

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	4
習慣について	5
戦略について	6
技術へのアンテナについて	9
授業の流れについて	9
環境とか VisualStudio とかコマンドについて	10
環境変数	10
VisualStudio のプロジェクト設定	17
プロジェクト設定の前に…	17
Debug と Release	17

32bit と 64bit について.....	18
プロジェクト設定.....	21
C/C++編	22
リンカ編	28
VisualStudio のデバッグ機能.....	30
ブレークポイント.....	30
ステップ実行	30
デバッグカーソル(?)をドラッグできちゃう.....	31
デバッグ中に値の変更もできちゃう…できちゃう.....	32
呼び出し履歴(コールスタック).....	34
発展的ブレークポイント.....	34
条件付きブレークポイント.....	34
データブレークポイント.....	35
メモリの中身を見る.....	37
出カウインドウ	38
ビルド…コンパイル/リンク.....	39
まずそもそもさあ…exe の場所、あんだけど、見ていかない?	39
コンパイルとは…	40
リンクとは…	42
#include 文について	43
#include のしくみ	43
インクルードガード.....	45
相互依存への対処とプロトタイプ宣言(前方宣言).....	46
#include<>と#include""の違い	49
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	51
プリプロセッサについて.....	53
#include.....	53
#define	53
コマンドラインについて.....	54
基本コマンド	55
就活の時に意外と役に立つ tree の活用法.....	56
コマンドラインでも変数、関数、ループが使えるぞ?	57
ループ	57
コマンドラインで使える便利系ソフト.....	57
Git(GitHub)について	58
概要	58

Git 本体について.....	59
TortoiseGit について.....	60
GitHub について	61
テキストファイルとバイナリファイルについて...	62
改行コードについて.....	63
ごたくはいいからとっとと使おうぜ!!.....	64
Git の基本的な利用方法.....	65
クローン	65
ソースコードの変更.....	66
コミット	66
プッシュ	66
チームメンバーを招待.....	66
GitHub プロジェクトについて	68
課題①	68
プログラミングテクニック.....	68

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…まあ、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わかりません。

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、まあぶっちゃけた話、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ム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ムを目指している』という体(てい)で授業しますし、お話しします。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、まあ何かしらの事情があるんでしょう。とはいえ、高学費と、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なる 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方は、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んです。今のうちに言っておきます。

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。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。

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2月半ばくらいから始まります。つーわけで約10ヵ月。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ヵ月というと約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てます。

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。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い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。

そしてぶくぶ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ことそうでもない。アホは戦略も立てずにやろうとして徒勞に終わる。

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ダメなんだよね。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、ちよつと違う…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。まあ〜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 が凄いでしょでシェータとか書きたい
- AI が凄いでしょ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。何故かというとはっと見が派手だと企業の人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いいいなので。シェータ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うておくけど、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』がきましょう。

あと、企業の人目に停めてもらう回数が多ければ多い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てい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。

まあ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ー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ツ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 を軽くやっとうかと思っています。本当に軽くな。

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(設定とかデバッグ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(シェータとか使わない)
5. DxLib で簡単なポストエフェクト(シェータ使ってみる)
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っています。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えよう。

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ン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。

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)。

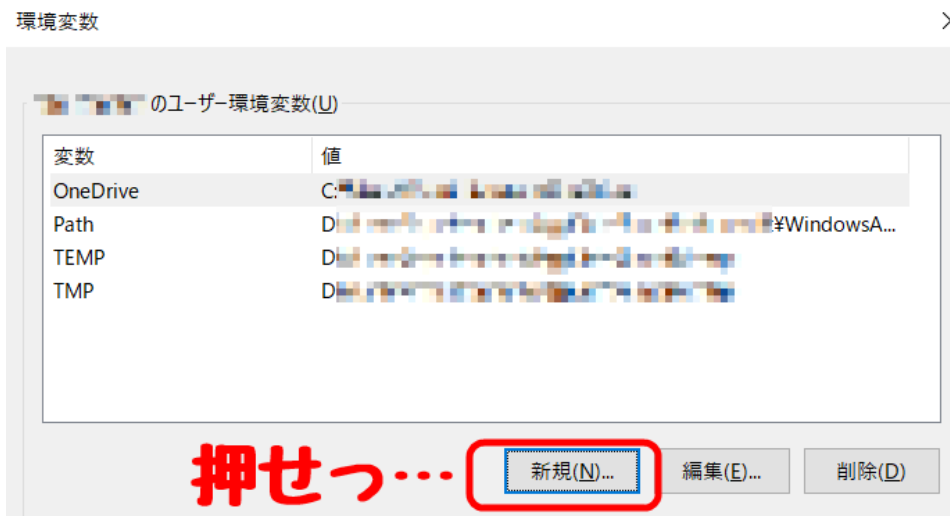
あと、新2年生にとってはライブラリのリンク経験は DxLib くらいしかないのかもしれませんが、今後のプログラミングをやって行くと分かると思いますが他にもいくらでもあります。

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。

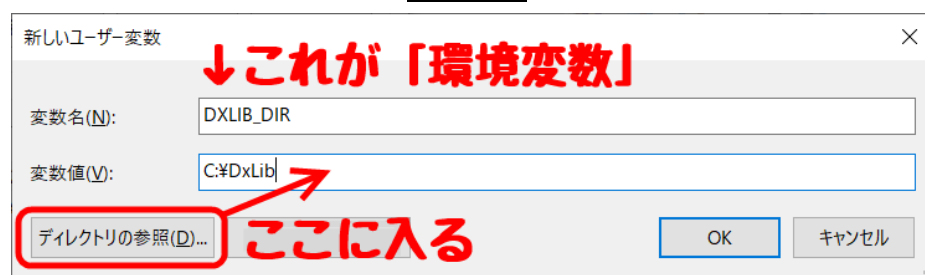
- ① まず自分の PC の DxLib のフォルダの場所を確認します。
- ② 次に Windows 左下の検索バーに『環境変数』と日本語で書き込みます
- ③ そうすると2つくらい候補が出てきますが『システムではないほう』を選択します。



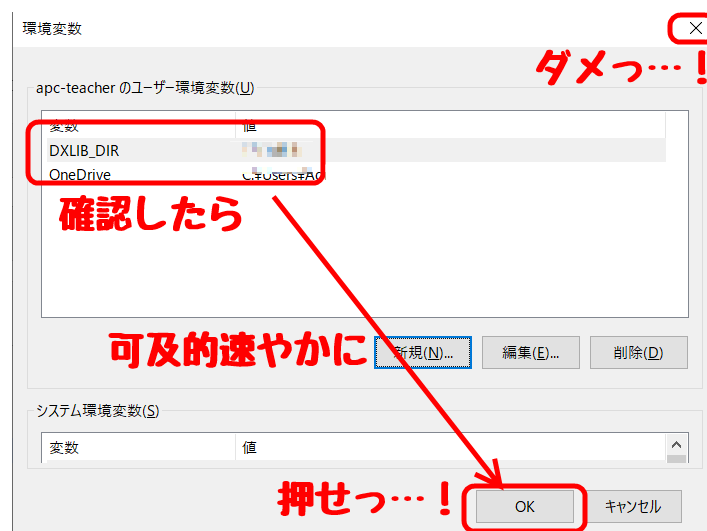
- ④ そうすると『環境変数』のウィンドウが立ち上がりますので、上下に分かれている画面の上側で『新規』というボタンを押します。押せよ!絶対押せよ!



- 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...!!

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…!!!巧妙に仕掛けられた悪魔的…罠!!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あ!? 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ア?

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、でーいー、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。

```
D:¥Users¥[redacted]>echo %DXLIB_DIR%  
%DXLIB_DIR%
```

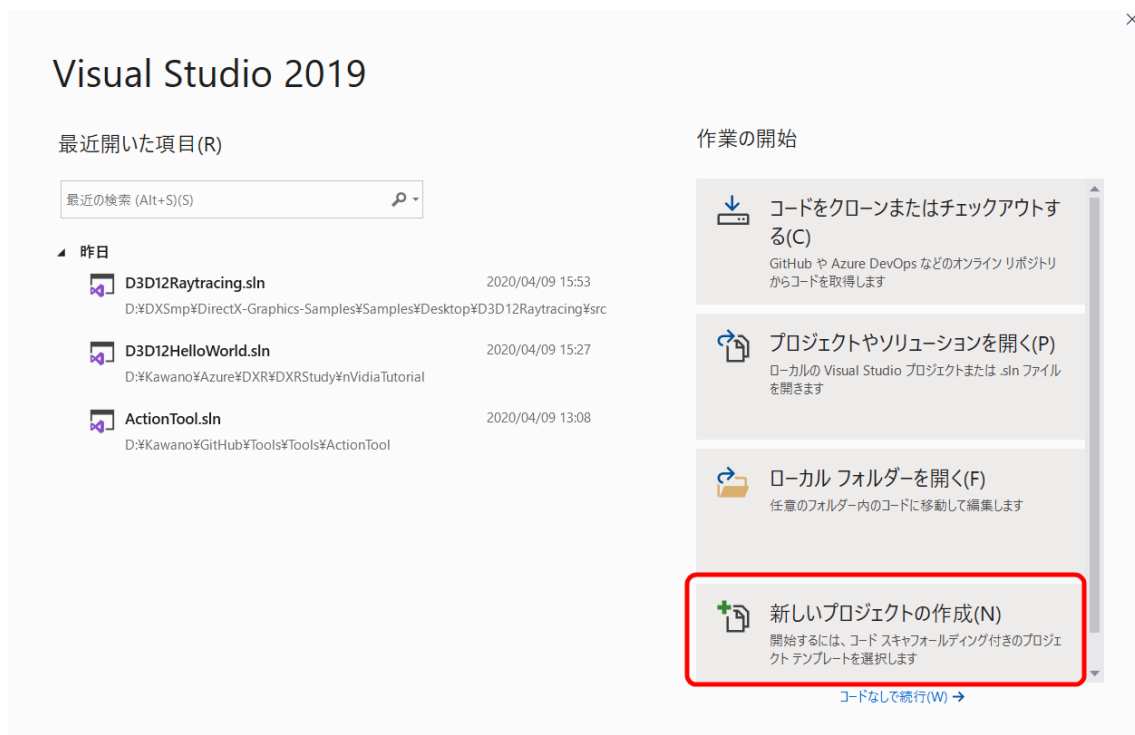
ダメなパターン

うまくいって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

```
D:¥Users¥[redacted]>echo %DXLIB_DIR%  
C:¥DxLib ← OK牧場
```

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。

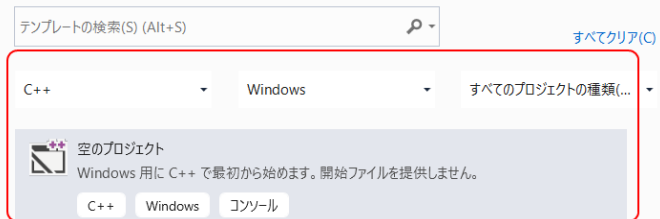


ここで作るのは『空のプロジェクト』(C++ Windows Console)です。

新しいプロジェクトの作成

最近使用したプロジェクト テンプレート(R)

