

アクションゲームプログラミング

目次

概要ミミミ	5
はじめに	5
半年後から『ちまちま』と就職準備が始まります	6
この授業の流れ	9
だいたい順序	10
環境設定	11
C++についてミミミ	12
C 言語復習(メモリ周り)	13
メモリとメモリ確保	13
配列と構造体のちょっとした違い	15
ポインタとアドレス	19
概要	23
カプセル化の意義	25
継承	26
ポリモーフィズム(多態性)	28
クラス(構造体)内関数	29
関数内クラス構造体	30
入れ子構造体(クラス)	31
自己参照ポインタ	31
前方宣言(プロトタイプ宣言)	32
参照	34
namespace と using namespace	36
C++における::の役割	40
クラスの作り方	41
private と public	41
クラスメソッド	43
メンバ変数(またの名をフィールド)	44
メンバ関数(またの名をメソッド)	45
メンバ関数ポインタ	46

static 変数,static 関数.....	50
const	52
継承	53
ポリモーフィズム.....	55
new と delete と delete()と... ..	55
コンストラクタ/デストラクタ	56
コンストラクト時引数.....	58
初期化子	59
参照をメンバ変数として持つ方法.....	61
仮想関数,純粋仮想関数.....	63
typedef	65
関数オーバーロード.....	67
オペレータオーバーロード(演算子オーバーロード).....	68
enum	70
enum class/enum struct.....	71
STL(StandardTemplateLibrary).....	73
std::vector	73
std::map	76
std::string	78
ストリームについてちょっとだけ.....	80
STL って組み合わせられるのよ?	80
テンプレートについて.....	81
関数テンプレート.....	82
クラステンプレート.....	83
新しい C++ 言語(C++11/14).....	84
nullptr	84
auto	85
ForEach 的な for	86
ラムダ式	87
参考資料	89
C++ 言語	89
スマートポインタ.....	91
shared_ptr	92
weak_ptr	93
unique_ptr	94
開発初期段階(初歩設計的なのと実験).....	95

設計的なの	96
一般的なクラス設計法(ストーリー法)	96
クラスの作り方	98
読み込み→表示(第一段階)	100
ただの表示	100
DrawRectGraph で画像の一部を表示	103
等幅切り取りアニメーション	104
等幅じゃない切り取りアニメーション	105
後々楽になるちょっとした部品を作っておく	107
バイナリ読み込み	111
相対パスをアプリの相対パスへ変換して画像読み込み	114
読み込んだデータをもとに切り抜く	117
読み込んだデータをもとにアニメーションさせる	117
中心点適用	119
パッド(キーボード入力)への対応	120
要件	120
実装	120
テスト	121
左向きでガタガタしないようにしよう	122
そもそもガタガタするのは何故?	122
ガタガタへの対処	122
ジャンプとか、基本動作を試みよう	124
地面を定義	124
ジャンプと重力	125
着地	127
friend	128
アニメーション切り替え	130
状態遷移	130
メンバ関数ポインタによる状態遷移	130
状態遷移をチョット修正	132
リファクタリング①	135
クソコードとは	135
リファクタリングとは	136
現状のコードをリファクタリング	137
Player クラスのコピペコードをどうにかする	137
main→Game クラス	139

Game クラスの背景系制御->Background クラス	140
インターフェイス系を HUD クラスに.....	141
ひとまずそこまでやれば…	141
ちょっとここでツール変更(重要!!).....	142
5/3 更新(攻撃矩形追加).....	144
矩形を表示する準備.....	148
矩形を表示する際の注意点…	149
敵さん入りまーす	151
雑魚の皆さん.....	151
設計?	152

概要ミミツミ

はじめに

今回は新2年生向けにまずは、小手調べとして『2D アクションゲーム』を作っていきます。君たちにとってはアクションゲームなんて簡単に



と言いたいところでしょう…最初だしね。皆の力量を測る意味でも 2D アクションゲームはちょうどいいかと思いました。

専攻科 3 年生にとってはおさらいになりますし、簡単すぎてしまうので来年度の就職に向けてフル 3D で作ってみるとかしてもいいと思います。もしくは、昨年に鍛えた力量に対して自信がない人はもう一度 2 年生になったつもりで授業を受けても良いでしょう。

でも、3 年生と 2 年生を同じ評価基準にすることはできません。そんなもん 3 年生有利に決まってるからです。

また、この授業は進行が『速い』と思う人もいるかもしれませんが、



デメエが遅すぎるんだよ…

どうあれ、ゲーム業界に行きたい人は多いのです。競争率は高いのです。レッドオーシャンなのです。生き馬の目を抜く世界なのです。遅い自分に甘んじていては生き残れない世界であることは自覚してください。

大半が次年度就職年次であることを考えると、ここでスピードを緩めるわけにも行かないの

です。ご了承ください。挫折する人はどうぞ挫折してください。

10単位以上落としますけれどもね？ 留年確定しますけれども

それでもよければどうぞ。

それがイヤなら放課後に残るなり家でやるなりして追いついてください。可能な限りその日のうちに、遅くとも土日間に授業に追いついてください。

何のためにこの学校のこの学校に来たのか、特に次年度就職年次は自覚しておいた方がいいですね。

君や保護者が学費として支払う 300 万円~400 万円を無駄にしないようにね。これだけ稼ぐのは大変だよ？ 稼ぐこと自体は大したことじゃないけど、貯めていくのが大変なんだ。その自覚は持っておいた方がいい。



まあというわけで覚悟は決めてください。

非常に申し訳ないですが、僕の言葉は『次年度就職年次』に向けた話が多くなってしまうので、専攻科2年生には苦勞かけます。苦勞かけますがどの道1年後には自分の状況なのでそのつもりで聞いておいてください。

半年後から『ちまちま』と就職準備が始まります

半年後と言えば10月半ば。来年就職の人はこの辺には戦える状態になっておく必要がある

のです。戦える状態と言うのは自分で自分ならではのゲームを作れるようになっておくという事です。

来年の今頃から就職活動のための制作しても遅いんですよ。来年の今頃は『就職活動』ですから、そのための武器は4月までには揃えておくべきなのです。もっと言うと2月くらいから募集する会社(バンナムとかレベルファイブとかコーエーテクモとか)もあるので、それに対応するためにはむしろ遅いくらいなのよ。

さて、特に次年度就職年次の人たちは、授業だけでは十分ではありません。そりゃお前、授業だけで十分だったら就職活動そんなしんどくないわい。全員がありとあらゆる検定に受かってるはずだべ？

現実はずじゃないんだ…授業を受けつつ個人作品も作らねばならぬという荊の道。それは自覚してくれ特に3年専攻科…。

さて、努力するからには効果を最大限に高める必要があります。どう高めるのかと言うと、自分の方向性を早いうちに決めておくことです。ゲームに必要な知識を極めると言ってもゲームの分野は広いんです。正直全部極めるなんて、大人…プロのゲームプログラマでもそうそうできる事じゃないのです。それを君たちはやろうというのかな？

もちろん全体的に一定の制作力は必要だし、一応のゲームにする力は求められる。しかしそれだけじゃ足りないという…何が足りないのかと言うと、専門に特化した能力だ。

必須の基礎能力

- 一応ゲームを完成させる力(DxLib やゲームエンジンを使ってもいい)
- DirectX や OpenGL を使って基本的な部分(ポリゴンの表示)ができていること
- CG、AI、ハードウェア、周辺機器の基礎知識(全体的な意味では知識だけでも OK)
- Unity、UE4、cocos2d-x などのゲームエンジンを一通り触れる
- VisualStudio がまともに扱える。まともにデバッグができる

トンガリ能力

- グラフィックス系(シェーダとか書けるようになって様々な効果を出せるようになる)
- AI 系(経路探索やナビメッシュをはじめとして、遺伝的アルゴリズム、プロシージャルへの応用など。※ディープラーニングは今の所やめておいたほうがいい)
- ネットワーク系①(通信対戦など…特化するなら4~10対戦できるように)
- ネットワーク系②(いわゆるインフラ。サーバ立てたり DB 作ったり)

- ネットワーク系③(クライアント側。SQL 発行したりだが、フロントエンドはゲーム側も担当することが多い)
- 物理系(様々な衝突判定から、液体などの物理的な挙動までを実装する。数学の知識が超必要)
- システム系(メモリやらスレッドやらを使いこなす感じ。OS の仕組みとかも理解しとく必要がある)

というわけで、自分が就職において何で攻めていくのかを今のうちからぼんやりと考えておいた方がいい(専攻科3年はガチで考えるよ?)。

んで、そんなの分かんねーよって人は仮想的にでもいいので、自分が行きたい会社を決めておこう。別にそこに行かなければいけないというわけじゃないよ?でも、ここだったら行ってもいいなという会社を見据えて努力しないと、間違った努力になっちゃう。

まず決めたいしてほしいのは

- コンシューマゲーム?
- スマホ/ソーシャルゲーム?
- VR 系?

VR 系の場合は機材を自分で購入するくらいの気概が欲しい。Oculus とか Vive は高いので、ハコスコと VR 用コントローラを買えば1万円以内には収まるかな。

参考

<https://www.amazon.co.jp/%E3%82%A8%E3%83%AC%E3%82%B3%E3%83%A0-%E3%83%AA%E3%83%A2%E3%82%B3%E3%83%B3-Bluetooth-%E3%83%B4%E3%83%AB%E3%83%BC%E3%83%A0SDK%E5%AF%BE%E5%BF%9C-JC-VRR02VBK/dp/B076X4M7BH>

次に後期までには決めたいしてほしい事は会社に入ってから何をしたいのかってこと

- グラフィクス系
- 物理系
- AI 系
- ネットワーク系
- システム系

ぼんやりとでいいんで決めておいて、そして、自分で勉強しておいてください。全体教育で教えられる部分は基本の部分だけですわ〜。

最近では特化して VR/AR などとトンがってアドバンテージをとる学生も多いし、本当に授業をやってる間に世の中がドゥンドゥン変化している。

毎年このテキストを変更しているのも、そういう状況を踏まえてなのよね。非常にめんどろだし、俺の寿命を削ってんだけど。

それでも遅いと思うんだ。世の流れについていけないかもしれないんだ。というわけで、学校の中だけ見てちゃダメで

コンテストには必ず応募する事(悪いけど狭い視野じゃ勝てないよ)
学外のイベントに2〜3回は参加するのが望ましい(8耐や勉強会など)

学外イベントでは学ぶことが重要なじゃなくて、外の人は何を考えているのかを感じることが重要だと思っておいてください。更に言うと、自分から能動的に働きかけるのが重要なんです。

この授業の流れ

流れ的には最初に C++ のおさらいと言うか説明と言うかおさらいと言うかなんとか、C++ での開発に関する説明をします。最初の課題なので、こまけえ事はさーっと流します。テンポ速いと思います。

わかるわからんはともかく、この場はとにかく…

つべこべ言わずとにかくやろう

申し訳ないが時間がまるでないのではな…。

分からんところがあったら『即質問』するか、『メモっというて後でググる』をして、なんとか自分の知識レベルを上げてくれ。タイムリミットは決まっているし、俺の体は一つしかない。つまり『ビビって質問できない小心者の僕のニーズに対応してください(つムシ)』はもう通用しないってことなのですよ。

だいたい1の順序

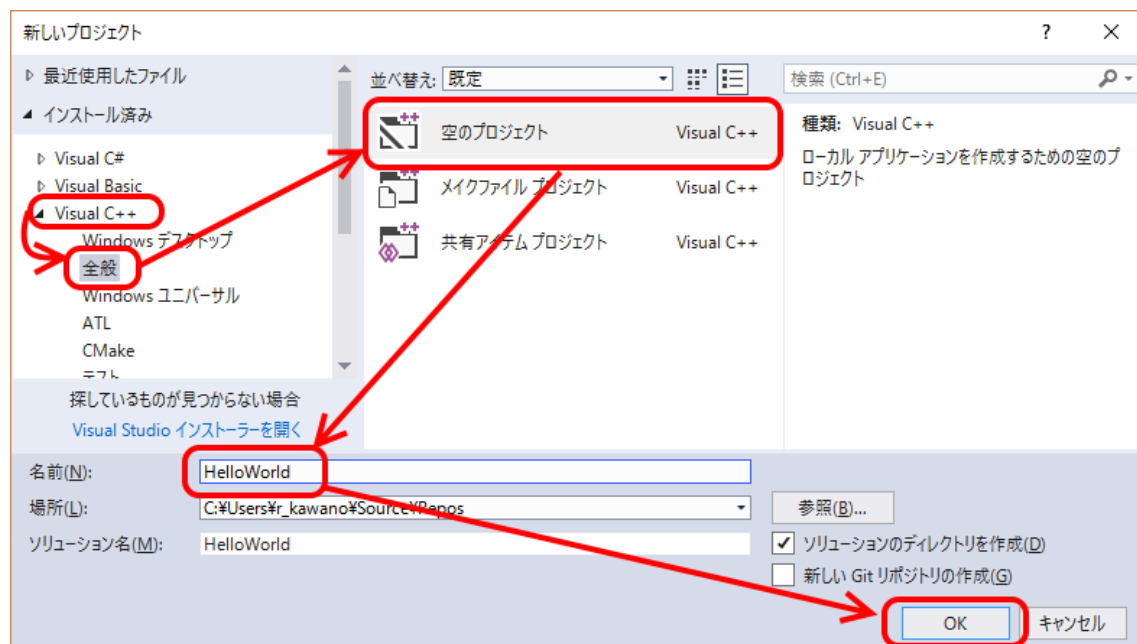
1. 環境設定→
2. 基礎知識の予習→
3. 軽いコーディング規約の説明→
4. 言語や VisualStudio の軽いおさらい→
5. よくあるエラーへの対処集→
6. C++ 言語についての解説→
7. ゲームの開発→
8. 画像の表示→軽く全体設計→
9. 入力への対応→
10. シーン遷移→
11. Prominence で作ったエフェクトでアニメーション→
12. アニメーション情報を元にアニメーション表示(ツールは公開します)→
13. 振り向き反転への対処→
14. 敵との衝突判定→
15. ヒットエフェクトの発生→
16. 斜め床における判定など→
17. 雑魚敵のスポン→
18. 落とし穴や梯子の配置→
19. ステージトラップの配置→
20. アイテム取得→アイテム装備→
21. 敵の思考ルーチン(敵動作パターン)→
22. ボス戦

あくまでもだいたいだし、やってるうちに変更もあり得ますので、これが絶対ではありません。

環境設定

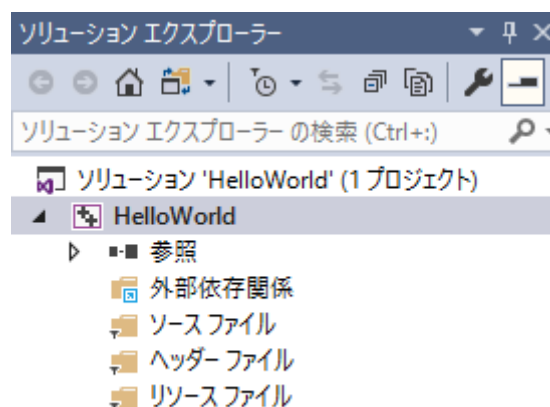
まずは開発をするための環境設定を行います。今回は VisualStudio2017 を使いますので、そこからやっていきます。VS2015 しか扱ったことがない人は実は『罨』的な部分がありますので、ご注意ください。

手始めにはかばかしいですが、それでも HelloWorld からやっていきましょう。



図のように『空のプロジェクト』を作ってください。

『コンソールアプリケーション』で作りたくなりますが、面倒な事になります。必ず図のように『空のプロジェクト』にしてください。



となって、何もないので、追加→新しい項目→CPP ファイルで追加してください。

main.cpp とでも名前を付けてください。

C++らしいHelloworld を書きましょうか。

```
#include<iostream>

using namespace std;

int main() {
    cout << "Hello World" << endl;
    getchar();
}
```

iostreamとかusingnamespace とかは後でぼちぼち説明はするけど、だいたいわかるでしょ？

C++についてミミツミ

序盤ではC++の話をざーっとする。本当にざーっとします。このクラスの人はある程度C++やっていると聞いておりますので、ざーっと話します。

C++はC言語よりも覚えること、理解するべきことが多く『めんどくさいなあ』と思う反面『便利だなあ』と感じることも多いので、何より『たっのしー』と思います。ポジティブな気持ちで学習していきましょう。

一応入門書に書いてないところとかまでやっていくつもりなので、それなりにしんどいかもかも。

でもでも、C++とかやる前にさ、メモリとかポインタとアドレス大丈夫かい？その辺がグタグタだと完全に置いてきぼりの授業だから一応復習する。

C 言語復習(メモリ周り)

メモリとメモリ確保



これがいわゆるメモリ

変数やらなんやらがここに格納される。言い替えると『書き込める場所』って事だ。
日常生活に例えていうと、黒板だかホワイトボードだか、ノートだかそんな感じ。
しかしこの書き込める場所のどこにでも書き込めるわけではなく、当然ルールがある。
ルールがないと、みんなが好き勝手書き込んだら、何書いてあるかわからなくなるだろ？だから『ここからは俺の領域』みたいなルールがあんの。

書き込んでいい領域は OS が管理をしてるので、OS にお伺いを立てて書き込める場所を借りるってイメージ。

変数を宣言すると、その変数のスコープに入った時にメモリ領域が確保され、書き込み可能になる→変数を使う事ができる、というわけだ。まあそんなん分かっと思うけど、もう少し我慢してくれ。分かってないまま使ってる人もいると思うし。

で、メモリを確保するというが、使用するためにはそのアドレスを記憶しておく必要がある。通常の変数を宣言した時にはメモリを確保して、その場所から一定の領域を変数として使う。そのイメージは持っておいてくれ。

ちょっと実験として、変数を宣言して、宣言直後に値を変更するようにプログラム書いて、ウォッチ式でそのアドレス(&変数名)をコピーして、デバッガのメモリにアドレスを張り付けて、プログラムカウンタを F10 で進ませると、そのメモリ部分が変更されるのが分かると思う。当たり前の事なんだけど、一度やっておくだけでも感覚変わってくるのでやってほしい。

変数の宣言だけで、メモリが確保されているのが分かるし、変数の値を更新するとそのメモリの内容が書き換わるのが分かると思う。

じゃあ配列や構造体ではどうなってんのか？

分かっていると思うので、もう答えを言うと『配列や構造体が必要なぶんだけ連続したメモ

り領域』です。『連続した』ってのがポイントなので、頭に入れておいてください。

ちなみに構造体の場合、後述する『アライメント』という面倒な問題と戦う必要が出てくるのですが、それは実際にトラブルになってからでいいでしょう。

さて、配列の話ですが、ちょっとおもしろい実験をしてみましょう。

```
int main() {  
    int a[5] = {1,3,5,7,9};  
    cout << a[3] << endl;  
    getchar();  
}
```

こういうコードを書いたら、コマンドプロンプトにはなんと表示されるでしょうか？やってみる前に予測して、実際にやってみましょう。大丈夫ですか？初歩的なミスをしてませんか？やっちゃったら恥ずかしいですよ？

それではこうしたらどうでしょうか？

```
cout << a+3 << endl;
```

これもどうなるか、予想してください。

まあ、どっかのアドレスが出るんですが、a[3]を格納している場所のアドレスって事は予想がつかますね。

では、このようにしてみてください。

```
cout << 3(a) << endl;
```

え？これ通るの？

通るので、予想して、実行してみてください。どうなったでしょうか？

a[3]と同じでしたね？

何かC言語のイケナイ部分に触れた気がしませんか？僕はこういうのちょっと背徳的でエロティックで好きなんです(ド変態)けれども。

そもそも C 言語は様々な『糖衣構文』によって、プログラムを書きやすく理解しやすく加工されているのだ。

<https://ja.wikipedia.org/wiki/%E7%B3%96%E8%A1%A3%E6%A7%8B%E6%96%87>

コンパイルをかました時点で

$a(3)$ も $3(a)$ も結局はコンパイル後には

$*(a+3)$

という意味になってしまうのです。要はその要素が存在するアドレスの中に入ってる『値』を返すという意味なのです。



そういう事なのです!!

そう考えると、実際 2 次元配列 3 次元配列などというものは、単なる連続したメモリに過ぎず、メモリの的には 1 次元配列と同じという事も分かると思います。

例えば

```
int b(2)(3) = { {1,2,3},{ 4,5,6 } };
```

```
cout << b(0)(5) << endl;
```

なんて書くと、どうなるか…わかるよね？

まあわざわざこんな書き方しても見づらいだけで、誰も得しないので、そういう書き方をし
てはダメだぞ。

配列と構造体のちょっとした違い

これも今更言う事ではないと思いますが、配列と構造体は『連続したメモリにまとめる』という意味においては似てますが、ちょっとしたことが違います。

配列は『並べてるだけ』で構造体は『型』ですから、微妙に違います。

例えば、当たり前の話ですが

```
int a = 0;
int b = 6;
cout << a << endl;
a = b;
cout << b << endl;
```

予想通りの結果だろう…では、配列の場合はどうかな？

```
int a[] = {0,0,0};
int b[] = {1,2,3};
cout << a << endl;
a = b; //怒られます。配列のアドレスなので変更できないんですね。
でも、これならどうかな？
```

```
int* a=(int*)malloc(sizeof(int)*3);
a[0] = a[1] = a[2] = 0;
int b[] = {1,2,3};
cout << a << endl;
a = b; //アドレスは変わるんだけど、メモリの書き換えは発生しない
```

はい、ネタバレをコメントで書いてて申し訳ないんですが、後述する『ポインタ』で宣言しても結局『右辺値をすべて左辺にコピー』することはできないんですよね。

で、そこで構造体に話が戻るんですが、同じようなメモリの配置をするものとして

```
struct A {
    int x, y, z;
};
A a = { 0,0,0 };
A b = {1,2,3};
```



```
cout << a.x << ", " << a.y << ", " << a.z << endl;  
a = b; //メモリが丸ごとコピーされる  
cout << a.x << ", " << a.y << ", " << a.z << endl;
```

こういう事をやっちゃいます。

前述のように構造体は『型』ですから、通常の変数の値代入のように型の内部のメモリの内容がそのままコピーされます。

では、これならどうでしょうか？

```
struct A {  
    int x[3];  
};  
A a = { 0,0,0 };  
A b = {1,2,3};  
for (int i = 0; i < 3; ++i) {  
    cout << a.x[i] << endl;  
}  
a = b; //やっぱりメモリが丸ごとコピーされる  
for (int i = 0; i < 3; ++i) {  
    cout << a.x[i] << endl;  
}
```

結果を見れば分かると思いますが、内部に配列を持っていたとしてもまとめてひとつの『型』になっ
ていればその占有するメモリまるごと『一つの型』という扱いになるため結果として配列の
中身もコピーされるわけです。

ちなみに個人的な好みでいうと、僕は memcpy とか memset とか ZeroMemory とかは嫌いなので
まず使いません。極力こういう構造体を使ったテクニックでコピーすることが多いです。もし
くは後述する STL を使うかです。

あと、これは構造体および後述するクラスについての注意点なんですが、先ほど述べたよう
に構造体は型に含まれるすべてをコピーします。この仕様は場合によってはパフォーマンス
の低下をもたらします。

よくあるのが、構造体を引数として渡す場合。例えば、構造体 A があったとして、

```
void Function(A a){  
    ～中略～  
}
```

```
A b;  
Function(b);
```

のようなコードを書けば当然、Function 呼び出しの時に構造体のメモリコピーが発生する。コピーすべきならばそれでいいのだけれども、ただ中身を参照する場合なんかはコピーコストが大きくなるときがある。3D のデータなんかでは数 100byte 渡してしまうこともざらで、それが何百何千回コールされたりする。

これは非常に高コストとなる。そのためアドレス渡しにして、コストを下げるという対処がとられる。アドレスであれば基本的には int 型と同じバイト数で表現できるため、4 バイト (32bit) もしくは 8 バイト (64bit) を渡すだけで済むわけだ。

```
void Function(A* a){  
    ～中略～  
}
```

```
A b;  
Function(&b);
```

もちろん 4～8 未満の構造体ならばそのまま渡してもコストは大して変わらないので、場合によりけりだ。

ただしノーガードでアドレスを渡せばいいわけじゃない。アドレスを渡す以上は呼び出した関数の中で書き換えられてしまう可能性もある。

これに対処するには const を使用する。const を使うと値の書き換えができなくなる。

```
void Function(const A* a){  
    ～中略～
```

```
}
```

```
A b;  
Function(&b);
```

というわけで、もし値の参照だけならば、それを明示的に示す意味もあって、引数に `const` を付けるのが C/C++ 使いの習慣になっている。ちなみに上の例ではアドレス渡しだけど、C++ における「参照渡し」も同様だ。

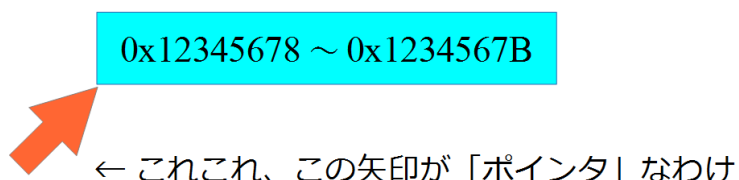
ポインタとアドレス

さて、ポインタですけども、ここはいわゆる難関と言われているところなので、ここだけで1コマ使っちゃう勢いなのでさらっと行きます。

ポインタってのはメモリ上に確保された何らかの情報を指し示すモノなわけです。

```
int a=5;
```

とかであっても、実際にはコンピュータのメモリの 0x12345678 番地やりに配置されているわけじゃん？



そして上の例のように `int` 型なら 4 バイト食いつぶしてるわけじゃん？そしたらポインタはどれくらい食いつぶしてるのかも知ってる必要があるわけ。

更に言うとポインタを利用するためには元の型を知っておかなければ使えねーじゃん？なのでポインタは元の型を覚えている作りになってるわけ。だからポインタの宣言は

```
int* p;
```

みたいに、型名の後にアスタリスクがついていたりするわけ。要は後で利用しやすいように「これは `int` 型を指し示すポインタですよ」って強調してるわけ。

もしそんな必要がなく、ただただアドレスだけを指し示したいのなら

```
void* p;
```

でいいわけだし。ね？ポインタの宣言が“`型名*`”の形式になっている理由とか意識したことある？別になくてもいいけど、なかなか理解できない人は、この理由を意識しておくといいでしょう。

