C++の奴隷になったプログラマたちが繰り広げる大冒険

ゲームプログラミング2017

PROGRAMMING OF HELL

はいどーもー、普炉愚(プログ)ラムでーす。

ウソです。

まぁそういうアカウント作ってみたら面白そうだなーとは思っています。

目次

はじめに....4

今年の方針....4

クラス設計的なとこ....10

柔軟剤について分かりやすいかも例…....10

入力まわりの柔軟性....10

そのための map…あとそのための…multimap？....12

あーもうクソメンドクサイなぁ…....16

じゃあ軽く実際のコーディングだけどさ…単体テストって知ってるかな？....20

柔軟性を損なうデザインパターンの濫用....26

シングルトンパターンおさらい....26

理由①:C++の static は初回コール時に「実体化」される....27

理由②:そもそもこのやり方じゃないと、マルチスレッドに対応するのがクッソ面倒....27

なぜシングルトンパターンが柔軟性を損なうのか？....29

シーン周り設計....31

State パターン....31

さらにスタック構造にする....32

リファクタリング....34

リファクタリングとは何か？....35

定義....35

おかしな名前....35

クソコメント....35

クソコード....39

プレファクタリング....40

一応のコーディング規約(目安)....40

ひとりでもバージョン管理システム(Git)....41

プログラミングの原則....43

switch 文撲滅のために…....44

メンバ関数ポインタ....45

似たようなことをラムダ式で や ら な い か? 48

はじめに

僕の授業は、毎年言われますけど「進行が早い」です。昔はクレーマー気質の学生に「速すぎる、ついていけるわけねーだろ!!」とキレられましたが、

ついてこれねえ奴が遅すぎるんだぜ…

ちなみにクレームを入れようが、泣こうが喚こうがスピードは緩めません。
理由はあるんですよぉ？意味もなく早くするわけじゃないんですよぉ～？
見た目じゃ区別突かないですけど、このクラスには次年度就職年次がいるんですよねぇ…。ホラ、次年度就職年次って就職のための作品を作らなければならないでしょ？

というわけで、飛ばさざるを得ないんですよね。
まぁ、僕の性格と言うか方針はボチボチ分かってくると思いますが、多少苦しめたいという意図もないわけじゃないです。

今年の方針

さてさて、今年の教育方針はどうしようかなあと考えています。1つ1つゲームプログラムをしっかり作っていくのか、ガッチガチの設計をやっておいて、部品とガワを作っておいて、量産していくのか…そのどちらかで迷っています。

- 1つ1つ丁寧に作る
- **設計**を十分にしておいて量産体制を整える

毎年この二つの思想で悩んでいます。普通に考えると後者の方が良さそうな感じがするんですが、ていうかプロはこっちなんですけど(そうとも限らないかぁ…)、かつて後者のやり方でやっていたら「プログラムを書けなくなった」学生が多発してしまった苦あい思い出がありましてねぇ…。

まぁ、今ここにいるみなさんは強力な教育を既に受けておられるプログラムマシーンでしょうから今年は…後者でいいかなぁ…って思っています。

ちなみに「**設計**」とは言うものの、これには「**正解がありません**」…完璧な設計と言うものは存在せず、より良い設計というものがあるだけです。どういう事かと言うと、まず状況によって必要な設計は違います。この状況ってのが作りたいゲームの内容は勿論の事、メンバーのスキルレベル、納期、文化に左右されます。
つまりどうしても「個性」は出てしまいます。それはいいんです。

逆に「この設計が正解なんだ!!」みたいに凝り固まらないでください。恥ずかしい事になりますよ？

ただ…正解はないんですが「マズい設計」というものは存在するし、設計をするための「武器/道具」は存在します。

この授業を受ける上で大事なのは「**柔軟性**」です。脳みそグニャグニャにしてください。「しっかり十分に設計」というのと相反する印象を受けると思いますが、その通り。今回の十分に設計ってのは、必要になる事前知識を頭に入れておいて、クラス図を描いてみて、全体を見てみて、一度くらいは再設計を行うという程度です。

ちなみに最初にクラス図とか書いて設計していきますが、これは実装したり話し合ったり、もっというとその時の自分らのスキルレベルで変化していきます。

一応、プログラミングの世界には「リファクタリング」という思想があって、最初からプログラム設計は完成品ではありえず、事あるごとに修正されるべきである。この時に「リファクタリング」でよりよいコードにしていくのですが、ルールとして

- リファクタリングと機能改変を同時にはおこなわない。
  リファクタリングをしてから新しい機能を追加する。

- リファクタリングを始める前と後にはユニットテストを実行し
  コードの機能が変更ないかを確認する。
- パフォーマンスよりもメンテナンス性を重視する。
- こだわり過ぎてはいけない。
- 小さなリファクタリングとテストの組み合わせを繰り返す。
  決して一度に大きなリファクタリングをしない。

こういうのがあります。で、ここで「リファクタリング」は何を改善するのかと言うと、速度や潤出やそういうのではありません。あくまでも「コードの可読性」と「コードの柔軟性」を高めるものです。ここを間違うと混乱しますので、ご注意ください。

コードの可読性とかコードの柔軟性とか言われても全然わからない人は

とにかくクソコードをぶっ潰せ!!!

とご理解いただければいいと思います。

クソコードとはわかりやすい代表的なのを書くと
- 見ても意味が分からないコード(訊くぞ？)
- 他人に理由を説明できないコード(怒るぞ？)
- マジックナンバーが多発しているコード(ぶっ飛ばすぞ？)
- コピペ乱発コード(ふざけるな!!!)
- グローバル変数が多い(プロジェクト壊れちゃ～う!)
- クソ長え関数(やめてくれよ…)
- クソでかいクラス(やべぇよやべぇよ…)
- バラっバラ(統一されてない)の変数名や関数名(あたまおかしなるで)
- クソ名前付け規約(a0 とか fuckyou99 とかやめろホンマ)
- クソコメント(嘘コメント、意図がわからないコメント、意味のないコメント)
- ノーコメント(だから意図がわかるようにしろや…特にヘッダ!!!)

あと、コード実験で、特定のコードをコメントアウトするのはいいけど、いつまでも残しておかない事。基本的にはコードのバージョン管理(Git とか)を行い、潔く消してください。

ちょっと難易度が高いクソ(?)とまで言えるかどうか…でもスーパーハカーから怒られるかもしれないコード
- 結合度が高すぎる…クラスやファイル間の依存関係が強すぎて簡単に分離できない

- 共通処理が共通化されてない…関数化やポリモーフィズムを駆使しましょう
- カプセル化が不十分…これは設計段階でマズい場合が多い
- STL とか使えば 1 行で済むところを 20 行くらい書いてる…勉強しろ
- Warning を無視しないでくれ～

とはいえ、初心者…経験が少ないうちは間違えたり、クソコード書いちゃったりするもんなんや。ええねんそれで…手が止まる事の弊害の方が大きいねん。バンバン作って、バンバン怒られて、バンバンリファクタリングしまひょか？

あと俺の偏った思想的には…

- switch 文絶対殺すマン

というのがありますあります。これは特に状態遷移的な部分で switch を使用するのを僕が嫌うためです。これに関しては多少…過剰すぎて不適切な部分があるかもしれないので、真似するかどうかは、ご自分の判断にお任せいたします。

さて、しっかり設計やっといて、量産体制でバンバン作っていきたいですが、ぼくとしては作りたいコアがあって、今年は 2D アクションゲームと格闘ゲーム(3D モデルの 2D ルール)をやっちゃいたいと思います。そのための…設計。あとそのための…リファクタリング？

そんな感じで可能な限り柔軟性のある設計とコードをやっていく事を前提として、授業を進めていきます。

今一度言っておきますが、**設計に正解はありません。**でも、「より良いコード」は存在します。なんかはっきりしねーんだよね。こういうのもプログラミングが難しい理由でもあります。それだけに自分自身が「しれっ」とレベルアップしておく必要があります。

ぼく自身もクソザコナメクジなので、どうやってレベル上げようとしてるかと言うと、「良質なオープンソースのコードを眺める」ですね。もちろんそれだけでレベルは上がらないので、自分なりにコーディングして、何度も失敗する必要があります。

あくまでも

自らコーディングして失敗＞コードリーディング

である事は間違えないように…

その前提で、オープンソースを一応紹介しておきますけど、う～ん、まずは2D の奴から紹介しましょうか

PAINTOWN

https://sourceforge.net/p/paintown/code/HEAD/tree/

アクションゲーム系の 2D エンジンです。まぁ既に古いプロジェクトなので、中のコードはちょっと古いですね。でも寧ろ古いくらいの方がちょうどいいかもね。

その他、様々なクローンゲームのソースコードが載っているサイトはある。

https://osgameclones.com/

品質は保証しませんけども…

DxLib

知ってた？これ、オープンソースなんだぜ。ただ、初心者向けのライブラリにしたいという思想があるので、独特のコーディングルールと設計になってる印象がありますね。

https://dxlib.xsrv.jp/DxLib/DxLibMake3_20a.zip

エンジン系

cocos2d-x

https://cocos2d-x.org/

Godot

https://godotengine.org/

UnrealEngine4

https://github.com/EpicGames/UnrealEngine

Lumberyard

https://github.com/aws/lumberyard

ちなみに、設計やプログラミングテクニックも見てほしいんですが、それ以外にも**フォルダの切り分け方や、他ライブラリの組み込み方**とかも見ておいてほしい。

また、オープンソースライブラリでいいコードを書いているのは Effekseer

https://github.com/effekseer/Effekseer

結局こういうソースコードを読むにも前提知識が必要になるんだけどね…

一応ね、参考くらいに考えておいてね。で、色々なの見ていると分かると思うけど、うん、本当に正解なんてないでしょ？
あといくつかのコードは何やってるのか全く分からないでしょ？つまり前提知識が圧倒的にまだまだ足りてないのよね。

