

# DirectX12

## ジョクノプログラミング■



じごくへようこそ

やつふおー!!おひさー!!はじめましての人は初めまして、1年ぶりの人はお久しぶりにござります。DirectX12のお時間がやつてまいりました。

前期でももう死にかけたのに、後期でさらに地獄の責め苦を味わう…そんな目に遭えば



という気持ちになるかもしれない。それは仕方ない。



ぼくは君たちに敬意を表して手は抜かない。全力でお相手いたします。誰かが死にかけていても仕方ない。君たちが隣のお友達をフォローするのは自由だ。だがそれによって遅れても僕はフォローしない。少なくとも授業中はフォローしない。

そんなもので乗り越えられるほど DirectX12 は甘くないのだ。そして俺の信念も固まったのだ。昨年までは「学生の間に DirectX12 なんて教えて大丈夫か?」と自問自答していたし、事実ちょっと適切ではなかったかもしれない。

だが、今まさにグラフィックスプログラミング/パラダイムの時代に突入しつつあります。これはもう「間違ってない!!!!」と言えます。川野先生の暴走はもはや止まらないのです。

# 目次

はじめに .....	4
流れ .....	4
環境設定 .....	10
C++言語のおさらいとか追記とか .....	13
STLについて .....	13
おさらい .....	13
イテレータについて .....	15
リバースイテレータについて .....	15
stringについて .....	16
stringstreamについて .....	17
C++の新しい仕様 .....	18
右辺値参照とムーブセマンティクス .....	19
配列の範囲 .....	24
Emplacement(emplace,emplace_back) .....	24
(おまけ)minmax と iota .....	25
const と constexpr .....	26
まずはプロジェクトを作ろう .....	27
じゃあウィンドウ出すか .....	29
基礎知識説明① .....	33
シェーダ .....	33

# はじめに

最初に言っておかねばならないことがあります。

この授業の主眼は『ゲーム技術の基礎研究』です。残念ながら『ゲーム作り』ではありません。こ↑こ↓注意してください。

えー、じやあ何すんのさ?と思われるかもしれません、先ほども言いましたが基礎研究です。ゲームを作る根本の部分ですね。DXLIB がやってくれていた事(隠ぺいしてくれていた事)が何なのかな…ゲームエンジンがやってくれている事はどういう事なのかな…を知るために今回は DirectX12 を使用して MMD のキャラを動かしてみようと思っています。

半数の学生さんにとっては2度目なので、ああ、またあれか、あれなのがと思っている事でしょう。

とはいっても昨年と同じであれば3年生にこの授業を受けていただけ意味があまりないため、計画としては、去年のやつ+αで『ポストエフェクト』をやろうかと思っております。

やろうと思ってるポストエフェクトは

- 画面にヒビ入れる
- ブルーム
- 被写界深度

です。

まあ、昨年の授業を受けてない人、昨年のテキストを見たこともない人のために流れを言っておくと

## 流れ

1. とにかく DirectX12 ポリゴンを出すまでがんばる(面倒だしシェーダが必要だし即死)
2. ポリゴンに 3D 変換行列をかけて 3D 化する(行列が分かってれば割と大丈夫)
3. テクスチャ貼る(テクスチャは思ったより面倒なんやで?)
4. MMD モデルを読み込んで表示する(まずは頂点情報のみ)
5. 面を貼る(インデックス情報が必要)
6. シェーディングする(数学がクソ出てくる。内積とか内積とか内積とか)
7. 深度バッファを有効にする(めんどう)

8. ボーン情報を読み込む
9. ボーンを回転させてみる
10. ボーンに合わせてスキニング(頂点ウェイトで頂点移動)する
11. WIC ローダを作る
12. DDS ローダを作る
13. ポージングさせる
14. アニメーションさせる(リバースイテレータ登場!!!)
15. ベジエで動かす(ニュートン法、二分法)
16. つぶれ影表示(行列で演して黒く塗るだけ)
17. シャドウマップでセルフシャドウ(シャドウアクネがさ…)
18. 簡易トゥーンレンダリング
19. 輪郭線
20. アンチエイリアシング(輪郭線との相性最悪)
21. IK(いけるかな…)
22. ポストエフェクト(をするために必要な事)
23. 色調整(ポストエフェクト)
24. 画面を割る(法線+ポストエフェクト)
25. ブルーム(ガウス+ポストエフェクト)
26. 被写界深度(深度値が重要ですねえ+ポストエフェクト)
27. インスタンシングで大量表示
28. 法線マップ(接ベクトルと従法線ベクトルが必要なんだよなあ…)
29. ディファードレンダリング
30. TBDR(小林先生のご提供となっております)

まあ、これが全部やれるとは俺も思っていない。時間がまるで足りないのだ。昨年よりかはスピーディにできるだろうけど、みんな死ぬでしょうし(笑)

まあシェーダを恐れずやれるようになっておくと、Unity 使おうが UE4 使おうがちょっとかっこいい!事ができてしまうので、シェーダには慣れておいた方がいいと思うよ。

あと、C++の効果的な使い方(?)についても、しつとやっていくので、頑張ってついていく。

で、ここまで説明に一切『ゲームの作り方』に関するものがない事からも『あつ…(察し)』だと思いますが、今期は『授業外でゲームを作ってください』

# 授業外でゲームを作ってください

大事な事ですね…。しんどい?しんどいよなあ…仕方ない。でも、やれ。



うーん。なんでそんなややこしいことを今やるのがと言うと

<https://www.youtube.com/watch?v=H3M07qR0j28>

のカメラ割れとか



の光が漏れている感じとか



のピントが合っている、外れているの感じとか

ゲームエンジンとかライブラリを使用しているとブラックボックスになってて、中身を理解していないと「アーティスト」や「プランナー」の「ああしたい、こうしたい」に対応できないことが多いんですよ。ちなみにゲームエンジンがキャラクターをアニメーションさせてますけど、あれDirectXが勝手にやってくれるんじゃないんですよ？数学とC++を駆使して実装してるんですよ？

僕らはプロです。プロを目指しています。



もちろんです。プロですから。

と胸を張れるようになるためには魔法使いレベルの事ができなきゃいけません。その辺の高校生ができるレベルができても自慢にならないし、その程度だったらなぜここにきて3~4年もやってんのさ。今すぐ仕事しろよ。

『ゲームエンジンの機能にないから実装できません』ではもうそれプロじゃないと思います。

正直ぼくがそういうやつがプロを名乗っていたら



野郎！ ぶっ殺してやる!!

と思います。

とはいって君たちの殆どに欠けているものがある。それは『知識』だ。CGの理論などは今か

ら1からやつていくのはしんどい…しんどいはずだ。

という事で、知識の正確なところはGoogle大先生に任せるとしたら人間は何をすればよいのだろうか？

そう、用語をある程度知つておかねばなるまい…。何故って？用語を知つていればGoogle先生へのお伺いのやり方がスムーズになります。また、そもそも用語を知らないと特定の技術の存在そのものを知らずに過ごしてしまうという事にもなりかねません。

昨今はゲームエンジンで、レンダリング部分やAIにおけるナビメッシュやビヘイビアツリーガブラックボックス化されて見えなくなっているけど、僕らプログラマはその見えない部分も意識しなければならない。ゲームエンジンの中の人が言ってるんだから間違いない。

<https://entry.cgworld.jp/column/post/201701-gameengine.html>

というわけで、UE4 やるにせよ Unity やるにせよ「プログラマ」を生業とするのならば中身をある程度理解しておく必要があるという事です。

さて、授業の大まかな流れですが、パンナムの研修の流れを参考にしてみましょう。まず、パンナムのはこんな感じでした。

- 1<sup>st</sup> week
  - Day 1: Direct3D12 基礎とポリゴン入門
  - Day 2: テクスチャマッピングとリソースバインディング、3D モデル
  - Day 3: 3D モデルのシェーディング、ライティング基礎
  - Day 4: レンダーターゲット、コマンドリスト
  - Day 5, 6: Shadow map
- 2<sup>nd</sup> week
  - Day 7: PBR と発展的なシェーディングテクニック
  - Day 8: Compute Shader
  - Day 9, 10: Deferred Rendering