以上のことから、例えば

```
int* a=0x12345678;
```

```
char* b=0x12345678;
```

といったように、全く同じアドレスを指し示している2つのポインタ、これは意味が違う。ぜんぜん違う。ということを理屈的にも直感的にも理解していただきたい。何がちやうかという

と

だから[*a](#)と[*b](#)は同じアドレスですが、全く違う結果になることを知っておきましょう。

また、

```
++a;
```

```
++b;
```

この時のポインタの進み具合も変わります。

[a](#)は `int` 型のポインタなので、`++a` で4バイト進み、`++b` は `char` 型なので1バイトしか進みません。つまり結果的には

[a](#) は `0x1234567B` になり、

[b](#) は `0x12345679` になります。

ここできちんと認識しておいて欲しいのは、ポインタが指し示しているものはアドレスであるので、アドレス自体は32bitの場合は4バイトの `unsigned int` と等価な型になっているわけよ。

とにかく数値で表されているわけ。

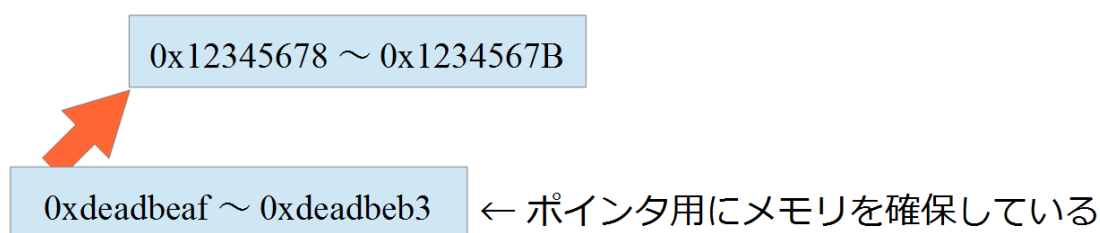
そうすると、当たり前の話なんだけどポインタを宣言した時点でまたメモリ上に4バイトは確保されているってことなのね。

だから例えば

```
int a;  
int* b=&a;
```

なんてやって

int 型の変数 a が 0x12345678 だったとして、ポインタ b が a のアドレスを指し示しているとすると、例えば、b 自体のアドレスは 0xdeadbeaf であり、そこから4バイト確保されることになる。



そう、ポインタであっても変数である以上はメモリをほんの少しだけ食いつぶしている意識は持って下さい。

例えば、アドレスがこういうものであるとする。

表みたいなもんやね

null ptr									
					a=5				
							c=0x3F		
			b=0x0F						

この例では 0x12345678 とかって書くとヤヤコシイのでメモリ上に 0~100 までアドレスが並んでいるとする。

メモリは一次元なので、一行目は 0~9 番目、二行目は 10~19 番目が並んでいて後は同じようにって感じだと思ってくれ。

さて、まず例えば

```
int a=5;
```

とかって書いたとする。それが実行されるとアドレス上の何処かに5が書き込まれるわけや。

そのアドレス値は15であることがわかる。アドレスであることを分かりやすくするために16進数で書いとくと0x0Fね。

ここでこの0x0Fというアドレス値を何かの変数に保存したとする。

```
int* b=&a;
```

そうするとこれもどこかのアドレスに確保されているわけや。つまりbにもアドレスがあると…。今回の例だと63にある。16進にすると0x3F。

これも変数に入れるならどっかに入っていることになる。

```
int** c=&b;
```

ポインタのポインタの話だが、ポインタのポインタを扱う時の主役はアドレス値なのね。今回で言うとbのこと。

DirectXのCreate系の関数とかでよくあるんだけど、あの最後の引数にポインタのポインタを渡すってのはポインタ型をbとすると

```
Create なんとか(1,2,3,&b);
```

ってなるわけ。この例であれば最後の部分にbのアドレス(この例では0x0F)が入っている。で、例えばCreate なんとかの場合はb=nullptrだったりするわけ。

うん、でそのアドレスを渡すということは、Create なんとか関数内で適当にメモリを確保して作られたオブジェクトのアドレスを変数bに振り込めるってことなんだ。

わかりにくいかな？

b=nullptrにしていると仮定する。

で、下のような関数がある。最後の引数がポイントなのね。外側からポインタ型の変数のアドレスが渡される…。関数内でこのアドレスの指す内容を書き換える。Createの場合であれば大抵は内部でメモリが確保され、確保したアドレスを以下のように値として渡してやる。

```
void Create なんとか(int a, int b, int c, int** d){  
    *d = new int;  
}
```

new は渡された型に応じて適当にメモリを確保し、そのアドレスを返すものだ。そのアドレスがポインタのポインタで渡された d に入っている。いや、*d なので、アドレス d が指し示す先(ここもアドレス)に、新しく確保したメモリアドレスを渡しているのだ。

結果として

```
int* q = nullptr;  
Create なんとか(1,2,3,&q);
```

とやれば、本来 nullptr のアドレスに、きちんと確保されたアドレスが入って来るってことです。

とりあえず『ポインタ』のおさらいについてはこんなものでいいでしょう。で、今のうちに言っておくと、実はポインタは極力使いません。更に言うと、『裸のポインタ』はほぼ使いません。

そこについてはぼちぼち話をしていきます…。

さて、C++らしいところの話をしていきましょうか

概要

まず、最初に断っておくと C++ は『オブジェクト指向言語』です。

そんなん分かっとなるわい!!! という人もいるでしょうが、そもそもオブジェクト指向を正しく理解していない人がプロでも沢山いますし、更に言うと C++ を誤解している人はそれに輪をかけて多いんですよ…。

※でも多分、今後は『オブジェクト指向をマスター』だけではプログラマとして生きていけない状況に、これからはなっていくでしょうね。(関数型プログラミングとか…この先どうなってしまうのか本当に分からない)

それはさておき、オブジェクト指向言語の基本って知ってますか？

そう…

- カプセル化
- 継承
- **ポリモーフィズム**

です。これが一応、オブジェクト指向の三大要素と呼ばれてるやつです。これももう古い…いい意味でも悪い意味でも枯れた概念ですけどね。

この中で一番…最も重要(だと僕が思ってる)なのは「**ポリモーフィズム**」です。オブジェクト指向の思想の中で一番わかりにくく、挫折者も多い。

だけど最も役に立つ。でもオブジェクト指向言語を使った時に最も大変なのがこの**ポリモーフィズム**です。まあこのポリモーフィズムに限らず、プログラミング言語の「難しい概念」を理解した時に、君たちのレベルはグリーンとレベルアップすることだろう。

ちなみにオブジェクト指向以外の、プログラミングの「難しい概念」としては

- 関数型プログラミング
- イベントドリブン
- RX(Reactive Extensions)
- マルチスレッド

などがあります。これもそのうち触れるかもしれませんが、ここでは触れません(難しいので)

ここではオブジェクト指向の3大要素について、ひとつずつ説明していきます。

まずね、C++には構造体みたいな概念として「クラス」ってのがあるんだ。

そして、実はこの「クラス」は「構造体」と殆ど同じなのです。だから別にビビる事はないんですよ？

違いは、デフォルトが「public」か「private」かってだけの話だよ。まあ、public やら private やらの話はあとで解説するんで待っててください。

クラスの定義は

```
class クラス名{  
    private:  
        メンバ変数の定義;  
        メンバ関数の定義();  
}
```



```
public:
    メンバ変数の定義;
    メンバ関数の定義();
};
```

こんな感じです。C++の特徴としては、メンバ変数だけでなく、上のようにメンバ関数も置くことができることです。

カプセル化の意義

カプセル化は**情報隠蔽**:つまりクラスにおいて private と public を使い分け、クライアント側に見せる情報(変数)や使わせる操作(関数)と、見せない情報や使わせたくない操作(関数)を区別します。

まあ簡単に言うと『C言語の構造体と違って、クラスの中のメンバ変数やメンバ関数は private か public の属性を持つぜ(protected もあるけどな)』ってこと。

効果としては『private』で設定すると、そのクラス自身以外からその変数関数に対するアクセスが禁止されます。このように**値の変化を一部に閉じ込めておけば**、何かしら不具合が起きた際に、**原因の特定が簡単**になります。

呼び出し側も必要な関数にのみ目が行くので、仕事の効率も上昇します。

これにより、特定のクラスを使用する時にクライアント側は何を見るべきか何を呼び出すべきかが限定され、注目すべき箇所を絞って作業できる。これがカプセル化です。今しばらくは有り難みがわからないと思いますので、機械的に

- private: 外から参照する必要が無い/外から呼び出す必要が無い
- public: 外から参照する必要がある/外から呼び出す必要がある

というわけです。

もうね、分かんなくてぶっちゃけめんどくさい時は全部 public でもいいです。でもそれだと構造体と同じだし、上司や先輩とかには絶対に怒られるけどね。『分からない』という理由で思考停止するよりはマシってことです。

思考停止ダメ絶対

ちなみにC++では private(非公開)、public(公開)に加えて protected(自分の子供にだけ公開)つてのがあります。子供って意味がわからないと思いますが、それは継承の話で出てきます。

継承

次に継承です。この概念は細かい点では誤解されることが多いものの、大きな意味ではご理解戴けることが多いですね。要はクラスというのは継承により概念の親子関係が構築でき、親の持ち物は子も使えるという仕様です。



言葉で言ってもナンノコッチャ分かんないと思いますんでコードを書いて説明します。

```
class Parent{//親クラス
    private:
        int hikokai;//非公開
        void HikokaiFunc();//非公開関数
    public:
        int kokai;//公開
        void KokaiFunc();//公開関数
    protected:
        int komieru;//子供にのみ公開
        void KomieruFunc();//子供にのみ公開関数
};
```

C++における継承は作り方的には

```
class 子クラス : public 親クラス{
    ~中略~
};
```

のように、継承先を左に、継承元を右に、間にコロンを打って
継承元に public をつけて作ります。

```

class Child : public Parent{//親クラス(Parent)から継承した子(Child)クラス
private:
    void Func1(){
        //特に何もしません
    }
public:
    void Func2(){
        hikokai=10;//ダメ！！非公開です
        kokai=12;//公開なので OK
        komieru=14;//protected なので子からは見えるので OK
        HikokaiFunc();//非公開やからダメゅーてるやる！！
        KokaiFunc();//公開だから呼び出せます
        KomieruFunc();//protected なので子からは呼び出せます。
    }
};

```

```

Child c;//子クラスの実体を作ります(ここは struct と同じと考えていい)
c.KokaiFunc();//○親クラスの関数ですけど、子クラスの関数でもあるので呼べます
c.HikokaiFunc();//×当然呼べません
c.KomieruFunc();//×Protected は子の内部からしか呼べません
c.kokai=30;//○OK です
c.hikokai=40;//×ダメよね
c.komieru=50;//×protected なのでダメです

```

ちょっとカプセル化の説明にもなってしまってますけど、まあ何となく分かるよね？継承すると親クラスの public,protected メンバ変数やらメンバ関数やらに関して、子は実装しなくても実装したことになるわけです。

イメージとしては家があってな？

ドアチャイムは public なわけや。なんでかつちゅーと外部の人が誰でも押せるわけやからな？友達も親戚も親も子供も、どっかの勧誘員も隣の人も押せるやる？あれが public のイメージなわけ。

で、継承のイメージは例えばな？家の冷蔵庫とかテレビあるやん？冷蔵庫とかテレビは親が買ったものなので所有権は親なんやけど子供も使えるやん？そういう意味で親が買ったものでも、ある意味子供が使えるって意味が継承のイメージや。でもよそのおじさんが扱っちゃ

いけんわけやん？

そういうイメージで言うと冷蔵庫とかテレビのイメージは protected なわけ。

最後の private は、親のものでも子は扱うことができない。これは親の財布とかそういうイメージや。おとんやおかんの財布は使っちゃアカンでしょ？…まあ家によってちやうのかもしれないけど、普通使っちゃダメでしょ？そういうイメージやね。

ポリモーフィズム(多態性)

最後のポリモーフィズムなんだけど、これホンマにややこしいから、ここで理解できなくても凹まんとしてや？ゆっくり理解したらいい。でも、最終的には理解してる前提でこちらも接するんでよろしくね。

プロでもポリモーフィズム理解できとるエンジニアはそうそういないのが現実なんですよ…残念やけどな。

だから、今はとりあえずは…大雑把なイメージを掴んで、そういうもんやと思っとして？

ポリモーフィズムは「謎の生物」や。

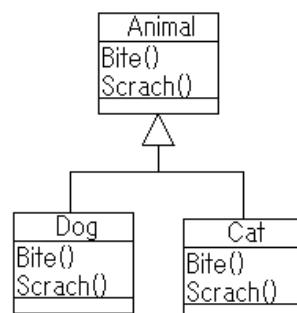
もうちょっといい方を変えると「親のフリしてる子」

である。う〜ん。変な比喻を使うと余計にわからんな…。とりあえず言うておくと継承をして親子関係ができている状態でないとポリモーフィズムは成り立たない。

継承先のクラスからできたオブジェクトは継承元のオブジェクトのように扱うことができます。

例えば

Animal から Cat と Dog が継承されていたとします。



Animal には Bite(噛みつき)とか Scrach(ひっかき)などがあって、それで攻撃をするとしておきます。それぞれ Cat も Dog も同名のメソッドを持ってるとします。継承して再利用するばかりがオブジェクト指向じゃないのよ?(ここよく誤解されてるんだよね)

そうすると

```
void Attack(Animal& animal){  
    animal.Bite();  
    animal.Scrach();  
}
```

なんていう関数を作ってやると

```
Cat cat;  
Dog dog;  
Attack(cat); //ネコの攻撃(噛みつき&ひっかき)  
Attack(dog); //イヌの攻撃(噛みつき&ひっかき)
```

のようにひとつの Attack 関数を Dog だろうが Cat だろうが 区別なく使えるわけです。これがポリモーフィズムです。

ちょっとわかりづらいですが『ポリモーフィズム』がないと、Dog 用の Attack や Cat 用の Attack 関数をいちいち作らなければならぬって事になります。

まあ今はよくわからないかもしれないけど、何度か使ったら分かるようになると思います。

とりあえず C++ の文法以前の『オブジェクト指向の三要素』を説明してみましたが無理なんでしょうか？

さて、では今から本格的に C++ の文法の話をするか…眠くなるかもしれないけどしゃーないなあ…

クラス(構造体)内関数

最初に C++ の特徴的な話をしよう。

C++ では構造体やクラスの中に『関数』を定義できます。C 言語の時は構造体の中に変数しか

定義できなかったよね？

C++ではそれに加えて関数を定義することができます。このため

```
struct A{  
    void Func(){  
        printf( "Func" );  
    }  
};
```

などと定義して

```
A a;  
a.Func();
```

などとして呼び出すことができます。

ちなみによく聞かれるのが『クラス』と『構造体』の違いについてなんですけど、C++ではただ単に『デフォルトアクセス制限』の違いだけです。

struct はデフォルトが public(公開)です。

class はデフォルトが private(非公開)です。

とりあえずこれだけの違いです。

※これが C#とか java とかだと構造体とクラスは意味合いが変わってくるので注意。C++はこういう所アバウトなのよね。

関数内クラス構造体

これは C 言語でもできてたとは思いますが、知らないといふので一応書いておきます。

関数内でクラス(または構造体)を宣言して、それを使用することができます。

```
void Function(){  
    class C{  
        private:  
            int x,y,z;  
        public:  
            C():x(0),y(0),z(0){  
  
            }  
            void Output(){  
                cout << "x=" << x << endl  
            }  
    }  
}
```

```
};
C c;
c.Output();
}
```

入れ子構造体(クラス)

```
class B {
public:
    class C{//クラス内クラス
    public:
        C() {}
        void Output() {
            cout << "TT" << endl;
        }
    };
    B() {
        C c;
        c.Output();
    }
};
```

ご覧のように入れ子にするのも OK です。

自己参照ポインタ

```
class D {
private:
    D* jibun;//自分と同じ型の『値』は宣言できないがポインタならOK
    D jibun2;//エラーおきます
public:
};
```

↑の例では、ポインタは OK だけど、値としては扱えません。なぜだか分かりますね？値として使用するには『型が確定』する必要があるため jibun2 は不正な扱いとなるのです。逆にポインタ扱いの場合は『型が確定』してなくても、『そういう型があるよ』という事が分かればポインタ扱いは OK なわけです。

つまり、まだ不完全な状態でもポインタとしては扱える…という事です。

前方宣言(プロトタイプ宣言)

前の話とちよつとかかわる話なんですけど、C/C++においては『前方宣言(プロトタイプ宣言)』というのが開発において割と重要になってきます。

前方宣言を知らなくてもプログラムは書けるんですが、効率的にシンプルに C/C++プログラムをしたいなら、覚えておくべきものです。

後々コーディング規約とか、クソコードとか、コーディングテクニックについて教えるときに詳しく話しますが、C/C++言語には『ヘッダファイル』というのがございます。そしてこのヘッダファイルにおける鉄則が二つあります



の二つです。それぞれまたおいおい話していきますが、非常にクソコードの温床となりやすい要素を持っています。で、ヘッダ内インクルードを使わないって言っても、そんなわけには行かんじゃない?

例えば、クラスのメンバにとある型を持っているとして、例えば `class Nanika` だったとする。これをメンバとして持つ `class Aruka` があるとします。

そうなる通常であれば `Aruka.h` 内のコードは

```
#include "Nanika.h"
class Aruka{
    private:
        Nanika nanika;
};
```

と書くことになると思いますが、こういう場合にも `Nanika.h` をインクルードするのは望ましくありません。悪くはないんですけどね。あまり良くないです。理由はインクルードというのが結局 H の内容をそのまま `#include` 部分に置換するため、`#include` 入れ子になってしまうとコードサイズが肥大すると `#include` 入れまくると、ファイル依存関係が物凄く煩雑になっち

やって、管理できなくなってきました。

…うん、まあ一度痛い目に遭ってみるといいよ。

ともかくヘッダファイルから#include するのは極力やめておきましょう。とはいえ上記のような構造を#include なしに実現しようとした場合はどうしたらいいのでしょうか？

まずは↑のクラスのメンバの Nanika nanika; をポインタ扱いにします。

```
class Aruka{
    private:
        Nanika* nanika;
};
```

ただし、これでも『Nanika の型が分かりません』と怒られますのでそこで『前方宣言』を行います。

```
class Nanika;
class Aruka{
    private:
        Nanika* nanika;
};
```

コンパイルは通りますが、このままでは『使え』ません。単なるポインタで、中身がないからです。

なので cpp 側に回って、実体化します。値として扱う時と同様にしたいのであれば

```
Aruka::Aruka(){
    nanika = new Nanika();
}
Aruka::~~Aruka(){
    delete nanika;
}
```

といった具合になります。

あとその他にも前方宣言は色々な使い道がありますが、今の所はこの辺で。次に行きましょう。

参照

C++には「参照」ってのがありましたね？

「参照」というのはポインタのようでポインタでない、でもちよっただけポインタばい、でも見た目は普通の変数…

という一風変わったものであります。

まず使い方を言っておきます。

参照の宣言は…

型& 変数=参照先;

とします。型の横に「アンパサンド&」をつけます。これでこの変数は「参照型」となり、ポインタのように最初に割り当てたオブジェクトを指し示すようになります。

ただし、ポインタと違って2つのルールがあります

- 必ず参照先を指定してやる必要がある
- 一度参照先を指定すると参照先の変更はできない

このルールにはメリットデメリットがあります。

メリット

「参照はさし示す先が存在することが保証されている」

「最初に設定した参照先であることが保証されている」

これは結構重要です。

ポインタなどの場合は、自由にアドレスを指定できるため、指し示す先が存在しないなど、結構困ったことを引き起こします。

デメリットはこれの裏返しですね

「融通がきかない」

です。

融通がきかないけど、関数の引数として使う文には非常に役に立ちます。

とにかく使ってみましょう。

```
int b=90;  
int& a=b;
```

とでも書いてやってあげると、a は b そのものを指します。ポインタと同様 a 自体には中身がなく、ただただ b の中身を見ているだけです。ですから上のコードを書いた後で

```
a=50;
```

とやって書くと b の中身が 50 に変更されます。

例えば、

```
void CheckHitEnemy(Player& player,Enemy& enemy){  
    if(IsHit(player.Rect(),enemy.Rect())){  
        player.Damage();  
    }  
}
```

など書くと、ポインタを使わずに、この関数の中で player や enemy そのものの値を変化させることができます→つまり呼び出し元の変数が変更されるってことです。

今まで何度も関数内で値を変更したいがために『ポインタのポインタ』を使うことがありましたが、『参照』を知っているとそこはちったあマシに書けるようになるということです。

例えば

```
void func(int* p){  
    *p=10;  
}  
int q=20;  
func(&q);  
cout << q << end; //q は 10 を出力します
```

こんなことにいちいちポインタを使いたくない。そういう場合はそっと*を&にします。そう

すると

```
void func(int& p){  
    p=10;  
}  
int q=20;  
func(q);  
cout << q << end; //qは10を出力します
```

参照を使うと分かりやすい。分かりやすいねえええ!!!

namespace と using namespace

namespace は日本語では『名前空間』という。直訳やね。

namespace に関しては C# の namespace とほぼおなじ意味。だけど C++ の場合は『とにかくまとめる』『区別する』ってくらいの意味。

例えば自分が `int Max(int a, int b)` などという関数を作っていたとする。

で、こんなもんクラスのメンバにしたいくもないし(いちいちオブジェクト生成したくないし、static メンバにするのも嫌)という場合があるわけだ。

そうしたら自然と、クラスのメンバにせずに関数単体で宣言するだろう？

その場合に問題が発生することがある。

何故かと言うと、他のライブラリとか使った場合に『名前がかぶる』ことがある。モチロンこれはエラーになる。

かといって `OreMax` なんてするのもバカバカしい。さてどうしようかといった時に、namespace を使用する。

namespace の使い方はいたって簡単。namespace と書いて、namespace 名を決めて、あとは囲むだけである。

```
namespace 名前{  
    ここに他のライブラリとかと特別したい関数を宣言しとく  
}
```

これだけ。ねっ？簡単でしょう？

さて、これと同じような事を `DxLib` や `Effekseer` や `STL` やらがやっている。

```
namespace DxLib  
{  
    //中略  
}
```

や

```
namespace EffekseerRendererDx11
```

```
{
```

```
//中略
```

```
}
```

という風に、だ。

じゃあ例えば、Ore なんていう namespace 作って、その中で Max という関数を定義し、それを使いたいとする。

どうするのかというと

```
namespace Ore{
    int Max(int a,int b){
        return a>b?a:b;
    }
}
```

などという風に定義する。この関数を外部から使うには4つの方法がある。

- その①自分自身が Ore の namespace の一員となる
- その②スコープ解決演算子を使う
- その③using を使う
- その④using namespace を使う

のどちらかの方法でアクセスできるようになる。更に言うと『自分自身が Ore の namespace の一員となる』には4つの方法がある。

まずは Ore の一員になってみましょう。

```
#include<iostream>
```

```
namespace Ore{
    int Max(int a, int b){
        return a>b?a:b;
    }
} //end of namespace Ore
```

```
namespace Ore{
```

```
int main(){
```

```
    std::cout << Max(2,3) << std::endl;
```

```

        getch();
        return 0;

    }
} //End Of namespace Ore

```

一応…コンパイルは通ります。通るのですがたぶんこの例だとリンクで失敗します。

为什么呢？

main 関数の鉄則として

「main 関数は必ず一つだけ存在しなければならない」

ってのがあります。

ところがこの例の場合は main 関数が Ore 名前空間内に入っているため、コンパイル後の名前

は

Ore::main()

ってなってるわけで、main 関数が存在しないといってリンクエラーを吐きます。

次に、スコープ解決演算子の話ですが、これは C++ でよく見かける :: のことです。

詳しくは口述しますが、C++ では名前空間やクラス名の入れ子によって、とある関数や変数が

直接指定できないことがあります、

その場合に、入れ子をたどっていくための演算子が :: なのです。慣れるまではちょっと難しいか

もしれませんが…。

ともかく、先ほどの例の場合だと Ore 名前空間内の Max 関数なのだから

Ore::Max(1,2);

などのように記述すればいいわけです。

ねっ、簡単でしょ？

おそらくこの使い方がメインにはなってくることでしょう。

次に using を使います。

これは、特定の関数や変数の名前を呼んだ時には、とある名前空間のやつを使うという約束事を宣言するものです。

```

int main(){
    Using Ore::Max;
    Max(1,2);
    Max(3,4);
}

```

とか記述しておく、main 関数内で Max を使った時には Ore::Max を使っているよと暗黙に宣言しておくわけです。

いくつも使わなければならない時には有効かもしれません。

しかしこれは後述するようにちょっと注意が必要です。

ちょっと先に、using namespace について説明しておきましょう。

using namespace 名前空間名;

と書く事により、これを書いたのと同じスコープにおいては、無条件で名前空間以下の変数関数を使用されるようになります。つまり

```
namespace Ore{
    int Max(int a, int b){
        return a>b?a:b;
    }
    int Min(int a, int b){
        return a<b?a:b;
    }
}

//end of namespace Ore
using namespace Ore;
int main(){
    std::cout << Max(1,2) << std::endl;//2 が出力
    std::cout << Min(3,4) << std::endl;//3 が出力
}
```

てな風に、特にスコープ解決演算子を使わなくとも Min や Max を使えるようになります。

ところが、この using のメリットはそのままデメリットになって跳ね返ってきます。

つまりいったん using や using namespace を使ってしまうと、それ以降のスコープで Max とか Min を使用すると問答無用で Ore のやつが使われることになります。

これが別の厄介なことを引き起こします。

using namespace 等が使われていることがわからないまま、それ以降の関数呼び出しを行っているとしても自分でも気づかずに別の挙動を書いている事があります。

この『自分でも気づかずに』って所がデメリットポイントで…まあずっとプログラマやってると身にしみると思いますが、自分の思ってるのと違う挙動が混ざっているととんでもなくバグが見つけにくいってことになります。

以上の理由から、基本的には using や using namespace を使用するのはいけません。
特にゼツタイにやめて欲しいのが

ヘッダ側で using を使用

するのは…

これはダメ!!ゼツタイ!!!

