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 5 回資料

竹内 亮太

(2009/6/12)

5 動的なメモリ管理・当たり判定の深遠なる闇

今日は二段構成です。最初に C++のメモリ管理にまつわる避けられない話をします。CMotionCharactor 内で使われているギミックの説明ですので、ここの話が分かれば、自力で読み解いて改造したりなんかもできるかもしれません。

後半は当たり判定に踏み込みます。とりあえずはサンプルを渡しますので、それをいじりつつ雰囲気をつかんでもらえればいいかと思います。

5.1 動的なメモリ管理

5.1.1 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。

5.1.2 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。

5.1.3 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。

5.2 当たり判定の深遠なる闇

時間がないからラフにまとめます。ごめんしてね。

とりあえず、初心者にも実現しやすくて、有用性のある当たり判定の方法は、

- 球
- AABB

です。できれば使いこなしたいものとして、OBB というものもあります。AABB は軸に平行な直方体のことで、OBB は軸に縛られずに回転する直方体のことを指します。

球は距離の比較だけで済むのでとにかくお手軽なんですが、いかんせんそれだけでまっとうなゲームを構築するのは厳しいです。そこで、カプセルという形状を私はおすすめします。球の理論に「2 直線間の最短距離」を求めるという要素をプラスすれば実現できるので、おすすめです。

今回のサンプルは一見 OBB と球の判定のように見えますが、実は裏ではかなり手抜きをしています。やってることは実質 球 対 AABB です。

そして衝突後の戻し方にバグがあります。角度が鋭角になってる面に突っ込むと押し出されてすり抜けます。直し方は思いついてはいるのですが、まだ未検証の状態です。

とりあえず皆さんは、自分の好きな位置にブロックを配置して、当たり判定の挙動を観察してみてください。そして判定の仕方、戻す処理をどうやっているのかを読み解いてみてください。

正直あまりほめられたプログラム構成ではないのですが、実装例がないと理屈だけを説明してもきついと思うので、実装例を先に出します。後は口頭でフォローします。