最近アクセスしたテンプレートの一覧は、ここに表示されます。



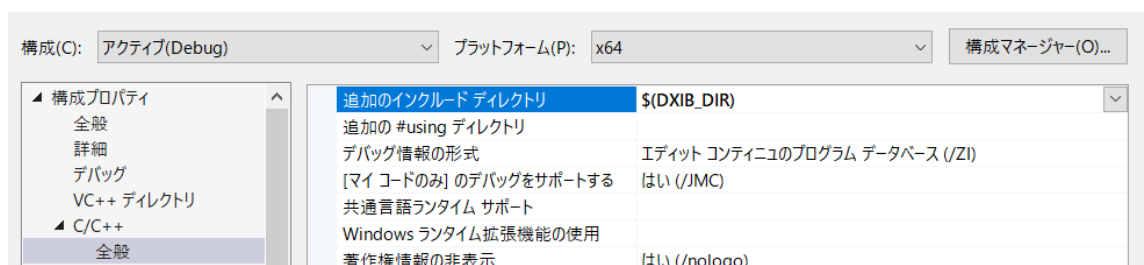
- ⑪ ソースコードとして main.cpp を追加してください。で、以下のコードを追加します。

```
#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;
}
```

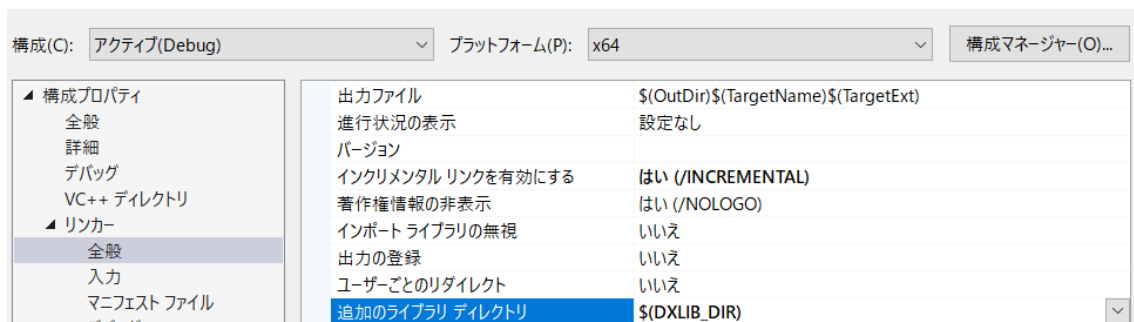
まあ、言うてもこんな本質にはかわりないので、写経しようがなんしようが好きにすればいいです。

- 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。

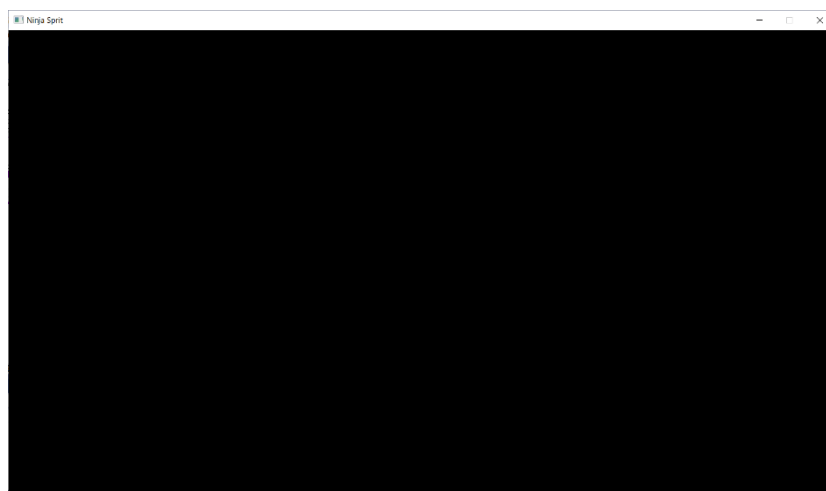
はい、一応これでスケルトンプロジェクトが立ち上がるはずです。簡単なのでさっさとやりましょう。全然プログラミングと関係ない部分です。

なお、インクルードとリンク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きリンク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てもかまいませんが、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。よろしくお願ひいたします。

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。

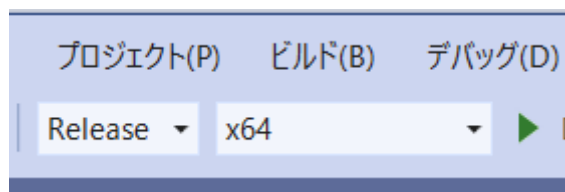
ちなみにコマンドラインのコマンドの echo は環境変数の真の姿を表示するものです。これは後述します。

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に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

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ブギャー!』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 にならんじやろが!!!と思われるかもしれません。そりゃごもつともです。

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、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、その名前が Intel8086 だったんですよ。そもそも、こ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!!!いゝゝ加減にしろ!!!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 ってすればいゝものを、通りがゝいゝからとそのまま 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よ。

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。

まあ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、昔のゲームのコンスト値でよく使われてた数値ですね、65535 ってのは。

まあ、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、あくまでも単純な話ですが…。

まあ、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。まあ、ややこしい理由があるのです。

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。

まあ、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ったわけです。

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、64bit CPU 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

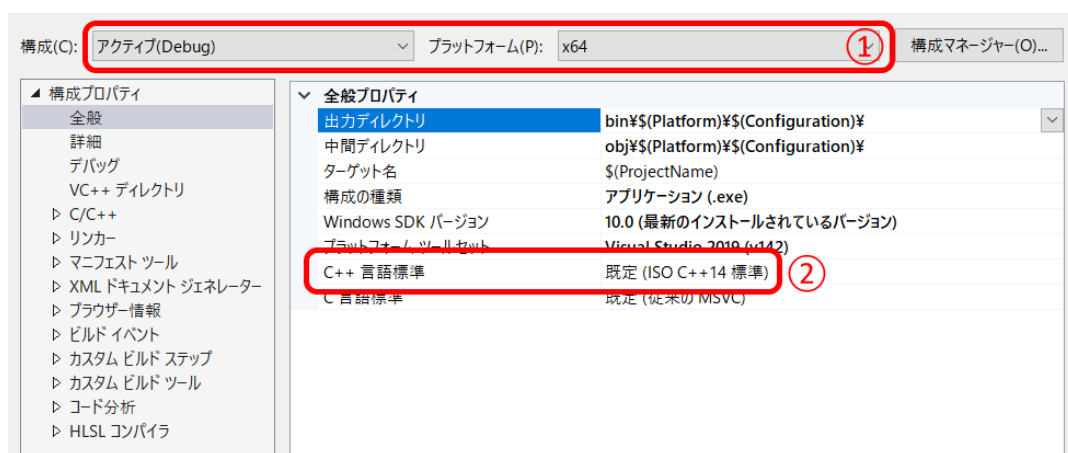
- Program Files
- Program Files (x86)
- Program Files (x64)

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

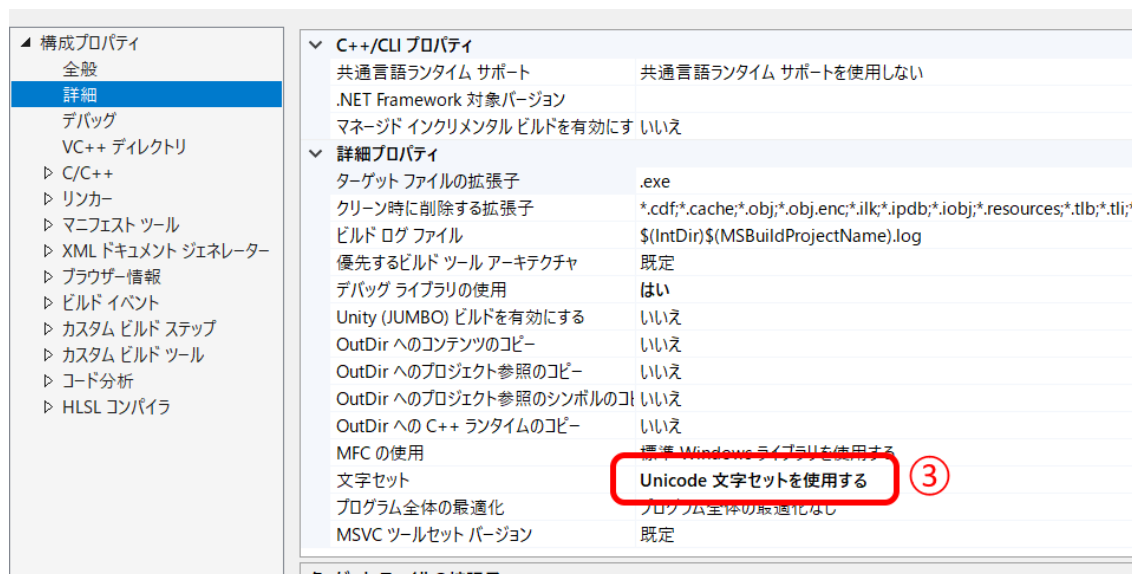
さてさて、プロジェクト設定の中身ですが開くとこんな感じになるかと思えます。



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。

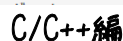
次に②ですが、これは C++ の機能をどれ標準にするのかを設定します。今の Visual Studio 2019 ですと、C++14 が既定になっていますが、現代は C++17 が標準なので、ここは C++17 にしておくことをお勧めします。

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。

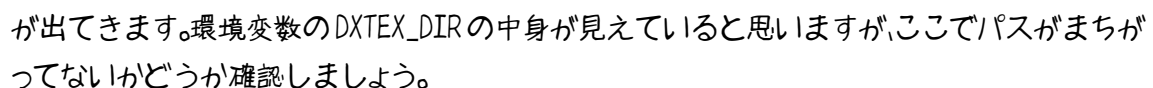


Unicode 文字セットとマルチバイト文字セットのどちらかを選択できます。

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。



追加のインクルード ディレクトリ



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、割と大事なものはその次の『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。

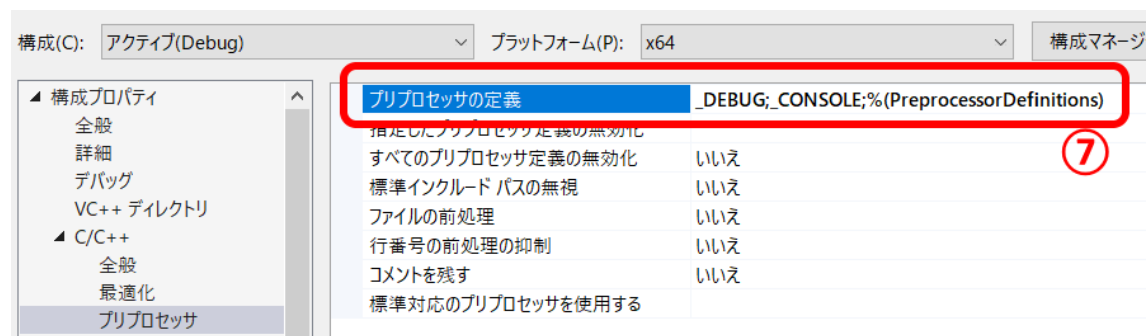
#include

#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てきたかな。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 DxDlib 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「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(ファツキュー!!)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…



LNSEAB
@Inseab

...

windows.hでNOMINMAXは有名だけど、3DCGとかで引っかけたりするのはnearとfarでNOMINMAXみたいなまとめで無効にできるマクロないのでundefするしかない

午前7:44 · 2021年4月15日 · Twitter Web App

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、define マクロとは define の機能を開散的な感じで使ってしまうというわけだ。