まあ、基本的には「::」を使用しとけてことです。あえてここまで書いたのは「C++の教科書」と呼ばれるものでは頻繁に using や using namespace が最初のサンプルで使われているからです。

気をつけてないと「実務で嫌われるプログラミング」を覚えてしまいますので、敢えて書きました。実務で嫌われるプログラミングはもちろん…就職活動の時にも嫌われるので気をつけてください。

まあちなみにこういう「構造をぶっ壊す記述」ができちゃうのが C++ の嫌われる理由でもあるんですよ。

C++における::の役割

前述した「名前空間」を解決するために出てきましたが、この::っていう演算子は「スコープ解決演算子」と言って、入れ子になっている構造をドゥンドゥン降りていくために使用するものです。

「名前空間」だけでなく「クラス(構造体)」や「入れ子クラス(構造体)」に使用することができます。

つまり

```
namespace Ore{  
    class A{  
        class B{  
            void func(){}  
        }  
    };  
}
```


の func にアクセスしたいならば

```
Ore::A::B::func();
```

のように使用します。

しばらくはお目にかからないとは思いますが、クラスを多用し始めるとバンバン使うことになります。

クラスの作り方

クラスの作り方は構造体と同じです。C++ではホント構造体と同じです。

違いは…

構造体はデフォルトでぜんぶ public(公開)で

クラスはデフォルトでぜんぶ private(非公開)

ってことだけです。

書き方は、こう。

```
class クラス名{  
    関数とか変数とか書く  
};
```

最後の;を忘れないように!!!

んで、ここで『アクセス指定子』ってのが大事なのだ。

『オブジェクト指向』の大事な要素の一つに『カプセル化』ってのがある。

これは『外に見せるものは見せて、見せるべきでないもんは見せない』っていう事を意味しているんだわ。

private と public

C#や Java でもあるであろう private と public やな。C++の場合は protected なんてのもあるけどな!!!

とりあえず軽く説明しとくと

private で宣言された変数や関数はクラスの中だけで使用でき、public で宣言された変数や関数はクラスの外からも参照できる。

ついでに言うと protected ってのは、宣言されたクラス本人とその子供だけが参照でき、外側からは参照できないという中途半端なヤツや。

C#とは宣言の仕方がちょっとだけ違う。

C#では

```
public void func();
```

こんな使い方だったと思うが C++では

```

class A{
    public:
        int valp;//公開変数
        void funcA();//公開関数
        void funcB();//公開関数
    private:
        int valq;//非公開変数
        void funcC();//非公開関数
        void funcD();//非公開関数
};

```

のように書きます。classAの中でpublic:を書いた行以降はすべて公開となり、private:を書いた行以降はすべて非公開となります。

そしてこの『クラス』も構造体同様に『型』の一つなので、フツーに型として使えます。
ですから

```
A a;
```

はい、これでもう実体ができました。
できましたので

```
a.funcB();
```

これでオッケーなんです。

えっ？aをnewしていないけど、いいの？って思うのはC#ユーザーですね。C++の場合は構造体とクラスの違いはデフォルトがprivateかpublicかって違いしかないと言ったはずですよ。

つまり通常の変数として使用する分にはnewは必要ないわけです。

逆にnewを使用することもできます。できますがその場合は左辺値はポインタ型変数である必要があります。

```
A a;//これはオッケー
```

```
A* b=new A();//これもオッケー
```

```
A c=new A();//ダメ
```

```
A* d;//通るけど、dの中身はありません。
```

ま、そんな感じ。

クラスメソッド

クラスの持っている関数のことを『クラスメソッド』と言います。C 言語の場合は構造体は変数だけの中に持てるものの、関数の定義はできません。

オブジェクト指向ってのは『振る舞い』を大事にするので、クラスが関数を持つのは必須なのです。そう C++ ならね。

というわけで、クラスのメンバとして関数を定義することができます。

まあ、フツーに定義すればいいので、

```
class B{
    public:
        void funcB();//公開関数
    private:
        void funcD();//非公開関数
};
```

これで宣言は OK です。ところが実装はどう記述すればいいんでしょう？

2つやり方があります。

一つはクラス宣言時に実装まで書きちゃう事です(やれちゃうけどお勧めしない)。

```
class B{
    public:
        void funcP(){
            funcQ();
            std::cout << "funcP" << std::endl;
        }
    private:
        void funcQ(){
            std::cout << "funcQ" << std::endl;
        }
};
```

この書き方も間違いでは…ないです。

とはいえあまり一般的な書き方ではありません。C++では『宣言はヘッダー側で』『実装は cpp

側で」っていうセオリーがあります。
で、この実装を書くのがちょっとヤヤコシイですよ。

例えば B.h というヘッダーに

```
#pragma once
class B{
    public:
        void funcP();//公開関数
    private:
        void funcQ();//非公開関数
};
```

と書いて、ここでは実装を書きません。そして B.cpp とかの中で
#include "B.h"

```
void B::funcP(){
    funcQ();
    std::cout << "funcP" << std::endl;
}
void B::funcQ(){
    std::cout << "funcQ" << std::endl;
}
```

と書けば funcP と funcQ の実体を書いたことになります。

なお、B::を書かないとどういう扱いになるのかというと、それは B のものではなく、グローバルなものって扱いになるので、B のメンバ関数(クラスメソッド)という扱いにはなりませんので注意が必要です。

とりあえず実装側の文法は

```
戻り値 クラス名::関数名(パラメータ){
    //実装
}
```

とでも覚えておいてもらったら良いと思います。

これでクラスのメソッドの実装はできるようになりましたね？

メンバ変数(またの名をフィールド)

これはもうすでにさらっと書きちゃってますけど、構造体におけるメンバと同じような要領で定義すればいいです。

構造体との違いはデフォルトが『外部公開(public)』か『外部非公開(private)』かって違いだけです。

ここらでそろそろプログラミング脳になってきてほしいんですけど、メンバ変数ってのはそのクラスの『持ち物』って意識をもってください。

```
class A{
    public:
        int value;//この value はクラス A の『持ち物』
};
```

あとは構造体のメンバと同じように使えます。

A a;

と宣言して

a.value

としてアクセスするか

A* b=new A();

b->value

としてアクセスします。

もちろん,private だと、外側からはアクセスできません。

メンバ関数(またの名をメソッド)

C と違って,C++ではクラスや構造体は関数も『メンバ』として持つことができます。この意味ではメンバ変数とメンバ関数の認識は同じでお願いします。

とにかくクラスや構造体の『持ち物』です。

フツーに関数を呼び出すように使えるんですけど、単品での呼び出しはできません。クラス自身は持ち主ですので、見た目上単品での呼び出しのように見えますが…。

```
class A{
    public:
        void Func();//この Func 関数はクラス A の『持ち物』
};
```

として

```
A a;  
a.Func();
```

もしくは

```
A* b= new A();  
b->Func();
```

として呼び出せます。しつこいですが『持ち物』って認識を忘れないようにしてください。忘れるとおかしなバグを引き起こしますよ。

メンバ関数ポインタ

はい、フツーにクラスのメンバ関数を作れるようになったところで、メンバ関数ポインタの話をしてします。

既に『関数ポインタ』についてはお勉強していると思いますので、特にそっちは説明しませんが、C++にはメンバ関数ってのがありましたね？

アレのポインタって取れるんでしょうか？

通常に関数ポインタはこうでしたよね？

例えば

```
int funcA(int p,int q);  
int funcB(int p,int q);
```

なんていう関数があったとすれば、これを代入できるポインタ宣言は

```
int (*pFunc)(int,int);
```

こういう宣言ができて、

```
(*pFunc)(5,7);
```

のような記述で関数呼び出しができましたね？

この関数ポインタには、実際の関数の名前を右辺値に置くことにより

```
pFunc=funcA;
```

と代入でき、

```
pFunc=funcB;
```

などのように関数を切り替えることができるため、状態遷移(ステートマシン)の実装が容易になるというものでしたね？

さて、では、クラスのメンバ関数のポインタもこのようにできるのでしょうか？

答えはNOです。

例えば

```
class A{
private:
    int Add(int p,int q){
        return p+q;
    }
    int Subtract(int p,int q){
        return p-q;
    }
    int (*pFunc)(int,int);
public:
    A(){
        pFunc=Add;//代入できません
    }
    void Change(){
        pFunc=Subtract;//代入できません
    }
    void Print(int p,int q){
        std::cout << (*pFunc)(p,q) << endl;//アカン
    }
};
```

こういう書き方はオツケーなののでしょうか？

実はメンバ関数ポインタは、フツーの関数ポインタと同じようには扱えません。面倒ですね。

関数ポインタに関数の実体を代入しようとした時にエラーが発生します。

何故か？

とりあえずエラーメッセージを見てみよう

'A::Add': 関数呼び出しには引数リストがありません。メンバーへのポインターを作成するために '&A::Add' を使用してください

'int(__thiscall A::*)(int,int)' から 'int(__cdecl*)(int,int)' に変換できません。

エラーメッセージを見てもわけがわからないよ。

ちょっとわかりづらいですけど、C++言語は、メンバ関数ポインタ(静的関数を除く)と通常の関数ポインタを区別しているのです。

つまり

通常の関数ポインタなら

```
int (*変数名)(パラメータ);
```

という型で済むのだが、これがメンバ関数の場合はそうはいかない。以下のように解釈される

```
int (クラス::*変数名)(パラメータ);
```

なんでこんなややこしい解釈になるのかというと C++ 言語のメンバ関数がコールされるとき、隠し引数として、そのクラスの実体自身(this ポインタ)が渡されるからである。

ということで、クラスメンバは暗黙のうちに『誰の持ち物か』という情報がくっついた状態になっており、これでは『型が違う』と判断されてしまうからなのだ。

さて、これに対処するにはどのように書く必要があるのかというと、メンバ関数ポインタが『誰の持ち物の関数を指し示すのか』を明記してやればいい。

つまり先程の例であれば

```
int (*pFunc)(int,int);
```

ではなく

```
int (A::*pFunc)(int,int);
```

である。クラス名を関数ポインタの前に持ってきてやればいい。ちょっとキモチワルイ記述の仕方になるが、まあこういうもんだと思ってくれ。

さて、次にこの関数をコールする時だが、これもまた面倒である。

```
void Print(int p,int q){  
    std::cout << (*pFunc)(p,q) << std::endl; //タメ  
}
```

この書き方だと、関数呼び出しの部分でコンパイルエラーが発生します。

ホントにメンバ関数ポインタってのは特別扱いなんだなあ…。

何がアカンかというと、前にも書いたように、メンバ関数ってのは持ち主のポインタが隠しパラメータとして渡されているため、コールする時に、明示的に『誰の持ち物の関数ポインタを読んでいるのか』を伝えなければなりません。

メンドクサイ

この例だと

```
(this->*pFunc)(p,q);
```

なんていうダッセー書き方をしなければなりません。もう文法なので仕方ないのですが、やっ

ぱり C++ が嫌われるところってこういう所だよなーって思う。

ちなみに持ち主がポインタでない場合は

```
(a.*pFunc)(p,q);
```

みたいな書き方になる。

で、これで終わりかというところ、まだ問題があるようで、このままコンパイルを通そうとすると通らない。エラーメッセージを見るとこうである。

'A::Add': 関数呼び出しには引数リストがありません。メンバーへのポインターを作成するために '&A::Add' を使用してください

よく分かりません…なにこれ。

一応

<https://msdn.microsoft.com/ja-jp/library/b0x1aaaf.aspx>

を見てみたが、古いコンパイラだとこの書き方をしなくてもいいらしい的な事が読み取れる…つまり、安全のために入れた文法エラーということだろう。まあ、これ以上これについて考えるのも無益なので『そういうもん』だと思っておこう

結果として

```
#include<iostream>
```

```
class A{
```

```
private:
```

```
    int Add(int p,int q){
```

```
        return p+q;
```

```
    }
```

```
    int Subtract(int p,int q){
```

```
        return p-q;
```

```
    }
```

```
    int (A::*pFunc)(int,int);
```

```
public:
```

```
    A(){
```

```
        pFunc=Add;
```

```
    }
```

```
    void Change(){
```

```
        pFunc=Subtract;
```

```
    }
```

```

        void Print(int p,int q){
            std::cout << (this->*pFunc)(p,q) << std::endl;
        }
};

int main(){
    A a;
    a.Print(1,2);
    a.Change();
    a.Print(1,2);
    getchar();
    return 0;
}

```

こういうコードだとコンパイルが通るメンバ関数ポインタの使い方ってことになります。
さらっと書く予定だったけど、長くなりました。

static 変数、static 関数

static っていうのは静的変数、静的関数を作るものです。

静的って何なんでしょう？何をもって静的と言っているんでしょうか？

例えば関数の中で宣言する変数は、関数呼び出しとともに生成され、関数が終われば破棄されます。そういう意味でダイナミック(static の反対)というわけです。

static っていうのはそういう枠にとらわれないやつです。

一度宣言されれば一生(アプリが終了するまで)メモリ上に存在する変数です。

つまり

```

void func(){
    int a=0;
    a++;
    cout << a << endl;
}

```

なんて関数を作って、

```

for(int i=0;i<bb;++i){
    func();func();func();func();func();
}

```

等と書いても、出力される数値は毎回1ですね。これ分らない人は1年からやり直しましょう。真面目な話で。

毎回毎回 a という変数が 0 初期化され生成され、それに ++ されるからずっと 1 が出力されます。

ところがこのローカル変数を static にすると話は別です。

```
void func(){
    static int a=0;
    a++;
    cout << a << endl;
}
```

ずっとメモリ上に残り続けることは関数を呼ぶたびに、インクリメントが繰り返されます。やってみればわかりますので、やっというてください。

ある意味危険なので、C++ 言語における static 変数の使用は慎重に…どうしても必要でないかぎりはずり使わないほうが良いでしょう。

さて、本題はここではなくて、static 関数の方です。結構お目にかかるので、知っておいたほうが良いと思います。

例えばメンバ関数を呼び出す場合

A a;

という風に型 A のオブジェクトを作ったうえで

a.func();

のように呼び出す必要があります。つまり『実体』がないとメンバ関数ってのは原則的には呼び出すことができません。

ところが、static を頭につけると実体がなくてもメンバ関数を呼び出すことができます。

例えばクラス A が

```
class A{
    public:
    void func(){cout << "call func" <<< endl;}
};
```

ならば

A a;

a.func();

でした。これを単に関数呼び出しだけしたい場合はこの func を static にします。

```
class A{
    public:
    static void func(){cout << "call func" <<< endl;}
};
```

```
};
```

そうすればこの func 単体で呼び出すことができます。ただし A の持ち物ではあるので…そう、『スコープ解決演算子::』を使います。

```
A::func();
```

こんな感じです。

const

const は簡単ですね。定数を作るものです。

```
const int STATUS_DEAD=16;
```

こんな感じです。簡単すぎです。

これが const 定数の使い方です。よく使いますので、きっちり使いこなしましょう。

さて、これを引数で使用するとう当然のように引数の数値を変更することができなくなります。

```
func(const int p){  
    p++; //エラー!!扱えないよ  
}
```

融通がきかなくなります。

これが何の役に立つのか？融通がきかなくなるということは、引数の値が入力時と同一であることは保証されるわけです。

プログラムが複雑になってくればくるほどこの手の『あえて融通を利かなくする』っていうテクニックは重要になるので覚えておきましょう。

さて、こんなもん知ってると思うんで別に本題でもなんでもないです。

本題は const メンバ関数です。

作り方は

```
型 関数名(パラメータ) const;
```

です。

おしりに const をつけるって所が新しいですねえ〜。

さて、コイツは何が嬉しいのか？

それは『おしりに const をつけたメンバ関数は『オブジェクトの状態を変更できない』』という仕様があるからです。

オブジェクトの状態ってのは何なのかというと、その関数呼び出しによりメンバ変数の値が変化することはないってこと。

変化させようとするコンパイルエラーになります。

例えば

```
class A{
    public:
        int a;
        void func()const{
            int b=10;//OK
            b++;//OK
            a++;//NG!!メンバの変更はできません!!!
        }
};
```

なんでこんなもんを使うかというさっきも言った『あえて制限かけることで安全にしたい』という思想が C++ 言語…いや、たいていの言語思想にはあります…この辺に慣れてくると、よりプロっぽいプログラムができるようになるでしょう。

継承

継承も簡単やね。でも C# とか Java とちょっとだけ書き方が違うので注意。

基本的には…こう

```
class 派生先クラス名 : public 派生元クラス名{
};
```

これが基本的な『継承』の文法です。: コロンが継承の合言葉なわけです (Java だと extends)。

C# との違いは『**public**』とか書いてある部分です。面倒だなあ。C++ の場合、これを書いていないと private 継承として扱われます。

基本的に private 継承は親の関数とかが使えなくなるので、まれなケースだと思ってください。忘れちゃってもいいです。ですから継承の時は public

private 継承ってナンヤネン？

うん、まあ、それはな、親の全てのメンバが private 扱いになるってことや…。

親の public メンバには子からはアクセスできるんやけどな？

つまり

```
#include<iostream>
```

```
using namespace std;
```

```

class Base{
public:
    void Baka(){
        cout << "バカ!!" << endl;
    }
};

class Derived : Base{//public なし継承
public :
    void Aho(){
        Baka();
    }
};

```

```

int main(){
    Derived d;
    d.Aho();//アクセス OK!!
    d.Baka();//アクセス不可!!
}

```

というわけ。

親の Baka にアクセスさせたいのならば

```
class Derived : public Base{~
```

と、public を追加します。

なんとも面倒な仕様です。更に言うと所謂『ポリモーフィズム』が働かなくなります。

public をつけていれば後述する『ポリモーフィズム』が働き

```
Derived d;
```

```
d.Aho();
```

```
Base& b=d;
```

こういう書き方は OK はずなのですが、public が入っていない場合、これは許されません。仕様です。

public 継承していないと…

```
Base& b=d;//NG!!プライベートクラスはポリモーフィズム使えねーよwww
```

となります。とりあえず皆さんはC++で継承を使いたい場合には「必ず」public をつけとくことを忘れないようにしてください。

ポリモーフィズム

はい、コイツは

「親のフリできる子」

とても覚えておいてください。つまり継承先のクラスからできたオブジェクトは継承元のオブジェクトのように扱うことが出来るってことです。前述のとおり Derived クラスが Base クラスから継承されていれば Base のフリができるわけです。

例えば

Animal から Cat と Dog が継承されていたとします。

Animal には Bite(噛みつき)とか Scrach(ひっかき)などがあって、それで攻撃をするとしておきます。それぞれ Cat も Dog も同名のメソッドを持ってるとします。

そうすると

```
void Attack(Animal& animal){
    animal.Bite();
    animal.Scrach();
}
```

なんていう関数を作ってやると

```
Cat cat;
```

```
Dog dog;
```

```
Attack(cat); //ネコの攻撃(噛みつき&ひっかき)
```

```
Attack(dog); //イヌの攻撃(噛みつき&ひっかき)
```

のようにひとつの Attack 関数を Dog だろうが Cat だろうが区別なく使えるわけです。

これがポリモーフィズムです。

簡単でしょ？

new と delete と delete()と...

C 言語において動的メモリ確保は、基本的に malloc で確保して free で解放していたと思います。でもC++でのメモリ確保の基本は new です。そして解放の基本は delete です。

まだ今しばらくは「動的メモリ確保」を行う機会はないかもしれませんが malloc 使いたくなったら new でやれないか考えてください。

特定の型を確保するための new の文法は...

```
型名* 変数名=new 型名();
```

となります。

この解放は
`delete 変数名;`

`malloc~free` よりシンプルなんじゃないかなと思います。なお、C++においては `malloc~free` で確保~解放するとクラスのコンストラクタやデストラクタが呼び出されないのでご注意ください。

さて、C 言語では動的配列を作るために `malloc` を使う場面も多かったかと思います。

```
int* array=malloc(配列要素数*sizeof(int));
```

こんな感じで作っていたのではないのでしょうか？
これは C++においては

```
int* array=new int(配列要素数);
```

となります。このようにすれば配列を動的に確保できます。ただこれも解放の手順が必要で、解放するためには

```
delete[] array;
```

となります。配列の解放は `delete` ではなく、`delete[]` であることに注意してください。
僕のオススメは配列には後述する `vector` を使うことです。自動で解放してくれるし、再確保やサイズの変更が容易です。

コンストラクタ/デストラクタ

さて、C++のクラスの特徴として、大きいのは確かに『メンバ関数』があることですが、それよりも大きいのが、コンストラクタとデストラクタがあることです。

これをうまく利用すれば、様々なリソースの解放し忘れを防ぐことができます。
さて、いきなり『constructor』『destructor』なんていう英語が出てきて、わけわからんことになっているかもしれませんが、一言で言うと、

- コンストラクタはクラスや構造体の実体ができる時に自動で呼ばれる関数
- デストラクタはクラスや構造体の実体が死ぬ(消滅する)前に自動で呼ばれる関数

これだけです。細かく言うとどんどん深みにハマりますが、基本はこれだけ押さえていればいいです。

で、どうやってこの便利な関数を定義するのかというと、これも覚えてしまえば簡単。

- コンストラクタおよびデストラクタは戻り値を持たない
- コンストラクタは、その型名と同じ名前の関数
- デストラクタは、その型名の前に~(ふにゃチルダ、~)をつけた関数

ってわけです。

サンプルコード:

```
#include<iostream>
using namespace std;
class A{
public:
    A(){
        cout << "A" << endl;
    }
    ~A(){
        cout << "~A" << endl;
    }
};
```

```
int main(){
    A a;
    return 0;
}
```

こんな感じに書くと、main 内の1行目で A 型の変数である a が実体化され、コンストラクタ A() が実行されます。で、main を抜けるときに、a が解放されるためデストラクタ ~A() が呼ばれる仕組みです。

これを動的に確保して、解放したい場合は

```
int main(){
    A* a=new A();
    delete a;
    return 0;
}
```

のように書きます。

コンストラクト時引数

ちなみにコンストラクタには引数を持たせることができます。(デストラクタには引数を持たせられない)

例えば特定のクラスを生成する時に引数が必要だとすると以下のように定義します

```
class A{
    public:
        A(int val){
            //中略
        }
};
```

などと宣言しておいて

```
A a; //×引数を必要としているのに引数を指定していない
A a(15); //○
A* b=new A(36); //○
A* c=nullptr;
c=new A(36); //○
```

のようにメモリが確保されるタイミングで引数を渡してやれば良いわけです。

このコンストラクタとデストラクタがあるおかげで便利な使い方があるとすると、クラスの実体を作った時の処理が、それが解放(明示的に delete 解放 or スコープを外れた)された時に後始末される事が期待される場面…例えばファイルのオープン/クローズや目盛りの確保/解放などのペアになっている処理に応用するとかなりの力を発揮します。

例えば

```
class File{
    private:
        FILE* _fp;
    public:
        File(const char* filename){
            _fp=fopen(filename, "rb" );
        }
};
```

```

        }
        ~File(){
            fclose(_fp);
        }
        //中略
};

```

みたいな作りにおけばファイルクローズのし忘れとかを防げるわけですよ。便利ですよ？

なお、C++言語では動的確保は new と delete を使用してください。くれぐれも C++ でありながらクラスや構造体の確保に malloc と free を使わないでください!!

いや、もうね、使い慣れたのを使いたい気持ちはわかりますけどね。これは実害が起きるんですよ!!

