目的

クラス の基本について学ぶ。

1. クラス
2. コンストラクタ
3. テンプレート クラス
4. 継承
5. オペレータ(演算子)のオーバーロード

基本方針

プログラムの main はできるだけシンプルにする。クラスを活用する。

サンプル

Code01

ごく基本的なプログラム例を CODE 1 に示す。キーボードから整数を入力し，その値の2倍を画面に表示する。

```
#include <iostream>

int main()
{
    int n;
    std::cout << "整数を入力してください。" << std::endl;
    std::cin >> n;
    std::cout << "整数 " << n << " の2倍は " << n + n << " です。" << std::endl;

     return 0;
}
```

Code01r : スコープ

変数の有効範囲（スコープ）について考える。

```
#include <iostream>

int main()
{
    int n;
    std::cout << "整数を入力してください。" << std::endl;
    std::cin >> n;
    std::cout << "整数 " << n << " の2倍は " << n + n << " です。" << std::endl;
```

```
    {
        int n=1000;  // このnのスコープは，{ }の内部のみ
        std::cout << "整数 " << n << " の2倍は " << n + n << " です。" << std::endl;
    }

    std::cout << "整数 " << n << " の2倍は " << n + n << " です。" << std::endl;

     return 0;
}
```

Code02：関数の作成

整数の値を2倍にする計算を関数とする。

```
#include <iostream>

// function
int makeDouble(const int& n)
{
   return n + n ;
}

int main()
{
    int n;
    std::cout << "整数を入力してください。" << std::endl;
    std::cin >> n;
    std::cout << "整数 " << n << " の2倍は " << makeDouble(n) << " です。" <<
std::endl;

     return 0;
}
```

Code03：クラスの作成

整数 (int) 型を発展させて，その値を2倍にする機能のある **新しい整数型** を作り出すとする。この機能をクラスとして実装する。クラス名を extendedInt とする。

まずは，クラスというものを作成し，先ほどの関数をクラスのメソッドとして実装する。

mainプログラムの中では，作成したクラスから，extIntという名前のインスタンス（実体）を作成する。クラスは，あくまでも型であり，実際に使用する時には実体が必要である。このインスタンスをオブジェクトと呼ぶ。

クラスで定義したメソッドは，オブジェクトの一部として存在する。mainプログラムでは，`オブジェクト名.メソッド名` と書く（オブジェクト名とメソッド名の間にドット演算子を置く）ことで，メソッドが利用できる。

なお，オブジェクトをポインタで操作している時には，ドット演算子に変わって，「アロー演算子」を使う。例えば，`someObjectPointer->someMethod()` のような記述となる。

```
#include <iostream>

// クラスの定義　スタート
class extendedInt
{
    public:

        int makeDouble(const int& n)
        {
            return n + n ;
        }
};
// クラスの定義　ここまで

int main()
{
    int n;
    extendedInt extInt;

    std::cout << "整数を入力してください。" << std::endl;
    std::cin >> n;

    // extendedIntクラスのメソッドmakeDouble(int)を利用する。
    std::cout << "整数 " << n << " の2倍は " << extInt.makeDouble(n) << " です。" <<
std::endl;

    return 0;
}
```

はじめてクラスを作ることだけを考えた例である。実際には，クラスの中には値を持つのが普通である。

Code04：クラスのもつメンバ変数

クラスを改良して，少しはクラスらしくしてみる。そのために，値自体をクラスが保持するように修正する。

先のコードで実行していた作業のほとんどはクラスで実行することにする。つまり，キーボードからの値の読み込み，2倍にする計算，計算結果の表示など，

```
#include <iostream>

// クラスの定義　スタート
class extendedInt
{
    int n_;  //メンバ変数，指定しない時はprivate

    public:

        // n_を2倍した値を返す。値自体は不変である。
        int makeDouble()
        {
            return n_ + n_ ;
```

```
        }

        // 2倍の値を表示する。
        void displayDouble()
        {
            std::cout << "整数 " << n_ << " の2倍は " << makeDouble() << " です。" <<
std::endl;
            std::cout << std::endl;
        }

        // メンバ変数に格納する値を読み込む。
        void read()
        {
            std::cout <<  " 整数を入力してください。" << std::endl;
            std::cin >> n_;
            std::cout << std::endl;
        }
};
// クラスの定義　ここまで

int main()
{
    extendedInt extInt;  // クラスからオブジェクトを作成

    extInt.read();
    extInt.displayDouble();

    return 0;
}
```

