C++界隈の話題

C++界隈で繰り広げられるうんちくや、トピック、ＴＩＰＳ

- ゲーム制作関連
  - カウントダウンタイマークラスを作ってみようｚ
  - AABB、OBBとは
  - 球と球の当たり判定(3D)（by チョコミント）
  - カプセルコライダーについて（by チョコミント）
  - 箱(AABB)と球の当たり判定
  - 2点間の距離

- 共有したくなる技術や知識
  - シングルトンとやらについて
  - singletonのサンプル？
  - 関数ポインタについて(by チョコミント）
  - 演算子のオーバーロードしてみようz
  - Visual Studioでimport stdする
  - stl(vector)の３種類のループの仕方

- C++よもやま話
  - C++とWindowsと非同期処理

- あいつは自分から見て右にいるの？左にいるの？
- 優先度付きキューの話（メロリンキュー）

ゲーム制作関連

C++とゲーム制作に関する話題

カウントダウンタイマークラスを作ってみようｚ

CDTimer.h

```
#pragma once
#include "Engine/GameObject.h"

class CDTimer :
    public GameObject
{
private:
 double CountDownTimer_;  //現在の残り時間
 double TIMER_INIT_TIME_; //初期時間
 bool isTimerRun_;//タイマーが動いているかどうか？
 DWORD oldTime_;
public:
 //コンストラクタ
 //引数：parent  親オブジェクト（SceneManager）
 CDTimer(GameObject *parent);
 CDTimer(GameObject *parent, double _initTime);


 //初期化
 void Initialize() override;

 //更新
 void Update() override;

 //描画
 void Draw() override;

 //開放
 void Release() override;

public:

 bool IsTimeOver();//タイマーは０になりましたか？ＹＥＳ？ＮＯ？
 void ResetTimer();//タイマーを初期時間に戻す
 void StartTimer();//タイマーをスタートします
 void StopTimer();//タイマーをストップします
 void SetInitTime(double cdTime) { TIMER_INIT_TIME_ = cdTime; ResetTimer(); }
 double GetTime() { return(CountDownTimer_); }
};
```

CDTimer.cpp

```
#include "CDTimer.h"


const int DEF_TIMER_TIME{ 5 };

CDTimer::CDTimer(GameObject* parent)
 :TIMER_INIT_TIME_(DEF_TIMER_TIME),
 CountDownTimer_(DEF_TIMER_TIME),
 isTimerRun_(true)
{
 oldTime_ = timeGetTime();
};
```

```
CDTimer::CDTimer(GameObject* parent, double _initTime)
 :TIMER_INIT_TIME_(_initTime),
 CountDownTimer_(_initTime),
 isTimerRun_(true)
{
 oldTime_ = timeGetTime();
};

bool CDTimer::IsTimeOver()
{
 return(CountDownTimer_ <= 0);
}

void CDTimer::ResetTimer()
{
 CountDownTimer_ = TIMER_INIT_TIME_;
 StartTimer();
}

void CDTimer::StartTimer()
{
 isTimerRun_ = true;
}

void CDTimer::StopTimer()
{
 isTimerRun_ = false;
}

void CDTimer::Initialize()
{

}

void CDTimer::Update()
{
 DWORD nowTime = timeGetTime();
 //下staticは要らんかった。。。
 //static int deltaTime = nowTime - oldTime_;
 int deltaTime = nowTime - oldTime_;
 if (isTimerRun_)
  CountDownTimer_ = CountDownTimer_ - (float)deltaTime/1000.0;
 oldTime_ = nowTime;
}

void CDTimer::Draw()
{
}

void CDTimer::Release()
{
}
```

ゲーム制作関連

AABB、OBBとは

AABB、OBBとは

こんにちは、チョコミントです。
今回はAABBとOBBについて聞きなれてない人も多いと思うので簡単に説明します。

- AABBとは**軸平行境界ボックス**といって**箱の各面の法線が座標軸と平行**なものです。
  箱の横と縦と奥へのベクトルそれぞれがXYZと平行ということですね。
- OBBとは**有向境界ボックス**といって**箱の各面の法線が座標軸と平行ではない**も のです。
  なんとなく理解できたでしょうか。

