インタラクティブゲーム制作
ゲームプログラミング講座
第 4 回資料

竹内 亮太

(2012/5/23)

4 変数の寿命とメモリ管理

4.1 変数が生まれて散りゆくまで

今まで皆さんは、変数の有効範囲として**スコープ**という存在があったのはご存じかと思います。変数はで囲まれた範囲でしか扱えない、というものです。

今日はここから更に踏み込んで、プログラムの裏で何が起きているのかの一端を皆さんに理解してもらいます。スコープは単に有効範囲を規定するだけでなく、変数の生と死を司る境界線としても機能しているのです。

```
void func(void)
{
    int hoge = 0; // ここで変数 hoge が作られる

    if(適当な条件式) {
     int hogehoge = 0; // ここで変数 hogehoge が作られる
    } // この時点で変数 hogehoge が破壊される

    return; // この時点で変数 hoge が破壊される
    // スコープエンドか、return や break で処理を抜ける時点で破壊が起きる
}
```

main() 関数の直下に作った変数の場合、プログラムを起動して宣言を通過する時点で生成され、プログラムが終了するまで内容を保持できることになります。なので、ゲーム全編を通じて必要な変数は main() 関数で宣言するのがおすすめです。

ちなみに参照やポインタ変数の場合、スコープを抜けても元の変数には影響を及ぼしません。参照もポインタも、既に別の場所で宣言してある変数を指し示しているだけなので、スコープアウトで内容が破壊されることはありません。

4.2 コンストラクタとデストラクタ

さて、int や double などの基本型では単に変数が生まれて消えゆくだけですが、クラス型の場合はこのタイミングで重要なイベントが発生します。それが、**コンストラクタとデストラクタの呼び出し**です。実験のために、次のようなクラスを作ってみましょう。

```
#include <iostream>

class ScopeChecker {
public:
    ScopeChecker(void)
    {
     std::cout << "こうして俺はこの世界に生まれた。" << std::endl;
    };
    ~ScopeChecker()
    {
     std::cout << "そして俺は世界から抹殺された。" << std::endl;
    };
};
```

ちなみにメッセージはなんでもいいです。このクラスの変数を、色んなスコープに作って挙動を確かめてみましょう。変数が出来た時点でコンストラクタが走り、破壊される時にデストラクタが走る。それを確認できればいいです。

```
    ScopeChecker sc;
```

こんな感じで、変数名はなんでもいいのであちこちに作ってみます。そうすると、同じメッセージばかりで何がどう対応しているのか、分からないと思います。そこで、作ったばかりのこのクラスを、実体生成時に名前付けできるようにしてみます。

```
#include <iostream>
#include <string>

class ScopeChecker {
private:
    std::string name;
public:
    ScopeChecker(const std::string argName)
    : name(argName)
    {
     std::cout << "俺の名は" << name;
     std::cout << << "…たった今生まれたところ
```

```
さ。" << std::endl;
    };
    ~ScopeChecker()
    {
     std::cout << "俺の名は" << name;
     std::cout << "…たった今、世界から抹殺されたところさ。";
     std::cout << "あばよ。" << std::endl;
    };
};
```

さあ、ちょっと謎記法が増えてきましたね？一番気持ち悪いのは： name(argName) ではないかと推測するのですが、どうでしょうか。引数付きのコンストラクタ、というのも最初のうちは理解しづらいかもしれません。この改造で、ScopeChecker クラスの変数を作る時には必ず文字列を渡して名前付けをしなくてはいけなくなりました。この場合は次のようにしないと実体生成できません。

```
ScopeChecker sc("ほげぇ");
```

例によって、名前はなんでも構いません。C++では**変数名 (変数生成時に渡す値)** という記法を用いる場面がいくつかあります。引数付きのコンストラクタを扱う時、またコンストラクタを作る際にクラスのメンバ変数に初期値を渡す際には、このような書き方をします。覚えておきましょう。

コンストラクタとデストラクタをきちんと作り込むことで、他人に渡して使ってもらうためのクラスがかなり作りやすくなります。究極は「ああ、そのクラスの実体作ればそれで終わりだよ」という境地に達しますが、そこまで行かずとも、準備と後片付けの予備忘れがなくなる、というのは大きなメリットと言えるでしょう。

ちなみにクラスのメンバ変数の寿命は、実体の寿命と同じです。試しに ScopeChecker を適当なクラスのメンバに持たせてみればよいでしょう。ただし、名前を付けるのにちょっと工夫が要ります。ScopeChecker の作り方にヒントがあるので、試してみて下さい。

4.3 動的なメモリ管理

さて、ここまでは C++のなかでもイージーモードな機能についてしか触れませんでした。十分ややこしかったかもしれませんが、まだイージーです。ここから先は、C++が忌み嫌われる要因となるような世界に突入します。覚悟しましょう。

4.3.1 普通の変数の限界

C++での配列は簡単に作ることができます。

```
int         iDs[10];
fk_Vector   posArray[5];
```

Java とかに比べればずっと簡単です。が、これには凄まじい落とし穴があります。**[] の中に直書きした個数分しか作れないのです！**これはとてつもない欠点です。Java の場合は変数を個数の指定に使うことができましたが、C++では許されません。もし今までそういうコードを書いてうまく動いてたとしても、それはたまたまです。絶対やっちゃならんコーディングの 1 つです。

実践的なプログラムにおいては、ある時点でいくつ分の配列を用意しなければならないかは、その時点になってみないと分からないケースの方が多いです。そういう場合はどうするか？

```
int     *iDs = new int [10];
```

