# OpenFoamのためのC/C++

第2回　~~OpenFOAMで勉強するクラス~~
CFDの例で勉強するクラス

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　~~OpenFOAMで勉強するクラス~~
  ⇒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

※第2回の予定変更理由
OpenFOAMはテンプレートクラス機能を前提としてるため

# 今回のテーマ

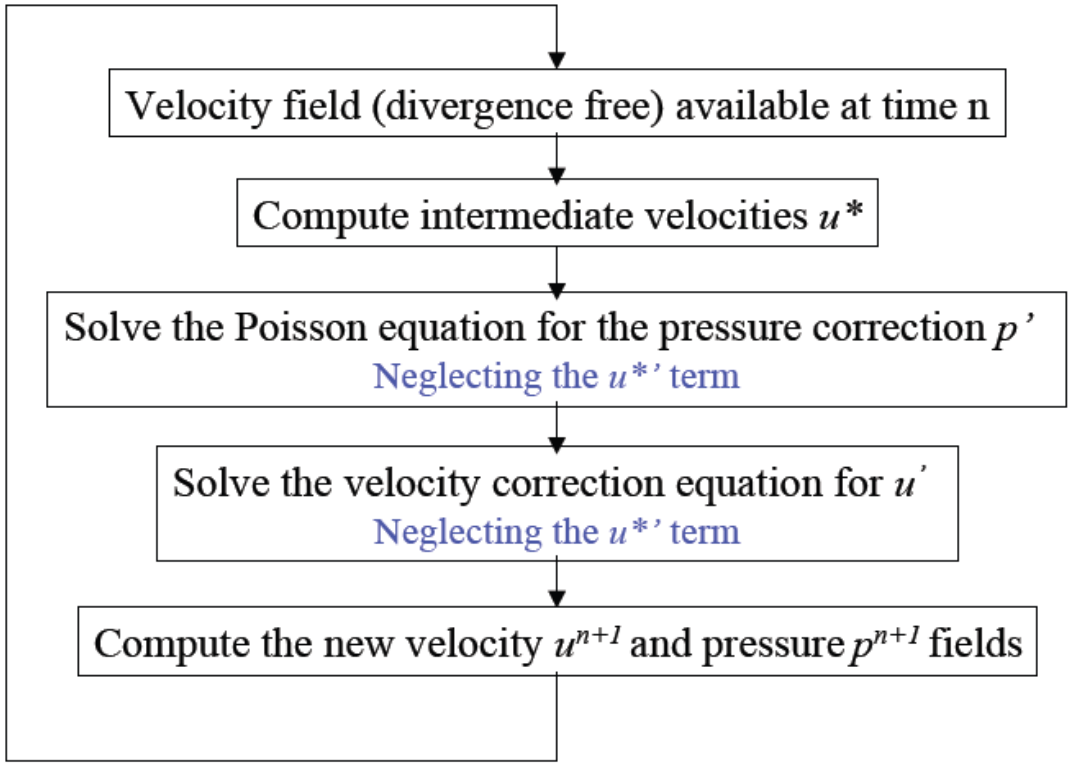
C++によるオブジェクト指向プログラミングの基本を学ぶ

簡単な例（2項演算）
CFDソルバーの例を使って勉強

# CFDの例

Implicit pressure-based scheme for NS equations (SIMPLE)

