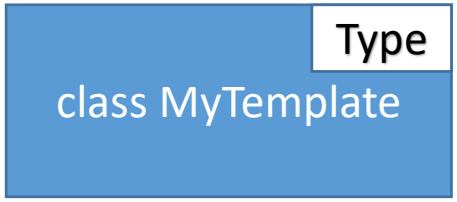
템플릿 문법 개요

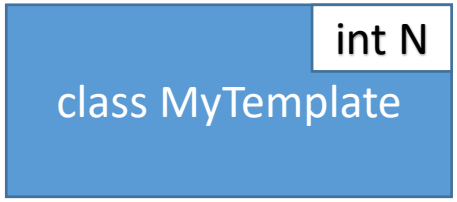
- 템플릿(Template) 구성
  - 템플릿 인자
  - 템플릿 바디 (class 또는 함수)

```
template<typename Type>
class MyTemplate
{
    ...
};
```

  - 템플릿 인자를 매개변수로 하는 일종의 함수로 볼 수 있음

- 템플릿 인자 특징
  - **원시자료형과 구조체/클래스 등을 입력**으로 사용할 수 있음
    - 자료형의 제한 또는 선별을 할 수 없음
    - 예) 숫자만 템플릿 인자로 강제하는 경우
  - 템플릿 인자에 대해 별도로 유효성 검사를 하지 않음
  - 원시자료형의 경우 계산이 가능함





템플릿 문법 개요

- 템플릿 클래스 생성자/소멸자 선언
  - 방법 1
    - 기존 클래스 생성자/소멸자 선언과 동일

```
DiagonalMatrix();
```

```
template<class Form>
DiagonalMatrix(const Matrix<Form, Type>&);
```

  - 방법 2
    - 생성자/소멸자에 템플릿 인자를 붙여 선언

```
DiagonalMatrix<Type>();
```

```
template<class Form>
DiagonalMatrix<Type>(const Matrix<Form, Type>&);
```

C++20 표준부터 지원하지 않는 선언 형식

템플릿 문법 개요

- 템플릿 인자 매칭 규칙
  - SFINAE (Substitution Failure Is Not An Error)
    - Exactly matching
    - Template
    - Variable argument

함수의 경우 반환타입은 구분하지 않음

```
template<typename T>
void foo(T t) { cout << "T" << endl; }

void foo(int n) { cout << "int" << endl; }

void foo(...) { cout << "..." << endl; }

int main()
{
    foo(7);
}
```

출처: https://blog.naver.com/hikari1224/221485187168

- 재귀적 구조가 가능
  - 템플릿 내에서 동일한 템플릿 호출
    - 중복 호출이 가능한 구조

템플릿 메타 프로그래밍

- 템플릿 메타 프로그래밍(Template Meta Programming, TMP)이란
  - 템플릿에 프로그래밍적 기능(분기, 루프, 계산)등의 기능을 부여한 템플릿 프로그래밍을 의미
    - 일종의 조건부 컴파일로 볼 수 있음
      - 템플릿 인자에 대한 유효성 검사
    - 컴파일 시간에 일부 코드를 계산하여 인라인시킬 수 있음
      - 런타임 실행 속도의 향상

템플릿 메타 프로그래밍

- 템플릿 메타 프로그래밍은 의도적이었나?
  - 1994, Erwine Unruh의 보고서에 나타난 소수(Prime number) 계산이 시작
    - 우연한 발견
  - C++ 문법이 아님
    - 템플릿 동작 방식을 역이용하여 기본적인 분기, 루프 같은 동작이 가능하게 만듦
- 템플릿 메타 프로그래밍을 왜 사용하는가?
  - 템플릿 인자를 선별하고자 할 때 (trait)
  - 컴파일 시점에 많은 것을 결정하도록 하여 실행 시점의 계산을 줄이는 것이 목적
- 그럼 왜 일반적으로 잘 안 알려져 있는가?
  - C++ 문법 사항이 아님
  - 재귀적 구조를 활용하므로 루프 구현이 직관적이지 않음
  - 디버그가 어려움
  - 최종 바이너리의 크기가 커짐

문법

- TMP와 일반 프로그래밍과의 차이
  - 자료형에 대한 연산 vs 값에 대한 연산
- 원리
  - 컴파일 시간에 값을 특정 지을 수 있어야 함
  - 클래스 변수이면서 리터럴 상수인 static const 자료형을 사용해야 함

문법	일반 C++ 구문	TMP 구문
변수 선언	<type> <name>;	static const <type> <name>;
분기문	if-else 구문	SFINAE를 활용한 템플릿 특수화
자기 참조	this	class 이름에 대한 typedef 또는 using을 활용한 별칭
함수 반환	return	typedef 또는 using을 활용한 별칭
반복문	for	재귀문

기본 예

- 팩토리얼 함수
  - 열거형: N! = N(N-1)(N-2) … 1
  - 재귀형: N! = N(N-1)!