どういう実害かという、malloc→free を使用してしまうと、コンストラクタ&デストラクタがコールされません。言語の機能を使うときは、きちんと調べて使いましょう。

初期化子

C++において「初期化子(初期化リスト)」というのは割と範囲が広く様々な意味があるが、ここではコンストラクタ時のメンバ変数の初期化、親クラスのコンストラクタ明示的呼び出しなどについて説明する。

自分のメンバ変数をコンストラクタ時に初期化したいとする。その場合、初期化子(初期化リスト)という文法を使用して初期化することができる。

```

class A{
private:
    int _member;//こういう変数があるとして...
public:
    A();
};

通常であれば _member を 0 など"で初期化したいとすると

A::A(){
    _member=0;
}

```

という風に初期化するだろう。

だが、初期化子を利用すると

```
A::A():_member(0){  
}
```

という風にかけることができる。そんなの何の意味があるの?と思うかもしれないが、いろいろな点で利点があり

[EffectiveC++](#)という本では『コンストラクタ内での代入より初期化子リストを使え』という格言がでてくる。

詳しくは立ち読みでも図書館でもいいので本を読んどいてね。

ここでは確実に『初期化子を使わざるを得ない』状況を考えます。



どういう場合かというと、const メンバ変数と参照メンバ変数です。

例えば、クラスのメンバとして const や参照があるとそのままではエラーを吐きます。

```
class A{  
    public:  
        const int _a;  
        int& _b;  
};
```

などという事を書くと『初期化されてねーよ www』ってエラーを吐きます。

そういう場合、例えば外側から値を渡してあげなきゃいけないときなんかはこう書きます。特に参照の場合に顕著ですけど

コンストラクト時に引数で初期化するなら

```
A::A(int a , int& b):_a(a),_b(b){}
```

とやります。なお、参照の場合は**b**の部分は引数も参照にしないと意味がないので注意してください。

参照をメンバ変数として持つ方法

実際のゲームプログラムをやっていくと誰かに誰かの情報を持たせたい時があります。そういう場合はどうすればいいのでしょうか？

Update 関数などで逐一参照を引数で渡していくのもまあ一つの手なのですが、正直煩わしいですね。わずらわしい、煩わしいですよホント。

煩わしくない人は Update 関数の引数にしてればいいです。

僕は煩わしいので、一回設定したらそれで終わりにしたい。例えば A というクラスと B というクラスがあって、A は B のことを知っておきたい。

そういう状況だとした場合、まずは A のメンバ変数として B への参照を持つとします。

あえてクラス B の定義は書きませんが…

```
class A{  
    private:  
        B& _b;//B への参照をもつと…  
};
```

当然ながら怒られます。原因は[参照の説明](#)を読んでいただきたいんですけどルールがありましたね？

- 必ず参照先を指定してやる必要がある
- 一度参照先を指定すると参照先の変更はできない

一つ目のルールに違反しているわけです。つまりクラスを実体化した時点でメンバ変数が初期化されてないため『参照』としては存在できないんですよ。

じゃあどうしましょうか？

ともかく『参照先』をどこからか調達してこなければならぬわけですが…。まあそこは『コンストラクト時引数』にするしかないですね

ちなみに、この時の引数も『参照渡し』しないとコピーになっちゃうと大元の変数への参照が切れちゃうため、不具合の原因となります。

さて、コンストラクト時に参照渡しするには

```
A::A(B& b){  
}
```

とするわけですが

この**b**を**_b**に代入すればいいんじゃないでしょうか？

```
A::A(B& b){  
    _b=b;//  
}
```

こう？でも残念ながら怒られます。

コンストラクタが呼ばれる前に**_b**に値が入っていないと**_b**の『初期化時』には当たらないわけです。

そこで代入ではなく『初期化子』を使います。

```
A::A(B& b):_b(b){  
}
```

のようにすればメンバ変数の**_b**が引数で渡された**b**そのものになるわけです。

仮想関数、純粋仮想関数

さて、こいつは継承とかポリモーフィズムと関連している話なのだが、Virtual 関数。つまり仮想関数の話をします。

例えばこんなコードを書いたとします。

```
#include<iostream>
```

```
using namespace std;
```

```
class Animal{
```

```
public:
```

```
    void Cry(){
```

```
        cout << "知るかよ" << endl;
```

```
    }
```

```
};
```

```
class Cat : public Animal{
```

```
    void Cry(){
```

```
        cout << "ニャーン(° 㐎 °)" << endl;
```

```
    }
```

```
};
```

```
class Dog : public Animal{
```

```
    void Cry(){
```

```
        cout << "わんわんお( ^ ω^ )" << endl;
```

```
    }
```

```
};
```

```
void CryAnimal(Animal& a){
```

```
    a.Cry();
```

```
}
```

```
int main(){
    Cat c;
    Dog d;
    CryAnimal(c);
    CryAnimal(d);
    getchar();
    return 0;

}
```

さて…結果はなんて出力されるんでしょう？

おそらく想定しているのは

ニャーン(° Ⅰ °)

わんわんお(^ω^)

と出力されるのを期待するでしょう。

だが結果は

知るかよ

知るかよ

である。現実是非情である。

C++の仕様として『ただ単に継承しただけでは『現在の型』が持っているメソッドが呼ばれる』
ってやつです。

つまり、CryAnimal 関数における『現在の型』は Animal であり、Dog や Cat ではないんだわ。

あくまでも Animal 型のオブジェクトなんやね。

それでは継承した意味が…という部分を解消するための機能がこちら

『仮想関数』

これは親の関数に virtual キーワードをつけることで、継承したオブジェクトを親として扱って
も、その際に呼び出される関数はそのオブジェクト自身の関数が呼び出されるってこと。

よくわかんねーと思うんで、専門的なことはともかく親の Cry 関数に virtual をつけてくだ
さい。

きちんとそれぞれの関数が呼ばれるようになったともいます。

次に C++には純粋仮想関数などという仕様がある。

フツーに考えて、Animal ってのは Animal 単体では使いません。必ず継承したものが使われま
す。その場合、Animal に関数を実装しても無意味です。なので、これを満たす仕様として

『この関数は継承先で実装するはずだからここでは定義しないでいいよ』って仕組みがある
このことを『純粋仮想関数』って言います。

文法は至って簡単。**=0**(イコールゼロ)をメソッドのおしりにつけるだけ。

```
virtual 戻り値 関数名(パラメータ)=0;
```

です。

こういうメソッドを持っているクラスを『純粋仮想クラス』と言い、純粋仮想クラスは単品でオブジェクト化することができなくなります。

どういう事かということ、メソッドの中身がないわけですから誰かが継承して、その関数の実装をすることが求められているということです。

DirectX でいうと

IDirectX~

とか

ID3DX~

とかのクラスがそれに当たります。こいつらは単品で存在できないので new することができません。ですから create 系関数が呼ばれることになります。create 系関数の中では別のクラスがあって、そいつが関数内で new されてそれを create の戻り値にされていると考えられます。

typedef

これは便利機能です。覚えなくてもゲーム作れますが、覚えておくと便利なので教えます。

文法は

```
typedef 元の型名 新しい型名;
```

ってやつです。

え〜？こんなもの何に使うのさあ…めんどくせーだけじゃん。

そうですね。例えば vector とか使用してやたらと長い型名とかになった時や、パット見型名がわかりづらい時とかに使用します。

例えば vector ならば

```
typedef std::vector<int> IntArray_t;
```

とかってやれば宣言時には

```
IntArray_t intarray;
```

などと宣言できるしイテレータ使う時も

```
IntArray_t::iterator it=intarray.begin();
```

なんて使い方ができるわけです。

typedef 使わないと

```
std::vector<int>::iterator it=intarray.begin();
```

なんてくっそ長い文字列を書かなければならなくなります。それを軽減するために使用します。

他には例えば関数ポインタとか使うとき、関数ポインタ型変数の宣言は

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

のように非常にわかりづらく、これが『変数(関数ポインタ型変数)』であることに気づきにくいです。初心者～中級者前半だとわからないです。

なのでこれを typedef 使ってもう少しわかりやすくします。

```
typedef int (*FunctionPointer_t)(int , int);
```

ここまで見ると『さっきと変わんねーじゃん』と思うでしょうが、さっきのカッコ内は『変数』であり、今回は『型』です。

つまり

```
FunctionPointer_t a;
```

```
FunctionPointer_t b;
```

とやれば a も b も同じ『関数ポインタ型の変数』として扱うことができます。

もうついでなのでメンバ関数ポインタもやっちゃいます。

例えば

```
class Test{
```

```
    public:
```

```
        void Func(Test::*pFunc);
```

```
};
```

これが通常のメンバ関数ポインタの宣言。

typedef を使用すると

```
class Test{
```

```
    public:
```

```
        typedef void (Test::*MemberFunctionPointer)();
```

```
};
```

となり

MemberFunctionPointer が『型名』の役割をはたすので

```
MemberFunctionPointer p;
```

```
MemberFunctionPointer q;
```

は両方共メンバ関数ポインタとなります。

関数オーバーロード

C++言語は非常に便利な言語です。何故かというと言数の数、言数の種類によって関数名を分けて済むからです。C言語であれば、同名関数を複数作ってしまうとアウトでした。

C言語の場合…

```
void Set(char c){
    //なんか処理
}
void Set(int i){//アウトお～！！
    //なんか処理
}
```

ところが C++言語の場合になると、言数の型や言数の数が別ならば、それは別関数とみなされます。

C++言語の場合

```
void Set(char c){
}
void Set(int i){//これはOK!!
}
```

これのおかげで、いちいち、C言語のときみたいに、abs 関数の float バージョンとして fabsなんて名前を考える必要はなくなるわけです。自分で組むときに役に立つのはこういうのでしょ

```
struct Vector3{
    float x,y,z;
    void Set(float inx, float iny, float inz){
        x=inx;
        y=iny;
        z=inz;
    }
    void Set(Vector3& vec){
        x=vec.x;
        y=vec.y;
        z=vec.z;
    }
}
```

```
};
```

みたいに書けちゃうわけ。どっちでも好みのやり方でどうぞと、呼び出し側に提示できるわけだ。さて、C++言語はさらに恐ろしいこともできちゃう。できちゃう。それは……

オペレータオーバーロード(演算子オーバーロード)



どういうことかというのを説明するより見てもらったほうが早いような気がしますね。一言で言うと、ある型に対する演算子を勝手に定義できちゃうってことです。

構造体 `Vector3` を例にとってみましょう。たとえば、

```
Vector3 a,b;
```

という風に、二つのベクトル `a`, `b` があったとして、数学におけるベクトルの足し算みたいなことをやりたいとします。

…どうするでしょうか？たとえば、足した結果をベクトル `c` に入れたいとする

```
Vector3 c;
```

```
c.x=a.x+b.x;
```

```
c.y=a.y+b.y;
```

```
c.z=a.z+b.z;
```

みたいなことをやらなければなりません。3D では結構このような演算を行うので、C 言語であれば通常は関数を作ってやったりします。

```

Vector3 Add(Vector3& a, Vector3& b){
    Vector3 ret;
    ret.x=a.x+b.x;
    ret.y=a.y+b.y;
    ret.z=a.z+b.z;
    return ret;
}

```

みたいになるでしょうね。でもベクトルの足し算なら、

```
c=a+b;
```

みたいにしたいですよ？そういう時に役に立つのが、演算子オーバーロードです。やりかたはいたって簡単。

```

戻り値 operator 演算子(引数リスト){
}

```

てな感じに定義します。具体的に、ベクトルの足し算ならば

```

Vector3 operator+(Vector3& a, Vector3& b){
    Vector3 ret;
    ret.x=a.x+b.x;
    ret.y=a.y+b.y;
    ret.z=a.z+b.z;
    return ret;
}

```

なんて感じに定義します。そうすると

```
c=a+b;
```

なんていう風に、ベクトルの足し算を、数学っぽく記述することができます。また、Vector3のメンバとしても記述できます。Vector3ならば

```

struct Vector3{
    ~中略~
    Vector3 operator+(const Vector3& vec){
        Vec3 ret;
        ret.x=x+vec.x;
        ret.y=y+vec.y;
        ret.z=z+vec.z;
        return ret;
    }
};

```

みたいになります。他にもいろいろな活用がありますが、ここではこのくらいに留めておいて、開発中に必要に応じて増やしていきましょう。

enum

enumですけど、これは知っておいたほうが良い。いやもう知ってると思いますが、活用してる人が少ないと思いますので、ぜひ使用してください。

使い方はいたって簡単

```

enum 型名{
    要素1,
    要素2,
    :
    :
};

```

という風にぞろぞろ一と並べていくものです。

で、Enumは内部的にはint型でできてて、それをenum型と言っているに過ぎません。どうやりやすいのかというと、例えば敵のデータで

スライム=1

ドクロ=2

コウモリ=3

てな感じで番号によって表す敵が違おうとします。これを1とか2とかの数値のまま運用すると、この番号を覚えておかなければいけないし、番号の変更があった場合の手間が半端ないわけです。

これがenumで扱うなら

```

enum EnemyType{//敵種別
    et_none,//種別なし(0)

```

```
et_slime, //スライム(1)
et_skelton, //ドクロ(2)
et_bat //コウモリ(3)
};
```

と言った具合に表すことができます。

enum は内部的には 0 番から開始して、通して番号が入るようになっているので、int にキャストして使用することも可能です。

こいつは『マジックナンバー』を防ぐためにあります。例えばなんかしらのフラグとか種別番号とかで 189 とか 32 って数値を書いているバカがいますが、そんな奴はプロのゲームプログラマーとしては認められません⇒ゲーム会社に受からない。

ということ覚えておいてください。

なお、enum はあくまでも、通しの種別等に使用するのに向いているものなので、ビットマスクとか画面の固定幅を表すには constant 等を使ったほうが良いでしょう。

ともかく『マジックナンバー』は極力避けていきましょう。

とりあえず『クソコード』と呼ばれるものには

- マジックナンバー(わけわかんねー数字)
- クソ長い関数(100 行超えるな)
- ゾンビコード(消せっ…!)
- クソ深いネスト(深すぎるっ…!)
- コピーコード(関数化しろよクソがっ…!)
- バカコメント、嘘コメント(さようなら〜、そんな害悪は消えてしまえ)
- 名前が不適切(もっと頭使えよ…)

などがあります。

プログラマーにとってソースコードは履歴書みたいなものです。いや…ゲーム会社ならば履歴書よりもその人のパーソナリティを表現するものだと思います。

とりあえず企業に提出するコードにこれらの欠点が含まれていたら、どんなに履歴書を綺麗に取り繕っても面接でそつなく返答したとしても、クズコードを書く奴はクズなのだ扱いされますので、十分に気をつけてください。

enum class/enum struct

enum にもちよつとした弱点があります。それは、ぶっちゃけ単なる constant 定数と同じなので、名前がかぶってることがあるわけです。

```
enum Color
```

```

{
    RED,
    BLUE,
    PURPLE,
};
enum TrafficLight
{
    RED, //怒られますよ
    YELLOG,
    GREEN
};

```

という風に、同じ名称の定数を作ることができないのです。ですから今までだとこの場合

cl_RED

とか

tl_RED

みたいな定数の書き方をしていました。これを解消するための仕組みが enum class です。↑
の enum を enumclass に書き換えると…

```

enum class Color
{
    RED,
    BLUE,
    PURPLE,
};
enum class TrafficLight
{
    RED, //( ^ w ^ )通るよ
    YELLOG,
    GREEN
};

```

というわけです。これも面倒な部分があると言えばあって、int に暗黙変換してくれないのです。その場合は明示的にキャストする必要があります。

また、enumclass の場合は RED を使おうとすると

Color::RED

TrafficLight::RED

のような書き方になります。

STL(Standard Template Library)

STL ってのは C++ における便利ライブラリの事です。

C++ を勉強するうえでは STL の理解は必須といってもいい。そういう奴です。もう文法の一部にした方がいいんじゃないんじゃね？ってレベルです。

https://ja.wikipedia.org/wiki/Standard_Template_Library

さて、ウィキ見てもらったところで基本的な STL のコンテナを紹介すると

vector

動的配列。

deque

両端キュー(Double Ended QUEUE の略)。

list

双方向リスト。

set

集合。順序付けられている(このためツリーにより実装されるのが通常で、他言語のライブラリによく見られるハッシュテーブルではない)。

map

連想配列。順序付けられている。実装に付いては set と同様。

multiset

要素の重複が許されている set。

multimap

要素のキーの重複が許されている map。

これくらいの種類がある。

ただ、ゲームで頻繁に使用するのは vector と map なので、最初は vector と map だけでいいだろう。

std::vector

vector ってのは動的配列を実装するものです。動的配列を作ろうとすると、一部の偏屈な人は vector を使わずに頑張っ

```
int* a=new int(要素数);
```

などと宣言します。これも間違いじゃないんだけどいろんな問題を含んでいる。まず、動的確

保なので malloc と同様に『解放』を明示的に行わなければならないんだが、その解放も面倒で

```
delete() a;
```

などを書かなければならない。非常に面倒。なので vector を積極的に使おう。

まず、vector を使うには、vector をインクルードします。

```
#include<vector>
```

これで準備完了であります。

で、こいつを使うには std のひとつだによって意味をこめて、std:: を頭に付ける必要があります。

なので、例えば int のベクタ配列を作りたいければ

```
std::vector<int> vectorhaireru;
```

なんていう風に宣言します。一般化して書くと

```
std::vector<配列にしたい型> 変数名;
```

となります。こいつは、std::vector<配列にしたい型> がまたひとつの型として扱われますので、要は、構造体みたいな扱いができるってことです。つまり関数の引数として渡したい場合は

```
void Sample(std::vector<Aho>& ahos ){  
    //うぼあー  
}
```

って言う風を書くことができます。ここでのポイントは&。なんで&をつけているのかというと、前にも言ったように、基本的に C++ の代入は『コピー』であって『参照』ではない話をしました。

つまり、関数の引数として、

```
void Sample(std::vector<Aho> ahos )
```

という定義をした場合、引数に入れた際に、ベクタ配列のコピーが発生する。コピーって結局すべての要素のコピーなわけ。つまり、こいつの数が 10000 とかあった場合、非常に処理時間が勿体無いことになる。

なので、vector を引数として渡す際には & をつけることをおすすめします。

さて、このベクタ配列がすぐれものなのは、ホントに配列っぽく使えること。

```
vector<int> vec(要素の数);
```

とでも宣言してやって、

例えば、先頭の要素にアクセスしたければ

```
vec[0]=56;
```

などというふうに、配列でやったような感じでアクセスできる。これは実は、C++ の演算子オーバーロードのおかげなのだ。いわゆる添字演算子オーバーロードである。

しかし、ここでそれ言ってもわかる人あまりいない。それ困る。やらない。俺、定義だけ書く

```
int& operator()(int i){ return i 番目の要素; }
```

こんな感じ。分かる人だけわかればいい。分かりたい人は自分で勉強するね。

じゃあよくある、配列の先頭のアドレスを返すってやつはどうやるのか？

それは

```
&vec[0]
```

とやってあげる。これが、ベクタ配列の先頭のアドレスとなる。なのでたとえば、f m f ファイルを読み取る時はこうやります。あ、その前に言うておくべきことが、ベクタ配列の大きさですが、

宣言時に

```
vector<型> 変数名(配列の大きさ);
```

って、指定する方法と、

```
vector<型> 変数名;
```

```
変数名.resize(配列の大きさ);
```

って指定する方法があります。どっちでも、好きな方で…

ちなみに配列要素数は

```
変数名.size();
```

でいつでもとってこれる。便利でしょ？

std::map

これはいわゆる『連想配列』というやつです。そもそも『連想配列』という言葉がわからないかもしれませんが、簡単に説明すると…通常の配列の場合、要素を特定するには整数のインデックスを使用しますよね？

```
a[0]=1;
```

```
a[1]=5;
```

```
a[2]=7;
```

ね？これは分かるよね？でもたまにこの『インデックスに文字列とか使えへんかなー？』って思ったことないですか？

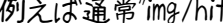
```
a["ぬこ"]=10;
```

```
a["イッヌ"]=5;
```

```
a["うさぎ"]=1024;
```

なんていう風に…それができるのが std::map なわけです。

まだそういう需要は無いのかもしれないけど、よくやるのはファイル管理で、『一度ロードしたものを2回ロードしたくない』なんて時に使います。

例えば通常何ていうファイルをロードしたいとするじゃない？

ところがヒットエフェクトなんざ色々なところで使用されているわけだ。通常の対処法としてはゲームの最初に必要なものをぜ〜んぶロードしてしまっって、その変数を公開状態にしてみんながそれを使用するって方法なんだけど、それだと設計段階でどのファイルが必要なのかどうかをぜ〜んぶ網羅しておかなければならない。面倒だし、絶対漏れが発生するし、場合によってはタブリも発生する。

いいか？ソフトウェアにおいて…特にゲーム開発においては、仕様の変更など当たり前に発生する。根っこから変わるのも珍しくない。

出来る限り決めておいてほしいものだが、そこはソフトウェアを作る際には仕方ないと思っておいたほうが良い。まず『完璧な仕様書、設計書などというものは存在しない』とおこう。

そうなると、プログラマに待っているのは絶望しかないわけで、だいたいプランナーやアーティストに『今更無理です』という。

これは確かに昔々ならばアリだった。だが、今は違う。特にソシャゲが優勢になってきて、理不尽なぐらいの仕様変更が当たり前になってきて、それに対応できないプログラマなどいないという時代になってしまった。

というわけで、プログラマとしては『いつ仕様変更が起きてもそれに耐えられる作りにしよう』これがプログラマの役割である。

というわけで、話が逸れたが map の例を示す。

予め

```
std::map<std::string,int> _resouceMap;
```

(中略)

```
int Load(std::string& filePath){
```

```
    //まずそのファイルをロード済みかどうか確認
```

```
    std::map<std::string,int>::iterator it=_resouceMap.find(filePath);
```

```
    if(it==_resouceMap.end()){//なかった
```

```
        _resouceMap[filePath]=LoadGraph(filePath);
```

```
    }
```

```
    return _resouceMap[filePath];
```

```
}
```

こんな感じ。もしくは

```
std::map<std::string,int> _resouceMap;
```

(中略)

```
int Load(std::string& filePath){
```

```
    //まずそのファイルをロード済みかどうか確認
```

```
    if(_resouceMap.count(filePath)==0){//なかった
```

```
        _resouceMap[filePath]=LoadGraph(filePath);
```

```
    }  
    return _resouceMap(filePath);  
}
```

てな感じでロード関数をつくれば無駄ロードを防ぐことができます。

std::string

あ、そういえば、しれっと使ってて、こいつの紹介を忘れていたなあ。

使うには

```
#include<string>
```

として使います。

こいつが便利なのは文字列をつなげたりするのがC言語のchar*なんかより超便利ってことです。

例えば

```
std::string str;
```

などという風に宣言します。(std::string までが『型名』です。std::vector とかと同じです)

中に値を入れる場合は

```
str=" 中身";
```

のようにすれば str の中身は"中身"という文字列になります。

文字列の長さが大きくなっても大丈夫。自動で長さを計算してくれるメモリを確保してくれます(vector と一緒やね…まあそれによるデメリットもあるんやけど、今は考えなくてもよからう)

さらにこの string 型文字列の連結も自動でやってくれます。つまり

```
str+="はかうだよ";
```

とすれば str の中身は"中身はかうだよ"という文字列になります。

便利でしょ？

あれ？でも例えばC言語の関数にこの文字列を入れようとするとうなるの？例えば

```
DrawString(0,0,str,0xffffffff);
```

ってのは可能なの？

残念ながらそこまでは親切じゃないんだなあ…。Cの関数に使うにはやっぱりC言語用の文字列表現にしなければいけない。

そこで便利な関数。

ここは重要ですよ？だから目を開いて耳をすましてください。



その関数の名はc_str()関数です。

```
c_str()
```

C アンダーバーstr()

まあそういう事です。string 的文字列表現をCの表現に変えますので char*ポインタ型に変換するものです。つまり

```
DrawString(0,0,str.c_str(),0xffffffff);
```

とやれば通ります。

さらにこのstringの偉いところは、文字列の比較も簡単だということです。

C言語の場合、文字列の比較には==は使えませんでした。

```
char name[]="abcde";
```

```
if(name=="abcde"){}
```

などという比較は不可能でしたね。

ところがstringはそれが可能なのです。

例えば

```
std::string name="abcde";  
if(name=="abcde"){}
```

は予想通りの結果となります。マジ便利でしょう？

ついでに便利な関数も紹介しておきます。

vector とほぼ同じ構造なので vector の関数はフツーに使えます。

つまり、size()は文字列の長さを表し、empty()で文字列が入っているかどうか判別できます。

それ以外の便利な関数は…

c_str()	C の文字列表現を返す
data	C の文字配列表現を返す
copy	文字列のコピー
find	文字列の中の指定文字列を検索して、その場所(インデックス)を返す
substr	文字列の一部を取り出す

など、まあ今のところはそれほどありがたくないかもしれませんが、知っておくのと知っていないのとでは結構プログラミングの戦略が変わってきます。

こいつと stream ってのを組み合わせると非常に強い効果を発揮しますが、アレは若干難しいので、またの機会にしましょう。もう少しスキルが上がってからね。

ストリームについてちょっとだけ

STL って組み合わせられるのよ？

思い込みというのは恐ろしいもので、map や vector というのは一度に一種類しか使えないと思っていたりします。

大丈夫、組み合わせさせて使えます。

ですから、

```
struct Nanika{省略};
```

```
std::map<int,vector<Nanika>> chaos;
```

などという使い方も可能です。

なお、

```
chaos[14].push_back({~初期化~});
```

という風に値を入れることもできます。

テンプレートについて

テンプレート、これはいいものだ。ただしダークサイドに陥る危険性もはらんでいる。ちょっと使うくらいならいい。

だが、

http://ja.wikibooks.org/wiki/More_C%2B%2B_Idioms

に載っているテクニックや

『ModernC++Design』って本のテクニックを読むと間違いなくダークサイド行きである。もはや頭がおかしいレベル。

[http://f3.tiera.ru/other/DVD-](http://f3.tiera.ru/other/DVD-009/Alexandrescu_A._Modern_C++_Design(C).Generic_Programming_and_Design_Patterns_Appl)

[009/Alexandrescu_A._Modern_C++_Design\(C\).Generic_Programming_and_Design_Patterns_Appl](http://f3.tiera.ru/other/DVD-009/Alexandrescu_A._Modern_C++_Design(C).Generic_Programming_and_Design_Patterns_Appl)
[ied_\(2001\)\(en\)\(271s\).pdf](http://f3.tiera.ru/other/DVD-009/Alexandrescu_A._Modern_C++_Design(C).Generic_Programming_and_Design_Patterns_Appl)

boostの中身を見ても、頭がオカシイことが分かるだろう。

テンプレートは便利すぎるために、使用法を誤ると KittyGuy になっちゃう。くれぐれも注意した上でこれからの話は聞いて欲しい。

テンプレートというのは型を特定しないで様々な関数やクラスを定義できるというスグレモノなのです。たとえば、2つの数のうち大きい方を返すような機能をほしいとする。このような機能は整数型だろうが浮動小数点型だろうが使ってみたい。

こういう時に使えるのだ。後で説明する関数テンプレートを使用すると

Max(1,7)は整数型の7を返し、Max(1.6f,9.0f)は浮動小数点型の9.0を返すことになる。

最後まで聞けば感じる人もいるけど、テンプレートは型を特定しないとかそんなチャチなも

のじゃ断じてねエ。もっと恐ろしい物の片鱗を味わうだろう。

関数テンプレート

先ほども言ったように関数テンプレートは型を特定せずに Max 関数やら Add 関数やらを作れるものである。

で、ちなみに関数テンプレートの使い方は

```
template<class 仮の型名> 戻り値 関数名(パラメータ){
    ほにゃらら、ほにゃらら
}
template<typename 仮の型名> 戻り値 関数名(パラメータ){
    ほにゃらら、ほにゃらら
}
```

である。キーワードは template と class もしくは typename である。で、class と typename にはとりあえず違いはないと思っておいていいです。ですから好きな方を使いましょう。
たとえば先ほど例に出した Max 関数を作るのならば

```
template<typename T> T Max(T a, T b){
    return a>b?a:b;
}
```

などという定義をすることができる。

で、仮の型名をここでは T と置いてはいるが、何でもいいて、数字や変数が入った時点で自動的に int 型の Max 関数や float 型の Max 関数が生成される。

つまり

```
cout << Max(1,9) << endl;
```

などと言った瞬間に

```
int Max(int a, int b){
    return a>b?a:b;
}
```

が生成され、

```
cout << Max(1.0f,7.2f) << endl;
```

と言った瞬間に

```
float Max(float a, float b){
```

```

        return a>b?a:b;
    }

```

というコードが見えない所で生成されるわけです。ここまで読んだ人はもしかしたら『え？テンプレートって指定できる型名は1つだけなの？』と思うかもしれませんが、複数設定してみましょう。

```

template<typename Ta,typename Tb> Tb Scaling(Ta value,Tb scale){
    return (Ta)value*scale;
}

```

なんて書いて

```
cout << Scaling(2,5) <<endl;
```

とでも書けば

```

int Scaling(int value,int scale){
    return (int)value*scale;
}

```

が生成されますし、

```
cout << Scaling(5,5.9f) <<endl;
```

とでも書けば

```

float Scaling(int value,float scale){
    return (float)value*scale;
}

```

というのが見えない部分で生成されます。ここに挙げた例だとあまりありがたみがありませんが、ぼちぼち利用できそうな場面を見つけて活用…別にしなくてもいいですが、知識としては覚えておきましょう。C#でも『ジェネレータ』という名前で使用されますので、こういう『型がコンパイル時に決定される』系の話は頭の片隅に入れておきましょう。