というわけで、この授業は高速でやっていかざるを得ないわけです。QED。

ともかく、僕がアクションゲーム好きなので、アクションやる前提で作っていきたいと思います。ただ、皆さんは自分で作りたいゲームがあると思いますし、それは授業外で作ってもらっても構いませんし、授業の進行に十分すぎるほどついてこれてれば、授業中にオリジナルゲームを作っていてもらっても構いません。寧ろバンバン作りましょう。

ただし、授業外の物を作っている事を、課題がクソな理由としては認められませんので、課題は課題。自作は自作で頑張ってください。まぁ何でこういう事を言うのかと言うと、自作に力入れすぎた挙句、その自作のクオリティが、課題として要求する遥か下の彼方にあったという事があり、結局そいつらは授業で教えたテクニックとかも身についてなかったので、授業の課題は授業の課題で、要求レベルまできっちり作ってもらいます。

あと、ここに3年生がいると思いますが、当然ながら2年生よりもハイレベルなものが求められますので、それはご了承ください。

クラス設計的なとこ

とはいえ 1 回でクラス設計完成させる気はない…ただ「柔軟性」とか言われてもよくわからんでしょ？

分からないファー

まぁ…分かりやすい部分からチョイと説明していこうかなと思います。

柔軟剤について分かりやすいかも例…

入力まわりの柔軟性

例えば…だ。柔軟性についての考えが皆無であるならばキー入力の部分なんかは

```
auto padstate = DxLib::GetJoypadInputState(DX_INPUT_KEY_PAD1);
if (padstate&PAD_INPUT_UP) {
	vy = -5;
}
if (padstate&PAD_INPUT_DOWN) {
	vy = 2;
}
if (padstate&PAD_INPUT_RIGHT) {
	x += 2;
}
if (padstate&PAD_INPUT_LEFT) {
	x -= 2;
}
```

このように書くであろう？そうに違いない!!

だが、例えばキーコンフィグを設定できるようにしたい…なんでこれが必要なのかと言うと、よくあるのが、クソやりづらいキー配置にする奴がいるのよ。それは君の趣味だから別にいいけどさぁ…変えたいわけよ。

クソゲーとかはキーコンフィグないよね。というわけで、まず、このコードは柔軟でないと言えるわけだ。
実際にはそこだけじゃないけどね(移動速度が定数なところとかダメだよね)

となると、キーコンフィグアリにするのは柔軟性が向上するとは言えないかね？となると入力を **DxLib の関数から直接調べて、それを動作に反映するというのは柔軟性的には悪手**と言える。少なくとも柔軟性の観点から言えばね。ああ、今までのゲームをそう作っていたからと言って悲観する事はないよ。別に初心者なら当たり前のことだし、悪くもないし、今回のこれはあくまでも「例」として挙げているだけだからね。

というわけで、ちょっと迷いどころなんだけど、入力マップというやつを考える。

あ、マップと言っても地図の意味じゃないですよ。ステージの情報じゃないですよ。なんていうか、入力とその意味の繋がりを表(テーブル)にしたようなデータの事だと思ってくれ。

難しいっすかね…。例を出したほうがいいかな。一応入力情報は文字列を元にテーブルが作られていると仮定します。すると例えば

入力値	入力コード
キーボード右,D,パッド右	"right"
キーボード左,A,パッド左	"left"
キーボード上,W,パッド上	"up"
キーボード下,S,パッド下	"down"
キーボード Z,パッド X	"jump"
キーボード X,パッド Y	"attack"

こういう風にしたいわけ。言いたい事は分かるかな？

さて…どうしようか？
実装…感じるんでしたよね？

というわけで、僕の授業でやたらと出てくるのが STL(Standard Template Library)なんですが、どうですかねぇ？vector と map くらいは把握しておりますでしょうか？

じゃあさぁ…STL を使って、これを実装してみるとしたら…どうかな？どうすっかなー俺もなー。

最初っから正解は書かないよ？思考の過程も大事だからね？

そのための map…あとそのための…multimap？

まぁまずは map<string,int>で作ってみる事にしようか？やるとしたら

```
map<string,int> inputmap;
inputmap["right"]=PAD_KEY_RIGHT;
```

こんな感じなんだけど、これでは複数設定ができないよね？じゃあ、こうしようか？

```
vector<int> inputcode={PAD_INPUT_RIGHT,KEY_INPUT_RIGHT,KEY_INPUT_D};
inputmap["right"]=inputcode;
```

まぁまぁ、ええんちゃうかなと思うんですが、これだと複数プレイに対応できないよね？

という事で、input 側を

```
struct InputInfo{
	int peripheralNo;//機器番号(パッド番号はそのまま…キーボード/マウスは-1とかで受け取ろう)
	int code;//入力数値
};
```

とでもしてやれば

```
map<string,vector<InputInfo>> inputmap;
```

という感じでキーマップを定義できる。ちなみに int がもったいないなら char でも short でも構わない。

また、map のお仲間には multimap というのもあるので、それも一応紹介しておきますね。お勉強ですし。

multimap は「重複あり」のマップの事です。聞いたことはあるけど使ったことがない人も多いのではないでしょうか？やり方は簡単です。重複ありにしたければ

```
multimap<string,InputInfo> inputmap;
```

という風にすれば「重複あり」になります。つまり同一キーの「値」を複数登録できます。これにより、先ほどのコードの vector が不要になります。

ただし、重複ありという事で、

```
inputmap["right"]=inputcode;
```

のような形での要素の追加や、値の変更ができませんし、値の参照もできません。そもそも[]オペレータがないみたいですね。理由は理屈をちょっと考えれば分かるはずですね。なので合えて解説しませんが、疑問な人は質問してください。

さて、それならどうやって要素を追加したり編集したり参照したりするのでしょうか？

うん、まぁ、まずは
https://cppref.jp.github.io/reference/map/multimap.html
を見とけよ見とけよ～

まず、追加に関してですが、insert か emplace を使う事になります。例えば

```
mm.insert(make_pair("apple",2));
mm.insert(make_pair("orange",5));
mm.insert(make_pair("apple",3));
mm.insert(make_pair("orange",8));
```

とやるか、make_pair 書くのが面倒な人は emplace を使います。

```
mm.emplace("apple",2);
mm.emplace("orange",5);
mm.emplace("apple",3);
mm.emplace("orange",8);
```

ていうか、emplace があるんだから insert 要らねーじゃんって思うんですが、emplace が実装されたのって、比較的最近の話だからね、仕方ないね。

ちなみに、初期化の際にもデータを突っ込む事ができます。

```
multimap<string,int> mm={{"grape",17},{"grape",20}};
```

一応これでデータが重複キー状態で入っています。全部見るには…いつもの範囲 for 文を使って見てみましょうか。

```
for(auto m : mm ){
```

```
        cout << m.first << " : " << m.second << endl;
}
```

とやれば

```
apple : 2
apple : 3
grape : 17
grape : 20
orange : 5
orange : 8
```

と出力されます。確かに全部登録した奴だぁ!!
でも、皆さんが期待するのはこういう機能ではございませんよね？そう…たとえば orange なら orange だけを全て抽出しとうございますよね？

その場合は equal_range 関数を用いて、キーに合致 begin と end が取得できます。orange だけ抽出してみましょう。

```
auto range=mm.equal_range("orange");
```

range には、begin と end が入っていますが、pair になっていますので、first と end です。つまり orange を列挙したければ…

```
for_each(range.first,range.second,[](const pair<string,int>& m){
                                cout << m.first << " : " << m.second << endl;});
```

と書きます。first,second なので残念ながら範囲 for 文ではいけないんですよねぇ…なので algorithm の for_each を使っております。見りゃだいたい意味が分かると思うので、特に解説しません。

さて、となると…キーを列挙したい場合がちょっと難しくなるんですね。例えば orange が2つ出てきちゃうから1キー1つだけ列挙したい場合ってのが面倒なんですよねえ…キーを vector に放り込んで、unique ドバーやっちゃってもいいんですけど、それもねぇ…て感じなので結局は

```
string key="";
for(auto m : mm ){
        if(key==m.first)continue;
        key=m.first;
        cout << m.first << endl;
}
```

てな感じになるでしょう。結果は

```
apple
grape
orange
```

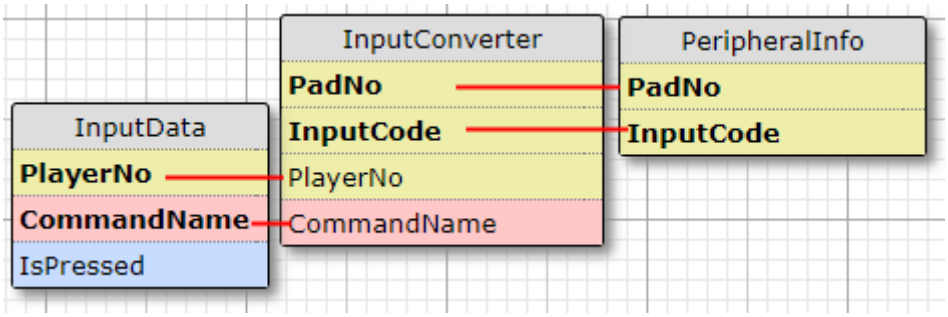
というわけです。

あと、このやり方だと、値が設定されてないキーがそもそもヒットしなくなるので、ちょっとなー俺もなーって感じです。というわけで僕は面倒ですがたいていの場合 map+vector にしています。

