ゲームアーキテクチャ

はい、ゲームアーキテクチャというヨクワカラナイ授業が今年から追加されました。
担当は、昨年まで常勤やってたけど、今年から非常勤になった川野です。

基本的に１年生からは隔離されていたので、初めて目にする人も多いと思いますので、自己紹介ししとこう。

名前…川野 竜一(Kawano Ryuichi)
前々職…大阪の某ゲーム会社で KOF とかの格闘ゲームを作っていました
教える内容:だいたい数学とか、ゲームアルゴリズムとか、CG のアルゴリズムとか周り
現在…フリーランスで非常勤なのに週 10 コマも担当しつつ、レンダリングエンジンの開発のお仕事とかやっています
趣味…ボクシングと自転車と猫と、CEDEC とかの公募に応募する事、あとなんかゲーム作りのコンテストとかに応募する事…とにかく目立ちたい
著書に『DirectX12 の魔導書』があります。あと、Cedil で僕の名前で検索するといくつか出てきます。何故か『まんがとイラストで分かる GPU 最適化』って本の後ろに僕の名前が載ってます。

な感じ。では始めましょう。

目次

はじめに

この授業が何のための授業かという事なんですが…まぁ、正直頼まれた内容がふわっとしすぎてて、僕としてもなんて言ったらいいのかよく分かりません。

昨年までは最上級コースの開発全部見てた感じだったんだけど、非常勤だから「**なんかクオリティアップにつなげられるような内容にしてください**」という、本っ当にふわっとした要求。なめとんのか!!

しかも、まぁ、ぶっちゃけた話、学生さんのプログラミングレベルにかなり差があるという…これが一番つらいんだけど僕としてもいちいち別々に授業組む余裕もないので、ある程度は統一します。

進度と深度をちょっと変えるくらいかな。
基本的に僕はゲームのプログラムしか教えられんので、この時点で IT 系に行くとか、一般職に行くとか言う人に対応しません。
色んな学生さんがいるって事は知ってるので、全員が全員ゲームプログラマを目指しているわけじゃないのは分かってるけど、それはそれとして、課題はきっちり出してもらいます。

ともかく「みんながゲームプログラマを目指している」という体(てい)で授業しますし、お話します。そもそも目指してない人は何でここにいるの？って感じなんだけど、まぁ何かしらの事情があるんでしょう。とはいえ、高え学費と、1 日のうちの何時間も消費してる事は自覚して、きちんとそれだけの対価を支払ってるという認識を持ってください。

その結果、ここにいるべきかどうかを判断するのは自分です。もう二十歳くらいにはなってるんだから、その選択については自分で責任を持ってください。

「目指さないんだったら出ていけ」と言ってるわけじゃないですよ？全然目指してなくてもなんとか 3 年なり 4 年やり過ごしてただ卒業する意味があった事例もたくさんありますからね。

ただ、特に次年度就職年次の人は、少し覚悟をしてもらった方がいいと思います。生半可な覚悟だと精神的にきついです。今のウチに言っておきます。

何でこういう事言うのかというと、昨年までに悲しい、つらい状況を沢山見てきたからなん

だよね。もし本当に目指す気があるんだったら、作る事を習慣づけたうえで、きちんと戦略を練ってください。

習慣について

何でここでいきなり習慣の話をするのかというと、**失敗する奴ってのは無理をしがち**だからです。たまに無理をするのならまだいいのですが、目標そのものが不可能な目標だったり、そもそも自分でも到達できると思ってなかったりしてて全くもって意味がない事になってる事が多いんですよね。

とりあえず次年度の人は今 4 月でしょ？で、就職活動がだいたい 12 月半ばくらいから始まります。つーわけで約 10 カ月。この間に就職活動ができるだけのゲームを作れるようになってる(そして実際に作ってる)必要があるわけです。

10 カ月というと約 300 日…土日を含めるか含めないかはお任せしますが、300 日まるまる使えるわけじゃないと考えると約 220 日くらい？の間にどこまで行けるかがカギなのですが
できない奴ほど無理をしようとします。

ちょっと言っておくと、できない奴ほど、ほんの少しずつほんの少しずつでいいからできるようになる事を考えてください。**いきなりは無理!!**
現実の世界に覚醒とか…ないから!!

授業で分からなかった部分を放課後に 30 分だけ復習するとか、そんなんでいいです。それすら難しいとは思いますが。ただ**この 30 分を怠ると課題提出前に何時間も、集中力もないまま苦痛な時間を過ごした挙句に何の成果もなく、能力も進歩しない**ってのはよくある事です。

無理をしないように、30 分がきついならまずは 10 分でも 5 分でもいいので、ちょっとだけ残って他の人が遊んでいる間に復習しましょう。何度も言いますがここで無理してはいけません。
最初はモチベーション高いんで 3 時間とかやっちゃいますがだいたいすぐ挫折します。
3 時間くらいが当たり前の人は良いですが、今現在できてない人にとっては 30 分すら大変なはずです。10 分から始めましょう。

よくいるのが3~4年ゲームプログラミングやってきたのに if 文や for 文も分からない人がいます。うん、全くの無駄なので、まずはそういう部分だけでも理解する所からやっていきましょう。

とにかく**無理をせず、「最初は習慣作り」と思って「なめとんのか？」ってくらいゆっくり始めましょう。**まだまだスタート地点です。エンジン全開まではまだ早い。

ゲームプログラマになりたきゃ、ほんの少しずつでいいからそこに近づく努力をすることを習慣化しましょう。意志の力だけでは…無理です。

そして、ごくごくたまになら無茶をしてもいいです。たまには無茶をしましょう。

さて、じゃあガムシャラに習慣化して努力してりゃいいのかというとそうでもない。アホは戦略も立てずにやろうとして徒労に終わる。

戦略について

戦略って何かっちゅーと、最終目標に到達するためのやり方って事ね。で、これは人によって違う。その人の特性によって違うから、自分で考えなきゃいかんわけ。

さっきも言ったけど、次年度なら残り 10 カ月。この 10 カ月の間に何を用意するのか…簡単に言うと計画なんだけど、計画を立てるだけでも夕メなんだよね。自分の持ち味をどう活かすのかを考えないと。

多分、皆さん少なくとも 1 年以上はプログラムしてるからもう薄々分かってると思うけど、同級生の中には「クッソ強いやつ」がいると思う。到底こいつには勝てないな…と。

こういう奴と張り合う必要は、ないです。

どうも就職活動を学校のテストと同じように思ってる人がいるかもしれんけど、ちょっと違う…いや、そんなに離れてるわけじゃないと思うけど違うのよね。何が違うのかというと、会社によって**そもそものニーズ**が違うし、自分の特性上としても伸ばせる部分とそうじゃない部分があるわけ。

まず戦略のポイントとしては

- 会社(大雑把でいい)選び
- 自分の何をウリにするのか

この 2 つがポイントなのよね。まぁ～だ実感がないかもしれないので、やれ任天堂だ、バンナムだ、サイゲームズだなんて出てくると思うんですが、そんなん就活考えてるとは言えないわけ。

あ、レベルが高すぎるって意味じゃないよ？確かにレベルが高いけどさ、こういう会社の名前が出て来るって事は正直その会社について…その会社の開発の仕事についてな～んも分かってないって事。小学生が自分の希望を言ってるのと変わらないわけ。

そんなんじゃうまくいかないんだ。例えば『橋本環奈と結婚した～い』なんてアホがいたとする。こいつがうまくいかないのは、別に不細工だからでも高望みだからでもなく、上辺だけしか見てないわけですよ。そんなんじゃうまくいくわけないんですよ。

当たり前だよなあ。

当たり前なのに、就活になると途端にそれと同じことを言う奴がホイホイ出てくる。じゃあ分かった、任天堂/バンナム、サイゲームズを受けたいのは良いけど、その会社の特性とかニーズって知ってる？って話よ。大抵答えられんのよ。

今回は例として大きな会社を言ったけど、その会社が何に強みを持ってて、自分はどういう面で貢献をしたいのかという所がはっきりしてないと、戦略も立てられんのよ。

とはいえ、会社の細かいところまで知るのは大変。だからまずは手始めに『何系を目指そう』くらいでいいから、イメージしてみよう。

- CGが凄いところでシェーダとか書きたい
- AIが凄い所でディープラーニングって奴で何とかしたい
- とにかくたくさんのゲームを作りたい
- メモリとかスレッドとかシステム周りをやりたい
- ゲームエンジンを作りたい

と言ったところから固めて来ると自分の方針も決まってくるし、どういう所を重要視してる会社かを考えながら探すとうまくいきやすい。こういうのを知りたい人は就職年次の担任の先生とか就職決まってる先輩に聞いてみよう。

で、前述の何系って話だけど、もっと大雑把に言うなら

- とにかく数をこなそう
- 技術力をつきつめよう

てな話になると思います。ゲームを沢山作ればそれだけ技術力も上がると思いますが、浅く広くって感じになるかなと思います。
逆に技術力を突き詰めるってのは、深く深くって感じです。深く深くの場合はちょっと気を付

けておかないと、自分に向いてなかったり会社のニーズが無かったりすると大外れになるので、それも就職年次の先輩とかに相談するといいと思います。

数をこなす人は 10 カ月の間に何本作れるのか…それを考えてください。技術力に自信がないなら、小さなゲームを沢山、とにかくたくさん作るのがいいでしょう。

あと技術系の人にお勧めなのは、CG の見た目に関わる部分を突き詰めるのはお勧めです。何故かというと、ぱっと見が派手だと企業の人の目に留まりやすいからです。何だかんだ言っても売込みってだいじなので。シェーダ書けるようになってポストエフェクトとかできるようになるとかなり強いです。

あと AI 系の人にちょっと言っておくけど、よほどのことがない限りディープラーニングはやめておいた方がいいです。基本の理屈がそもそも高難易度なので、最初は経路探索(ダイクストラ法とか、それ以前の計算幾何学とか)とか、それでも難しければ AI というよりきめられたパターンで動くもの。それも状態遷移を用いて動かすものから作った方がいいでしょう。
よく AIAI 人工知能っていう人は何か勘違いして失敗することが多いです。思ったよりもややこしく、概念そのものを勘違いしていることが多いです。

とはいえ、たぶん大半の人は数をこなした方がいいでしょう。つまるところ、技術を突き詰めるよりもたくさん作った方がいいと思います。その中で何かのめり込めるテーマがあったら、それを突き詰めてもいいでしょう。ただし、ニーズが全然ないものを突き詰めても意味がないので、自信がない人は誰かに相談しましょう。

その際にきれいなプログラミングを心がけて欲しい所ですが、それで手が止まったら元も子もないので、まずは沢山作って、企業に出す前に「リファクタリング」かましましょう。

あと、企業の人の目に停めてもらう回数が多ければ多いほど有利になりますので、魅せるためのデザイン的な所は軽くでいいからお勉強しておきましょう。

あと、できるだけコンテスト等に応募したり、twitter 等で活動を公開したりしてとにかく露出を増やしましょう。それが現代の戦略ってもんです。

まぁ色々と話してきましたが、この残り 10 カ月をどう過ごせばムリなく最大効果を得られるのかを考えながら行動しましょう。

それをやらない人は多分 10 か月後に後悔すると思います(就職年次の先輩の半数が後悔しています)

技術へのアンテナについて

皆さんはゲームプログラマを目指しているので、アンテナを技術に向けて立てておきましょう。

そもそもゲームでどういう技術が使用されているのか？どういう技術が必要なのか？どんな本やサイトを見ればいいのか？
クソ下らねえ芸能人とか Youtuber のゴシップ見てる暇があったら、そういうのにアンテナを張ってください。というかここにいる以上、そういう生き方にもう片足突っ込んでる自覚を持ってください。

ちなみにアンテナの参考になるのが
Qiita
https://qiita.com/

SlideShare
https://www.slideshare.net/

SpeakerDeck
https://speakerdeck.com/

ゲームつくろー
http://marupeke296.com/GameMain.html

Cedil
https://cedil.cesa.or.jp/

あと twitter つよつよエンジニア系のひととかフォローしとこう。でもあんまりツヨツヨばかり見てると自分の未熟さに嫌気がさすので、ほどほどに。

授業の流れについて

で、ここまで色々と話してきましたが、この授業自体は、皆の開発の面倒を見るという感じで

はなく、様々な開発のテクニックを伝授するものです。なので、それをどう活用するかは皆さん次第だし、前述の戦略を自分で立てて、役立つと思ったらメッチャ利用してほしいし、そうでないと判断したら、単位を取るためだけにこなしていけばいいかなと思います。
プログラミング系では多分この学校では一番利用者が多いであろう DxLib で説明していこうと思います。ですが、他の環境でも応用できる話にする予定です。あくまでも DxLib を使うというだけ。

あと、昨年の授業で「他校が DxLib で3D やってるのにビビっちゃった」学生さんがいたので、DxLib で 3D を軽くやっとこうかと思います。本当に軽くね。

で、流れですが(ある意味ここがコマシラバスね)
1. まず開発環境 VisualStudio について(設定とかデバッガとかコマンドラインとか)
2. Git とか GitHub とかについて(チーム制作以外でも)
3. DxLib 使いつくしてる？DrawGraph くらいしか使ってなくね？
4. DxLib で簡単なポストエフェクト(シェーダとか使わない)
5. DxLib で簡単じゃないポストエフェクト(シェーダ使ってみる)
6. DxLib で 3D やってみる
7. ちょっと特殊な当たり判定(題材は DxLib を使うが…)

こんな流れにしようかと思います。週 2 コマのはずなんで上の 1 単元に 4 コマ使う感じで…みんなのレベルが高ければここに書いてる以上の事をやると思いますし、皆がボンクラーズなら全然進まないかなと思います。

あと余裕があれば合間合間で「色の話(効果/組み合わせ)」とか「フォントの話」をしようかなと思います。

どこまで進むかは皆さん次第です。せっかくだからなるべく沢山僕から吸収してしまいましょう。

環境とか VisualStudio とかコマンドについて

環境変数

意外とこれ知らない人が多いんで確認しておきますが、環境変数って知ってます？
3 年生はすでに分かっていると思うのですが、2年生にとってはこれからお話しすることは初耳かもしれません。

「DxLib のフォルダは C:¥DxLib や D:¥DxLib とは限りません」

というか自宅で作業したことのある人ならわかると思いますが、DxLib のライブラリも自分でダウロードすんだよ当然だろ。というのが2年生以降の話です。一応 PC をセットアップした時点で僕が学校内の全 PC の C か D の直下に置きましたが、それができない環境もあったりしますので、

「DxLib をどのフォルダに置いていても環境設定さえすればどの PC でも同じように使える」ということをお教えします。めんどうですよね。

なぜこういうのが必要かというと、就職活動などをする際に、相手方は C や D の直下に DxLib のフォルダなんてまず置きません(もっと言うとダウンロードもしないと思います)。

あと、新2年生にとってはライブラリのリンク経験は DxLib くらいしかないのかもしれませんが、今後のプログラミングをやって行くと分かると思いますが他にもいくらでもあります。