AABBとOBBでは当たり判定をする際に処理速度が変わる！？

AABBとOBBでは**当たり判定をする際の処理速度**がかなり変わってきます。
箱の各軸が**座標軸と平行ではない**だけで衝突判定がものすごくめんどくさいんです.....
簡単にいうと**箱の各軸を分離軸としてその分離軸で射影してあげる**必要があるからです。
そして射影しているときに**分離超平面がみつかれば衝突していない**ということになります。

この手法は**様々な3Dプリミティブの当たり判定**で使われます。

1. OBBとOBB
2. OBBと球
3. OBBとカプセル
4. OBBと三角形

これ以外でもありそうですが.....
文章だけ読むと複雑そうには思いますが、実際に実装してみると仕組みさえ理解してしまえばわかると思います。
分離軸で射影して分離超平面を見つける手法もいつか記事で書きたいと思います。

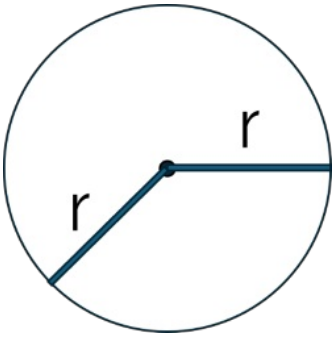
最後に

今回はAABBとOBBについての記事でした。
分からないことや質問などがあれば気軽にコメント待っています！！

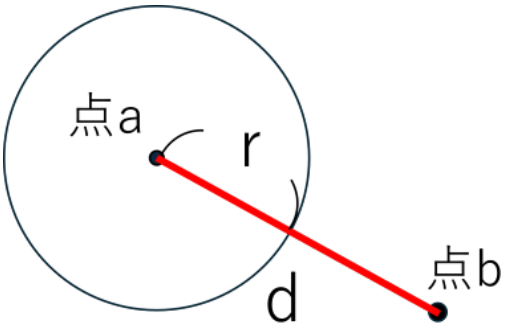
球と球の当たり判定(3D)（by チョコミント）

球と球の当たり判定

こんにちは、チョコミントです。 今回から当たり判定について軽く触れていこうかなと思います。 まずは比較的簡単な**球と球の当たり判定**についてです。 もう知っている方も多いと思いますが、よければ見ていってください。



球は**中点から円周の距離が全方向半径(r)の長さ**で構成されています。 では、球と点が衝突しているか確かめるにはどのように考えればいいのでしょうか。



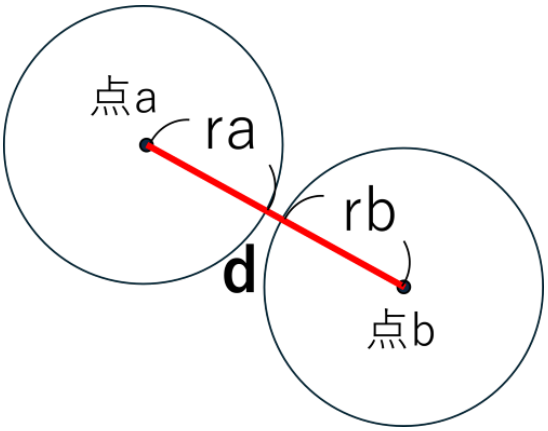
上の図では点a(球体に中点)から点bの距離をdとし、球体の半径をrと定義しています。 もうぴんっときた人もいるかもしません。 **dの距離がr(半径)以下だったら衝突**しているのです。 ってことは半径はもうわかっているので、**点aと点bの距離**が分かれば衝突しているかわかりそうですね！！