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。

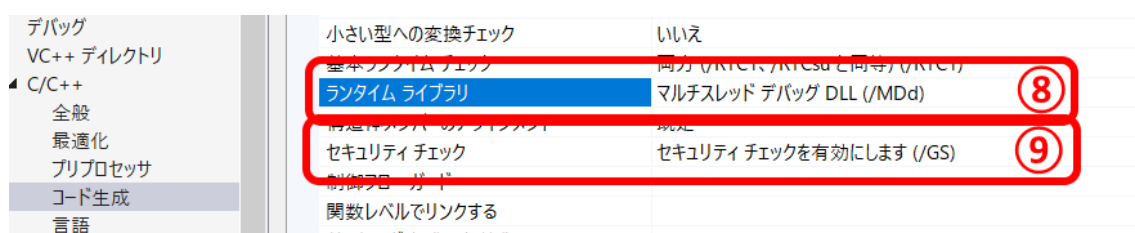
まあ、それはともかくこいつは副作用がヤバくてプログラム文中に min(0,0)という文字列を

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)を生成するのかが聞いてきます。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、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。何でこういうややこしい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こと、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、リンク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

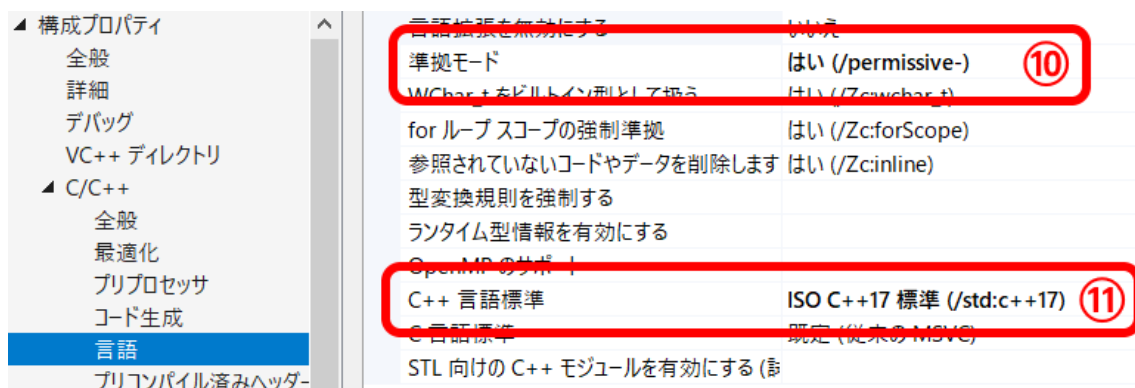
かだけは確認しておきましょう(最近は大抵のフリーの便利ライブラリの場合 GitHub でオープンソース化しているため、自前でコンパイルすることになりますが、そのライブラリのコンパイル時に、この DLL 使ってるかどうかの設定は合わせておかなければなりません)

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な妙なバグに遭遇する原因になるかも…な 2 か所



はい、⑩は何かということ 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。

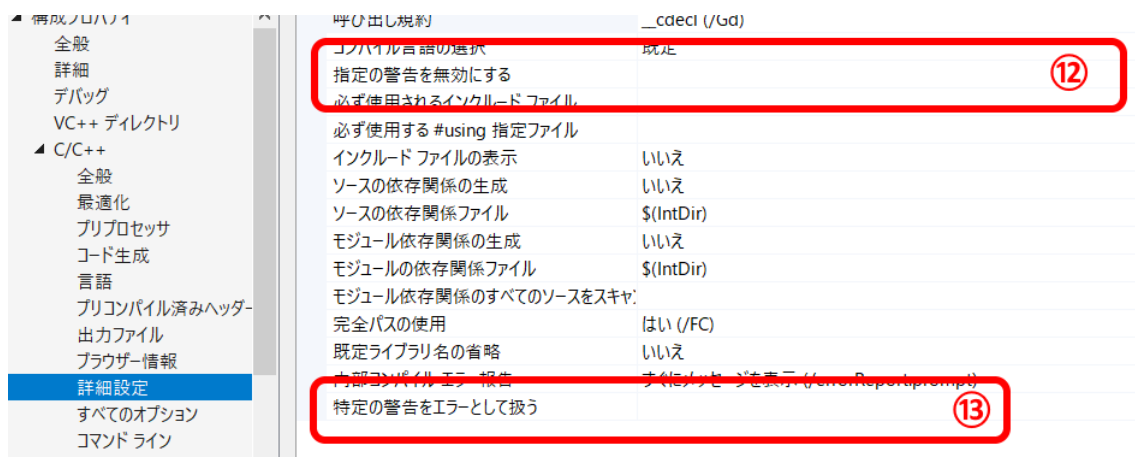
広く使われている言語になった C++において、これでは学習者はたまったものではありません。ということで『C++標準化委員会』というのが発足し、今でも会議が開かれています。MS は自分の所の優位性を保ちたいので、割と最近まで反抗していましたが、流石にそうも言ってもらえず、とはいえ MS 製品使用者とそれまでの互換性も必要だという事で、こういう部分で選択できるようになっています。

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『動きやいれんだよう!!』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、『これを対処してい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』といった場合は、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⑩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。

```
warning C4244: '初期化中': 'float' から 'int' への変換です。データが失われる可能性があります。  
が終了しました。
```

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⑩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③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、将来的に良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』って時にこの③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⑩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。次にリンクです。こちらはそれほど多くありません。

リンク編

ブラウザ情報	出力の登録	いいえ
詳細設定	ユーザーごとのサマリー	いいえ
すべてのオプション	追加のライブラリ ディレクトリ	\$(DXLIB_DIR) ①
コマンドライン	ライブラリ依存関係のリンク	はい
▲ リンカー	ライブラリ依存関係の入力の使用	いいえ
全般	リンク ステータス	
入力		

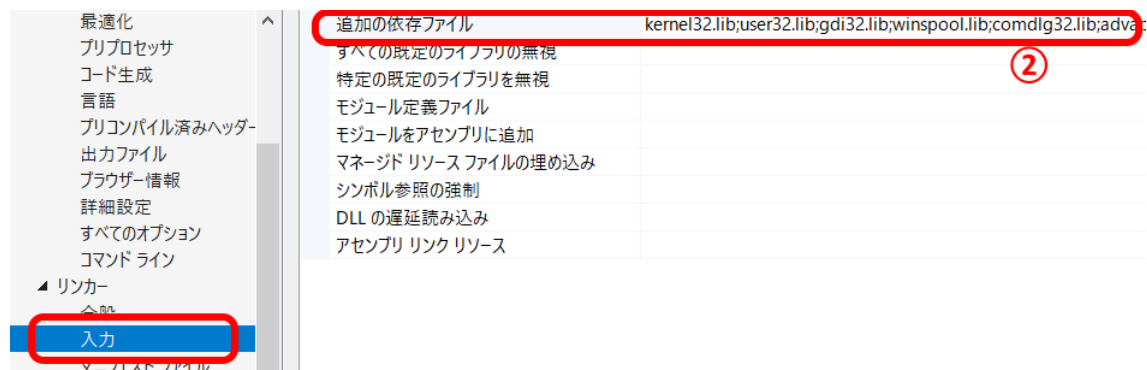
はい、まず何よりもここですね。DxLib など、外部のライブラリを使用する場合は、ここを必ず設定する必要があります。これ DXLIB しか書いたことないから、複数書くやり方が分からない人もいるかと思いますが、それはセミコロンで区切れます。

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

まあ、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、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 DxDiag 以外に Effekseer などを使用する場合はここに Effekseer.lib などを書くこととなります。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。

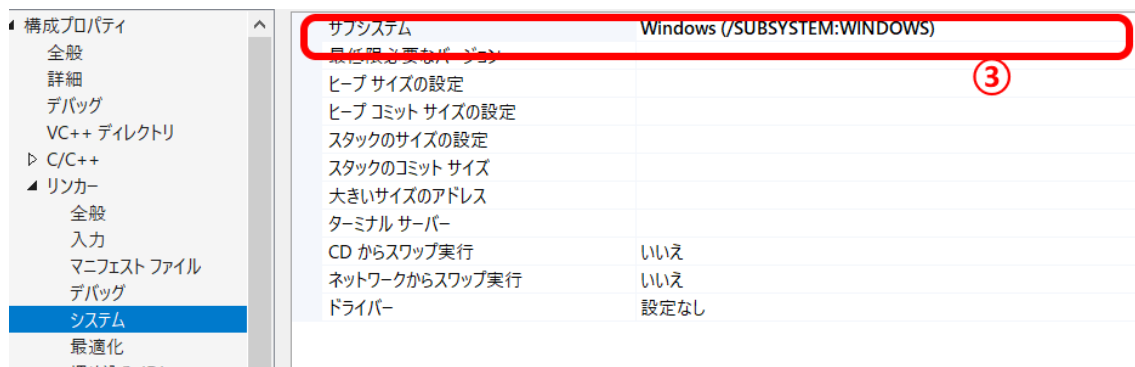
『あれれ〜？DxDiag ってライブラリじゃないの？なんでここに名前が無いの？』

と思うかもしれませんが(うぜえ)、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。まあそれはまたあとでお話します。DxDiag 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、ウィンドウス(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。

ひとまず把握したい方がいれば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、これ知ってるって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、もしかしたらちよ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あす！