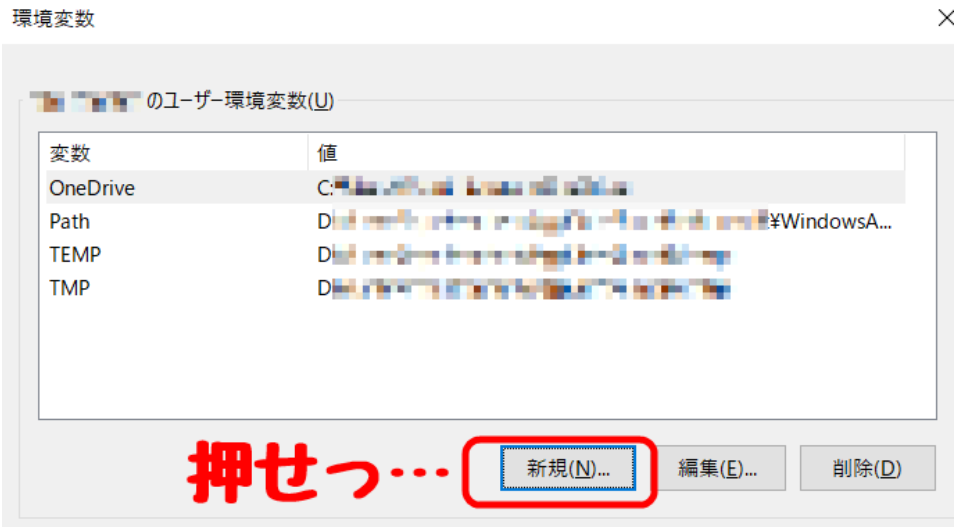
というわけで、今後はライブラリのフォルダを直で書くのはやめましょう。1年生の頃はね、覚えることが多かったからこういう細かいことは教わらなかったと思います。頭がフットーしちゃいますからね。

① まず自分の PC の DxLib のフォルダの場所を確認します。

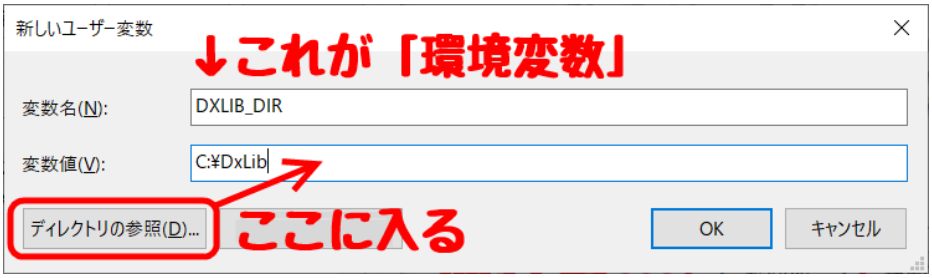
② 次に Windows 左下の検索バーに「環境変数」と日本語で書き込みます

③ そうすると2つくらい候補が出てきますが「システムではない ほう」を選択します。

④ そうすると「環境変数」のウィンドウが立ち上がりますので、上下に分かれている画面の上側で「新規」というボタンを押します。押せよ!絶対押せよ!



⑤ そうすると新規で『環境変数』を作るためのウィンドウが出てきます。今回の環境変数はディレクトリのためのものなので『ディレクトリ』ボタンを押して、DxLib が置いてあるディレクトリを選択してください。
また、上の段はそのディレクトリを表すマクロ変数みたいな感じになるので、そこに DXLIB_DIR と書いておく。終わったら **OK を押す**(☆右上の×を押すなよ!絶対押すなよ!)



⑥ OK 押すと元のウィンドウに戻る。この時点で環境変数が出来上がっていると思うだろう？素人はまんまと引っかかる。

ところが大間違いなんだ。よくやるやつが右上の×を押しちゃうやつ。違う…っ!!

ウィンドウを閉じるときに反射的に×を押したくなる気持ちはわかる…っ!!しかし…それが罠っ…!!!巧妙に仕掛けられた悪魔的…罠っ!!今までの作業が台無しになってしまう。必ず OK を押すようにしよう。

⑦ さて、これで話は終わりではない。コマンドプロンプトを立ち上げてくれ。何ィ？コマンドプロンプトを知らないだぁ!?お前ここをカルチャーセンターと間違えてんじゃねえのかァ？

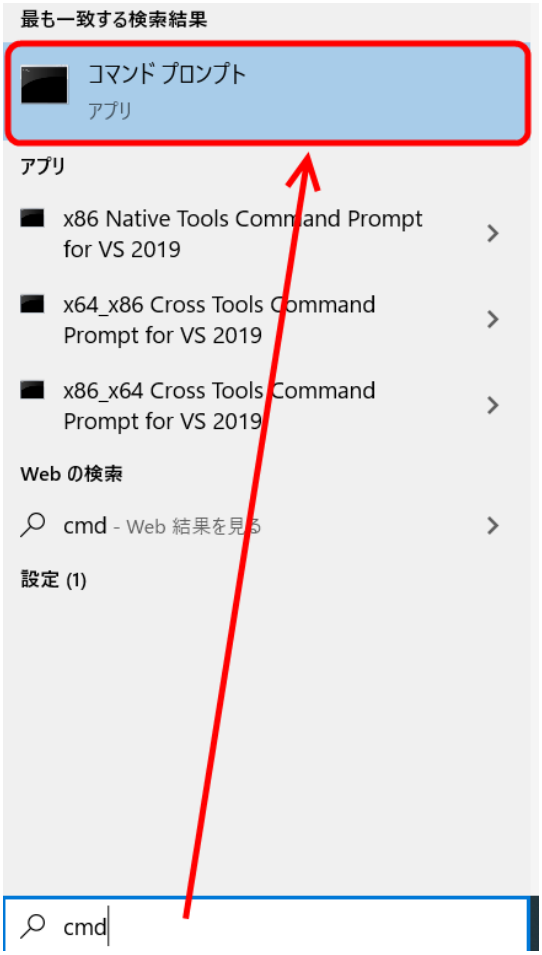
コマンドプロンプトというやつは、システムコマンドを打ち込むためのものです。Linux とか使ったことのある人なら Linux コマンドは知っていると思うけど、ls だの cp だの cd だのというコマンドで OS に命令を出したり情報を引き出したりするためのものだ。

出し方はいたって簡単。ご注目!!いきますよ～よく見といてくださ～い。

先ほど環境設定の時にやったように左下の検索バーに"cmd"と3文字打ち込んでください。

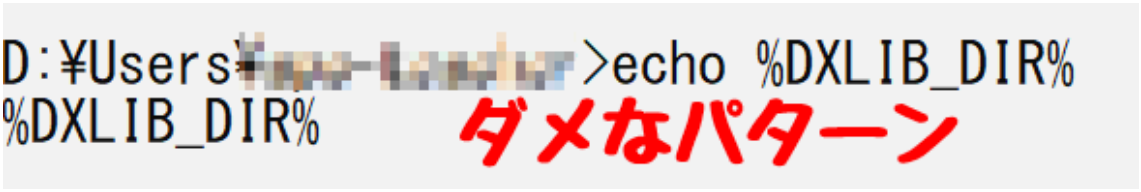
いいですかー、しー、えむ、でぃー、ですよ～。

はいこれで、立ち上がります。見覚えのある人もいるかと思います。デフォルトが黒背景の白文字なので、説明では見づらさ軽減のために設定で白背景の黒文字で表記しますね。



⑧ コマンドプロンプト上で"echo %DXLIB_DIR%と打ってエンター。まともに設定できていれ

ば、完全なパスが表示される。表示されなければ設定に失敗してんだよもっかいやんだよあくしろよ。

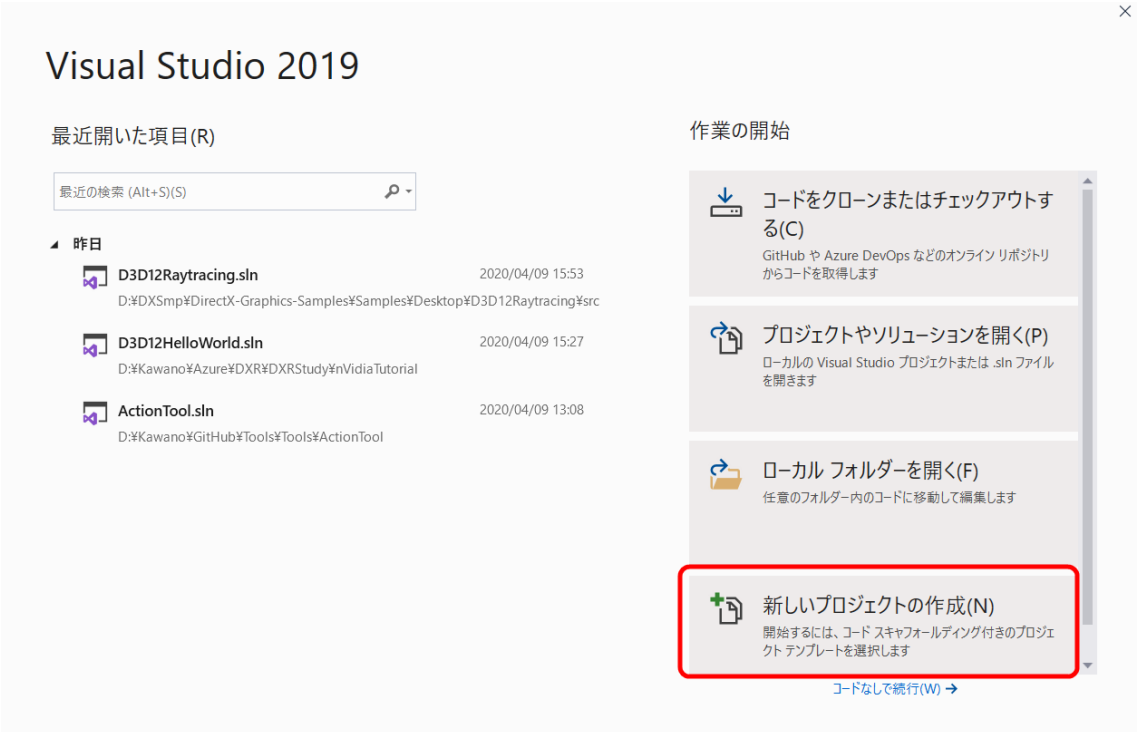


うまくいっていればフルパスが表示されます。なお、やり直す際はいったん全てのコマンドプロンプトを落としてもう一度立ち上げなおしてください。OK なら

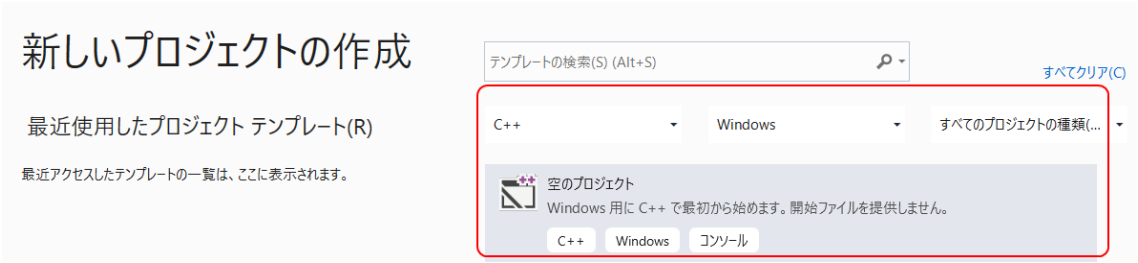
このようにフルパスになります。これで設定できていることが確認できました。

⑨ 次にもし VisualStudio を立ち上げているのであればすべて終了してください。一つでも立ち上がっている状態だと、この環境変数の変更の影響を受けないからです。

⑩ そして VisualStudio を再び立ち上げて
新しいプロジェクトを作成します。



ここで作るのは「空のプロジェクト」(C++ Windows Console)です。

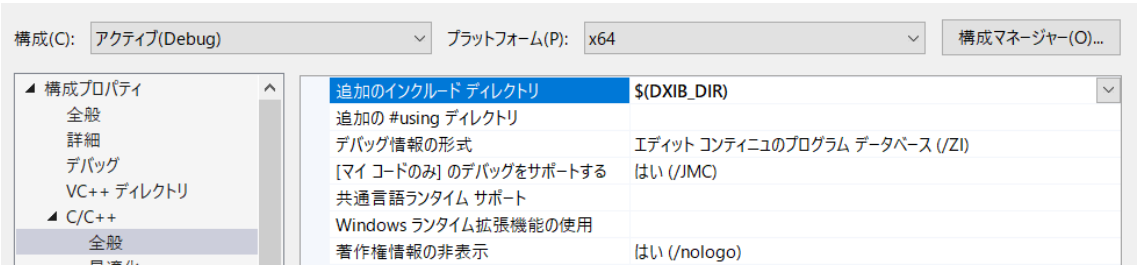


⑪ ソースコードとして main.cpp を追加してください。で、以下のコードを追加します。

```
#include<DxLib.h>

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int) {
	DxLib::SetGraphMode(1280, 720, 32);
	DxLib::ChangeWindowMode(true);
	DxLib::SetWindowText(L"Ninja Sprit");
	if (DxLib_Init()) {
		return -1;
	}
	DxLib::SetDrawScreen(DX_SCREEN_BACK);
	while (DxLib::ProcessMessage() == 0) {
		DxLib::ScreenFlip();
	}
	return 0;
}
```

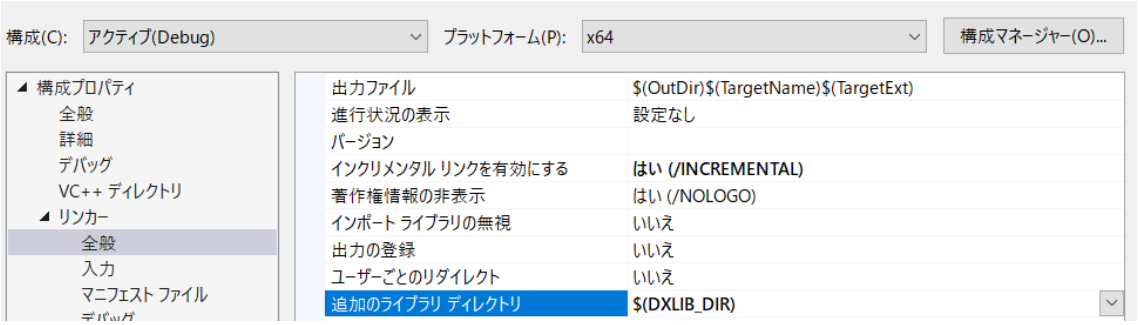
まぁ、言うてもこんなん本質にはかかわりないので、写経しようがなんしようが好きにすればいいです。

⑫ このままでは DxLib へのインクルードパスもリンクパスも通ってないので、動かすことができません。そこでプロジェクトの設定に移ります。設定のところで



追加のインクルードディレクトリで$(さっき作った環境変数名)と書きます。この文字列が先ほど作った環境変数が表すディレクトリパス文字列に置換されます。

⑬　次にリンク部分にも同じことをやります。



追加のライブラリディレクトリに$(環境変数名)とします。

はい、一応これでスケルトンプロジェクトが立ち上がるはずです。簡単なのでさっさとやりましょう。全然プログラミングと関係ない部分です。
なお、インクルードとリンカの部分を話しましたが、そのうち「コンパイル」と「リンク」の意味についてもお話しします(大事なことです)。