SIMPLE: Semi-Implicit Method for Pressure-Linked Equations

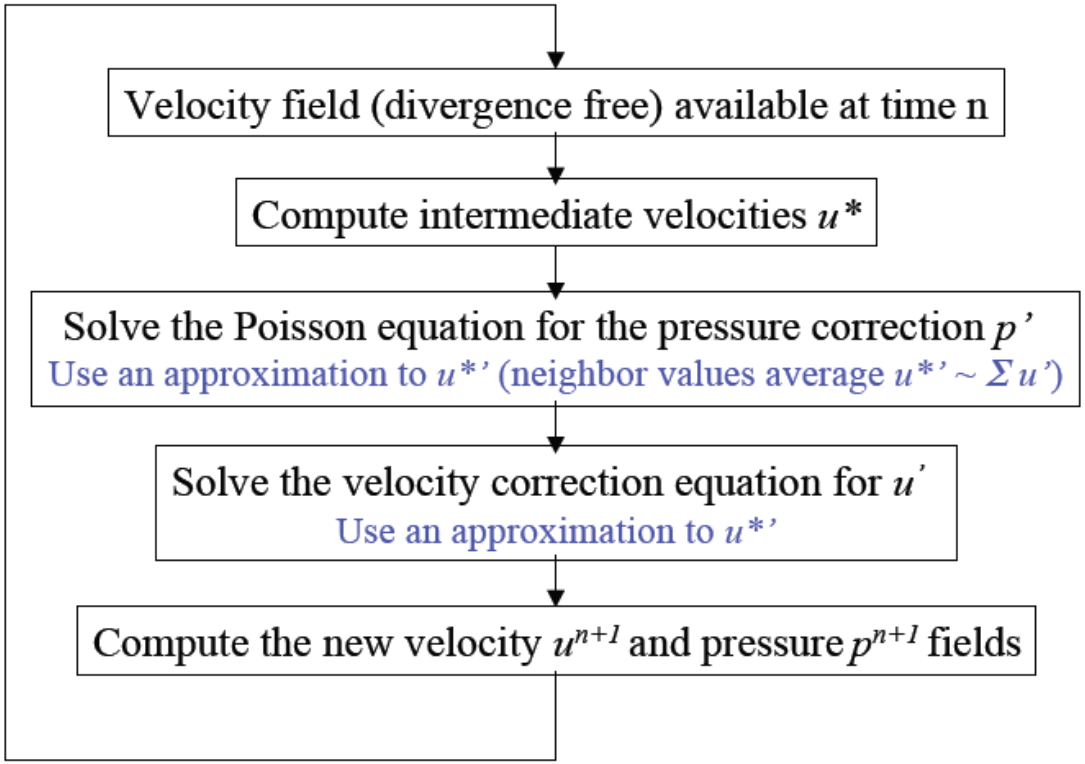



※ http://www.icmc.usp.br/~gustavo.buscaglia/cursos/mfc_undergrad2/Iaccarino_incompressible.pdf

# CFDの例

Implicit pressure-based scheme for NS equations (SIMPLEC)

SIMPLE: SIMPLE Corrected/Consistent




※ http://www.icmc.usp.br/~gustavo.buscaglia/cursos/mfc_undergrad2/Iaccarino_incompressible.pdf

# CFDの例

## C言語チックなSIMPLE法 / SIMPLEC法

```
typedef struct Vector2D_ Vector2D;
typedef struct fv_ fv;
typedef struct vectorField_ vectorField;
typedef struct scalarField_ scalarField;

fv* m = NULL;
vectorField* velocity = NULL;
scalarField* pressure = NULL;

void initialize(fv** m, vectorField** v, scalarField** p);
void calcIntermediateVelocity(fv* v, vectorField* velocity, double dt, double gravity);

void calcPoisson_SIMPLE(fv* v, vectorField * velocity, scalarField* pressure);
void update_SIMPLE (fv* v, vectorField* velocity, scalarField* pressure);

void calcPoisson_SIMPLEC(fv* v, vectorField * velocity, scalarField* pressure);
void update_SIMPLE C(fv* v, vectorField* velocity, scalarField* pressure);

void navieStokes_SIMPLE();
void navieStokes_SIMPLEC();
```

# CFDの例

## C言語チックなSIMPLE法 / SIMPLEC法

SIMPLE法実行関数:

```
void navieStokes_SIMPLE()
{
  initialize(fv, velocity, pressure);
  for(int i = 0; i < numSteps; ++i)
  {
    calcIntermediateVelocity(fv, velocity, dt, g);
    calcPoisson_SIMPLE(fv, velocity, pressure);
    update_SIMPLE(fv, velocity, pressure);
  }
}
```

SIMPLEC法実行関数:

```
void navieStokes_SIMPLEC()
{
  initialize(fv, velocity, pressure);
  for(int i = 0; i < numSteps; ++i)
  {
    calcIntermediateVelocity(fv, velocity, dt, g);
    calcPoisson_SIMPLEC(fv, velocity, pressure);
    update_SIMPLEC(fv, velocity, pressure);
  }
}
```

※“numSteps”, “dt”, “g”はグローバル変数

# 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

- オブジェクト指向
- プログラムの分割・整理
- 計算フローの明確化
- プログラムの流用
- 機能追加の簡単化

# オブジェクト指向

- オブジェクトと呼ばれる機能の部品
- 機能の部品を組み合わせてソフト開発

# オブジェクト指向

## 基本的概念

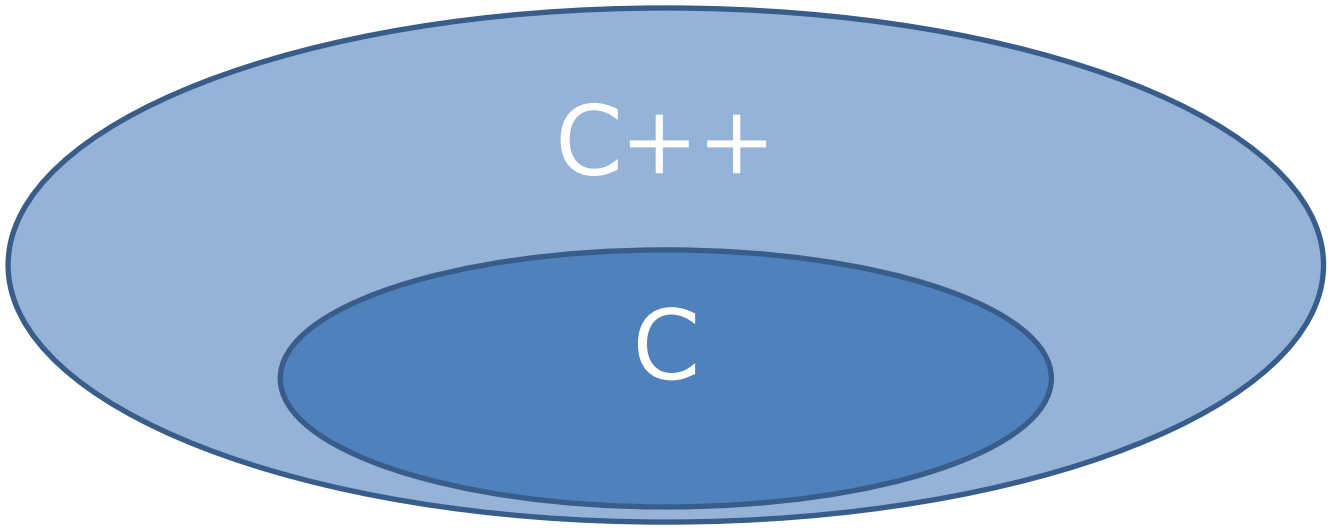
| 概念 | 効果 |
| --- | --- |
| クラス | プログラムの分割・整理 |
| 情報隠蔽 | うっかりミスの回避 |
| 継承 | プログラムの流用 |
| 多態性 | 機能追加の簡単化 |