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());
```

こんな関数を書いてある行で 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、この関数を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、次は半数くらいの方がびっくり!! するだろう

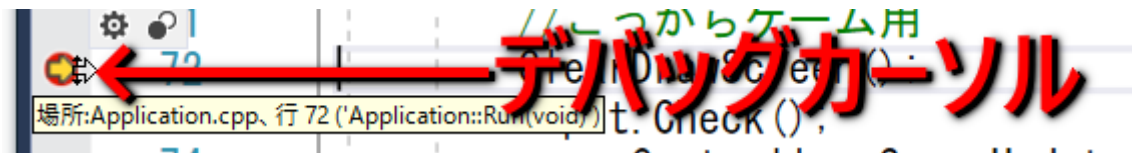
デバッグカーソル(?)をドラッグできちゃう

知ってた?

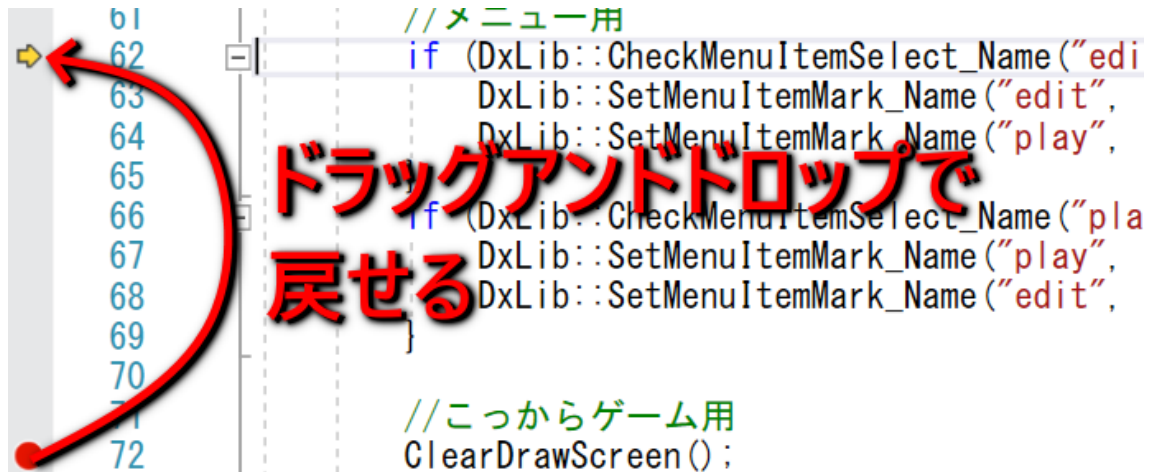
あっ、そう!!! まあ、それはそれとして解説しよう。

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。嘘だと思ふんならドラッ



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。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。

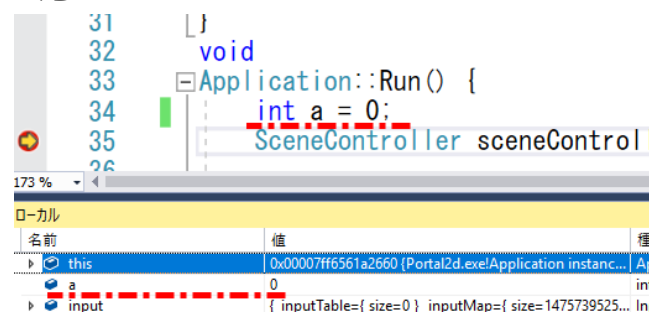
この機能を用いると、デバッグ情報の信頼性が著しく低下するので、使うのはなるべくやめておいた方がいいとは思いますが、でも便利なので、デバッグの信頼性が下がっても大して問題ない時は利用してもいいと思います。

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。

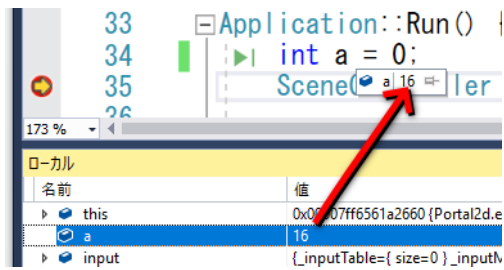
それではそういうデバッグの信頼性を下げるけど、便利な機能をもう一つ…

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。

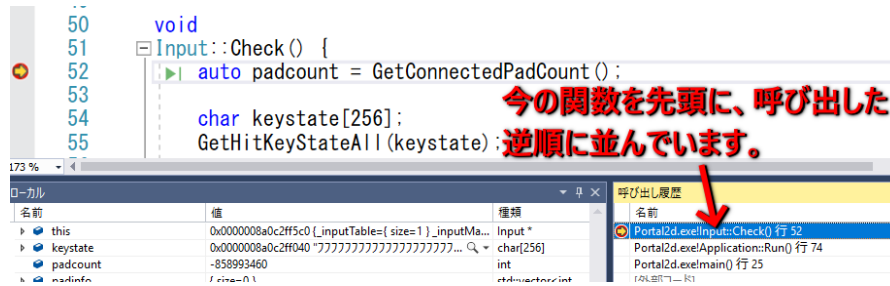


こんなことができます

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？このコードに対して意味なんてないよ。

```
#include<iostream>
```

```
using namespace std;
```

```
int main() {
```

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

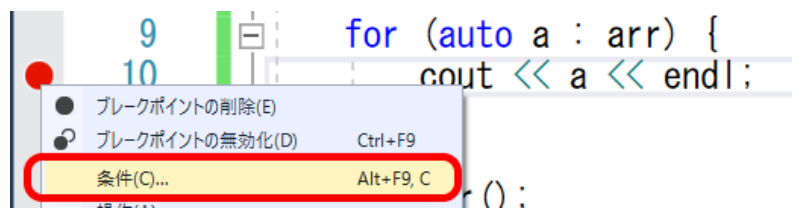
```
    for (auto a : arr) {  
        cout << a << endl;  
    }
```

```
    getchar();
```

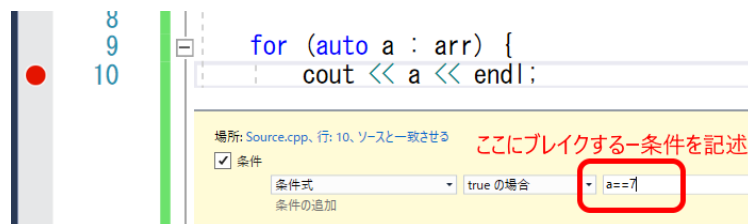
```
return 0;  
}
```

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…。

まあ、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イクするため、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、条件ブレイクを置いている箇所は若干処理スピードが落ちます。まあ、バグった時にしか使わないから問題ないと思います。

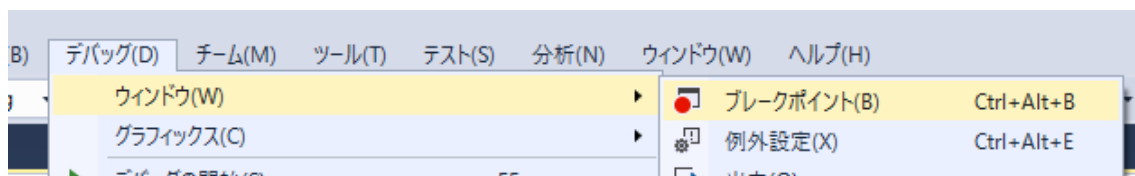
データブレークポイント

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。

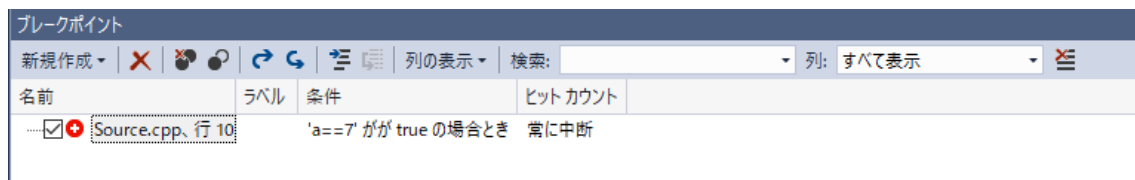
例えば、なんかの値が変化した瞬間を捕まえたいときってありますよね？そういう時に役に立つ機能です。

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、誰が犯人が分からない…と言うのがあったと思います。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。

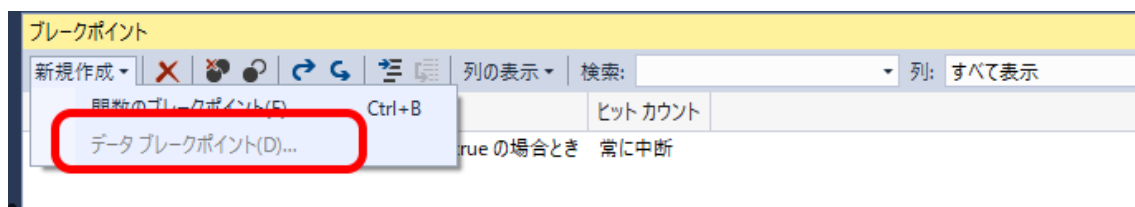
まず、デバッグ→ウィンドウ→ブレークポイントをクリック



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。为什么呢？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。

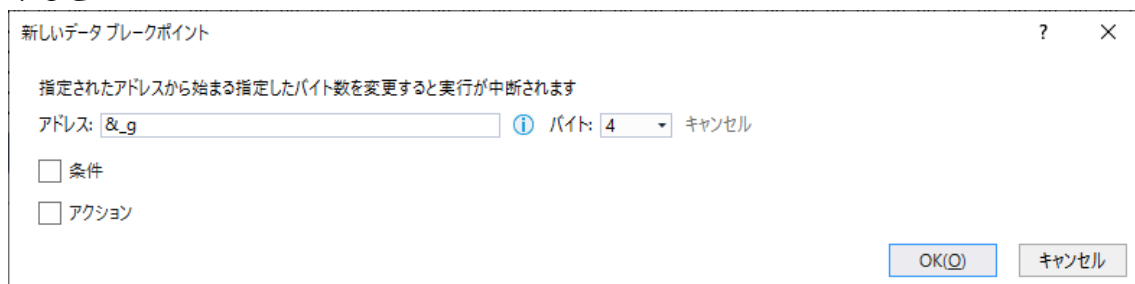
で、どうするのかというと、たとえばグローバルに