パンナムの連中にできるんなら、俺たちにもできるはずだよなア？

…嘘です。

というか、大事な大事なアタックチャーンスではなく、大事な大事なスキニングとトゥー

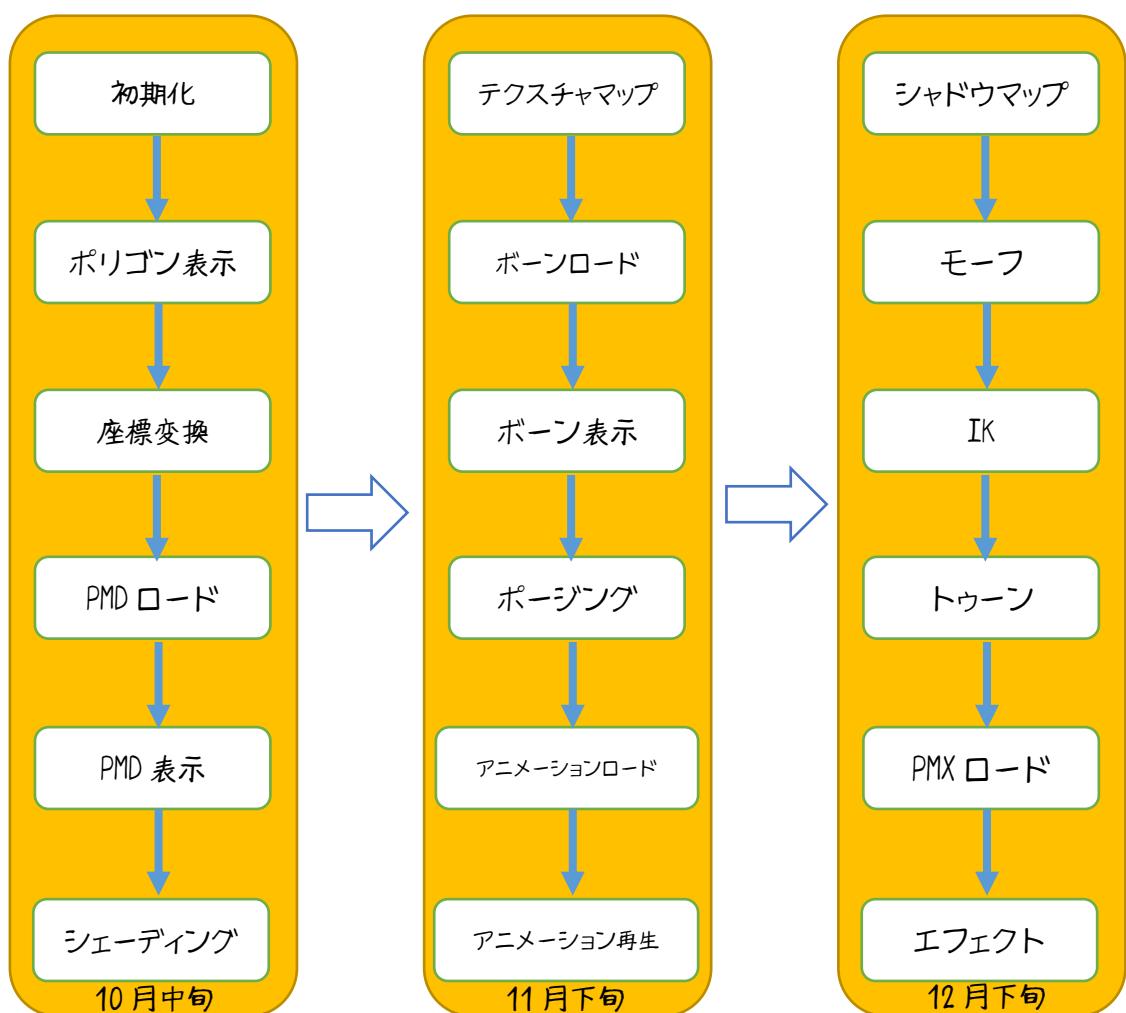
ンとポストエフェクトがないではありませんか!!!

という事で、ComputeShaderとかPBRを後回しにして、その代わりにスキニングとトゥーンを入れたいと思います。

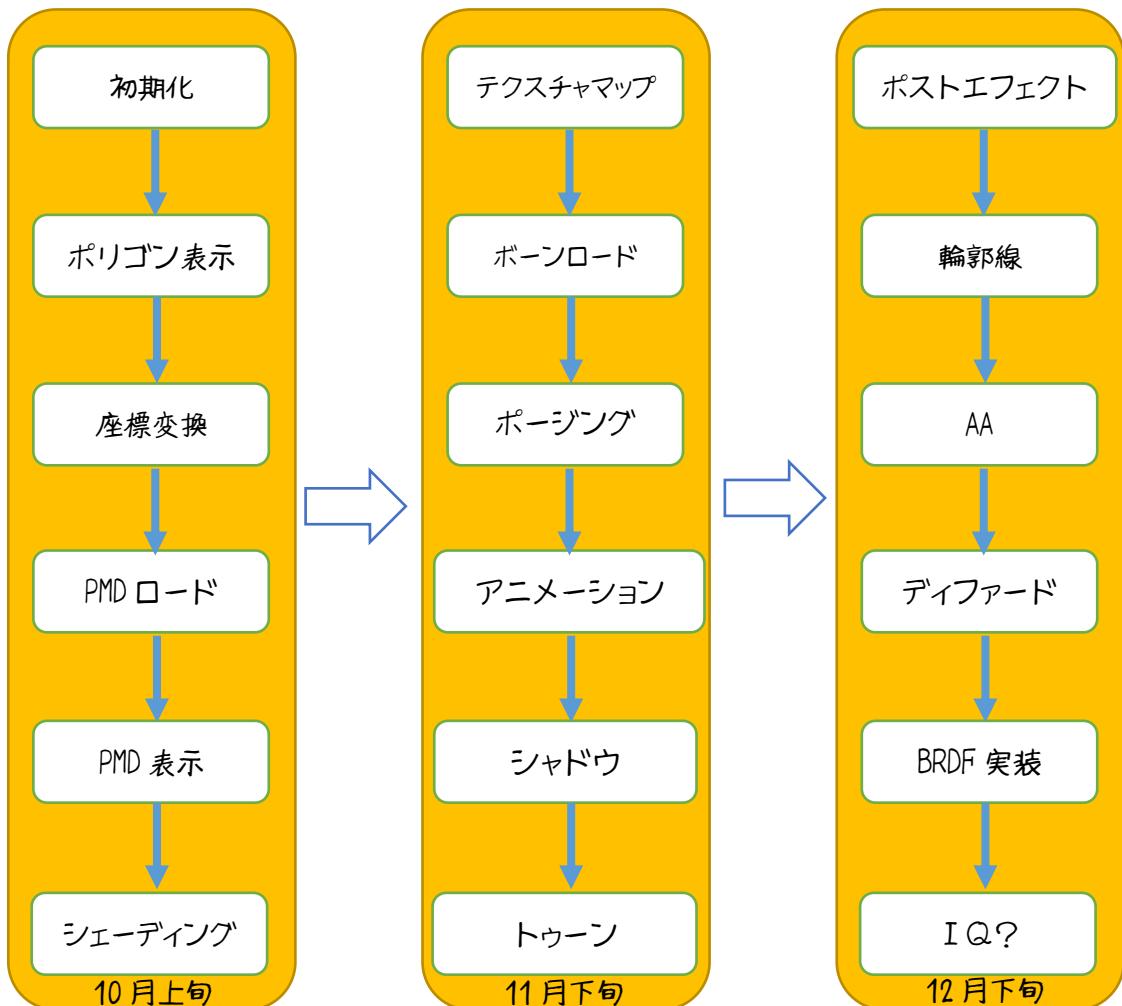
昨年まではやってなかったのですが、DX11からの定番としてディファードレンダリングというのがあるので、今年はそれを入れていきたいと思います。さすがに欲張りセットかな?