プログラムの機能（役割）ごとに部品化
大規模開発の効率化を実現

# CとC++の違い

C++はCより色々なことができる

- C仕様はC++仕様のサブセット
- C++はオブジェクト指向を利用可能



# Agenda

- オブジェクト指向
- プログラムの分割・整理
- 計算フローの明確化
- プログラムの流用
- 機能追加の簡単化

# オブジェクト指向

## 基本的概念

| 概念 | 効果 |
| --- | --- |
| クラス | プログラムの分割・整理 |
| 情報隠蔽 | うっかりミスの回避 |
| 継承 | プログラムの流用 |
| 多態性 | 機能追加の簡単化 |

# オブジェクト指向

- オブジェクトと呼ばれる機能の部品
- 機能の部品を組み合わせてソフト開発

**「オブジェクト」は機能の部品（変数の値+関数）**
**「オブジェクトの定義」はクラス（変数+関数）**

# クラスのC++流書き方

## 構造体を拡張(**+関数**)

2項足し算構造体例:

```
struct AddOperator
{
 int a_, b_;

 AddOperator(int a, int b)
 {
  a_ = a, b_ = b;
 }

 int opreate()
 {
  return a_ + b_;
 }
};
```

2項足し算クラス例:

```
class AddOperator
{
public:
 int a_, b_;

 AddOperator(int a, int b)
 {
  a_ = a, b_ = b;
 }

 int opreate()
 {
  return a_ + b_;
 }
};
```

構造体も概念はクラス

# メンバ変数

## データをひとまとまりにする

2項足し算クラス例：

```
class AddOperator
{
public:
 int a_, b_;

 AddOperator(int a, int b)
 {
  a_ = a, b_ = b;
 }

 int opreate()
 {
  return a_ + b_;
 }
};
```

メンバ変数：
Cの構造体のメンバとほぼ同じ

# メンバ関数

## 操作をひとまとまりにする

2項足し算クラス例：

```
class AddOperator
{
public:
 int a_, b_;

 AddOperator(int a, int b)
 {
  a_ = a, b_ = b;
 }

 int opreate()
 {
  return a_ + b_;
 }
};
```

コンストラクタ：
変数のメモリ確保時に自動実行される

メンバ関数：
“a_”, “b_” は
自身のクラスのメンバ変数

メンバ関数内は、無条件にメンバ変数アクセス可能

# クラスの利用

## 型がクラス、変数がオブジェクト

2項足し算実行例（実体）:

```
void main()
{
  AddOperator add(1, 5);
  std::cout << “add result = “ << add.operate() << “¥n”;
}
```

```
add result = 6
```

2項足し算実行例（ポインタ変数）:

```
void main()
{
  AddOperator* add = new AddOperator(1, 5);
  std::cout << “add result = “ << add->operate() << “¥n”;
}
```

```
add result = 6
```

# メリット

- 計算に必要なパラメータ(変数)と
  計算処理(関数)を1つにまとめられる

  ⇒どのパラメータがどの計算処理に利用
  　されているか探しやすい

  ⇒複数人開発の際、分担が容易

# CFDの例の改善
# （クラス化）

# CFDの例

## C言語チックなSIMPLE法 / SIMPLEC法

```
typedef struct Vector2D_ Vector2D;
typedef struct fv_ fv;
typedef struct vectorField_ vectorField;
typedef struct scalarField_ scalarField;

fv* m = NULL;
vectorField* velocity = NULL;
scalarField* pressure = NULL;

void initialize(fv** m, vectorField** v, scalarField** p);
void calcIntermediateVelocity(fv* v, vectorField* velocity, double dt, double gravity);

void calcPoisson_SIMPLE(fv* v, vectorField * velocity, scalarField* pressure);
void update_SIMPLE (fv* v, vectorField* velocity, scalarField* pressure);

void calcPoisson_SIMPLEC(fv* v, vectorField * velocity, scalarField* pressure);
void update_SIMPLE C(fv* v, vectorField* velocity, scalarField* pressure);

void navieStokes_SIMPLE();
void navieStokes_SIMPLEC();
```

# 例の改善（クラス化）

## 設定・陰解法をクラスとして作成

設定クラス：

```
class Setting
{
public:
  int numSteps;
  double dt;
  double viscosity;
  …

  Setting()
  {
    numSteps = 10000;
    dt = 0.0001;
    viscosity = 0.1;
  }
  …
}
```

SIMPLE法クラス：

```
class NavierStokesSIMPLE
{
public:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;

  void initialize(Setting* setting);
  void calcIntermediateVelocity();
  void calcPoisson();
  void update();

  void mainLoop();
}
```

