OpenFoamのためのC/C++

第3回　OpenFoamで勉強るテンプレート

田中昭雄

目的

この勉強会の資料があれば、
OpenFoamカスタマイズ時にC/C++で迷わない

予定

- 第1回　メモリ管理
- 第2回　CFDの例で勉強するクラス
- 第3回　OpenFOAMで勉強するテンプレート
- 第4回　OpenFOAMカスタマイズ
- 第5回　未定
- 第6回　未定

今回のテーマ

テンプレート機能を使えるようになる

今回の前提

- C言語で
  - 配列を使ったことがある
  - 構造体を使ったことがある
  - 関数を使ったことがある
  - includeファイルを使ったことがある

- クラスという言葉を聞いたことがある

- C++ベースで解説していきます

Agenda

- テンプレート概要
- テンプレート関数
- テンプレートクラス
- OpenFoamのVector

Agenda

- テンプレート概要
- テンプレート関数
- テンプレートクラス
- OpenFoamのVector

テンプレートとは

型指定を利用側で定義することで、コード重複を防ぐ（型に対して汎用的なコード記述が可能）

テンプレートの活用例

平均計算関数

```
template<typename T>
T average(int n, T* seq)
{
  T sum = 0;
  for(int i = 0; i < n; ++i)
  {
    sum += seq[i];
  }
  return sum / n;
}
```

利用方法

```
int main()
{
  int n = 3;
  double[] seq = {0.1, 2.5, 3.2};
  std::cout  << average<double>(n, seq) << “¥n”;
  return 0;
}
```

型違い同じ機能を1つの定義で実現

テンプレートとは

型違い同じ機能は酷似したコードになりやすい

テンプレートの活用しない場合

float型の平均計算関数

```
float average(int n, float* seq)
{
 float sum = 0;
  for(int i = 0; i < n; ++i)
  {
    sum += seq[i];
  }
  return sum / n;
}
```

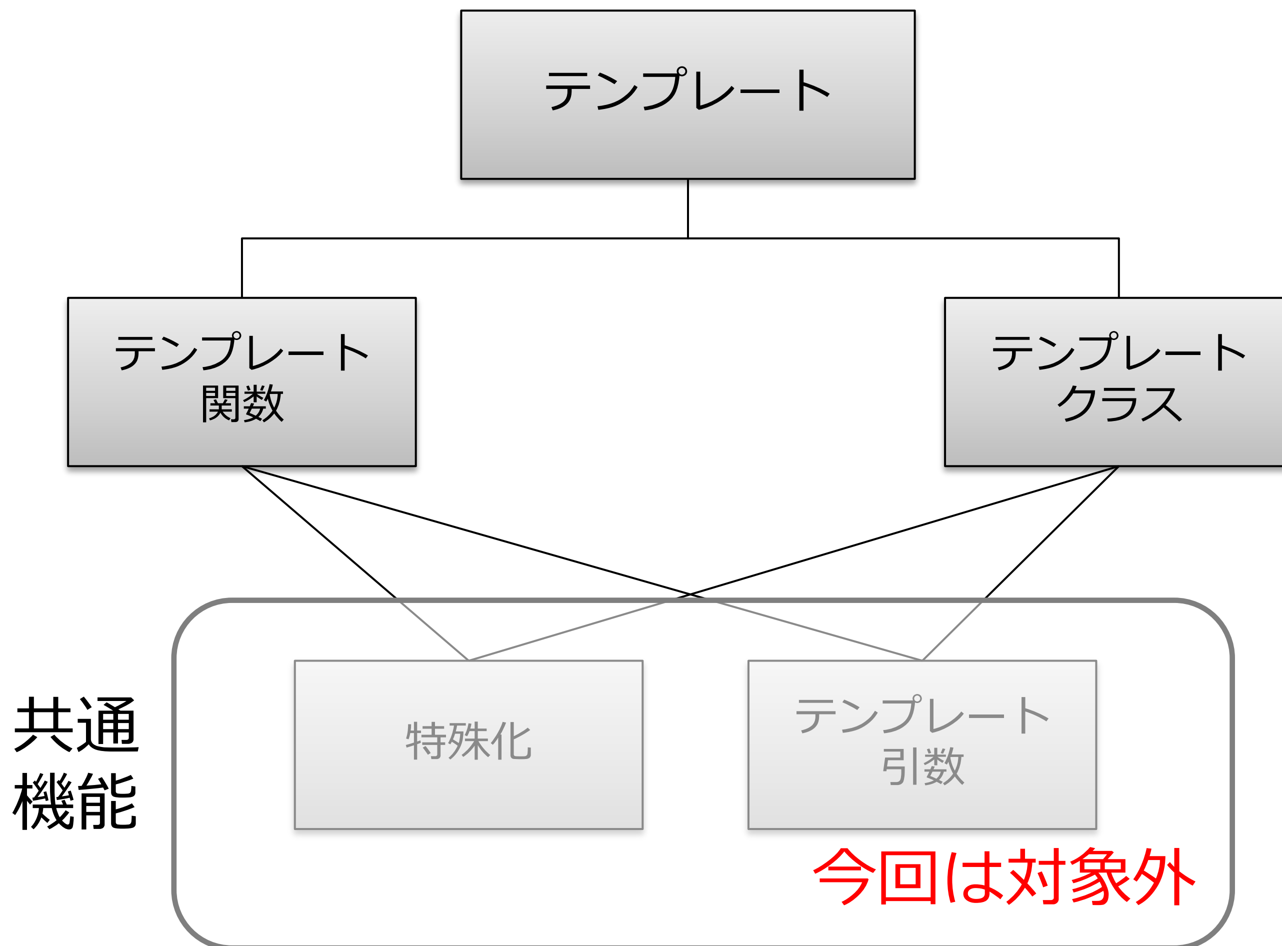
double型の平均計算関数

```
double average(int n, double* seq)
{
  double sum = 0;
  for(int i = 0; i < n; ++i)
  {
    sum += seq[i];
  }
  return sum / n;
}
```