あと、昨年に失敗したこととして、僕も余裕がなかったからなんだけど、設計的な事後回しにしてたから、かなりクソコードになってたので、多少の設計はやりながら進めていこうと思います。

昨年まではこう…



今年は



こんな感じで行こうかなと思います。十分に時間ができたら、前期にやった IQ を DX12 で作ってみるというのもいいと思います。

辛いと思います。死ぬと思います。死にます。死にましょう。学期末にアレイズするんで安心してください。勝負はそこからだと言いたいところですが、次年度就職年次の皆さんはゾンビになってでもゲーム作ってもらいます。

## 環境設定

じゃあ早速作り始めよう!!と言いたいところですが、いちど最新の状態で環境設定をしておきたい。何故かと言うと DX12 は Windows SDK のバージョンが変わると同じコードが動かなくなったり、挙動が変わったりするので非常に面倒だが少なくとも WDK は揃えておきたい。

現在のところ WDK の最新版は 10.0.17134.12 である。ともかく WindowsSDK で検索したら WinSDKSetup.exe が落とせるので、落としたらインストールしてください。



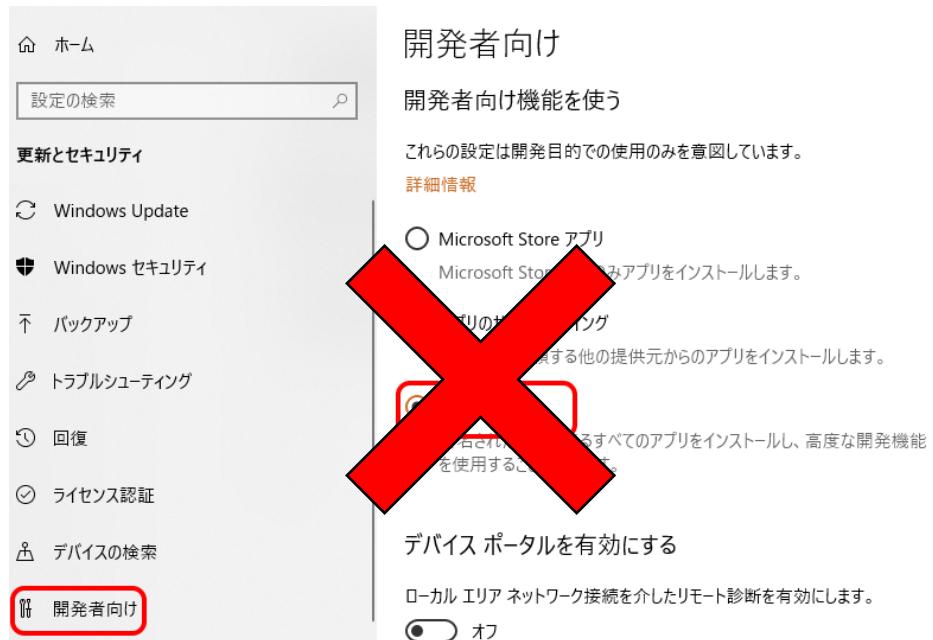
一応学校のサーバーにもインストーラを置いておきますが、インターネットにつながっていないとインストールできないのでご注意ください。

また、学校の外付けHDDにダウンロード済みのを入れてますので、遅い場合はそれを使用してください。

あ、そういえばこのバージョンの面倒くささがあるんで DX12 を活用したものを企業に送る際には SDK バージョンと Windows のバージョンは明記しておいた方がいいかも。

んでも、Windows の最新バージョンが 1803 ではあるんだけど、もしかしたら開発者モードじゃないと 1700 番台までしか更新できないかも…。

開発者モードにするには WindowsUpdate 画面で



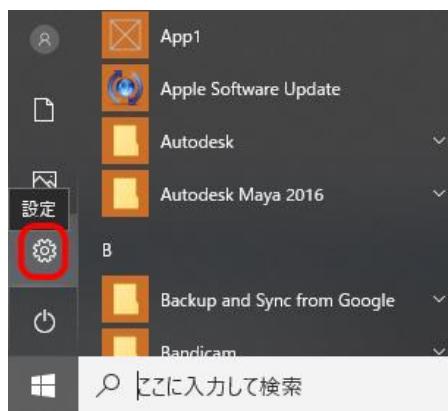
の状態にする必要があるのかなあ…もしくは

<http://www.atmarkit.co.jp/ait/articles/1807/24/news010.html>

に書かれてましたが、WindowUpdate の詳細設定にある延期日数によって出ないこともあります。

どっちにしてもちょっとめんどくさそうな部分(管理者権限が必要な部分)をいじることになりそうなので、最初の方はちょっと手間がかかると思います…とかいろいろとやってみましたが、何故か 1803(4 月に更新されたはずの最新版)に更新されません。もし現段階で 1803 なら更新の必要はありません。

バージョンの見方は



Window ボタン→歯車マーク

Windows の設定

Windows の設定

設定の検索

システム デバイス 電話

個人用設定 アプリ アカウント

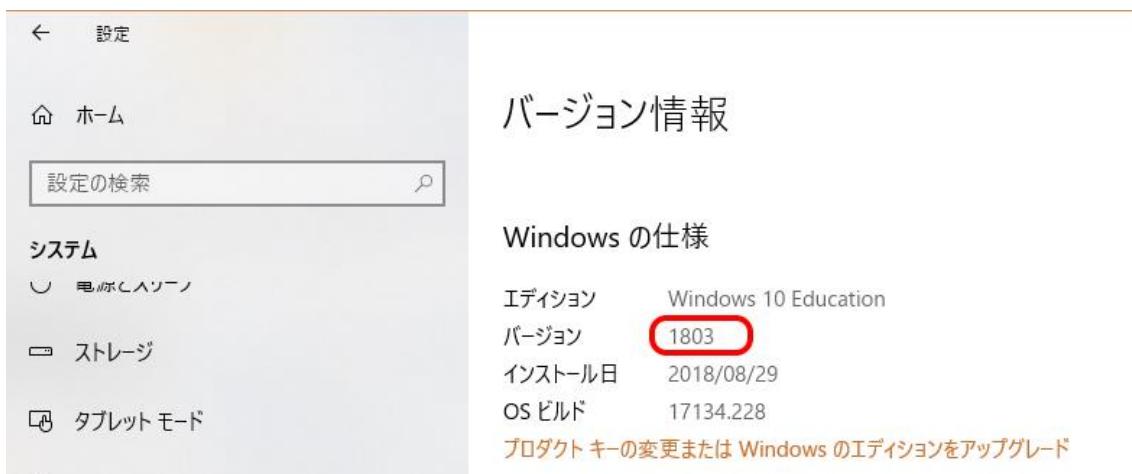
→システムを押すと、別画面が出てくるので左下の「バージョン情報」をクリック。

分辨率  
1920 × 1080 (推奨)

向き  
横

バージョン情報

そうすると右側に Windows のバージョンが表示されます。人によっては既に 1803 かもしれませんので、その人は Windows の更新は必要ありません。



そうでなければ直接インストールです。もちろん Windows Update 側に 1803 のインストールが見えてるならば、落とさないに越したことはありません。

<https://www.microsoft.com/ja-jp/software-download/windows10>

これをやるとガチで Windows の更新が始まります。クソ時間かかります。フリーズしどのんちゅうか?っていうくらい…。

無事終わったら Windows のバージョンが 1803 になっているはずですので、確認しておいてください。何度も再起動かかるんで待ちましょう。

さて、ここに手間取っている間に、昨年のテキストでも見ながら「予習」しておいてほしい。

## C++言語のおさらいとか追記とか

STLについて

おさらい

前回から当然のように STL を使っていると思いますが、とりあえず vector と map と list と set の区別はついているでしょうか?あと string に関してはな…。

コンテナ名	概要
vector	動的配列として使用できる。ガチでメモリが連続しているので、様々な用途に使える。 ただし、あまり push_back してると動的確保が頻繁に行われるため速度低下の原因となる。その場合は予め予測した大きさを reserve する。 また、要素が増えてくると配列同様に要素の挿入、削除の時間コスト

	高いが、殆どの場合は気にならない(それ以外のメリットの方が大きい)
map	連想配列として使用できる。インデックスではなく文字列を使用したり、飛び飛びのインデックスの配列としても使える。 また、map の要素は結局のところ 2 つ値<Key, Value>のペアに過ぎないため、そう捉えると様々な応用が可能。
list	名前の通りリスト構造でできているコンテナ。vector と違って、メモリが連続していない。要素と要素がリンクによってのみ繋がっているため、挿入と削除のコストが一定。 ただし、検索のコストは、要素が増えるほどに増えていく。大抵の場合は vector を使っておいた方が幸せである。
set	要素がソート済みとなるコンテナ。まあほとんどの場合 vector や map で事足りる。

つまるところやっぱり vector と map で十分って事やな!!