次は，クラスに整数を受け取るコンストラクタ（初期化専用メソッド）を作成する。

Code05：コンストラクタ

主プログラムでは，extendedIntクラスの実体としてextIntオブジェクトを作成している。この生成時に，主プログラムから初期値を与えることができる。初期にを与えるための仕組みを **コンストラクタ** と呼ぶ。

このためには，クラスの中にコンストラクタと呼ばれる特別なメソッドを作成する。コンストラクタは，インスタンスを生成した時だけに実行される。コンストラクタはクラス名と同じ名前のメソッドである。

この例では，引数が1つのコンストラクタを作成した。引数の種類や数に応じた複数のコンストラクタを用意することができる。

```
#include <iostream>

class extendedInt
{
    int n_;

public:
```

```
    // constructor
    extendedInt(const int& n )
    {
        n_ = n;
    }

    int makeDouble()
    {
        return n_ + n_;
    }

    void displayDouble()
    {
        std::cout << "整数 " << n_ << " の2倍は " << makeDouble() << " です。" <<
std::endl;
        std::cout << std::endl;
    }
};

int main()
{
    int n;

    std::cout << "整数を入力してください。" << std::endl;
    std::cin >> n;

    extendedInt extInt(n);
    extInt.displayDouble();

    return 0;
}
```

Code05r1：宣言と定義の分離

メソッドの定義は，クラス外部にも記述できる。

外部に定義を書くため，スコープ解決演算子 "::" を使用する。`extendedInt::makeDouble()` のように書くことで，extendedIntクラスのmakeDoubleメソッドであることを表現する。

クラス内に記述したものはインライン関数となり，呼びだし部にインライン展開される。1行程度の短いメソッドは，インライン関数とする。（呼びだし負荷低減でき，コードの肥大化が小さいため。）

クラス外部に記述したものは，明示的にinline指定しなければ，インライン関数とはならない。複数行にわたる定義は，外部に記述することが多い。

OpenFOAMでは，多くのメソッドはクラス外部に書かれる。宣言部だけをヘッダファイル（*.H）に記述し，メソッドの定義はソースファイル*(.C)に書く。インライン関数は，インライン宣言を付けて，別のヘッダファイル(*I.H)に書かれている。

```
#include <iostream>
```

```
class extendedInt
{
    int n_;

public:

    //宣言のみ
    extendedInt(const int& n );

    int makeDouble();

    void displayDouble();
};

//定義はこちらに
extendedInt::extendedInt(const int& n )
{
    n_ = n;
}

int extendedInt::makeDouble()
{
    return n_ + n_;
}

void extendedInt::displayDouble()
{
    std::cout << "整数 " << n_ << " の2倍は " << makeDouble() << " です。" <<
std::endl;
    std::cout << std::endl;
}


int main()
{
    int n;

    std::cout << "整数を入力してください。" << std::endl;
    std::cin >> n;

    extendedInt extInt(n);
    extInt.displayDouble();

    return 0;
}
```