クラステンプレート

関数テンプレートはまだわかりやすかったかもしれませんが、次は少しだけマニアックになってきます。なってくるのですが、実際はこいつが本番なんですねー。C++さんはホンマ初心者殺しやでえ…。

クラステンプレートってのは、要はクラスのメンバの型をコンパイル時に決定するというものです。

定義は

```

template<typename 仮の型名> class クラス名{
    仮の型名 メンバ変数;
    仮の型名 メンバ関数();
    int a;
}

```

```
void func();

};
```

と言った具合に定義します。ポイントは赤字で書いてある部分です。関数テンプレートの時と同様に `typename` または `class` というキーワードで仮の型名を定義するところまでは同じで、クラステンプレートの場合、メンバに対して仮の型名を使えるというところが強力なのです。

例えば、前回使ったベクトルクラスは `int` 型向けにつくりました。これを `float` 型向けに作るとして、わざわざ

`Vector2f` なんてクラスを作りますか？作ってもいいですが、このクラステンプレートを使用すると簡単で。

```
template <typename T>class Vector2{
public:
    T x;
    T y;
    Vector2():x(0),y(0){}
    Vector2(T inx,T iny):x(inx),y(iny){}
    void operator+=(const Vector2<T>& v){
        x+=v.x;
        y+=v.y;
    }
    (略)
```

などという感じに書き換えられます。これで `Vector2<T>` というのは `int` 型だけではなく、`float` 型など様々な型に対して対応することができるようになります。

新しいC++言語(C++11/14)

言語仕様ってのは定期的に進化しています。C++も2011と2014で大きな変化を遂げています。僕が現役の頃とはまた仕様が変わっているわけです。恐ろしくもあり(ﾟдﾟ)おもしろくもあります(´▽`)。VisualStudio2015からはこの辺の機能が色々糖鎖卵衣されてきたので、せっかくだから知っておきましょう(使わなくてもゲームは作れますが…)

それではいくつか紹介します。

`nullptr`

ぬるぽ

C 言語や新しくない C++ の場合は無効メモリを示すために NULL を define 等で定義して使っていました。これは実は中身が 0 であり、厳密なヌルオブジェクトとは言えませんでした。

これはキモチワルイし将来的に色々困るかもしれませんので nullptr という正式なヌルオブジェクトが C++ で定義されました。ヌルポを指定する時に C++ では nullptr を指定しましょう。

auto

アウトじゃないです。オート(自動)です。つまり自動で型を決定してくれる機能です。
つまり...

```
auto seisu=10;  
auto shosu=3.14f;
```

などと記述することが可能なのです。

え? そんなの C/C++ じゃないやい!!! と思うかもしれませんが、そこは大丈夫。C/C++ はそれでもまだ静的型付け言語ですよ。auto で宣言された変数もきちんと型を持っていますし、実行中に型を変更することはできないです。

タネを明かすと、

auto で宣言される変数の型はプリコンパイル時に右辺値の型により決定される。

です。つまり僕らにとっては『型なし』に見えるけど、コンパイラは右辺値を見て、あ int 型やな? float 型やな? ってやってくれるわけです。ですから前の例は

僕らには

```
auto seisu=10;  
auto shosu=3.14f;  
こう見えていますがコンパイラには  
int seisu=10;  
float shosu=3.14f;  
のように見えてるわけです。
```

これが auto です。便利でしょ? まあこの使い方だとまだ便利さは大して見えてこないのですが『範囲 for』と組み合わせると結構便利に感じると思います。

ForEach 的な for

『範囲 for 文』と言って、C++では `for(i=0;i<MAX;++i)` 的な for 文だけではなく、ちょっと変わった for 文が搭載されています。最近では大抵の言語で搭載されているんですが…言語によっては `foreach` なんて言い方をしたりします。

なんじゃそりゃ…と思うかもしれませんが、とりあえず C 言語の for 文を思い出してみよう。

```
int a[]={1,3,5,7,9,2,8,4,6,0};
```

という配列を出力するには

```
for(int i=0;i<sizeof(a)/sizeof(int);++i){  
    printf("%d\\n",a[i]);  
}
```

とするでしょう？

こういうのをもう少しすっきり書く方法が C++に追加されています。

文法は

```
for(型 変数名 : 配列とか){
```

配列の内容が変数に入った状態でループを回ります

```
}
```

という文法が追加されていますので、先ほどのループを書き換えると

```
for(int e : a){  
    printf("%d\\n",e);  
}
```

こうなります。シンプルでしょ？

さらに `auto` を使うと

```
for(auto e : a){  
    printf("%d\\n",e);  
}
```

と…実はこの書き方が最近の定番なので覚えておいたほうが良いかもしれません。

あ、この書き方の注意点ですけど…値の取得だけならこれで問題ないんですが、値の設定をしようとする、上の書き方では問題あるんですね。例えば

```
for(auto e : a){  
    e=rand();  
}
```

としたとするじゃないですか？

そしたら、実際にはaの配列の中身の値がランダムになってなきやいけないんですけど、このやり方ではそうはなりません。何故なのでしょう？

それはeが『配列要素のコピー』に過ぎないからです。ですからループを抜けた後では代入の意味がなくなっちゃってるわけです。これに対処するには…

```
for(auto& e : a){  
    e=rand();  
}
```

アンパサンド(&)をつけます。

最後にちょっとキツツイやつを教えます

ラムダ式

C++11 から追加された機能として『ラムダ式』ってのがあります。関数プログラミングでよく出てくる話なんですけど、関数プログラミングからのアプローチだとかなりややこしいので、結論から書くと…

文法は

```
[束縛値](パラメータ){  
    ここに関数の中身を書く
```

}

となっています。

…何それ？関数名ないの？…ありません。

いわゆる『無名関数』となっており、変数に代入することもできる

一番簡単なラムダ式は

```
()(){}; // 角括弧、丸括弧、中括弧、セミコロン
```

である

何もしない関数です。

もちろんこれを変数に入れることもできる

```
auto function=(){ };
```

```
function(); // 呼び出し
```

などというふうにも使える。使えるが大してうま味はないようにみえるね？

例えば『何かを 10 回繰り返す関数』という相当アバウトなものがあつたとする。この『何か』つてのがここで代入するラムダ式だとすると

//とにかく 10 回繰り返すマン

```
void Repeat10Times(void(*func)()){
```

```
    for(int i=0; i<10; ++i){
```

```
        func();
```

```
    }
```

```
}
```

を予め作っておいて

```
Repeat10Times(){ }{printf( " lambda\n" );};
```

なんて書くと lambda が 10 回出力されます。

何となく分かるだろうか？

では次に引数とかについて考えてみよう。引数は丸括弧の中を書く引数に2つの int を使うのならば


```
{ }(int a, int b) {};
```

という風に記述する。普通の関数と同じやね。

通常は関数内関数が定義できないんだけどラムダ式だけは定義できるのだ。

では最後に角括弧は何を表しているんだろう。

先ほど『束縛値』と書いたが、これは『環境』や『クロージャ』と呼ばれるもので、C++ではこれを『キャプチャ』という。

このラムダ式を定義した時のオブジェクトそのものを、ラムダ式内で使えるようにするためのものだ。

通常はラムダ式の中からは引数に対してしかアクセス出来ない。

だが、この『キャプチャ』を使えば呼び出し側の持っている情報に直接アクセスできる。

例えば

```
func=(this)(){  
    this->hensu=10;  
};
```

としてクライアント(呼び出し側は)が

```
func();
```

としたとすると this には呼び出し側の this オブジェクトが入るのです。

これが結構強力なツールになります。そのうち必要になるので何となく頭の片隅に入れておきましょう。

もっともといういろいろとありますけど、頭パンクするでしょうしこんなもんで。

参考資料

C++ 言語

C++そのものの本を買う必要はないです。敢えて勧めるなら

[プログラミング言語 C++ 第4版](#)

です。ただね…税込みで 9504 円なのよね。これ消費税率 10%になるともっとヤバイことになるわけです…まあ正直 C++ 言語研究者とか言語実装者でもないかぎりここまでの必要はないです。

んで次に思い浮かぶのは『独習 C++』ですが、これはちょっと色々足りてない(初期化リストとかの記述がない)ので、お金がかからないぶん、まだインターネットの Web サイトのほうがマシです。

おすすめサイト

[猫でもわかるプログラミング C++ 編](#)

[ロベールの C++ 教室](#)

[1週間で身につく C++ の基本](#)

[C++11 の文法と機能](#)

こんな感じですね。ひと通り見れば大体身につくと思います。これでもし『うん、俺ある程度 C++ わかってきた』って実感してきたら…



[ゲームプログラマのためのコーディング技術](#)

がオススメです。

それでもこの本も 2,678 円するので買う前に SlideShare の

[オブジェクト指向でできていますか？](#)

を見てから作者の傾向を考えて、自分に合いそうだったら購入を検討してみましょう。

もしその上で C++ を極めていきたいのであれば

- [Effective C++](#)
- [More Effective C++](#)
- Effective STL (絶版…見たい人は僕にご相談ください)
- [Effective Modern C++](#)
- C++ Coding Standard (絶版…見たい人は僕にご相談ください)
- Modern C++ Design (絶版…見たい人は僕にご相談ください) ← オススメはしないが興味深い

を読んで『良い C++』というものを知っていきましょう。

あと、SlideShare の大圖氏のレポートは読む価値はあると思います。

https://www.slideshare.net/MoriharuOhzu?utm_campaign=profiletracking&utm_medium=sssite&utm_source=ssslideview

あっ、ごめん忘れてました。最後に重要なものを書いておきます。

スマートポインタ

C++ 言語において、ポインタを生のまま扱うのはよろしくない。実にいただけない。美しい。

だから僕は極力ポインタを使いません。どうしても使用しなければならない場合はスマートポインタと言うものを使います。

通常であればポインタと言うのは、生成したら削除するものです。つまり new したら必ず delete する。そういうものではあるのですが、この delete 忘れによるバグ、メモリリークがとにかく多いのです。これに対処するために考え出されたのが『スマートポインタ』というものです。

シンプルに言うと delete のタイミングをプログラマが管理するのではなく、必要がなくなったら自動で削除される仕組みのことを総称してスマートポインタと言います。

ですので、僕の授業においては自分から delete を使用することはほとんどないと思ってください。

C++ 標準で使われるスマートポインタはだいたい3つです。よく使用する順に書くと

- shared_ptr
- weak_ptr
- unique_ptr

です。僕としては scoped_ptr が欲しいところなのですが、標準ではないためだいたい僕は自作します。スマートポインタの中で最も自作しやすく、練習に持ってこいと言う意味でも使えるものです。

こいつらを使用する際はちょっとテンプレートの書き方に慣れておく必要があるけど、まあスマポ使い始めたら頻繁に使用するので、すぐに慣れるでしょう。

shared_ptr

シェアドポインタは、コスプレイヤーさんと、カメラ小僧の関係です。1人のコスプレイヤーさんを沢山のカメラで参照することができます。



ところが、残念ながら物事には飽きというものがあり、多かったカメラ小僧も1人、また1人と減っていきます。そして最後のカメラ小僧に飽きられた時…コスプレイヤーさんはコスプレイヤーさんでなくなります。普通の少女に戻るのです(メモリ解放)

使用するには

```
#include<memory>
```

をインクルードします。これで使えるようになります。

ひとまず、参照人が一つだけの時は `ScopedPtr` と同じ挙動になるので、

```
{
    shared_ptr<A> a(new A);
    a->Func();
}
```

これで同じ結果になるのを確認しましょう。

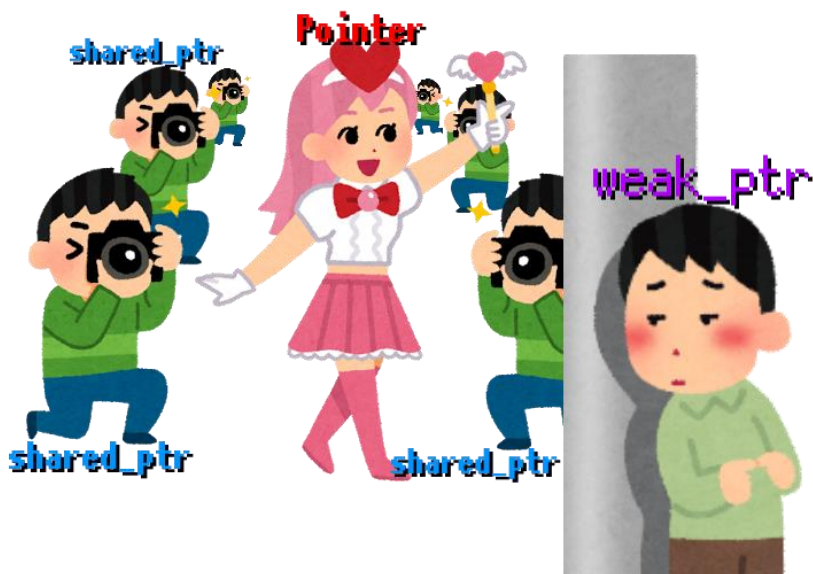
次にスコープ外の他の `shared_ptr` に代入した場合を考えてみましょう。

```
shared_ptr<A> b;
{
    shared_ptr<A> a(new A);
    a->Func();
    b=a;//ここで所有権が共有されて、スコープ抜けても解放されない
}
```

この場合ならばスコープ外の `b` が参照カウンタを上げているので、解放されないはずです。

まああとは使いながらとか C++ の授業で深いところまでやっていきましょう。

weak_ptr



これはこの名の通り『弱参照ポインタ』です。その名の通りとか言われても弱参照なんてわかんねーよ! って思うでしょうね? こいつは監視と参照はしてるんだけど、破棄に感しては全く関与しないポインタです(遠くで見てるようなやつです)。つまりさっきのサンプルの b の部分を weak_ptr にすると... a のスコープを外れた際に解放されてしまいます。

```
weak_ptr<A> b;  
{  
    shared_ptr<A> a(new A);  
    a->Func();  
    b=a; //ここで解放されてしまう  
}
```

ただ、ここまでだったら b をわざわざ weak_ptr なんかにした意味が無いよね? これが通常のポインタであれば、解放されたことを知る術が無いんだけど、weak_ptr の場合は expired って関数を持っています。

https://cppref.jp/github.io/reference/memory/weak_ptr/expired.html

もし expired() の戻り値が true だったら解放されてるんですよ。

つまり、これを使えばタングリングポインタに悩まされることがなくなります。

```
if(!b.expired()){  
    b を利用  
}
```

って書けば、b が既に解放されていれば処理を行わないって事ができるわけです。弾が生きて

るのかどうかをコイツを使って監視することができますね。

unique_ptr



…この絵で全てを察しろ。

unique_ptr とは本彼(ほんかれ)である。基本的に二股は許されませんポイントです。

もし他の unique_ptr に代入しようとする、それまでの本彼は元カレになり、新しい彼が本彼になります。

これを所有権の移動と言います。



悲しい表現ですが、こういうことです。こうなってしまうと悲しいかな元カレはアクセスを禁

止されます。

ただ、略奪愛にはそれなりの面倒さがあるようで…

```
unique_ptr<Test>test(new Test());
```

```
unique_ptr<Test>test2;
```

```
test2=test;//エラー！！そんな簡単に略奪させないよ！！！！
```

は許されません。move セマンティクスってやつが必要になります。略奪したければ…

```
test2=std::move(test);
```

というわけです。

さて…いよいよ開発に入りましょうっ…!!

開発初期段階(初步設計的なものと実験)

今回は古いゲームの『スプラッターハウス』を題材とします。

特に教育的効果がどうかじゃなくて、ただ単に僕が好きだからです。

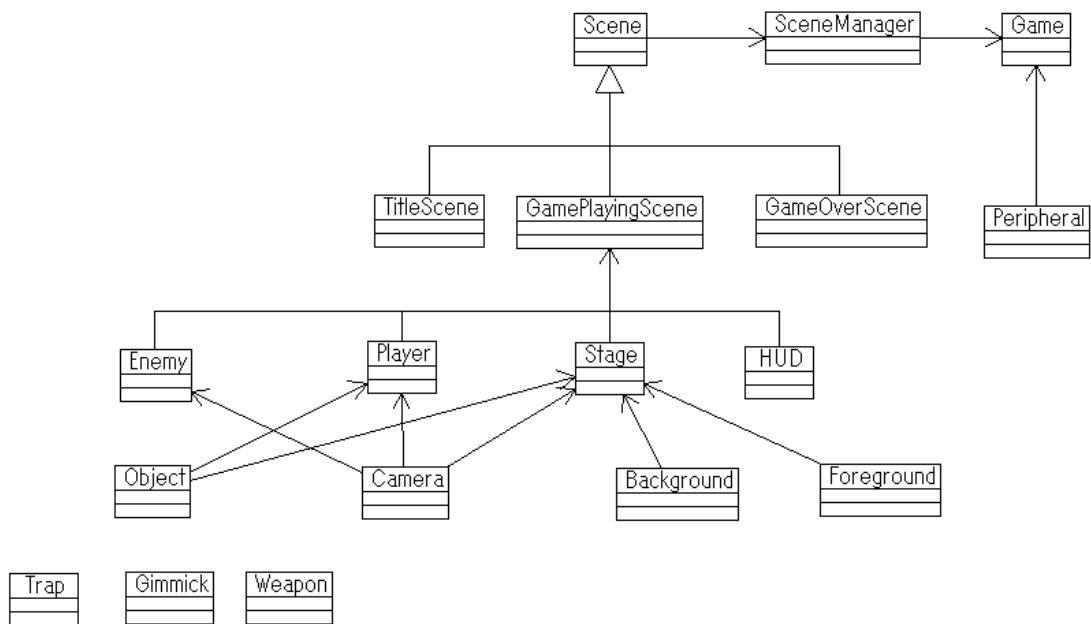


こういうちょっとおどろおどろしいゲームです。

<https://www.youtube.com/watch?v=lcS1hxTIFZU>

苦手な人は言ってください。

設計的なの



設計に関しては最初から凝った設計にはしません。おおざっぱに分かりやすいレベルからでいいです。

設計してからでないとは手が動かないというのは最悪だから、おおざっぱです。で、↑のやつも多分コーディングしながらリファクタリングしつつ修正します。

ちなみに、↑のクラス図を作るときにどうやって作ったのかっていうのですが、基本的には「登場人物」をクラスとして定義することにして、それ以外の「概念」系の奴は「名詞」として扱えるものをクラスとしています。

一般的なクラス設計法(ストーリー法)

参考までに一般的なクラス設計法のひとつを紹介しておきます。迷ったときに参考にしてみるのも良いでしょう。

- ゲームを表現する要素を文章、もしくは箇条書きにする
- 動詞にはあたるものはメソッド(メンバ関数)に、名詞に当たるものはクラスもしくは変数にならないか考える
- ↑の作業によりクラスがいくつかできてきたら、同じような役割のクラスがないか調べる。
 - ◇ 統合すべきであれば一つに統合することを考える
 - ◇ 統合すべきでないなら汎化(継承元をつくる)を考える

☆ 統合すべきでないなら、機能、特性をバラしてまとめなおしてみる

☆ 継承が不適切であればコンポジション(内包)を検討する

…軽くないか(°•ω•°)

例えばアクションゲームであれば

『ゲームはタイトル画面とゲーム画面とゲームオーバー画面を遷移します。それぞれの画面シーンは独立しており同時に表示されることはありません。タイトル画面はタイトル画像を表示。ゲームオーバー画面はゲームオーバー画像を表示します。ゲーム中では、プレイヤーを示すキャラクターと背景と敵が存在し、ステージの中には罠やアイテムがあります。ユーザーはプレイヤーを PAD で操作し、プレイヤーは罠からダメージを受け、アイテムを武器として装備し、使用する事ができます。背景には背景画像と前景(プレイヤーの手前に表示されるモノ)があり、また、プレイヤーの体力や点数やアイテムや残機を示すための UI(HUD)が存在します。プレイヤーが左右に動くと背景や敵や前景がスクロールします。プレイヤーをカメラが追いかけているイメージです。』

というゲームの仕様を示す文章から抜き出していくのですが、『名詞』に当たるのは

- タイトル画面
- ゲーム画面
- ゲームオーバー画面
- プレイヤー
- ステージ
- 背景
- 敵
- UI

くらいは出てくるでしょうから、これらはひとまずクラス化しておくことになります。

さらに画面遷移の画面は汎化して『シーン』という概念にまとめられそうです。

また、システム的な部分としては、シーン遷移をコントロールする SceneManager を用意し、入力部分を『外部入力装置』という事で Peripheral というクラスにしておくのと、プレイヤーに合わせてスクロールをコントロールする Camera を導入します。あと全体的なシステム管理として Game クラス(シングルトン)を作っておきます。

なお、文章中に出てきた動詞はそれぞれの『メソッド/メンバ関数』になるという設計になりますがそれは後々考えましょう。

まあ今はそこまで細かく考える必要はない。さっさと手を動かしましょう。

クラスの作り方

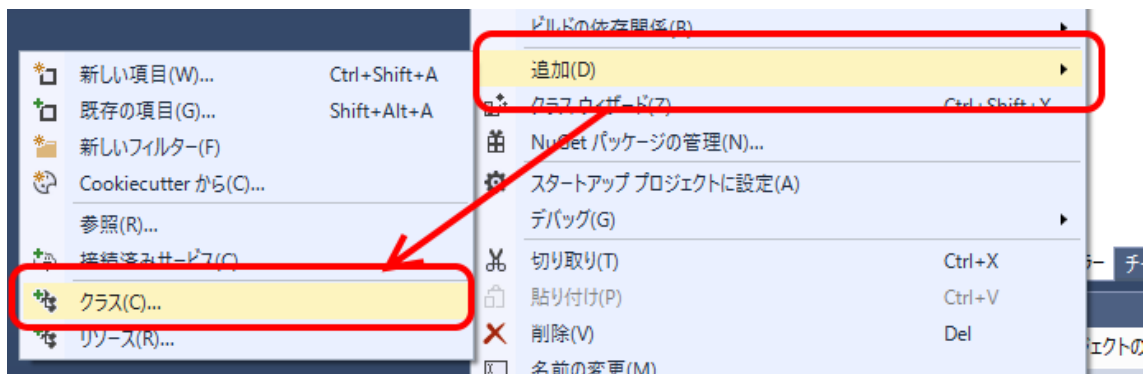
クラスを作っていきます。クラスの作り方と言っても、全然大したことないのです

```
class Enemy{  
    public:  
        Enemy();  
        virtual ~Enemy();  
};
```

で作れちゃうので、まあなんてことはない。

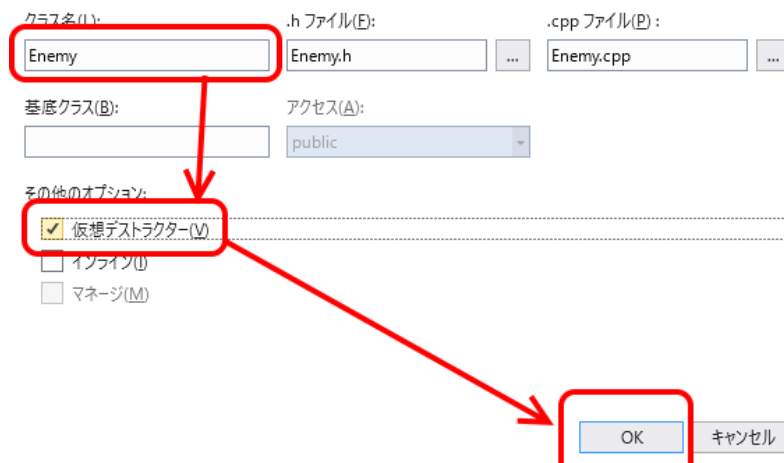
のですが、せっかくだから俺はこの VisualStudio の機能を使ってクラスを作るぜ!!

