

Stateパターン の書き方



スライドの内容

- ・前提知識と注意事項
- ・Stateパターンとは
- ・状態とは
- ・状態の呼び出し
- ・状態遷移
- ・状態間でのデータ共有
- ・難しいお話 (std::unique_ptrとインターフェイス)
- ・3種類の変数

0. 前提知識と注意事項

- ・クラス(C++)

→変数と関数を持ち、インスタンスを作れる

- ・仮想関数と継承(C++)

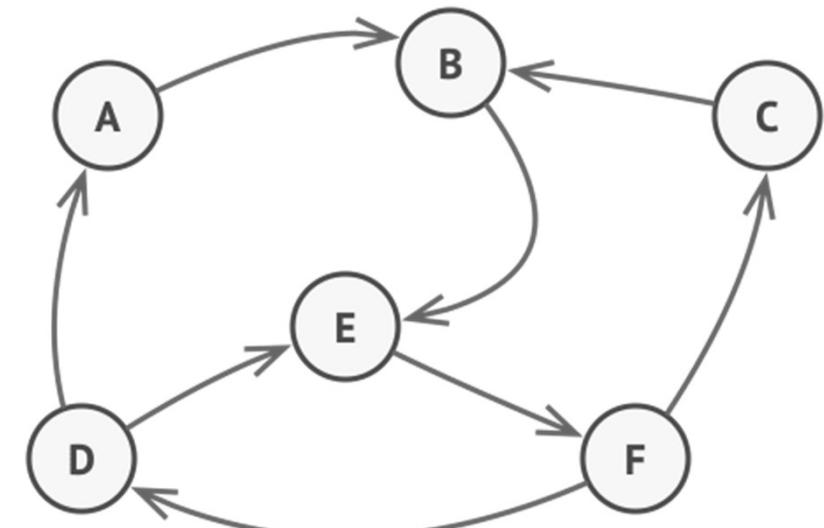
→難しい話で使います 知ってると楽しいかも？

- ・説明のために途中までは厳密には動かないコードを使います

・最後まで説明を聞くちゃんと動くコードになります

I. Stateパターンとは

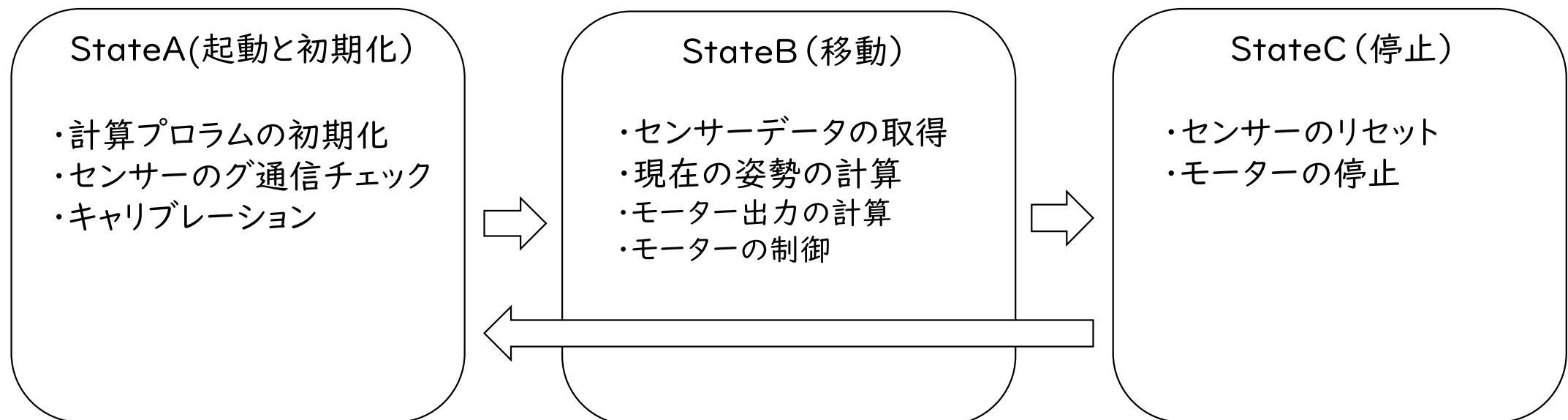
- ・有限個の状態を決まった条件で遷移するときに使える
- ・遷移するまで、同じ状態の処理を繰り返す