バグがあると全ての型違いに対して修正が必要

テンプレート機能一覧

今回の対象はテンプレート関数、テンプレートクラス



Agenda

- テンプレート概要
- テンプレート関数
- テンプレートクラス
- OpenFoamのVector

テンプレート関数

関数の戻り値・引数を抽象化した関数

平均計算関数

```
template<typename T>
T average(int n, T* seq)
{
  T sum = 0;
  for(int i = 0; i < n; ++i)
  {
    sum += seq[i];
  }
  return sum / n;
}
```

利用方法

```
int main()
{
  int n = 3;
  double seq = {0.1, 2.5, 3.2};
  std::cout  << average<double>(n, seq) << “¥n”;
  return 0;
}
```

テンプレート関数

テンプレート引数は複数指定が可能

最小値取得関数　例1：

```
template<typename T>
T min(T a, T b)
{
   if(a < b) return a;
   return b;
}
```

利用方法

```
int main()
{
 int a = 3, b = 4;
 int m = min(a, b);
 return 0;
}
```

※型が自明の場合は
利用時に省略可能

最小値取得関数　例2：

```
template<typename T1, typename T2, typename T3>
T3 min(T1 a, T2 b)
{
   if(a < b) return a;
   return (T3)b;
}
```

利用方法

```
int main()
{
 int a = 3;
 float b = 2.1;
 double m = min<double>(a, b);
 return 0;
}
```

Agenda

- テンプレート概要
- テンプレート関数
- テンプレートクラス
- OpenFoamのVector

テンプレートクラス

メンバ変数をテンプレート・メンバ関数をテンプレート関数

3次元ベクトルクラス例：

```
template<typename T>
class Vector3D
{
public:
  T x, y, z;

  Vector3D(T X, T Y, T Z)
  { x = X, y = Y, z = Z; };

  ~Vector3D()
  {};

  T innerProduct(const Vector3D<T>& vec) const
  {
    return x * vec.x + y * vec.y + z * vec.z;
  };
};
```

利用時の注意

型が一致していること

3次元ベクトルクラス利用例　**OKな場合**：

```
template<typename T>
class Vector3D
{
public:
  T x, y, z;

  Vector3D(T X, T Y, T Z)
  { x = X, y = Y, z = Z; };

  ~Vector3D()
  {};

  T innerProduct(const Vector3D<T>& vec) const
  {
    return x * vec.x + y * vec.y + z * vec.z;
  };
};
```

```
int main()
{
  Vector3D<int> a(10, 10, 10);
  Vector3D<int> b(2, 3, 4);

  std::cout << "inner product = “
                << a.innerProduct(b) << "¥n";
  return 0;
}
```