また、前回もちょっとだけ出てきましたが、algorithm などを使うとコードの量をぐっと減らせます。

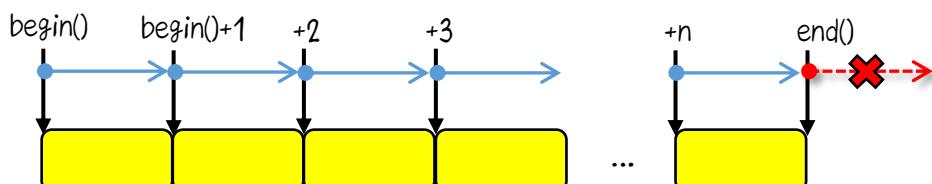
関数名	概要
for_each	指定された範囲内の要素に対して、特定の処理を行う
find	指定された値を検索しイテレータを返す
find_if	条件に合う要素を検索し、イテレータを返す
find_first_of	条件に合う要素の中で最初に見つけたイテレータを返す
find_last_of	条件に合う要素の中で最後に見つけたイテレータを返す(いらんかも)
min_element	要素の中から最も小さい要素を持つイテレータを返す
max_element	要素の中から最も大きい要素を持つイテレータを返す
sort	要素をソート
count	指定した値を持つ要素数を返す
count_if	条件に合致する要素数を返す
all_of	全てが条件を満たせば true
none_of	全てが条件を満たさなければ true
any_of	一つでも条件を満たせば true
fill	指定した範囲内に指定した値を代入。memset みたいになもん。
copy	指定した範囲内に、別の指定した範囲をコピー
copy_if	指定した条件を満たすもののみコピー

generate	指定した範囲に対して特定の関数を適用
transform	指定した範囲に対して特定の関数を適用した値を別のイテレータに代入(generateと似てるがちょっと違う)
remove	指定した範囲の要素を取り除く。実は削除されてないので注意。eraseとの組み合わせで本当に削除される。
remove_if	指定した条件に合致する要素を取り除く。↑と同様に削除されてないので注意。
replace	指定した値 A を別の指定した値 B に書き換える
replace_if	条件に合致する要素の値を、値 B に書き換える
unique	重複した要素を取り除く。実は削除されてない(略)
sample	指定された範囲内からランダムに要素をいくつか抽出する
shuffle	要素をミッドナイトシャッフルする

### イテレータについて

STL の vector などの map ってのは「コンテナ」と呼ばれて、色々なルールで要素の集合を格納するものです。そこは大丈夫だと思いますが、要素へのアクセスには色々とやり方があるんですよ。

で、一番基本的なアクセス方法が「イテレータ」と言うものを介すものです。今までしつつ何度も出て来てたんですが、基本的には begin() だの end() だので得ることができます。こいつはポインタのようなもんで、その要素のアドレス先頭を指示していると思ってく



ださい。

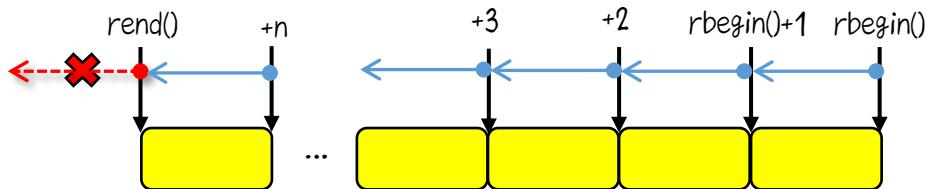
ここまで理解できましたか？

そして、\*などで「値」にアクセスする場合はこのイテレータから先に向かう方向にデータを取ってくるイメージなのだ。なので↑の図のように end() はその先にデータがないため、end() に対して値アクセス要求を出すとクラッシュ(というよりアサート)するわけ。

大丈夫？ この辺非常に大事なのでしっかりしてね？ そしてこのイテレータには変な親戚がいるのだ。それが「リバースイテレータ」というやつ。

### リバースイテレータについて

リバースという名前から想像がつくと思いますが、逆方向イテレータです。begin() とか rend() から取得します。データとの関係を図にするとこんな感じ。



ただ、逆方向に向いているだけのイテレータだが、これが「スキンマッシュアニメーション」の時とかに意外と使えるのだ。なんとなく頭には入れておこう。

### stringについて

`string`は何度も使ってるからもう使う事に対して苦手意識は持っていないかなと思います。`string`は結局のところ `vector<char>`に文字列用のメソッドをいくつか追加しただけのものです。

文字列用のメソッドとは

関数名	概要
<code>+オペレータ</code>	文字列連結
<code>==オペレータ</code>	同じ文字列か比較
<code>c_str()</code>	C言語の文字列表現(つまり文字列ポインタ)を返す
<code>length()</code>	文字列の長さを返す
<code>substr()</code>	部分文字列を返す

こういう関数が独自にあります。それ以外にもいろいろあつたりするので、これ以外は自分で調べましょう。

ちなみに `string`についてですが、文字列をチョットどうこうしようと思った人ならこう考えるんじゃないでしょうか?

そういうえば `string`ってのは `char` の集合体だよね? 文字って、1バイト文字だけじゃなくて「マルチバイト文字」ってあったよね? あれも `string`を使うの?

と、考えた人は良いところに目を付けていると思います。

実は `string`ってのは、`string`単品で宣言されているわけではなく、見てないところでこう宣言されています。

```
using string    = basic_string<char>; // シングルバイト文字
using wstring   = basic_string<wchar_t>; // ワイド文字
using u16string = basic_string<char16_t>; // UTF16 文字
using u32string = basic_string<char32_t>; // UTF32 文字
```

勘のいい人はお判りでしょうが、ワイド文字を扱う場合は `wstring` を使用します。当然ながら `string` と `wstring` には型の互換性がないため、もし変換をする際にはかなり面倒な処理が必要になります。

`MultiByteToWideChar` とか `WideCharToMultiByte` とかそういうのを介す必要があります。今の所は本題ではありませんので、飛ばしますが、文字列いじり始めると色々とややこしい事は心に留めておいてください。

### stringstreamについて

そういえば C 言語の時は、数字をフォーマットして文字に変換する関数として `sprintf` とかあったけど、`string` でそれに相当するものはないの？ `string` は連結と部分抽出しかできないの？

まあ `string` はそうなんだけどね。モチロンその辺の対応がない C++ ではない。

`stringstream`…つまり文字列ストリームと言うのがある。

例えば `cout` を使う際に

```
cout << "Hello World!" << endl;
```

なんて書くと標準出力に対して `HelloWorld` と出力できるだろう？ これの文字列版があるのだ。

まず

```
#include<sstream>
```

で使う準備だ。

次に文字列ストリーム用のオブジェクトを用意する。

```
ostringstream ss;
```

あとは `cout` の時と同じ要領で数字などをぶっこんでいく

```
ss << "Age=" << 42 << ", Height=" << 160;
```