で、実行しようとするとリンカエラーが起こることがあります。その場合はサブシステムが「CONSOLE」になっているので「WINDOWS」に変更してください。(なお、サブシステムはリンカーのシステムの項目を見ればあります)

あと、今回の授業では 64 ビットで統一しようと思います。皆さんはどちらを選んでもかまいませんが、もし 32 ビットを選択して起きたトラブルに関してはご自分でご対応ください。よろしくお願いいたします。

ともかく実行するとこんなウィンドウが出ます。

出ましたね？次行きます。

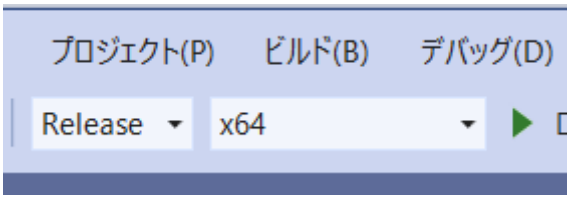
ちなみにコマンドラインのコマンドの echo は環境変数の真の姿を表示するものです。これは後述します。

じゃあまず、そもそもの道具の VisualStudio についての理解を教えてくれるかな？

VisualStudio のプロジェクト設定

プロジェクト設定の前に…

そういえばプロジェクト設定の話の前にさ…



これ、きちんと気にしてます？これって、デバッグでもリリースでもともかく実行ファイルを作るときに指定するものなんだけどね？そもそもデバッグとリリースの区別を知らん人がいるっぽいので説明しておくと

Debug と Release

左のプルダウンメニューには通常「Debug/Release」があって、

- Debug はデバッグするための exe を作るモード
- Release は実際に配布するアプリとして exe を作るモード

なのよ。なので**コンテストに出す時とか、企業に提出する時には Release モードでビルドしたものを提出して**ね。この辺キチンとしてないと「そんなのも知らないのプギャー!」ってなって中身も見られずに落とされたりするからね。

あと、その右側の x64 ってあるけど、普通は x64 以外に Win32 か x86 ってあるけど、これは古いＰＣ向けって事ね(32bit マシンの事)今時 32bit しか受け付けないってＰＣの会社もないだろうし…というかそんな今時 32bit マシンのＩＴ人権のない会社にはいかない方がいいです。

なので、基本的には「x64」にしておきましょう。ここをまずは「Debug」「x64」にしておいてください。後述するプロジェクト設定はここに合わせてやることになります。

なお、面倒ですが、きっちり設定を使用と思ったら Debug の設定、Release の設定の2つを設定しなければならないためそれは把握しておきましょう。

32bit と 64bit について

そもそもなんで 32bit の指定が x86 で、64bit の指定が x64 なのさ。x64 はまだ理解できるけど、86 をど～～～いじっても 32bit にならんじゃろが!!と思われるかもしれません。そりゃごもっともです。

何でと言われても、もうこりゃ歴史的経緯としか言いようがなくて、32bit のプロセッサ…いや、16bit プロセッサをインテルが開発して、それの名前が Intel8086 だったんですよ。そもそも、そこが始まりなのよね。で、何故かこの設計を元にした Intel のプロセッサは 86 系と呼ばれることになった。

ぜんぜん 32bit 関係ないやないか!!いいから加減にしろ!!!と思うかもしれないけど、そもそも 32bit プロセッサは 16bit プロセッサの拡張に過ぎなかったのだ。で、IA-32 って名前がついてたから、x32 ってすればいいものを、通りがいいからとそのまま x86 と言い続けてきたのが今に至るわけ。

で、64bit の時代になって、IA-64 というのが使われるようになったんだけど、こいつは 8086 と互換性がないため、もはや 86 と呼べなくなった。ということで、IA-64 から 64 をとって x64 って言ってるだけなのだわ。

そもそもこの 32bit だの 64bit だのってのはなんなのさ？何が 32bit で何が 64bit なん？

…ちょっとファミコン時代にさかのぼりましょうか。

ファミコン 8bit マシン 8bit＝1byte

ファミコン時代のプロセッサはご存知 8bit でした。いやご存知ないかな、生まれる前だもんね。実物すら見たことないでしょう。

まぁ、それはともかくファミコン時代は 8bit マシンだったので、一度に計算できるデータの量が 1 バイト(8bit)だったわけです。

一度に計算できる量が 8bit ということは、実は数値としては 8bit マシンでは
00000000～11111111
しか扱えないのです。とはいえ、ファミコン時代にもシューティングゲームのハイスコア等を見れば分かりますが、256 以上の値を取り扱っています。これはどういうことなのでしょうか？

たとえば 16bit ならば 2 バイト。これが取り扱える値は
0000000000000000～1111111111111111
です。つまり 0～65535 ですね。レトロゲー好きなら分かると思いますが、昔のゲームのカンスト値でよく使われてた数値ですね、65535 ってのは。

まぁ、それはともかく、8bit マシンでも 16bit の値を演算することは可能です。どうやるのかというと、上位 8bit と下位 8 ビットに分けます。

00000000,00000000～11111111,11111111

でそれぞれを 8bit として別々に演算します。で、キャリーオーバーというか、溢れた分は上位ビットに渡すためにぶっちゃけると 16bit 同士の加算でも 8bit 演算器が少なくとも 2 回は走ってたわけです(繰り上がりがあれば 3 回)

ということで、一度に計算できる量が 16bit になったのが 16bit マシンです。これの代表はメガドライブであったり、PC エンジンであったり、スーパーファミコンだったわけです。単純に考えて2～3回かかってた演算が 1 回で済むので、速度が倍になるというわけです。ま、あくまでも単純な話ですが…。

まぁ、実際にはそれだけじゃなくて、クロック周波数も上がってるから、比べ物にならないくらい性能が上がっているわけです。

最も大きな違いは色数で、ファミコンが 256 色のうちの 8 色しか使えないって感じなのですが、実際には 54 色のうちの 8 色です。まぁ、ヤヤコシイ理由があるのです。

これに対してスーファミは 32768 色のうちの 256 色が使え、かなり表現力が上がりました。

まぁいいことづくめに見えますが、デメリットもあります。
それは、1 回の演算の枠が広がっちゃったことで、それだけメモリを食うように、レジスタを食うように、ストレージも食うようになっちゃうわけです。

え？ちっちゃいものも共存できないの？
と思うかもしれませんが、CPU の特性上１回の演算自体が 16bit 演算になっているため、例えば

```
struct Sample{
        char c;//8bit→1バイト
```

```
        short w;//16bit→2バイト
};
```
等と書いた場合、素直にメモリ配置が行われ、1バイトの c の直後に 2 バイトの w が来てこれまた素直に演算機が動いてしまうと、困ったことになります。

メモリ上の配置が 16bit(2 バイト)の倍数になっていないばっかりに 2 個目の w の演算が 8bit と 8bit で別れてしまい、非常に非効率な計算になってしまうのです(バグらないだけマシ)。

このためコンパイラ君は非常に賢い事をやってのけます。

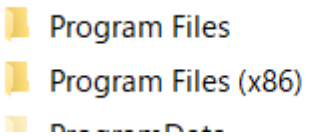
```
struct Sample{
        char c;//8bit→1バイト
        //目には見えない1バイトの空白
        short w;//16bit→2バイト
};
```
こういう配置にする事に酔って 1 バイト勿体ないですが、計算は効率的になります。計算公立自体はいいので、そこは基本的にはコンパイラ君にお任せしてればいいのですが、外部のバイナリファイルを使う時はそうもいかないので注意してください。
ひとまず頭の片隅に「そういう事やってる事がある」という風に覚えておいてください。

だから、プロセッサの使用ビット数が上がれば上がるほど、それだけメモリも必要なわけです。

さて、それはさておき、Windows7 以降の PC はだんだんと 32bit から 64bit になってきて、今時は殆どが 64bit マシンになっています。

ここまで話したように、64bit と 32bit では CPU の仕様が違うため、32bit のアプリケーションは 64bit では動かないはずなのですが。そうなのですが、32bit のアプリケーションはまだまだ使用されており、64bitCPU 側では OS のお力添えで「エミュレート」してる状態なわけです。

このため、ProgramFiles を見てもらえばわかるように

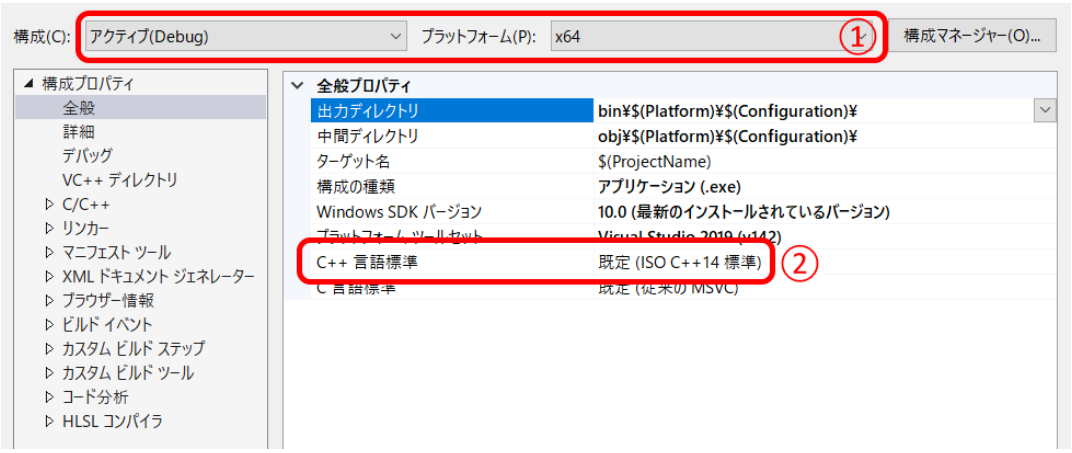


もはや別々に分けられています。ちなみに VisualStudio などはまだ今の所互換性を保つために x86 側に入っていますが、blender や画像処理ソフトなど、高速な演算が必要なものは順次 64bit 版に切り替わっています。

今時ゲームに関わる会社で 32bit マシンを探す方が大変なくらいなので、皆さんにとって高速化は大事な事でしょうから、プロジェクトは 64bit つまり x64 でつくるようにしましょう。

プロジェクト設定

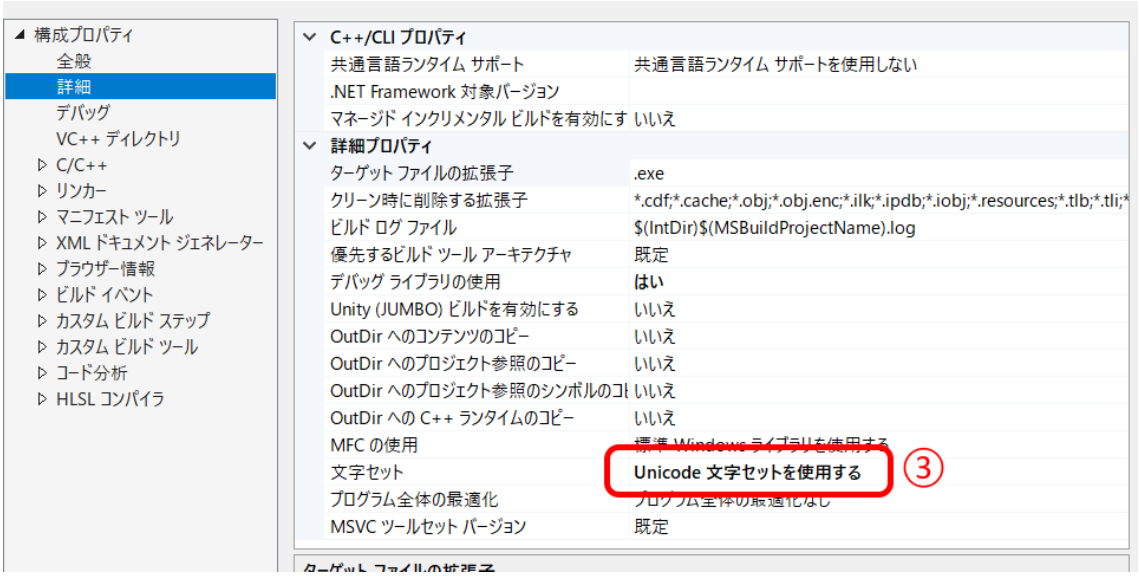
さてさて、プロジェクト設定の中身ですが開くとこんな感じになるかと思います。



まず①は必ず確認しましょう。今設定中のものが Debug なのか Release なのか x86 なのか x64 なのか…ここを間違えるとせっかく設定したものが実行時に反映されておらず、あれ？あれ？ってなります。

次に②ですが、これはＣ＋＋の機能をどれ標準にするのかを設定します。今の VisualStudio2019 ですと、Ｃ＋＋14が既定になっていますが、**現代は C++17 が標準なので、ここは C++17 にしておくことをお勧めします。**

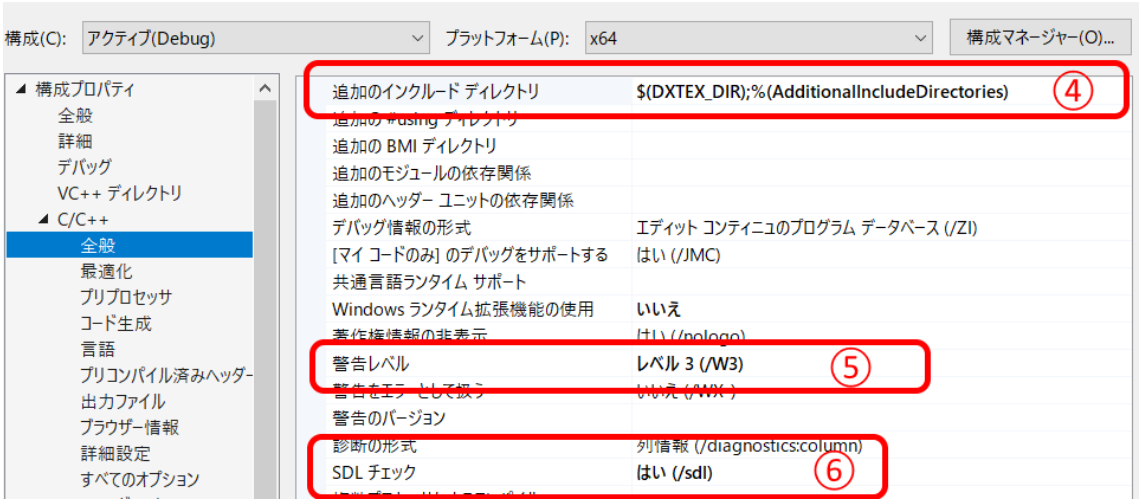
次は「詳細」をクリックしてください。ここは見るのは③の部分だけでいいです。



Unicode 文字セットとマルチバイト文字セットのどちらかを選択できます。

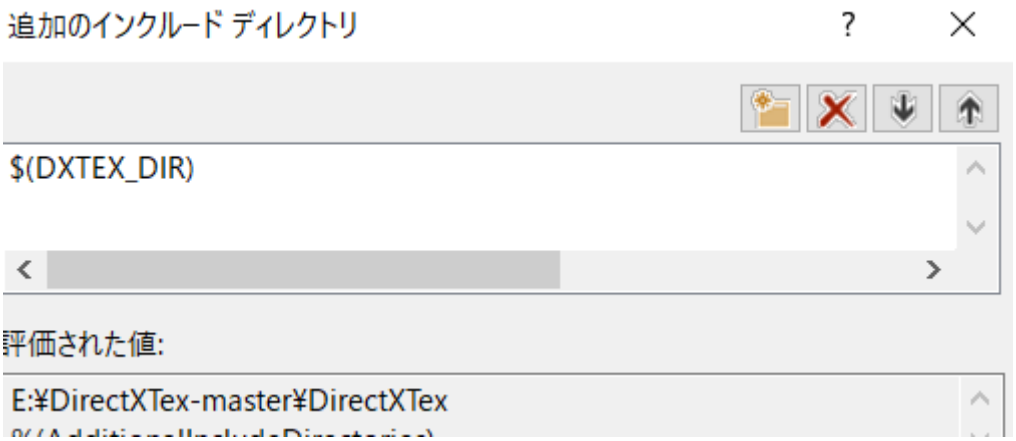
プログラミング初心者は「マルチバイト文字セット」がいいと思いますし、今後の拡張性を考えるなら「Unicode 文字セット」の方がいいと思います。