<https://refactoring.guru/ja/design-patterns/state>

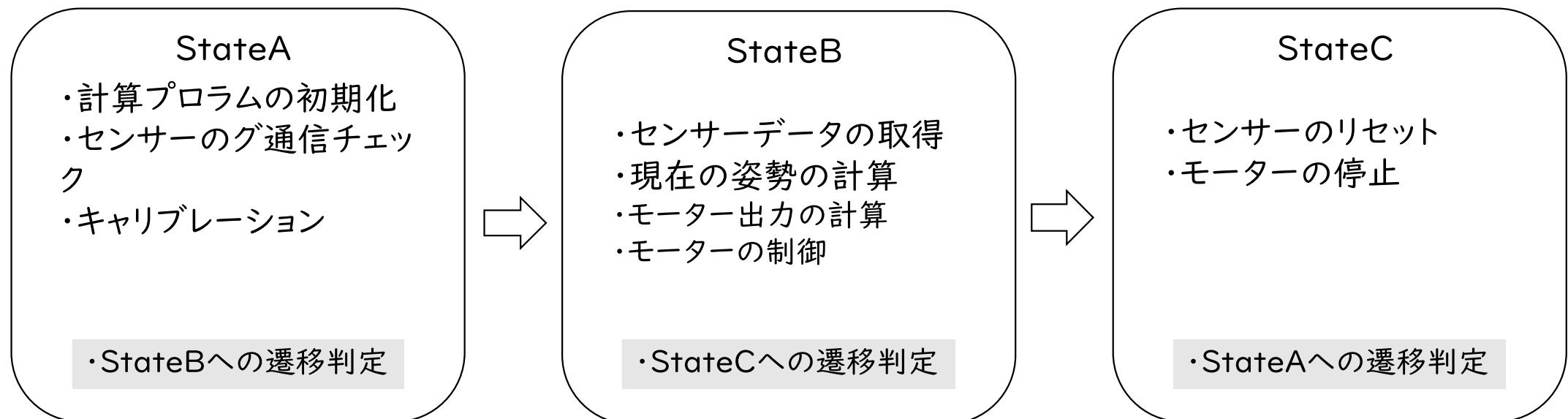
2. 今回使う状態について

- ・起動・移動・停止の3つの状態をもつ簡単なモデル
- ・それぞれをStateA・B・Cとする



3. 状態の実装(Ⅰ)

- 各状態に次の状態に遷移するかの判定を追加する
- 遷移の判定は通信結果やプロポのスイッチなどを使う



3. 状態の実装(2)

- 呼び出しのために各状態の処理をUpDate()にまとめる
- 各状態は基本的にクラスにする（理由はのちほど）

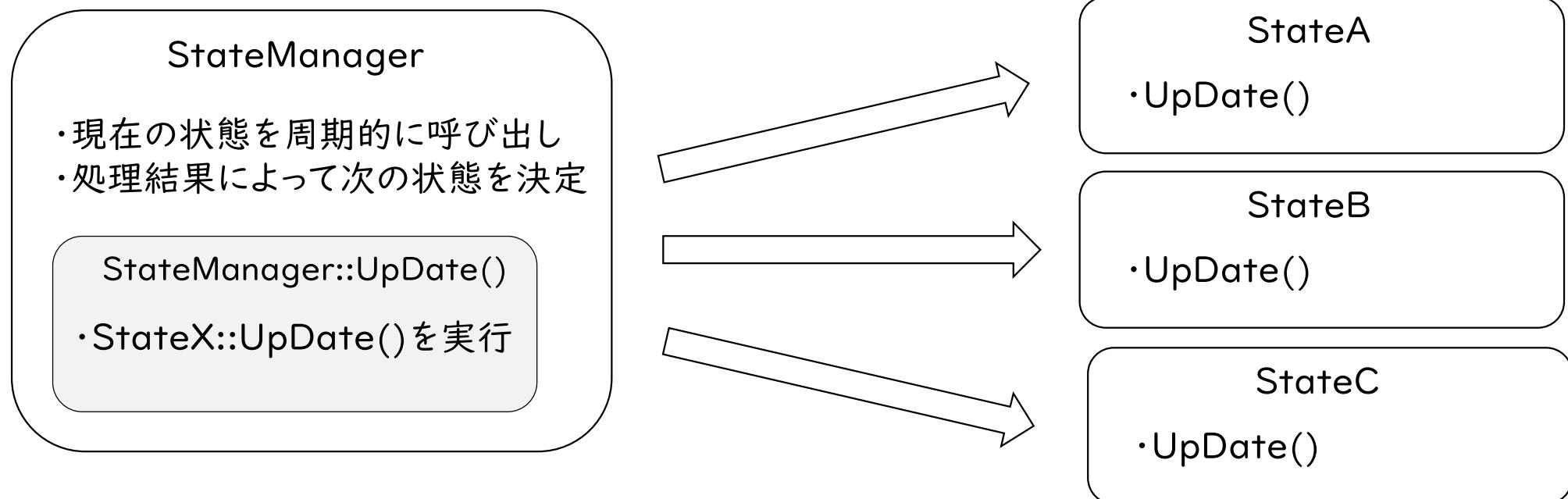
StateA::UpDate()