などと書けば `ss` の中に "Age=42, Height=160" という文字列ストリームができている。確認した

ければ

```
cout << ss.str() << endl;
```

とでも書けばいい。なお str()ってのは文字列ストリームを文字列に変換する関数だ。

ちなみに 16進数とかにしたければ hex を書くことによってそれ以降が 16進数になる

```
ss << hex << "Age=" << 42 << ", Height=" << 160;
```

と書けば

```
"Age=2a,Height=a0"
```

という文字列が得られる。

ちなみに hex は 16進数、dec は 10進数、oct は 8進数である。

また桁数揃え等をしたければ setw を使用します。

0埋め等をしたければ setfill を使用します。

例えば

```
array<ostringstream,5> sss;
for (int i = 0; i < sss.size(); ++i) {
    sss[i] << "Texture_" << setw(3) << setfill('0') << i;
}
for (auto& ss : sss) {
    cout << ss.str() << endl;
}
```

とでも書けば

```
Texture_000
```

```
Texture_001
```

```
Texture_002
```

```
Texture_003
```

```
Texture_004
```

という出力が得られます。STLに関してはこんなもんですね…。

## C++の新しい仕様

前期の授業で C++の新しい仕様として、

auto, nullptr, 範囲 for 文, enum class ラムダ式

仕様	概要
auto 変数名	右辺値から型を推測して決定
nullptr	NULLとかタッセー奴じゃなくてちゃんとしたヌルオブジェクト
範囲 for 文	インデックスを指定しなくても、全要素のループを記述できる
enum class	先頭に型名を付加することで、従来の enum のような名前重複を防止
ラムダ式	{}でお手軽に関数オブジェクトを作れます

を紹介したわけなんですが…実は代表的で分かりやすい一部しか伝えてないんですね。

非常にありがたい資料があったので紹介しますが

<https://www.slideshare.net/Reputeless/c11c14>

をちょっと読んどいてください。157 ページと、川野先生に負けず劣らずボリューム多しですが、さっと目を通しておくといいと思います。

見る限り、意外と思ったのが代入におけるメモリコピーの無駄をなくす仕様に偏っており、逆に C++ らしくなったよなという印象です。

新しい仕様で知つておいた方がいいのは

- 右辺値参照とムーブセマンティクス
- 配列の範囲
- Emplacement
- minmax, iota

あと、なんか constexpr がないんでそれも本当は追加したい。

まあ一番ややこしい奴から行きましょうか

## 右辺値参照とムーブセマンティクス

実はムーブセマンティクス自体は、前期の unique\_ptr の時に一度だけ出てきているんですね。std::move を使用して所有権の移動を行ってところで。

で、右辺値参照の話なんですが

代入の際の

左辺値 = 右辺値

の図式で考えちゃうとたぶん理解できないし誤解する。

## とりあえず右辺値が変数ではなく一時オブジェクトの場

合に限って 考えてほしい。ちょっと限定的な状況の話だ。

で、今回の限定的な状況の話においてはプログラマが意識してプログラムの方法を変えると  
かそういう事じゃなくて、それっと C++のメモリ部分の仕様変更が行われてメモリ周りが効率化されたと考えてほしい。

だから僕らが頭を悩ます必要がなくて、逆に昔一生懸命効率化しようとしてた部分が不要になつたと思って良い。

ひとまず、右辺値も左辺値も変数である場合を見てみよう。

```
std::vector<int> lv;
std::vector<int> rv={0,1,2,3,4};
lv = rv;
for (auto v : lv) {
    cout << v << endl;
}
```

これは当然のように全値のコピーが発生します。これはいい。意図したとおりだから。しかし、もし以下のようないふり

```
lv = std::vector<int>(100, 0); //右辺値を破棄してもいいんだからコスト無駄じゃね?
for (auto v : lv) {
    cout << v << endl;
}
```

この場合も古い C++なら一時オブジェクトを生成して、さらにデータコピーが発生していたのだ。ところが新しい C++の場合ならば『所有権の移動』が行われコピーコストが発生しない。つまり、どういうことかと言うとコピーするまでもなく左辺に直接値が入るイメージである。

ここでちょっとした疑問が湧く。2つだ。

- 本当にコピーが発生していないのか？
- 所有権の移動と言うのは一時オブジェクトに参照が変わる事なのか？

いや、仕様なんやから信用しようや。確かにそうなんだけど気持ち悪いのだ。分かんないかな

あ…こういう気持ち。

まあ、そもそも皆さんとしても、検証もしないでって、いうのは納得いかないでしょ？という事で検証

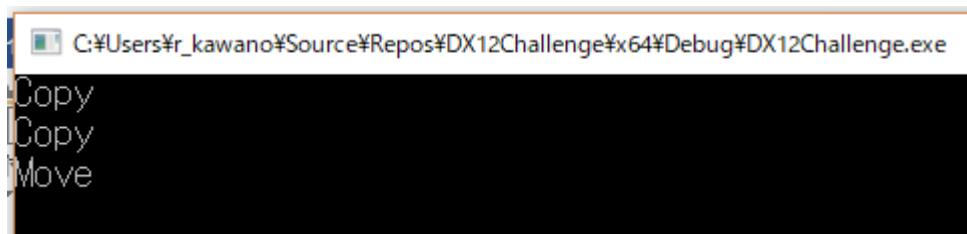
まず、こういうクラスを作る。

```
struct Boo {  
    Boo& operator=(const Boo& b) { // コピー代入オペレータ  
        cout << "Copy" << endl;  
        return *this;  
    }  
    Boo& operator=(Boo&& b) { // ムーブ代入オペレータ  
        cout << "Move" << endl;  
        return *this;  
    }  
};
```

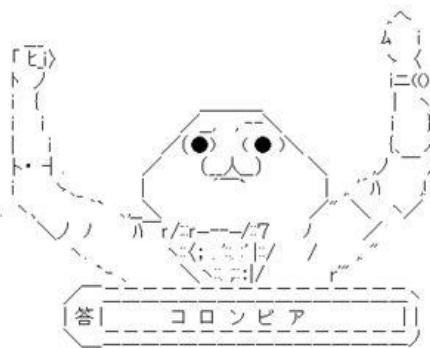
検証用なので、適当である。ともかく重要なのは、どちらが呼ばれるか、である。

```
Boo a, b, c;  
b = c;  
a = b;  
a = Boo();
```

こういうコードを書いて、結果を予想する。もし右辺値参照が言ってる通りの仕様であるならば、Copy Copy Move と表示されるはずだ。実行!!

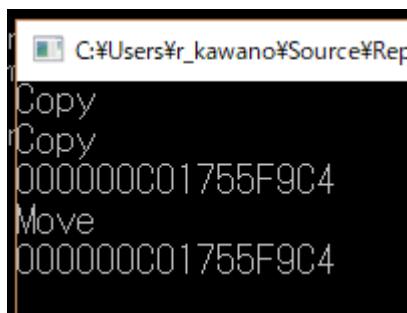


```
C:\Users\Yuki-kawano\Source\Repos\DX12Challenge\x64\Debug\DX12Challenge.exe  
Copy  
Copy  
Move
```



という事で、信用していいわけだ。おっともう一つ検証しなければならないのか。もし一時オブジェクトの所有権が左辺値に移るという事が、左辺値の参照先が一時オブジェクトを指し示すという事ならばアドレスが変わっているはず。逆に左辺値の値を直接書き換えていたりする状態ならアドレスは変わらない。さあどちらだ!!!

```
cout << hex << &a << endl;
a = Boo();
cout << hex << &a << endl;
```



The screenshot shows a terminal window with the following output:

```
C:\Users\r_kawano\Source\Repl
Copy
Copy
000000C01755F9C4
Move
000000C01755F9C4
```

オッケー!!安心してこの仕様に乗つかろう!!

ちなみにしれっと書いたけど

```
Boo& operator=(Boo&& b)
```

これがムーブオペレータである。ちなみに関数の戻り値として使用する場合だが

```
Boo GetBoo() {
    Boo b;
    return b;
}
```

(中略)

```
a=GetBoo();
```

とか書くとどうなんだろう?