また、map に関してですが、順序が重要でないならば map よりも unordered_map の方が高速です(最大 2 倍くらい…だけどハッシュ値を作る関数が重要になる)。要素の追加も参照も高速なので、今回みたいに順序が重要じゃないなら unordered_map の方がいいかもしれません。

使い方に関しては map や multimap と同じで、違うのは中身のアルゴリズムなので、順序が重要じゃない時はパパパッと unordered に決めちゃってさ、終わりでいいんじゃない？

ところでさぁ…ER 図って…知ってるかな？
データベースとかで、



こういう図…見たことあるかな？

テーブルとテーブルの関係とかプライマリキーでリンク張ったりするものなのね。

まぁ、別に E-R 図はどうでもいいんだけど、要は図で考える…というか、頭の中でゴチャゴチャ考えてても、脳味噌って意外とフワフワしてるんだよね。

だからさ、可能な限り図で書いてほしいんだわ。ノートに落書きでいいからさ

さて、PeripheralInfo ってのは、もう DxLib の API から得られるデータの事です。
で、InputData ってのがインターフェイス(自作)から返されるデータの事です。

で、間にコンバータが入っていますが、ユーザは直接 API データを見るのではなく、InputData を見る…。キーコンフィグなどで変更されるのは InputConverter の部分で、API から分離されているので、柔軟性が保たれる…と、こういうわけです。

今回のコンバータにあたるのは、間の map データという事になります。ただし、検索効率から言うとちょっと難がありますね。ちょっとだけ効率の観点から見てみましょう。

あーもうクソメンドクサイなぁ…

既に疑問に思ってる人もいると思いますが、実際に使う際には入力値→入力コードへの変換が必要です。つまり逆方向ですね。ということで

```
map<PeripheralInfo,InputInfo> intputTable;
```

いわゆる「逆マップ」を使用する事になります。

map である以上はキーを検索しなければなりません。しかし今回はキーが InputInfo というオレオレ構造体…どうする？

これをやるには InputInfo の「大小を定義」しなければなりません。なに…難しい話ではない。オペレータオーバーロードを使用するのだ。通常は＜演算子ね。Lesser だっけ？Greater だっけ？どっちでもいいけど。

```
struct PeripheralInfo {
	int padno;//生入力機器番号
	int code;//生入力コード
	bool operator<( const PeripheralInfo & rval) {
		if (padno == rval.padno) {
```

```
                return (code < rval.code);
        }
        else if (padno < rval.padno) {
                return true;
        }
        return false;
    }
};
```

こんな感じになります。あ、この場合演算子オーバーロードをメンバ関数として作るとmap側で怒られることがあるのでその場合は

```
bool operator<(const PeripheralInfo & lval, const PeripheralInfo & rval);
```

比較演算子関数を外部関数として定義しておきましょう。
これで自分で定義した構造体の時も検索をすることができます。

で、次にプレイヤーへのマップとしてのInputInfoですが、よく考えたらプレイヤーが増えても配列でいいですね。返すべき入力コマンドの種別はプレイヤーの番号に関係ありませんし…となると

```
vector<map<string,PeripheralInfo>> inputTable;//一覧出すため用
map<PeripheralInfo,pair<int,string>>  inputMap;//入力コードからプレイヤーとコードを得るため用
```

これでとりあえず先ほどちょっと前に書いたER図の構造を双方向的に実現する事ができます。うーん、柔軟性って大変なんだなあと思います。

まぁ「ありえる事象」をどこまでカバーするのかを割と最初の方に決めておかないと実装の見積もりが立てられないという事が分かりますね？

というわけで、最初の方で、どのようなゲームにするかは予めある程度確定しておかないと、今回やったようなことを後からやるのは大変なんだよ。

例えば、何も考えずにKeyStateやらPadStateやらが散らばってしまっている状況から、キーコンフィグを作ろうとか、プレイヤー増やそうとか想像してみ？クッソ大変っすよ？

とはいえ、ある程度作ってからでないと気づかない事も多いという事で、経験とさじ加減なん

だけど、2 年生以降の開発はこのように

「変化に対応できるように」

というのと

「初めからある程度は見通し立てておこう」

というのが重要だと理解しておいてほしい。

ちなみに今回は入力コマンドを文字列としているが、入力ログを残す(リプレイやコマンドログ)ために、文字列ではなく、enum とか、ビットフィールド でも構わない。

あとアナログ入力系をやろうとするとちょっと面倒なので、今回は通常の一般的な 2D アクションゲームなどにありそうな入力についてのみ考慮した。

アナログあるとねぇ…キーボードありにするとマウスとの連携を考える必要があるので、かなりねえ。
それでも一応、マウスの動きとアナログスティックを関連付けする事はできる。…できるんだが…まぁ、そこはゲームデザイン的な部分になっちゃうので、ちょっと設計からは離れてしまうねえ。

ちなみに、map を unordered_map にする場合は、必要な比較演算子が違うのと、あとハッシュ関数オブジェクトも定義しなければならないので、非常にめんどう。

比較演算子はイコールとノットイコールだけだから簡単だけどさ……

```
bool operator==(const PeripheralInfo & lval, const PeripheralInfo & rval);
bool operator!=(const PeripheralInfo & lval, const PeripheralInfo & rval);
```

は？関数の実装？

この程度自分で実装しろよ

面倒なのはハッシュ関数なんだからさ…。

さて、ハッシュ関数という…わけわからん言葉が出てきましたね？
https://ja.wikipedia.org/wiki/%E3%83%8F%E3%83%83%E3%82%B7%E3%83%A5%E9%96%A2%E6%95%B0

あぃご存知？簡単に言うと中身が一意に識別できるような ID(数値)を返す関数って所ですよね。今回は PeripheralInfo を一意に識別したぃ…そんな数値どうやって作ろうかな？

プレイヤーは最大でも 16 名…キーコードも 128 あれば十分…つまり PAD 番号を 4 ビットずらせば OK やな!!!つまり

```
p.padno | (p.code << 4);
```

こういう演算で OK なんだよなあ…さて､unordered_map には､キーと値の型以外に 3 つ目のパラメータとして､ハッシュ関数オブジェクトの型を指定してあげる必要があります。

これがさぁ…ラムダ式で行けたら簡単なんすけど…なぁんかうまくやれないんすよねえ。というわけで仕方ないので『関数オブジェクト』を作る事にします。

ちなみにどのサイト見ても自作型をキーとした unordered_map はクソややこしぃ説明なんで､とりあえず諦めてくださぃ。

```
unordered_map<キーの型, 値の型, 関数オブジェクト>
```

これが何かの関数の中ならラムダ式でええねんけどな…ヘッダ内でメンバ変数として扱うんやったら､関数オブジェクトを作る必要がある。

で､関数オブジェクトって難しそうに思えるんだけど､ひとことで言うと

『()演算子をオーバーロードしたメソッドを持つクラス(または構造体)である』

というわけだ。どういう事かと言うと

```
struct Functor{
        void operator()(){
                cout << "Fuck you!" << endl;
        }
};
```

という型を作っておくと

Functor func;
というオブジェクトを作れば
func();
で、ファッキューしてくれます。

ほら、関数呼び出してるみたいでしょ？これを関数オブジェクトと言って、ファンクタとか言われたりしています。

で、unordered_map はファック関数もといハッシュ関数を要求してくるので、メンバ関数(じゃなくてもいいけど)として作ってしまいます。

```
///unordered_map用ハッシュ関数
///一意な値を返せばいいだけなんで
///パッド番号と入力値をor演算してるだけっす
struct HashFunc {
	size_t operator()(const PeripheralInfo& p)const {
		return p.padno | (p.code << 4);
	}
};
```

で、これを使えば…

```
unordered_map<PeripheralInfo, pair<unsigned short,string>,PeripheralInfo::HashFunc> _inputMap;
```

のように unordered_map として宣言する事ができるようになります。

じゃあ軽く実際のコーディングだけどさ…単体テストって知ってるかな？

今回は Input 系単体で作ってみたので、ゲームのプログラムに組み込みたいところなんだけど、一旦単体でテストしてみよう。一応ソフトウェア開発においては「ユニットテスト」と呼ばれるもので、別モジュールとして用意しておいて、たいていの場合自動テストするんだけど、今回は PAD 等の入力が関わってるんで、手動テストすることにしてみよう。

自動でやれない事もないのだが…入出力の自動化は分かりづらいですしね。

で、このテストやってる最中に DxLib のバグを発見してしまいました。一応公式サイトに通報して、こちらでも原因を探しています(DxLib のソースコード見ながら)。

こういうのがあるから、単体テストやっといた方がいいんだよね。恐らく原因は二重カウントじゃないかなと思う。
中を見ると「Xbox360 用のコード」があって、Xbox 系のコードがあるんだよね。恐らくは 360 ではないが Xbox という事で、本来通ってはいけないパスを通って、二重カウントしてるんじゃないかなと思います。

とりあえず掲示板にしつこく連続投稿兄貴してやったりなので、そのうち思い知ると思いますので授業が始まる頃には治っていると思います。

まぁ、DxLib の入力周りを見ることができて、勉強にはなったけど、授業始まるまでに治してよ～頼むよ～間に合わんかったら本気で俺が治すよ～。

…むっちゃしつこくお願いしたら治してくれたよ～。バグのせいで仕様を勘違いしてて、くっそ作り直したよ…OTL。でも即日治してくれたから、作者さんありがとうって感じです。

やる準備が整ったんで、実際に実装してやるお!!