일반 함수형

```
int factorial(int n)
{
    if (n==1)
    {
        return 1;
    }
    else
    {
        return n* factorial(n - 1);
    }
}

int main()
{
    std::cout << factorial(5) << std::endl;
}
```

TMP 형

```
template<int N>
class Factorial
{
public:
    static const int value = N * Factorial<N - 1>::value;
};

template<>
class Factorial<1>
{
public:
    static const int value = 1;
};

int main()
{
    std::cout << Factorial<5>::value << std::endl;
}
```

TMP를 적용한 OpenFOAM 자료형

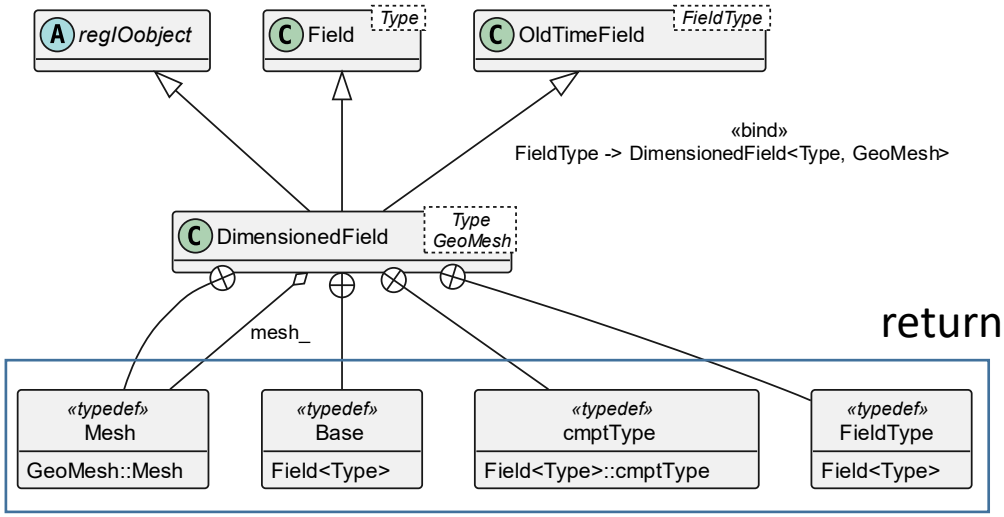
- 필드 자료형
  - DimensionedField
  - GeometryField

- 벡터공간 자료형
  - VectorSpace
  - MatrixSpace

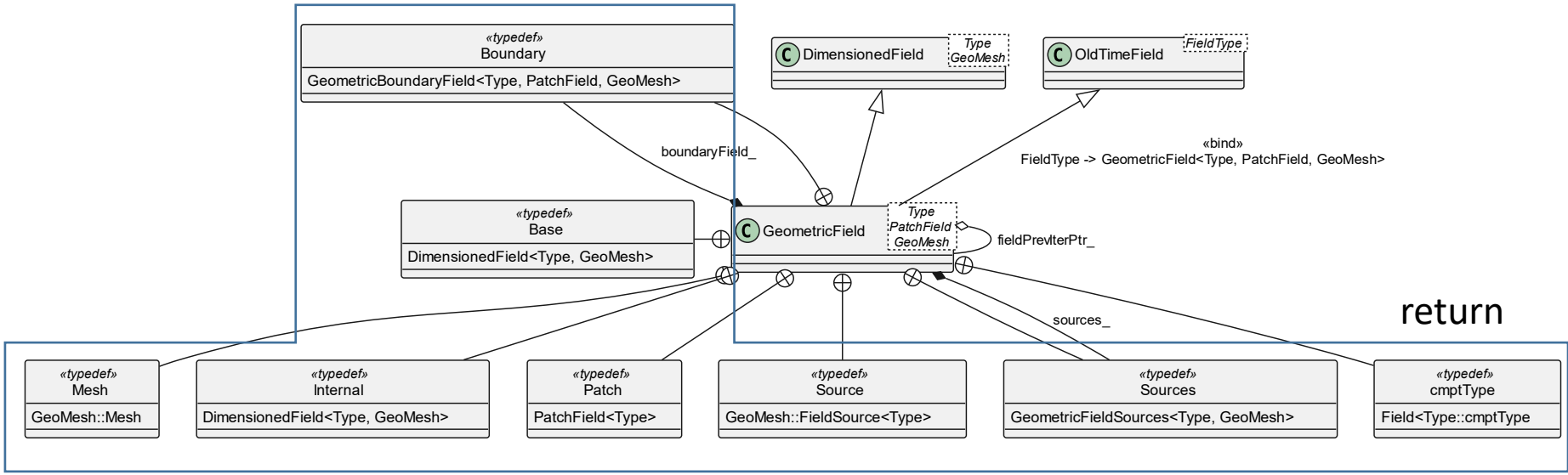
- OpenFOAM의 가장 기본 자료구조로 활용됨
  - VectorSpace
    - Vector
    - Tensor
  - GeometryField
    - Volume 필드
      - volVectorField
      - volScalarField
    - Surface 필드
      - surfaceVectorField
      - surfaceScalarField