利用時の注意

型が一致していること

3次元ベクトルクラス利用例　**NGな場合**：

```
template<typename T>
class Vector3D
{
public:
  T x, y, z;

  Vector3D(T X, T Y, T Z)
  { x = X, y = Y, z = Z; };

  ~Vector3D()
  {};

  T innerProduct(const Vector3D<T>& vec) const
  {
    return x * vec.x + y * vec.y + z * vec.z;
  };
};
```

```
int main()
{
  Vector3D<int> a(10, 10, 10);
  Vector3D<double> b(2.001, 3, 4);

  std::cout << "inner product = “
            << a.innerProduct(b) << "¥n";
  return 0;
}
```

コンパイルエラー解説

メンバ関数innerProduct()の引数の型の不一致

3次元ベクトルクラス：

```
template<typename T>
class Vector3D
{
  …
  T innerProduct(const Vector3D<T>& vec) const
  {
    return x * vec.x + y * vec.y + z * vec.z;
  };
};
```

innerProduct()の引数はVector3D<T>型

利用側：

```
int main()
{
  Vector3D<int> a(10, 10, 10);
  Vector3D<double> b(2.001, 3, 4);

  std::cout << "inner product = “
                 << a.innerProduct(b) << "¥n";
  return 0;
}
```

変数aの型はVector3D<int>

- Vector3D<int>のメンバ関数 innerProduct()の引数はVector3D<int>
- Vector3D<double>は 引数として受け取れない

対策

メンバ関数innerProduct()をテンプレート関数化

3次元ベクトルクラス：

```
template<typename T>
class Vector3D
{
 …
 template<typename T2>
 T innerProduct(const Vector3D<T2>& vec) const
 {
  return (T)(x * vec.x + y * vec.y + z * vec.z);
 };
};
```

型Tにキャスト
自動型変換(たとえばdoubleからint)の
コンパイル時警告発生を防ぐため

利用側：

```
int main()
{
 Vector3D<int> a(10, 10, 10);
 Vector3D<double> b(2.001, 3, 4);

 std::cout << "inner product = “
           << a.innerProduct(b) << "¥n";
 return 0;
}
```

```
Inner product = 90
```

対策

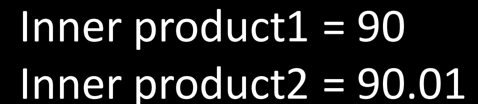
型違い（= 数値精度の違い）わかりづらいので要注意

3次元ベクトルクラス：

```
template<typename T>
class Vector3D
{
  …
  template<typename T2>
  T innerProduct(const Vector3D<T2>& vec) const
  {
    return (T)(x * vec.x + y * vec.y + z * vec.z);
  };
};
```

利用側：

```
int main()
{
  Vector3D<int> a(10, 10, 10);
  Vector3D<double> b(2.001, 3, 4);

  std::cout << "inner product1 = “
                  << a.innerProduct(b) << "¥n";
  std::cout << "inner product2 = “
                  << b.innerProduct(a) << "¥n";
  return 0;
}
```



ビルド時の注意

ヘッダファイルに定義記述が必要

Vector3D.h（宣言を記述）

```
template<typename T>
class Vector3D
{
public:
  T x, y, z;

  Vector3D(T X, T Y, T Z);
  ~Vector3D();

  template<typename T2>
  T innerProduct(const Vector3D<T2>& vec) const;
};
```