文字セットの話はちょっとややこしいので、ここでは 2 種類あるという事だけ把握しておいてください。次に C/C++です。



C/C++編

④はみんな大好き追加のインクルードディレクトリですね。おなじみですね。上では環境変数を入れてますが、④の枠の右にあるプルダウンメニューから編集と押すと



が出てきます。環境変数の DXTEX_DIR の中身が見えていると思いますが、ここでパスがまちがってないかどうか確認しましょう。

次に⑤ですが、これは警告レベルです。数値を上げれば上げる程警告が厳しくなります。セキュアなプログラミングをしたい場合には上げまくりましょう。

⑥もセキュリティ関するものです。ここを「はい」にしているとメモリアクセス系の標準関数

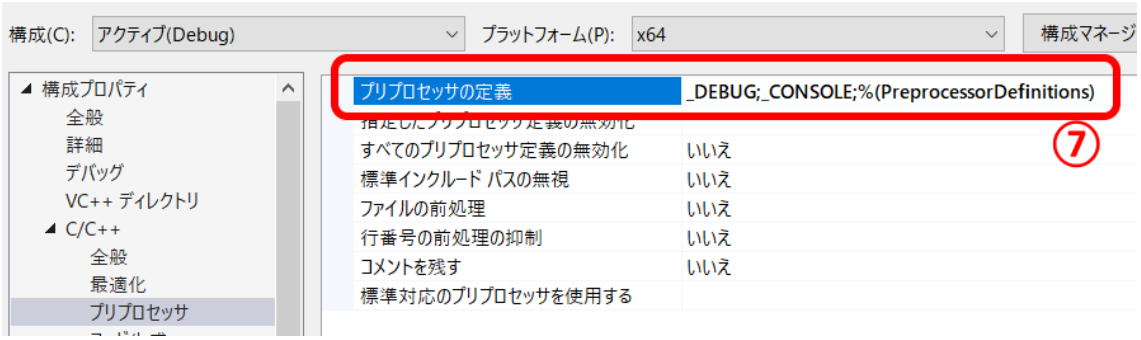
には_s をつけたものを使用しないと怒られます。

scanf←SDL がハイの時は不可。scanf_s とすべき。なんですが、これ MS でしか通用しないし、色々と問題あるので、ぼくは「いいえ」にしておくことをお勧めします。

で、次の最適化の所は、Debug/Release の区別がついてりゃいいので、飛ばしてよくて、割と大事なのはその次の「プリプロセッサ」です。

プリプロセッサの細かい話は後述しますが、大雑把に言うと pre(前の)process(処理)からきてる言葉なので「前処理するやつ」みたいな意味です。

ここでは C/C++言語の中でよく見かける＃〇〇に関わっていると思ってください。
#include
#define
#pragma
#ifdef～#end
などですね。実際に設定画面を見てみましょう。



はい、基本的には↑の⑦…ここしかいじる所はございません。
「プリプロセッサの定義」なんて書いとりますけれどもこの「定義」ってのが define の直訳だという事に気づけば…あとはわかるな？

つまるところここで#define と同じことができるわけ。なんだけどそこまで意識して使ってる人が何人いるのかなって感じもする。

ちなみに上のレイだと_DEBUG やら_CONSOLE やら書いてますが、最近はこれすら書かなくなったかな。デフォルトだと_MBCS って書いてるけど、これは「マルチバイト文字セットを使用します」って事ですね。

通常の DxLib を使用している分にはあまり見る事はありませんが、一つだけ覚えておいた方

がいル「定義」があります。それは

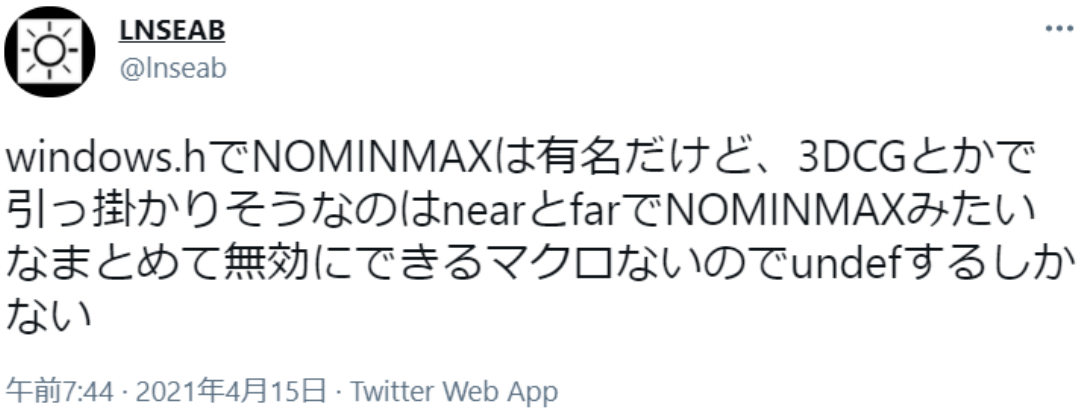
NOMINMAX
（農民マックス！）

です。これは何かというと

Windows 系の開発をしているときに Windows.h のクソマクロ(ファッキュー!!)の影響で min とか max のつく関数が誤動作を始めるのだ。恐ろしい恐ろしい…。

で、そもそもそのクソマクロを無効化してくれる「定義」なのだ。恐らくは Windows.h を作った側もこれが「クソマクロ」だという事は自覚しているのだろう…。

ちなみに朝にこれを twitter で呟いたらこういう事をつぶやいてる人がいまして…



あー…near far もそうなのか。本当に define マクロは害悪だよ…と思いました。
define マクロを知らない人のために軽くだけ説明しておくと、define マクロとは define の機能を関数的な感じで使ってしまおうというわけだ。

例えばこのように定義する

```
#define min(x,y) x<y?x:y
```

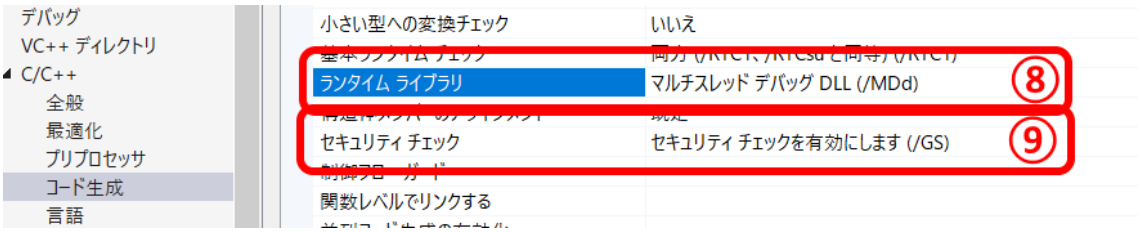
これが何やってるか分からない人はもう一回 C 言語を勉強しなおしましょう(三項演算子だよ)ね。
まぁ、それはともかくこいつは副作用がヤバくてプログラム文中に min(〇,〇)という文字列を

見つけるや否や上の展開をしてしまうのだ。これの何がヤバいって

oremin(a,b)だろうが noumin(米,人参)だろうが問答無用で展開しやがる困った奴なのだ。ちなみにここで「いやでもそれをやっちゃうと min 関数と max 関数使えなくなるんでしょ？」と思われるかもしれないが、心配ご無用!!

C++には安全な min と max がございますのでそれをお使いください。std::min(a,b)と std::max(c,d)というふうに。

次にコード生成です



これはコンパイル時にどのようなコード(ネイティブ機械語)を生成するのかを聞いてきます。意味が分からないと思いますので、気を付けておくべき設定を 2 か所言います。

⑧は、Windows の OS に関連した VisualStudio 標準ライブラリをどうリンクすべきかで機械語が変わってしまうのだ。意味が分からんと思うので、言っておくと⑧は DLL って書いてる奴と、DLL って書いてない奴がある。
で、大抵の場合は **DLL って書いてないほうを選択した方がいい。**何でかというと、DLL はダイナミックリンクライブラリ(動的にリンクするライブラリ)を使用するという意味で、今使用している VisualStudio に関連した DLL も添付して配布しなければならなくなる。

つまり exe 単品では動かない設定なのだ。何でこういうヤヤコシイ事をするのかというと、標準の物を DLL 化しておけば exe のサイズが小さくて済むからだ。しかし最近の潤沢な環境とそれぞれの環境の複雑さを考えると DLL 化はデメリットの方が大きいと言える。

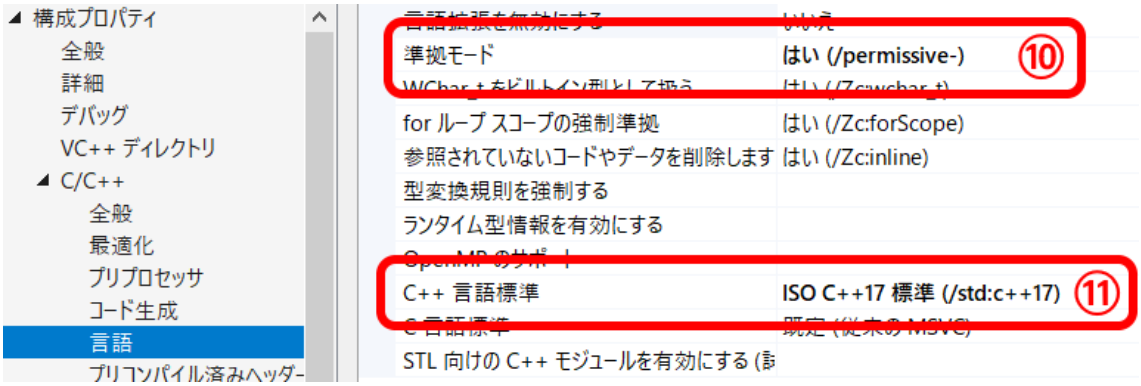
しかし気を付けておかねばならないのが、この DLL がデフォルトという事だ。じゃあ片っ端から DLL じゃないやつにして行きゃいいかというと、そうでもないのだ。何かというと、使用している外部ライブラリが DLL つきのやつでコンパイルされている場合、リンカエラーを起こすのだ。

このため DxLib ではどちらでもリンクできるように細工を施しています。が、世の中そんな気を使うライブラリばかりではないので、自分が使用するライブラリが DLL 系じゃないかどう

かだけは確認しておきましょう(最近は大抵のフリーの便利ライブラリの場合 GitHub でオープンソース化しているため、自前でコンパイルすることになりますが、そのライブラリのコンパイル時に、この DLL 使ってるかどうかの設定は合わせておかなければなりません)

次に⑨の「セキュリティチェック」ですが、こいつは基本的に ON にしておきましょう。
これはランタイム時にバッファオーバーランを検出するとクラッシュしてくれます。え？クラッシュ？やめてよと思った人はまだ甘い。

バグがあるならさっさとクラッシュした方がいい!!事故現場に近い方が直しやすいのです。
はい次は言語について…
正直ココは全部把握してほしい所さんなのですが、プログラミング歴 1 年くらいの人にそれを言うのも酷なのでここでは知らないと妙なバグに遭遇する原因になるかも…な 2 か所



はい、⑩は何かというと C++の標準に準拠するかどうかという事です。意味が分からん人もいると思うので、プログラミング業界の闇…ではないけれども事情と歴史について少しだけ、お話します。

その昔、C++のコンパイラってのは、各社が勝手に作って売っていました。
ところがそれだと問題が浮上するんですね。基本的な文法は C++の考案者 Bjarne Stroustrup さんの設計に則ってはいるんですが、まだまだコンピュータサイエンスが、今ほど成熟していた時代ではなかったため、コンパイラが実際に生成する機械語というのは各社まちまちでした。

生成される機械語が本当に全然違ってて、全く互換性がありませんでした。これだけならまだしも、実務で使っていくうちに当然ながら「言語拡張」(特定の業務に合わせて文法を都合よく変えてしまう)などというものもできてき始めました。

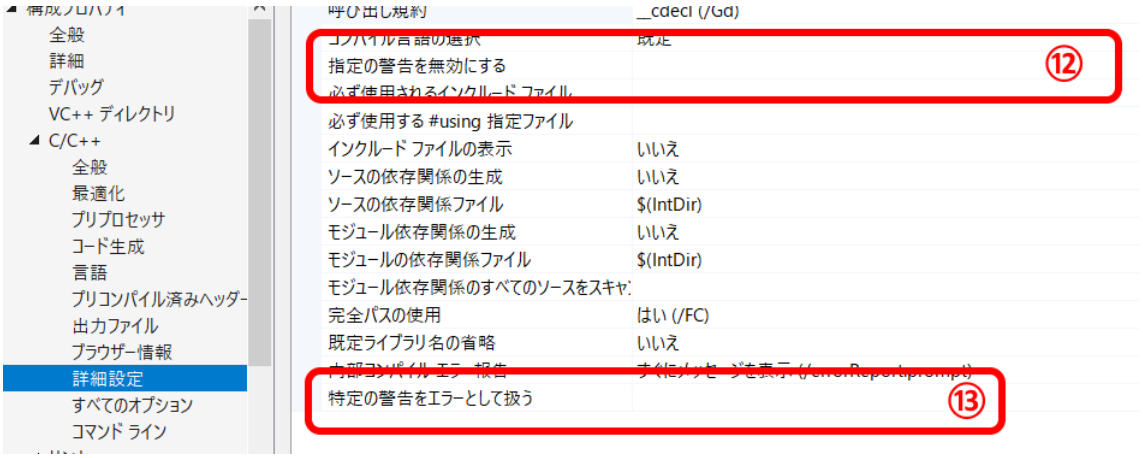
1990 年代はもうそういう会社が勝手にコンパイラを出しまくって、A 社と B 社で文法がかなり違って、これもう別言語じゃんみたいな感じになってました(MS もこの時好き勝手に拡張し

ていました)。
広く使われている言語になった C++において、これでは学習者はたまったものではありません。ということで「C++標準化委員会」というのが発足し、今でも会議が開かれています。MS は自分の所の優位性を保ちたいので、割と最近まで反抗していましたが、流石にそうも言ってられず、とはいえ MS 製品使用者とそれまでの互換性も必要だという事で、こういう部分で選択できるようになっています。

