関数ポインタ

ポインタは番地、コンピューター上のメモリの住所地を格納することが
できます。これまでは、主に、変数などの"情報"、Javaでいうところの
属性を格納するイメージが多かったかと思いますが、
よくよく考えると、クラス自体を格納して、メンバ関数などを呼び出すことが
できています。

ということは、
関数用のポインタを用意すれば、対応した好きな関数を切り替えて使用することが
できます。
これを関数ポインタと言います。

宣言の形式は、

```
戻り値の型 (*関数ポインタ名)(引数の型, 引数の型, ...);
```

書き方も用途もわかりづらいので、
SkyDomeの簡易的Stateデザインパターンに関数ポインタを適用した際に、
どのようなメリットがあるか確認していきましょう。

SkyDome.h

```
private:

    ～

    // 関数ポインタ
    void (SkyDome::*stateUpdate_)(void);
```

今回は、この関数ポインタに、状態別のUpdate関数を入れていく予定です。
Update関数は、戻り値の型も引数も、無し(void)で宣言されていますので、
このような書き方になります。
クラスのメンバ関数をポインタに入れる際には、
上記のようにクラス名の指定も必要です。

SkyDome.cpp

```
void SkyDome::Update(void)
{

    //// 更新ステップ
    //switch (state_)
    //{
    //case SkyDome::STATE::NONE:
    // UpdateNone();
    // break;
    //case SkyDome::STATE::STAY:
    // UpdateStay();
    // break;
    //case SkyDome::STATE::FOLLOW:
    // UpdateFollow();
    // break;
    //}

    // 更新ステップ
    (this->*stateUpdate_)();

}
```

SkyDome の Update関数は、
状態別に処理を振り分けているのですが、状態の種類が多くなると、
コードが縦に長くなります。

ところが関数ポインタを使用して、
実行する関数を切り替えることで、上のコードのように、
１行で済ますことができます。(this は自分のポインタ)

関数ポインタの切替処理については、
ChangeState = 状態切替時に記述すれば良いですので、

SkyDome.cpp

```
void SkyDome::ChangeStateNone(void)
{

    stateUpdate_ = &SkyDome::UpdateNone;

}

void SkyDome::ChangeStateStay(void)
{

    stateUpdate_ = &SkyDome::UpdateStay;

}

void SkyDome::ChangeStateFollow(void)
{

    stateUpdate_ = &SkyDome::UpdateFollow;

    transform_.pos = syncTransform_.pos;
    transform_.Update();

}
```

ブレイクポイントを貼って、キチンと状態ごとのUpdate処理が
実行されているか確認しましょう。

しかし、この関数ポインタ、生ポインタです。
できるだけ生ポインタを使いたくない方は、function を使用していきましょう。
関数ラッパーで、関数オブジェクトなどを格納できます。

SkyDome.h

```
#include <functional>

～

private:

    ～

    // 状態管理(更新ステップ)
    std::function<void(void)> stateUpdate_;
```

SkyDome.cpp

```
void SkyDome::Update(void)
{

    // 更新ステップ
    stateUpdate_();

}

void SkyDome::ChangeStateNone(void)
{
    stateUpdate_ = std::bind(&SkyDome::UpdateNone, this);
}

void SkyDome::ChangeStateStay(void)
{
    stateUpdate_ = std::bind(&SkyDome::UpdateStay, this);
}

void SkyDome::ChangeStateFollow(void)
{
    stateUpdate_ = std::bind(&SkyDome::UpdateFollow, this);
    transform_.pos = syncTransform_.pos;
    transform_.Update();

}
```

これで、生ポインタを排除することできました。

もし、もっとコードを短くしたければ、ChangeState関数で
状態別に処理を切り分けていたswitch文を削除することができます。

SkyDome.h

```
private:

    // 状態管理(状態遷移時初期処理)
    std::map<STATE, std::function<void(void)>> stateChanges_;
```

SkyDome.cpp

```
SkyDome::SkyDome(const Transform& syncTransform)
: syncTransform_(syncTransform)
{

    state_ = STATE::NONE;

    // 状態管理
    stateChanges_.emplace(
        STATE::NONE, std::bind(&SkyDome::ChangeStateNone, this));
    stateChanges_.emplace(
        STATE::STAY, std::bind(&SkyDome::ChangeStateStay, this));
    stateChanges_.emplace(
        STATE::FOLLOW, std::bind(&SkyDome::ChangeStateFollow, this));

}

void SkyDome::ChangeState(STATE state)
{

    // 状態変更
    state_ = state;

    // 各状態遷移の初期処理
    stateChanges_[state_]();

}
```

事前に、map配列に状態別の関数オブジェクトを格納しているのですが、
同じ要領で、Updateにも適用できますが、
毎フレーム、map配列から関数オブジェクトを取り出すことになるので、
その分、オーバーヘッドがかかります。

速度を優先するのであれば、控えた方が良いでしょう。

生ポインタ、スマートポインタ、関数ポインタ、
Stateデザインパターンの改良等々、色々取り組んできましたが、
あくまで、"プログラムの書き方" が変わっただけで、
極端に早くなったり、遅くなったりしているわけではありませんし、
効果的なゲーム演出・技術が実装されたわけでもありません。

今回学んだ書き方が、見づらいという人もいるでしょう。

とはいえ、色々な書き方がある、ということは知っておいて欲しいですし、
こういった言語知識への学習意欲が高い学生さんが欲しい！という企業様も
いらっしゃいます。

ゲーム開発の現場に入ると、もっとややこしい言語知識を学ぶ必要がありますし、
この先ずっと、そういった学びを続ける必要があります。

それでもゲーム業界を目指したいという方は、
もっともっと、勉強の時間を増やしてください。

資格試験であったり、言語知識であったり、ゲーム技術、他の言語、
学ぶことはまだまだたくさんあります。