アツハイ

ムーブしか発生してませんね。なるほどなるほど。

ちなみにこの右辺値参照を強制(つまりコピーを禁止)するにはどうするかというと

型名`&&` 变数名;  
の宣言を行う。

`Boo&& boo=GetBoo();`

であるとか

`Boo&& Foo = Boo();`

のように書くと右辺値は一時オブジェクトである事を強要される。つまり`&&`がつけられた左辺値には変数を右辺値に取ることはできない。

つまり

`Boo a, b, c;`

`Boo qoo = a; //OK`

`Boo& poo = a; //OK`

`Boo&& woo = a; //NG`

となる。あああああああああああややこしい。で、最後の行がNGになっているんだけど  
`std::move` を使えば強制的に所有権が移動し、OK 牧場となる。

`Boo&& hoo = std::move(a); //OK`

さて、この場合は流石に所有権の移動が行われているだろう。つまり

`cout << hex << &a << endl;`

`Boo&& hoo = std::move(a); //OK`

`cout << &hoo << endl;`

の結果は



A screenshot of a terminal window titled 'Terminal' showing two lines of memory dump output. The first line shows address 0000005F6D8FF874 followed by a colon and a question mark. The second line shows the same address again.

```
C:\Users\r_kawano\Source
0000005F6D8FF874
0000005F6D8FF874
```

となる。完全に所有者が`hoo`になってしまっている。では元の`a`はどうなっているのかという  
と

`cout << hex << &a << endl;`

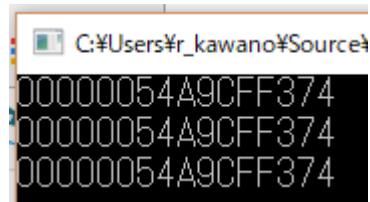
`Boo&& hoo = std::move(a); //OK`

`cout << &hoo << endl;`

`cout << &a << endl;`

と書いたところ

ということで、まだ`a`も所有権を持っていそうだが、一応仕様上は所有権がないということな



```
C:\Users\r_kawano\Source\repos\test>00000054A9CFF374
00000054A9CFF374
00000054A9CFF374
```

ので、どうなっても知らんよという事。つまり move してしまったら中身を使えると思うなという事。

もうガチでややこしい仕様やったわ…。ちょっと横道に逸れるだけのつもりやったのにガチ説明したわ。

## 配列の範囲

これはあれやな。配列の場所をイテレータとして使えるんやけど前まではポインタとそれ+範囲の先という指定をしておった。

つまり

```
int a[] = { 1,3,5,7,9,11 };
std::for_each(a, a + _countof(a), [](auto v) {cout << v << endl; });
```

こう書いていたのを

```
int a[] = { 1,3,5,7,9,11 };
std::for_each(begin(a), end(a), [](auto v) {cout << v << endl; });
```

こう書けるようになった。

いや~、でかい、でかいよこれは。ありがたい。

## Emplacement(emplace,emplace\_back)

例えばこのようなクラスを作る。

```
struct Vector3 {
    Vector3() {};
    Vector3(float inx, float iny, float inz):x(inx),y(iny),z(inz) {}
    float x, y, z;
};
```

こいつのベクタを作る。

```
vector<Vector3> vertices;
```

で、こいつに push\_back したいとする。但し、push\_back の引数は Vector 型であるため、  
vertices.push\_back(Vector3(1, 2, 3));

とする必要がある。

が、emplace\_back を使用すれば、Vector3 の一時オブジェクトを使う必要がない。

```
vector<Vector3> vertices;
```

```
vertices.push_back(Vector3(1, 2, 3)); // 一時オブジェクト生成 & コピー
```

```
vertices.emplace_back(4, 5, 6); // 直接生成 & 値の設定
```

```
vertices.emplace_back(7, 8, 9); // 直接生成 & 値の設定
```

```
std::for_each(vertices.begin(), vertices.end(), [](auto v) { cout << v.x  
<<, << v.y <<, << v.z << endl; });
```

ご覧のように emplace\_back の方がコード量も若干少なくなりますし、一時オブジェクトも作られないないので、場合によってはメモリの効率化にもつながります。若干だと思いますが。

### (おまけ)minmax と iota

最後はオマケみたいになもんやな…。まずは minmax から

<https://cppref.jp.github.io/reference/algorithm/minmax.html>

『同じ型の 2 つの値、もしくは initializer\_list による N 個の値のうち、最小値と最大値の組を取得する。』

最後の引数 comp は、2 項の述語関数オブジェクトであり、これを使用して比較演算をカスタマイズすることができる。

例えば

```
auto mm=minmax({ 0,1,2,3,4 ,8,2,-6,-2,3,100,120,-12 });
```

```
cout << mm.first << "≤value≤" << mm.second << endl;
```

なんて実行すると



The screenshot shows a terminal window with the path 'C:\Users\r\_kawano\Source'. Inside the window, the command 'cout << mm.first << "≤value≤" << mm.second << endl;' was run, resulting in the output '-12≤value≤120'.

なるほど

じゃあこれは…？

```
std::vector<int> rv = { 0,1,2,3,4 ,8,2,-6,-2,3,100,120,-12};
```

```
auto mm=minmax(rv.begin(), rv.end());
```

これはダメなんです。

begin の大きさと end の大きさを比べてしまうので、予想したような結果になりません。

minmax はあくまで二つの値もしくは initializer\_list の大きい方と小さい方を返すだけっぽいです…惜しいなあ。

次に iota ですが、これは要素を連番で埋めるというものです。

```
#include<numeric>
```

で使います。

```
std::vector<int> rv = { 0,1,2,3,4 ,8,2,-6,-2,3,100,120,-12};  
std::iota(rv.begin(), rv.end(), 0);  
for_each(rv.begin(), rv.end(), [](auto v) {cout << v << endl; });
```

とやると

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12
```

となります。

役に立つかなあ…。

ちょっと minmax と iota は微妙やつたかも。

## const と constexpr

もう一つ新しい仕様として const のもっと厳密な奴と言うかコンパイル時 const にあたる constexpr と言うやつが追加されている。

もともと C 言語の時に