点と点の距離

点と点の距離は**三平方の定理**を利用して求めることができます。 3次元の場合はこのような感じです。

```
XMFLOAT3 c = { a.x - b.x,a.y - b.y,a.z - b.z };
return sqrtf((c.x * c.x) + (c.y * c.y) + (c.z * c.z));
```

簡単に説明するとX,Y,Z軸でそれぞれの始点(例:点a)から終点(例:点b)へのベクトルを求めて、 それらの合成ベクトルの長さが距離となるのです。 点と点の距離が分かってしまえばもう球と球の当たり判定は簡単です。



この図をみて理解した人も多いと思いますが、 **dの距離が(ra(半径) + rb(半径))以下だったら衝突**しているのです。

```
//点と点の距離を求める
XMFLOAT3 a, XMFLOAT3 b;
XMFLOAT3 c = { a.x - b.x,a.y - b.y,a.z - b.z };
float d = sqrtf((c.x * c.x) + (c.y * c.y) + (c.z * c.z));

float ra = 0.5f; //球体aの半径
float rb = 1.0f; //球体bの半径

//aとbの距離がそれぞれの半径を足した合計値以下なら当たっている
if (ra + rb >= d)
{
 //衝突している
}
```

サンプルプログラムです。

最後に

今回は球と球の当たり判定についての記事でした。 物足りない人も多かったのではないでしょうか。 次回は球と箱(AABB)の当たり判定について記事を書いてみます。

カプセルコライダーについて（by チョコミント）

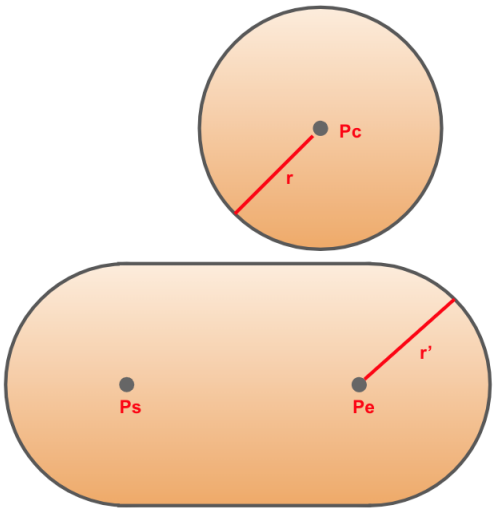
コライダーでよく使われるカプセル(線分ボリューム)コライダーについて

こんにちは、チョコミントです。 今回はカプセルコライダーについてお話ししようと思います。

簡単に説明すると上の写真のような形をしているコライダーです。

線分ボリュームとは

カプセルのことを別名で「線分ボリューム」とも言います。 なぜ「線分ボリューム」と呼ばれるのかというと、線分の全方向に対して指定した半径分線分を 太らせればカプセルが完成するからです。



上の写真を見ていただくとPsとPeを結ぶ線分に対して全方向にr'太らせればカプセルが完成するのが分かりやすいです。

なぜカプセルコライダーは頻繁に使われるのか

1. 衝突判定の際に処理が軽い
2. 人の形にフィットしやすい

この2つくらいだと思います。 1つずつ簡単に説明していきます。

衝突判定の際に処理が軽い

みなさんは球体型のコライダーは、衝突判定の際に処理が比較的軽いことは知っていますか?? なぜなら点との距離を求めればいいだけだからです。

さっきの話の中ではカプセルは「線分を太らせただけ」でしたよね。 ってことはカプセルは線分との距離を求めればいいだけなんです！！ 処理が軽そうですよね！！ ただこの記事を見てくれているみなさんの中には「線分と点」、「線分と線分」との距離って どうやって求めればいいのかわからない人もいますよね。 そこに関しては、また違う記事で説明します。