- 計算プログラムの初期化
 - センサーの通信チェック
 - キャリブレーション
- ・StateBへの遷移判定

```
class StateA{  
public:  
    void Update(){  
        // センサーの初期化やキャリブレーションなど  
        // StateBへの遷移条件  
        if(condition_to_stateB == true){  
            // StateBへの遷移（後述）  
        }  
    };
```

4. 状態の呼び出し

- StateManagerを作成し、呼び出しをする



4. 状態の呼び出し(2)

- 現在の状態によって呼び出す状態を変える
(状態の更新方法についてはのちほど)



※StateA::Update()は実際には動きません(StateAへの参照がないため)

```
class StateManager{  
public:  
    // StateXを呼び出す関数  
    void Update(){  
        // 現在の状態に適したUpdate()を呼び出す  
        if(current_state == 0){  
            StateA::Update();  
        } else if(current_state == 1){  
            StateB::Update();  
        } else if(current_state == 2){  
            StateC::Update();  
        }  
    }  
  
private:  
    // 現在の状態を保持する変数  
    int current_state = 0;  
};
```

5. StateManagerの呼び出し(1)

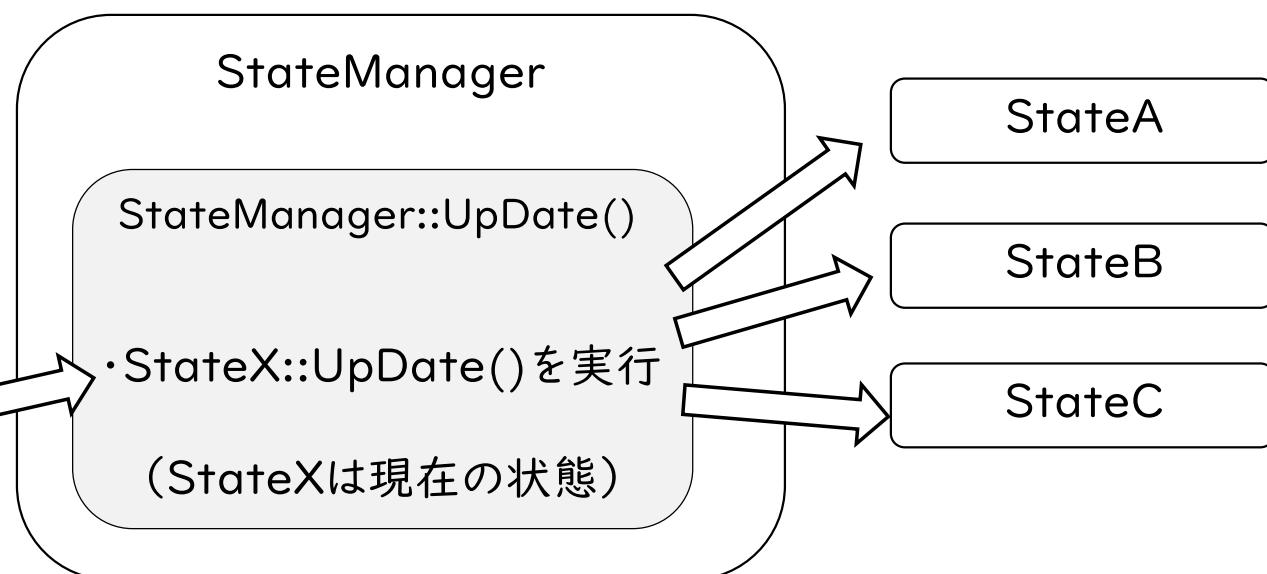
- main関数でStateManager::UpDate()を呼び出す
- 一定周期で呼び出せるような工夫が必要

```
// StateManagerのインスタンス作成
StateManager stateManager;

void loop(){
    // 現在時間の取得
    TimerUpdate(current_time, pretime);

    // 一定時間経過したら処理をする
    if(current_time > pretime + interval){

        // 状態の更新処理
        stateManager.Update();
    }
}
```



