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

(2010/5/19)

4　ポインタ・参照・メモリ管理

4.1　まずはおさらい

今回も引き続き、ポインタにまつわる基礎知識を整理していきます。まずは前回の宿題の 1 つを解決しておきましょう。こんな感じでどうでしょうか？

```
void swapInt(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
    return;
}
```

ちなみに、この int を double に置き換えれば、実数用の関数に早変わりです。呼び出す時には、

```
    int valueA = 5, valueB = 8;
    swapInt(&valueA, &valueB);
```

このようにして使います。「変数の場所を渡す」という発想がないと、こういう処理を関数化することはできなかったはずです。

4.2　実体のような実体でないもの

もう一つ、ポインタ以外に C++ならではの書き方があります。「参照」というシロモノです。

```
void swapInt(int &a, int &b)
{
    int t = a;
    a = b;
    b = t;
    return;
}
```

引数の型が int & となっている点に注意してください。こうした場合、関数を呼び出す際にはこうなります。

```
    int valueA = 5, valueB = 8;
    swapInt(valueA, valueB);
```

あれ、実体を渡しちゃっていいの？と思うかも知れませんが、この組み合わせと先に示した組み合わせは全く同じ動作になります。& を付けて変数を受けることで、ポインタを使った場合と同じような動作を自動的にしてくれる、というだけのことです。「同じ変数を別名で扱える」という解釈でも構いません。主に関数の引数で、ポインタを直接操作するのが嫌な場合に使います。以前私はこれを多用するのを嫌っていましたが、FKUT を構築する際に「変数に & を付けてね」と皆さんにお願いする方がもっと嫌だったので、fkut_SimpleWindow では結構多用しています。どこで参照を使っているのかはヘッダを見れば分かりますから、使いどころを調べて見るのも良いと思います。

4.3　クラスオブジェクトのポインタ

そして、今までは int 型の場合の話だけしていましたが、クラスオブジェクトを扱う場合はもう一つ追加ルールがあります。

- 実体 (もしくは参照) のメンバにアクセスする時は「変数名 . メンバ」
- ポインタを通じてメンバにアクセスする時は「ポインタ名->メンバ」

なんでこうしちゃったんかなーと私もたまに思うのですが、ピリオド (.) と矢印みたいなヤツ (->) を使い分けることになっています。矢印みたいなヤツのことはその名の通り「アロー演算子」と呼んだりもします。まぁこうすることで、今扱っている変数が実体なのか、ポインタなのか、区別ができるので面倒くさいだけではないと思います。

ちなみに「(&変数名)->メンバ」とか「(*ポインタ).メンバ」なんてこともできますが、**キモイのでやめましょう。**

4.4 実体やらポインタやらのまとめ

とりあえず、ごちゃっと出てきたのでまとめます。

表 1: C++における変数関係のまとめ

宣言の仕方	CHoge a;	CHoge *a;	CHoge &a;
種類	実体	ポインタ	参照
代入物	実体・定数	アドレス	実体を必ず代入
メンバへのアクセス	.	->	.
& 付け	アドレス	ポインタのアドレス	アドレス
*付け	エラー！	実体	エラー！

多くの場合、実体とポインタの使い分けさえできれば問題はありません。参照は宣言と同時に実体の代入が必要なため、ほとんどの場合で関数の引数にしか使われません。

ポインタに & を付けると「アドレスを覚えている変数のアドレス」が手に入ります。ダブルポインタと言うこともあります。Direct3D を使っていると有無を言わさず使うはめになります。自分のプログラムの場合、通常の設計では使うことはないか、もしくは使う必要がありそうな場合でも、使わずに済む方法があるはずですので、設計を見直した方が良いでしょう。

4.5 Car サンプルの価値は

さて、ポインタに関する知識を身に付けたところで、Car サンプルをもう一度見直しましょう。Car クラスと World クラスが main 関数で利用されています。これはゲームで言うところの「プレイヤーキャラ」と「背景」に相当します。実践的なゲームプログラムになると、これ以外にクラス化するべきものは、

- 敵キャラ
- 画面上のゲージやメーターなどの表示物
- それらの表示物を含めた「画面全体と動作モード」

などが考えられます。うへぇ、となりそうですが、**一気にやろうとするのが挫折のもと**です。手の動くところからコツコツ進めていきましょう。特に「オブジェクト」として考えやすいものならまだしも、「画面全体や動作モード」をオブジェクトとしてとらえるのは、なかなかのセンスが必要になります。初めから全てをクラス化しようと思わず、出来るところからやっていくのがコツです。オブジェクト指向は理解の途中段階で運用しても、ある程度の効率化が図れるのがいいところだと思います。

また、最初から上手にクラス化出来る人などいません。一度クラス分けしたものでも、開発を進めていくとしっくり来なくなったりします。そういう場合はまた再設計したり見直したりしていくことで、構造が洗練されていきます。まずは今出来ることやる。これが重要です。

次にメンバ関数として何を持たせるかですが、とりあえず必殺パターンを伝授します。

- 初期化処理
- ループ 1 回分の処理
- シーンへの登録・消去処理
- キー入力処理

とりあえずはこんくらいあれば十分です。先週 Car サンプルをよーく読んだ人は「パクリじゃん？」と思うかもしれません。が、それでいいんです。最初のうちは他人のクラス構成を真似るのが手っ取り早いです。それがうまく馴染めば儲けもの、段々合わなくなってきたら、その頃には自力で設計を考えられるだけの経験値は積んでいると見て良いでしょう。

シーンへの登録処理やキー入力では、fk_Scene や fk_Window の変数が必要になります。じゃあこれらもメンバ変数に持たないといけないの？と思ってしまいがちですが、そんなことはありません。必要な時にだけ**ポインタを引数で渡す**ようにすれば、不要なメンバ変数を増やす必要はありません。Car の entryScene などはそうやって解決していますよね？確かめてみてください。

関数名も最初は真似っこでいいでしょう。クラスを設計する場合は**変数は名詞形、関数は動詞形 (+目的語)** というのがお決まりになっています。初期化処理は”init”、ループ 1 回分の処理は”update”などとするのが定番です。

Car サンプルは FK のプログラムをオブジェクティブに書くヒントが詰まっています。これをたたき台にして、皆さん自身のプログラムもどんどんクラス化していきましょう。

4.6 動的なメモリ管理

4.6.1 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。

4.6.2 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。

4.6.3 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。

4.7 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。次の 2 年生向け講義までを期限とします。

1. Car のサンプルを改造し、車をキー操作で動かせるようにしてください。操作方法はどのように設定しても構いませんが、操作方法はコメントに記述すること。
2. Car のサンプルを改造し、何か特定の操作で建物が増減できるようにしてください。その際建物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。