人の形にフィットしやすい

ゲームでは人型のプレイヤーや敵などが多く登場します。 単純に形が人型にフィットしやすいからです。

最後に

今回はカプセルについての記事でした。 質問やわからなかったことなどがあればぜひ気軽にコメントしてください！

箱(AABB)と球の当たり判定

箱(AABB)と球の当たり判定　by チョコミント

こんにちは、チョコミントです。
今回は**箱(AABB)と球**の当たり判定についてです。

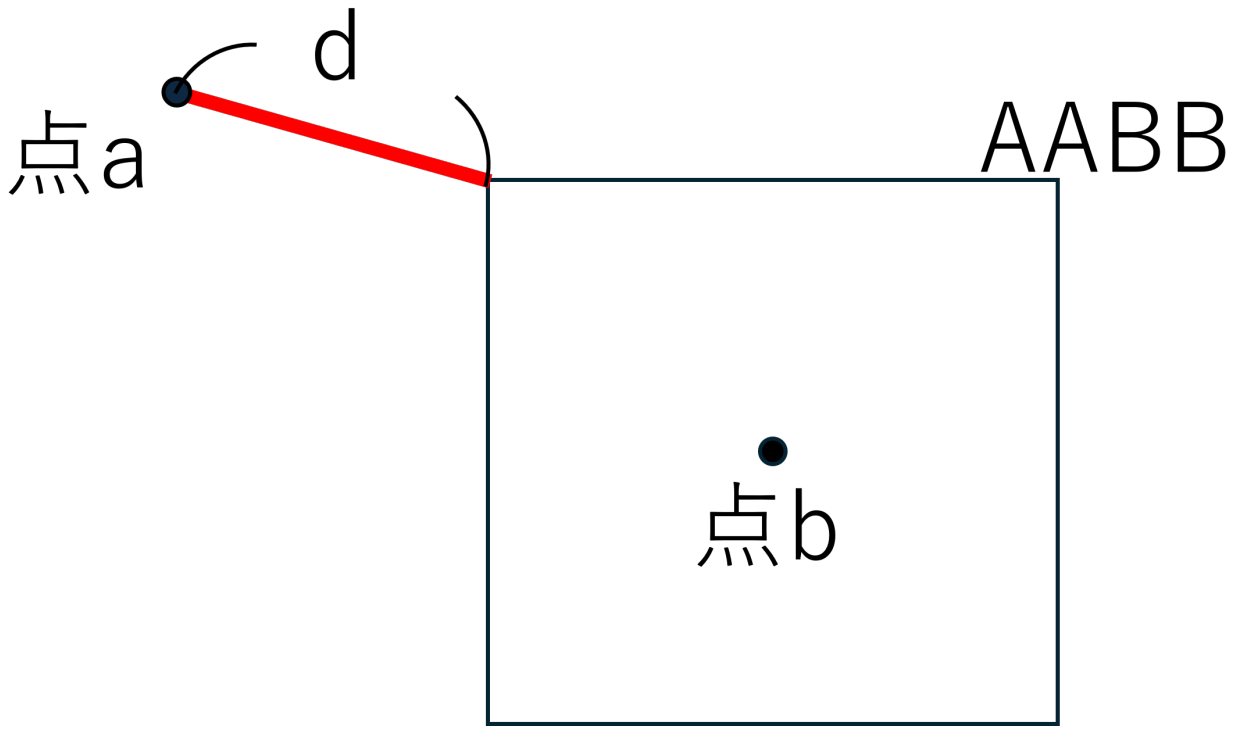
AABB,OBBとは

AABBとOBBについて聞きなれてない人も多いと思うので簡単に説明します。

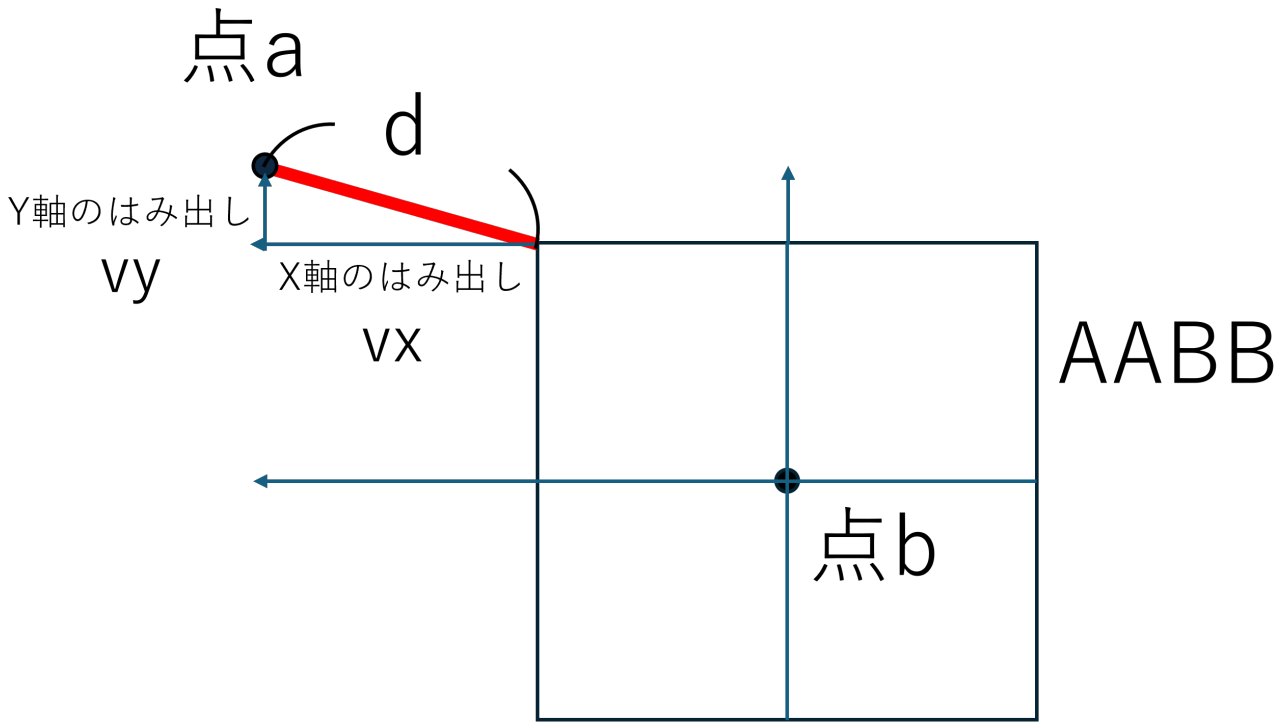
- AABBとは**軸平行境界ボックス**といって**箱の各面の法線が座標軸と平行**なものです。
  箱の横と縦と奥へのベクトルそれぞれがXYZと平行ということですね。
- OBBとは**有向境界ボックス**といって**箱の各面の法線が座標軸と平行ではない**も のです。
  なんとなく理解できたでしょうか。
  AABBとOBBでは**当たり判定をする際の処理速度**がかなり変わってきます。
  OBBについての当たり判定もいつか記事に書く予定です。

AABBと点の距離

さて本題に入っていきましょう。



上の図にある通り、AABBと球の当たり判定を行うには**AABBと点の距離**が分かればいいのです。(図に書いてあるdです)
AABBと点の距離を調べるには、**AABBの各軸(XYZ)で点がどのくらいAABBからはみ出ているか**を調べればいいのです。



