Ｃ＋＋ワンポイントレッスン

コンテナに格納されるクラスのデザイン

ソフトウエア研究会

コンテナに格納されるクラス

- データを保持するクラス(構造体)は、コンテナクラスに格納されて管理される。
- 最もシンプルなコンテナは、C++のプリミティブな配列だが、STLのvector等を使うのが良い。
- 格納されるデータクラスは、生成、初期化、破棄、コピー、スワップなどの基本的な機能を提供する。
- それらの機能の実装について考える。

クラスオブジェクトのコピー

- クラスオブジェクトのコピーには、コピーコンストラクタが呼ばれる。
- ＝オペレータを呼び出すことによって、コピーを行うことができるが、＝オペレータはオーバーライド可能なので、必ずしもコピーが行われるとは限らない。

✓ オブジェクトのコピー機能は、コピーコンストラクタで実装しよう。

✓ ＝オペレータを代入以外の動作で定義しなおすのは極力避けよう。

クラスオブジェクトの初期化

- 初期化の例

```
Hoge hoge(original);  // コピーコンストラクタが呼ばれる
Hoge hoge = original; // デフォルトコンストラクタが呼ばれた後に＝オペレータが呼ばれる(効率が悪い)。
Hoge* hoge = new Hoge(); // コンストラクタで例外がthorwされると生成されたオブジェクトがメモリリークする。
```

- ✓ 既存のクラスの複製を作成する場合は、コピーコンストラクタを使おう。
- ✓ ＝オペレータでの初期化は、見た目は1行だが実際はオブジェクトの生成と代入という2ステップになり、効率が悪い。
- ✓ コンストラクタは例外をthrowしてはいけない。なぜならば、newの処理で例外がthrowされた場合、オブジェクトをdeleteできなくなる。

コンストラクタの実装

- デフォルトコンストラクタの実装例

```
Hoge()
:  m_member(0) // メンバ変数は、かならず初期化する。
{
   m_member = 0; // NG!  =オペレータによって初期化しない！
}
```

✓ メンバ変数の初期化は、＝オペレータではなくそのオブジェクトのコンストラクタを呼び出す。

✓ ＝オペレータは例外を投げる可能性があるため、コンストラクタの中で＝オペレータを使用すると、例外を投げないことを保障できなくなる。

コピーコンストラクタの実装

- コピーコンストラクタの実装例

```
Hoge(const Hoge& src)
:  m_member(src.m_member) // m_memberのコピーコンストラクタで初期化する。
{
    m_member = src.m_member; // NG!  =オペレータによって初期化しない！
    *this = src;             // NG! 自身の=オペレータを呼んではいけない！

}
```

✓ メンバのコピーは、そのメンバのコピーコンストラクタを使用する

✓ ＝オペレータで代入してはいけない。例外安全でなくなる。

✓ 自身の＝オペレータやメソッドを初期化前に呼ぶことは避けよう。コンストラクタの処理が完了するまえにメンバ関数を呼ぶと予期せぬ事態が発生することがある。

コンストラクタで＝を使用した場合の不具合例

以下のように、char*で文字列を保持するクラスがあるとする。

```
class Str {
public: Str(const char* text) : m_text(strdup(text)) {} // textのコピーを作成して保持する
        Str(const Str& src) {
          *this = src; // =オペレータで初期化!
        }
       ~Str() { clear(); }  // デストラクタでメモリを解放
  void  clear() {   // 文字列のメモリを解放する
           if (m_text != 0) {
             free(m_text); m_text = 0;
           }
        }
  Str&  operator = (const Str& src) {
          clear();    // 以前の文字列を解放する
          m_text = strdup(src.m_text);  // srcから文字列をコピー
          return *this;
        }
  char* m_text;
}
```

- 鋭い方ならコピーコンストラクタのoperator=から呼ばれたclearが、初期化前のm_textを参照し、未定義な値のm_textをfreeしていることに気づくだろう。このように、コンストラクタでの初期化処理でクラスメソッドを呼ぶのは危険を伴う。

swapの実装

- 自身と他のオブジェクトを入れ替える機能を実装する
- swapは、決して例外を投げてはいけない。そのルールによって、さまざまな恩恵を得ることができる。

```
void swap(Hoge& hoge) {
    //swapの実装は、すべてのメンバについてswapを実行する。
    swap(m_member, hoge.m_member);
}
// class定義のスコープ外で、std:swapのオーバーライドを実装する。
class Hoge {
…
};
inline void swap(Hoge& a, Hoge& b) { a.swap(b); }
```