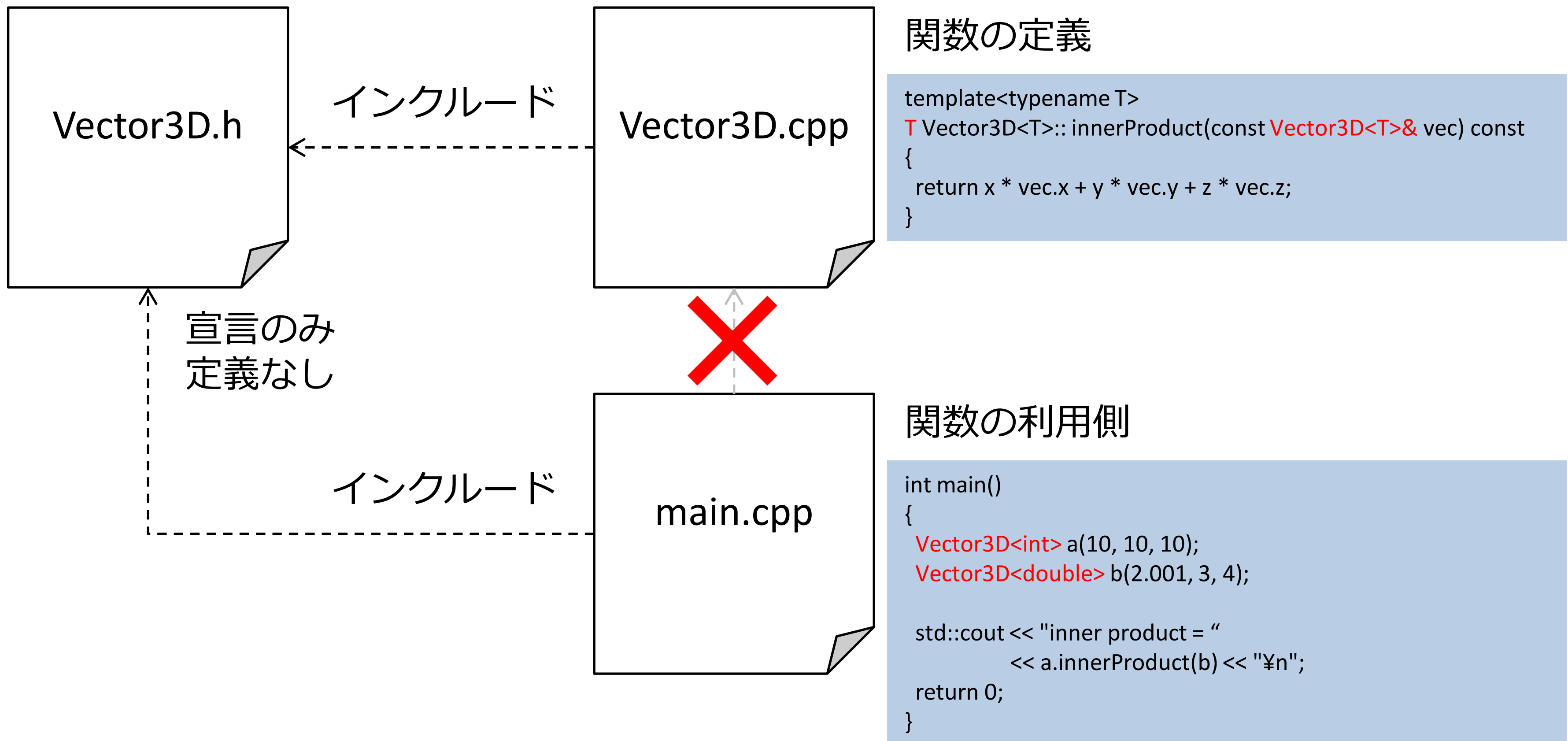
Vector3D.cpp（定義を記述）

```
#include <Vector3D.h>

template<typename T>
Vector3D<T>::Vector3D<T>(T X, T Y, T Z)
{ x = X, y = Y, z = Z; }

template<typename T>
Vector3D<T>:: ~Vector3D()
{};

template<typename T>
T Vector3D<T>:: innerProduct(const Vector3D<T>& vec) const
{
  return x * vec.x + y * vec.y + z * vec.z;
}
```

