シングルトンとやらについて

シングルトンパターン

シングルトンパターンについては、そんなに語ることもないと思うけれども、そのプログラムの中でインスタンスを１つに統一して使うデザインパターンのこと。

ゲームで言うと、ゲームの中でプレイヤーキャラを1体だけだして、2人目以上を登場させようとしたら自動的に出てこないように抑制できる仕組みです。

作ってみるよ

まずは普通にクラス作った場合を見てみよう（見る必要ない気もするけど）

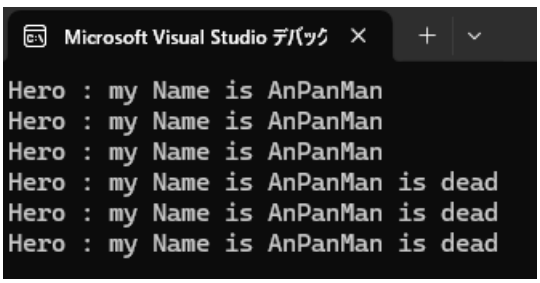
✔ソースコード

```
#include

class Hero
{
private:
 std::string name;
public:
 Hero()
  : name("AnPanMan")
 {
  std::cout << "Hero : my Name is " << name << std::endl;
 }
 ~Hero()
 {
  std::cout << "Hero : my Name is " << name << " is dead" << std::endl;
 }
};

int main()
{
 Hero hero;
 Hero hero2;
 Hero hero3;
 return 0;
}
```

✔実行結果



こんな感じで、いくらでも同じキャラのインスタンスが生成できます（まぁインスタンスってそういう話だったよね）。
次に、このワールド（世界）にアンパンマンは一人しか存在しないように変更してみる。

シングルトンパターンとやらを試してみる

✔シングルトンのつくりかた♡

0. （オプション）クラスを継承不可に指定
1. コンストラクタを private に指定する 2.静的なポインタ変数を宣言する

    - んで、こいつにインスタンスのアドレスを格納する
    - 後は、この変数だけが唯一のインスタンスになるように頑張る
2. 静的なポインタを初期化する
3. インスタンスを返す**静的な GetInstance 関数**を実装する
    - 静的な変数 == nullptr
        - ⇒新しくインスタンスを生成する
    - 静的な変数 != nullptr
        - ⇒静的な変数（のポインタか参照）を返す

0.（オプション）クラスを継承不可に指定

クラス宣言するときに、後ろに`final`キーワードを付加するとそのクラスは継承できない！とコンパイラに伝えることができるよ。

```
class Hero final
{
private:
//省略
};
```

1. コンストラクタを private に指定する

```
class Hero
{
private:
 std::string name;
public:
 Hero(){//省略
   }
   //省略
}
```

これを ↓　以下の様に変更する！これで、勝手にコンストラクタが呼べなくなる＝好きにインスタンスを生成することができなくなる。
じゃぁどうやってインスタンス作るの？ってことになるが、それは次。

```
class Hero final
{
//省略
private:
   std::string name_;
 Hero()
  : name("AnPanMan")
 {
  std::cout << "Hero誕生 : my Name is " << name_ << std::endl;
 }
   ~Hero()
 {
  std::cout << "Hero : my Name is " << name_ << " is dead" << std::endl;
 }
public:
//省略
};
```

2.静的なポインタ変数を宣言

お外から見えないところ＝privateなところに、`static Hero* s_hero_;`みたいな感じで、静的なメンバ変数を生成

```
class Hero final
{
private:
 //クラスの静的なポインタを宣言する
 static Hero* s_hero_; // これがスタティックなポインタ変数
 std::string name;
 Hero()
  : name("AnPanMan")
 {
  std::cout << "Hero誕生 : my Name is " << name << std::endl;
```

```
 }
 ~Hero()
 {
  std::cout << "Hero : my Name is " << name << " is dead" << std::endl;
 }
public:
//省略
};
```

3. 静的なポインタを初期化する

```
class Hero final
{
private:
//盛大に省略
};
//どこかグローバルスコープなところで
Hero* Hero::s_hero_ = nullptr;// 静的メンバ変数の定義
```

4. インスタンスを返す**静的な GetInstance 関数**を実装する

ここが一番のポイント！
nullptrで初期化された静的変数を`GetInstance`が呼ばれるたびに確認し、インスタンスが既に存在していれば、そのインスタンスを返す。
nullptrだった場合は、新しいインスタンスを作って、それを返す。
逆に言うと、nullptrだった時だけインスタンスが生成されるってだけ。

```
public:
 static Hero& GetInstance();// 静的な参照を返す関数（宣言文）
};

// Hero クラスのインスタンスを取得する
Hero* Hero::GetInstance()
{
 if (s_hero_ == nullptr) //インスタンスが無かったら
 {
  s_hero_ = new Hero(); //新しく生成
 }
 return *s_hero_;//インスタンスがあればそれを返す
}
```

実行してみようズ

プログラムの全体像と、実行のためのmain関数を以下に示す。

```
#include <iostream>

class Hero final
{
private:
 // Graphics クラスの静的なポインタを宣言する
 static Hero* s_hero_;
 std::string name_;
 Hero()
  : name_("AnPanMan")
 {
  std::cout << "Hero誕生 : my Name is " << name_ << std::endl;
 }
 ~Hero()
 {
  std::cout << "Hero : my Name is " << name_ << " is dead" << std::endl;
 }
public:
 static Hero& GetInstance();


};
```

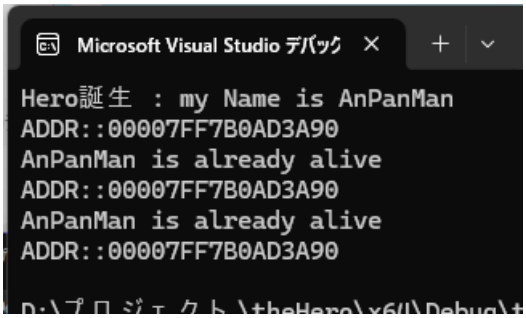
```
// Hero クラスのインスタンスを取得する
Hero& Hero::GetInstance()
{
 if (s_hero_ == nullptr)
 {
  s_hero_ = new Hero();
 }
 else
 {
  std::cout << s_hero_->name_ << " is already alive" << std::endl;
 }
    //ついでにインスタンスの持ってるアドレスを表示する！
 std::cout << "ADDR::" << std::hex << &s_hero_ << std::endl;
 return *s_hero_;
}

// 静的メンバ変数の定義
Hero* Hero::s_hero_ = nullptr;

int main()
{
 static Hero& hero = Hero::GetInstance();
 static Hero& hero2 = Hero::GetInstance();
 static Hero& hero3 = Hero::GetInstance();
 return 0;
}
```

実行結果

`GetInstance`した時に、ついでに、インスタンスのアドレスを表示するようにしてみた。 つまり、同じインスタンスが参照されていれば、同じアドレスが毎回帰ってくるし、毎回作っちゃっていれば、毎回違うアドレスになるはずだよね！



2回目以降の`GetInstance`では、初めに作られたインスタンスが参照されて返されていることがわかる！

でもね

一部では、結合性が云々とかでやたら嫌われてたりもするので、自分の属しているグループの流儀に習って使いましょう！

Revision #12
★Created 1 May 2025 04:12:22 by youe2
Updated 2 June 2026 17:30:59 by youe2