# 例の改善（クラス化）

## 設定・陰解法をクラスとして作成

改善前：

```
void navieStokes_SIMPLE()
{
 initialize(fv, velocity, pressure);
 for(int i = 0; i < numSteps; ++i)
 {
  calcIntermediateVelocity(fv, velocity, dt, g);
  calcPoisson_SIMPLE(fv, velocity, pressure);
  update_SIMPLE(fv, velocity, pressure);
 }
}
```

改善後（SIMPLEクラス　メンバ関数）：

```
void NavierStokesSIMPLE ::mainLoop()
{
 for(int i = 0; i < setting->numSteps; ++i)
 {
  calcIntermediateVelocity();
  calcPoisson();
  update();
 }
}

void main()
{
 Setting setting;
 NavierStokesSIMPLE simple;
 simple.initialize(&setting);
 simple.mainLoop();
}
```

SIMPLEC法も同様

# Agenda

- オブジェクト指向
- プログラムの分割・整理
- 計算フローの明確化
- プログラムの流用
- 機能追加の簡単化

# オブジェクト指向

## 基本的概念

| 概念 | 効果 |
| --- | --- |
| クラス | プログラムの分割・整理 |
| 情報隠蔽 | うっかりミスの回避 |
| 継承 | プログラムの流用 |
| 多態性 | 機能追加の簡単化 |

# private / public

## オブジェクト外からアクセス可/不可を決定

改善前（2項足し算クラス）:

```
class AddOperator
{
public:
 int a_, b_;

 AddOperator(int a, int b)
 {
  a_ = a, b_ = b;
 }

 int opreate()
 {
  return a_ + b_;
 }
};
```

```
void main()
{
 AddOperator add(0, 0);
 add.a_ = 1, add.b_ = 5;
 std::cout << “add result = “ << add.operate() << “¥n”;
}
```

“a_”, “b_”は**public**なので
上プログラムは**問題なくコンパイル可能**

# private / public

## オブジェクト外からアクセス可/不可を決定

改善前（2項足し算クラス）:

```
class AddOperator
{
private:
  int a_, b_;

public:
  AddOperator(int a, int b)
  {
    a_ = a, b_ = b;
  }

  int opreate()
  {
    return a_ + b_;
  }
};
```

```
void main()
{
  AddOperator add(0, 0);
  add.a_ = 1, add.b_ = 5;
  std::cout << “add result = “ << add.operate() << “¥n”;
}
```

“a_”, “b_”は**private**なので
上プログラムは**コンパイルエラー**
**（privateなメンバをオブジェクト外からアクセス）**

# private / public

## オブジェクト外からアクセス可/不可を決定

改善前（2項足し算クラス）:

```
class AddOperator
{
private:
  int a_, b_;

public:
  AddOperator(int a, int b)
  {
    a_ = a, b_ = b;
  }

  int opreate()
  { return a_ + b_;  }

  void setA(int a) { a_ = a; }
  void setB(int b) { b_ = b; }
};
```

```
void main()
{
  AddOperator add(0, 0);
  add.setA(1), add.setB(5);
  std::cout << “add result = “ << add.operate() << “¥n”;
}
```

“a_”, “b_”を変更できるpublicな関数で解決

※“private”, “public”はアクセス修飾子と呼びます

# メリット

- 呼び出し側から
  変更できるメンバ変数、
  呼び出せるメンバ関数を選択可能

  ⇒うっかりミスによる
  メンバ変数の変更、
  不要なメンバ関数の呼び出し
  を避ける事が可能

# CFDの例の改善
# （private　/　public）

# 例の改善（private / public）

## 呼び出し側で利用されないメンバをprivate化

改善前：

```
class NavierStokesSIMPLE
{
public:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;

  void initialize(Setting* setting);
  void calcIntermediateVelocity();
  void calcPoisson();
  void update();

  void mainLoop();
};
```

改善後（private / publicの整理）：

```
class NavierStokesSIMPLE
{
private:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;

  void calcIntermediateVelocity();
  void calcPoisson();
  void update();

public:
  void initialize(Setting* setting);
  void mainLoop();
};
```

# 例の改善（private / public）

## 呼び出し側で利用されないメンバをprivate化

改善後（private / publicの整理）:

```
class NavierStokesSIMPLE
{
private:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;

  void calcIntermediateVelocity();
  void calcPoisson();
  void update();

public:
  void initialize(Setting* setting);
  void mainLoop();
};
```

呼び出し側:

```
void main()
{
  Setting setting;
  NavierStokesSIMPLE simple;
  simple.initialize(&setting);
  simple.mainLoop();
}
```