5. StateManagerの呼び出し(2)

- ・センサーの処理を変えてもStateAの1行を変えればいい
- ・バグの特定や修正がしやすい

```
// StateManagerのインスタンス作成
StateManager stateManager;

void loop(){
    // 現在時間の取得
    TimerUpdate(current_time, pretime);

    // 一定時間経過したら処理をする
    if(current_time > pretime + interval){
        // 状態の更新処理
        stateManager.Update();
    }
}
```

```
class StateManager{
public:
    // StateXを呼び出す関数
    void Update(){
        // 現在の状態に適したUpdate()を呼び出す
        if(current_state == 0){
            StateA::Update();
        } else if(current_state == 1){
            StateB::Update();
        } else if(current_state == 2){
            StateC::Update();
        }
    }

private:
    // 現在の状態を保持する変数
    int current_state = 0;
};
```

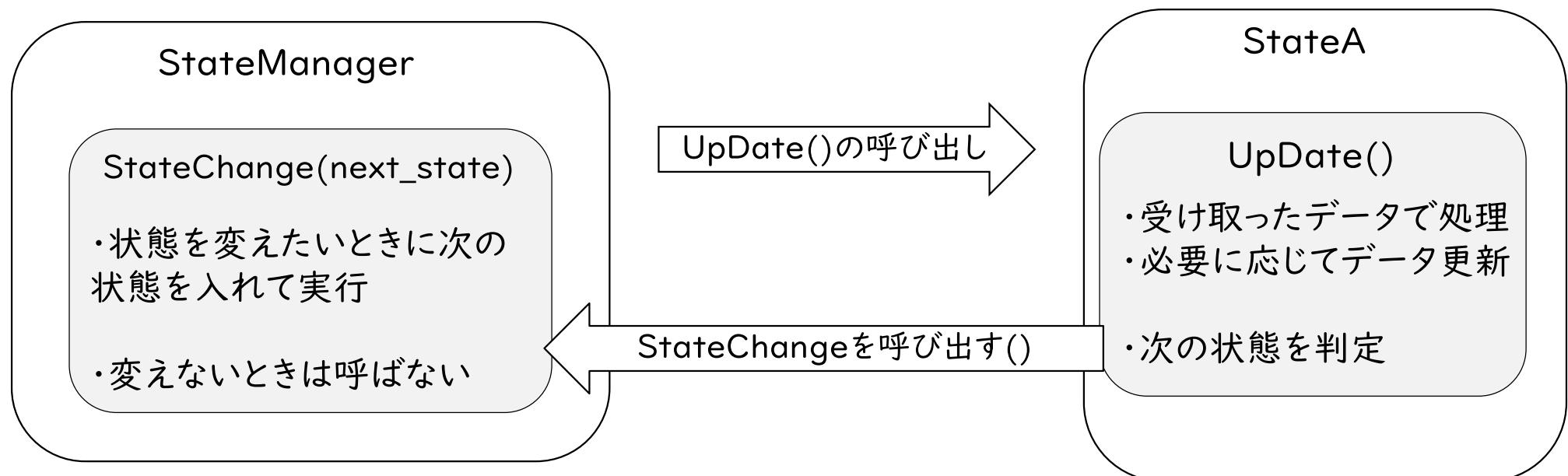
```
class StateA{
public:
    void Update(){
        // センサーの初期化やキャリブレーションなど
        // StateBへの遷移条件
        if(condition_to_stateB == true){
            // StateBへの遷移（後述）
        }
    };
};

class StateB{
public:
    void Update(){
        // センサーの初期化やキャリブレーションなど
        // StateCへの遷移条件
        if(condition_to_stateC == true){
            // StateCへの遷移（後述）
        }
    };
};

class StateC{
public:
    void Update(){
        // センサーの初期化やキャリブレーションなど
    };
};
```