で、「はい」を選択すると、C++標準化委員会が決めている「標準」に合わせるということになります。これを選択すると MS のコンパイラでしか通らないコードというのがぐっと減ります(なくなるわけではない)。基本的に「はい」にしておきましょう。

次に C++言語標準ですが、前にも書いたのと同じです。C++というのは 3 年ごとに上記の「標準」の改正が行われており、基本的には便利な標準ライブラリが追加されていくという流れなのですが、たまに言語そのものが変わる事もありますので、これが「いつの」基準なのかは結構重要です。

最新の(C++20)に合わせよう!!と言いたいところですが、なんかまだドラフト状態なので C++17 に合わせておきましょう。
C/C++の最後に重要なのが「詳細設定」ですね。



見ておくべき場所は 2 か所。
指定の警告を無効にする⑫と、特定の警告をエラーとして扱う⑬です。

一部の人は「動きゃいいんだよっ!!」とエラーはともかく警告を無視しがちですが、プロとして働くことを考えると基本的に警告は 0 にしてください!(エラーは当たり前ですが)。
とはいえ、使用しているライブラリ等の関係上、どうしても警告が消えなかったり、「これを対処してるとプログラムが煩雑になりすぎて、よりバグの温床になりうる」と言った場合は、特定の警告を無視する必要があります。

そういう時に使うのが⑫の特定の警告を無視する。です。

使い方はというと、エラーや警告には特定の番号がついています。

例えばこういうものですが、この警告を消したい場合には C4244 の 4244 の部分を⑫の部分に入れます。

もし複数無視したい警告がある場合には;(セミコロン)で区切ります。もし入れるとしても本当に必要な…せいぜい2～3個にとどめておいてください。

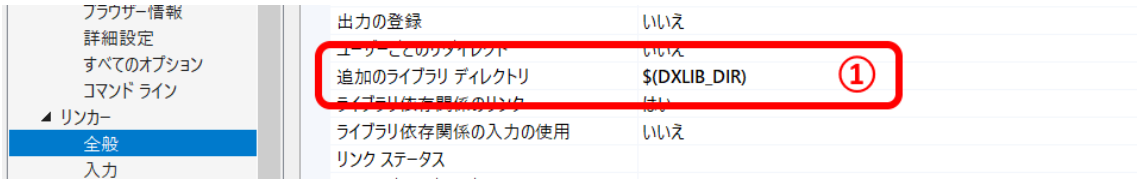
それに対して⑬はその逆ですね。皆さんも見に覚えがありまくると思いますが、警告は出てもどうしても無視しがちです。ところがこの姿勢は見えないバグの温床となって、将来的によくない結果を引き起こします。

なので、特にチーム制作をやるときなどがそうなのですが、チームメンバーにずぼらなやつがいて「あー、あいつは絶対警告は無視するだろうなあ…でもこれを許したらあかんな」って時にこの⑬にその番号を書きます。そうすると無視したくてもエラーとなってしまいますので、対処せざるを得ません。

こちらも⑫と同じで番号を書くだけです。

さて、これで C/C++編は終わりです。次にリンカです。こちらはそれほど多くありません。

リンカ編



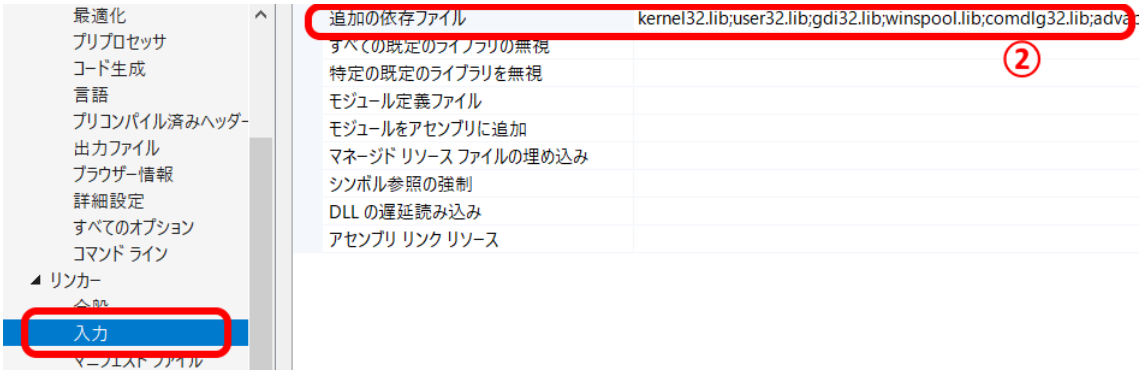
はい、まず何よりもここですね。DxLib など、外部のライブラリを使用する場合は、ここを必ず設定する必要があります。これ DXLIB しか書いたことないから、複数書くやり方が分からない人もいるかと思いますが、それはセミコロンで区切れます。

ここで注意点ですが、

リンクのパスに絶対パスを指定してはいけません

まぁ、これは追加のインクルードディレクトリも同様ですが、可搬性(よそに持って言った時にきちんと動く確率)が酷く低下します。

以前にもお話した、環境変数を使うか「相対パス」で指定してください。相対パスで指定できると言っても、プロジェクトフォルダの下のフォルダか、プロジェクトの一個上＋Lib フォルダ等にしてください。それ以上遠ければ環境変数を使用してください。



次ですが入力→追加の依存ファイルです。まず最初っから色々と入ってますが、これは

絶対に消さないでください！

下手をすると普通の HelloWorld すら通らなくなります。
で、ここは何をする所かというと、詳細は後述しますがアプリケーション(exe)を作る際に、コンパイルとリンクという作業を行います(これを合わせてビルドと言います)。
で、このリンクというときに、obj ファイルやら lib ファイルやらを合成していくんですが、その時に必要な lib ファイルをここに書きます。

たとえば DxLib 以外に Effekseer などを使用する場合はここに Effekseer.lib などを書くことになります。その際にその外部のライブラリには「パスが通ってる」必要がありますのでご注意ください。

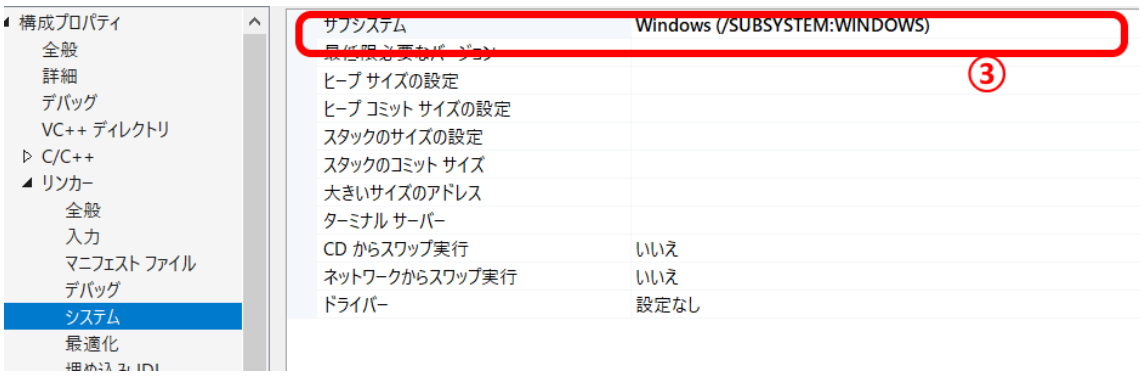
「あれれ～？DxLib ってライブラリじゃないの？なんでここに名前が無いの？」

と思うかもしれませんが(うぜぇ)、ライブラリのリンクはなにもここでだけ行うわけではなく、ソースコード文中のどこかに

```
#pragma comment(lib,"使用ライブラリ.lib")
```

と書くことでそのライブラリをリンクすることができます。まぁ、それはまたあとでお話します。DxLib は初心者向けライブラリなので、自分のヘッダファイルの中にこれを書いてくれているんですね。

次で最後です。サブシステムです



はい、これは exe の元になるシステムが何かという所です。基本的にはコンソール(CONSOLE)か、ウィンドウズ(WINDOWS)かです。

コマンドラインが出てくるのが CONSOLE。出てこないで自分の設定したウィンドウだけが出てくるのが WINDOWS です。

これ知らない人が結構多くてびっくりするんですが重要なので、しっかり把握しておきましょう。
ひとまず把握しといた方がいい設定はこのくらいなので次行きます。

VisualStudio のデバッグ機能

うーん。意外と VisualStudio のデバッグ機能についてきちんと知らない人が多いみたいなので、んで、これ知ってるとバグ起きた時の対処の時間が大幅に短縮されるので、簡単なはずだけど、もしかしたらちょいと難しいかもしれないけど、パパパッと説明しますね。

ブレークポイント

デバッグの基本のブレークポイント…これ、正しく扱えてますか？当然ながらブレークポイントをソースコードの特定の行に置けば、そこでコードの実行は中断されまぁす!

ブレークしてからですが、そこで F5 を押せば中断したところから再実行されます。

ステップ実行

F10 を押せばステップ実行と言って、1 行ずつ進めることができます。ただし関数の中には入らず、関数の呼び出し行で F10 を押せばその関数が実行されたことになって、次の行に進みます。

ここで F11 を押せば関数の中に入ります。ただし常に F11 を押してるといちいち関数の中に入るので面倒ではあります。例えば

Function(Func1(),Func2());

こんな関数が書いてある行でF11を押せば関数の出入りを3回繰り返すのでウザいです。こういう場合は、この関数が書いてある部分にブレークポイントを置いておいて、一旦止めておいて、怪しいと思う関数の中にまたブレークポイントを置いてF10実行した方がいいです。

ちなみにF10をステップオーバー実行、F11をステップイン実行と言ったりします。いちおうどちらも「ステップ実行」と言います。

ここまでは知ってる人がほとんどなんじゃないかなと思います。知らなかった人は覚えてね。
で、次は半数くらいの人がびっくり!!するだろう

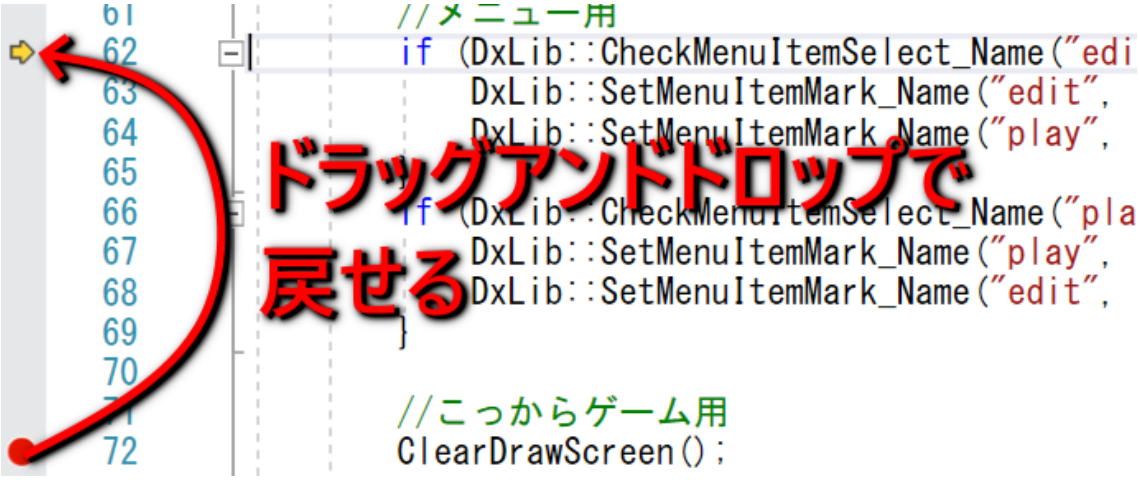
デバッグカーソル(?)をドラッグできちゃう

知ってた？
あっ、そう!!!まぁ、それはそれとして解説しよう。

まず、どっかでブレークポイントしかけて、止まってる状態にしてみよう。そしたらデバッグカーソルが黄色で表示されているはずだ。

ここに対して、自分のマウスポインタを合わせてみよう。

驚くべきことに…こいつをドラッグアンドドロップできるのです。嘘だと思うんならドラッ



グアンドドロップしてください。
ドラッグアンドドロップで戻せるし、逆に進めることもできます。あと、目的の行でコントロールキーを押してると黄色い矢印ボタンが出るので、それを押してもらうとデバッグカーソルがそこに飛びます。

どういう時に役に立つのかと言うと、F10 でステップ実行して、うっかり目的の行を素通りしちゃうことがありますが、これで戻せばいいわけですね。

ただし注意点として F10 で一度実行されてる処理は、カーソルを戻しても「なかったコトに」はできません。そらそうだ。

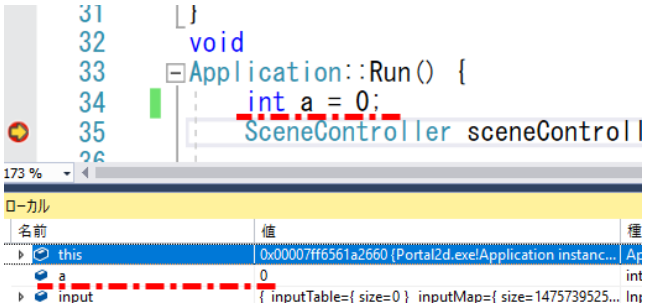
この機能を用いると、**デバッグ情報の信頼性が著しく低下する**ので、使うのはなるべくやめておいた方がいいとは思います。でも便利なので、デバッグの信頼性が下がっても大して問題ない時は利用してもいいと思います。

結局使う人のスキル次第。よくこういうのは「教えない方がいい」と言う人がいますが、知ったうえで使わないのと、知らないから使わないのでは意味が違います。知ったうえで、その危険性は認識しておくべきだと思います。

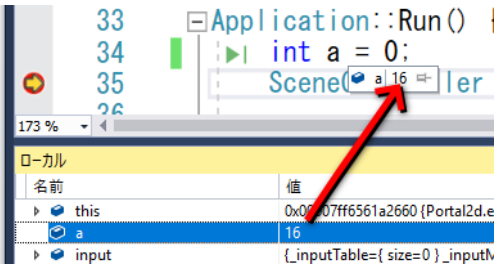
それではそういうデバッグの信頼性を下げるけど、便利な機能をもう一つ…

デバッグ中に値の変更もできちゃう…できちゃう

例えば以下のような場合



当然ながら変数 a の中身は 0 です。この中身を変更できるでしょうか？できますんよ。ローカルもしくはウォッチで a を探すと、名前と値の表がありますが、この「値」に適当な値…16 でも入れてあげましょう。

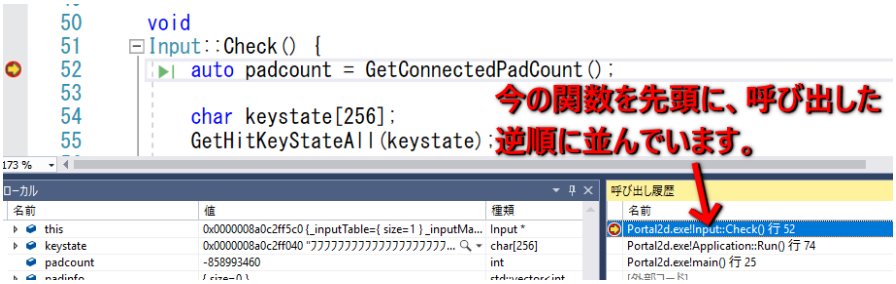


こんなことできます

ま、これもお分かりのようにデバッグの信頼性を損なうので、ホントにテスト的な事をやる目的以外には使用しないようにしましょう。

呼び出し履歴(コールスタック)