Code06：2つの型に2つのクラス

整数と同じようなものを、実数でも実現したいとする。

単純に考えると、整数用と実数用の2つのクラスを作る方法がある。

```
#include <iostream>

// 整数専用クラス
class extendedInt
{
    int n_;

public:

    int makeDouble()
    {
        return n_ + n_;
    };

    void displayDouble()
    {
        std::cout << "整数 " << n_ << " の2倍は " << makeDouble() << " です。" <<
std::endl;
        std::cout << std::endl;
    }

    void read()
    {
        std::cout << "整数を入力してください。" << std::endl;
        std::cin >> n_;
    }
};

// 実数専用クラス
class extendedFloat
{
    float f_;

public:

    float makeDouble()
    {
        return f_ + f_;
    }

    void displayDouble()
    {
        std::cout << "実数 " << f_ << " の2倍は " << makeDouble() << " です。" <<
std::endl;
    }

    void read()
    {
        std::cout << "実数を入力してください。" << std::endl;
```

```
        std::cin >> f_;
    }
};

int main()
{
    extendedInt extInt;
    extInt.read();
    extInt.displayDouble();


    extendedFloat extFlt;
    extFlt.read();
    extFlt.displayDouble();

    return 0;
}
```

ほとんど同じことを書いたクラスが2つ。内部に持つ型が異なるだけ。まとめられないの？templateを使ってまとめましょう。

Code07：テンプレートクラス1つで複数の型に対応

　先ほどの2つのクラスで，int または float だったところを，Type と名前をつけたテンプレートクラスで宣言します。クラスの定義内で，Type となっている部分は，mainから呼び出す際に定めたクラスとなります。

```
#include <iostream>
#include <string>

template<class Type>
class extendedType
{
    Type value_;
    std::string typeName_;

public:

    extendedType(const std::string& type)
    {
        typeName_=type;
    }

    Type makeDouble()
    {
        return value_ + value_;
    }

    void displayDouble()
    {
        std::cout << typeName_ << value_ << " の2倍は " << makeDouble() << " です。"
<< std::endl;
```

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

    void read()
    {
        std::cout << typeName_ << " を入力してください。" << std::endl;
        std::cin >> value_;
    }
};


int main()
{
    // template を int としてextendedTypeクラスのインスタンスを生成
    extendedType<int> extInt("整数");

    extInt.read();
    extInt.displayDouble();

    // template を float としてextendedTypeクラスのインスタンスを生成
    extendedType<float> extFlt("実数");

    extFlt.read();
    extFlt.displayDouble();

    return 0;
}
```

コンストラクタを作成しました。クラスからオブジェクト(インスタンス)を作る時に、型の名前を渡すようにしました。

mainでの宣言部分に注意してください。宣言時に、クラス側のtemplateが決定されます。

Code08：テンプレートのおかげで，別の型にも対応可能

これで、色々な型に対応できます。文字列型にも。ただし、演算子+が使える必要があります。

次のコードでは，クラス部分は変更せず，mainの中で文字列型を使っています。

```
#include <iostream>
#include <string>

template<class Type>
class extendedType
{
    Type value_;
    std::string typeName_;

public:

    extendedType(const std::string& type)
```

```
    {
        typeName_=type;
    }

    Type makeDouble()
    {
        return value_ + value_;
    }

    void displayDouble()
    {
        std::cout << typeName_ << value_ << " の2倍は " << makeDouble() << " です。"
<< std::endl;
        std::cout << std::endl;
    }

    void read()
    {
        std::cout << typeName_ << " を入力してください。" << std::endl;
        std::cin >> value_;
    }
};


int main()
{
        extendedType<int> extInt("整数");
        extInt.read();
        extInt.displayDouble();

        extendedType<float> extFlt("実数");
        extFlt.read();
        extFlt.displayDouble();

        extendedType<std::string> extStr("文字列");
        extStr.read();
        extStr.displayDouble();
}
```

Code09：演算子のオーバーロード（多重定義）

オペレータ(演算子)のオーバーロードを実装してみる。これまでに作成したクラスでは，インスタンス同士の和を求めることはできない。作成したクラス同士の足し算が可能となるように，プラス（+）演算子のオーバーロードを実現してみる。

```
#include <iostream>
#include <string>

template<class Type>
class extendedType
```

```
{
    Type value_;
    std::string typeName_;

public:

    extendedType(const std::string& type)
    {
        typeName_=type;
    }

    Type makeDouble()
    {
        return value_ + value_;
    }

    void displayDouble()
    {
        std::cout << typeName_ <<": " << value_ << " の2倍は " << makeDouble() << "
です。" << std::endl;
        std::cout << std::endl;
    }

    void read()
    {
        std::cout << typeName_ << " を入力してください。" << std::endl;
        std::cin >> value_;
    }

    void display()
    {
        std::cout << "typeName_ : " << typeName_ << std::endl;
        std::cout << "value_    : " << value_ << std::endl;
        std::cout << std::endl;
    }