6. 状態の変更(I)

- StateManagerに状態が変わることを伝える必要がある
- 状態変更用の関数をStateXから呼び出せるようにする



6. 状態の変更(2)

- 現在の状態を変更できる関数を作成する



```
class StateManager{  
public:  
    // StateXを呼び出す関数  
    void Update(){ ... }  
    void StateChange(int next_state){  
        // 状態の変更  
        current_state = next_state;  
    }  
private:  
    // 現在の状態を保持する変数  
    int current_state = 0;  
};
```

6. 状態の変更(3)

- 作成した関数を、各状態から呼び出す
- これによって状態を更新できる



```
class StateA{  
public:  
    void Update(){  
        // センサーの初期化やキャリブレーションなど  
        // StateBへの遷移条件  
        if(condition_to_stateB == true){  
            StateManager::StateChange(1); // StateBへの遷移  
        }  
    }  
};
```

6. StateManagerの呼び出し(4)

- 状態遷移(1回目のループ)

```
// StateManagerのインスタンス作成
StateManager stateManager;

void loop(){
    // 現在時間の取得
    TimerUpdate(current_time, pretime);

    // 一定時間経過したら処理をする
    if(current_time > pretime + interval){}

    // 状態の更新処理
    stateManager.Update();
}
```

```
class StateManager{
public:
    // StateXを呼び出す関数
    void Update(){

        // 現在の状態に適したUpdate()を呼び出す
        if(current_state == 0){
            StateA::Update();
        }
        else if(current_state == 1){
            StateB::Update();
        }
        else if(current_state == 2){
            StateC::Update();
        }

        void StateChange(int next_state){

            // 状態の変更
            current_state = next_state;
        }

private:
    // 現在の状態を保持する変数
    int current_state = 0;
};
```

```
class StateA{
public:
    void Update(){

        // センサーの初期化やキャリブレーションなど
        // StateBへの遷移条件
        if(condition_to_stateB == true){

            StateManager::StateChange(1); // StateBへの遷移
        }
    }
};

class StateB{
public:
    void Update(){

        // センサーの初期化やキャリブレーションなど
        // 遷移条件
        if(to_stateC == true){

            // StateCへの遷移（後述）
        }
    }
};
```

次の状態を指定してループ終了

6. StateManagerの呼び出し(5)

- 状態遷移(2回目のループ)

```
// StateManagerのインスタンス作成
StateManager stateManager;

void loop(){
    // 現在時間の取得
    TimerUpdate(current_time, pretime);

    // 一定時間経過したら処理をする
    if(current_time > pretime + interval){}

    // 状態の更新処理
    stateManager.Update();
}
```

```
class StateManager{
public:
    // StateXを呼び出す関数
    void Update(){

        // 現在の状態に適したUpdate()を呼び出す
        if(current_state == 0){
            StateA::Update();
        }
        else if(current_state == 1){
            StateB::Update();
        }
        else if(current_state == 2){
            StateC::Update();
        }

        void StateChange(int next_state){

            // 状態の変更
            current_state = next_state;
        }

private:
    // 現在の状態を保持する変数
    int current_state = 0;
};
```