というわけでプロジェクトを選択した状態で右クリック→追加→クラスをクリックしてください。

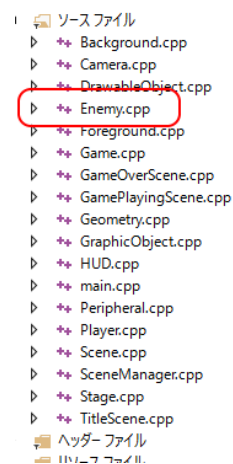


↓こんな画面が出てきますので、クラス名称入れて『基底クラス(自分から継承する可能性がある)』の場合は仮想デストラクタにチェックを入れておいて OK ボタンを押してください。

クラスの追加



これでクラスの生成ができますので、おおざっぱに必要なクラスを作っておいてください(実装はまだ必要ないです)



ほら、このように

ともかく、これであらかたクラス設計ができたかなと思います。main 関数を持っているのは main.cpp という事にしています。しばらくはそこをいじっていきましょう。

読み込み→表示(第一段階)

ただの表示

あ、一応コンソールアプリケーションとして作ります。コンソールアプリケーションとして作っても DxDLib_Init を使えばウィンドウは出ますし。

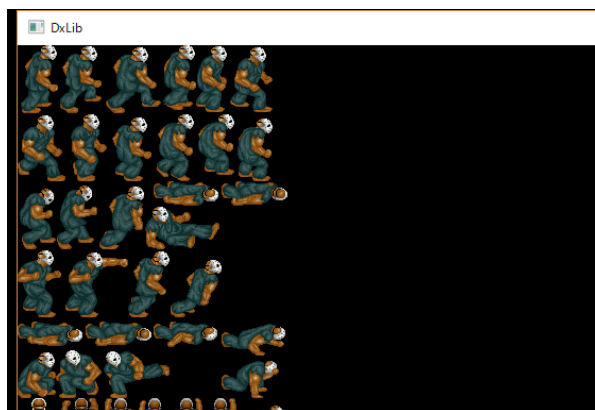
ちょっと昨年を思い出しながらまずはウィンドウを出してみましょう。いかがですか？別に DirectX でやっても構いませんが…。ちなみに DirectX でやるときにコンソールからウィンドウハンドルを取得する方法は GetModuleHandle(0) です。

一応、サーバーの

[¥¥132sv¥gakuseigamero¥rkawano¥Action18¥img](#)

の中に画像ファイルがありますので、それを読み込んで表示してください。

まあ、こうなるんちゃうかと…



それで構いません。DxDLib における HelloWorld ですね。

寂しいので背景も表示させましょうか



こんな感じ。

背景のデータがないんですよ…申し訳ないのですが…ゲームからぶっこ抜いてこれませんでした。

あつ『ただの表示』とか言って舐めてると SetDrawScreen 間違ってる可能性がありますので、この SetDrawScreen の戻り値が0であることは実行して確認しておいてください。

それはともかく左右反転して並べてみましょう。



まあ、カッコ悪いけど、仕方ないね。そして反転してるのがバレバレなので、ちょっと加工した上下のバーを上からかぶせてあげます。



とりあえずここまでやってみていただけますでしょうか？

あ、画面の大きさは768の448で表示しております。中途半端みたいですが、ワイド画面状態にしてます。4:3時代の作品だからって4:3のままやってもつまないからね。

次に、↑の状態ではプレイヤーが小さすぎますね。実際のゲーム画面では



この比率なので、プレイヤーの画像は倍くらいにすればちょうどいいと思います。

DrawRotaGraph では比率の設定ができますので、それで倍率を調整してください。

http://dxlib.o.oo7.jp/function/dxfunc_graph1.html#R3N10

こんな感じになればいいでしょう



大きさの比率もそこそこですね。

まああとは好みでバックに暗いBGMでも鳴らしておきましょう。

さて、このままでは主人公が何人もいることになってしまいますので、主人公を切り取りましょう。

DrawRectGraph で画像の一部を表示

まあ、やったことありますよね？ 忘れた人は

http://dxlib.o.oo7.jp/function/dxfunc_graph1.html#R3N20

を見るがいい。とにかく切り取ってください



…あつ、でもこれ拡大縮小機能ないですね。

DrawRectRotaGraph

というやつがありますので、それ使ってください。ちなみにマニュアルないですが、名前から想像してください。DxLib には割と『マニュアルにない』関数が多くて、そいつらが意外と使える奴だったりするので、困りものです。

ともかく



こうなれば OK です。

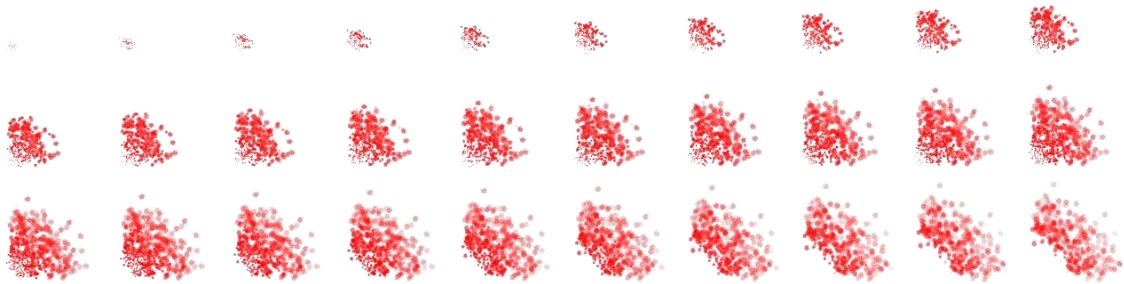
とはいえ、このままじゃ『動かない』のでツマンナイです。

等幅切り取りアニメーション

実はこの画像は、非等幅なのですが、別の画像素材で等幅のアニメーションから練習してみましょう。

Prominence というソフトを検索してダウンロードして立ち上げます。

で、適当にエフェクト作りますと



こういう画像が出力されます。

実はこれが等幅アニメーション状態となっており全部で $10 \times 3 = 30$ 個の構成要素となっている。これを順序良く実行するにはどうすればいいのだろうか？

まずフレーム毎に絵を切り替える必要があるのならば、フレーム変数(frame)を用意する必要があります。1 フレームごとにコマが切り替わると仮定するとこのアニメーションなら 30 フレームでアニメーションが終わります。

とすると、ループアニメーションであれば 30 になったらまた 0 に戻す必要があるのでフレームは

```
frame=(frame+1)%30;
```

となるでしょう。

一般化して書くと

```
frame=(frame+1)%総フレーム数;
```

てな感じですね。

わざわざ『一般化』して書いたのは、行き当たりばったりやっていると応用が利かなくなるからです。最初は数値を入れててもいいんですが、最終的には一般化して考えられるようにする癖をつけてください。

さらに DrawRectGraph の呼び出しを一般化して書くと

```
DrawRectGraph(X 座標, Y 座標,  
              ((frame)%列数)*1つ当たり幅, ((frame) / 列数)*1つあたり高さ,  
              1つあたり幅, 1つあたり高さ,  
              ハンドル, 透明フラグ, 反転フラグ);
```

こんな感じですね。

さて、これで等幅切り取りアニメーションの練習は OK ですね。では次に等幅じゃない切り取りアニメーションについて考えてみます。

等幅じゃない切り取りアニメーション

ちなみに今回のこのドット絵データに関してですが、



ご覧のように、等幅アニメーションではございません。

このような場合、非常に面倒ですが、画像のデータとは別に、画像を切り取るためのデータも一緒にロードする必要があります。

そのためのデータはサーバの

[¥¥132sv¥gakuseigamero¥rkawano¥Action18¥Action¥player.act](#)

というファイルに置いてあります。

ちなみに同フォルダに ActionTool.exe というのがありますが、これを用いて作ったデータです。自作のツールです。C#ソースコードは

¥¥132sv¥gakuseigamero¥rkawano¥Action18¥ActionTool

においてますので、好きに魔改造してください。

ともかく、このツールから吐き出されるデータは、元の画像への相対パスと、各アニメーションの切り抜き情報および中心点情報です。

ひとまずこの情報をもとに切り抜きますが、データはバイナリファイルなので

```
0F 00 00 00 2E 2E 2F 69 6D 67 2F 72 69 63 6B 2E
70 6E 67 02 00 00 00 12 00 00 00 E3 82 A2 E3 82
AF E3 82 B7 E3 83 A7 E3 83 B3 30 30 30 07 00 00
00 00 00 00 00 00 00 00 00 26 00 00 00 40 00 00
00 10 00 00 00 3C 00 00 00 0A 00 00 00 27 00 00
00 00 00 00 00 28 00 00 00 40 00 00 00 0E 00 00
00 3C 00 00 00 0A 00 00 00 51 00 00 00 00 00 00
00 36 00 00 00 40 00 00 00 0D 00 00 00 3B 00 00
00 0A 00 00 00 87 00 00 00 00 00 00 00 21 00 00
00 3D 00 00 00 0D 00 00 00 37 00 00 00 0A 00 00
00 A8 00 00 00 00 00 00 00 1D 00 00 00 3F 00 00
00 0B 00 00 00 3A 00 00 00 0A 00 00 00 C7 00 00
00 00 00 00 00 29 00 00 00 3F 00 00 00 0D 00 00
00 39 00 00 00 0A 00 00 00 00 00 00 00 41 00 00
00 2D 00 00 00 3F 00 00 00 0F 00 00 00 39 00 00
00 0A 00 00 00 12 00 00 00 E3 82 A2 E3 82 AF E3
```

こういうデータなわけですが

でも、このデータは僕が作ったモノなので、データ構造は分かっています。

まずヘッダ。

- 4byte: 画像ファイルパス文字列長⇒ (n)
- (n)byte: 画像ファイルパス文字列
- 4byte: アクションデータの数

で、ここからがデータとなります。データは、アクション⇒カットデータの入れ子構造

- 4byte : アクション名文字列長⇒ (m)
- (m)byte : アクション名
- 4byte: 切り抜き情報(カットデータ)の数
 - 4byte: 矩形中心 X
 - 4byte: 矩形中心 Y
 - 4byte: 矩形幅
 - 4byte: 矩形高
 - 4byte: 中心点 X
 - 4byte: 中心点 Y
 - 4byte: 継続フレーム数

となっています。

そう、本来はまず、これを読み込む必要があるのです。

(※あと、すみません。これ、僕がソールの仕様を変更するたびにデータ変更します。その時は告知しますので、実機の方で対応してください。実際の仕事もそんなもんですのでよろしくお願いします。)

ちょっとその前に読み込み以降を楽にする準備をしておきましょうか。

後々楽になるちょっとした部品を作っておく

いきなり読み込んでいってもいいのだが、先にちょっと幾何学的な部品を作っておきましょう。ひとまず 2D ベクトルクラスを表す Vector2D クラスを作ってください。

色々作っていくので、まとめたファイルとして、Geometry.h の

- 要素 X と Y を持つ(メンバ変数)
- + 演算子を定義する(オペレータオーバーロード)
- += 演算子を定義する(オペレータオーバーロード)

ひとまずは、基本としてこの定義を作ってください。

あー、テンプレとオペレータオーバーロードを組み合わせたり const 参照を入れたりするといいかな？

もちろんテンプレで作るなら

```
template <typename T>
struct Vector2D{
    Vector2D(T inx,T iny):x(inx),y(iny){}
    T x;
    T y;
};
```

こういう書き方になりますが、+ 演算子に関してはクラスのメソッドとしてより、クラスの外で定義した方がいいでしょう。

例えば、こう

```
template <typename T>
Vector2D<T> operator+(const Vector2D<T>& lval, const Vector2D<T>& rval) {
    return Vector2D<T>(lval.x + rval.x, lval.y + rval.y);
}
```

実質3行…もっと言うと1行で書けます。何をやっているかお判りでしょうか？

以前にも言ったようにテンプレートであったとしても

Vector2D<T>

までが、一つの『型』という認識ですので、例えば Vector2D の T=float だとすると

```
Vector2D operator+(const Vector2D& lval, const Vector2D& rval){
    return Vector2D (lval.x + rval.x, lval.y + rval.y);
}
```

となるわけですから、…まあ、何となく言ってる意味は分かっていただけるかな？と思いますが、一応説明すると、オペレータ+をオーバーロードしています。別にクラスのメンバである必要はありません。Vector2D である lval と rval のそれぞれの x 成分と y 成分どうしを足して、それを Vector2D のコンストラクタに入れて返しているだけです。

ちなみに+=などはクラスメンバ関数として定義した方がいれます。左辺値に変化が起ころうな場合はクラスメンバ関数にした方がいれます。やり方は分かりますかな？

```
void operator+=(const Vector2D& invec) {
    x += invec.x;
    y += invec.y;
}
```

こうです。まあ…解説はするまでもないかな

次にサイズを表す構造体を作ってください。この要件は1つでいいです。もっと言うと

- 幅と高さを持つ

これを満たせばいいです。

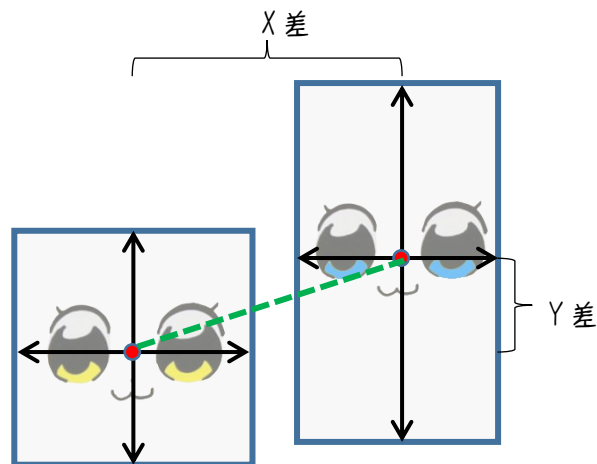
- int のほうは typedef で Vector2 という名前にしてください。
- ↑は座標としても使えるように Position2 という名前でも使えるようにしてください。

ベクターについてはそんな感じです。

次に矩形を表現する構造体を作ります。通常であれば通常は矩形は左上と幅と高さで定義するんですが今回は中心点と幅と高さで定義しようと思います。

何故かと言うと、矩形の当たり判定をしやすくするためです。

中心を基準に矩形を定義すると



ポプ子 とピピ美に見えてきた僕はもう重症です

さて、このように定義すれば中心点間のXの差とYの差が分かれば

$$X \text{ 差} \leq \frac{A \text{ 幅}}{2} + \frac{B \text{ 幅}}{2} \wedge Y \text{ 差} \leq \frac{A \text{ 高}}{2} + \frac{B \text{ 高}}{2}$$

が成り立つときに当たっているというわけです。1行で書けるようになるので楽です。なお、

Λ

は、数学記号で『かつ』を表すものです。つまり左辺の式と右辺の式が同時に成り立つという意味です。C 言語で言うと&&ですね。

データの持ち方を変えると、このように計算しやすくなることも多々ありますので、そういうのを常に考えるようにしてください。

また、当たり判定の矩形としては前述のような状況ではありますが、切り抜き矩形としては左上の座標も必要なので、Left 関数と Top 関数を作りましょう(参照のみで OK)

どうですかね～？

出来ましたか〜？

ぼくはこう書いてみました。

///矩形構造体

```
struct Rect {
    Position2 pos; //中心座標
    int w, h; //幅, 高さ
    Rect() : pos(0, 0), w(0), h(0) {}
    Rect(int x, int y, int inw, int inh) :
        pos(x, y), w(inw), h(inh) {}
    Rect(Position2& inpos, int inw, int inh) :
        pos(inpos), w(inw), h(inh)
    {}
    void SetLeftTopWidthHeight(int left, int top, int width, int height) {
        w = width;
        h = height;
        pos.x = left + w / 2;
        pos.y = top + h / 2;
    }
    void SetCenter(int x, int y) {
        pos.x = x;
        pos.y = y;
    }
    void SetCenter(const Position2& inpos) {
        pos.x = inpos.x;
        pos.y = inpos.y;
    }
    Vector2 Center() {
        return pos;
    }
    int Left()const { return pos.x - w / 2; }
    int Top()const { return pos.y - h / 2; }
    int Right()const { return pos.x + w / 2; }
    int Bottom()const { return pos.y + h / 2; }
    int Width()const { return w; }
    int Height()const { return h; }
```

```

    ///自分の矩形を描画する
    ///@param color 矩形の色(デフォルトは白)
    void Draw(unsigned int color=0xffffffff);

    ///自分の矩形を描画する(オフセット付き)
    ///@param offset 矩形座標のオフセット
    ///@param 矩形の色(デフォルトは白)
    void Draw(const Vector2& offset,unsigned int color=0xffffffff);
};

```

そしてバイナリを読むのが初めてな人もいるかもしれませんので、一応解説する

バイナリ読み込み

バイナリの読み込みは、クツクツクツクツクツノ簡単です。テキストの読み込みよりも簡単です。
ドシンプル!!!だしコストも少ないし、便利!!!

ひとまず

fopen と fread で説明しますね。今回はバイナリデータで、読み取り専用オープンなので rb
でオープンします。

次に必要なものを必要なサイズぶん fread します。最後に fclose すれば終わり。ごっつ簡単。
ひとまずは、画像ファイルのパスをロードしましょうか。

```
fopen("player.act","rb");
```

```
fread(&imgfilepath, sizeof(imgfilepathsize), fp);
```

で画像ファイルパスの文字列長を取ってこれます

最初の4バイトがパスの長さになります。可変長文字列なので STL の string を使用します。
#include<string>しといってください。

文字列長は分かっているので imgfilepath という string 型の変数を宣言したら、

このサイズを vector みたいに変更します。

```
imgfilepath.resize(imgfilepathsize);
```

あとは読み込みます。

```
fread(&imgfilepath(0),imgfilepath.size(),fp);
```

ひとまずこれで、画像ファイルのパスが得られることを確認しましょう。あとはこの使い方を繰り返すだけでいいのですが、とりあえずノーヒントでできますかね？
ちょっとやってみてください。

あと、ちょっと付け加えておくと、バイナリ読み込みの場合は読み込み用の構造体を作っておくとかなり楽にロードできるって思っておいてください。

ちょっとアクションのデータについて考えてみましょうか…。

今回はアクションはアクション名で区分けされているため、アクション名でデータへのアクセスができた方がいいと考えられます。つまり map を使うといいんじゃないかなと思います。

そしてアクションデータというのはカットデータの集合体と考えられます。更にデータの最大数が不定であるため、配列よりも、ベクタをつかうのが良いと思います。で、ベクタの中身の型は

僕はこう定義しているのですが

```
struct CutData{  
    Rect cutrect;//切り抜き矩形  
    Position2 center;//画像中心  
    int duration;//継続時間  
};
```

つまるところ

```
struct CutData{  
    int rectCX;  
    int rectCY;  
    int width;  
    int height;
```



```
        int centerx;  
        int centery;  
        int duration;  
};
```

の28バイトになっています。この場合一つのカットデータの読み込みはシンプルで

```
fread(&cutdata,sizeof(CutData),fp);
```

でいっぱい読み込むことができます。

となるとデータ構造的には

```
std::vector<CutData>
```

が1アクションデータを表している。そして、アクションが複数あるが、アクション名で区別されているため、mapにするという事は

```
std::map<std::string,std::vector<CutData>> actiondata;
```

と定義できます。

さて、ここまでヒントを言ったうえで、自前でちょっと考えて読み込んでみてください。

如何でしょうか？

ぼくはこんな感じでやりました。

```
int actionH=DxLib::FileRead_open(_T("player.act"));
```

```
int filenamesize = 0;
```

```
//画像ファイル名の読み込み
```

```
FileRead_read(&filenamesize, sizeof(filenamesize), actionH);
```

```

_actionData.imgfilename.resize(filenameesize);

FileRead_read(&_actionData.imgfilename[0], filenameesize, actionH);

int actionCount=0;
FileRead_read(&actionCount, sizeof(actionCount), actionH);
for (int i = 0; i < actionCount; ++i) {
    int actionnamesize = 0;
    FileRead_read(&actionnamesize, sizeof(actionnamesize), actionH);
    std::string actionname;
    actionname.resize(actionnamesize);
    FileRead_read(&actionname[0], actionnamesize, actionH);
    int animcount = 0;
    FileRead_read(&animcount, sizeof(animcount), actionH);
    std::vector<CutData> animcutinfoes(animcount);
    for (int i = 0; i < animcount; ++i) {
        FileRead_read(&animcutinfoes[i], sizeof(animcutinfoes[i]), actionH);
    }
    _actionData.animdata[actionname] = animcutinfoes;
}
FileRead_close(actionH);

```

ちなみに、open と read は DxDlib のやつを使っています。非同期読み込みに対応する事を想定しています。

相対パスをアプリの相対パスへ変換して画像読み込み

まあぶっちゃけ難しい。文字列の操作と言うのはだいたい難しい。意外と難しい。それでも string は比較的使いやすくしてくれています。

アクションデータというのは、画像の切り抜きのための情報が入っているデータなのですが、ここの中に、そのアクションを表示させるための画像のパスも入ってるんですよ。

```

magefilenameesize 0000000F
magefilename[0] 2E 2E 2F 69 6D 67 2F 72 69 63 6B 2E 70 6E 67  ../img/rick.png

```

それはいい。それは既に読み込めているはずなんだが、このパスを…こいつをよく見てくれ。こいつをどう思う？

すごく…アクションファイルからの相対パスです。

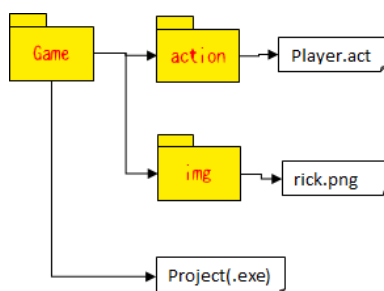
ここで問題に気付く。アクションファイルの中に格納されているパスは

『アクションファイルからの相対パス』であって

『アプリケーションからの相対パス』ではないのだ。

ということで、文字列を読み込んでそのまま LoadGraph してもファイルは見つかりません。表示されません。あたりまえよ。

例えば今回の場合、`../img/rick.png`となっているが、実際のフォルダ構成はこうだ。



で、デバッグ中は実行ファイルのワーキングディレクトリは、`vcxproj` が置いてあるのと同じ場所からパスを探索し始めます。これは覚えておいてください。3D の PMD ファイルとかでもテクスチャのファイルは PMD からの相対パスですので、まず確実に将来的に立ちふさがる問題だと思っておきましょう。

確かに `player.act` のおいてある `action` フォルダから見ると `../img/` ではあるが、実行作業ディレクトリであるプロジェクトが置いてあるのは一つ上の `Game` フォルダである。

これはどうしたらいいのだろうか？

まあつまり、`Player.act` と `rick.png` はセットで読み込まれるものとする、`Player.act` の相対パスと `../img/` を組み合わせればいい事が分かる。

ここで例えば

```
actPath="action/player.act";
```

```
imgPath="../img/rick.png";
```

だとする。

アクションファイルのパスは、作業ディレクトリ(プロジェクト)から見た相対パスで正しい。だが、`imgPath` は現状正しくない。

…となると `actPath` の中から `action/` までを抜き出せばいい事が分かる。

ここで使用する string 系の関数は2つだ。

find_last_of: 特定の文字列を後ろから検索し、最初にヒットしたインデックスを返す

substr: 文字列の中から文字列の一部を抜き出して返す。

なお、リファレンスは

https://cppref.jp.github.io/reference/string/basic_string/find_last_of.html

https://cppref.jp.github.io/reference/string/basic_string/substr.html

である。読みやすくないドキュメントだが、関数のリファレンスはいちいち、しつこいくらい読む癖をつけておこう。それが君を助ける時もあるだろう。

しばらく知恵を絞って、ここから相対パスを導き出すプログラムを考えてみよう。

算数の問題を解くような、パズルを解くような気持ちで…実際数学の授業をしてるのは、こういう時の問題解決の糸口をどう見つけ出すか鍛えるための物なのだ。

さて、思いついたかな？答えはこうだ。

```
imgPath = actPath.substr(0, actPath.find_last_of('/')+1)+imgPath;
```

である。

なぜこうなるのか…おわかりいただけるだろうか？

まず `actPath.find_last_of('/')` が、最後の/の場所インデックスを返す。今回の例であれば `b` が返っているはず(0 オリジンで7 番目にあるため)。

次に `substr` だが、これは第一引数が開始インデックスで、第二引数が文字列の数である。ちなみに+1 しているのは、戻り値の `b` を指定してしまうと '/' の手前からとなってしまう(つまり "action" しか戻らない)ためである。

それに元の `imgPath` を連結したものを `imgPath` に代入してやれば欲しい画像のパスが得られるというわけです。

それではプレイヤーの画像を表示させてください。うまくいかない人は、遠慮なく質問してください。

読み込んだデータをもとに切り抜く

とりあえず先頭のアクションの先頭の1コマだけ切り抜いて表示させてみましょうか…。



既に矩形データを読み込んでおりますので、そのデータと
`DrawRectRotaGraph2`(表示 X 座標, 表示 Y 座標,

切り抜き左, 切り抜き上,
切り抜き幅, 切り抜き高,
中心点 X, 中心点 Y,
拡大率, 回転角度(ラジアン),
ハンドル, 透明にするかフラグ,
左右反転するかフラグ);

関数があればなんとかなるでしょう。ひとまず自力でやってみてください。

中心点は今の所どこでもいいです。画像の中心あたりに設定しとけばいいです。そのうち変更する必要が出てきますけど。

表示できましたか？

では、アニメーションさせましょう。

読み込んだデータをもとにアニメーションさせる

データの中に `Walk` というアニメーションデータがありますので、まずはそれを再生させましょう。

もしこの段階まで `main.cpp` 一本でコーディングしてる人はそろそろしんどいので `Player.cpp` に移して作業してください。

アニメーションさせるという事は、メンバ変数として

- 現在再生中のアニメーション名
- 現在再生中のコマ(インデックス)
- 経過フレーム数

が必要になります。

それは分かりますね？こういう『何が必要な？』というのは自分で考えられるようになりましょう。で、いっぺんに考えようとすると脳がストップしちゃいますので、一つずつ潰していきましょう。

今回はひとつのアクションアニメーションを再生させるという事だけ考えます。

とにかく↑の情報を保持しつつ更新していけばいいわけです。

つまり

```
ActionData _actionData; //アクションデータのマップ
std::string _currentActionName; //現在再生中のアクション名
int _currentCutIdx; //現在表示中のカットインデックス
unsigned int _frame; //経過フレーム
```

こういったものを Player のメンバとして用意しておきます。

で、Player の Update 関数(毎フレーム呼ばれる)かなんかで、_frame がインクリメントするようにプログラミングしておきましょう。

そのうえで毎フレーム

1. _frame をインクリメント
2. duration と _frame を比較し、duration を越えたら
 - ① 次のカットに変更
 - ② _frame をリセット

とします。さらにインデックスが最終インデックスの時は動きを場合分けします

- 「ループ」すべき場合は最初のカットに戻る
- 最後のカットを維持すべき場合は飽和加算状態にしておく

- ボタンを押されていない場合はニュートラルもしくは歩きもしくはしゃがみニュートラルに移行する

のどれかになります。

今回は歩きなので『ループ』と考えてやってみましょう。無事歩いているように見えれば正解です。正解なのですが…

ガツタガタやる？

为什么呢？というと、切り抜きが等幅ではないため、合わせるために『中心点』が必要なんですよね。切り抜きの中心ではないんですよね。キャラの重心が常に画像の中心というわけでもありませんし…。という事で、中心点を適用するコードを書きましょう。

中心点適用

で、それをガツタガタにならないように目で見ても中心点を職人が設定してあげる必要があります。で、既に中心点情報をデータに入れているので、それを使いましょう。

※矩形の『中心点』ではない事に注意してくださいね？

既に読み込んで centerx,centery もしくは center.x,center.y に入ってるので、それを使います。なお、中心点を設定するために

DrawRectRotaGraph2 を使用します。仕様はこれまた DxLib の HP に登録されておりましたが、DrawRectRotaGraph2(表示 X 座標,表示 Y 座標,

切り抜き左,切り抜き上,
切り抜き幅,切り抜き高,
中心点 X, 中心点 Y,
拡大率, 回転角度(ラジアン),
ハンドル, 透明にするかフラグ,
左右反転するかフラグ);

になっておりますので、受け取った中心点を 中心点 X,中心点 Y に設定してください。

きれいにアニメーションすると思います。

パッド(キーボード入力)への対応

Peripheral クラスの実装に入りましょうか。Peripheral クラスの要件を定義します(本当はここも自分で考えて定義してほしい…最終的には全て自分でやる。自分だけでやるのだ…自分でできない奴が雇われるわけなかる?)