```
int _g=10;
```

なんてのがあったとします。

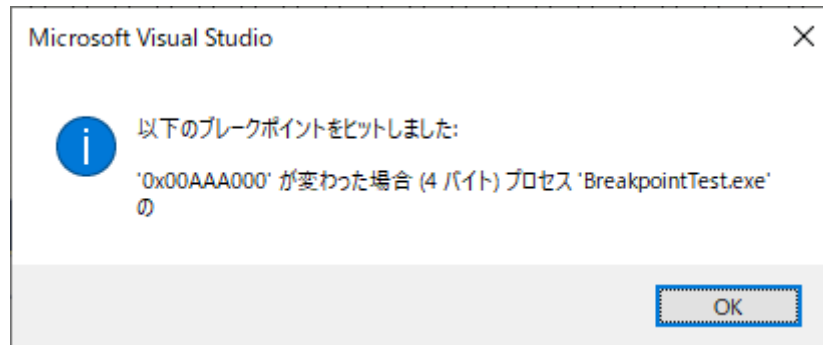
で、誰かがどこかでこれを変更しているのだが、誰だかどこだかわからない。それを知りたいとき…まあ、実行時にしか分からないので、main 関数の最初にでも普通のブレークポイントを置いて止めます。

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

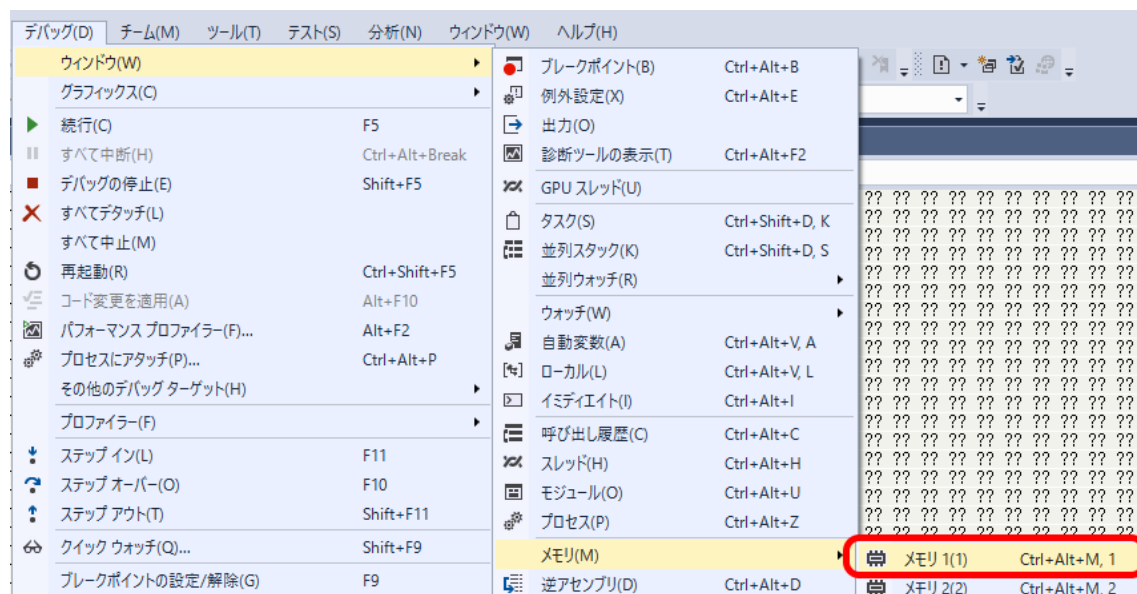


みたいなメッセージボックスとともに処理が中断されます。

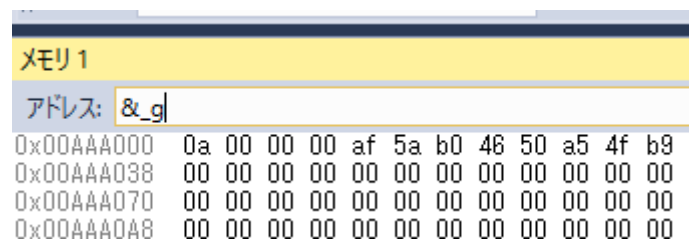
まあ、この便利さは、C++でクソみたいなバグに悩まされない、このありがたみは分からないともいいますが、このクラスで開発するなら、まあそのうちお目にかかれます。

メモリの中身を見る

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こと、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

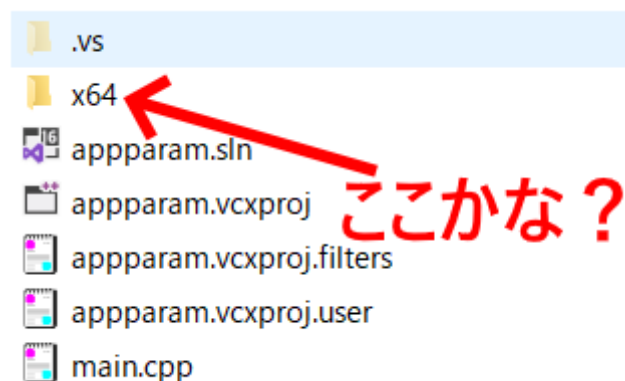
多分 2 年生になったばかりの皆さんはプログラム書いたら→デバッグ実行～ってな感じでや
ってると思います。それはそれでいいです。全然問題ないしプロがやってる事もあまり変わり
ありません。

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。

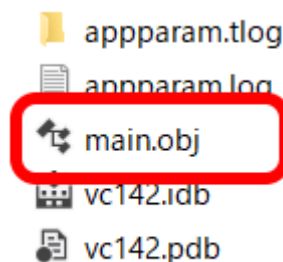
まずそもそもさあ…exe の場所、あんだけど、見ていかない？

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。

まあ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たりしてたというのだから、まあ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。

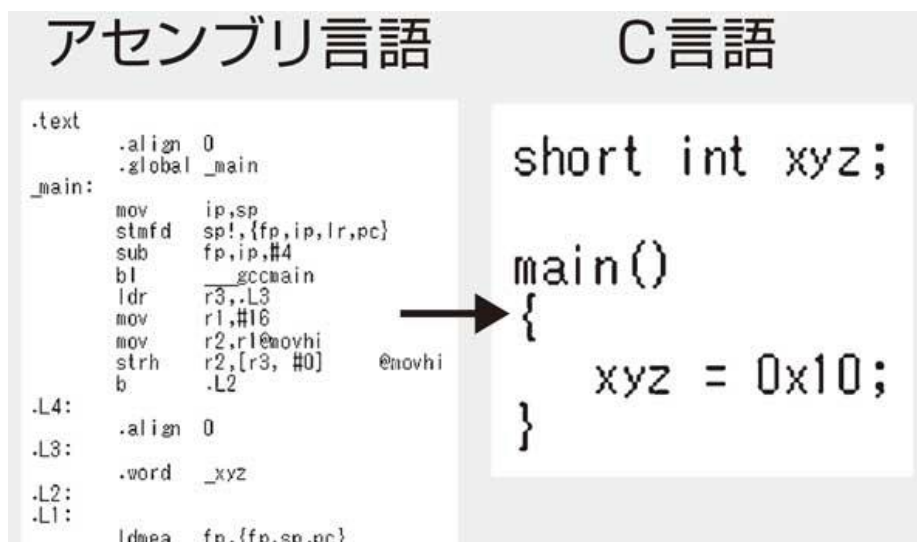
アセンブリ

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

ファツ!?ぶざけるな!!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 はスタックに乗つける、mov は代入、sub は引き算ですね〜

いやいや、ちょっとまで、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言語でもまあ〜〜〜だマシに思えてきませんかねえ〜

まあ、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のかなと思います。

じゃあ、コンパイラだけでOKか?というところ、そうはいかないんですよ。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。

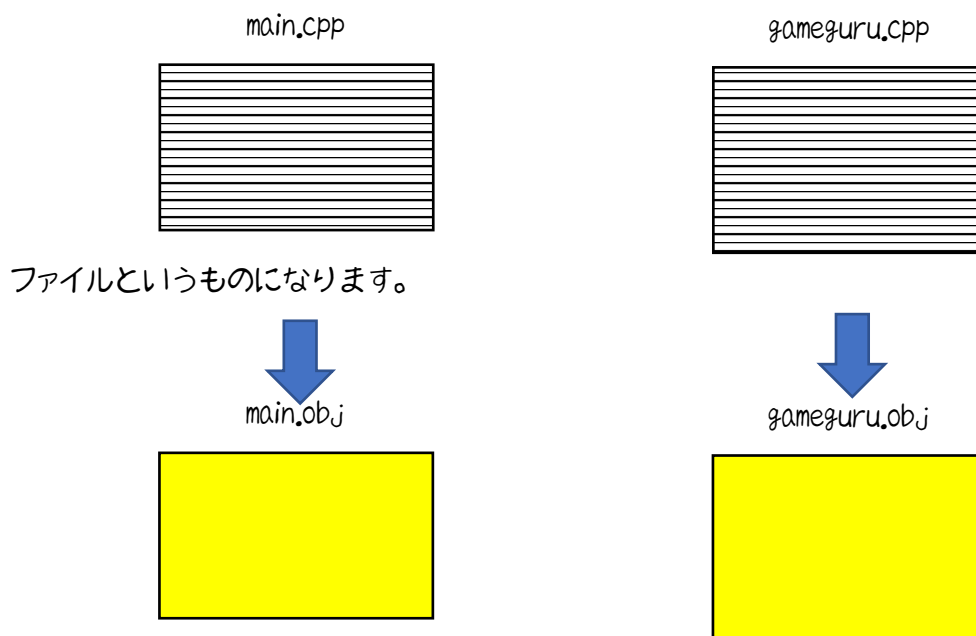
リンクとは…

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ョツと固めて、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。

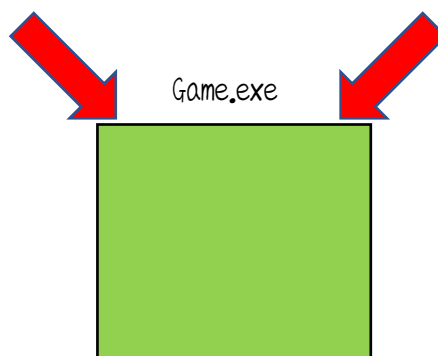
複数あります。

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

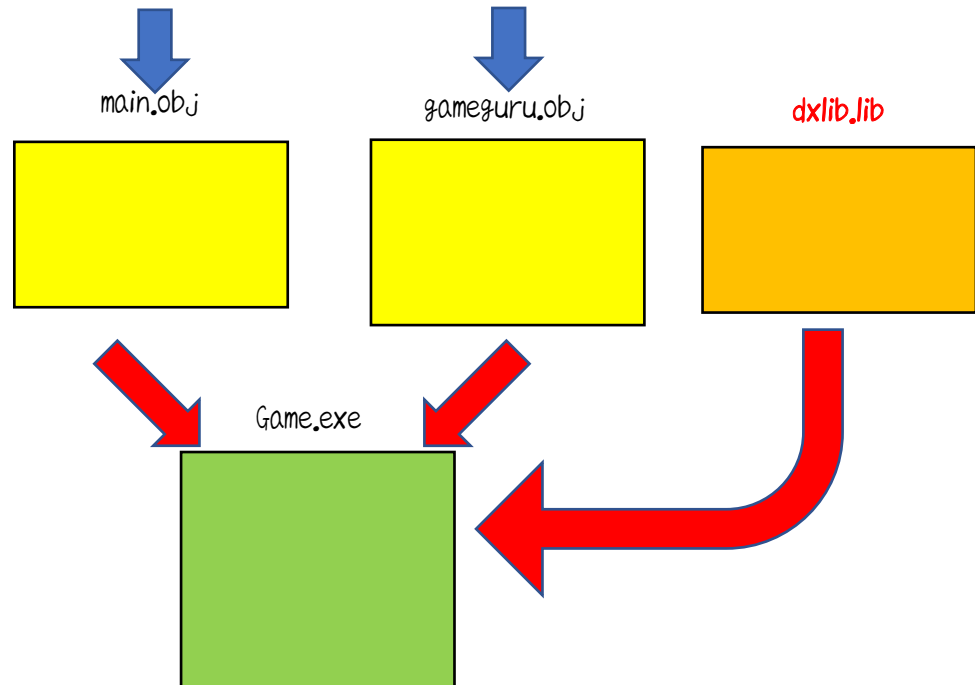


これらはバイナリです。なお obj フォルダに入っているはずなので、よかったら見てみて下さい。でもこれではバラバラの obj ファイルなので『リンク』という処理が必要になります。簡単に言うと、2つのバイナリを合成する作業です。特に何も考えずにただ合成されます。



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 と一緒にたにされて exe ができあがっています。

ここまではわかりますね？ちよつ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エ…馬鹿にしすぎだズエ…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。

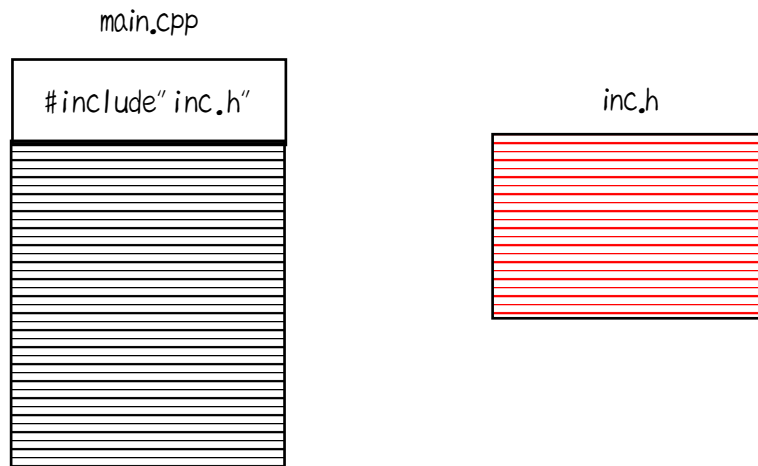
#include のしくみ

#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…アプリコンパイル後は

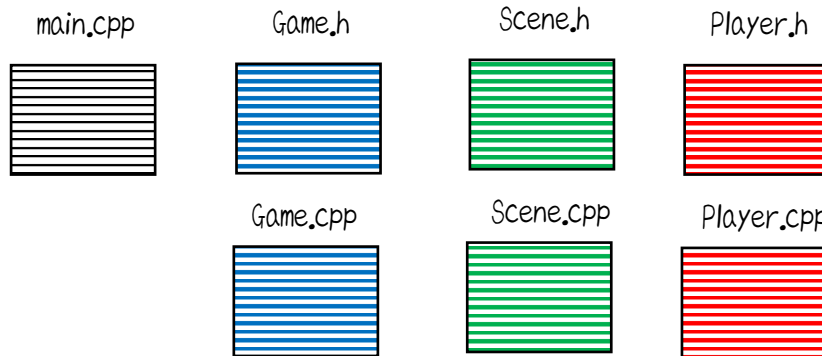


こうなります。なのでヘッダファイルに色々置きすぎるといろんなところでこのコピーが生成されプログラムサイズがでかくなります。さらに言うと、この後にお話しする翻訳単位(要はコンパイル対象)の関係上、ヘッダ側に実体が混ざっている場合に面倒なことになります(おなじみリンクエラー)。

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 があつたとします。

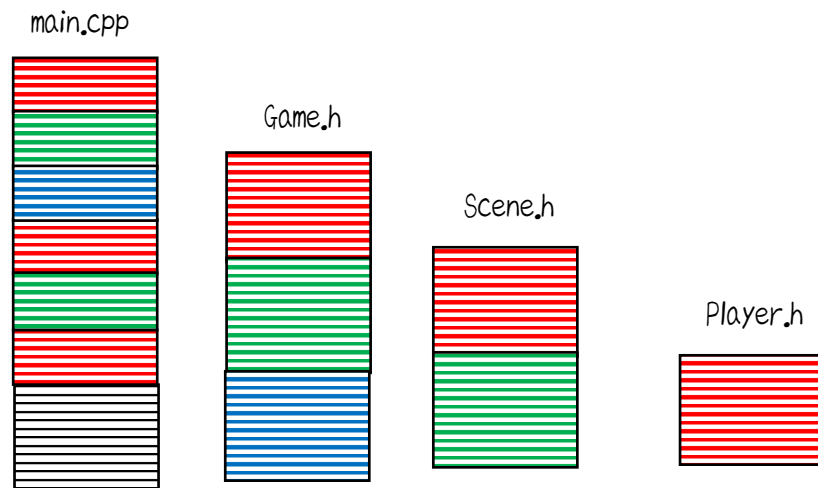


あ, cpp 書いてますが、今回の説明はどちらかというとヘッダ寄りなので, cpp との関連におけるややこしさは次の『リンカ』の話に回します。ヘッダに注意を向けてください。

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。そうするとプリプロセス後にどうなるかということ



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` されていないので `#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 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した。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ことがあります。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
};
```