まま、これは分かり切ってると思うんで、軽く流しますが、例えばどこかでブレークポイントさせるか、どこかでアサーション起こすと、当然処理が止まるのですがその時に呼び出し履歴(コールスタック)ウィンドウを表示させると



で、右下のコールスタックの各行をダブルクリックすると、その関数に飛び、さらにはその関数呼び出し時の周囲のローカル変数なども参照できます。非常に重宝します。

発展的ブレークポイント

さて、再びのブレークポイントですが、使い方をもう一歩進めると、非常にバグの検出の役に立ちます。

条件付きブレークポイント

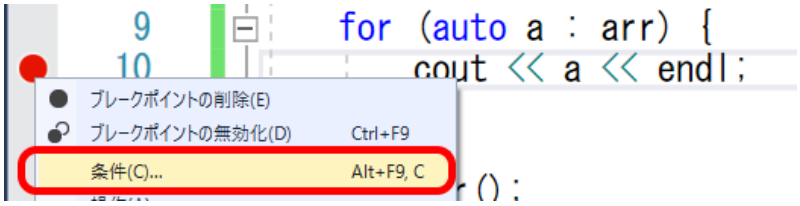
例えばこんなコードを考えてみる。例えばだよ？このコードに対した意味なんてないよ。

```
#include<iostream>

using namespace std;

int main() {

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

	for (auto a : arr) {
		cout << a << endl;
	}

	getchar();
```

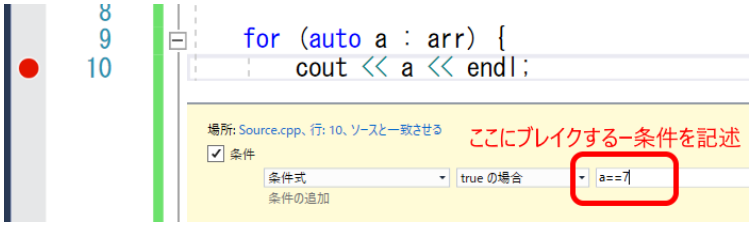
```
        return 0;
}
```

さて、なんかしらの理由で、要素が 7 の時の状況を知りたいとする。君だったらどうするだろう？7 回ループを回す？それじゃあもしこれが 1298 ループ目の状況を知りたいときだったらどうするんだろう…。

まぁ、現実的じゃないね。そんなときに役に立つのが条件付きブレークポイントだ。まずいつものようにブレークポイントを仕掛けてみる。そして、ブレークポイント上で右クリックしてみる。



そして「条件」という項目をクリックすると…



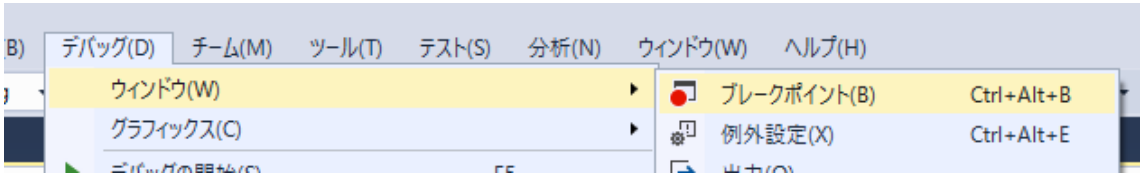
こんなのが出てくるので、ブレイクさせたい条件を記述する。そうすると条件が一致した時だけブレークするため、特定の条件で止めたいときは重宝します。ただし副作用として、条件ブレークを置いている個所は若干処理スピードが落ちます。まぁバグった時にしか使わないから問題ないと思います。

データブレークポイント

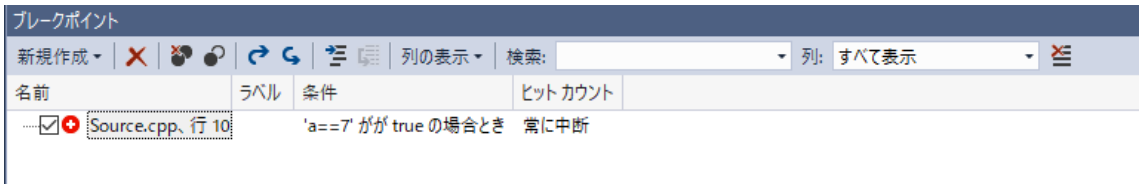
それでは次に、ちょっと難しいのを紹介します。「データブレークポイント」です。
例えば、なんかの値が変化した瞬間を捕まえたいときってありますよね？そういう時に役に立つ機能です。

よくあるトラブルとして、値を代入した覚えもないのに値が変わってるとか、あとは特にグローバル変数とかを使用してる場合ですが、ありとあらゆるところから変更されるため、誰が犯人か分からない…と言うのがあると思います。そういう時に役に立つのがデータブレークポイントです。

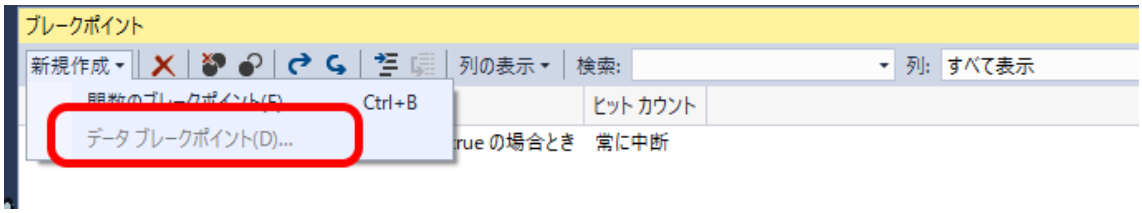
まず、デバッグ→ウィンドウ→ブレークポイントをクリック



はい、そうすると下にこんなウィンドウが出てると思います。



で、この新規作成をクリックするとデータブレークポイントって項目が出てきます。なお、これはデバッグ実行時しか有効ではないので、デバッグしてないときは

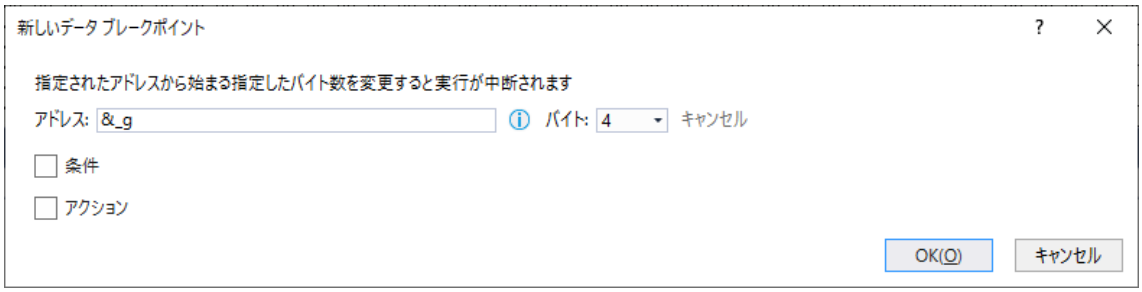


こんな感じで使用できません。なんでかというと、データの置き場所(つまり変数のアドレスなど)というのは、実行時にしか確定しないからです。

で、どうするのかというと、たとえばグローバルに
int _g=10;
なんてのがあったとします。

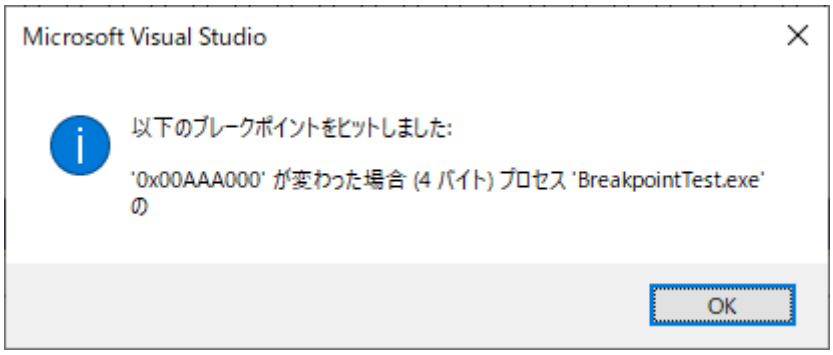
で、誰かがどこかでこれを変更しているのだが、誰だかどこだかわからない。それを知りたいとき…まぁ、実行時にしか分からないので、main 関数の最初にでも普通のブレークポイントを置いて止めます。

そうしたら先ほどのデータブレークポイントが使えるようになってるので、選択します。そうすると



こういう画面が出てきますので、アドレスの部分に対象となる変数のアドレスを入れます。くれぐれも間違えないようにしてほしいのですが、アドレスです。なので上の例では _g では

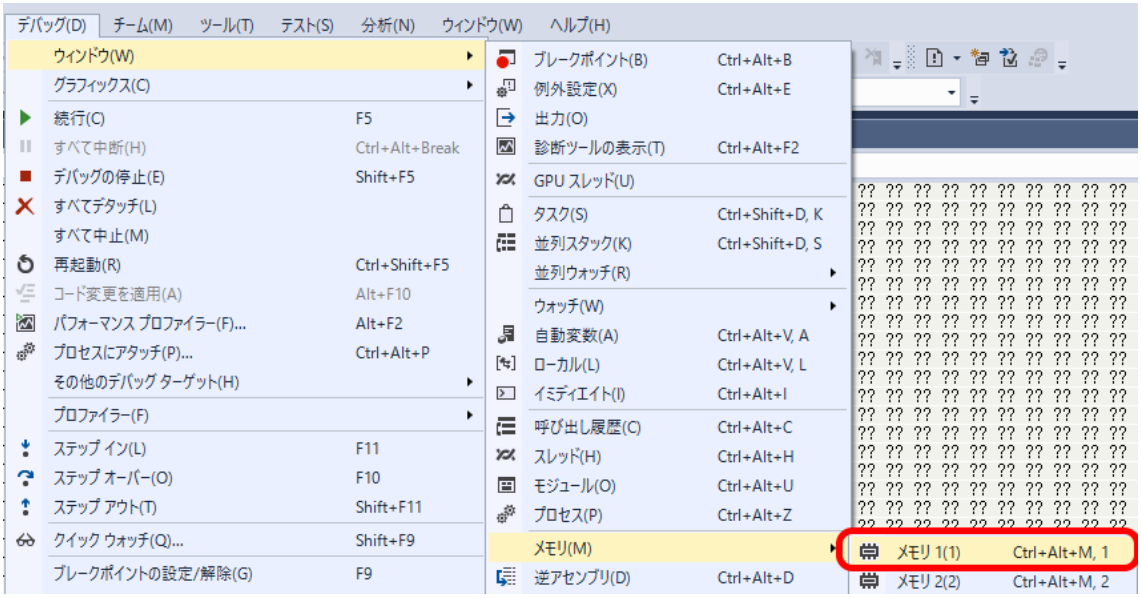
なく、&_g としています。そうすると、変更されたタイミングで



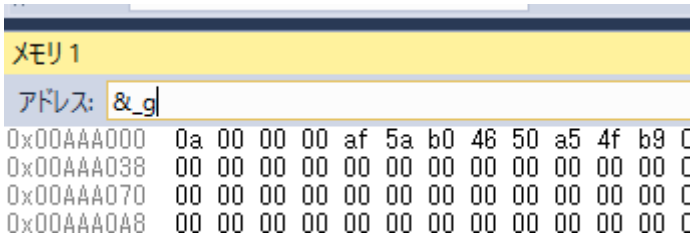
みたいなメッセージボックスとともに処理が中断されます。
まぁ、これの便利さは、C++でクソみたいなバグに悩まされないと、このありがたみは分からないともいますが、このクラスで開発するなら、まぁそのうちお目にかかれます。

メモリの中身をみる

これまたデバッグ時にしか見れないものですが、



こんな感じでメモリを選択すると現在のメモリの状況が見れます。例えば先ほどの&_g のアドレスを入れると…



メモリの中…アドレスさえ渡してあげれば、現在のメモリの状況を 16 進数で見ることがで

きます。これが何の役に立つのかというと、バイナリファイルの読み込みや、なんかしらのバイナリ計算の結果やアドレスの状況を知るために使えます。まだまだ高度すぎるかもしれませんが、そのうち役に立つと思います。

出力ウィンドウ

最後に忘れちゃいけないのが「出力ウィンドウ」です。これ意外と知らん人が多かったから描いておきます。
VisualStudio には「出力ウィンドウ」というものがあり、最初からウィンドウがある事も多いのですが、出てないときは

デバッグ→ウィンドウ→出力

出力ウィンドウを開く事ができます。コンソール対象時はどうせコマンドラインに文字出力できるので、そっちを使えばいいのですが、ウィンドウアプリを作るときはコマンドラインがないので、こちらに出力しましょう。

文字列の出力には
OutputDebugString
という関数がありますのでそれを利用します。ただしこの関数の役割は「文字列出力」のみですので、printf や DrawFormatString のようにフォーマット文字列を出力する機能はありません。どういうことかというと、数値などを出力することはできないということです。

このためなんかしらの数値情報を出力したければ sprintf や stringstream のお世話になる事になります。これも知らん人が多いっぽいので、後々解説します。

あと、DxLib やら他の外部ライブラリは結構情報をここに出力しているので「なんか知らんが壊れた」みたいな時はまず出力ウィンドウになんかログが出てないか確かめておきましょう。

あと最後に注意点ですが、この出力ウィンドウへの出力は結構コストがかかりますのでデバッグ時以外には利用しない事と、毎フレーム固定で何行も出力するのはやめましょう。処理落ちの原因になったりします。

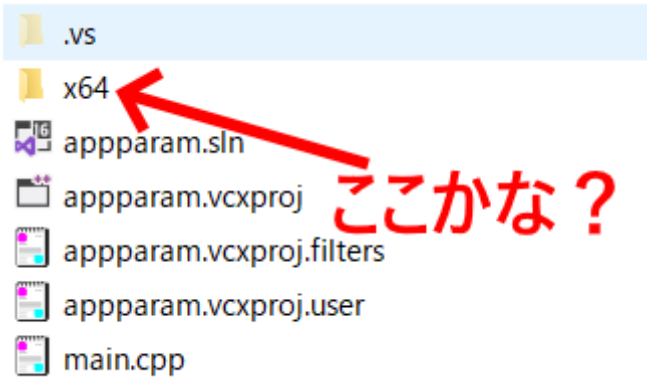
他にもスレッドだのなんだのありますが、それはもうちょっと先(就職してから実際のややこしいバグに悩まされてから)でいいでしょう。今回はこのくらいで十分だと思います。

ビルド…コンパイル/リンク

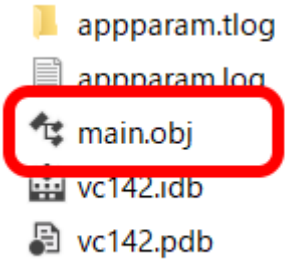
多分 2 年生になったばかりの皆さんはプログラム書いたら→デバッグ実行～ってな感じでやってると思います。それはそれでいいです。全然問題ないしプロがやってる事もあまり変わりありません。
それはそうなのですが、もう 1 ランク先に行くには…というかわけが分からないエラーに対応するためには、あのデバッグ実行ボタンを押した時に何が行われているかを知っておいた方がいいと思います。

まずそもそもさぁ…exe の場所、あんだけど、見ていかない!?