✓ C++の名前検索ルールにより、namespace外からでもHogeのswapはここで定義したswapが呼ばれるが、namespaceの外で定義しないと他の型のswapとの競合が起こるため、namespace外で定義したほうがよい。

std::swapのオーバーライド

- swapの実装を行わない場合、std::swapが呼ばれる。std::swapの定義は以下のようになっている。

```
template <typename T> inline void swap(T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

✓ swapを実装しないと、効率の悪いSTLのswapが呼ばれしまう。3回の代入よりも効率よくswapができる場合は、は必ず実装しよう。

＝オペレーターの実装

- ＝オペレータをswapを使って実装する。
- swapを使うことで、シンプルで例外安全なコードを書くことができる。

```
Hoge& operator = (Hoge src) {
    swap(src);
    return *this;
}
```

✓ swapを使うことで、わずか3行で＝オペレータの実装は完了する。

＝オペレーター実装の解説

```
Hoge& operator = (Hoge src) {
  swap(src);
  return *this;
}
```

Step-1
const Hoge&ではなく、Hogeとすることでsrcのコピーを作成する。コピーの作成は、コピーコンストラクタが呼ばれる。

Step-2
Step-1でコピーしたsrcと自分自身を入れ替える。

Step-3
srcに格納された以前の自分自身は、ここで破棄される。

swapを使用した再初期化

- インスタンスを初期化する、clear()というメソッドの定義を行う。

```
class Hoge {
…
void clear() {
    swap(Hoge());
}
```

✓ swapを使用してデフォルトコンストラクタによって生成されたオブジェクトと自分自身を入れ替える。これにより、シンプルで例外安全なclear()の実装が可能になる。以前のインスタンスは、Hoge()の一時オブジェクトと入れ替わった直後に破棄される。

swapを使うことのメリット

- swapというクラスの中身を入れ替える機能を実装することで、さまざまなメリットが生まれる。

✓ 代入やクリアの処理が安全かつシンプルに実装可能。

✓ 初期化やコピーの作業がコンストラクタに集約され、破棄はデストラクタを使用するため、コードの二重化がなくなり、デバッグ効率の良いコードが書ける。

✓ std:swapをオーバーライドすることにより、パフォーマンスを向上させることができる。

✓ ＝オペレータの実装が、コンパイラによって最適化されやすくなる。以下のようなコードも無駄な処理が最適化によって省略されることが多い。

```
Hoge hoge = src; // operator=による初期化と代入
```

コンテナに格納されるクラスの留意点

- デフォルトコンストラクタ、コピーコンストラクタ、operator=, clear, swap等の基本機能を実装する。
- インスタンスのコピーはコピーコンストラクタに集約させ、できる限り実行コストを抑える。
- コンストラクタ、デストラクタ、swapは決して例外を投げてはいけない。
- std::swapをオーバーライドする外部関数を定義する。

Slide7の不具合サンプルの正しい実装例

```
namespace hoge {
class Str {
public:
        Str(const char* text = 0) : m_text(text == 0 ? 0 : strdup(text)) {} // textのコピーを作成して保持する
        Str(const Str& src) : m_text(src.m_text == 0 ? 0 : strdup(src.m_text)) {}
       ~Str() {
          if (m_char != 0) free(m_text);
        }
  void  clear() { swap(Str()); }
  Str&  operator = (Str src) {
          swap(src);
          return *this;
        }
  void  swap(Str& src) {
         std::swap(m_text, src.m_text);
        }
private:
        char* m_text;
}
} // namespace hoge
namespace std {
inline void swap(Str& a, Str& b) { a.swap(b); }
}
```

- Slide7の実装と見比べてみて、どちらが効率的でデバッグが容易か検証してみよう。std:swapのオーバーライドをnamespace外で行っていることに注意。namespace内だとオーバーライドにならず、std::swapのlookupに失敗する。C++標準化委員会はstdネームスペースに何かを追加することを禁止しているが、外部関数のオーバーライドは例外的に認める方向にあるらしい。