OpenFOAM에 사용된 TMP

- DimensionedField

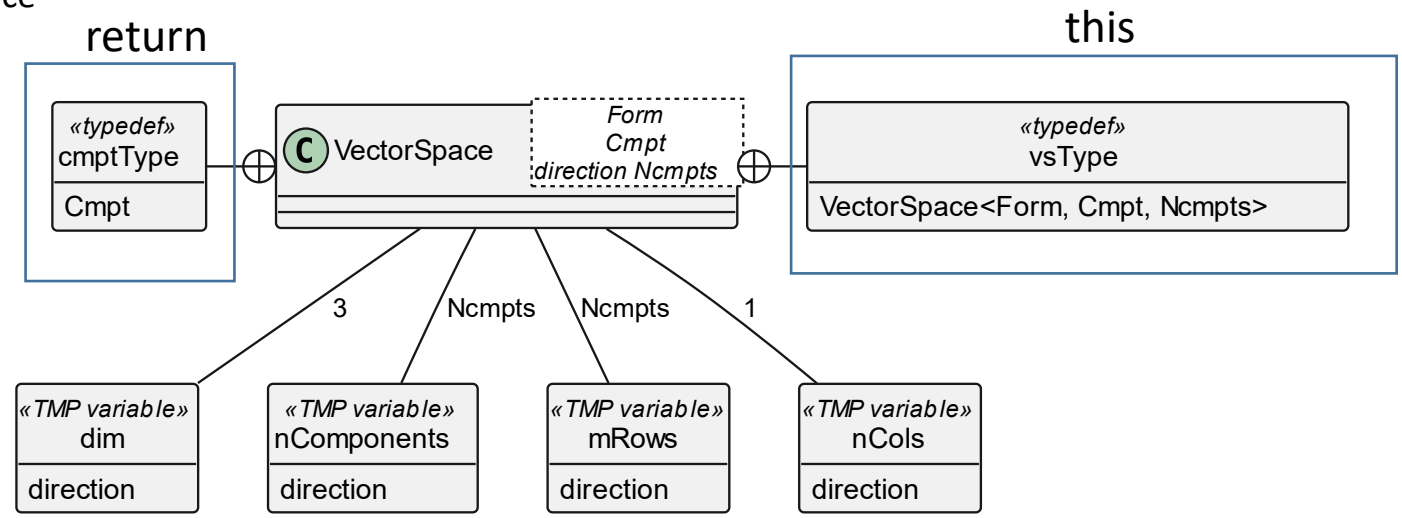


- GeometricField

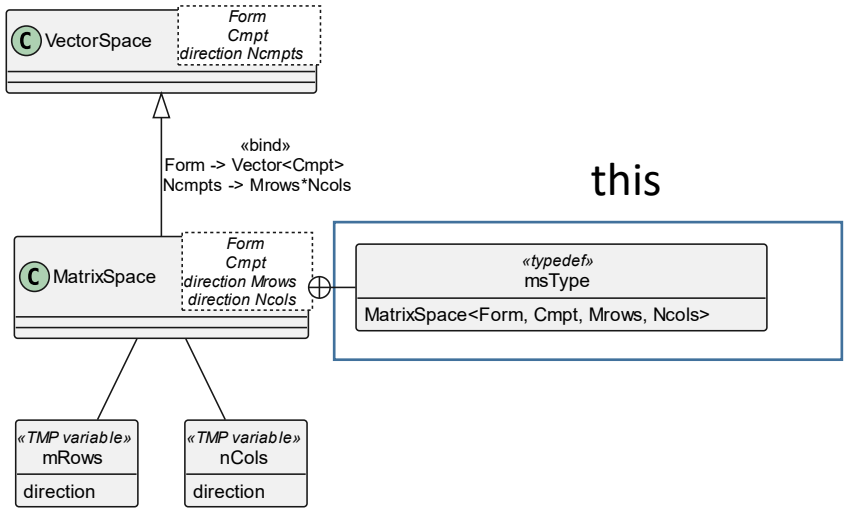


OpenFOAM에 사용된 TMP

- VectorSpace

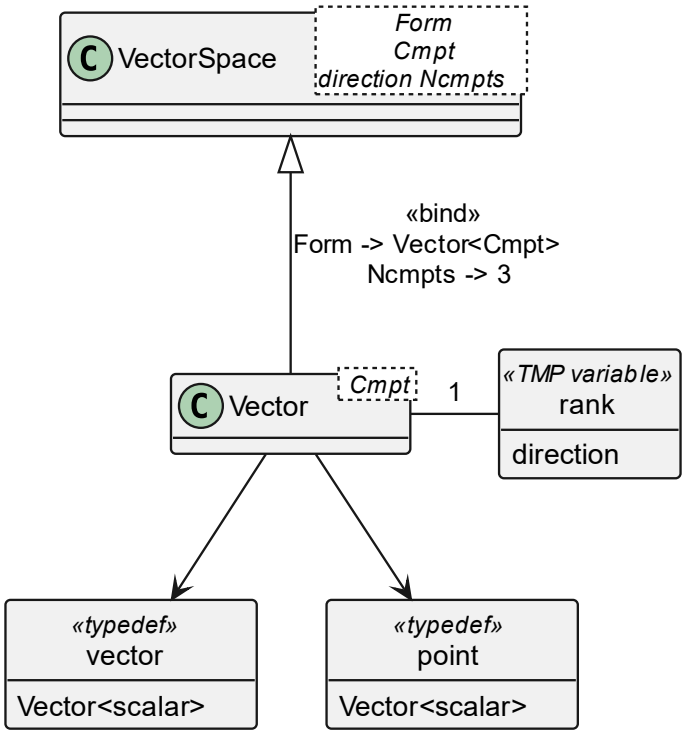