利用側コードのコンパイル時に、
コンパイラが定義を見つけられずビルドエラー

ビルド時の注意

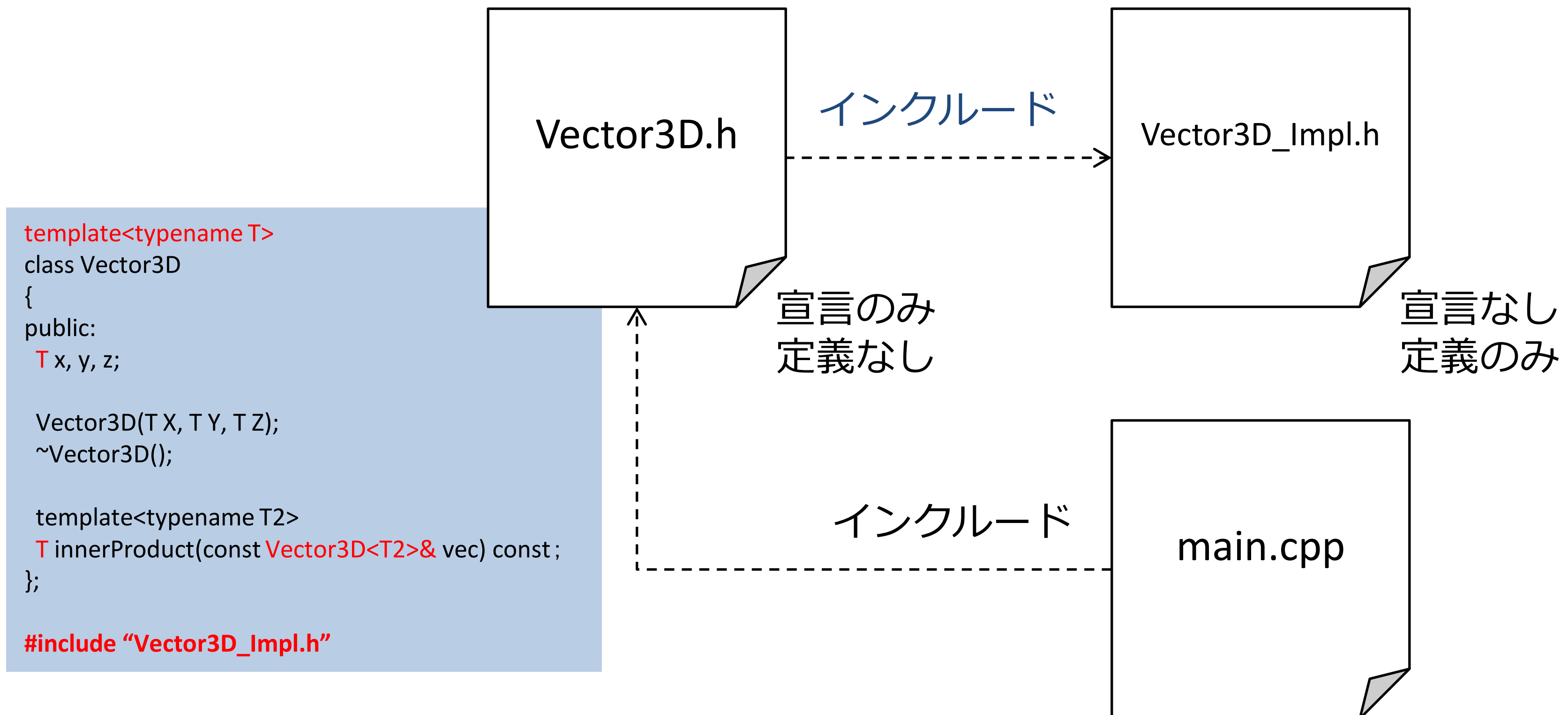
利用側コードコンパイル時にテンプレート定義たどれないため



ビルド時の注意

解決策

- 定義も全てヘッダファイルに記載
- 定義のみを記述したヘッダファイルをインクルード

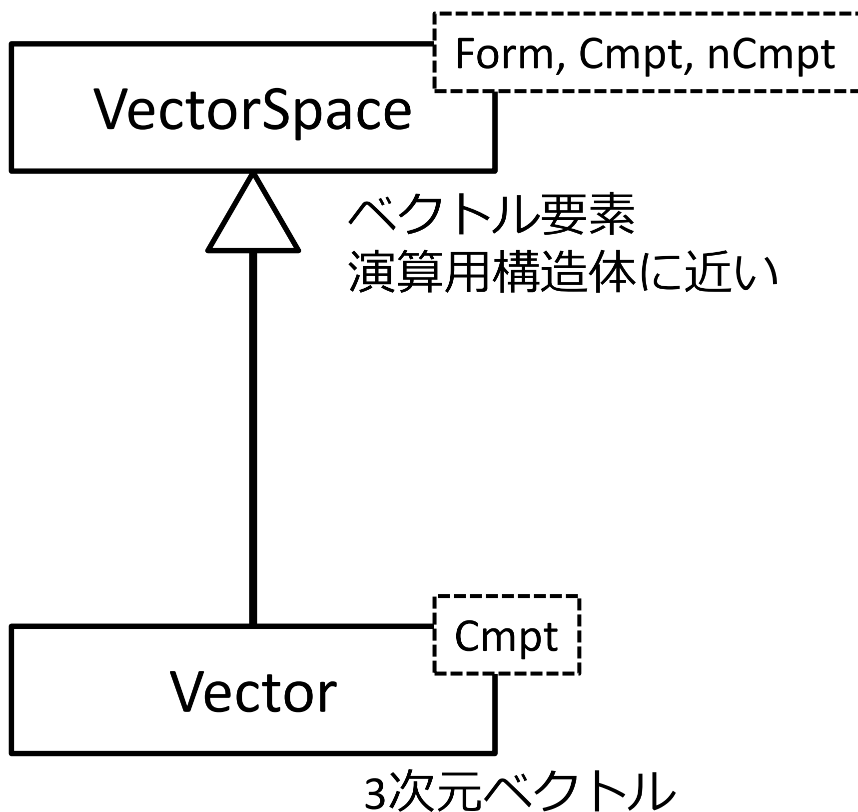


Agenda

- テンプレート概要
- テンプレート関数
- テンプレートクラス
- OpenFoamのVector

OpenFoamのVector

ベクトルはテンプレートクラス




```
template<class Form, class Cmpt, int nCmpt>
class VectorSpace
{
  …
  Cmpt v_[nCmpt];
  …
};
```

Cmpt型（int / float / double）の
nCmpt次元ベクトル
(FormはCmptと基本的に一致。※CRTPイディオム)

```
template<class Cmpt>
class Vector
 : public VectorSpace<Vector<Cmpt>, Cmpt, 3>
{
  …
};
```

3次元ベクトルとして定義
ただし型（int / float/ doubleなど）は利用側で決定

OpenFoamのVector演算

ベクトル演算はテンプレート関数

ベクトル同士の足し算（演算子のオーバーロード）

```
template<class Form, class Cmpt, int nCmpt>
inline Form operator+
 (
   const VectorSpace<Form, Cmpt, nCmpt>& vs1,
   const VectorSpace<Form, Cmpt, nCmpt>& vs2
 )
{
   Form v;
   VectorSpaceOps<nCmpt, 0>::op(v, vs1, vs2, plusOp<Cmpt>());
}
```

足し算の実装は

- VectorSpaceOps<nCmpt, 0>
- plusOp<Cmpt>

Formは
VectorSpace<Foam, Cmpt, nCmpt>