参考までにコード見せますが、ただただ写すような事はしないようにしてください。あくまでも参考として考えてください。**<u>コード写経が許されるのは１年生とアホだけ</u>**です。

```
///入力情報
struct PeripheralInfo {
	PeripheralInfo() {};
	PeripheralInfo(int pdno, int cd) :padno(pdno), code(cd) {}
	int padno;//生入力機器番号
	int code;//生入力コード
	///unordered_map用ハッシュ関数
	///一意な値を返せばいいだけなんで
	///パッド番号と入力値をor演算してるだけっす
	struct HashFunc {
		size_t operator()(const PeripheralInfo& p)const {
			return p.padno | (p.code << 4);
		}
	};
};
bool operator<(const PeripheralInfo & lval, const PeripheralInfo & rval);
bool operator==(const PeripheralInfo & lval, const PeripheralInfo & rval);
bool operator!=(const PeripheralInfo & lval, const PeripheralInfo & rval);
```

ときて

```
///入力周りクラス
class Input
{
private:
	std::vector<std::unordered_map<std::string, PeripheralInfo>> _inputTable;//プレイヤー番号と入力対応テーブルのセット
	std::unordered_map < PeripheralInfo, std::pair<unsigned short, std::string>, PeripheralInfo::HashFunc> _inputMap;//入力情報と利用情報のセット
	std::vector<std::unordered_map<std::string, bool>> _currentInputState;//現在の押下情報
public:
	Input();
	~Input();
	///接続中パッド数を返す
	int GetConnectedPadCount()const;
```

```
        ///プレイヤー数(パッド数とは関係ないよ)を設定
        void SetPlayerCount(int pcount);

        ///コマンド設定
        ///@param pno プレイヤー番号
        ///@param cmd コマンド文字列
        ///@param periNo 入力番号
        ///@param code 入力コード
        void AddCommand(unsigned short pno, const char* cmd, int periNo, unsigned int
code);

        ///入力情報更新
        ///@remarks 毎フレーム呼び出してください
        void Update();

        ///押下チェック
        ///@param playerNo プレイヤー番号
        ///@param cmd コマンド文字列
        bool IsPressed(unsigned short playerNo,const char* cmd);

        ///現在の押下状況を返す
        const std::vector<std::unordered_map<std::string, bool>>& CurrentState()const
{ return _currentInputState; }
};
```

という感じですね。で、テストコードとしては

test.cpp

```
Input input;
input.AddCommand(0, "up", 0, KEY_INPUT_UP);
input.AddCommand(0, "down", 0, KEY_INPUT_DOWN);
input.AddCommand(0, "right", 0, KEY_INPUT_RIGHT);
input.AddCommand(0, "left", 0, KEY_INPUT_LEFT);
input.AddCommand(0, "jump", 0, KEY_INPUT_Z);
input.AddCommand(0, "attack", 0, KEY_INPUT_X);

auto padcount = input.GetConnectedPadCount();
for (int i = 0; i < padcount; ++i) {
```

```
		input.AddCommand(i, "up", i+1, PAD_INPUT_UP);
		input.AddCommand(i, "down", i+1, PAD_INPUT_DOWN);
		input.AddCommand(i, "right", i + 1, PAD_INPUT_RIGHT);
		input.AddCommand(i, "left", i + 1, PAD_INPUT_LEFT);
		input.AddCommand(i, "jump", i + 1, PAD_INPUT_1);
		input.AddCommand(i, "attack", i + 1, PAD_INPUT_2);
}
```

こんな感じで設定しておいて…

```
while (ProcessMessage()==0) {
		ClearDrawScreen();
		input.Update();
		int row = 0;
		int idx = 0;
		auto inputstate = input.CurrentState();
		for (auto& m : inputstate) {
			for (auto& s : m) {
				DrawFormatString(50, 50 + row * 25, 0xffffff, "PNO=%d ,
INPUT=%s , INPUT_CODE=%d", idx,s.first.c_str(),s.second);
				row++;
			}
			++idx;
		}
```

チェックします。一応 USB ハブに繋いでパッド3つ＋キーボードでテストして大丈夫でした。
一応、input.cpp の中で実装に迷いそうな部分は、AddCommand と Update かなと思うのでこれも参考までに…

```
void
Input::AddCommand(unsigned short pno, const char* command, int periNo, unsigned int code)
{
		assert(pno < _inputTable.size());
		_inputTable[pno][command] = PeripheralInfo(periNo, code);
		_inputMap.emplace(PeripheralInfo(periNo, code), make_pair(pno,command));
		_currentInputState[pno][command] = false;
}
```

ちなみに emplace は普通に insert の代わりに使えます。基本的には push_back や insert の代わりに emplace や emplace_back を使えるなら使った方が望ましいです。

まぁ～、そこら辺に関しては
https://www.slideshare.net/Reputeless/c11c14
https://cedil.cesa.or.jp/cedil_sessions/view/1953
を見るといいと思うよ。

柔軟性を損なうデザインパターンの濫用

柔軟性を損なうデザインパターンって何かご存知かな？
ほら、皆が好きなアレ…アレだよ。

スィングルトン(Singleton)パターンでございます。他のパターン全て忘れてもこれだけは覚えているという人もいるのではないかな？

ちなみにシングルトンは人によって実装が違いますが、ちょっとおさらいも兼ねて僕の実装を紹介しますねぇ。

シングルトンパターンおさらい

原則として僕が用いるシングルトンは

```
class Singleton
{
private:
	//生成禁止
	Singleton();
	//コピー、代入禁止
	Singleton(const Singleton&);
	void operator=(const Singleton&);
public:
	static Singleton& Instance() {
		static Singleton instance;
		return instance;
	}
	~Singleton();
};
```

こんな感じです。ポインタは一切用いない…まぁ、これには理由があるんだ。

理由①:C++の static は初回コール時に「実体化」される

これはどういうことかと言うと、C++の仕様では、変数のためのメモリはプログラムカウンタがそこに来た時に確保されるというのがあります。

実はこの仕様は static に対しても有効で、この Singleton クラスが実体化されるのは、最初に Instance()関数を呼び出したその時で、それ以降一切実体化されません。

static ですから、2 度目に通るときには既に実体化されているメモリが使用されます。

if 文もな～んにも必要ありません。楽でしょ?

よくある批判として、シングルトンは生成と破棄のタイミングが掴めないという事がありますが、それに関しては生成タイミングは前述のように最初の呼び出しの時ですし、そもそも Singleton にしてる時点で破棄のタイミングはつかめないので、議論しても仕方がないです。

ちなみに破棄のタイミングについて議論し始めたら、ガベージコレクションや、スマートポインタまで疑惑の目が向けられることになります。

static なので破棄のタイミングはアプリケーションが終わるときに決まってるだろいい加減にしろ!!

理由②:そもそもこのやり方じゃないと、マルチスレッドに対応するのがクッソ面倒

まぁ、新 2 年生にとってはマルチスレッドとかはまだまだ関わりがないのかもしれないけど、これからのプログラミングはマルチスレッドは意識しておいた方がいい。

その場合、ポインタを用いてシングルトンを実装するとちょっとばかりややこしい事になる。

```
class Singleton
{
private:
	static Singleton* _instance=nullptr;
	//生成禁止
	Singleton();
	//コピー、代入禁止
	Singleton(const Singleton&);
	void operator=(const Singleton&);
public:
```

```
        static Singleton& Instance() {
                if(_instance==nullptr){
                        _instance=new Singleton();
                }
                return *_instance;
        }
        ~Singleton();
};
```
などというシングルトンにしていたとする。

これの何が問題なのだろう？問題は if 文と new の間だ。

シングルスレッドならば何一つ問題はないように思われるが…そう、マルチスレッドの場合、他のスレッドで if 文を評価している最中に他のスレッドで Instance 関数を呼ばれることもあるのだ。

どうなる？

そう…シングルトンのくせに2つ生成できてしまうのだ。

ヤバいのは言うまでもありませんね。

ちなみに<u>クリティカルセクションもしくはアトミック</u>を使いこなせばポインタ相手でも前述の問題は解消できる…が、そのためには<u>マルチスレッディング</u>および<u>スレッドセーフ</u>のお話をしなければならない事になります。

…流石にそこまでやるとちょっとやりすぎだと思うし、前述したように生成と破棄のコントロールレベルで言うと同じことだしね。まぁやり方は書いておいたので手法はお任せいたします。