ああ～、いいっすねえ!!と条件反射で答えたい所さんですが、場所分かります？
慣れない人は cpp の場所を基準に探しに行くから見当たらないかもしれません…。



ココにある事もありますが、たいていは違います。この中にあるのは後述する obj ファイルです。



一旦はこの obj ファイルを頭に入れておいて、別の場所に行きましょう。cpp のまだ上のフォルダに行くとソリューションフォルダがあって、その中にさっきと同様に x64 フォルダ→Debug フォルダがあると思います。

そこまで行くと漸く exe が見つかると思います。

基本的に僕の授業はソースコード提出はさせません。採点でソースコードなんて見てらんないからです。何人いると思ってんだよ……。

ソースコードがキレイか汚いか、どうやったら改善するかは

「リーダブルコード」と「ゲームプログラマのためのコーディング技術」を読んでおこう。
自分で改善する気が無ければどっちみちコードなんてきれいにならん。でもコードが汚い奴はゲーム業界には行けないと思ってください。

まぁ心配だったら見せてくれてもいいけど、それは授業中にしてくれよ～頼むよ～～。

ともかくソースコードは見ないので、exe と、その exe がまともに動くためのリソースを提出するようにしてください。

exe は見つかりましたか？よろしい。

では次にコンパイルとリンクについて…お話します。

コンパイルとは…

さて、皆さん、コンパイルは知ってますかね？「エラーを報告するもの」ですか？違います。皆さんが記述するプログラミング言語を機械の言葉に変換するものです。この機会の言葉を機械語、マシン語と言います。

機械語

```
55
8B EC
81 EC E4 00 00 00
53
56
57
8D BD 1C FF FF FF
B9 39 00 00 00
B8 CC CC CC CC
```

「俺にとっては C 言語も機械の言葉だよ!!!」と思ってる人もいるかもしれませんが、違います。機械語はガチ機械の言葉です。

それでも昔の人は直接これを書いてたりしてたというのだから、まぁ、やべーなと思いますが、これよりもうちょっとマシなのが、「アセンブラ(アセンブリ言語)」というやつです。

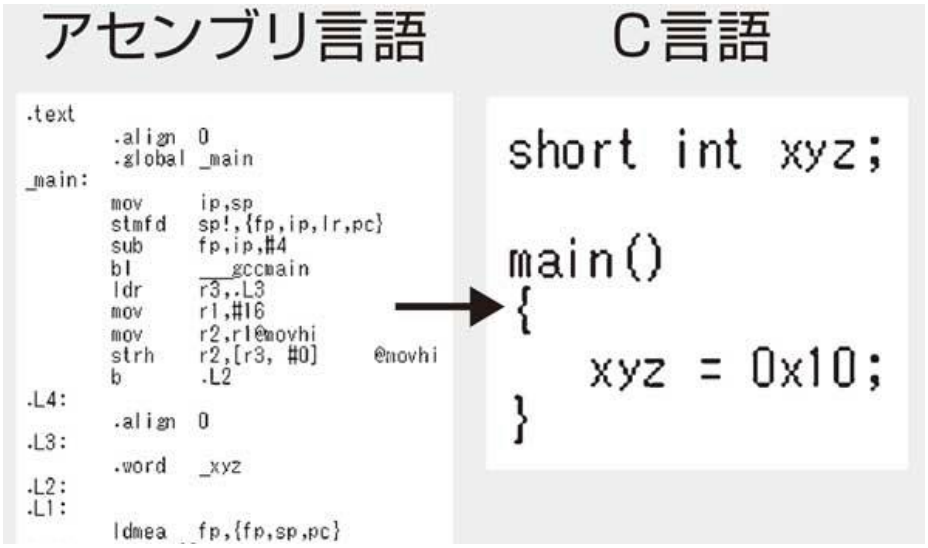
アセンブリ

```
push        ebp
mov         ebp, esp
sub         esp, 0E4h
push        ebx
push        esi
push        edi
lea         edi, [ebp-0E4h]
mov         ecx, 39h
mov         eax, 0CCCCCCCCh
```

ファッ!?ふざけるな‼なんだこれは、これが…言語か？言語なんですよねえ、これが。push はスタックに乗っける、mov は代入、sub は引き算ですね～

いやいや、ちょっとまて、ちょっとまで…

となるので、もうちょっとマシになったのが C 言語とかなんですね。



この流れを知ると、機械の言葉に見えてたＣ言語でもまぁ～～～だマシに思えてきませんかねぇ～

まま、それはさておき、コンパイラってのはＣ言語を頑張ってさっきの機械語に変換する作業というのは分かっていただけたかなと思います。

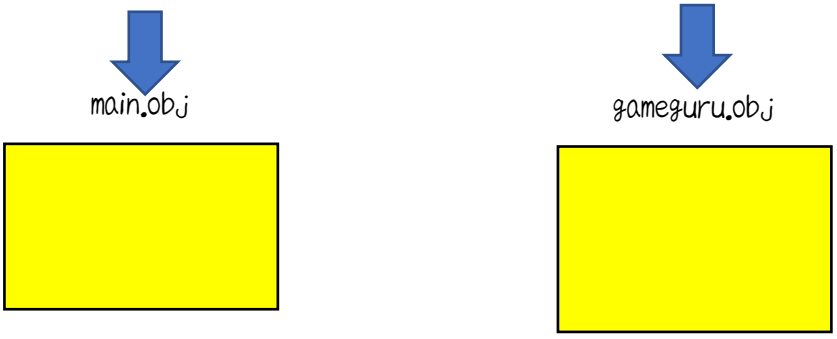
じゃあ、コンパイラだけでＯＫか？というと、そうはいかないんですよね。使える exe にするた

めにはリンク…という作業が必要になってきます。

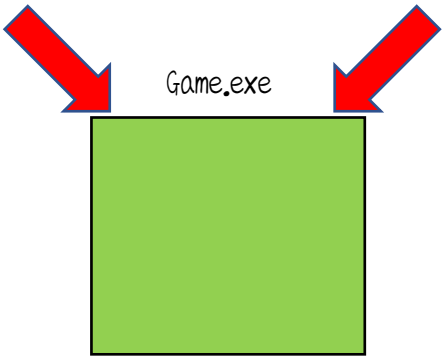
リンクとは…

ゼルダの伝説の主人公じゃないですし、データ型でもありません。簡単に言うと、exe の元になる要素をギュッと固めて、起動可能な exe を錬成する作業です。なんでリンクというのかというと、cpp を変換すると obj ファイルというのになるんですが、ご存知の通り一つのプロジェクトに cpp ファイルが一つとは限りません。
複数あります。
これを固めるのです。ついでに lib ファイルも一緒に固めてしまいます。1つしかファイルがなければ obj がそのまま exe の役割を果たすのですが、ゲーム作る上に置いて main.cpp しかないってのは稀なので、複数ファイルがある場合を考えます。
さて、ここに見えます main.cpp と gameguru.cpp ですが、これはコンパイルするとそれぞれ obj

ファイルというものになります。

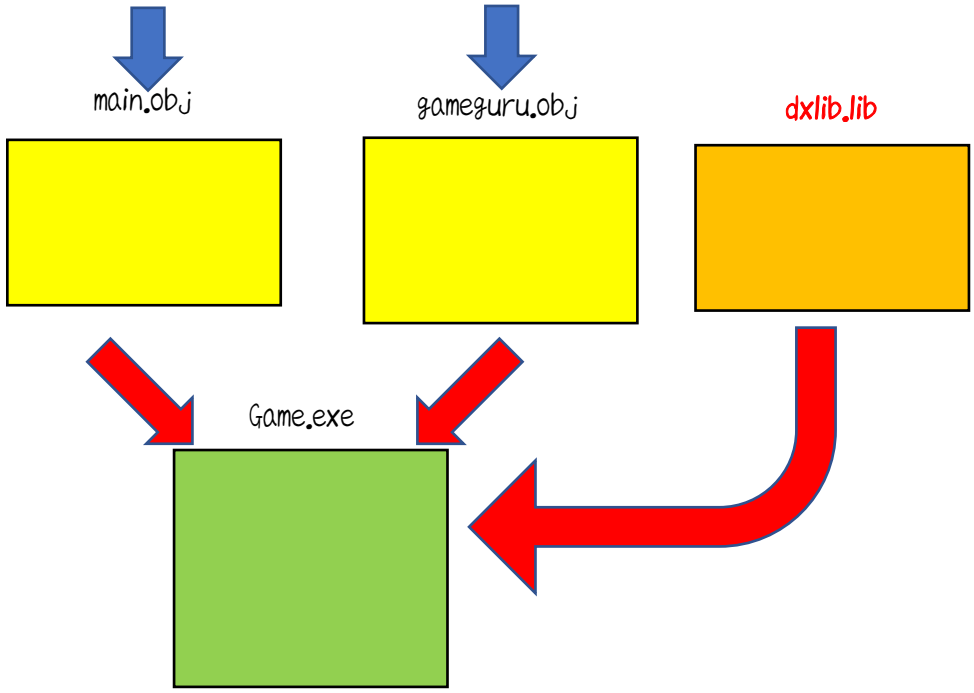


これらはバイナリです。なお obj フォルダに入っているはずなので、よかったら見てみて下さい。でもこれでは**バラバラの obj ファイルなので「リンク」という処理が必要**になります。簡単に言うと、2 つのバイナリを合成する作業です。特に何も考えずにただ合成されます。



こうやって動く exe ができているのです。

もうちょっと言うと、lib をリンクしている場合は



こんな感じに obj と一緒くたにされて exe ができあがっています。

ここまでは分かりますね？ちょっとこの事を頭に置きながら、この後の話を聞いてください。
直接は関係ないようですが、最終的に関係あります。

#include 文について

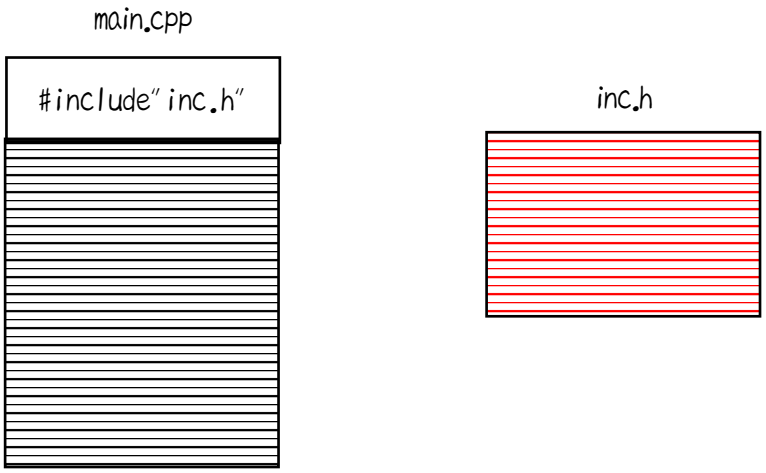
はい、#include 文についてなんて、今更言われなくてもわかってるズェ…馬鹿にしすぎだズェ…と思っているかもしれませんが、分かってない人も結構いたりするので、あえて、軽くお話しいたします。

#include のしくみ

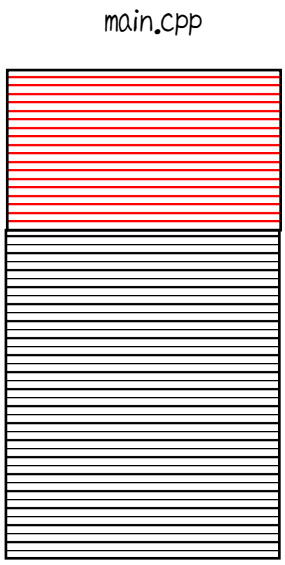
#include…先頭にシャープがついてますね？ということはこれは#define の仲間で「プリプロセッサ」と呼ばれるものです。C 言語で頭に#がついてたら「プリプロセッサ」だと思ってください。
では「プリプロセッサ」とは何なのか？ご存じでしょうか？

pre-processor つまり、プロセッサ前の処理…コンパイル前に行われる事前処理のことなのです。#define もそうですね。コンパイル前に実際の値や式に変換されるのです。

では#include は何なのかというと、こいつは#include<>もしくは#include""で囲まれた部分に書いてあるファイル名を検索し、そのファイルの内容をコピーして、その#include と置換するのです。



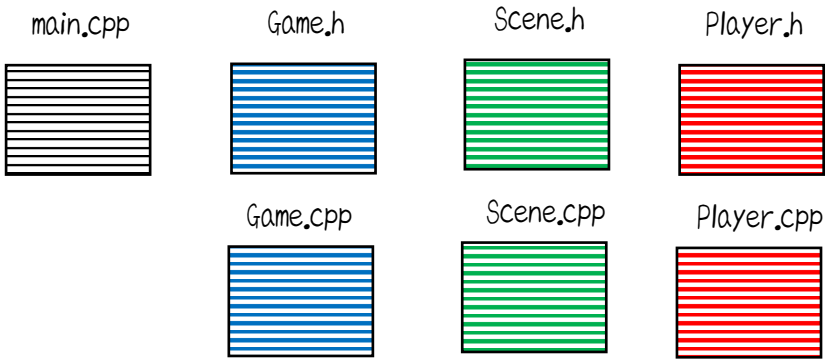
例えば、main.cpp が inc.h をインクルードしていたとします。そうすると#include"inc.h"の部分が inc.h の内容そのものに置換され…プリコンパイル後は



こうなります。なのでヘッダファイルに色々置きすぎると、いろんなところでこのコピーが生成されプログラムサイズがでかくなります。さらに言うと、この後にお話しする翻訳単位(要はコンパイル対象)の関係上、ヘッダ側に実体が混ざっている場合に面倒なことになります(おなじみリンカエラー)。
さらに言うと、ファイルを複数のファイルに分割したとするとややこし～い問題が発生します。

インクルードガード

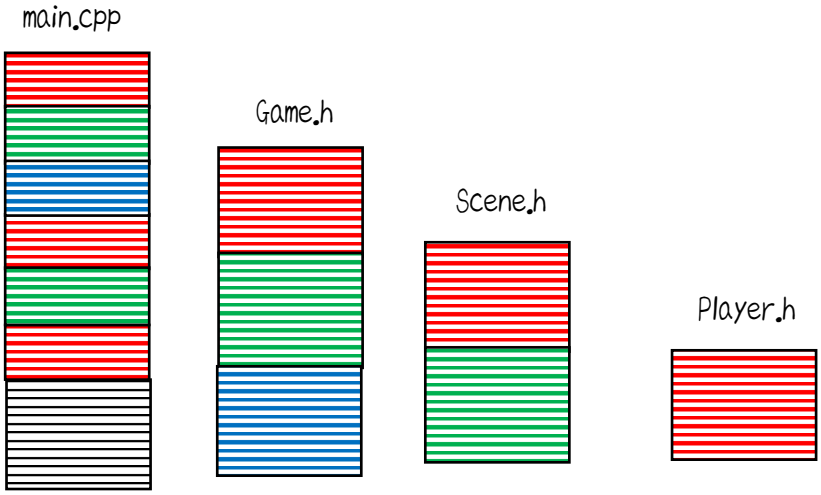
main.cpp と Game.cpp,Game.h,Scene.cpp,Scene.h,Player.cpp,Player.h があったとします。