    //overload of operator "+"
    const extendedType operator +(const extendedType obj) const
    {
       // value and type is added!
       extendedType<Type> tmp(typeName_);
       tmp.value_ = value_ + obj.value_;
       tmp.typeName_ = typeName_ + "+" + obj.typeName_;
       return tmp;
    }
};


int main()
{
    extendedType<int> extInt01("整数A");
```

```
    extInt01.read();
    extInt01.displayDouble();

    extendedType<int> extInt02("整数B");
    extInt02.read();
    extInt02.displayDouble();

    std::cout << "拡張整数Aと拡張整数Bとの和「+」" << std::endl;
    extInt01 = extInt01+extInt02 ;
    extInt01.display();
    extInt01.displayDouble();

    return 0;
}
```

Code10：継承

クラスの継承について考える。

extendTypeクラスを継承し，インスタンス名も保持するクラスとしてnamedExtendTypeというクラスを作る。（親クラス：extendType，派生クラス，子クラス：namedExtendType）

派生クラスでは，親クラスの機能を利用できる。

主プログラムでは，拡張したクラスを使用している。

```
#include <iostream>
#include <string>

template<class Type>
class extendedType
{

protected:  // this will be accesible from child-class.

    Type value_;
    std::string typeName_;

public:

    extendedType(const std::string& type)
    {
        typeName_=type;
    }

    Type makeDouble()
    {
        return value_ + value_;
    }

    void displayDouble()
    {
```

```
        std::cout << typeName_ <<": " << value_ << " の2倍は " << makeDouble() << "
です。" << std::endl;
        std::cout << std::endl;
    }

    void read()
    {
        std::cout << typeName_ << " を入力してください。" << std::endl;
        std::cin >> value_;
    }

    void display()
    {
        std::cout << "typeName_ : " << typeName_ << std::endl;
        std::cout << "value_    : " << value_ << std::endl;
        std::cout << std::endl;
    }

    //overload of operator
    const extendedType operator +(const extendedType obj) const
    {
       // value and type is added!
       extendedType<Type> tmp(typeName_);
       tmp.value_ = value_ + obj.value_;
       tmp.typeName_ = typeName_ + "+" + obj.typeName_;
       return tmp;
    }
};

//--------------------------
//
template<class Type>
class namedExtendedType
:
    public extendedType<Type>
{
    std::string instanceName_;

public:

    // Step 1
    namedExtendedType(const std::string& type)
    :
        extendedType<Type>(type)
    {
    }

    // step 2
    void setInstanceName(const std::string& name_)
    {
        instanceName_ = name_;
```

```
    }

    void display()
    {
        std::cout << "instanceName_ : " << instanceName_ << std::endl;
        extendedType<Type>::display();
    };

    //overload of operator
    const namedExtendedType operator +(const namedExtendedType obj) const
    {
       // value and type is added!
       namedExtendedType<Type> tmp(extendedType<Type>::typeName_);
       tmp.value_ = extendedType<Type>::value_ + obj.value_;
       tmp.typeName_ = extendedType<Type>::typeName_ + "+" + obj.typeName_;
       tmp.instanceName_ = instanceName_ + "+" + obj.instanceName_;
       return tmp;
     }
};

int main()
{
    //extendedType<int> extInt01("整数A");
    namedExtendedType<int> extInt01("整数A");
    extInt01.read();
    extInt01.displayDouble();
    extInt01.setInstanceName("FirstInt");
    extInt01.display();

    namedExtendedType<int> extInt02("整数B");
    extInt02.read();
    extInt02.displayDouble();

    std::cout << "拡張整数Aと拡張整数Bとの和「+」" << std::endl;
    extInt01 = extInt01+extInt02 ;
    extInt01.display();

    return 0;
}
```