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２回資料

竹内 亮太

(2012/5/9)

2　C++で実践するOOP

2.1　はじめに

みなさんが前年度で作ったゲームは、どんな構造になっているでしょうか？授業内で教えた要素だけでプログラムした場合、かなりのコードがループ内に足されて膨れあがったと思います。ミニ企画のゲームでこれですから、本ちゃんの企画で同じように作ろうもんならどれほどおぞましいプログラムになることでしょう。想像もしたくないですね。

今回はこの膨れあがっていくコードをなんとかするために、クラスの概念を皆さんが書くプログラムにも導入していきたいと思います。今回は今までの内容から、一般的なプログラミングの話に寄りますので、ちょっと理解が難しいかもしれません。時折見た目もいじつつ、あくまでゲームプログラミングで利用するという前提を意識しながら、よりプログラミングの本質に迫っていきましょう。

2.2　ゲームで使うためのOOP

クラスとはなんぞや、オブジェクト指向 (OOP) とはなんぞや、を語る資料は数多いですが、「だからそれがどう嬉しいのか」を端的に述べているモノは数少ないと思います。それを述べたのが今回のパワーポイント資料です。まずはそちらをご覧ください。

そして、ゲームで使うための OOP を基礎的な部分で実践しているいいサンプルがあります。FK のサンプルページに置いてある、Car.cpp を開いてみてください。車を表す Car クラス、背景を表す World クラスを用意し、それを main 関数内で carObj, worldObj という変数名で実体化することによって利用しています。もしここでクラスを利用していなかったら、車や背景を用意する処理 (init 関数) や、1 フレームごとに車を進める処理 (forward 処理) が全て一緒くたになって main の中にぶちまけられるところでした。このように、

- あるものを表す変数のグループを分離できる
- 分離した変数に関わる処理も分離できる

というだけでも、ゲームプログラミングで OOP を導入する価値は十分にあります。今回配布しているプログラムは、この Car サンプルを関数のみで表現したり、より細かくクラス分けしたバリエーションが入っています。どれも動作としては同じ挙動を示しますが、読みやすさや手の入れやすさが優れていると感じるのはどれでしょうか？見比べてみてください。

2.3　ソースの分離

Car.cpp のオリジナル版では、クラス分けは出来ていてもファイル分けが出来ていません。C++では１つのファイルのなかで複数のクラスを作れますが、どうせだったらクラスごとにファイルを分けて、１つ１つのファイルの見通しは良くしておくべきです。このファイル分けをする時に覚えて欲しいのが、「お品書き」と「中身」の関係です。

再び Car.cpp をよく見てみましょう。このプログラムは先頭から以下のような順番で書かれています。

- FK を使う宣言と定数の宣言
- Car クラスのお品書き
- World クラスのお品書き
- main 関数
- Car クラスの関数の中身たち
- World クラスの関数の中身たち

このように、C++ではクラスを「お品書き」と「中身」に分けて書くのが普通です。クラスを利用する場合、利用したい場所より前にクラスの情報が書かれてないといけません。ですが完全な情報がないとダメ、というわけではありません。**お品書きさえ先に書いてあれば、そのクラスは利用できる**のです。お品書きが分かれば、中身は後ろに書いてあっても、別のファイルに書いてあってもよい、という決まりになっていますので、関数の中身たちは別ファイルに括り出せそうな気はしてきますよね。

となると、お品書きも括り出したくなってきます。お品書きだけ利用する側にわざわざ書くというのはあまりに非合理的ですからね。お品書きだけ括りだしたファイルのことを「ヘッダファイル」と呼びます。拡張子は.h です。FK を利用する際には

```
#include <FK/FK.h>
```

と書いてもらってましたが、この FK.h が FK の全機能のお品書きに当たるわけです。これと同じように、自分が作ったクラスのお品書きを利用する時も

```
#include "Car.h"
#include "World.h"
```

このようにして、**自分で作ったクラスを利用する宣言**をしてやれば、めでたくそのファイル内でクラスが使えるようになります。以降、この宣言のことを「インクルード」と呼びます。

では早速、この Car.cpp を複数のファイルに分割してみましょう。Car クラスを Car.h と Car.cpp に、World クラスを World.h と World.cpp に分けて、main 関数が書いてある部分は car_main.cpp とでもしましょう。Visual Studio で新しいプロジェクトを作って、以下の手順でファイルを追加してみてください。

1. ソリューションエクスプローラからプロジェクト名を右クリック
2. 「追加」→「クラス」
3. 「C++クラス」→「追加」
4. クラス名、ヘッダファイル名、cpp ファイル名を入力して「完了」