- MatrixSpace

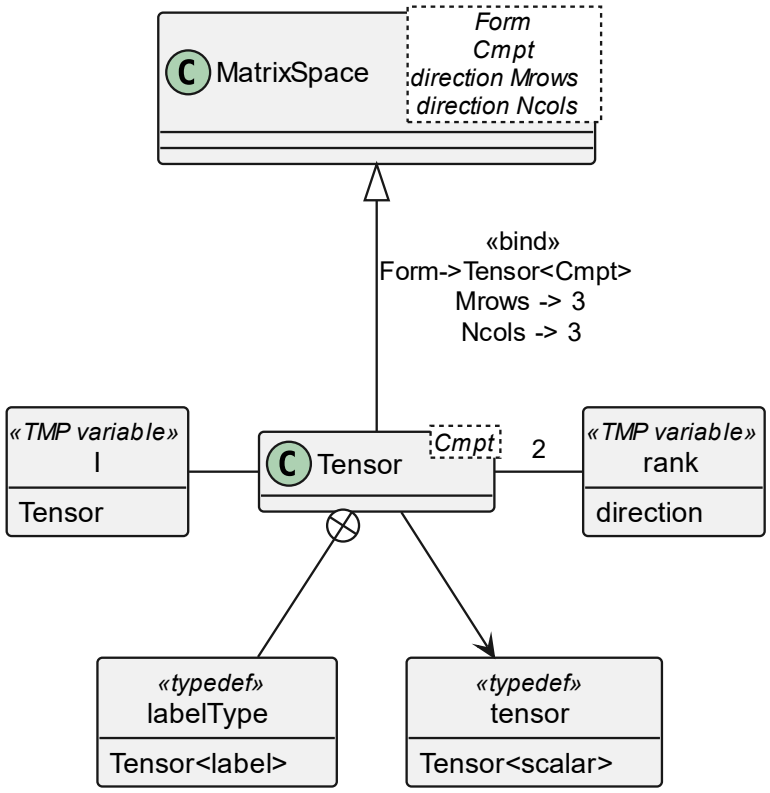


OpenFOAM에 사용된 TMP

- Vector



- Tensor



루프가 사용된 TMP

- VectorSpaceOps.H
  - 다양한 대입 연산(EqOp)에 대한 연산 정의
    - 연산자를 입력으로 하여 일반화함
    - +=, -=, /= 등
  - 원소(element) 단위 연산