要件

- キーボードとパッド双方の入力に対応する
- 現在の押下状態の検出ができる
- トリガー(この瞬間に押したばかり)の検出ができる

実装

となるとクラス定義はこんな感じになると思います。

///周辺機器の入力情報を管理

```
class Peripheral
```

```
{
```

```
private:
```

```
    int _padState; //現在の入力状態
```

```
    int _lastPadState; //直前フレームの入力状態
```

```
public:
```

```
    Peripheral();
```

```
    ~Peripheral();
```

```
    ///入力情報の更新(毎フレームコールしてください)
```

```
    void Update();
```

```
    ///現在の押下状態の検出
```

```
    ///@param keyid 調べたいキー番号
```

```
    ///@retval true 押してる
```

```
    ///@retval false 押してない
```

```
    bool IsPressing(int keyid) const;
```

```
    ///現在のトリガー状態(押した瞬間)の検出
```

```
    ///@param keyid 調べたいキー番号
```



```

    ///@retval true 押してる
    ///@retval false 押してない
    bool IsTrigger(int keyid)const;
};

```

とりあえず、

```

DxLib::GetJoypadInputState(DX_INPUT_KEY_PAD1);
http://dxlib.o.oo7.jp/function/dxfunc\_input.html#R5N4

```

を使って実装してください。

んで、この Peripheral をメインループ内で更新されるようにして、そのオブジェクトへの参照を各自の Update 関数に渡せるようにしておいてください。つまり…

```

Peripheral peripheral;
while (!DxLib::ProcessMessage()) {
    (中略)
    peripheral.Update();//入力情報の更新
    (中略)
    _player->Update(peripheral);//プレイヤーの更新
    (中略)
}

```

として、プレイヤーを左右に動かせるようにしてください。

テスト

では、キーボードまたはパッドの左右を押したら左右に動くようにしてください。これに関し
てはほぼ説明が要らないですよ？

左で左に2ピクセルくらい、右で右に2ピクセル動くようにしてください。

また、左右のキーだけパッドだけで左右の向きを切り替えられるようにしてください。左を押したら左向き(ターン ON)に、右を押したら右向き(ターン OFF)にするようにしてください。

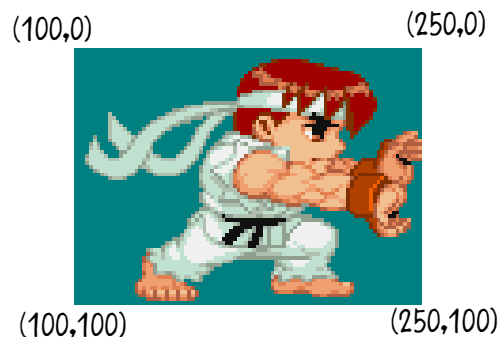
さて、何かに気が付きませんか？そう…ガタガタしてますよね？

左向きでガタガタしないようにしよう

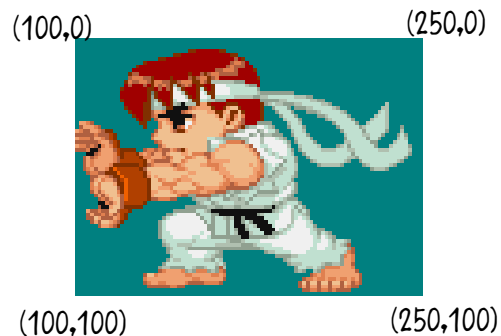
そもそもガタガタするのは何故？

これはですね、DrawRectRotaGraph2 の仕様のせいなんですよね。ターンフラグを ON にするとき第7、第8引数に中心点を設定するから、その中心点を線対称として、反転処理が行われるような気がしますがそうじゃないんですよね。これはあくまでも『回転』の中心点です。

ですから、反転した時に保持されるのは中心点ではなく矩形の4隅の座標なんですよね
例えば



という配置になっているなら反転したところで

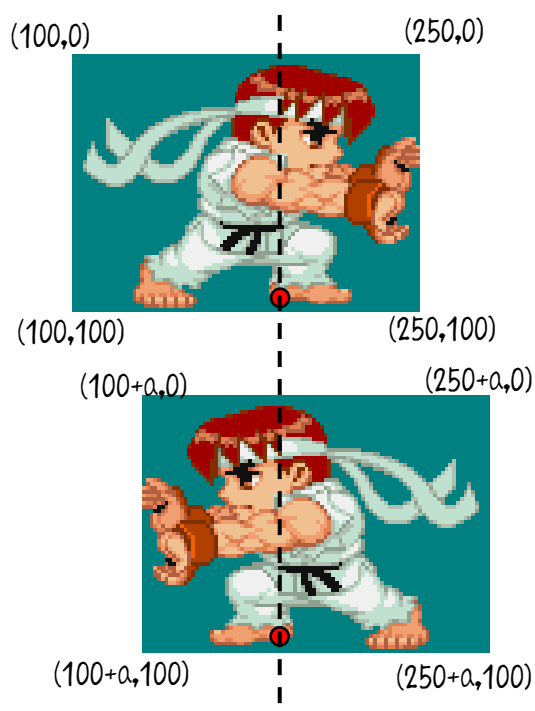


4 隅の表示座標は変わらない。つまり画像の中心に中心点がないような場合はズレズレになる。画像が等幅なら、中心点が中心にあってもいいんだが、等幅じゃない時はそうはいかない。

ガタガタへの対処

等幅であってもポーズによってはそうはいかないため、反転時の処理を特別に考えなきゃならない。

例えば、前足のかかとが中心になっているとすると以下のようにしてほしい。



いったいどうすればいいのでしょうか？

画像の反転自体は、画像の中心(ちょうど真ん中…上の例で言うと 100 と 250 の間… $\frac{100+250}{2}$) になっています。

そのうえで、第一引数および第二引数の x,y の座標に、第 7, 第 8 の x,y 座標を合わせるように配置されます。ああ、それだと一見大丈夫そうに思えてしまいますね。

順番は

① 1,2 引数の x,y 座標に 7,8 の中心座標を合わせる

② 画像の中心を中心に反転

となっていると考えたほうが分かりやすいですかね？

そうなるとう然ずれてしまいます。こういう時にどう計算すればいいのかというと、あるべき中心がそれだけずれる。左端からの座標が反転時は右端からの座標と一致すればいいという事になります。

つまり

- 非反転時はデータの中心 x 座標を中心点 x 座標とする
 - 反転時はデータの中心 x 座標を幅-中心点 x 座標とする
- としてみてください。

左を向いた時に自然に反転し、反転時にもガタつがなくなったならば、それが正解です。喜んでおきましょう。

ぼくはこんな感じにしています。

```
int centerx = _isTurn ? cutrect.Width() - info.center.x : info.center.x;
```

を中心Xとして採用しています。

ガタガタしなくなりましたか？しなくなったら次に行きましょう。

ジャンプとか、基本動作を試みよう

地面を定義

ひとまず 340ピクセルくらいの所にしてみましょうか…。

この位置に DrawLine で線を引いてみて、適切な座標かどうか確認してみましょう。



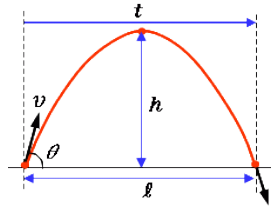
一応、これ、クラス化します。Y座標 340しか持ってないのでクラス化してしまうのもアレなんですけど、実は斜め床とか考えておりますので、そのへんの制御まで考えると一応クラス化しておいた方がいいかなと思います。

まあ、平坦床だったとしてもたいして邪魔にはならんでしょ…

ジャンプと重力

ジャンプの本格実装と言うよりむしろ放物線的な動きを実装してみましょう。で、放物線の動きってどういう事ですかね？

高校物理的に書くところだね？



速度はこう…

$$v_x = v \cos \theta$$

$$v_y = v \sin \theta - gt$$

座標はこう…

$$x = v \cos \theta t$$

$$y = v \sin \theta t - \frac{gt^2}{2}$$

なんです が !!それほど難しく考えなくてもいい。↑の式も簡単ですが、プログラムはもっと簡単です。座標と速度の変数を作っておきます。

- pos:(position の略)⇒座標
- vel:(velocity の略)⇒速度

で、Player が毎フレーム実行する関数として Update()があるとすると…

```
Update(){  
    pos+=vel;//座標が速度によって微小変化(dx)  
    vel.y+=g;//速度が重力加速度によって微小変化(dx^2)  
}
```

で、この辺、数学(微積分)とか物理とかお得意な人なら『運動方程式』を思い出すような作りになっていると思います。

事実、

$$\int f(x) dx$$

的なものを普通『積分』と言いますが、もっと細かく言うとゲームにおける積分ってのは、これ

だけではなく

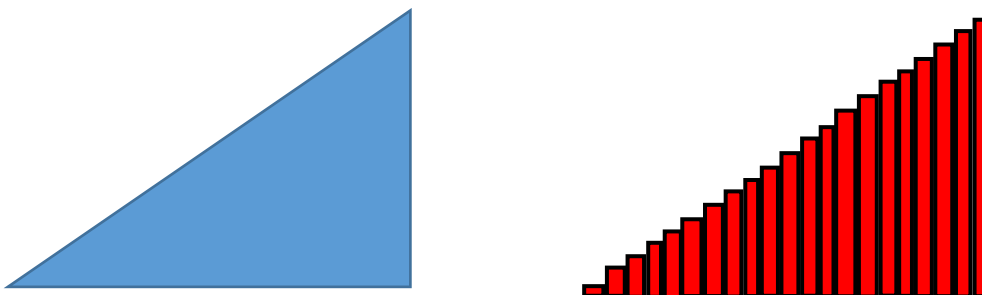
$$d_{x+h} = d_x + (f(x+h) - f(x))$$

こういうを示しています。この場合、 $f(x+h) - f(x)$ は微小変化を示しています。例えば座標の微小変化はそれすなわち瞬間速度という事になります。

なので↑の数式が

```
pos+=vel;//座標が速度によって微小変化(dx)
```

にあたると思ってください。つまり変化の間が十分に小さければ積分と同じと見なしてよいです。ただし、厳密に言うと



階段の部分で誤差が蓄積はしていくので、ガチ物理と合わせ始めるとおかしいことになってきます。今しばらくは関係ないので、足し算オンリーでの物理で構いません。

ただし、

```
pos+=vel;//座標が速度によって微小変化(dx)
```

```
vel.y+=g;//速度が重力加速度によって微小変化(dx^2)
```

のコードはただただ自由落下を表している部分に過ぎないので、ジャンプの事を考えてあげる必要があります。ただし、ジャンプの瞬間と言うのは簡単で、強制的に速度を上向きに与えてあげればいいわけです。例えば Jump という関数があるとして

```
Jump(){  
    vel.y=jump_power;  
}
```

とでもやればいいでしょう。あと、プログラマ的な事を言うと、地上と空中ではステートマシンとか、挙動が違いますので、空中フラグとか作っておきましょう。

着地

ジャンプと重力を実装しただけでは、一旦ジャンプすればドゥンドゥン落ちていきますので、着地のイベントを作りましょう。先に定義した地面クラスを利用します。

Ground クラスはひとまずとにかく地面の高さを返します。もっと言うと、現在のプレイヤーがいる場所の地面の高さを返します。

プレイヤーがいる座標をもとに地面の高さを得る予定なので、Ground はプレイヤーへの参照を持っているとします。

つまり

```
///地面を表すクラス
///地面と言うか、プレイヤーの現在座標に対応した
///着地点をコントロールする
class Ground
{
    Player& _player;
public:
    Ground(Player& player);
    ~Ground();

    ///現在のプレイヤー位置での
    ///『あるべき地面』の高さを返す。
    int GetCurrentGroundY()const;
};
```

さて、ここで注目すべきは、メンバ変数の Player への参照ですね。そのまま使用しようとする
ととにかくコンパイラから怒られますけど、そこは以前にも話した初期化リストを使用して
回避してください。

また、Player の型がわからんという事で怒られますがそれへの対処は…include しちゃだめだ
ぞ!!

まあそれはそれとして、当たり判定的な部分は後からまとめるとして、今は Game.cpp に直で
書いていいので、自分の座標(足元)と地面の線を比較して、地面の線より下に来たら戻すとい
うようなことをやってください。なお中心点の定義がそもそも



足元に定義しています

これはこのような場合に備えての定義だったのです。一応売り物の某格闘ゲームでもこのように定義してました。これだと楽なんです。ちなみに矩形の中心点とは異なる概念ですので、くれぐれも間違えないようにしてください。

ああ、もしかしたら、Player の中心点の要素に直接アクセスしたいときがあるかもしれません。本来はダメなのですが、特別に Game.cpp からだけアクセスできるという方法を教えましょう。

friend

通常のカプセル化のやり方から言うと、public と private と protected でアクセスの権限はそれ以外にないと思ってる事でしょう。まあ教えてないですしおすし。

ところがまだアクセス系のキーワードはあるんです。それが friend(ともだち)です。



キーワードの名前がちょっと分かりづらいんですが、friend(友達)です。こいつはアクセス指定に使うキーワードなんですけど、private や public などとはちょっと使い方が異なります。

ゆーたら、他のクラスに対して『友達認定』するキーワードです。

で、その友達認定したクラスに対しては全てを開示するという恐るべきキーワードなのです。だからよっぽどな時にのみ使います。一時的な間に合わせて使用する場合は必ず friend を使用したことを覚えておいて、リファクタリング時にでも friend 認定解除してください。大丈夫

です。『絶交』なんてされませんから…

というわけで、ひとまずは Player 側は Game クラスに対して『ともだち認定』
しましょうか…。

やり方はいたって簡単。クラスの定義部分に friend Game; と書くだけ。

///プレイヤークラス

```
class Player
```

```
{
```

```
    friend Game;
```

```
private:
```

```
    :
```



はい、これで Game クラスは Player クラスのおともだちです。



しかし対等なお友達ではない。Game クラスは Player の内臓にまで手を突っ込むことはできます。しかし Player クラス側は Game 側に対しては通常のアクセスしかできません。



これは友達と言えるのか…

…つまりそういうことです。まあクラスの話ですし。

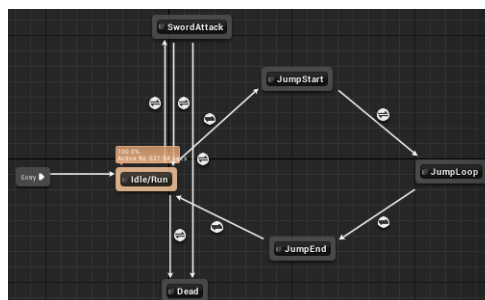
こいつを使って、規定ライン以下に行かないようにしてください。

アニメーション切り替え

歩きとジャンプができたのでアニメーションを切り替えられるようにしてみましょう。その前に、プレイヤーのアクションにおいては『状態遷移』の考え方が重要になってきます。

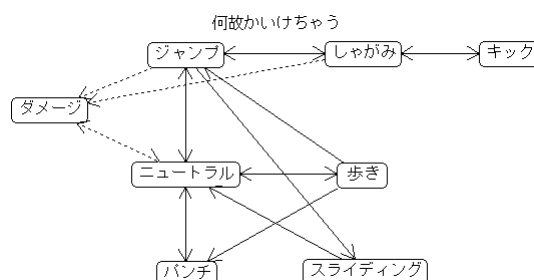
状態遷移

状態遷移というのは、エイゴで言うとステートマシンと言って、UnrealEngine4 やら Unity にもあるアレです。



こういうやつです

例えば、今回のプレイヤーの挙動で言うと



こういう感じです

で、UnrealEngine4 でやる時もそうですが、状態が変わるときのトリガー(きっかけ)が必要です。トリガーには以下のものがあります。

- 時間経過(パンチ、キックの動作が終わった後など)
- ボタン入力(パンチ、キック)
- レバー入力/解除(歩き、しやがみ)
- 着地/ヒット(ジャンプ後の着地、ダメージ食らった時など)

メンバ関数ポインタによる状態遷移

で、今回は以前にもちよっと話しましたが、こういう状態遷移を『メンバ関数ポインタ』を利用していいかなと思っています。

ひとまず現在の Player::Update 関数がこのように定義されておりますので…

```
//プレイヤーの状態を更新する
//@param peripheral 入力情報
void Update(Peripheral& peripheral);
```

これと同じような関数を、ニュートラル、歩き、ジャンプ、パンチ、キック、スライディング、しゃがみ、ダメージぶん作ってみましょう。

ただしそれらの関数は private にしておいてください。外側からは呼び出し禁止です。

あと、アップデート(メンバ関数ポインタ)も追加してくださいつまり…

```
//アップデート
void (Player::*_updateFunc)(Peripheral&);
```

```
//ニュートラル状態
void NeutralUpdate(Peripheral&);
```

```
//歩き
void WalkUpdate(Peripheral&);
```

```
//ジャンプ
void JumpUpdate(Peripheral&);
```

```
//しゃがみ
void CrouchUpdate(Peripheral&);
```

```
//パンチ
void PunchUpdate(Peripheral&);
```

```
//キック
void KickUpdate(Peripheral&);
```

```
//ダメージ
void DamageUpdate(Peripheral&);
```

ひとまずは、ニュートラル、歩き、ジャンプを実装しましょう。

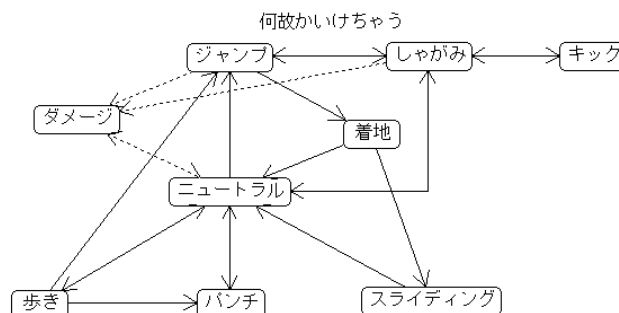
最初はニュートラルなので、
_update=&Player::NeutralUpdate;

で、通常の Update 関数の中では
(this->*_update)(peripheral);
だけ呼び出しているような状態にしてください。基本的に歩いていない(左右キーが押されてない状況)時には歩かない(アクションは Walk だけど、カウンタを進ませない)としておき、左右キーが押されている間はアップデートを WalkUpdate にします。

面倒だけど、ここを一旦作っておくと楽になりますねー。

状態遷移をチョット修正

で、色々やってみた結果こうなりました。



//アップデート

```
void (Player::*_updateFunc)(Peripheral&);
```

//ニュートラル状態

```
void NeutralUpdate(Peripheral&);
```

//歩き

```
void WalkUpdate(Peripheral&);
```

//ジャンプ

```
void JumpUpdate(Peripheral&);
```

//着地

```
void GroundUpdate(Peripheral& p);
```

```
//しゃがみ
```

```
void CrouchUpdate(Peripheral&);
```

```
//パンチ
```

```
void PunchUpdate(Peripheral&);
```

```
//キック
```

```
void KickUpdate(Peripheral&);
```

```
//スライディング
```

```
void SlidingUpdate(Peripheral&);
```

```
//ダメージ
```

```
void DamageUpdate(Peripheral&);
```

```
//ジャンプ
```

```
void Jump();
```

```
//パンチ
```

```
void Punch();
```

```
//キック
```

```
void Kick();
```

```
//しゃがみ
```

```
void Crouch();
```

みたいになりました。

で、例えば NeutralUpdate の中で

```
void
```

```
Player::NeutralUpdate(Peripheral& p ) {
```

```
    if (p.IsPressing(PAD_INPUT_RIGHT)) {
```

```
        _isTurn = false;
```

```

        _pos.x += 2;
        _updateFunc = &Player::WalkUpdate;
    }else if (p.IsPressing(PAD_INPUT_LEFT)) {
        _isTurn = true;
        _pos.x -= 2;
        _updateFunc = &Player::WalkUpdate;
    }
    if (p.IsPressing(PAD_INPUT_DOWN)) {
        Crouch();
    }
    if (p.IsTrigger(PAD_INPUT_1)) {
        Jump();
    }
    if (p.IsTrigger(PAD_INPUT_2)) {
        Punch();
    }
}

```

こんな感じで切り替えます。ちなみに Jump 関数などは、この関数の中で状態を JumpUpdate に切り替えています。ちなみに Jump 関数の中身はこんな感じです。

```

void
Player::Jump() {
    if (!_isAerial) {
        _vel.y = -10.f; // 上向き速度を追加
        _isAerial = true; // 空中フラグON
        _updateFunc = &Player::JumpUpdate; // 状態をジャンプに変更
        ChangeAction("Jump"); // 再生アクションをジャンプに変更
    }
}

```

とりあえず、最初から割とそんなにきれいには書けません。あまり悩まずに、でも状態遷移を構成できてるようなコードにしてください。

リファクタリング①

さて、ここまで書けたら、小休止？小休止じゃない？そこは人によって違うと思いますが、リファクタリングと言うものを行います。

人によっては小休止に感じるかもしれませんが、勢いでプログラミングやってる時より面倒に思えるかもしれません。

ひとまずはクソコードにならないようにクソコードの定義から書いていきます

クソコードとは

「クソコード」とは・・・

読む人を怒りの渦に 叩きこむコードである

クソアニメならまだ、楽しみようもあるものだが、クソコードはただただ害悪である。

ただし、どんなに気を付けていても、クソコードと言うのは混じるものである。



他人を怒らせるコードは当然就職活動などには使えませんね？さて、クソコードの例を挙げていきましょう。

一般的なクソコーディング

- クソ非効率コード…ジュラル星人なみの回りくどいコードを書いている(可読性を損なわないレベルで可能な限り短くせよ)
- クソコピペコード乱立(頭が悪い証拠です…関数化するなり工夫して回避しましょう)
- クソマジックナンバー(意味が分からない数値を使用しない。コメント書けばいいだろうみたいな発想はやめよう)
- クソネスト(やたらと for 文やら if 文やらをネストにしてかなり右側に寄ってしまっている。内側の処理は関数化するなりしてネストを減らそう…そもそものネストは必要なのか？こうならないコツはタブを 8 にしておくといい。ネストしなくなるから)

- クソネーミング(04 などの意味が分からないネーミング。変数名や関数名や型名がクソ短すぎて意味が分からないとか、英語で書くべきところをローマ字にしたりとか、英語のスペリングを間違えてたりとか…クソッ恥ずかしいんでやめてください…今、自分のコード見直してみて、不安なところがあったら、辞書とか Web で調べてスペリング確認して直ちに修正してください。後半戦でクソスペリングしてたら罵倒します)
- クソ include(ヘッダの include は極力減らしましょう。プロトタイプ宣言、前方参照などで減らせるはずですよ。くれぐれも all in みたいなクソヘッダを作らないようにしましょう)
- クソインデント(インデントは tab でやるべきであってスペースでやるべきではない)
- クソグローバル(グローバル変数は基本的に 0 が望ましいです)
- 警告クソ無視(警告は意味があるから出ているんです。まあ一部の警告は無視していいですけども警告が出たら基本的には対処してください)
- クソ省略(if 文とか for 文の {} 省略はやめましょう)
- クソフルパス指定(おまえそれ客先で動くと思ってんの?)
- クソロード(while の中で完了復帰ロード。同じの何度もロードしてる)
- クソ呼び出し(えっ…? その処理…main 関数に入る前にコールされてるんですけど…)
- クソでかファイル(1 ファイルあたり三千行…だと?)