なぜシングルトンパターンが柔軟性を損なうのか？

ひとことで言うとシングルトンパターンとは

「ちょっとばかしお行儀のいいグローバル変数の集合体」

に過ぎないからです。
そもそもグローバル変数の一番利点であり問題とは

どっからでもアクセスできる

これに尽きます。どっからでもアクセスできるという事は、いつ中身をいじくられるかの特定がしづらく、デバッグを非常に困難なものにします。

一目で把握できる程度の小さいプロジェクトならいいのですが、大きくなるにつれ問題は指数関数的に肥大化します。
この性質は、マルチスレッドとも非常に相性が悪く、グローバル変数自体は余程の事がない限り非推奨です。

ではシングルトンパターンはどうか？インスタンスを取得するメソッドを使用しさえすれば「どっからでもアクセスできる」のです。つまり同じ問題を孕んでいるのです。

にもかかわらず「一見するとグローバル変数ではない」という理由により、許されてしまうパターンです。

違う!!取り扱い注意な事には変わりないんだ!!安易な気持ちでシングルトンパターンを乱用するのは絶対に許さないよ。

しかし、この呪われた血を引くシングルトンも気を付けて取り扱えば有用なものです。気を付けるべきポイントは

- 使用は必要最小限に(せいぜい１つのみが望ましいが…)。
- シングルトンクラスの数も必要最小限に。
- 直接メンバ変数にアクセスしない。必ずメンバ関数からアクセスする。
- 外からシングルトンの状態に変更を加えるような操作は極力しない。
- 可能な限り中の値を書き換えない。
- 中の値を外からいじる際にはクリティカルセクションでガードする。
- 参照のみのメンバ関数は必ず const をお尻につける。

ちなみに、グローバル変数とは違うシングルトンのいい点とは…

- アクセスを集約できる(事故現場を特定しやすい)
- アクセスを制限できる(読み取り専用などであれば事故はかなり減る)

ちなみに、シングルトンがメンバとしてオブジェクトを持つとき、それらもまたシングルトンオブジェクトと同様に、グローバル変数と大差ない事を知っておいてほしい…。

評価は…グローバル変数よりマシ…普通だな!

とか書いてしまうと、使いたくなくなるかもしれないので、脅すのもほどほどにしておこう。ただし濫用にはくれぐれも気を付けよう。

ちなみに今回は Application クラスをシングルトンにする。いつも通りなんだけど main.cpp の main 関数で

```
#include"Application.h"
#ifdef _DEBUG
int main() {
#else
#include<Windows.h>
int WINAPI WinMain(HINSTANCE ,HINSTANCE,LPSTR,int) {
#endif
        Application& app = Application::Instance();
        app.Initialize();
        app.Run();
        app.Terminate();
}
```

とやって、main のお仕事はこれでおしまい。基本的には Initialize(初期化)して、Run(ゲームループ実行)して、Terminate(後処理)するだけです。

うん、Application クラスの存在意義は基本的には

「俺が main 関数にあんましコード書きたくないから作ったクラス」

なので、これは別にシングルトンでも咎められまい

テストコードは main に書いてもいいけどさ…本番コードは嫌じゃない?ちなみに冒頭に #ifdef が書いてあるけど、これは DEBUG バージョンはコンソール出すけど、RELEASE バージョンは出さない仕様にしたいがためで、それ以外の深い理由はないです。

ー実行時に Debug か Release をすぐに判別できるようにですー
ちなみに**コンテストに提出したり、就職活動で企業に提出する時は Release にしてください**。
「Release にするとバグる」的な時はきっちりとバグを修正して提出してください。理由は色々とありますが、最大の理由は

「デバッグの DLL は再頒布パッケージには含まれないんだよぉおおお!!!!」

です。あくまでも Debug は配布向けではないという事です。

シーン周り設計

シーンまわりの設計についてですが、とりあえず僕からひとこと言っておくと「switch 絶対許さないマン」なので、シーン遷移に関しては…「switch なんぞ使ってんじゃねえ!!」です。

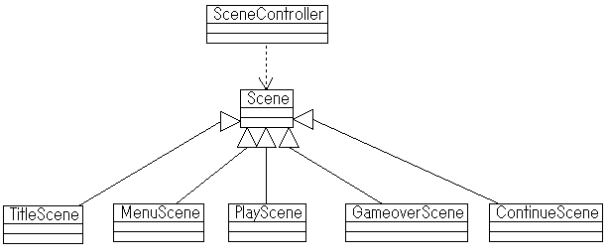
シーン遷移に限らず「状態遷移」に switch を使うなど悪手じゃろ…。switch しか使った事ない人は今から

- State パターン
- メンバ関数ポインタ

を覚えて…どうぞ。ひとまずシーンでは State パターンを使用する設計にしたいと思います。

State パターン

大雑把に図を描くとこんな感じ



SceneController が Scene オブジェクトを持っていて、シーンが切り替わるときに入れ替えるようにすればいい…。

そうなると実装的には

```
///シーン全体を制御するクラス
class SceneController
{
private:
	//シーン管理(現在実行中のシーン状態)
```

```
        std::shared_ptr<Scene> _scene;


public:
        SceneController();
        ~ SceneController ();
        ///シーン状態の変更
        ///@param newscene 新しいシーン
        ///@remarks 基本的に古いシーンは
        ///自動で削除されます
        void ChangeScene(Scene* newscene);
        void Update(Input& input);
};
```

こんなんしといて…

```
void
SceneController::ChangeScene(Scene* scene) {
        _scene.reset(scene);
}
```

こうやっておいて…

```
void
SceneController::SceneUpdate(const Input& input) {
        _scene->Update(input);
}
```

とでもしておけばいい。

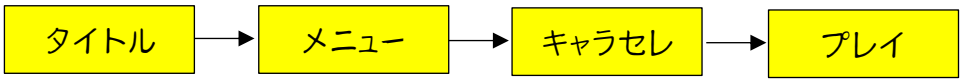
しかしここでも更に柔軟に考えてみよう。例えばポーズシーンがあるとしたら、ポーズ解除後にゲームシーンに戻りたいですね？

Scene が1つだけしかないならば、戻り先がわからない…さらにメニューが階層型になっている場合…例えば、メインメニュー→コンフィグ→キーコンフィグとなったとして、キーコンフィグが終わった→コンフィグ→メインメニューに戻りたいとします。

さらにスタック構造にする

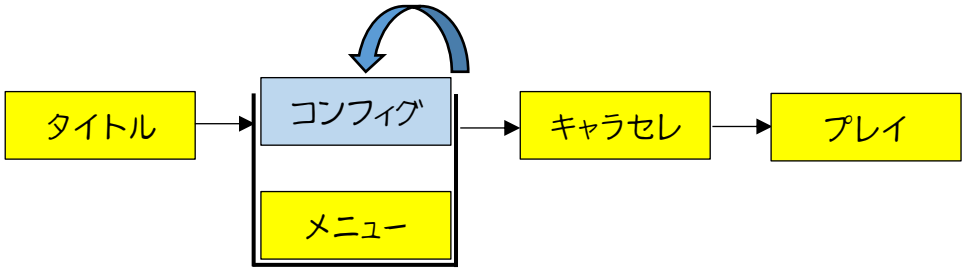
こういう時に便利なのは「スタック(stack)」という仕組みですが…ご存知？
https://ja.wikipedia.org/wiki/%E3%82%B9%E3%82%BF%E3%83%83%E3%82%AF
に見られるように FILO な仕組みです。この仕組みが何の役に立つのかと言うと…

通常の流れが

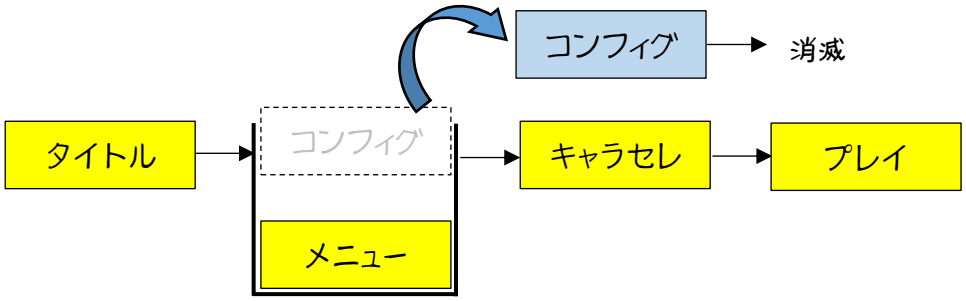


てな感じだとすると

メニューの子メニューがあるなら、こういう風に push で積んで……



コンフィグが終わったら pop で自動的にメニューに戻る…と



こうしたい。モチロン積むのではなくただ単に入れ替えたいなら、push 前に pop すればいい。もし、一番下の階層から一気に別の全く違うシーンに行きたいときは、面倒だけど全部 pop して empty なのを確認して切り替えればいい。

たとえばポーズ(停止)からコマンド表の表示とか、多重階層メニューとかを実装したいときに利用するといいでしょう。

とすると実装としては…

```
#include<stack>
#include<memory>
class Scene;
class Input;
///シーン管理クラス
class SceneController
{
```

```
private:
	std::stack<std::shared_ptr<Scene>> _scene;
```

てな感じで用意しといて

```
void
SceneController::ChangeScene(Scene* scene) {
	_scene.pop();
	_scene.emplace(scene);
}
```

で、

```
void
SceneController::SceneUpdate(const Input& input) {
	_scene.top()->Update(input);
}
```

と言う感じで、元と同じ操作を実装できます。でもこのままぢゃわざわざこんな構造にした意味がない。

そう、スタックしないとね。

ここまで話したことは、一応設計なんだが、正解があるわけじゃない。敢えて指針を上げるならば、可能な限り柔軟にしておく事。現在考えられる将来の可能性を潰さないような設計にしておけばいい。それが難しいんだろうけどね。

リファクタリング

さて、最初に色々と設計とか、デザインパターンとかに関する話を聞いてきたかと思うが、それ(クソコードになるの)を恐れてプログラムできなくなったり、設計終わるまで実装しないとか、そんな事になっては本末転倒です。

…ホンマ手が止まってしまうのが最悪の状況だからな!!それを…闇の波動を感じたら直ちにコーディング•シタマエ!!