SIMPLEC法も同様

# Agenda

- C++におけるオブジェクト指向
- プログラムの分割・整理
- プログラムの流用
- 機能追加の簡単化

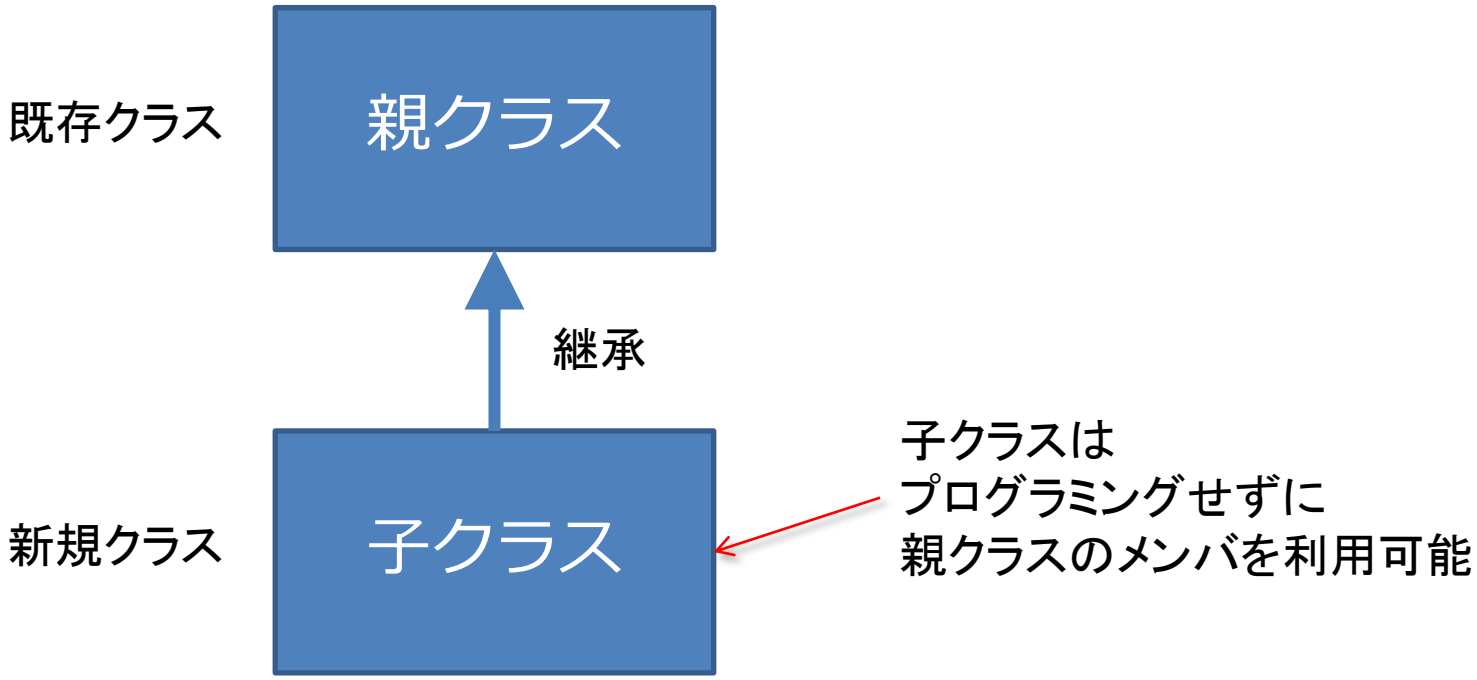
# オブジェクト指向

## 基本的概念

| 概念 | 効果 |
| --- | --- |
| クラス | プログラムの分割・整理 |
| 情報隠蔽 | うっかりミスの回避 |
| 継承 | プログラムの流用 |
| 多態性 | 機能追加の簡単化 |

# 継承

既存クラスをベースに新しいクラスを作成




追加機能分だけプログラミング
⇒共通機能のプログラミング不要

# 継承

親クラス（2項演算）:

```
class BinaryOperator
{
protected:
  int a_, b_;

public:
  BinaryOperator(int a, int b)
  { a_ = a; b_ = b;  }
};
```

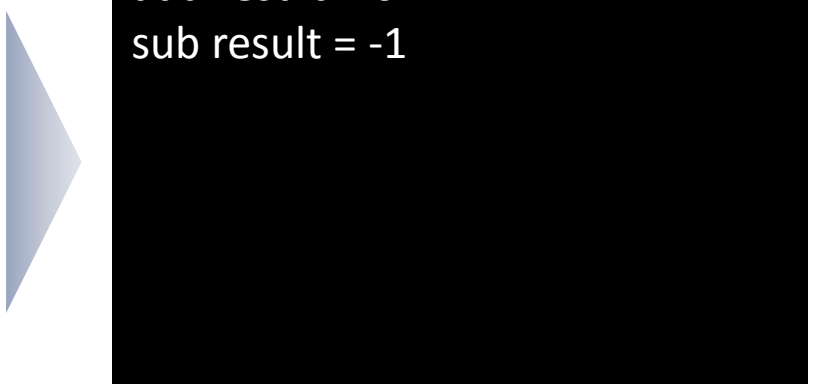
子クラス（足し算、引き算）:

```
class AddOperator : pubic BinaryOperator
{
public:
  AddOperator(int a, int b) : BinaryOperator(a, b)
  {}

  int operate()
  { return a_ + b_; }
};

Class SubtractOperator : pubic BinaryOperator
{
public:
  SubtractOperator(int a, int b) : BinaryOperator(a,
b)
  {}

  int operate()
  { return a_ - b_; }
};
```

# 継承

呼び出し側：

```
void main()
{
  AddOperator add(1, 5);
  std::cout << “add result = “ << add.operate() << “¥n”;

  SubtractOperation sub(2, 3);
  std::cout << “sub result = “ << sub.operate() << “¥n”
}
```



# protectedメンバ

オブジェクト外からはアクセス不可能
子クラスからはアクセス可能

親クラス（2項演算）:

```
class BinaryOperator
{
protected:
  int a_, b_;

public:
  BinaryOperator(int a, int b)
  { a_ = a; b_ = b;  }
};
```

子クラス（足し算）:

```
class AddOperator : pubic BinaryOperator
{
public:
  AddOperator(int a, int b) : BinaryOperator(a, b)
  {}

  int operate()
  { retur a_ + b_; }
};
```