Eclipse ほどではありませんが、Visual Studio も必要最低限の親切さは持ち合わせてます。これでファイルの土台は出来たので、サンプルからそれぞれ適切な部分をコピペしてきてください。出来たと思ったら、ビルドして実行してみましょう。結果は変わらないと思いますが、管理のしやすさは 1 ファイルの時よりも上がっているはずです。

2.3.1 インクルードに関する補足

include を書く際に、ファイル名を囲う記号が <> と””の二種類あります。どういう違いかは、

- 誰かが用意してくれたものを利用する時は <>
- 自前で作ったものを利用する時は””

と覚えておいてくれれば、だいたい間違いないです。また、私がこの授業で提供する.h/cpp ファイルを利用する時は、直接プログラムを突っ込んでいるので、自前用意扱いになります。

Visual Studio でクラスを作ると、ファイルの先頭に謎のキーワードが付加されます。「# pragma once」はヘッダファイルに付けるおまじないです。付けないと、少し複雑なプログラムになってきたときにコンパイルエラーが起きます。詳細は「インクルードガード」で検索してみてください。

クラスを追加した際、cpp ファイルの先頭でそのクラスのヘッダをインクルードするようになっています。中身を書くためにはお品書きを知っていないといけないためです。更に、クラスのメンバに他のクラスを使ってる場合は、お品書きや中身の記述の前に利用しているクラスのヘッダをインクルードする必要があります。FK を利用しているクラスなら、当然 FK.h をインクルードしなくてはいけません。インクルードの仕方次第ではコンパイル速度を速くしたりできるのですが、慣れるまではあまり変な書き方をしない方が良いでしょう。そういった細かいテクニックは、後々の授業で述べようかと思います。

2.4 そろそろポインタの話をしようか

さて、クラスを自前で書くようになってくると、避けて通れなくなってくるのがこのポインタというシロモノです。これまでも C/C++初心者の多くがは、ここを学ぶところで挫折してきました。とはいえ、ポインタ自体は決して複雑なものではなく、世間がさわぎすぎな気もします。今日はポインタの概念と、使い方のほんの一例を紹介しておきます。

2.4.1 ポインタの何が嬉しいのか

ポインタの正体については「変数の場所を示すもの(アドレスの入れ物)」ということで、世の多くの解説書や Web サイトで述べられている通りです。多くの人

が悩むのは、その「変数の場所」とやらを使うと何が嬉しいのか、という点じゃないかと思います。

ここでいったんプログラミングから離れて、普段皆さんが使っているであろう「場所」を指し示すものを例に出して考えてみます。ポインタを広義な意味で解釈すれば、場所を指し示すものは全てポインタと呼べるので、もちろんマウスポインタも間違いではありません。でもマウスポインタではさすがに説明に困るので、ここでは 2 つほど例を挙げてみたいと思います。

2.4.2 例 1:URL はアドレスで、それを覚えている変数はポインタだ

例えば、私がみなさんに読んでもらいたい記事があったとします。こういう時、実際にみなさんに読んでもらう方法は大まかにわけて 2 つ考えられます。

- メーリングリストに記事をそのまま書いて流す
- 記事の URL をメーリングリストに流す

これはどちらにもメリットとデメリットが存在します。考えてみましょう。

まず、記事をそのまま流すのは通信量が非常にでかいです。テキストだけの記事ならまだいいとしても、画像がふんだんに使われている記事をメールに添付されたら、回線速度が遅い人にとっては大迷惑です。それに対して、URL(アドレス) だけを記載して流すのは断然スマートです。URL はたかだか数十バイト程度にしかなりませんしね。

一方、メリットとデメリットが逆転する場合もあります。アドレスが送られてきた場合、それを見るためにはネットに接続できる環境が必要です。まぁメールを受信できる環境なら問題なさそうですが、肝心の記事を置いてあるサーバがダウンしている場合もあり得えます。さらに、ネット上の記事はいつまでもそこに存在しているとは限りません。ニュースサイトなどは、記事へリンクを張ったつもりでも数日後には 404 not found になっていることがザラにあります。こんな場合には、メールで直接記事の中身をもらっておいた方が、いつでも好きな時に読み返せて便利だとも言えますよね。

普段からネットを利用することに慣れていれば、この例で言っていることは理解できるはずです。この感覚をプログラミングにスライドさせて考えれば、ポインタの本質は掴んだも同然なので、この 2 つの方法におけるメリット・デメリットは押さえておいてください。同じような問題がプログラミングでも発生します。

2.4.3 例 2:ショートカットもポインタだ

もう一つ例を出します。みなさんのほとんどは Windows をメインに使っているでしょうから、ショートカットの存在はおなじみだと思います。これもある意味ポインタです。