루프가 사용된 TMP

- EqOp의 정체
  - 함수 객체
    - src/OpenFOAM/primitives/ops/ops.H 에 정의됨
      - 함수 객체를 생성하는 매크로 정의
        - EqOp, Op, weightedOp

```
#define EqOp(opName, op)
    template<class T1, class T2>
    class opName##Op2
    {
    public:

        void operator()(T1& x, const T2& y) const
        {
            op;
        }
    };

    template<class T>
    class opName##Op
    {
    public:

        void operator()(T& x, const T& y) const
        {
            op;
        }
    };
```

함수객체1	함수객체2	연산
eqOp	eqOp2	x = y
plusEqOp	plusEqOp2	x += y
minusEqOp	minusEqOp2	x -= y
multiplyEqOp	multiplyEqOp2	x *= y
divideEqOp	divideEqOp2	x /= y
eqSqrOp	eqSqrOp2	x = sqr(y)
eqMagOp	eqMagOp2	x = mag(y)
plusEqMagSqrOp	plusEqMagSqrOp2	x += magSqr(y)
maxEqOp	maxEqOp2	x = max(x, y)
minEqOp	minEqOp2	x = min(x, y)
minMagSqrEqOp	minMagSqrEqOp2	x = (magSqr(x)<=magSqr(y) ? x : y)
maxMagSqrEqOp	maxMagSqrEqOp2	x = (magSqr(x)>=magSqr(y) ? x : y)
andEqOp	andEqOp2	x = (x Op&Op& y)
orEqOp	orEqOp2	x = (x ¦¦ y)
notEqOp	notEqOp2	x = (x != y)
eqMinusOp	eqMinusOp2	x = -y
nopEqOp	nopEqOp2	(void)x

루프가 사용된 TMP

일반조건

```
template<direction N, direction I>
class VectorSpaceOps
{
public:

    template<class V, class S, class EqOp>
    static inline void eqOpS(V& vs, const S& s, EqOp eo)
    {
        eo(vs.v_[I], s);
        VectorSpaceOps<N, I + 1>::eqOpS(vs, s, eo);
    }
}
```

경계조건

```
template<direction N>
class VectorSpaceOps<N, N>
{
public:

    template<class V, class S, class EqOp>
    static inline void eqOpS(V&, const S&, EqOp)
    {}
```

사용예

```
Inline VectorSpace<Form, Cmpt, Ncmpts>::VectorSpace(const
Foam::zero)
{
    VectorSpaceOps<Ncmpts,0>::eqOpS(*this, Zero, eqOp<Cmpt>());
}
```

```
inline void VectorSpace<Form, Cmpt, Ncmpts>::operator+=
(
    const VectorSpace<Form, Cmpt, Ncmpts>& vs
)
{
    VectorSpaceOps<Ncmpts,0>::eqOp(*this, vs, plusEqOp<Cmpt>());
}
```

템플릿 인자를 구분하는 TMP

- Trait 객체
  - 자료형의 정보를 담고 있는 객체
  - 템플릿 인자를 구분하기 위한 용도로 사용

- OpenFOAM에서 사용된 곳
  - products.H
    - 벡터공간의 product 공간 연산을 담당
    - typeOfRank는 Scalar, Vector, Tensor 자료형에 정의되어 있음
      - return 값으로 type 별칭을 가짐

```
template<class arg1, class arg2>
class innerProduct
{
public:
    typedef typename typeOfRank
    <
        typename pTraits<arg1>::cmptType,
        direction(pTraits<arg1>::rank) + direction(pTraits<arg2>::rank) - 2
    >::type type;
};
```

Inner product의 결과(자료형)을 알려줌

Scalar

```
template<class Cmpt>
class typeOfRank<Cmpt, 0>
{
public:

    typedef Cmpt type;
};
```

