# C++ のオベンキョ 第 1 回 リソース管理

@meryngii

2014/04/17

## 1 リソース管理

C++ は，Java や C#や D などに備わっている GC が存在しません．プログラマは，コンストラクタとデストラクタを利用してリソースを管理します．

- グローバル変数
  - プログラムが開始すると construct され，終わると destruct される．
  - グローバル変数をたくさん使うプログラムは，設計がゴミ．
  - デザインパターンの言葉でラップされている Singleton も結局グローバルにすぎず，使い過ぎは同様にゴミ設計といえる．
  - 関数内に static と書いた変数は，construct のタイミングを制御できるグローバル変数．やはり使いすぎは NG.
- 自動変数
  - 関数の中で普通に定義した変数.
  - スコープから抜けると destruct される．
  - ポインタや参照として他の関数に渡す場合は要注意.
- ヒープ変数
  - new で定義した変数.
  - delete しないとメモリリークするし，デストラクタが呼ばれないので謎のバグの危険あり．
  - 好きなタイミングで確保・開放できるが，その処理は重い．
  - 現代的 (C++11 以降) にはすべてスマートポインタ (`shared_ptr` or `unique_ptr`) で管理する．`delete` を直接書く C++ プログラマは雑魚.

## 2 一時オブジェクトの寿命

一時オブジェクトとは，式の結果として得られる値のことです．C++ で作られる一時オブジェクトの寿命は，「文の終わりまで」(End-of-statement）と規定されています．

例えば，こんな使い方は NG です．

```
1 const std::string& str = "abc"; // 見えない型変換コンストラクタで一時オブジェクトが生成される
2 std::cout << str; // ERROR: strが指す一時オブジェクトはすでにdestructed
```

## 3 仮想デストラクタ

ポリモーフィズムを利用する場合，基底クラスには仮想デストラクタを挿入することが強く推奨されます．基底クラスのポインタを delete したときに，そのインスタンスが実は派生クラスのものであることまではチェックされない

からです.

```
class Base {
public:
    Base() { }
    /*virtual*/ ~Base() { }

    virtual void show() { std::cout << x << std::endl; }

private:
    int x;
};

class Derived
    : public Base
{
public:
    Derived() { }
    ~Derived() { }

    virtual void show() { std::cout << y << std::endl; Base::show(); }

private
    int y;
};

void g(Base* b)
{
    b->show();

    delete b; // ERROR: 基底クラスのつもりでdeleteする
}

void f() {
    Derived* d = new Derived();
    Base* b = d;
    g(b);
}
```

上の例はあまりに実践的でない例ですが，下のようにスマートポインタを使う例でも同様の問題は残ります.

```
// 戻り値は共有された後，Baseとして破棄される
std::shared_ptr<Base> h() {
    return std::shared_ptr<Base>(new Derived());
}
```

## 4 shared_ptr と unique_ptr の違い

基本的に，複数のクラス間で共有したいものは shared_ptr，一つのものからしか参照しないものは unique_ptr を使います.

shared_ptr の使い方は簡単なので省略.

コピー毎に参照カウントの上げ下げのコストが生じることに注意してください．なので，引数として受け取るときは const 参照で受け取ると better です.

```
void f(const std::shared_ptr<A>& ptr) {
    // ...
}
```

また，自分で new したポインタを渡す場合は，std::shared_ptr 内部で参照カウント用の整数を置くメモリを別に new しています．アロケーションの手間を減らすために，make_shared を使用することが推奨されます.「今時の C++ プログラマは，delete どころか new すら書かない」ということが言われるのはこれが理由です.

```
std::shared_ptr< std::vector<double> > ptr =
    std::make_shared< std::vector<double> >(10000, 1.0); // 10000個のdoubleを1.0で初期化

std::shared_ptr< std::vector<double> > ptr2 = ptr;

// もちろんauto使ってもいいよ
auto ptr3 = std::make_shared< std::vector<double> >(10000, 1.0);

auto ptr4 = ptr3;
```

unique_ptr も，move セマンティクスを使わない限り非常に簡単に使えます．なので使い方は略．std::move や rvalue reference については，また今度．

```
void f() {
    std::unique_ptr< std::vector<double> > ptr(new std::vector(10000, 1.0));

    //auto ptr2 = ptr; // NG: uniqueだからコピーはできない

    // スコープから抜けたら破棄される
}
```

make_unique*なるものを作っておくと，auto と相性が良いし便利．ただし，性能面でのメリットはない．C++14 では標準に入る予定．

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}

// ...

auto ptr = make_unique< std::vector<double> >(10000, 1.0);
```

* http://stackoverflow.com/questions/17902405/how-to-implement-make-unique-function-in-c11