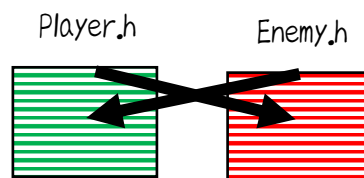
```
//Enemy.h
struct Enemy{
    Player* player_;//プレイヤーの情報
};
```

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
};
```

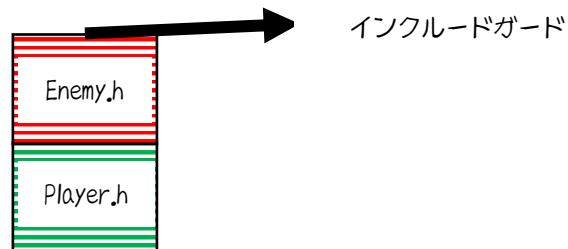
なんてやると



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、それはともかく次の話です。

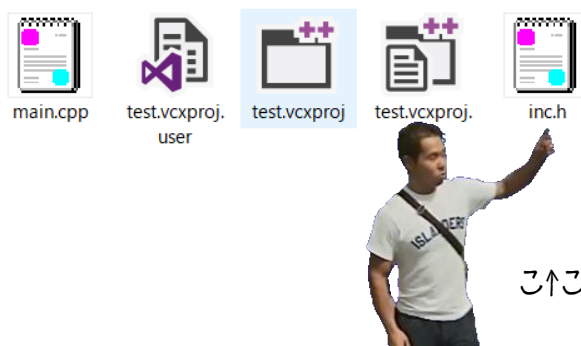
#include<>と#include""の違い

#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、違いはわかりますか？
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。

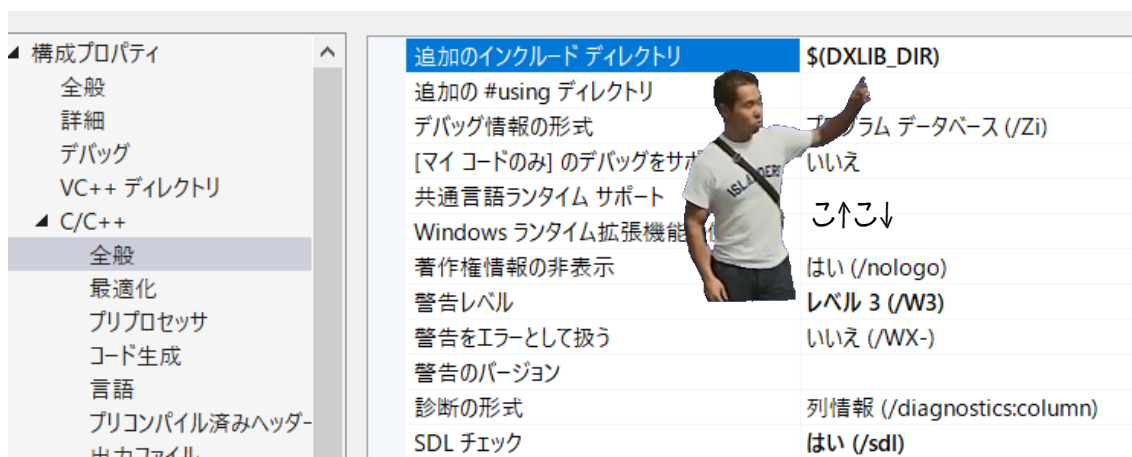
#include"〇〇"...その場を調べる

#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 があるフォルダと同じフォルダから検索するんですね。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りますが、



ここになります。で、ここだけで終わるかということ、そうではなくて



ここも含まれます(ここは studio 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てちゃダメな理由

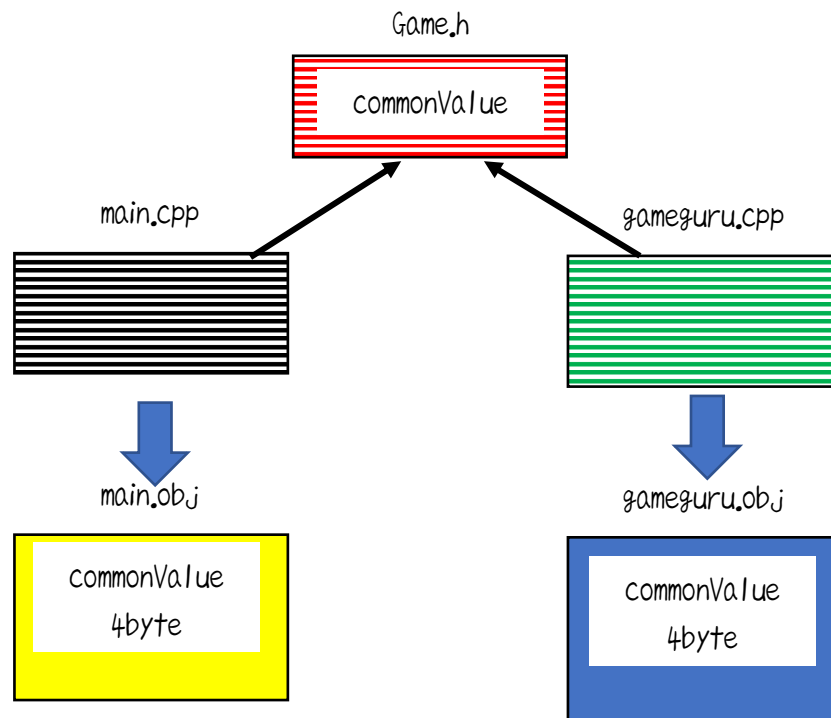
もし、2つの cpp が同じヘッダファイルをインクルードしていたとします。当然それはあり得ますよね？

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。

「その何がアカンねん!!」あかんのですよ。どういうことが説明します。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、お前この名前がぶつとるがな。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ことご存じ extern 修飾子をつけます。この意味は『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(gameguru.obj)も使えるようになるというわけです。

ここで覚えたいほしいのは 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ことです。

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こと static 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。別に支障はないの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 を使うとまあ、確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？

どっかで言ったかもしれませんが、PreProcessor(前処理)の事なので、役目的には『本来のコンパイル』の前に行われる処理の事です。

なにそれ？と思われるかもしれませんが、しれっと使っているものです。よく使うものから紹介していきます。

```
#include
#define
#if~#else~#elif~#end
#ifdef
#ifndef
```

こいつらは、というかプリプロセッサはすべて『コンパイル前に処理される』のがポイントです。処理とは大半の場合『展開』となります。

#include

#include はご存知の通り、#include の後に書かれるファイルをまんまその場所にコピーしてきます。このため、前述しておりますが、ヘッダに#include を書くのは必要最小限にしましょう。

ヘッダ側に#include を書く指針として

- 標準ライブラリは OK(流石にこれはね)
- 使用中の外部ライブラリのヘッダも、必要最小限なら OK
- 自作ライブラリの『共通基本型』は…できるだけプロトタイプ宣言してほしいけど#include 我慢したせいでよけいに煩雑になってしまうのなら、まあ OK
- 必要な自作関数や変数…ダメ。プロトタイプ宣言で回避してください

これに気を付けてれば特にいうことないです。

次は#define です。

#define

#define には 3 種類の用途があります。以下に記します

- ①定数や関数に別名をつける(副作用に注意…定数なら `const` か `constexpr` を使おう)
- ②マクロ関数を定義する(副作用に注意…テンプレート関数を使おう)
- ③後述の `#ifdef` や `#ifndef` と組み合わせて、そもそのコードの挙動を変える
まず①についてですが、

よくある使用例が

```
#define PI 3.14159265358979
```

です。あ、プリプロセッサについては `#include` と同様に、(セミコロン)をつけないようにしてください。必要ありません…ってかエラーになりますので注意。定数として使用する場合はそんなに深刻な副作用を引き起こしませんが、他の定数と名前が被ってしまって別の物に変換されても特にエラーもなんも出ず上書きされます(警告は出ます)

```
#define PI 3.1415925358979
```

```
#define PI 2.7
```

(※怒られません)

なお、C++的には `const` か `constexpr` を使いましょうとのこと。単なる定数なら `constexpr` の方を使いましょう。

```
constexpr float PI = 3.14159265358979f;
```

```
constexpr float PI = 2.7f; //再定義なのでコンパイルエラーが出ます
```

あと、『定数？なにそれ？おいしいの？』って人は 1 年生からやりなおしましょう。マジックナンバー(数値リテラルなどを名前も付けずそのまま使う事)はクソコードへの第一歩です。

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なかった!って人もいるかもしれませんが、実は結構使います。

色々理由はありますが、主な理由は

コマンドラインの方が手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

基本コマンド

よく使う主なコマンド

- echo: 値を表示する
- cd: 現在のフォルダを変更する
- dir: 現在のフォルダ(指定したフォルダ)にあるファイル一覧を表示する
- mkdir: フォルダを作る
- rename(ren): ファイル名を変更する
- set: 変数に(一時的に)値を代入する
- ping: 特定の ip アドレスに通信し通信できてるかどうか確認
- ipconfig: 自分のネットワーク設定を表示(自分以外もできる)

役に立つかもしれないコマンド

- tree: 現在のフォルダ以下のツリー構造を可視化する(文字列として出力)
- cls: コンソールのクリア
- **rm: ファイルを削除**
- **rmdir: フォルダを削除**

(※取扱注意)

てな感じのがあります。これ以外にももちろんありますが、危険だったり、使わなかったりする
ので、だいたいこの辺で…。

あと、サブコマンドとして

more

ってのがあります。これは、表示系のコマンドや、やたらと長々とログやヘルプを吐くコマンドに対して、『一部表示』をしてくれます。

エンターで次の行に移ってくれますね。

さらに

>

という記号を使うと、コマンドラインが吐く文字列をファイルに格納してくれます。基本的にはアペンド(追加)書き込みなので、0から書き込みたい場合は元のファイルを削除してください。

例:

```
dir > directory.txt
```

なんてやるとディレクトリー一覧を directory.txt に格納することができます。

就活の時に意外と役に立つ tree の活用法

就活の際には自分の作品を、どっかファイル共有サービスに置くか、DVD-ROM に焼いて送ると思うのですが、向こうさん(企業の人)も見るのが大変です。

逆に言うとみてほしいものがどこにあるのか、ナビゲーションしてほしいのですが、たいていはなんか ReadMe.txt に『就活作品です』って書いて終わりなのよねえ～。

うん、見る側の人にはたくさん見なきゃいけないし、通常業務もあるしで時間ないんだ。

なので、Readme.txt にはどこに何があるのかも明記してほしい。なので、フォルダ構成とかを書いといてほしいんだ。

そこで役に立つのが tree コマンド

自分の作品集を作った後でトップフォルダに移動し

```
tree > DirectoryTree.txt
```

などと書くと、

フォルダー パスの一覧

ボリューム シリアル番号は 00AF-D147 です

D:.