Vector

```
template<class Cmpt>
class typeOfRank<Cmpt, 1>
{
public:

    typedef Vector<Cmpt> type;
};
```

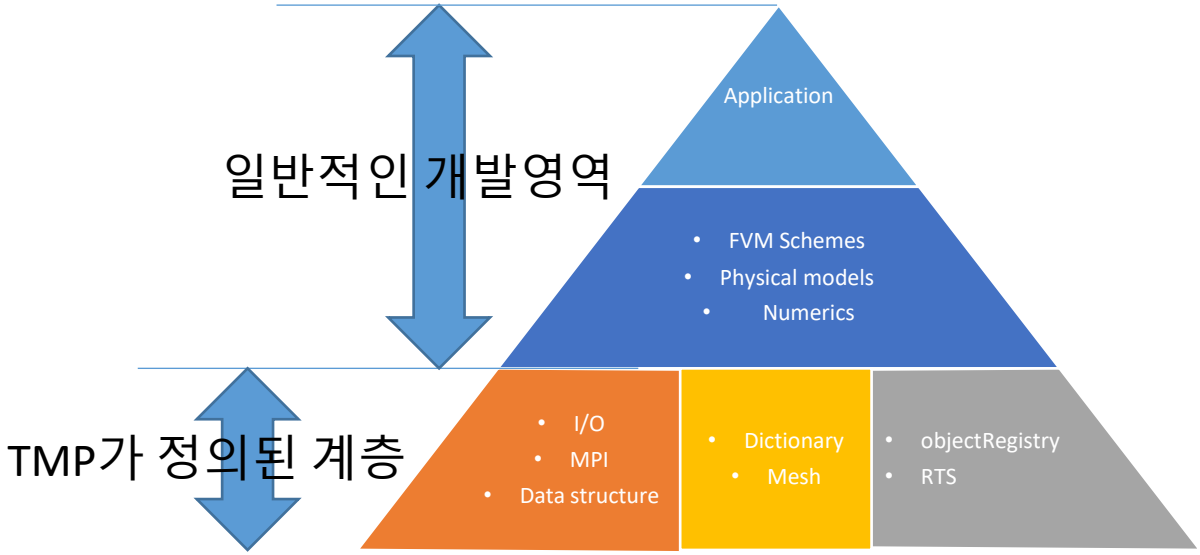
Tensor

```
template<class Cmpt>
class typeOfRank<Cmpt, 2>
{
public:

    typedef Tensor<Cmpt> type;
};
```

그럼

- OpenFOAM 개발하는데 TMP를 알아야 할까?
  - GeometricField, VectorSpace는 모두 OpenFOAM core 자료형
    - 굳이 몰라도 개발하는데 큰 지장 없음
  - 깊이 있는 이해를 원하는 경우
    - 왜, OpenFOAM 컴파일 시간은 그렇게 오래 걸리는가?
    - C++의 속도 병목을 어떻게 해결했는가?
    - 벡터공간을 코드로 어떻게 구현할 수 있는가?



결론

- TMP는 C++ 템플릿 문법의 의도치 않은 기능의 발견으로 시작
  - C++ 표준 문법에서 지원하지 않음

- Modern C++ 표준에서 TMP 지원 기능 도입
  - C++11
    - constexpr ➔ static const에 대비되는 키워드
  - C++20
    - concept와 require 키워드 도입
      - 템플릿 인자를 선별할 수 있는 기능
        - 특정 멤버함수 정의가 있는 자료형만 허용
        - 특정 연산자가 정의되어 있는 자료형만 허용
    - module 기능 도입
      - 특정 헤더에서 필요한 함수 또는 자료형만 선별적 사용
      - 컴파일 시간 절약
  - C++23
    - Std module library 지원 (2024년 기준, 아직 구현된 컴파일러 없음 - visual studio가 부분적 지원)

- 현재 OpenFOAM 작성 표준: C++14
  - Module 기능이 안정화되면 서서히 적용할 것으로 예상됨

Thank You

Korea Research Institute of Ships & Ocean Engineering

선박해양플랜트연구소
KOREA RESEARCH INSTITUTE OF SHIPS & OCEAN ENGINEERING

대전광역시 유성구 유성대로 1312번길 32 선박해양플랜트연구소
Tel. 042-866-3114 / Fax. 000-000-0000 / E-mail. kriso@kriso.re.kr / www.kriso.re.kr