“a_”, “b_”はprotected、子クラス内なのでOK

呼び出し側:

```
void main()
{
  AddOperator add(1, 5);
  std::cout << “a = “ << add.a_ << “¥n”;
}
```

“a_”, “b_”はprotected、
オブジェクト外なのでNG
（コンパイルエラー）

# privateメンバ

## privateメンバは子クラスでもアクセスできない

親クラス（2項演算）:

```
class BinaryOperator
{
private:
  int a_, b_;

public:
  BinaryOperator(int a, int b)
  { a_ = a; b_ = b;  }
};
```

子クラス（足し算）:

```
class AddOperator : pubic BinaryOperator
{
public:
  AddOperator(int a, int b) : BinaryOperator(a, b)
  {}

  int operate()
  { retur a_ + b_; }
};
```

“a_”, “b_”はprivate、子クラス内でもNG
(コンパイルエラー)

# メリット

- 共通の計算処理を
  親クラスだけで定義すれば、
  子クラスでも使える

  ⇒似た機能(役割)内のみで処理を流用

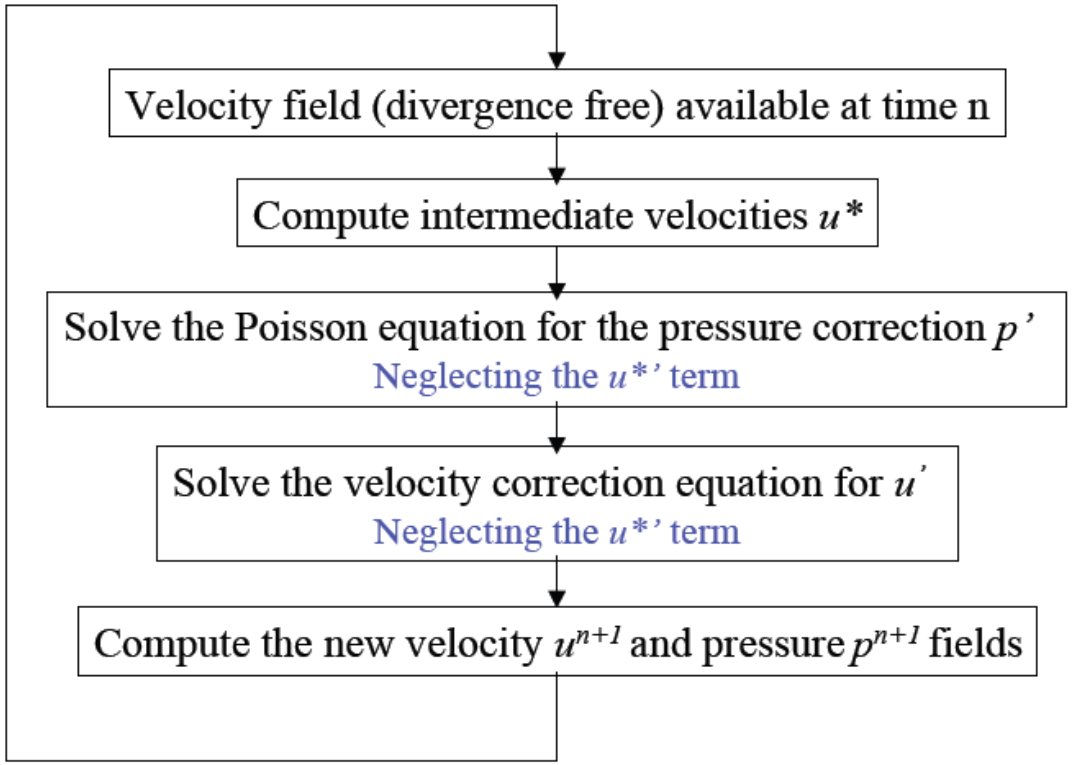
※通常の関数の場合、役割関係なく流用可能

# CFDの例の改善（継承）

# CFDの例

Implicit pressure-based scheme for NS equations (SIMPLE)