```
#define RIGHT_VALUE 16
```

てな感じで定数を定義していたやつを

```
const int RIGHT_VALUE=16;
```

って書いてたんだよね。間違ってないんだけど、今まで const 使ってたから分かるでしょ？

所詮 const ってそのスコープの中で書き換えが発生しないって事やから実行時に右辺値が分かつてなくても OK なんよね。それはそれでいいし、使える仕様なんですが define の代わりに

使う意味の const としては弱くなつたんよね。

それで出てきたのが constexpr です。

コンパイル時に右辺値が決定できないとエラーを吐くわけ。だからマジックナンバー回避のための定数などには const ではなく constexpr を使うのが最近の流れです。

だから

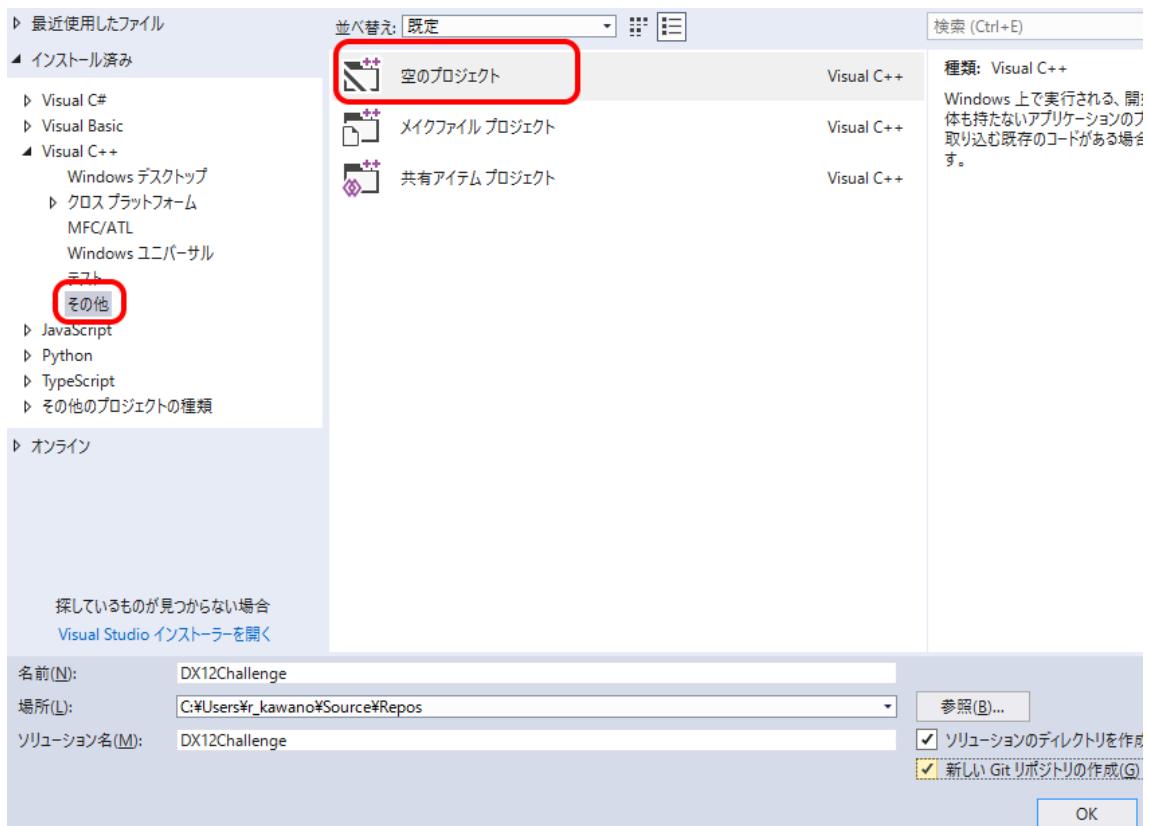
```
constexpr int ppp = Get(); // NG
```

```
const int qqq = Get(); // OK
```

というわけですね。

とりあえず新しい仕様としてはこんなもんかな。なんでこれ話してきたかと言うとたぶんこの先、僕がしぶと新しい仕様に沿ったコード書いてみんな混乱するかもしだへんので最初に言っておきました。

## まずはプロジェクトを作ろう



空のプロジェクトを作るとこからですね。

で、メイン関数を作るのでですが、mainでもWinMainでもどっちでもいいです。mainを出すとコンソール画面が出てくるくらいの違いしかないです(他にはHINSTANCE hInstでアプリハンドルを取ってこれるくらいしか違ひがない)。

ぼくはエラーが出しやすいという理由でコマンドラインの方を使います。

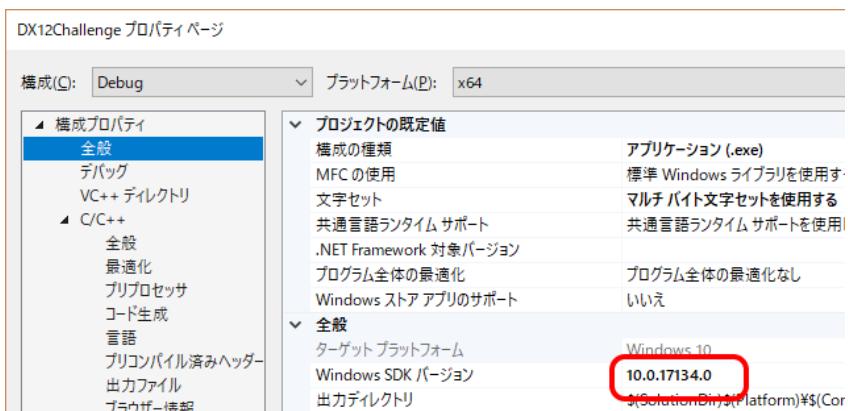
```
int main() { //①…コマンドラインありの時
    int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, int){ //②…コマンドラインなしの時
        cout << "Fuck You" << endl;
        getchar();
        return 0;
    }
}
```

別にどちらでも構いません。あ、Windows アプリケーションなので

```
#include<Windows.h>
```

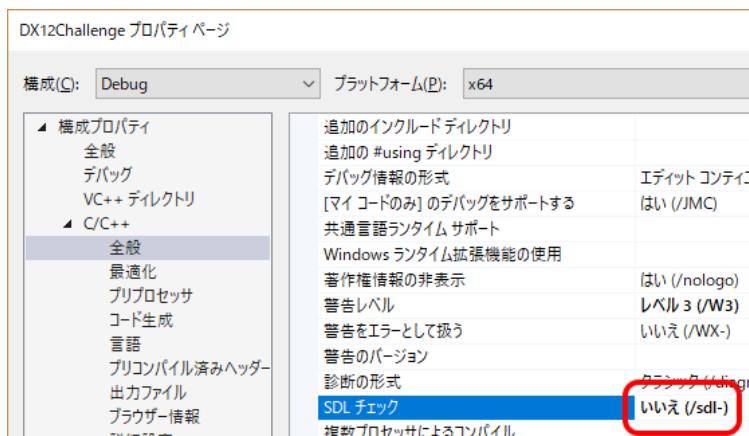
をインクルードはしておいてください。あ、ちなみにメイン関数があるcppはmain.cppとします。ただただ main を実行するのみの関数ですね。

で、プロジェクトの設定に入りますが

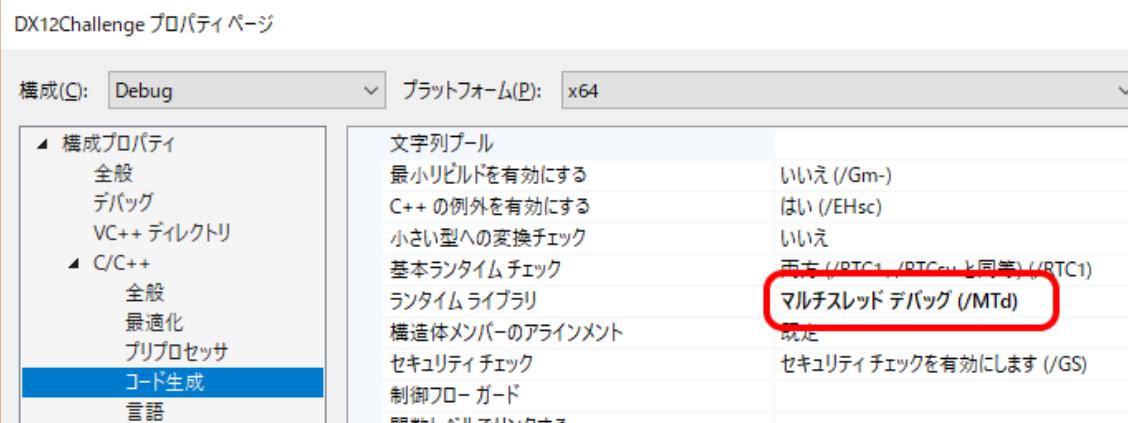


WindowsSDK が最新なのを確認してください。

次に C++ の全般で SDL チェックをいいえにします。



これしてないと、C言語標準のメモリ処理の関数やら文字列処理の関数やら出てきたときに\_sを使えとかローカルルール押し付けられてうざいので、こうしておきます。はい、次にちょっと面倒な部分ですが、コード生成を「マルチスレッドデバッグ」に書き換えます。



とりあえずウィンドウ出すまでならここまでで十分。

## じゃあウィンドウ出すか

ここから既に DxLib とは違いますので頑張りましょう。クソみたいに面倒くさいです。でも、細かく解説しません。ウィンドウ生成について細かく知りたい人は「猫でも分かるプログラミング」でも読んでください。

あまり main.cpp は汚したくないので Application クラス作りましょう。シングルトンで作っときましょうか。

で、DxLib の時と似たような感じで作っていきます。とりあえず