上の図ではAABBのX軸、Y軸で点がはみ出ている分、vx、vyでベクトルを作ってみました。
あれこの形どこかで見たことある人いませんか??
そうです、**点と点の距離を求めるのとそっくり**ですね。
つまり**vx、vy、vz(図にはない3Dの場合)の合成ベクトルを求めればAABBと点の距離**になるのです！！簡単ですね！！

```
//AABBの各軸
    XMVECTOR boxX = { 1,0,0 };
    XMVECTOR boxY = { 0,1,0 };
    XMVECTOR boxZ = { 0,0,1 };

    //AABBの各軸の大きさ
    float boxXSize = 0.5f;
    float boxYSize = 0.5f;
    float boxZSize = 0.5f;

    //AABBの中点と点の位置
    XMFLOAT3 boxPos = { 0,0,0 };
    XMFLOAT3 pointPos = { 1,1,1 };

    //各軸での距離を調べる
    float x = fabs(boxPos.x - pointPos.x);
    float y = fabs(boxPos.y - pointPos.y);
    float z = fabs(boxPos.z - pointPos.z);

    //最終的な距離を表すベクトル
    XMVECTOR dis = XMVectorZero();

    //各軸のはみだし距離をdisに加算
    if(x > boxXSize)
        dis += boxX * (x - boxXSize);
    if(y > boxYSize)
        dis += boxY * (y - boxYSize);
    if(z > boxZSize)
        dis += boxZ * (z - boxZSize);

    //disの長さがAABBと点の距離
    return XMVectorGetX(XMVector3Length(dis));
```

サンプルコードです。
ここまでくればAABBと球の当たり判定はちょこっとたすだけです！
AABBと点の距離が球体の半径以下だったら衝突していることになりますね！！

```
//半径以下なら
if (radius >= XMVectorGetX(XMVector3Length(dis))
```

```
{
    //衝突している
}
```

これでAABBと球の当たり判定は完璧です！

最後に

今回はAABBと球の当たり判定についての記事でした。
分からないことや質問などがあれば気軽にコメント待っています！！

ゲーム制作関連

2点間の距離

2点間の距離のはなし

ゲームを作っているとよく2次元でも3次元でも2点間の距離を出したくなるよね。
それで登場するのが次の公式

$p_1(x_1,y_1,z_1)$ と $p_2(x_2,y_2,z_2)$ があるときに、その2点の距離は

$$ dist = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2} $$

で計算できる。というもの。
これはこれでよく使うし、計算は合ってます。（結構座標書くときに間違って混乱するよね）

DirectXのプログラム作ってるみなさんでは、ソースコードでは以下の様になると思います。

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    //3次元空間上の2点
    XMFLOAT3 v1{ 2.0, 3.0, 4.0 };
    XMFLOAT3 v2{ 4.0, 6.0, 8.0 };

    //距離計算 2点間の距離
    float dist1 = sqrt((v1.x - v2.x) * (v1.x - v2.x)
                + (v1.y - v2.y) * (v1.y - v2.y)
                + (v1.z - v2.z) * (v1.z - v2.z));
    cout << "距離（ふつうの計算）　：" << dist1 << endl;
}
```

XMFLOAT3とかめんどくさい。。。

そんで、DirectXMathに以下のような関数があります。
XMVector3Length 関数

ある2点間の距離は、その2点を始点と終点（どっちがどっちでも同じだよ）とするベクトルの長さに他ならないので、2点をベクトル化してしまえば、この関数が結果を返してくれます。

```
#include <iostream>
#include <DirectXMath.h>

using namespace DirectX;
using std::cout;
using std::endl;

int main()
{
    //3次元空間上の2点
    XMFLOAT3 v1{ 2.0, 3.0, 4.0 };
    XMFLOAT3 v2{ 4.0, 6.0, 8.0 };

    //距離計算 2点間の距離
    float dist1 = sqrt((v1.x - v2.x) * (v1.x - v2.x) + (v1.y - v2.y) * (v1.y - v2.y) + (v1.z - v2.z) * (v1.z - v2.z));
    cout << "距離（ふつうの計算）　：" << dist1 << endl;