```
├─Asset
|   └─Adventurer-Hand-Combat
|       └─IndividualSprites
└─Skelton
    ├─Skelton
    |   └─x64
    |       └─Debug
    |           └─Skelton.tlog
    └─x64
        └─Debug
```

などと出力されます。ただし、このようにいらん情報まで出るので、可能な限り不要なフォルダを削除したうえで実行してほしい。無茶苦茶改裝が深くなると使い物にならないからだ。あと最初書いてるボリュームとか D とかも必要ないですね。ということで、不要なフォルダ

を削除してもう一度やり直し、不要な情報を消すと

```
└─Asset
|   └─Adventurer-Hand-Combat
|       └─Individual Sprites
└─Skelton
    └─Skelton
```

となります。あとはここに各自でコメント等で説明を書いていけばいいと思います。

コマンドラインでも変数、関数、ループが使えるぞ？

正確に言うとコマンドラインというよりも『バッチファイル』内ですが、バッチファイル自体がコマンドラインコマンドをまとめたものといえるのでこれでいいでしょう。

ループ

コマンドライン内ループは非常に便利です。数字でループするよりも、ファイルの数だけループするとかいう事が多いでしょうから

```
FOR %%変数 IN (*.txt) DO (
    やりたい処理
)
```

みたいになります。なお、変数名に%%がついてますが、コマンドラインの時に変数を使う時は
%変数

なのですが、なぜかバッチファイルの時は%が増えて%%変数という扱いになります。

なお、このループの意味は、*.txt つまり、拡張子が txt のファイル名でループします。%%変数にはそのファイル名が入ります。

コマンドラインで使える便利系ソフト

Git(GitHub)について

概要

Git について…お話しします。

Git…ってというのはね？バージョン管理システムなんだ。

バージョン管理システムって何？

例えば『ソースコードを変更しました』この時に
変更の履歴を残すシステムである。

誰が変更したか

どこを変更したか

いつ変更したか

Git の特徴は、オリジナル+差分のみを DB に登録していくシステム
これにより DB のサイズを小さく抑えられる。

複数人でプログラムしているとよくあるトラブル

→先祖返り(行った変更が『なかったこと』にされる)

→同時に同じファイルを変更したときにつじつまが合わなくなる

→変更の勇気が出なくなる

→先輩が後輩の首を絞める

個人開発でも…

履歴が残ることで『なんか壊れた』がほぼなくなる。

もし、昨日までは正しく動いてたとすると、今日の変更部分がバグってる
バージョン管理システムを利用すると『差分』が分かるため、その
変更部分をすぐ特定できる→バグの特定につながる。

学生さんにお勧めしたいのは Git+『Git をサポートしているサーバ』を利用する
事で『ソースファイルの紛失』『なんか壊れた』『GW 後にどこまでやってたか忘れた』
などというトラブルを回避できる。

この『Git をサポートしてるサーバ』のデファクトスタンダードが GitHub である。

ちなみにバージョン管理システムには Git 以外にも

•VisualSourceSafe(有料)

•subversion(無料)+trac or redmine

•Alienbrain(高い)

などがあります。

Git 本体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…、急遽 GitHub の話をします。

なんかかという、なんか隔週リモートとかそんな風になったりするっぽいので早めに GitHub 運用を教えといたほうがいいと判断しました。

で、もういきなり使うところから行きますので、

<https://github.com/>

にアクセスしてアカウントを作りましょう。まずはそこからです。

Git 本体について

で、使ったことある人はいいのですが、使ったことがない人には何が何やらだと思います。まず自分の PC に Git がインストールされているかどうかを確認してください。

コマンドプロンプトを立ち上げて Git と入力してみてください。

```
D:\>git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
          [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
          [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
          [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
          <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
  clone Clone a repository into a new directory
  init   Create an empty Git repository or reinitialize an existing one