SIMPLE: Semi-Implicit Method for Pressure-Linked Equations

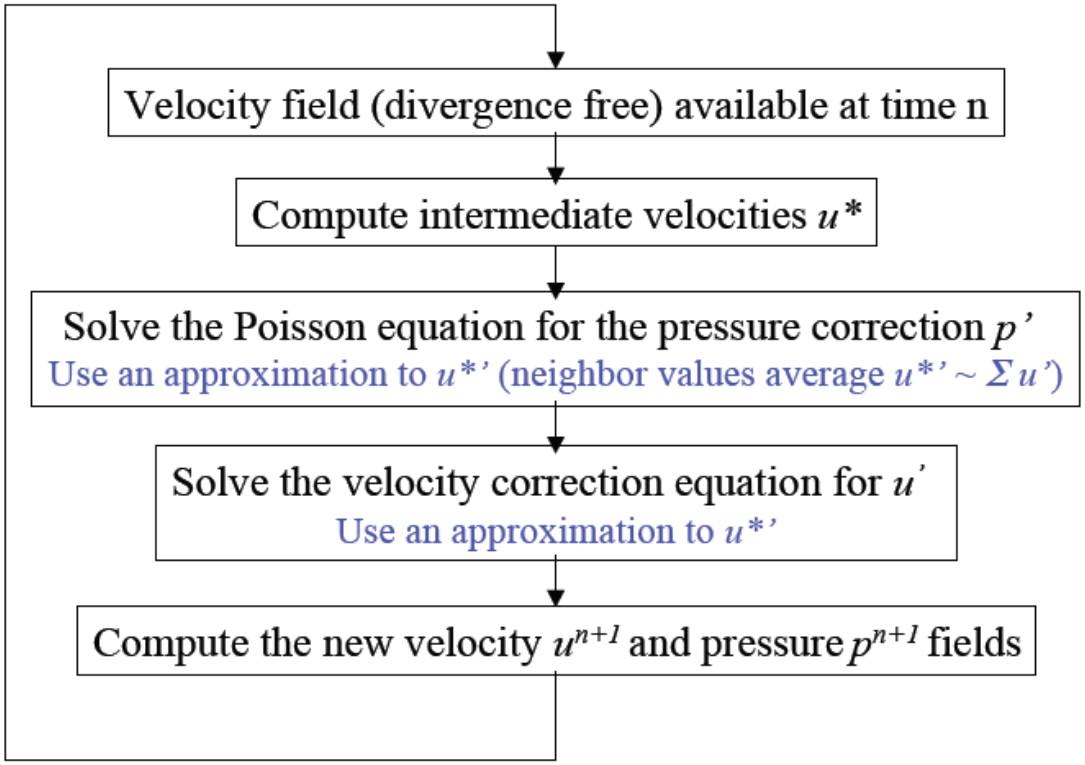



※ http://www.icmc.usp.br/~gustavo.buscaglia/cursos/mfc_undergrad2/Iaccarino_incompressible.pdf

# CFDの例

Implicit pressure-based scheme for NS equations (SIMPLEC)

SIMPLE: SIMPLE Corrected/Consistent




※ http://www.icmc.usp.br/~gustavo.buscaglia/cursos/mfc_undergrad2/Iaccarino_incompressible.pdf

# 継承の利用どころ検討

## 計算処理の共通部分 / 非共通部分を探す

SIMPLEクラス:

```
class NavierStokesSIMPLE
{
private:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;

  void calcIntermediateVelocity();
  void calcPoisson();
  void update();

public:
  void initialize(Setting* setting);
  void mainLoop();
};
```

SIMPLECクラス:

```
class NavierStokesSIMPLEC
{
private:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;

  void calcIntermediateVelocity();
  void calcPoisson();
  void update();

public:
  void initialize(Setting* setting);
  void mainLoop();
};
```

役割(関数名)は同じで、計算処理も同じ

# 継承の利用どころ検討

## 計算処理の共通部分 / 非共通部分を探す

SIMPLEクラス：

```
class NavierStokesSIMPLE
{
private:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;

  void calcIntermediateVelocity();
  void calcPoisson();
  void update();

public:
  void initialize(Setting* setting);
  void mainLoop();
};
```

SIMPLECクラス：

```
class NavierStokesSIMPLEC
{
private:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;

  void calcIntermediateVelocity();
  void calcPoisson();
  void update();

public:
  void initialize(Setting* setting);
  void mainLoop();
};
```

役割（関数名）は同じだが、計算処理は異なる

# 例の改善（継承）

## 共通部分を親クラス/非共通部分を子クラス

改善後：

```
class NavierStokesSIMPLE_Base
{
protected:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;
  void calcIntermediateVelocity();

public:
  void initialize(Setting* setting);
};
```

```
class NavierStokesSIMPLE : public NavierStokesSIMPLE_Base
{
private:
 void calcPoisson();
 void update();
public:
 void mainLoop();
};

class NavierStokesSIMPLEC : public NavierStokesSIMPLE_Base
{
private:
  void calcPoisson();
  void update();
public:
  void mainLoop();
};
```

メンバ関数calcIntermediateVelocity()は親クラスのみで定義でOK

# Agenda

- C++におけるオブジェクト指向
- プログラムの分割・整理
- プログラムの流用
- 機能追加の簡単化

# オブジェクト指向

## 基本的概念

| 概念 | 効果 |
| --- | --- |
| クラス | プログラムの分割・整理 |
| 情報隠蔽 | うっかりミスの回避 |
| 継承 | プログラムの流用 |
| 多態性 | 機能追加の簡単化 |

# 多態性

- 親クラスのポインタ・参照で子クラス保持
- 仮想関数による処理選択

# 多態性（仮想関数）

親クラス（2項演算）:

```
class BinaryOperator
{
protected:
  int a_, b_;

public:
  BinaryOperator(int a, int b)
  { a_ = a; b_ = b;  }

  virtual int operate()
  {};
};
```