- ショートカットは、他の場所にあるファイルを示すポインタである
- ショートカットを開くことで、そのファイルを直接開いたのと同じ結果が得られる
- ショートカットは多くの場合、示すプログラムやデータの実体よりもサイズが小さいので、ショートカットのコピーには時間がかからずに済む
- ショートカットが示している実体を、削除したり移動したりすると困ってしまう

ほとんど URL と共通の特性ですが、1 つ URL より更にポインタ的な要素があります。上記の 2 番目の特性にも通じる要素ですが、

- ショートカットから開いたファイルに変更を加えると、実体を直接開いて変更を加えたのと同じ結果が得られる

という点です。この点が非常に「ポインタ的」な挙動なので、注目してください。**実体のコピーは同じ内容の別物ができる**のに対して、**ショートカット (ポインタ) のコピーは同じ物を指すショートカットができる**という違いです。これ、重要です。

2.4.4 ポインタと実体の違い

さて、例を出したところでプログラミングの話に戻します。まず、普通の変数は以下のようにして宣言します。

```
int iA = 0;
```

これで iA という変数に整数を代入したり、計算に使ったりすることが出来ます。int 型の変数の実体を作った

わけですね。この何気なく宣言した iA という変数にも、当然「アドレス」が存在します。コンピュータ上のメモリのどこかに、iA 用のスペースが確保されているからです。このアドレスを記憶しておくための変数として「ポインタ変数」というものがあります。ポインタ変数は以下のように宣言します。

```
int *pA = NULL;
```

ポインタ変数は、宣言しただけでは意味を持ちません。他に作った「実体の場所 (アドレス)」を代入して初めて意味を持ちます。試しに先ほど作った iA のアドレスを入れてみましょう。次のようにして取得できます。

```
    pA = &iA;
```

さあ、みなさんの顔が引きつった音が聞こえてきました。謎な記号が出てくるとドン引きしたくなるのは分かりますが、グッとこらえて付き合ってください。この行の意味は、「pA という int 型のポインタ変数に、iA のアドレスを代入する」という処理になります。普通の変数に& を付けることで、そのアドレスを取り出すことが出来るわけです。それをアドレス記憶専用の変数であるポインタに代入しているだけの話です。

これを使うと何ができるかというと、**「この変数にこういう値を入れてやってください」という処理を関数化できる**ようになります。たとえば、

```
void setValue(int argA)
{
    argA = 5;
    return;
}

// 以下は main の中だとして
    int iA = 0;
    setValue(iA);
    // この時点での iA の値は？
```

このようにしても、iA の値は 0 のままです。え、5 じゃないの？と思った人はプログラミング演習のメソッドの回を全力で復習してください。引数には値のコピーが渡されるので、main 側の iA と setValue 側の argA はそれぞれ別々の変数になっています。

しかし、ポインタを投入すると話が変わります。

```
void setValue(int *argpA)
{
    *argpA = 5;
    return;
}

// 以下は main の中だとして
    int iA = 0;
    setValue(&iA);
    // この時点での iA の値は？
```

これだと iA の値は 5 になります。まず、setValue の引数が int のポインタ型になっているのに注目してください。main 側で& iA として、iA のアドレスを引数として渡しています。

そして setValue 側では受け取ってきたポインタ変数に*を付けて、代入式の左辺に配しています。これは、実体に& をつけるとアドレスになったように、**ポインタに*を付けると実体と同じように扱える**というルールによるものです。なので、argpA のアドレスが指す実体に、5 という値を代入するという処理が成立し、結果 main 側の iA の値を書き換えることができたわけです。

2.5 よくある例題

ポインタについて解説する時に、よく 2 変数の値を入れ替える関数が例に挙がります。わざわざ宿題にするのもしょっぱいので、さくっと作ってみましょう。こんな感じになります。

```
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。

2.6 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。

2.7 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。**

2.8 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	Hoge a;	Hoge *a;	Hoge &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。

2.9 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。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。

2.10 今回の課題 (2 年生向け)

ではそろそろ課題をやってきてもらいましょう。毎回期限は切りませんが、学期末にまとめて提出してもらいますので、1 週間を目処に消化していった方が賢明だと思います。

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. 昨年度 11 回のサンプルを、今後ゲームとして拡張していくことを前提としてクラス分けし、動くようにしてください。

それぞれどう設計すればいいか、細かい部分の書き分けはどうすればいいかなど悩む部分もあるかと思いますが、とりあえず目的が達成できれば良しとしますので、試行錯誤してみてください。