    XMVECTOR xv1 = XMLoadFloat3(&v1);
    XMVECTOR xv2 = XMLoadFloat3(&v2);
    float dist2 = XMVectorGetX(XMVector3Length(xv1 - xv2));

    cout << "距離（ベクトルの長さ）：" << dist2 << endl;
```

```
}
```

`XMVector3Length`関数は戻り値が`XMVECTOR`型です（距離とか長さなのに）、これは最適化とかメモリ転送の都合でデータの幅を合わせておいて、大きさの揃ったデータだけを使うことで最適化を図っているからです。 戻ってくるデータは`{length, length, length, length}`　になっています。 なので、`x,y,z,w`のどれかをとってやれば長さが取り出せます。

`XMVECTOR`からデータを取り出す命令は

- XMVectorGetX
- XMVectorGetY
- XMVectorGetZ
- XMVectorGetW

などが用意されているのでお好きなものをどうぞ。 これで、距離計算はDirectXMathさえ使っていれば簡単だね。

共有したくなる技術や知識

プログラミング関連でみんなに共有しておきたくなるテクニックやテンプレート、いっつも忘れるからメモっておいた方がいいテクニックなどを書き留めたもの（言語は問わない）が、増えてきたら言語で分けたい気持ちもある。現状はＣ＋＋かな。

共有したくなる技術や知識

シングルトンとやらについて

シングルトンパターン

シングルトンパターンについては、そんなに語ることもないと思うけれども、そのプログラムの中でインスタンスを１つに統一して使うデザインパターンのこと。

ゲームで言うと、ゲームの中でプレイヤーキャラを1体だけだして、2人目以上を登場させようとしたら自動的に出てこないように抑制できる仕組みです。

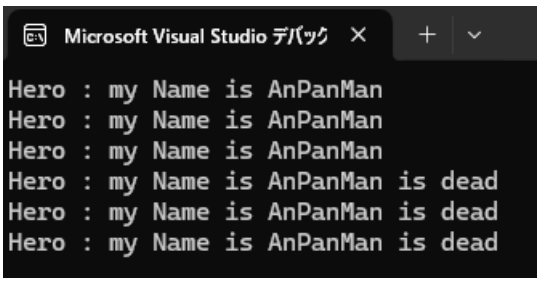
作ってみるよ

まずは普通にクラス作った場合を見てみよう（見る必要ない気もするけど）

✔ソースコード

```
#include

class Hero
{
private:
 std::string name;
public:
 Hero()
  : name("AnPanMan")
 {
  std::cout << "Hero : my Name is " << name << std::endl;
 }
 ~Hero()
 {
  std::cout << "Hero : my Name is " << name << " is dead" << std::endl;
 }
};

int main()
{
 Hero hero;
 Hero hero2;
 Hero hero3;
 return 0;
}
```

✔実行結果



こんな感じで、いくらでも同じキャラのインスタンスが生成できます（まぁインスタンスってそういう話だったよね）。
次に、このワールド（世界）にアンパンマンは一人しか存在しないように変更してみる。

シングルトンパターンとやらを試してみる

✔シングルトンのつくりかた♡

0. （オプション）クラスを継承不可に指定
1. コンストラクタを private に指定する 2.静的なポインタ変数を宣言する
   - んで、こいつにインスタンスのアドレスを格納する

    - 後は、この変数だけが唯一のインスタンスになるように頑張る
2. 静的なポインタを初期化する
3. インスタンスを返す**静的な GetInstance 関数**を実装する
    - 静的な変数 == nullptr
        - ⇒新しくインスタンスを生成する
    - 静的な変数 != nullptr
        - ⇒静的な変数（のポインタか参照）を返す

0.（オプション）クラスを継承不可に指定

クラス宣言するときに、後ろに`final`キーワードを付加するとそのクラスは継承できない！とコンパイラに伝えることができるよ。

```
class Hero final
{
private:
//省略
};
```

1. コンストラクタを private に指定する

```
class Hero
{
private:
 std::string name;
public:
 Hero(){//省略
   }
   //省略
}
```

これを ↓　以下の様に変更する！これで、勝手にコンストラクタが呼べなくなる＝好きにインスタンスを生成することができなくなる。
じゃぁどうやってインスタンス作るの？ってことになるが、それは次。