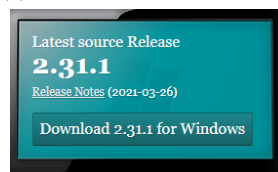
work on the current change (see also: git help everyday)
  add    Add file contents to the index
  mv     Move or rename a file, a directory, or a symlink
  restore Restore working tree files
  rm     Remove files from the working tree and from the index
  sparse-checkout Initialize and modify the sparse-checkout
```

こういうのが出てくれば準備ができてますが、どうでしょうね？

できてない人は

<https://git-scm.com/>

から落としてきておいてください。↓の画像のリンク先から落とせます。



さて、いかがでしょうか？

インストールできましたかね？

インストールできているかどうかはコマンドプロンプトを立ち上げて `git` と打ち込んでもらえれば OK です。

TortoiseGit について

とはいえ、このままでは非常に使いづらい。いや、使いづらいことはないのですが、コマンドをいちいち覚えなさいといけなさい。おそらく 2 年生はコマンドライン自体に不慣れなのではないかなと思います。

というわけで、それを支援するためのツールを紹介します。TortoiseGit です。

<https://tortoisegit.org/download/>

ここからダウンロードしてください。おそらくほぼ全員が 64bit ウィンドウズなので、

for 64-bit Windows

[Download TortoiseGit 2.12.0 - 64-bit \(~19.6 MiB\)](#)

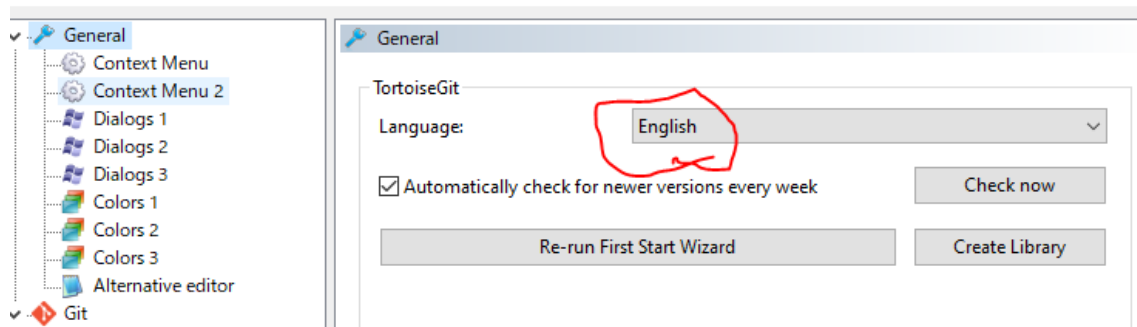
をクリックして落としてそのままインストールしていただければいいと思います。なお、学校から PC の貸し出しを受けてる人は、すでに入っているので、この作業の必要はありません。

次に、言語パックを落とします。そのままですと英語なので、中学英語すら苦手な人にとっては結構なハードルです。それで本筋が阻害されるのもよくないので日本語化します(留学生の方は自国のに合わせてもらって)

 Japanese	ja	100%	 Setup	 Setup
 English	en	85%	 Setup	 Setup

で、これもインストールします。で、これで日本語化してるかということはまだやる必要があります。

エクスプローラ立ち上げて、適当なところで右クリックするとメニューが出てきますが、すでに TortoiseGit の項目があると思います。日本語化されていなければ、この TortoiseGit の先に Setting というのがあると思いますが、それを選択してください。



ここを日本語に設定しなおせば日本語化完了です。終わったら OK を押してください。

GitHub について

さて、今回は GitHub を利用しますので、

<https://github.co.jp/>

にアクセスしてください。アカウントを作っていない人は、アカウントを作ってください。

こんな感じの画面になってるかなと



で、Start a Project でプロジェクトを作っておいてください。まずは名前を決めてください。

あと

GitHub の Create Project で出てくる Project Template についてですが、これは元となる管理方式のことです。

そこで出てくる Kanban について...

所謂トヨタ看板方式ってやつです。ToDo / Doing / Done って感じで3つのレーンを用意する。必要な作業をすべて ToDo に入れておいて、作業中の項目は Doing に移動、終わったものは Done に移動する。

やるべきこと、やりかけのこと、終わったことが混乱しないようにする。

これが海外でも好評だったらしく、カンバン(Kanban)などとアルファベットで表記されます。似たようなのだと Kaizen(カイゼン)ってのもありますね。まあ、PM の授業ではないので、そこは各自調べてもらうとして、この Kanban 方式が手動か自動かってあります。どっちでもいいですが、Auto Kanban にしておきましょう。

はい、プロジェクトができたらいったんトップページに戻って、リポジトリを作ります。

リポジトリ(repository)とは、ソースコードの履歴を記録するデータベースの定義のことです。

こいつに名前を付けることで、どの DB にどのプログラムの履歴が残ってるかを確認することができます。

とにかく今は名前を付けてください。テスト用に TestRepository とでも名前を付けておきましょう。チーム制作時は、チーム名かゲーム名かを入れましょう。

Add .gitignore について

これは『無視するファイル』のテンプレートを指定する。

Git などのバージョン管理システムは、差分をためていく方式なので、

テキストファイルである必要があります。

ですから、管理できないものは外します。その対象の名前や拡張子を .gitignore に登録するわけです。

ひとまずは VisualStudio を選択しておきましょう。

あ、これを教えたのでサーバに余計なものを混ぜたら減点します。

テキストファイルとバイナリファイルについて…

Git とか GitHub と直接関係はありませんが、これを知っておかないと GitHub 使う時にトラブルるんで解説します。

そもそも、テキストファイルって何？

Windows で取り扱うファイルには2種類あります。

テキストファイル / バイナリファイル

テキストファイルの特徴

人間が見て、わかる。←当たり前と思うかもしれませんが、コンピュータはテキストを見て理解しません。コンピュータが理解するのは

0101001010010101010010000010101001010101001010100101010

のみです。こんなもん人間が見てもわかんないし。

アルファベットだの、日本語だの、人間が読んでわかる状態のデータをテキストファイルといいます。

バイナリファイルの特徴

テキストファイルの特徴の反対。一見何を意味してるのか分からない。
プログラムで読み取る。例:画像ファイルや、音声ファイルなど
すべて、数値データとしてみるため、例えば改行が 0A っていう数値として
認識されてしまう。

改行コードについて

※バイナリとして解釈すると『改行コード』はありません。テキストとして解釈するから改行という風に使えるのです。

改行コードを改行とみなすのがテキスト
では『改行コード』ってなんなの？

皆さんご存じ'¥n'が改行コードなんですが…
¥n は改行コードなんだけど、改行を表すものが3種類あります。

- ①¥n: Linux
- ②¥r: Mac
- ③¥r¥n: Windows

SJIS のテキストエディタで最も使用されているのが③の
¥r¥n なんですよ…。

¥r: Carriage Return (CR)

¥n: Line Feed (LF)

¥r¥n: Carriage Return Line Feed (CRLF)

というわけです。もはや何を言ってるのかわからないかもしれませんが、頭の片隅にでも置いとかないとこれでトラブルすることは結構あります。

さて、改行コードを Git の説明のところに書いたのは訳があって、

行番号をもとに差分を作っていく Git としては、改行を改行と認識しない
バイナリファイルを扱うと非常にでかくなる。

GitHub の1チーム当たりの最大容量は 100MB です。お金払うとあげてくれますが、

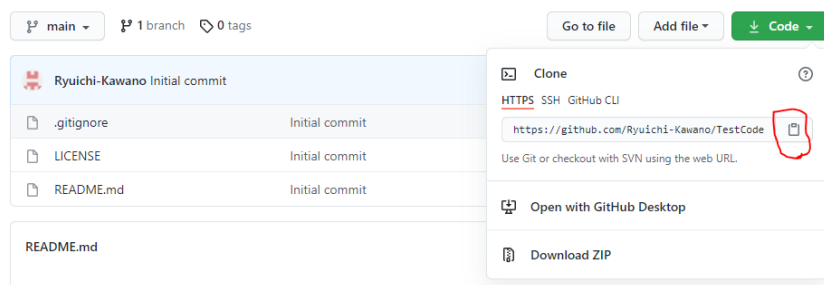
皆さんは無料で使うので、バイナリファイルを上げないようにしましょう。

ここで役に立つのが、`.gitignore`です。`.gitignore`に登録されている
拡張子のものは、git 登録の際に無視します。最初に `exe` とか `png` とか、そういうもの
は外す設定をするのが `.gitignore` です。

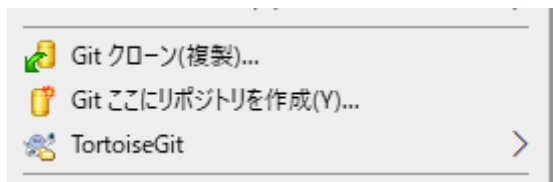
ごたくはいれからとっとと使おうぜ!!

さて、上のようにリポジトリができれば、さっそくクローンしましょう。

リポジトリに Code ってあるんで、そこプルダウンして、なんか変なマークをクリックします。

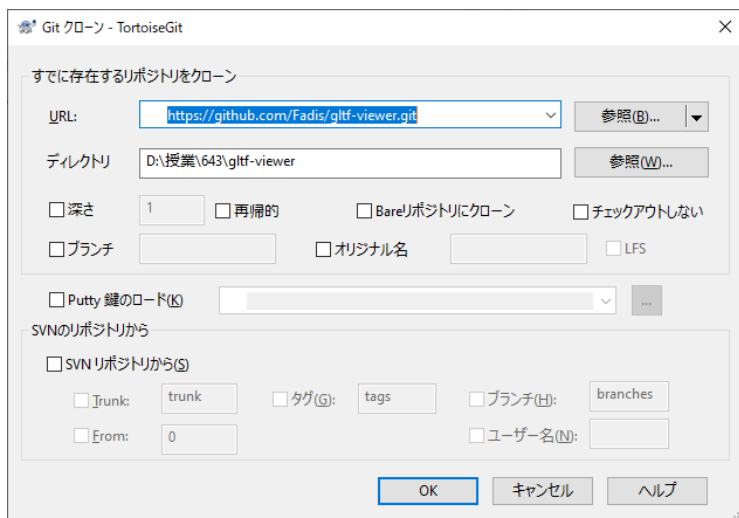


これはリポジトリのURLをコピーしています。



で、クローン

たいていは自動で張り付けられるので、そのまま OK



はい、いいかがでしょうか、ローカルに落ちましたね？

では適当に変更して、コミット&プッシュしてください。
プッシュまでしないとサーバに反映されないのご注意ください。

Git の基本的な利用方法

Git の基本的な利用サイクルは

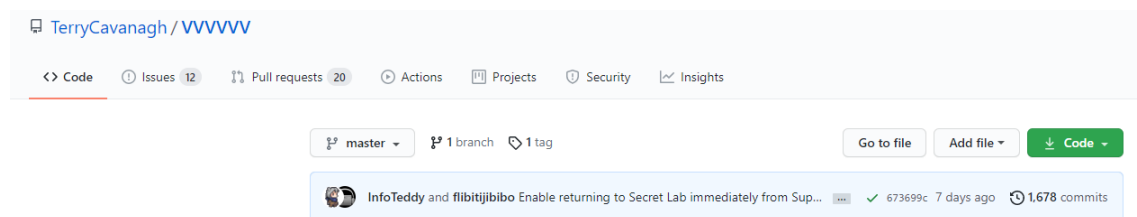
まずクローンをするところから始まりますので、これだけはできるようになっておきましょう。

クローン

自分のプロジェクトをクローンするのはもちろんのこと、ゲームに関する(別にゲームに限らなくてもいいけど)ソースコードも『公開されていれば』落としてみるすることができます。

今朝見た記事だと WWW のソースコードが見れるようになってるらしく

<https://github.com/TerryCavanagh/vvvvvv>



うおおおお!!熱いぜ!!となります。この手の『実際に使われてるソースコード』はプログラマにとって非常に参考になります。このほかにも itch.io 系の公開プロジェクトなんかも見れたりするのでいろいろ参考にするといい。

<https://itch.io/>

あと、itch.io はリソースを集めてくるにも最適なサイトなので、定期的にチェックしておくといい。

<https://itch.io/game-assets>

そのほかにもゲームエンジンのソースコード

<https://github.com/stride3d/stride>

(stride3d...元 xenko)

<https://github.com/godotengine/godot>

Godot

<https://github.com/EpicGames/UnrealEngine>

UE4(Private だけど、Epic アカウント作れば見れるよ)

<https://github.com/cocos2d/cocos2d-x>

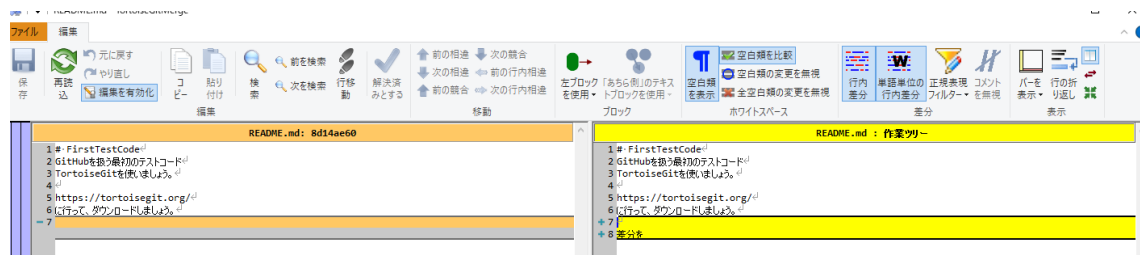
cocos2d-x

などなど、あと、Microsoft とか nVidia とか Google が公式で出してる GitHub とかもめっちゃ役に立つので興味があるプロジェクトをチェックしておきましょう。

クローンが作れたからって、本体の変更ができるかというとそうでもない。Collabollator にしてもらわんといかんので、今は自分のチームだけにしておこう。

ソースコードの変更

次に、落としてきたソースコードを変更して、『差分』をチェックすると、最後のコミット後にどこを変更したかがわかります。



コミット

普通にコミットコマンドを実行すればいいです。これにより変更が確定し、差分がデータベースへ登録されます。

基本的に何か変更して動作確認できた時点でコミットしてほしいですが、あまりやると履歴が煩雑になるので、少なくとも1日の作業終了、もしくは授業終了を目安に『コミット』しましょう。

プッシュ

GitHub などのリポジトリサーバを利用している場合は、コミットするだけではサーバに反映されません。Push コマンドを行うことでサーバに反映され、自分の変更がサーバ DB に保存されます。

チームメンバーを招待

個人として使うなら、ここまでの話で十分でしょう。しかし、したい場合は、複数人で作るようになります。

このため、GitHub でチーム開発する際には誰かのアカウント内にリポジトリを作って、それをチームメンバーで寄ってたかっていじっていく必要があります。

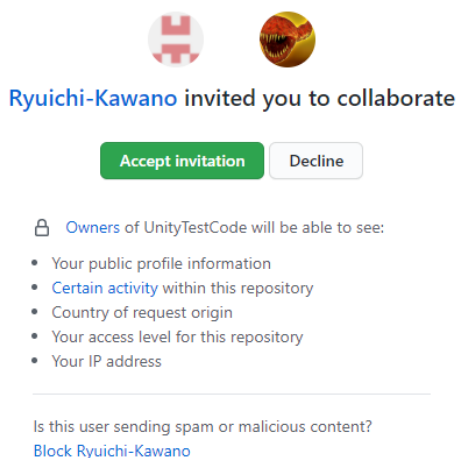
なので、自分以外にもそのリポジトリをいじれるようにする必要があります。
やり方としては、リポジトリの Setting に行って、

Manage Access と言って、Invite a Collaborator で、ほかのメンバーのアカウントを指定して、招待してください。

で、これで終わりじゃなくて、招待された側のメアドに通知が届くので、そのメールから Invite の URL に飛んでください。

@Ryuichi-Kawano has invited you to collaborate on the Ryuichi-Kawano/UnityTestCode repository.
Visit <https://github.com/Ryuichi-Kawano/UnityTestCode/invitations> to accept or decline this invitation.
You can also head over to <https://github.com/Ryuichi-Kawano/UnityTestCode> to check out the repository or visit <https://github.com/Ryuichi-Kawano> to learn a bit more about @Ryuichi-Kawano.
This invitation will expire in 7 days.
Some helpful tips:
- If you get a 404 page, make sure you're signed in as boxerprogrammer.
- Too many emails from @Ryuichi-Kawano? You can block them by visiting https://github.com/settings/blocked_users?block_user=Ryuichi-Kawano or report abuse at <https://github.com/contact/report-abuse?report=Ryuichi-Kawano>

そうすると以下のような画面になりますので、Accept Invite ボタンを押してください。
Decline だと拒否できますが、そんなことやんな(笑)



はい。

GitHub プロジェクトについて

課題①

ここまででいったん課題を出します。

- ① GitHub アカウントを作る事
- ② その中に自分のスケルトンプロジェクトを作る事
- ③ そのリポジトリを僕にチャットで送る事(メールは見ないのでチャットで)

あ、必ずしもスケルトンじゃなくてもいいです。なんかしら技術的なものがあるなら、それを課題として上げてください。

で、お願いします。

期限は来週末(5/28)とします。

僕のアドレスは kawanory@s.aso.juku.ac.jp です。メールでなくてチャットでお願いします。

提出の注意点

- ① 期限厳守(守れない場合は20~30点がつつり減点します。正当な理由なく遅れた場合は0点になることがあります)
- ② スケルトンプロジェクトのタイトルバーの左上に学籍番号、名前が入るプログラムにしてください(不正防止のためです…これができてない人はカンニングとみなし0点です)
- ③ リポジトリに余計なものが入らないようにしてください。obj や exe ファイルが上がっていれば減点します。

プログラミングテクニック

いよいよ本題といったところに入っていきます。ここまでは前座というか準備ですね。