と一致する必要あり

OpenFoamのVector演算

VectorSpaceOpsクラスは抽象化したベクトル演算用

VectorSpaceOpsクラスの定義：

```
template<int N, int I>
class VectorSpaceOps
{
  public:
  static const int endLoop = (I < N-1) ? 1 : 0;
  …
  template<class V, class V1, class Op>
  static  inline void op(V& vs, const V1 vs1, const V1& vs2, Op o)
  {
    vs.v_[I] = o(vs1.v_[I], vs2.v_[I]);
    VectorSpaceOps<endLoop*N, endLoop*(I+1)>::op(vs, vs1, vs2, o);
  }
  …
};
```

メンバ関数opはN次元のベクトルのI番目の要素の演算
演算終了後、I+1番目の演算を実行するVecstorSpaceOpsのメンバ関数opを呼び出し
（再帰処理（のような）プログラミング）

OpenFoamのVector演算

各演算はテンプレートクラスのファンクタ

足し算の実行呼び出しはVectorSpaceOps<nCmpt, 0>::op(v, vs1, vs2, plusOp<Cmpt>());

```
template<int N, int I>
class VectorSpaceOps
{
 …
 static  inline void op(V& vs, const V1 vs1, const V1& vs2, Op o)
 {
   vs.v_[I] = o(vs1.v_[I], vs2.v_[I]);
   VectorSpaceOps<endLoop*N, endLoop*(I+1)>::
 }
 …
};
```

```
template<class T>
class plusOp
{
Public:
 T operator()(const T& x, const T& y) const
 {
   return x + y;
 }
};
```

※コンパイル時に自動生成されるコード

OpenFoamのVectorまとめ

Vector使うのは簡単だが、内部実装はややこしい