とは言え、まだ 2 年生になったばかりの人も多いでしょうから、あまり最初から飛ばしてガッチリやりすぎるのもお勧めしないし…そもそも設計など移ろいゆくものなのだ。実装してみなければ分からない事も多いしね。

そういう場合はさっさと実装してしまった後に「リファクタリング」してもよい。

リファクタリングとは何か？

定義

リファクタリングとは「アプリの挙動を変えずにコードを改善する」事だ。

最も簡単な部分で言うと、既にクソコードになってしまっている自分のコードを、動作に影響が及ばない範囲で改善する所からかな。

例えば

- 変数や関数やクラスや構造体の名前を適切なものに変更する
- マジックナンバーを徹底排除する
- 実行されないコードブロックを削除する
- 不適切なコメントがあったら修正する
- コメントアウトを削除する
- {}などのルールを統一する。if や for の{}の省略は基本許さない方向で

などですね。

これくらいであれば、エディタの「リファクタリング」機能だけでも十分対応可能なので、やってみてほしい。

おかしな名前

不適切な変数、関数名というのは…ある。チームでやってると特になんだが、よくあるクソネーム(˘ω˘)について挙げると

- aとかbとかぱっと見で意味が分からん奴
- plessプレス？とかgoolゴール？とかbottonボタン？とかスペルを間違えてる奴
- jiki,tekiなど、ローマ字使ってる奴(英語くらい調べようね!)
- a1～a14など「配列使えや!」って思える奴
- ax,ay,az,bx,by,bzなど「構造体使えや!」って思える奴
- 意味と適合してない名前(例えばdogなのにcatとかやってる)

はい、既にご自分のコードにこういうのがあると自覚してる人は、悔い改めて、どうぞ。まぁ、さっさとエディタで「リファクタリング」して、名前の変更してください。

クソコメント

つぎにクソコメントについてですが、分かりやすい奴をいくつか

- 嘘コメント
- 無意味なコメント

- 意図がわからないコメント

はい、嘘コメントは、その名の通り、嘘のコメントです。書いてる事と実際がちがうってやつで無意味なコメントとは、「//ふぁっきゅー」とか絵文字とかですね。一人で作るならいいんですが、クソ忙しい時にそういうコメントを同僚が書いてると、イラっとします。ノイズになりますしね。
また、意味がないコメントとは、こういうものも指します。

```
x++;//x をインクリメント
```

↑これ、意味ないよね？流石にこれはあまり見ないんだが、実際にたまにあるのが

```
Application& app=Application::Instance();//アプリケーションインスタンスを代入
DrawGraph(中略);//ドローグラフ
```

↑ウソみたいだろ？本当にあったんだぜ？コードを見て意味が明白なものはコメントを書く必要が無いです。寧ろ、最初の例で x をインクリメントするなら「なぜ x をインクリメントするのか」を、理由と目的を書いてください。

最後に「意図が分からないコメント」ですが、これは無意味なコメントにも書きましたが、コメントはしているんだけど、「意図」が分からないものです。意味は分かるけど意図がわからない。そういうコメント。

「意図を伝える」これは他人にソースコードを見てもらう時や、他人と仕事する時に大事な事だ。もしかしたら一人でやるときにも重要かもしれない。1 週間後には「なぜそう書いたのか」を忘れてしまっているのかもしれない…自分が書いたコードでもだ。自分が書いたコードならそのうち思い出すだろうが、はっきり言って時間の無駄だ。

だから例えば

```
//現状だけを見るとわざわざテーブルを作るのが面倒に思えるが
//あとでキーコンフィグ機能をつけるために敢えて作っている
```

などという長文コメントもやぶさかではない。何故なら「いちいち説明してられない」「いちいち思い出してられない」からだ。

ちなみに、関数の頭にコメントを付けると思うが、おそらく

- 関数の働き
- 引数の意味

- 戻り値の意味

を記述する事と思う。これはいい。寧ろ書いてほしい…が、ここで一つアドヴァイス

関数のコメントは(とくに public は)ヘッダ側に書こう

バババーン!!どうだまいったか!!!

これさ、理由を書いておくとやね。クラスの「利用者側(クライアント)」は基本的に中身に興味がなく、「使い方」がわかればそれでいいわけよね？そしたら、クライアント側からするとまず見に行くのはヘッダだよね？

実感わかない？

例えばさ、DxLib の DrawGraph の「実装」とか見たい？いや、興味があるとかじゃなくて、自分の**ゲームを作るために DrawGraph の実装コードを見ることが必要か？**って話。

ちなみに実装コードはこうなんだけど

```
int DrawGraph( int x, int y, int GrHandle, int TransFlag )
{
	int Result ;
	DXFUNC_BEGIN
	Result = NS_DrawGraph( x,  y,  GrHandle,  TransFlag ) ;
	DXFUNC_END
	return Result ;
}
```

これが君がゲームを作るために必要なのか？どうなんだ？ええ!?

うん…まぁ、必要ないよね('•ω•`)
ちなみに DxLib の中でも cpp にはコメントがあまりなく、ヘッダ側に用法や注意が書いてあります。

分かるやろ？というわけで、関数のコメントはヘッダ側に書けって事。ちなみに今回の Input クラスはこんな感じで書いてます(ヘッダー)。

```
///入力周りクラス
class Input
```

```
{
private:
        vector<unordered_map<std::string, PeripheralInfo>> _inputTable;//プレイヤー番号と入力対応テーブルのセット
        unordered_map < PeripheralInfo, std::pair<unsigned short, string>, PeripheralInfo::HashFunc> _inputMap;//入力情報と利用情報のセット
        vector< unordered_map<string, bool>> _currentInputState;//現在の押下情報
        vector< unordered_map<string, bool>> _lastInputState;//現在の押下情報
public:
        Input();
        ~Input();
        ///接続中パッド数を返す
        int GetConnectedPadCount()const;

        ///プレイヤー数(パッド数とは関係ないよ)を設定
        void SetPlayerCount(int pcount);

        ///コマンド設定
        ///@param pno プレイヤー番号
        ///@param cmd コマンド文字列
        ///@param periNo 入力番号
        ///@param code 入力コード
        void AddCommand(unsigned short pno, const char* cmd, PeripheralType periNo, unsigned int code);

        ///入力情報更新
        ///@remarks 毎フレーム呼び出してください
        void Check();
        void UpdateHistory();

        ///押下チェック
        ///@param playerNo プレイヤー番号
        ///@param cmd コマンド文字列
         bool IsPressed(unsigned short playerNo, const char* cmd)const;

        ///トリガーチェック
```

```
        bool IsTriggered(unsigned short playerNo, const char* cmd)const;

        ///現在の押下状況を返す
        const vector<unordered_map<string, bool>>& CurrentState()const {return
_currentInputState; }
        ///直前の押下状況を返す
        const vector<unordered_map<string,bool>>& LastState()const { return _lastInputState; }
```

ちょっと抜けてたりすることもあるけど…ヘッダにほとんどが書かれている事が…分かるだろう？

で、別に cpp に書いたらイケナイというわけではないが、h と cpp 両方に書いてたらコメント修正の際に修正箇所が 2 か所になるのでメンドクセー＆ミスが増えるだけなので、僕は cpp 側に関数の説明コメントは殆ど書きません。

クソコード

もうちょっと範囲を広げて、世に言う「クソコード」とは何かを書いてみます。

- コピペコード(関数化しろ!!)
- どっからか拾ってきたパッチワークコード(理解してから自分のものにして使え!)
- スペルミス(Google 先生に聞けばええやろ)
- 統一されてない名前付け規約(キャメルケースとスネークケースが…)
- クソ深ネスト(見にくくない？関数化しろ!!)
- マジックナンバー(お前それサバンナでも意味わかるの？)
- 絶対通らないコード(さっさと消せ!!)
- 揃えてないインデント(帰れ!!)
- クソ長関数くん(見づらくね？関数分割しろ!!)
- クソ長ファイルくん(いや、分けろよ)
- 大量のフラグ管理変数(それ、そんなに必要？)
- warning 大量発生(目を背けるな!!)

うん、
設計とかのマズさとかはさ、経験も少ないしさ、俺も大目に見…見ますよ。だが↑のクソコード…お前だけは許さん!!