現在の状態がStateBになっている

```
class StateA{
public:
    void Update(){

        // センサーの初期化やキャリブレーションなど

        // StateBへの遷移条件
        if(condition_to_stateB == true){

            StateManager::StateChange(1); // StateBへの遷移
        }
    }
};

class StateB{
public:
    void Update(){

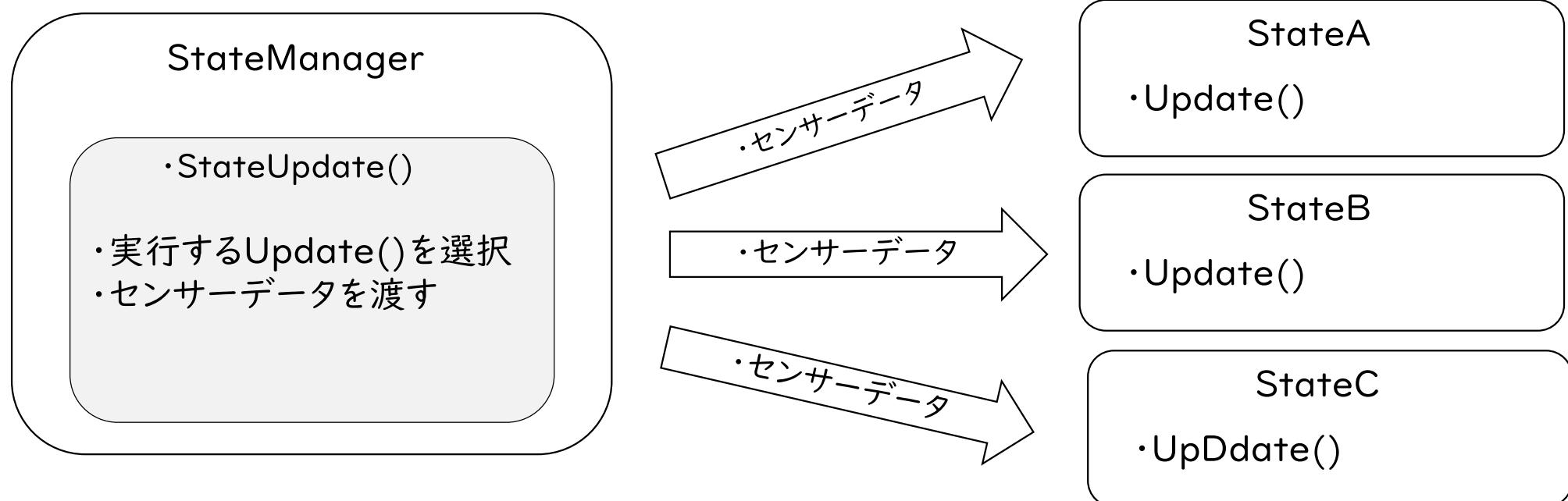
        // センサーの初期化やキャリブレーションなど

        // StateCへの遷移条件
        if(condition_to_stateC == true){

            // StateCへの遷移（後述）
        }
    }
};
```

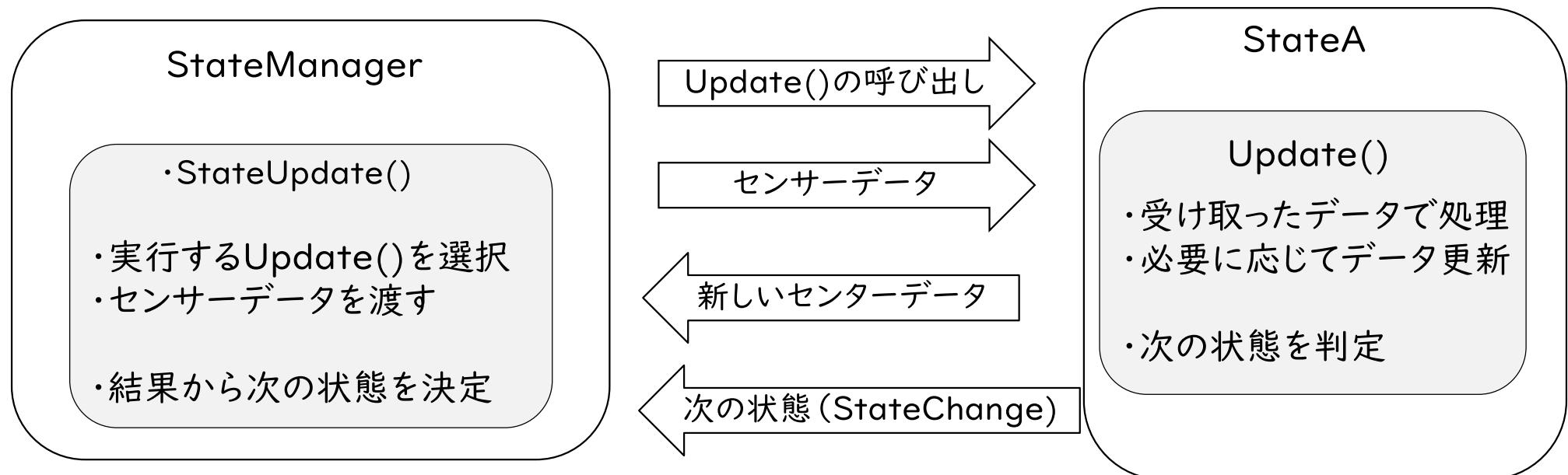
7. 変数の共有(Ⅰ)

