メンバ変数のメンバ関数内でのリソース管理

高橋晶 (Akira Takahashi)
id:faith_and_brave
@cpp_akira

Boost.勉強会 #8 大阪 2012/02/11(土)

RAII

- C++には、RAII(Resource Acquisition Is Initialization：リソース確保は初期化である)というイディオムがある。
- 簡単に言えば、確保したリソースはデストラクタで自動的に解放する、というもの。

```
void f()
{
    File file;
    file.open("a.txt");

    if (!file.write("xxxxxxxx")) {
        return; // 途中で抜けてもファイルは閉じられる
    }
} // ファイルが閉じられる
```

- これはローカル変数には非常に有効。

Boost C++ Libraries

メンバ変数のリソース管理

- メンバ変数の寿命がクラスと同じでよければ、RAIIが有用。

```
class X {
    std::vector<User> users;

public:
    ~X()
    {
    } // ここでusersが解放される
};
```

ケーススタディ

ボタンの設計を考えてみよう

```
class Button {
public:
    void down(); // ボタン押した
    void up();   // ボタン離した

    bool is_down() const;         // ボタン押されてる？
    bool in_rect(Point p) const; // ある点が範囲内かを判定
};
```

down(), up()関数ではそれぞれ、ボタンの押下状態によって
表示画像を切替える処理が入っているとする。

ボタンが離されたら確実にup()関数を呼びたい。
どうするか？

ケーススタディ

ボタンを包含する画面クラスはこんな感じになるでしょう。

```
class View {
    Button ok_button;
public:
    void on_down(Point p)
    {
        if (ok_button.in_rect(p)) // 範囲内なら押す
            ok_button.down();
    }

    void on_up()
    {
        if (ok_button.is_down()) { // 押されていたら離して押下処理
            ok_button.up();
            on_ok_button();
        }
    }

    void on_ok_button() {} // OKボタンが押された
};
```

ケーススタディ

この設計だと、ボタンが複数あると破綻する。

```
class View {
    Button ok_button, cancel_button;
public:
    …
    void on_up()
    {
        if (ok_button.is_down()) { // 押されていたら離して押下処理
            ok_button.up();
            on_ok_button();
            return; // 余計な処理はしないで終了
        }
        if (cancel_button.is_down()) {
            cancel_button.up(); // ボタンが離されない可能性がある
            on_cancel_button();
            return;
        }
    }
    …
};
```

ケーススタディ

離されたら、全てのボタンが確実にup状態になるようにしたい。

Boost.ScopeExitを使おう。

```
void on_up()
{
    BOOST_SCOPE_EXIT((&)) { // スコープを抜けたら全てのボタンを離す
        ok_button.up();
        cancel_button.up();
    };
    if (ok_button.is_down()) { // 押されていたら離して押下処理
        on_ok_button();
        return; // 全てのボタンが離される
    }
    if (cancel_button.is_down()) {
        on_cancel_button();
        return; // 全てのボタンが離される
    }
} // 全てのボタンが離される
```

これで、メンバ変数が、特定のメンバ関数の抜けた際に、指定した処理を確実に行わせることができるようになった。

Boost.ScopeExit

Boost.ScopeExitは、スコープを抜ける際に実行されるブロックを記述するためのライブラリ。

```
int value = 0;

void f()
{
    value = 1;

    BOOST_SCOPE_EXIT((&)) { // スコープを抜ける際に実行されるブロック
        value = 3;
    };

    value = 2;
}

f();
assert(value == 3);
```

Boost.ScopeExitがやっていることは、デストラクタでそのブロックを実行するクラスとそのオブジェクトを自動生成してるだけ。

Scoped Lockingパターン

Scope Exitの目的特化した例として、Scoped Lockingパターンというのがある。

```
class Logger {
    Mutex mutex_;
public:
    void write(const std::string& s)
    {
        LockGuard<Mutex> lock(mutex_); // ロックする
        …
    } // スコープ抜けたらロック解除
};
```

複数ヶ所でロックされる可能性のあるミューテックスを、
それぞれの処理が終わった段階で確実にアンロックする。

Boost C++ Libraries

まとめ

- RAIIはとても便利だが、メンバ変数をメンバ関数内で自動的に解放処理したい場合には、もう一つのRAIIを用意する必要がある。
- Boost.ScopeExitはこの手間を減らしてくれる。
- Scoped Locking Patternのように、いろいろな個所で同じことをするならScopeExitを直接使うのではなくライブラリ化しよう。