これらも僕の目に触れる前に今のうちに修正しておく事をお勧めします。

プレファクタリング

ちなみに「プレファクタリング」という造語も生まれました。リファクタリングという概念が出て以降

「クソコード書いてもあとでリファクタリングするからいいでしょ？」

的な事を言う阿呆がチラホラ出てきた

ダメでしょ

そういう時期がアッーたので、生まれた造語。あまり流行らなかったけど、言いたい事はわかるし、もっともである。なお

https://en.wikipedia.org/wiki/Prefactoring

英語のページはあるが、それ以外の言語には翻訳されていない模様…どこで差がついた。

まま、それはええやろ。とりあえずプリファクタリングっていうのは、リファクタリングしやすいような準備をはじめからしておくって事やね。リファクタリングをするからといって滅茶苦茶にコーディングしていいわけじゃない。でも、気を付けていても気が付いたらクソコードは混じっているものだし、あんまり気にしちゃ先に進めない。

だが、無策でコーディングするのは…無謀だ。Hip の You(匹夫の勇)だ。

では、コーディングする前にしておく策とは何だろうか

例えば「コーディング規約を決めておく」事だったり、「言語に対する知識と理解を深めておく」事だったり、「"プログラミングの原則"を一通り見ておく」だったり、「1 人開発でもバージョン管理システムを導入する」だったり、「友人とコードチェックをする習慣を設ける」だったり、プログラミング以前の部分の改善策である。

ワシも教える立場だけど、いくつかは実践しているぞい

一応のコーディング規約(目安)

コーディング規約ですが、これは僕が個人的にやってる奴なので、正解じゃないです。真似しなくていいです。あくまでも一例として見てください。設計や書き方に正解はないのです。でも統一感は守ろう…な!

- 変数は先頭小文字のキャメルケース(camelCase)
- 関数、クラス、構造体、型名などは先頭大文字のキャメルケース(CamelCase)
- define 系の定数は大文字スネークケース(SNAKE_CASE)
- enum などの定数は小文字スネークケース(snake_case)
- メンバ変数の頭には_アンダーバーをつける
- グローバル変数には g_ジーアンダーバーをつける
- 変数も関数も型も名称は英語をベースとする
- public 関数コメントはヘッダ側に書く
- 関数コメントは doxygen 規約に合わせる
- タブは 8 文字
- よそのヘッダをインクルードする前にプロトタイプ宣言で事足りるか検討する
- 関数の引数はプリミティブ型以外は参照渡しにする。
- 参照渡しは可能な限り const 参照にする。
- クラス内の変数を参照するだけの関数(読取専用インターフェイス)は const 関数にする
- C++11 以降の規約に従う(NULL→nullptr、true/false→TRUE/FALSE)
- コンパイル時定数は const ではなく constexpr を使用する
- 変数宣言時に auto が可能な部分は auto を使う
- 可能な限り final や override などの修飾子を指定する
- メンバ変数の初期化は可能な限りヘッダ側で行う
- STL の恩恵を可能な限り享受する(車輪再発明禁止)
- シングルトンクラス作るときはコピー、代入禁止を忘れない！
- コピペしない！
- ポインタ型を極力使用しない！
- ポインタを使用する際にはスマートポインタを使用する
- シングルトンを乱用しない！
- switch 文を撲滅する
- メンバ変数に bool 型のフラグを使う場合、落ち着いて必要か再考する

先ほども言いましたが、これはあくまでも「僕の」コーディング規約なので真似する必要はありませんが、一人で開発してても、この表くらいは作っておいた方がいいです。作ってもしれっと違反しますんでね…？作ってないとしたら全てクソコードになる可能性が微し存…？

ひとりでもバージョン管理システム(Git)

GitHub でも VisualStudioTeamServices でも CloudSourceRepositories でも AWSCodeCommit でもいいんだが、とにかくバージョン管理システムは使っておこう。

クラウドにおいておけばコードが壊れたら～だの、そういう心配は一切ご無用。多少お金を払ってでもクラウドを使用する事をお勧めする。

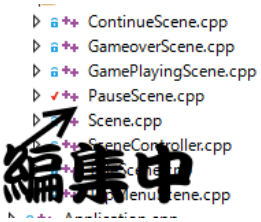
まぁ、GitHub とかだとタダだけどな。

みんな Git の使い方はわかるかな？あ、わかる。僕は VisualStudioTeamServices(AzureDevOps)を使用しています。無料枠でかなり使えますんでね。

あと、これだと、VisualStudio との親和性が高いんで、非常に重宝する。

MS の回し者じゃないよ？便利だから使ってるタケダヨ!!

ちなみにクラウドと言えば Dropbox だが、あれ最近使用可能 PC を 3 台に制限したとかって話題になってるんで、気を付けた方がいい。ちなみにここでアカウント作っておいて、VS からプロジェクトを作る際にここのリポジトリを設定すればエディタ上で



編集中のコードにチェックマークが入ったりする

まぁ、これは関連付けさえしてたら GitHub のプロジェクトでも可能なんだけどね。Azure だと色々自動なので便利だよってだけ。

Git で忘れがちなのが、コミットからのプッシュだ。コミットばかりしてプッシュを忘れるので、

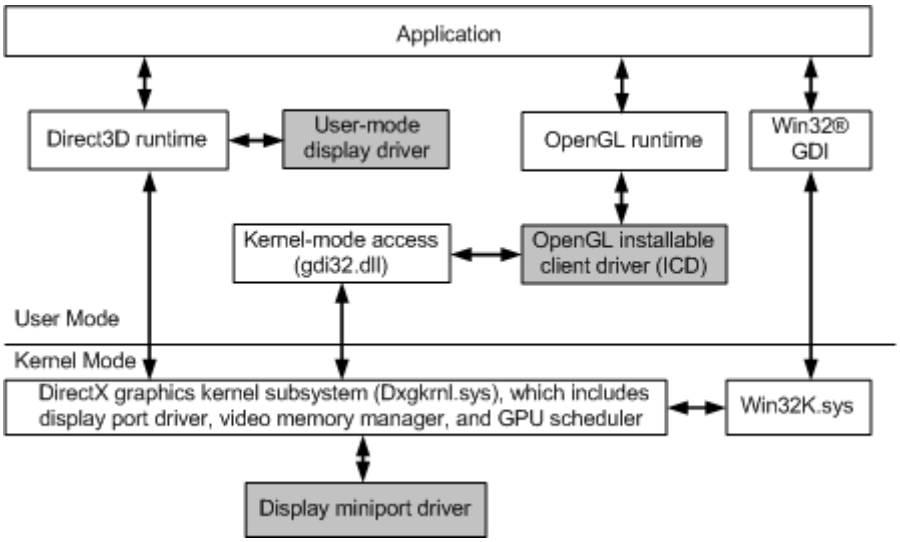
気を付けよう…な!!

プログラミングの原則

うーん。あまりガチガチにプログラミングするのはお勧めしないが、後で苦しまないための原則があることくらいは知っておこう。
https://qiita.com/ryotanatsume/items/018cae5c5be8faba367a
おっと既にYAGNI原則に違反してる気がするなあ。まぁ今年はキーコンフィグ作る予定だから多少はね？
という事で、あまりガチガチにならないよう、原則として

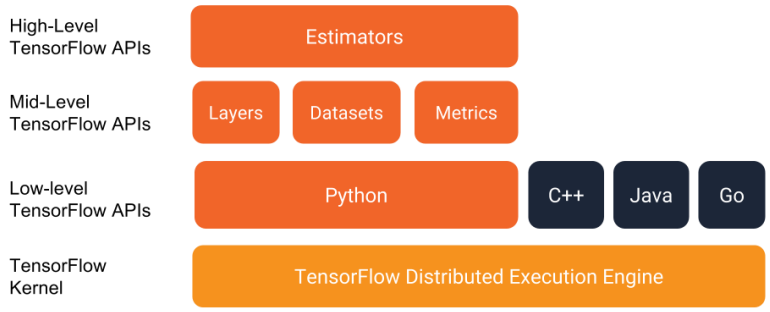
- シンプルさを意識しよう(KISS)
- 重複したコードは書かないようにしよう→重複したデータもメモリ上に置かないようにしよう(DRY)
- 未来のことを考えすぎてコードを複雑にしないようにしよう(YAGNI)
- コーディングは『読みやすさ』を重視しよう(PIE)
- 抽象化レベルを統一しよう(SLAP)

最後のSLAPがちょっと分かりづらいんですが、これは抽象化レベルによって、構造を階層化して、抽象化レベルが混ざらないようにしようってことです。例えばDirectXのレイヤーは



となっており、ハードウェアに近い方から
Kernel→User→Applicationと分かれています。ここはきっちり分けとこうやということ。

DirectXとかに限らず機械学習とかでも



まぁこんな感じでAPIとかってのは大抵この原則をまもっていますね。たとえば今からみんなが作るゲームにおいては大雑把に
①DxLibの関数を直接叩く層
②①を使ってゲームを構築していく層
に分かれるかと思います。ちなみに今回の場合②のトップを叩く層がmain関数になっており

```
#include"Application.h"

#ifdef _DEBUG
int main() {
#else
#include<Windows.h>
int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int) {
#endif
	auto& app=Application::Instance();
	app.Initialize();
	app.Run();
	app.Terminate();
}
```