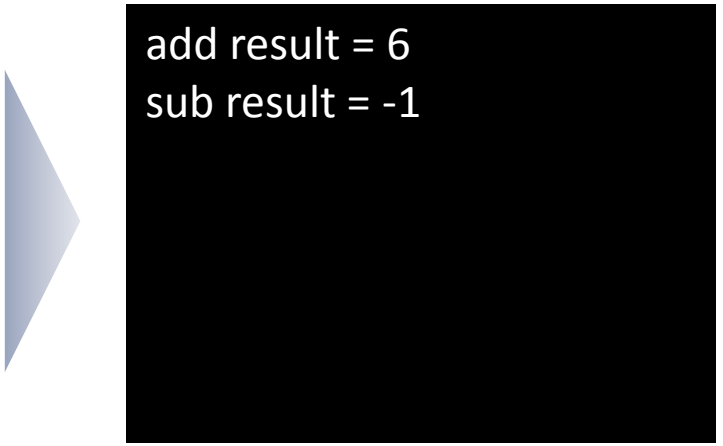
子クラス（足し算、引き算）:

```
class AddOperator : pubic BinaryOperator
{
public:
  AddOperator(int a, int b) : BinaryOperator(a, b)
  {}

 virtual int operate()
  { return a_ + b_; }
};

Class SubtractOperator : pubic BinaryOperator
{
public:
  AddOperator(int a, int b) : BinaryOperator(a, b)
  {}

 virtual int operate()
  { return a_ - b_; }
};
```

# 多態性（仮想関数）

呼び出し側：

```
int operation(BinaryOperator* operation)
{
  return operation->operate();
}

void main()
{
  BinaryOperator* add = new AddOperator(1, 5);
  std::cout << “add result = “ << operation(add) << “¥n”;

  BinaryOperator* sub = new SubtractOperation (2, 3);
  std::cout << “sub result = “ << operation(sub) << “¥n”
}
```




親クラスのポインタで子クラスのオブジェクトを保持

# 多態性（仮想関数）

呼び出し側：

```
int operation(BinaryOperator* operation)
{
  return operation->operate();
}

void main()
{
  BinaryOperator* add = new AddOperator(1, 5);
  std::cout << “add result = “ << operation(add) << “¥n”;

  BinaryOperator* sub = new SubtractOperation (2, 3);
  std::cout << “sub result = “ << operation(sub) << “¥n”
}
```

```
add result = 6
sub result = -1
```

仮想関数による処理選択

# 多態性（仮想関数）

## 機能追加が容易

呼び出し側：

```
int operation(BinaryOperator* operation)
{
  return operation->operate();
}

void main()
{
  BinaryOperator* add = new AddOperator(1, 5);
  std::cout << “add result = “ << operation(add) << “¥n”;

  BinaryOperator* sub = new SubtractOperation (2, 3);
  std::cout << “sub result = “ << operation(sub) << “¥n”

  BinaryOperator* mul = new MultiOperator(-2, 5);
  std::cout << “mul result = “ << operation(mul) << “¥n”;
}
```

```
add result = 6
sub result = -1
mul result = -10
```

呼び出し側（opreation()関数）を変更不要
（オブジェクト生成時に処理を選択）

# メリット

- 親クラスのポインタ・参照で子クラス保持
- 仮想関数による処理選択

⇒呼び出し側を変更不要
　子クラスを追加することで機能追加
　(オブジェクト生成時に処理選択可能)

# CFDの例の改善
# (多態性)

# 例の改善（多態性）

## 仮想関数による処理選択

改善前：

```
class NavierStokesSIMPLE_Base
{
protected:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;
  void calcIntermediateVelocity();

public:
  void initialize(Setting* setting);
};
```

```
class NavierStokesSIMPLE : public NavierStokesSIMPLE_Base
{
public:
 void calcPoisson();
 void update();
 void mainLoop();
};

class NavierStokesSIMPLEC : public NavierStokesSIMPLE_Base
{
protected :
  void calcPoisson();
  void update();
  void mainLoop();
};
```

# 例の改善（多態性）

改善後（クラス宣言）:

```
class NavierStokesSIMPLE_Base
{
protected:
  Setting* setting;
  fv* m;
  vectorField* velocity;
  scalarField* pressure;
  void calcIntermediateVelocity();
  virtual void calcPoisson();
  virtual void update();

public:
  void initialize(Setting* setting);
  void mainLoop();
};
```

```
class NavierStokesSIMPLE : public NavierStokesSIMPLE_Base
{
public:
 virtual void calcPoisson();
 virtual void update();
};

class NavierStokesSIMPLEC : public NavierStokesSIMPLE_Base
{
protected :
 virtual void calcPoisson();
 virtual void update();
};
```

# 例の改善（多態性）

改善後（mainLoop()関数）:

```
Void NavierStokesSIMPLE_Base:: mainLoop()
{
  for(int i = 0; i < setting->numSteps; ++i)
  {
    calcIntermediateVelocity();
    calcPoisson();
    update();
  }
}
```

改善後（呼び出し側）:

```
void run(NavierStokesSIMPLE_Base* solver,
  Setting* setting)
{
  solver->initialize(setting);
  solver->mainLoop();
}

void main()
{
  Setting setting;
  NavierStokesSIMPLE_Base* simple
     = new NavierStokesSIMPLE();

   NavierStokesSIMPLE_Base* simplec
     = new NavierStokesSIMPLEC();

   run(simple, &setting);
}
```

メンバ関数calcIntermediateVelocity()は親クラスのみで定義でOK

# Appendix

# その他

オブジェクト指向ではないが、頻繁に使われる機能

- オーバーロード
- 名前空間
- const変数 / const関数

# 参考文献


- CFD
  http://www.icmc.usp.br/~gustavo.buscaglia/cursos/mfc_undergrad2/Iaccarino_incompressible.pdf
- 5分で分かるオブジェクト指向
  http://www.atmarkit.co.jp/im/carc/special/fiveoo/01.html
- 情報隠蔽に関する議論
  http://d.hatena.ne.jp/bleis-tift/20090201/1233426011