```
Initialize()  
Run()  
Terminate()
```

のそれぞれの関数を作つておきます。main 側からはこの3つを呼ぶだけにしておきたいです。もちろん Run の中にメインループが入っているイメージです。

で、ウィンドウ作るときにやたらと「ハンドル」ってのが出づります。

## HINSTANCEとかHWNDとか

一応 Windows とか DirectX 界隈では当然のように Handled-Body / パターン的のが使用されていて、実際 DxLib におけるリソースのほとんどの戻り値もこれだ。あれは int で使いやすいけどね。

ただ、Windows プログラミングにおいてこいつの型は単なる整数型(というかアドレス型)のくせに windows.h で typedef だかなんだかやってるせいで windows.h(windef.h) をインクルードしなければ使えないんですが、その値を Application クラス内で保持するためにはヘッダ側へのインクルードとなって、ちょっとイヤ。

こういった時に選択肢は3つくらいある。1つではないと思ってください。プログラミングに一つの答えなんて存在しない。必ずいくつか選択肢を見つけて、その中から明確な根拠で選んでください。場合によっては「一番シンプルで簡単そうだから」でもいいです。ただし、必ず選択肢をいくつか用意してください。

で選択肢ですが

1. 割り切ってヘッダでインクルードする
2. ハンドルをヘッダ側で使用せず cpp 側のグローバル的な領域(cpp スコープ)で宣言、初期化、使用する
3. Window などのデコレートクラスもしくは DxLib のように別テーブルで int 管理する

正直ここは後々の拡張性まで考えて、闊沢な時間さえあれば 3 番を用いたいところだけど、ここは 2 番くらいが時間的な意味でも妥当かなと思う。1 番はやっぱり生理的にイヤ。

とりあえず「ウインドウズのウインドウを作るのに、DxLib の時は DxLib\_Init で済んでたんだけど(ホントはそれだけじゃなくてデバイスとかその他初期化してくれる)、ウインドウを「作る」だけで

```
WNDCLASSEX w = {};
w.cbSize = sizeof(WNDCLASSEX); // これ、何のために設定するのさ…?
w.lpfWndProc = (WNDPROC)WindowProcedure; // コールバック関数の指定
w.lpszClassName = _T("DirectXTest"); // アプリケーションクラス名(適当でいいです)
w.hInstance = GetModuleHandle(0); // ハンドルの取得
RegisterClassEx(&w); // アプリケーションクラス(こういうの作るからよろしくって OS に予告する)
```

```
RECT wrc = { 0,0, WINDOW_WIDTH, WINDOW_HEIGHT };//ウィンドウサイズを決める  
AdjustWindowRect(&wrc, WS_OVERLAPPEDWINDOW, false); //ウィンドウのサイズはちょっと面倒なので関数を使って補正する
```

```
HWND hwnd = CreateWindow(w.lpszClassName,//クラス名指定  
_T("DX12テスト"),//タイトルバーの文字  
WS_OVERLAPPEDWINDOW, //タイトルバーと境界線があるウィンドウです  
CW_USEDEFAULT, //表示X座標はOSにお任せします  
CW_USEDEFAULT, //表示Y座標はOSにお任せします  
wrc.right - wrc.left, //ウィンドウ幅  
wrc.bottom - wrc.top, //ウィンドウ高  
nullptr, //親ウィンドウハンドル  
nullptr, //メニューハンドル  
w.hInstance, //呼び出しアプリケーションハンドル  
nullptr); //追加/プラメータ
```

このくらいのコードが必要になる。

で、ウィンドウ出るかい？まあ出ないんだな、これが『ウィンドウハンドル』というウィンドウの素を作っただけなんだわ。

ここでしくじることは99.9%くらいないと思うけど、あ、最初に#include<windows.h>しといてね。

もし失敗した時にキャッチできるよう

```
if (hwnd == nullptr) {  
    LPVOID messageBuffer = nullptr;  
    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM |  
        FORMAT_MESSAGE_IGNORE_INSERTS,  
        nullptr,  
        GetLastError(),  
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),  
        (LPWSTR)&messageBuffer,  
        0,  
        nullptr);
```

```
OutputDebugString((TCHAR*)mssageBuffer);
cout << (TCHAR*)mssageBuffer << endl;
LocalFree(mssageBuffer);
}
```

のコードも追加しておいた方がいいね。まだウィンドウは出ないよ。

ただ、ここまでがウィンドウの初期化処理なので、これを InitWindow 的な関数を作って、その中に入れておいてください。

で、一応ウィンドウ出すのなんてハンドルがあればあとは ShowWindow 関数で終わるんだけど

```
ShowWindow(hwnd, SW_SHOW); // ウィンドウ表示
```

これはちょっと InitWindow に入るのはやめておこう。どっちかというと Run に入れたいい。

次に DxLib の時にもあったと思うけどメインループだ。これは Run の中に書いてほしい。一応やり方としては無限ループがまして、ウィンドウ破棄のタイミングでループを抜けられるイメージで。

```
if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) { // OSからのメッセージを msg に格納
    TranslateMessage(&msg); // 仮想キー関連の変換
    DispatchMessage(&msg); // 处理されなかったメッセージを OS に投げ返す
}
```

```
if (msg.message == WM_QUIT) { // もうアプリケーションが終わるって時に WM_QUIT になる
    break;
}
```

こんな感じでループ抜けを書いておく。

で、Terminate()あたりに

```
UnregisterClass(w.lpszClassName, w.hInstance); // もう使わんから登録解除してや
```

と書けば、一応ウィンドウ表示まで完成です。ひとまずお疲れ様。言いたいところやけど、ひとつ忘れとったわ…いつも忘れる。ウィンドウプロシージャを忘れてた。こいつは

「コールバック関数」と言って、OSから呼ばれる関数を定義しなあかんのですよ。ということで定義

//めんどくせーし、あまりゲームに関係ないけど書かなあかんやつ

```
HRESULT WindowProcedure(HWND hwnd, UINT msg, WPARAM wparam, LPARAM lparam) {
    if (msg == WM_DESTROY) { // ウィンドウが破棄されたら呼ばれます
        PostQuitMessage(0); // OSに対して「もうこのアプリは終わるんや」と伝える
        return 0;
    }
    return DefWindowProc(hwnd, msg, wparam, lparam); // 標準の処理を行う
}
```

こいつはクラス内関数やなくて、通常の関数として宣言して置いてください。結果的には main.cpp が

```
#include "Application.h"
```

```
int main() { // ①…コマンドラインありの時
    // int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int){
        auto& app = Application::Instance();
        app.Initialize();
        app.Run();
        app.Terminate();
        return 0;
    }
}
```

このようになるようにしておいてください。

## 基礎知識説明①

### シェーダ

シェーダ、シェーダと言うとりますけれども、「誰やのあんた!?」って思ってる人も多いと思います。こいつは言うたら、表示に関わる言語で C/C++ と違うものです。GPU 上で動作する言語でございます。HLSL(High Level Shader Language)と言って C 言語っぽい見た目はしておりますが、別物でございますので、ご注意ください。

シェーダの種類は現在の所

- VS: 頂点シェーダ(バーテックスシェーダ)
- PS: ピクセルシェーダ(フラグメントシェーダ)
- GS: ジオメトリシェーダ

- HS:ハルシェーダ(テセレーションシェーダ)
  - CS:コンピュートシェーダ
- などの種類があります。

最初に使われるものが恐らく頂点シェーダとピクセルシェーダでござりますね。DX11以降においては少なくともVSとPSの2つを定義しないとそもそもポリゴンを1枚表示することもできません。

という事で、みなさん、このDX12の授業ではシェーダは避けて通れないんですね。フヒヒ。

ちなみにこの中に仲間外れがいます。CS:コンピュートシェーダです。そもそもシェーダというものは名前から想像できると思いますが、本来は陰影をつけるための計算をするものでした。

ところが、GPU自体が並列処理に優れているという理由で「シェーディング」や「幾何学」と関係のない部分で使用されました。これをGPGPUと言い、それを行うためのシェーダを「コンピュートシェーダ」と言います。ですから、この後に説明する「レンダリングパイプライン」の環から外れた存在なのです。