- ・センサーデータなどは、状態間で共有しなければならない
- ・StateManagerにデータを仲介する役割を持たせる



7. 変数の共有(2)

- 使いたいデータをすべて渡してあげれば解決



7. 変数の共有(3)

- ・そのままデータを渡すと大変なことになる
- ・新しい変数ほしくなったら書き直し、.....

```
// 現在の状態に適したUpdate()を呼び出す
if(current_state == 0){
    StateA::Update(accel_x, accel_y, accel_z, gyro_x, gyro_y, gyro_z, mag_x, mag_y, mag_z, temperature, pressure, altitude, motor1, motor2, motor3, motor4);
}
```

7. 変数の共有(4)

- StateManager自身の情報を全部投げちゃえばいいのでは?
- このクラスのインスタンスへの参照を提供する(*this)

```
class StateManager{  
public:  
    // StateXを呼び出す関数  
    void Update(){ ... }  
  
    void StateChange(int next_state){ ... }  
  
    // センサーのデータを保持する変数  
    float accel[3];  
    float gyro[3];  
    float mag[3];  
    float temp;  
    float pressure;  
    float altitude;  
  
private:  
    // 現在の状態を保持する変数  
    int current_state = 0;  
};
```

センサーデータなどをまとめて渡せる

```
// StateXを呼び出す関数  
void Update(){  
    // 現在の状態に適したUpDate()を呼び出す  
    if(current_state == 0){  
        StateA::Update(*this);  
    }  
    else if(current_state == 1){  
        StateB::Update(*this);  
    }  
    else if(current_state == 2){  
        StateC::Update(*this);  
    }  
}
```

7. 変数の共有 (5)

- ・StateManager &managerという引数をもらっている
- ・これを使ってデータやStateChange()にアクセスする

```
class StateManager{  
public:  
    // StateXを呼び出す関数  
    void Update(){ ... }  
  
    void StateChange(int next_state){ ... }  
  
    // センサーのデータを保持する変数  
    float accel[3];  
    float gyro[3];  
    float mag[3];  
    float temp;  
    float pressure;  
    float altitude;  
  
private:  
    // 現在の状態を保持する変数  
    int current_state = 0;  
};
```

```
class StateA{  
public:  
    void Update(StateManager &manager){  
        // センサーデータを使う  
        SetSensorData(manager.accel);  
        SetSensorData(manager.temp);  
  
        // StateBへの遷移条件  
        if(condition_to_stateB == true){  
            // managerの参照があるので、ここからStateChangeを呼び出せる  
            manager.StateChange(1); // StateBへの遷移  
        }  
    }  
};
```

publicの中の関数や変数にアクセスできる

8. 難しいお話(1)

- ・このif文(switch文)を数字で管理するのわかりにくい
- ・呼ぶのはUpdate()だから、StateAの部分を変数にしたい

```
void Update(){  
    current_state::Update(*this);  
}
```

こうやってかけたらしあわせだね

```
// StateXを呼び出す関数  
void Update(){  
  
    // 現在の状態に適したUpDate()を呼び出す  
    if(current_state == 0){  
  
        StateA::Update(*this);  
    }  
    else if(current_state == 1){  
  
        StateB::Update(*this);  
    }  
    else if(current_state == 2){  
  
        StateC::Update(*this);  
    }  
}
```