あ、cpp 書いてますが、今回の説明はどちらかというとヘッダ寄りなので、cpp との関連におけるややこしさは次の「リンカ」の話に回します。ヘッダに注意を向けてください。
よくあるパターンですね？ここで main.cpp は Game.h,Scene.h,Player.h をインクルードしているとします。

さらに、Scene では Player を操作するため(メンバに持つため)に Scene.h が Player.h をインクルードしているとします。

同様に Game.h も Scene.h をインクルードしているとします。そうなるとプリプロセス後にどうなるかというと



ごっつい極端な例ですが、こうなります。前にも話したように#include はヘッダの中身をそのまま置換するため main.cpp がとんでもないことになっています。
じゃあ、main.cpp は Game.h だけインクルードすればいいじゃない!と思われるかもしれませんが、これが確かに3～4個のファイルなら管理できるでしょうが最終的に何十、下手すると百個くらいの(場合によってはもっとある)構成になったときに、管理できるでしょうか？

これは理論上管理できないのです。組み合わせ爆発起こしますし。

ということで考え出されたのがインクルードガードです(とはいえこれも廃れますが)。

役割は「翻訳単位内で 2 回以上インクルードされないようにすること」です。
やりかたはいたって簡単。

```
//Game.h
#ifndef GAME_H_INCLUDED
#define GAME_H_INCLUDED
        中略(ヘッダの内容)
#endif
```

はい、2 回目以降は#ifndef と#endif で囲まれた部分が無視されます。理屈はお判りですかね？
まず、#ifndef の解説をしましょうか。
#ifndef は#ifdef の否定形で、#if not define を意味しています。つまり「もし define されていないなら」if と endif で囲まれた部分を解釈しコンパイルするという意味です。

となると、例えばこれが初回#include されたときは当然 define されてないので、#ifndef の中に入ります。
ただし 2 回目以降は既に GAME_H_INCLUDED が定義されているため#ifndef 以降は解釈されなくなります。

これがインクルードガードです。2010 年くらいまでこのやり方が使用されていたのですが、その後にもうこれが決まり文句になっていたためファイル先頭に#pragma once と書くようになりました。これ 1 行で先ほどのインクルードガードと同じ意味になります。

最近は VisualStudio でヘッダファイルを作ると自動で入るようになっています。もし別のエディタ(sakura など)で作ったヘッダファイルを使うときには気を付けましょう。

しかしインクルードにおける問題がこれで終わったわけでもない。もう一つあるのが相互依存だ

相互依存への対処とプロトタイプ宣言(前方宣言)

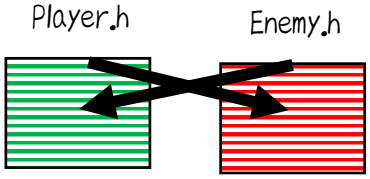
色々なゲームを作っていると、あるファイルとあるファイルが相互参照していて#include が循環参照してしまう事があります。例えば Player と Enemy がそれぞれをメンバに持っているような感じですね。C++の話はまだなので C 言語における構造体の話をしますが…

```
//Player.h
struct Player{
	Enemy* enemy_;//敵の情報
};

//Enemy.h
struct Enemy{
	Player* player_;//プレイヤーの情報
};
```

さて、こういう場合、相手の構造体を参照しようと

```
//Player.h
#include"Enemy.h"
struct Player{
	Enemy* enemy_;//敵の情報
};

//Enemy.h
#include"Player.h"
struct Enemy{
	Player* player_;//プレイヤーの情報
};
```

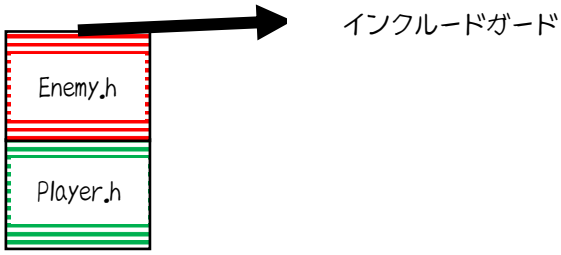
なんてやると



このような形で循環参照してしまう。もちろんこの場合にも「インクルードガード」は有効だ。
だが、問題が残ってしまう。

Player.h を起点として考えてみよう。まず Player.h が起点なら#include 文により Enemy.h の内容が Player.h の先頭にコピーされる。

そして、Enemy.h の先頭でも#include"Player.h"をしているが、インクルードガードによって阻まれます。



めでたしめでたしに限りなく近い何か。…とはいきません。

```
#include"Player.h"//←インクルードガードにより無効化
struct Enemy{
	Player* player_;//Player って…誰？何この型！？
};
```

だって、Enemy 構造体の定義の時点で、構造体 Player が見えないのだから…こういう時に役に立つのが前方宣言(プロトタイプ宣言)です。

もし構造体の持ち物が実体でないならばポインタ(4 バイト、64bit の場合 8 バイト)に必要なバイト数だけ確保すればいいため、型の中身が分からなくてもいいのです。ここでの約束事は「こういう名前の型が存在する。それは後で分かるから、お前は黙って 4 バイト(8 バイト)確保しとけ」という意味になります。

さて、プロトタイプ宣言ですが、この場合は

```
struct Player;//Player という名前の構造体があるよ！内容は後で分かるよ！
struct Enemy{
	Player* player_;//中身はわからんけど、４もしくは８バイト確保しとくわ
};
```

と書いておきます。ただし、これには型情報がまったくないため、メンバの呼び出しはできません。メンバを呼び出すのは Enemy.cpp 側でやってください。Enemy.cpp 側で#include"Player.h"をやっても、循環参照になることはありませんので大丈夫です。

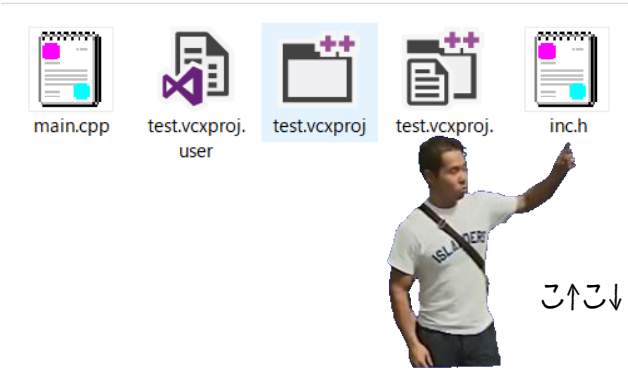
まま、それはともかく次の話です。

#include<>と#include""の違い

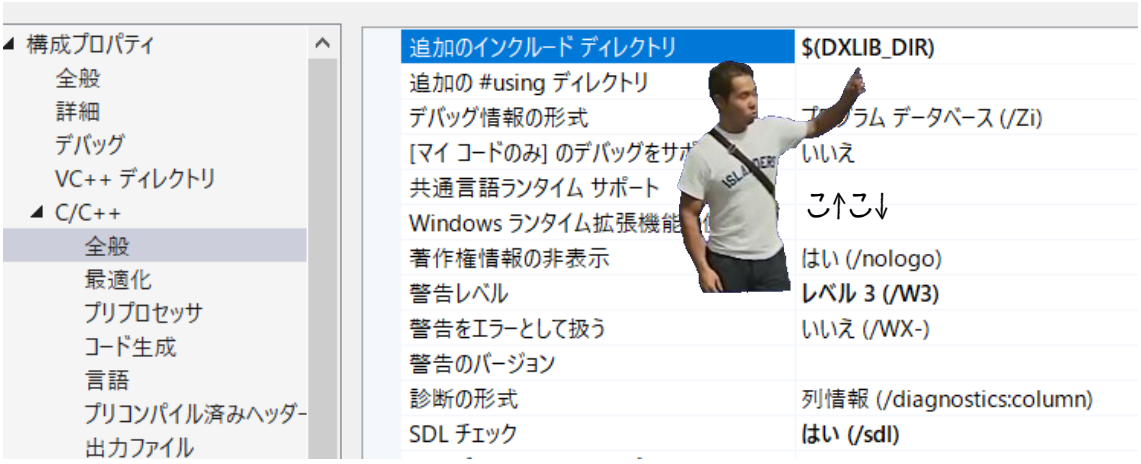
#include はと""両方でインクルードするファイルを指定できますが、違いは分かりますか？
そんなに難しくはないのですが…簡単に言うと「検索場所が違う」これだけです。

#include"〇〇"…その場を調べる
#include<〇〇>…設定されている場所を調べる

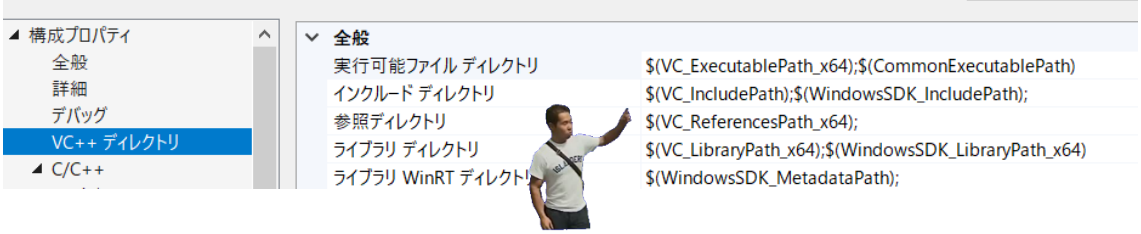
となります。""の例を見せますね？例えば main.cpp で inc.h を検索するといった場合



となります。main.cpp がいるフォルダと同じフォルダから検索するんですねぇ。次に<>の場合はどうかというと、プロジェクトの設定を開いてください。で、追加のインクルードディレクトリというのがあると思いますが、



ここになります。で、ここだけで終わるかというと、そうではなくて



ここも含まれます(ここは stdio などの基本ヘッダが入っている場所です)
<>で指定すると、ここをもとに検索するという事です。

ヘッダ側に関数の実体や変数の実体を置いちゃダメな理由

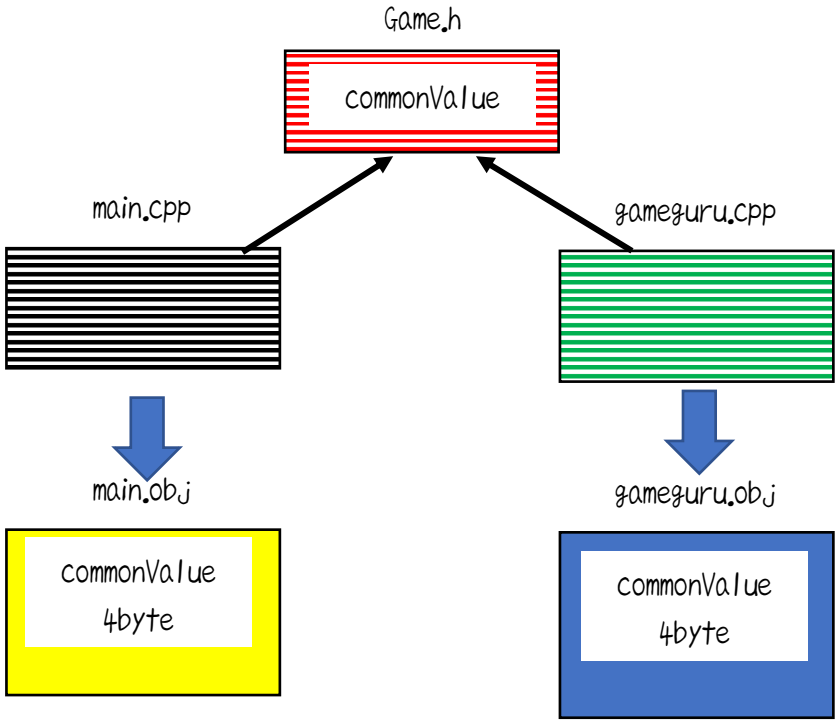
もし、2つの cpp が同じヘッダファイルをインクルードしていたとします。当然それはあり得ますよね？

これがただ単に型の定義やプロトタイプ宣言だけならば問題ないのですが、**関数や変数の実体(関数なら中身の処理まで書くこと、変数なら通常の宣言があること)**があるという事は、

「その名前でメモリを確保する」という事になります。

「それの何がアカンねん!!」あかんのですよ。どういうことか説明します。分かりやすいように変数が、共通のヘッダ Game.h 内で int commonValue;と宣言されていたと仮定します。

いいですか？これを宣言するという事は"commonValue"という名前で 4 バイト確保するという意味になります。



ここまではうまくいくのです。完全に別物として動いていますから。ところがこれを合成しようとすると問題が生じます。何故なら"commonValue"で 2 か所のアドレスにメモリが確保さ

れているからです。

リンカは「え？いや、お前この名前がぶっとるがな。どっちやねん!!はっきりせえ!!」と言って、リンカエラーを起こします。

そういう場合はどうするのかというとご存じ extern 修飾子をつけます。これの意味は「この名前の変数がどこかの obj(cpp)にあるから、それを探して使え」という意味になります。
例えば今回の場合などは Game.h には
extern int commonValue;
として宣言しておきます。しかしこれだけだと今度は「実体がないです」と言ってまたリンカエラーを起こします。めんどうですね。

そこで実体を main.cpp にも同じ名前で宣言します。
これによってメモリ上は main.cpp(main.obj)内に確保されるが、そのメモリを commonValue という名前として gameguru.cpp(gameguku.obj)も使えるようになるというわけです。

ここで覚えといてほしいのは extern 宣言しても本体は存在してなくて、必ずどこかの翻訳単位に1つだけ本体を宣言する必要があるという事です。

これは関数に関しても同様です。関数であってもメモリを使ってどこかに配置されますので、本体は一つだけ、他の翻訳単位でも使いたければ extern を使って指定をします。

なお、同様の働きをするものとして static がありますが、これは意味が決定的に違います。
static int commonValue;
と宣言をすると、これ自体が本体になります。メモリも確保されます。ではなぜ同じような挙動をして、さらにリンカエラーを起こさないかというと staitc には重要な性質がありますそれは

「この名前の変数はこのアプリケーション内でひとつだけ」

というわけです。ただこの場合、リンカの合成順序によって、main.obj 側なのか gameguru.obj 側なのかは分かりません。別に支障はないですが気持ち悪いので static をこういう使い方しないほうがいいです。
extern とは明確に意味が違いますし、その意味を理解しないまま static を使うとまぁ、碌な事にならないからです。明確な理由がない限り extern を使って、本体はどこかの cpp に一つだけ置きましょう。

プリプロセッサについて

プリプロセッサとは、あれだよ。頭に＃がついてる一連のアレだよ。ちょっと前に＃include のところでも学びましたね？

コマンドラインについて

新 2 年生はコマンドラインを使った経験もあまりないと思います。そもそもそんなのがある事すら知らんかった!って人もいるかもしれませんが、実は結構使います。

色々と理由はあるんですが、主な理由は
コマンドラインの方が手っ取り早いことがある
バッチファイルを作るため
コマンドラインでしか動かないアプリケーションがあるため

主なコマンド

Git について

Git については別紙を用意してるんでちょっとそっちを見てもらうとして…