というわけです。最下層のレイヤーの使用者がさらに上のレイヤーから使用され…を最上位までさかのぼるとmainになるというわけです。

例えばmain一本でゲーム作るときは、層が１個しかない状態ですね。

今回はDrawいじる層とInputいじる層を最下層にしたいかなと思っています。

switch文撲滅のために…

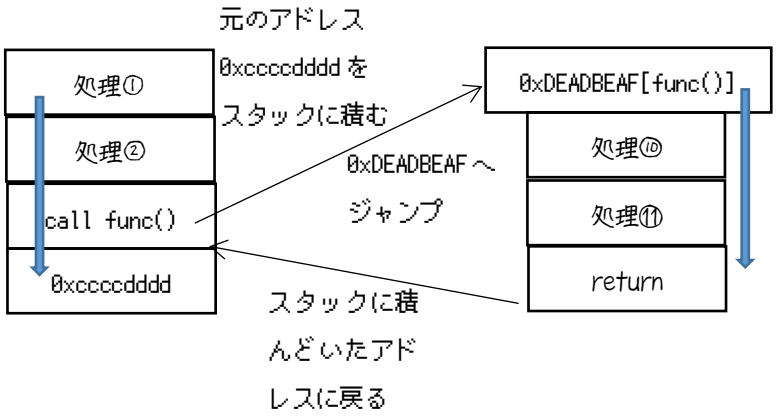
ホントしつこいな、俺。

一応、
シーンの遷移は State パターンでやってのけてるんだけど、例えばフェードイン、フェードアウト的な仕組みにしたい場合はどうしたもんかなと思います。

メンバ関数ポインタ

皆さんはメンバ関数ポインタというのをご存知でしょうか？関数ポインタとも違いますよ？メンバ関数ポインタです。関数ポインタを知らない人もいるかもしれませんので、関数ポインタについて、軽く話しておくと…変数だけじゃなく、関数にもまたアドレスがあるんですよ。

結局「関数」なんて言っても、内部的には「ジャンプ」して戻ってるだけです。関数呼び出しというのは、メモリの特定の番地に処理の内容が書いてあって、return もしくは関数の終わりに来たら呼び出し元アドレスに戻るという…ただのそういう仕組みです。



なんとなくイメージ掴めますかね？ちなみに関数呼び出しにおいてはスタックメモリが重要な役割を持っています。意識する必要はないけど、興味がある人は Lua と C++の連携とかやってみると、スタックと関数の関係が理解できると思います。

まぁ、学習コストがそれなりに高いのでお勧めはしませんけれども…興味を持ったらね。ちなみに Lua などの言語の関数やメモリ確保周りの実装とかを見とくと所謂「システム周り」の理解が深まるので、ゲーム会社に入って「システム周り」とやらをやりたい人におすすめです。

…まぁ、ともかく関数も「アドレス」に過ぎない事がオワカリイタダケタダロウカ？

関数がアドレスであるため今回やるメンバ関数ポインタも利用できるし「コールバック」の仕組みやタスクシステムも成り立つわけです。今回は

関数呼び出しも所詮はアドレスジャンプ!!はっきり飛ぶんだね

という事をご理解いただければよろしいかと思います。
というわけで、ステートが変わるたびに内部変数を変更し、switch 等で処理を分けてしまうのではなく

「関数ポインタを定義し、実行される関数を切り替えていく」

事で状態遷移を実装していきたいと思います。

まずは普通の関数ポインタのおさらいですが、例えば足し算引き算の関数が

```
int Add(int lval,int rval){
	return lval+rval;
}
int Subtract(int lval,int rval){
	return lval-rval;
}
```

と定義されているとします。この時に戻り値の型、引数の数と型がそれぞれの関数で一致している事が必須条件です。

こうしておいて、あとは関数を示す変数を作るのですが、んまぁちょっと変わってますが

戻り値 (ポインタ変数名)(パラメータ);
つまり

```
int (*func)(int, int);
```

のように宣言します。変数名は func です。こいつが関数のアドレスの入れ物になります。あとは

```
func=Add;
```

のようにすれば func は Add 関数として動作します。普通のポインタの時のように

```
int result=(*func)(15,9);
```

とやれば result には 24 が入る事になります。その後で

```
func=Subtract;
int result=(*func)(15,9);
```

とやれば result には 6 が入る事になります。例えば加算か減算かをランダムにしたければ

```
func = (rand()%2)==0?Add:Subtract;
int result=(*func)(12,35);
```

等とやれば 47 になるのか-23 になるのか、実行するまで分からないわけです。まぁこんな使い方はあまりやりませんけどね。大体コールバックとかがよくある使い方とかかな。

それはそれとして、この「関数ポインタ」…クラスのメンバ関数でも使えるのか？という話。結論から言うと、使えます。ただ、文法がちょっとややこしいです。

まず、メンバ関数ポインタ変数の宣言ですが、

戻り値の型 (クラス名::*メンバ関数ポインタ名)(パラメータ);

と言う感じになります。具体的にクラスで書いた方が分かりやすいかもしれないので書くと

```
class Test{
private:
	void UpdateState1(const Input& input);//指し示す関数①
	void UpdateState2(const Input& input);//指し示す関数②
	void UpdateState3(const Input& input);//指し示す関数③
	void (Test::*_updater)( const Input& input);//メンバ関数ポインタ
};
```

これで_updater がメンバ関数ポインタになります。ポインタ指定子(*)の前にメンバ指定(Test::)をしなければいけません。ここがちょっと面倒ですね。

さて、それではメンバ関数ポインタにメンバ関数のアドレスを代入する部分ですがこれもちょっと違ってて

メンバ関数ポインタ = &クラス名::対象関数;

のように、&アンパサンドを付ける必要があります。
具体的コードだと

```
_updater=&Test::UpdateState1;
```

となります。ここでも若干や面倒ですね。でも一番ややこしいのは呼び出しです。これをクラス内の別の関数から実行するとなると…

(オブジェクト名->*ポインタ変数名)(パラメータ);
で関数呼び出しを行います。

自オブジェクトから呼び出すには
(this->*_updater)(input);
のようにします。呼び出し側は現在がどの状態なのかなど気にせずに呼び出せます。ちなみに状態の切り替えは切り替え用の各関数やイベントが行う事になると思います。

```
void
PauseScene:: UpdateState1(const Input& input) {
	if (_frame <= scale_time) {
		_frame = (_frame + 1);
	}
	else {
		_frame = 0;
		_updater = &PauseScene::UpdateState2;
	}
}
```

のような感じで遷移を行います。そうすると実行部分自体はシンプルになり、各関数も「次のステート関数」にだけ目を向けてればいいので、考えをまとめやすくなります。

似たようなことをラムダ式でやらないか？

やれますねえ!

ラムダ式自体はご存知だと思います。
ただしメンバ変数に利用するにはもうちょっと知識が必要です。それはラムダ式の「型」がなんなのかという知識です。

いつもだったら、ラムダ式を代入する先の変数は auto 指定をしていると思うんですが、これって型はどうなってるんでしょうか？

ラムダ式は関数オブジェクト(ファンクタ)に代入されます。さて、ファンクタの代入先の型はどう作ったらいいんでしょうか？

それは function 型を使います。まず#include<functional>します。そのうえで

```
std::function< 戻り値(パラメータ) > ラムダ変数;
```

という感じで宣言します。このラムダ変数に対して、ラムダ式を代入することができますので、内容の切り替えが可能!!というわけです。

```
ラムダ変数 = [ バインド ]( パラメータ ){ 処理 };
```

で代入できます。で、今回みたいに状態遷移に利用したい場合には予めメンバの const ラムダ変数に初期化子で代入する事になります。例えば

```
const std::function<void(const Input& input)> lambda_fadein;
const std::function<void(const Input& input)> lambda_fadeout;
```

のように宣言しておいて、コンストラクタ初期化子(初期化リスト)で

```
PauseScene::PauseScene(SceneController& controller):Scene(controller),
lambda_fadein ([this](const Input& input) {
        DxLib::SetDrawBlendMode(DX_BLENDMODE_ALPHA, _frame);
        DxLib::DrawBox(0, 0, 640, 480, 0x000000, true);
        DxLib::SetDrawBlendMode(DX_BLENDMODE_NOBLEND, 0);
        --_frame;
}),
lambda_fadeout([this](const Input& input) {
        DxLib::SetDrawBlendMode(DX_BLENDMODE_ALPHA, _frame);
        DxLib::DrawBox(0, 0, 640, 480, 0x000000, true);
        DxLib::SetDrawBlendMode(DX_BLENDMODE_NOBLEND, 0);
        ++_frame;
}){~
```

のようにします…何やってるかは…分かりますよね？あっ(察し)…ふーん。

とりあえずですね？普通はメンバ変数に const 使うと宣言時に代入してないと怒られるん

ですが、const ついてても初期化子で代入すれば、コンパイラ君は ゆ る す よ 。

という事で、初期化子部分で代入しといてあげます。ちなみに僕はヘッダ側にコードを書きたくないので、このような感じになっていますが、別にヘッダ側にコード書いても構いません。

ちなみに、const の初期化については、ヘッダ側でやる事を推奨されているのですが、今回のラムダ式のように、中身に DxLib 関数等を使用している場合、ヘッダで DxLib を include しなければならなくなり、非常に面倒なので、わざわざ cpp の初期化子で初期化しています。

その上で

```
_lambda=lambda_fadein;
```
を
```
_lambda=lambda_fadeout;
```
としたりして、実行を切り替えていく事は可能なのです。

さて、状態遷移の武器が色々と増えましたね。君はどの状態遷移を使うかい!?

君は switch 文を使う事もできるし、使わなくてもいい。どちらを選ぶかは君の自由だ。状況と好みで判断してくれ。