はい、一気に Java 的になりましたね。この時の [] の中には変数を使うことができます。そしてポインタ型変数に課せられた新たな役割にも注目しましょう。今までは既存の変数のアドレスを記憶するのにしか使っていませんでしたが、**new で作り出した変数 (配列) の場所を記憶する**のにも、このポインタを使います。むしろこちらの方が本来の用途とも言えます。こうやって作った配列のことを**動的な配列**といいます。

ここでは配列を例に取り上げましたが、単品の変数を作る際にも new を使う場合が多いです。ここでも Java 的な書き方になります。

```
// 今までの作り方だと
fk_Model    normalModel;
// new な作り方だと
fk_Model    *pModel = new fk_Model();
```

実に JavaJava しいですね。まぁJava の方が後発ですけ

どね。状況によってはこういう作り方が要求されるケースが出てきます。具体的には、スコープを抜けても死んで欲しくない変数を作る時などですね。ScopeChecker クラスを使って、それを確かめてみましょう。

4.3.2 変数 (配列) の作り方・片付け方

まとめます。C++で変数を作るには静的な方法と動的な方法の 2 種類があります。静的な変数は、

- 今まで通り普通に「型名 変数名;」で宣言する
- 宣言してある**スコープに処理が進入した瞬間に**実体が生成される
- **スコープから処理が抜けたら自動的に**実体は破壊され、片付けられる
- 配列を作ることもできるが、その場その場でサイズが変化したり、ド派手なサイズのものは作れない

という特徴があります。スコープに入ったとき、抜けたときにそういうことが起きているということを覚えておいてください。

翻って動的な変数は、

- 「new 型名;」で実体を作成する
- new した結果、実体ができた場所のアドレスが返ってくるので**それをポインタ型の変数で捕まえて、ポインタを通じて利用する**
- 何らかのタイミングで勝手に片付けられることはない
- **だから自分で片付けなくてはならない**
- 片付けるには「delete new したポインタ;」とする
- 配列を作った場合には「delete [] new で作った配列のポインタ」とする
- **うかつな delete は死を招く (まだ利用しているものを delete したりとか)**
- **かといって delete しないで放置しているとプログラムが、OS が死ぬ**

という素敵な特徴があります。

なんだろう、私は事実しか述べていないのに C++がとても恐ろしい言語に思えてきました。これが C++が「テクニカルなマニュアル車だ」と言った理由です。Java はここらへん、とてもオートマチックです。Java には delete というものがありませんからね。その代わり処理速度がかなり犠牲になっています。ゲームというカリカリなアプリを作るには、このくらいやらないといかんっちゅうことです。

私の経験則ですが、まだ試作段階の時は無理な動的メモリ管理には手を出さなくてもいいです。たとえばマップデータを読み込んで、それを元に空間を構築するだとか、そういうことをやり始めるようになったら new,delete の出番だと思っていいでしょう。

4.3.3 STL のすすめ

とはいえです。配列の制限は試作段階においても鬱陶しい場合があります。もうちょっと使いやすい配列はないものか、とお嘆きのあなた。そんなあなたにぴったりなのが STL の「vector」というクラスです。

fk_Vector とごっちゃになりがちなんですが、別物なので注意してください。これは「可変長配列」と呼ばれる代物で、

- どんな型のものでもしまえる
- 最初にサイズを決めてもいいし、決めなくてもいい
- 好きなタイミングでサイズを変更できる
- 配列の最後に新しい要素を付け足していったりできる
- 配列を作ったスコープから抜ければ勝手に消滅してくれる

と、いいことずくめです。素晴らしいですね。

詳しい使い方なんですが、授業資料ページに丸投げスパイラルします。そっち見て勝手にやってください。

ただし、FK のクラスオブジェクトを配列にしようとした場合は注意が必要です。正確には**配列の各要素のポインタを利用する場合**がデンジャラスです。

配列の各要素のポインタは `&array[3]` のようにすることで取り出せます。それをその場で使う分には構わないのですが、fk_Model のように、他のクラスオブジェクトにポインタを渡したり、他から受け取ったりするような物は危険です。具体的には、fk_Scene の entryScene にポインタを渡して登録したり、または逆に setShape で形状データのポインタを受け取ったりする場合、vector 配列の要素に&を付けた物を渡すと大惨事になります。

これは vector 配列のメモリ上の位置が、かなり頻繁に入れ替わるために起こる現象です。可変長配列を実現するために、裏でかなり高度なことをやっているんですね。じゃあどうすればいいかというと、**実体の配列ではなく、ポインタの配列にする**が現実的でお手軽です。

- `vector<fk_Model *> modelPArray;` のように配列を作る
- `modelPArray.push_back(new fk_Model);` のように、一つ一つの要素を new して配列にたたき込む
- `modelPArray[3]->getPosition();` のようにして利用
- 最後は一つ一つの要素を delete して始末する

こういう使い方になります。CMotionCharactor クラスはまさにこれを利用して作っているので、読み解く際には意識してみてください。他にも STL には便利なクラスが色々あります。紹介した資料以外にも、Web 上で STL の利用例や解説は豊富にありますので、是非利用してみてください。

4.4 今回の課題

ではそろそろ課題をやってきてもらいましょう。期限は定めませんし、提出の有無も未定です。しかし、調べて考え、手を動かして実践してみて欲しい内容です。

1. スタックとヒープというキーワードについて調べて理解を深めてみましょう。C++以外の言語でそれらがどう扱われているかも調べてみるとよいでしょう。
2. プラグちゃんのサンプルを改造し、何か特定の操作でプラグちゃんが増減できるようにしてください。その際プラグちゃんを表す変数は動的に生成すること (あらかじめ配列を決めうちの個数で作っておいて、表示の ON/OFF だけで増減させるのは NG)

2 つめの課題は難易度が高いですが、色々悩んでみてください。