関数編

- 1つの関数がクソ長い
- 関数のパラメータがクソ多い
- 呼ばれない関数がある
- 内部でパラメータの更新してないのにパラメータに const ついてない
- 一つの関数に2つ3つの役割を持たせている

クラス/構造体編

- 名前が適当
- クラスがでかすぎる
- 考えなしの継承
- 結合度の高すぎるクラス関係



まあゴチャゴチャ書きましたけれども、なんか思い当たる場所あれば修正するように心がけましょう。

リファクタリングとは

リファクタリングとは Martin Fowler 氏の提唱する用語で、

「プログラムの外部から見た動作を変えずにソースコードの内部構造を整理するこ

と」である。

言ってる意味が良く分からないかもしれないけど、とにかく

「アプリの挙動を変えずにクソコード、もしくはクソコードの育つ土壌を排除する」

って事である。

「アプリの挙動を変えずに」ってのがポイントで、例えばリファクタリングの鉄則として

「リファクタリングの際には機能追加や挙動の修正を行ってはならない」

というものもあります。これはいっぺんに違う事をやると「ぜったいバグる」という経験則があるんですね。

リファクタリングという言葉が定着するまでは

「一度正常な動作をしたプログラムは二度と手を触れるべきではないと言われていた。なぜなら、下手に手を加えて動作が変わってしまうと、それに伴って関連する部分にも修正が加えられ、やがてその修正作業はプロジェクト全体に波及し対処しきれない」

と言われていた。僕がプログラムの仕事に就いた 20 年前は本当にそういわれていた。「寝た子を起こすな」だのなんだの言われました。とにかく「動いてるんだから触るなっ…!」とね。

ところが、自動テスト(UnitTest)などのテスト手法が普及したことにより、やり方によっては後から修正してもプログラムはグッと良くなるという考えが定着してきました。これがリファクタリングです。

現状のコードをリファクタリング

さて、ここまでの時点でキャラクターの基本動作は終わっている事と思います。手が遅い人でも歩きと、停止状態くらいはできていると思います。できない人は言ってください。罵倒しながら丁寧に教えます。

Player クラスのコピペコードをどうにかする

多分、ここまで記述していると、コピペコードというか、各ステート関数内に似たようなコードが散見されるようになっているんじゃないかと思います。

とにかくそれをまとめて関数化してみたりしましょう。例えばアクションの切り替え部分などはほぼ同じような事を書いていると思いますので、ChangeAction 的な関数を作って、

```
Player::ChangeAction(const char* name) {  
    _frame = 0;  
    _currentActionName = name;  
    _currentCutIdx = 0;
```

ステート変更時にはこれを呼び出して変更するようにしてください。あと、再生の部分も結局

- フレームを進める
- フレームが duration に到達したらカットインデックスを進める

の部分は共通していると思いますので、この部分を共通化しましょう。なお、ループアニメーションかどうかはデータとして持っておりますので、関数内にて処理を切り替えましょう。

例として関数名を ProceedAnimationFrame としてみましょうか。

で、戻り値として、アニメーションが終わったら true を返すとしします。なおループアニメーションの場合は常に false を返します。

例えば、そこまでやると、パンチとかキックの Update はこのようになります。

```
//パンチ
```

```
void
```

```
Player::PunchUpdate(Peripheral&) {  
    if (ProceedAnimationFrame()) {  
        _updateFunc = &Player::NeutralUpdate;  
        ChangeAction("Walk");  
    }  
}
```

```
//キック
```

```
void
```

```
Player::KickUpdate(Peripheral&) {  
    if (ProceedAnimationFrame()) {  
        _updateFunc = &Player::CrouchUpdate;  
        ChangeAction("Crouch");  
    }  
}
```

ずいぶんすっきり書けます。ループアニメーションを持つ Walk なんかも…

```

//歩き
void
Player::WalkUpdate(Peripheral& p) {
    if (p.IsPressing(PAD_INPUT_RIGHT)) {
        _isTurn = false;
        _pos.x += 2;
    } else if (p.IsPressing(PAD_INPUT_LEFT)) {
        _isTurn = true;
        _pos.x -= 2;
    }
    else {
        _updateFunc = &Player::NeutralUpdate;
    }
    ProceedAnimationFrame();
    if (p.IsTrigger(PAD_INPUT_1)) {
        Jump();
    }
    if (p.IsTrigger(PAD_INPUT_2)) {
        Punch();
    }
}

```

となります。

main->Game クラス

もしかしたら現状 main があるファイルの中で作業をしているかもしれませんが。その人たちはひとまず main.cpp に書いてある処理を Game クラスに移動させましょう。

この手の大元になるクラスは基本的にシングルトンで作ります。ですから

///アプリ全体をコントロールするクラス

```

class Game
{
private:
    std::shared_ptr<Player> _player; //これもそのうちゲームシーンに移行する
    Game();
    //コピー、代入を禁止する

```

```

Game(const Game&);
Game& operator=(const Game&)const;
public:
    static Game& Instance() {
        static Game instance;
        return instance;
    }
    ~Game();
    void Init();
    void Loop();
};

```

で、Initに初期化処理。Loopの所にループを放り込んでください。

Gameクラスの背景系制御→Backgroundクラス

背景系をBackgroundクラスに移行します。背景はUpdateとDrawを持っています。GameのLoopの中でそれが呼ばれるだけのイメージです。

```

class Background
{
public:
    Background();
    ~Background();

    //背景情報更新
    void Update();

    //背景描画
    void Draw();
};

```

あと、くれぐれも言っておきますが、リファクタリングする際は、**1個のリファクタリングが終わったら、そのつど実行して挙動が変わってないか確認**するようにしてください。だーっといくつものリファクタリングをして、その後実行テストなんてやめてください。ほぼ把握できませんから。

インターフェイス系を HUD クラスに

とりあえずライフバー系を HUD クラスに置きましょう。今の所…すぐですが、一応 Update 関数と Draw 関数を用意しておきましょう。また、内部にスコアおよびプレイヤーのステータスを持っているため、プレイヤーへの参照とスコア変数を持たせておきましょう。

//画面上に表示される「UI」

```
class HUD
{
private:
    int barTopH;//上バーのUI画像ハンドル
    int barBottomH;//下バーのUI画像ハンドル
    const Player& _player;//プレイヤーへの参照
    int _score;//スコア
public:
    HUD(const Player& player);
    ~HUD();

    ///更新
    void Update();

    ///描画
    void Draw();
};
```

ひとまずそこまでやれば…

とりあえず main 関数はこのくらいシンプルになります。

```
int main() {
    Game& game = Game::Instance();
    game.Init();
    game.Loop();
}
```

ちなみに GameMain のループ関数の中のループは

```
while (!DxLib::ProcessMessage()) {
    DxLib::ClearDrawScreen();
```

```

        peripheral.Update();
        if (peripheral.IsPressing(PAD_INPUT_9)) {
            break;
        }

        _player->Update(peripheral);

        //足場のチェック
        CheckScaffold(*_player, ground);

        //描画系命令
        bg.Draw();
        _player->Draw();
        foreground.Draw();
        hud.Draw();
        DrawLine(0, 340, 768, 340, 0xffffaa, 2);

        //シーン管理系
        _scenemanager.SceneUpdate(peripheral);

        ScreenFlip();
    }

```

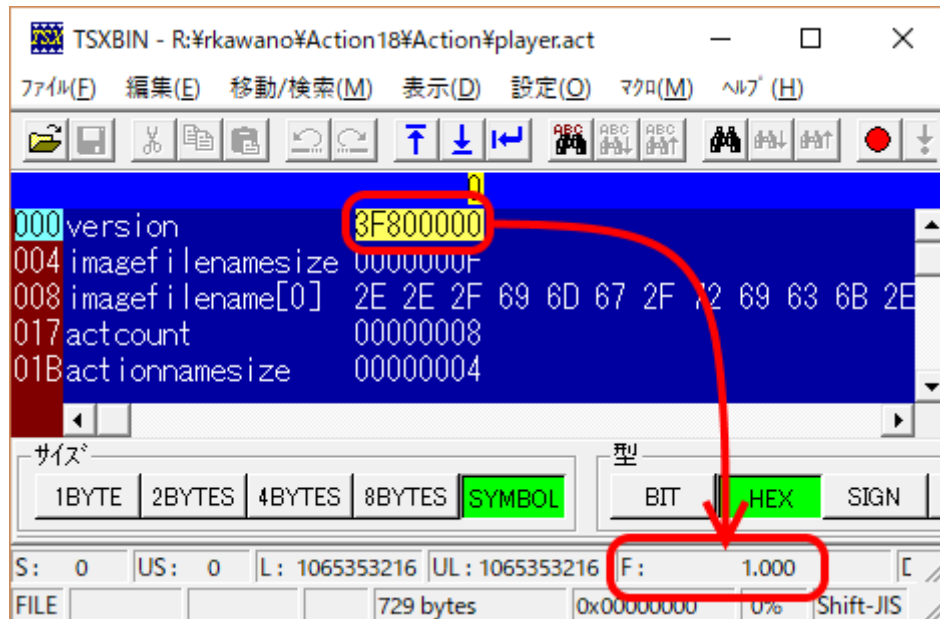
こんな感じになり、だいぶすっきりしてきたんじゃないでしょうか？

ちょっとここぞツール変更(重要！！)

すみません。重要な事なので、読み落とさないでください。

アクション情報をちょっと変更しました。今後も頻繁に変更が入りますが、その事を見越して、それに対応するような形にしようと思います。

どういう事かと言うと、データの先頭4バイトに『バージョン』を入れました。そしてこれはFloat型です。なのでバイナリエディタで見ると



このように 3F800000 というちょっと言ってる意味が良く分からないデータとなっておりますが、先ほども言いましたようにFloatです。それが分かっている場合はステータスバーのFの項目を見てください。1.000 となっているでしょう？
これ、MMDのファイルでも同じ事をやっているのですが、FLOATをバイナリ書き出しするところなのです。

そして、実機側はどう対応するのかと言うと…。
読み込みの先頭にこれを追加します。

//バージョンの読み込み

```
float version = 0.0f;
```

```
FileRead_read(&version, sizeof(version), actionH);
```

ただ、このままでは大して意味がないため、今回はassertを使用します。assertはご存知だと思いますが、途中であえてクラッシュさせるためのものです。

使用するにはcassertをインクルードします。

```
#include<cassert>
```

んでバージョン読み込み後に例えばこう書きます。

```
assert(version == 1.0f && "アクションファイルのバージョンが一致しません");
```

これはバージョンが一致していれば何も問題ないのですが、バージョンが一致しないと問題が発生します。

ちなみに右側のはつけてもつけなくてもあまり変わりありません。ちなみに assert の仕様ですが

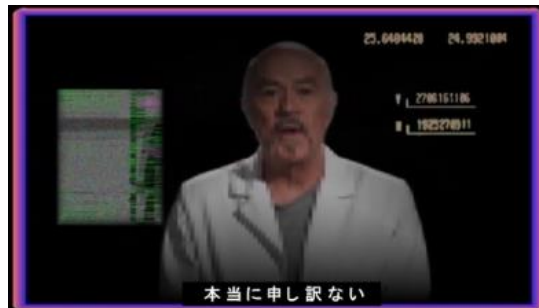
引数が false であった時にクラッシュする

です。↑のコードは && でくくっているため左側の評価式はバージョンによって変化しますが、右側は単なる文字列です、これが null 文字でない限りは true 扱いなので、基本的な文法的には「評価」に影響しません。コメントみたいなもんです(標準出力には出力されます)

そしてゴールデンウィークに突入…

5/3 更新(攻撃矩形追加)

舌の根も乾かぬうちに、バージョンアップ…1.01 になりました。本当に申し訳ない。



お休み中も休まず調子に乗っております。

変更点は攻撃矩形の追加。つまりカット一つ一つに矩形情報が乗っていることになる。

ちなみに以下がアクションデータのバイナリシンボルファイルの中身だ。

```
#include "typedef.h"
```

```
const DWORD      FILESIZE = GetFileSize();
```

```
$float version;//4バイト
```

```
$DWORD imagefilenamesize;//画像名サイズ(4Byte)
```

```
$BYTE imagefilename(imagefilenamesize);//アクション名(?Byte)
```



```

$DWORD actcount; // アクション数(4Byte)

for(int i=0; i < actcount; i++){
    $DWORD actionnamesize; // アクション名サイズ(4Byte)
    $BYTE actionname(actionnamesize); // アクション名(?Byte)
    $BYTE loop;
    $DWORD count; // 切り抜き情報カウンタ
    for(int j=0; j<count; ++j){
        $DWORD rectCX; // 切り抜き矩形中心 X
        $DWORD rectCY; // 切り抜き矩形中心 Y
        $DWORD width; // 切り抜き矩形幅
        $DWORD height; // 切り抜き矩形高さ
        $DWORD centerx; // 中心点 X
        $DWORD centery; // 中心点 Y
        $DWORD duration; // フレーム数
        $DWORD actrcnt; // 攻撃矩形などの数
        // 矩形情報
        for(int k=0; k<actrcnt; ++k){
            $DWORD ActRcType; // 矩形種別
            $DWORD ActRcCX; // 矩形中心オフセット X
            $DWORD ActRcCY; // 矩形中心オフセット Y
            $DWORD ActRcW; // 矩形幅
            $DWORD ActRcH; // 矩形高さ
        }
    }
}

```

正直なところ、ここだけ見て、あとは何をすればいいのかわかりませんが、自分で考えられるようになってほしい。3年ならもうそうになっていると思いますが、2年生だとまだまだ厳しいかなあ。

ということで、このシンボルを元に新しいファイルを見るところになっている。

```

actionname[0] 50 75 6E 63 68 Punch
loop 00
count 00000002
rectCX 00000015
rectCY 000000E0
width 0000002A
height 00000040
centerx 00000010
centery 0000003C
duration 00000001
actrcnt 00000000
rectCX 0000004A
rectCY 000000E0
width 0000003C
height 00000041
centerx 00000010
centery 0000003C
duration 00000004
actrcnt 00000001
ActRcType 00000001
ActRcCX 00000021
ActRcCY FFFFFFFD0
ActRcW 0000001E
ActRcH 0000000E

```

いかがだろうか？



それほど難しくもないので、分かっていたかとは思うのだけれども、簡単に説明すると、今までの duration の後ろに矩形の数がある。もちろん矩形数が0の時は、矩形なしで打ち切りだ。

だが、矩形の数は可変としているので、ここで得られた矩形数×矩形情報のサイズぶんが duration の後に続くわけだ。可変にしている理由は、状況によって矩形が一つでは足りないこともあるし、そもそも攻撃矩形とやられ矩形が同時に存在することが多いからだ。



格ゲー好きなら一度や二度は見たことある図

ここまで書いておいてなんだけど、僕自身は格ゲーばっかり作ってたので、思考にバイアスがかかっている可能性はある。一般的なアクションゲームだともうちょっと思想が違ってくるのかもしれない。

大差ないとは思いますが、柔軟性は持たせておいた方が発想力が広がると思います。

ちょっと話が脱線しますが、ゲームプログラマの役割というのは、プランナーやらアーティストやらが、安心して発想を広げられる下地を作るものだと思います。

例えば、企画を話し合う際に『ああ、それ無理』『難しいからダメ』『仕様だから仕方ない』と端っから言っちゃうプログラマは、事実上、思考を制限してしまっています。そりゃクソゲーにもなるよ!!

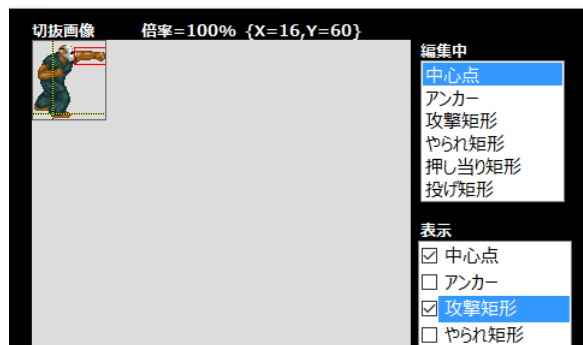
まあ、期間とかハードの制約で泣く泣く削ることはあれど…いや本当にクソプログラマがチームいると発想が痴作まとまっちゃうんだよ!!!プランナーも安心して企画立てられないんだよ。

というわけで、可能な限り柔軟に考えられるようにしておきましょう。

なお、今回の矩形の仕様としては…

```
enum class RectType {  
    anchor, // アンカー(0)  
    attack, // 攻撃矩形(1)  
    damage, // やられ矩形(2)  
};
```

こんな感じで矩形種別が定義されています。なお『アンカー』というのは武器を持った時の目印ポイントです。武器を持つまでは使いませんので後回しです。ひとまず今回は攻撃矩形だけつけています。



あとは単なる矩形データに過ぎないので、読み込んだデータを実機側から DrawBox などに表示してやれば



このように攻撃矩形として表示できます。

矩形を表示する準備

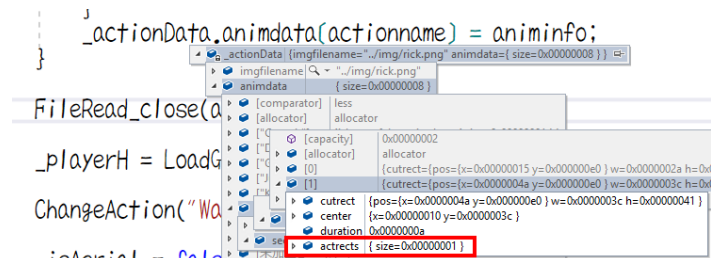
まずは情報を受け取るために必要な準備をします。具体的にはデータ構造の変更です。
カット1つ当たりの情報が増えてしまいます。

```
enum class RectType {
    anchor, //アンカー(0)
    attack, //攻撃矩形(1)
    damage, //やられ矩形(2)
};

///アクション矩形定義
struct ActRect {
    RectType rt; ///矩形種別
    Rect rc; ///矩形情報
};

///カットひとつ当たりの情報
struct AnimCutInfo {
    Rect cutrect; ///切り抜き矩形
    Position2 center; ///画像中心
    int duration; ///継続時間
    std::vector<ActRect> actrects; ///アクション矩形
};
```

こんな感じになる。うーん。ある程度慣れてきたと思うので、ひとまずここまでのヒントで攻撃矩形まで読み込んでみてよ。



一通り読み込んで、こんな感じになって、データが壊れてなければ正解。元通りに動いてたらオツケーって事。

矩形を表示する際の注意点…

これは実際に当たり判定をする際にも言える事なのですが今のキャラクター画像は2倍に引き延ばされています。

そして矩形は元の画像を元にして設定しています。これがどういう事が分かるかな？

特に工夫しなければこうなります。



あら、かわいい矩形ですこと…。

どういう事かと言うと、この矩形は画像拡大前の腕の画像に合わせて表示されてるわけなんです。これを画像と合わせてやらないとカッコつきませんねえ。

というわけで合わせてあげましょう。なお、元々の矩形データと言うのは、中心点(centerpos)からのオフセットとして設定されています。

つまりまずこのオフセットをn倍して、さらに幅と高さもn倍して、プレイヤーの現在座標(centerpos)に足してあげます。

結構面倒なところなので、関数化します。また、倍率はGameクラスに集中管理することにしますので、Gameクラスに

`GetObjectScale()`

という関数を作ります。

こいつに倍率を返させます。その倍率を使って、元の矩形から実際の矩形を返す関数

`GetActuralRectForAction(const Rect& rc)`

を作ります。

この関数内で

- ①倍率によってオフセット値を拡大
- ②倍率によって幅、高さを拡大
- ③プレイヤーのターンに対応
- ④1~3で得られた座標ベクトルをプレイヤーの現在座標に足す。
- ⑤2,4で得られた値を元に矩形を作成し、返す

終わり。

で、例えば

```
GetActuralRectForAction(actrc.rc).Draw(color);
```

で、適切な矩形が表示されるようにしてみてください。ちなみに color は矩形の色です。攻撃矩形なら赤、やられ矩形なら緑といった具合です。

この辺から具体的なコードはお見せしませんので、自分で考えてください。



こんな風になってればいいんじゃないかと思います。

敵さん入りまーす

ここまでで基本の動きができたと思いますが、こんな状態だとひたすら寂しいので敵を用意しましょう。ちなみに雑魚敵の種類がフツソ多いのもこのゲームの特徴ですね。

雑魚の皆さん



デッドマン



トップベビー



バット



ハンギングデッド



ウォーターデッド



イビルドッグ



ノブ



カラス



ジョーカー(かわいい)



マスターデッド



オヴァ



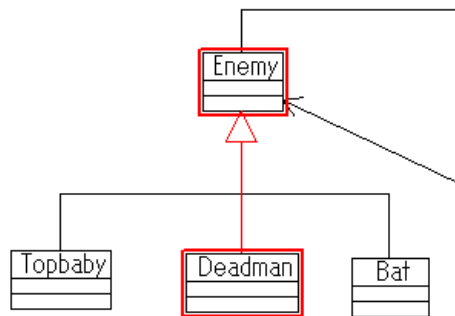
リバイバルデッド



ジュラル星人(ファイアーデッド)

そのほかにももちろんたくさんいるんですが、これだけでも結構な分量でしょう？正直この分量を作るのは大変です。アニメーション作る僕もつらい…。ひとまずは最初の雑魚であるデッドマンを作ります。

-設計？



一応 Enemy からの継承と言う体(てる)で作りますけれども、デッドマン中心に考えて作っていきます。ルールとしては外側から呼ばれるのは

- Update 関数
- Draw 関数
- GetPosition 関数
- GetRects 関数

とします。

それ以外の部分は、好きに作ります。

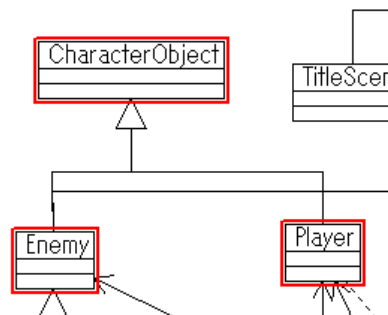
ひとまず Enemy の準備をします。

///敵基底クラス

```
class Enemy
{
public:
    Enemy();
    virtual ~Enemy();
    virtual void Update() = 0;
    virtual void Draw() = 0;
};
```

見ておわかりのように、何もしないクラスと化した Enemy 先輩になります。

結局のところ敵も自機と同じで、アクション再生機ではあるので共通部分に関しては汎化して、CharacterObject クラスを Player と Enemy の親クラスにしましょう。



こいつが protected でそれぞれ共通で使用する関数 CharacterObject に移動させればいいんじゃないかなと思います。つまり…

//アクション変更

```
void ChangeAction(const char*);
```

//アニメーションフレームを進める

```
bool ProceedAnimationFrame();
```

の2関数および

```
int actionH=DxLib::FileRead_open(path.c_str());
```

///バージョンの読み込み

```
float version = 0.0f;
```

```
FileRead_read(&version, sizeof(version), actionH);
```

```
assert(version == 1.01f && "アクションファイルのバージョンが一致しません");
```

```
int filenamesize = 0;
```

```
//画像ファイル名の読み込み
```

```
FileRead_read(&filenamesize, sizeof(filenamesize), actionH);
```

```
_actionData.imgfilename.resize(filenamesize);
```

```
FileRead_read(&_actionData.imgfilename[0], filenamesize, actionH);
```

```
int actionCount=0;
```

```
FileRead_read(&actionCount, sizeof(actionCount), actionH);
```

```
for (int i = 0; i < actionCount; ++i) {
```

```
    int actionnamesize = 0;
```

```
    FileRead_read(&actionnamesize, sizeof(actionnamesize), actionH);
```

```
    std::string actionname;
```

```
    actionname.resize(actionnamesize);
```

```
    FileRead_read(&actionname[0], actionnamesize, actionH);
```

```
    bool isLoop;
```

```
    FileRead_read(&isLoop, sizeof(isLoop), actionH);
```

```
    AnimationInfo animinfo;
```

```
    animinfo.isLoop = isLoop;
```

```
    int animcount = 0;
```

```
    FileRead_read(&animcount, sizeof(animcount), actionH);
```

```
    auto& animdata = animinfo.animdata;
```

```
    animdata.resize(animcount);
```

```
    for (int i = 0; i < animcount; ++i) {
```

```
        FileRead_read(&animdata[i], sizeof(animdata[i))-
```

```
sizeof(animdata[i].actrects), actionH);
```

```
        int actRcCnt = 0;
```

```
        FileRead_read(&actRcCnt, sizeof(actRcCnt), actionH);
```

```

        if (actRcCnt == 0)continue;
        animdata(i).actrects.resize(actRcCnt);
        auto& actrects = animdata(i).actrects;
        FileRead_read(&actrects(0), actrects.size()*sizeof(ActRect), actionH);
    }
    _actionData.animdata(actionname) = animinfo;
}
FileRead_close(actionH);

```

これを

```
ReadAction(const char* filepath);
```

ってな感じで関数化して CharacterObject クラスの持ち物にしてみましょう。

つまり,CharacterObject は

```

#pragma once
#include<string>
#include<vector>
#include<map>
#include"Geometry.h"

enum class RectType {
    anchor,
    attack,
    damage
};

struct ActRect {
    RectType rt;
    Rect rc;
};

struct AnimCutInfo {
    Rect cutrect;//切り抜き矩形
    Position2 center;//画像中心
    int duration;//継続時間

```

```

        std::vector<ActRect> actrects;
};

struct AnimationInfo {
    bool isLoop;
    std::vector<AnimCutInfo> animdata;
};

struct ActionData {
    std::string imgfilename;
    std::map<std::string, AnimationInfo> animdata;
};

class CharacterObject
{
protected:
    ActionData _actionData; //アクションデータのマップ
    std::string _currentActionName; //現在再生中のアクション名
    int _currentCutIdx; //現在表示中のカットインデックス
    unsigned int _frame; //経過フレーム
    int _graphH;
    AnimCutInfo _currentCut; //現在のカットオブジェクト

    void ChangeAction(const char* actionname);
    bool ProceedAnimationFrame();
    void ReadAction(const char* filepath);

    std::string GetRelativeImagePathFromActionPath(const std::string& path, const
char* imagename);

public:
    CharacterObject();
    virtual ~CharacterObject();
    virtual void Draw() = 0;
};

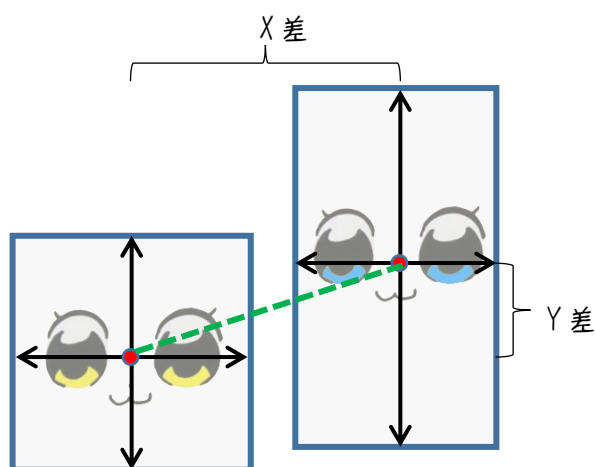
```

こんな感じになります。なお、Player クラスはだいぶ小さくなりました。

あたり判定

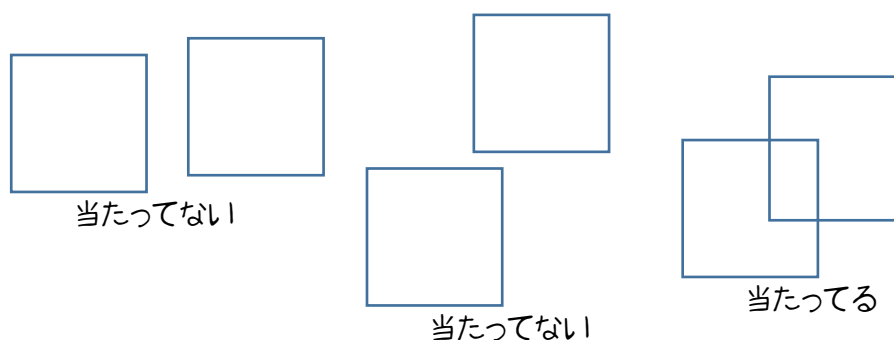
当たり判定はひとまず大したことはしないです。

前述のとおり元々当たり判定しやすい構造として作っておりますので、比較的楽かなと思います。



もしかしたら矩形の当たり判定を習ってないとか、覚えてないとか、忘れちゃったとかいう人がいるかもしれませんので、改めて解説します。

そもそも矩形同士が重なっているのかどうかを数学的に判定するわけですが、数学の問題と違って、1個の答えだけですべてを表す必要はありません。なお、



人間の目で見ると一目瞭然のこの状態。コンピュータにやらせようとなると結構面倒です。

- 左の矩形の右側が、右の矩形の左側より左にあれば当たってない
- 上の矩形の下側が、下の矩形の上側より上にあれば当たってない
- ↑の2つに当てはまってないなら当たっている

ということです。二つの矩形の位置関係によって、それは逆転しますが、行ってる意味は分か

りますよね？

で、

矩形でなく、円の当たり判定の場合は中心点間の距離と、半径ですよね？これを矩形に応用すると

X方向:中心点間のX座標の差<二つの幅/2

Y方向:中心点間のY座標の差<二つの高さ/2

で、両方とも満たしてれば当たっているという事です。

結果的にこういうコードになります

bool

```
CollisionDetector::IsCollided(const Rect& rcA, const Rect& rcB) {  
    return abs(rcA.pos.x - rcB.pos.x) < (rcA.w + rcB.w)/2 && abs(rcA.pos.y -  
rcB.pos.y) < (rcA.h+ rcB.h) / 2;  
}
```