※StateA::Update()が実行できないので、正しい手段での呼び出しをします

8. 難しいお話(2)

- ・変数型として扱いたいので、状態型の変数を作る
- ・ここでは、各状態の基底クラスにその役割を持たせる

基底クラスを作成

```
class StateInterface{  
public:  
    virtual void Update(StateManager &manager) = 0;  
};
```

各クラスを基底クラスの派生クラスにする

```
class StateA : public StateInterface {  
public:  
    void Update(StateManager& manager) override;  
};
```

8. 難しいお話(3)

- これによって通常の変数のように使えるようになった

インスタンスの変数を使えるようになった

```
std::unique_ptr<StateInterface> current_state;
```

代入はちょっと特殊（状態クラス名を入力できる）

```
current_state = std::move(std::make_unique<StateA>());
```

※`std::unique_ptr` はC++14で実装されています

変数として扱えるようになった

```
void StateManager::Update() {  
    // current_state が nullptr でないことを確認  
    if (current_state) {  
        // 現在状態の更新処理  
        current_state->Update(*this);  
    }  
}
```

8. 難しいお話(4)

- これによって通常の変数のように使えるようになった

StateManager::StateChange()は代入をする std::moveに変更

```
void StateManager::StateChange(std::unique_ptr<StateInterface> new_state) {  
    current_state = std::move(new_state);  
}
```

呼び出すときは、 std::make_unique<StateName>

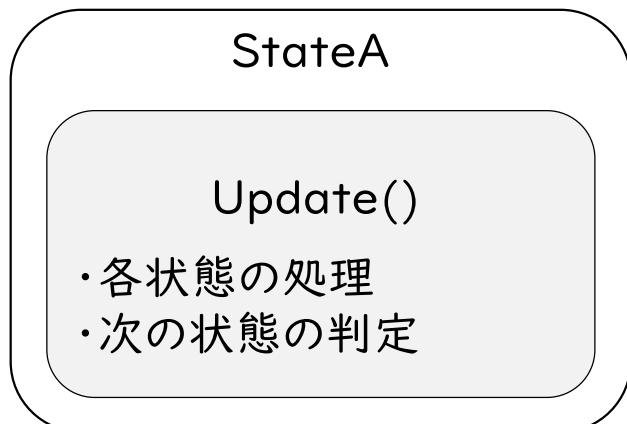
```
manager.StateChange(std::make_unique<StateB>());
```

※厳密にはstd::moveは所有権の変更をしています

8. 難しいお話(5)

- ・この実装の場合は、必要な状態クラスのインスタンスを生成
- ・次の状態に遷移するときに破壊する

1. StateAに遷移



StateAのインスタンスを生成

2. StateAの呼び出し



StateAにいる間は、繰り返し
同じインスタンスを使用

3. StateAからの遷移



StateAのインスタンスを破壊

8. 難しいお話(6)

- 内部の状態は、次の状態に遷移したら消滅する

1. StateAに遷移

StateA

Update()

- 各状態の処理
- 次の状態の判定
- loop_count = 0;

StateAのインスタンスを生成
カウンターは初期化

2. StateAの呼び出し

StateA

Update()

- 各状態の処理
- 次の状態の判定
- loop_count = 100;

StateAにいる間は
カウントが進む

3. StateAからの遷移

Update()

- 各状態の処理
- 次の状態の判定
- loop_count = 100;

StateAのインスタンスを破壊
カウンターも破壊される

9. 3つの変数

- ・使いたい範囲が最小になるように定義しよう

使いたい範囲	定義する場所	使い方	使用例
すべての状態	StateManager(public変数)	UpDate()実行時に渡す	センサーデータ 現在の状態
1つの状態で繰り返し	State(private変数)	通常のメンバー変数	ループカウント
1回の処理の間のみ	StateX::Update()	通常の関数内の変数	処理用の変数 データバッファー

10. サンプルコード

- ESP32用のコードと解説を書いたリポジトリがあります
- 部室にあるESP32に書き込むといい感じに動きます



https://github.com/aoi-netai/ESP32_StatePattern_Sample

Stateパターン の書き方

おわり