```
class Hero final
{
//省略
private:
   std::string name_;
 Hero()
  : name("AnPanMan")
 {
  std::cout << "Hero誕生 : my Name is " << name_ << std::endl;
 }
   ~Hero()
 {
  std::cout << "Hero : my Name is " << name_ << " is dead" << std::endl;
 }
public:
//省略
};
```

2.静的なポインタ変数を宣言

お外から見えないところ＝privateなところに、`static Hero* s_hero_;`みたいな感じで、静的なメンバ変数を生成

```
class Hero final
{
private:
 //クラスの静的なポインタを宣言する
 static Hero* s_hero_; // これがスタティックなポインタ変数
 std::string name;
 Hero()
  : name("AnPanMan")
 {
  std::cout << "Hero誕生 : my Name is " << name << std::endl;
 }
```

```
 ~Hero()
 {
  std::cout << "Hero : my Name is " << name << " is dead" << std::endl;
 }
public:
//省略
};
```

3. 静的なポインタを初期化する

```
class Hero final
{
private:
//盛大に省略
};
//どこかグローバルスコープなところで
Hero* Hero::s_hero_ = nullptr;// 静的メンバ変数の定義
```

4. インスタンスを返す**静的な GetInstance 関数**を実装する

ここが一番のポイント！
nullptrで初期化された静的変数を`GetInstance`が呼ばれるたびに確認し、インスタンスが既に存在していれば、そのインスタンスを返す。
nullptrだった場合は、新しいインスタンスを作って、それを返す。
逆に言うと、nullptrだった時だけインスタンスが生成されるってだけ。

```
public:
 static Hero& GetInstance();// 静的な参照を返す関数（宣言文）
};

// Hero クラスのインスタンスを取得する
Hero* Hero::GetInstance()
{
 if (s_hero_ == nullptr) //インスタンスが無かったら
 {
  s_hero_ = new Hero(); //新しく生成
 }
 return *s_hero_;//インスタンスがあればそれを返す
}
```

実行してみようズ

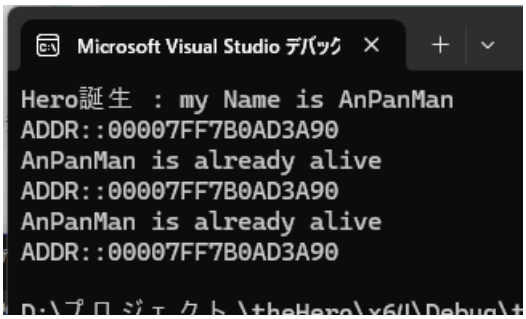
プログラムの全体像と、実行のためのmain関数を以下に示す。

```
#include <iostream>

class Hero final
{
private:
 // Graphics クラスの静的なポインタを宣言する
 static Hero* s_hero_;
 std::string name_;
 Hero()
  : name_("AnPanMan")
 {
  std::cout << "Hero誕生 : my Name is " << name_ << std::endl;
 }
 ~Hero()
 {
  std::cout << "Hero : my Name is " << name_ << " is dead" << std::endl;
 }
public:
 static Hero& GetInstance();


};
```

```
// Hero クラスのインスタンスを取得する
Hero& Hero::GetInstance()
{
 if (s_hero_ == nullptr)
 {
  s_hero_ = new Hero();
 }
 else
 {
  std::cout << s_hero_->name_ << " is already alive" << std::endl;
 }
    //ついでにインスタンスの持ってるアドレスを表示する！
 std::cout << "ADDR::" << std::hex << &s_hero_ << std::endl;
 return *s_hero_;
}

// 静的メンバ変数の定義
Hero* Hero::s_hero_ = nullptr;

int main()
{
 static Hero& hero = Hero::GetInstance();
 static Hero& hero2 = Hero::GetInstance();
 static Hero& hero3 = Hero::GetInstance();
 return 0;
}
```

実行結果

`GetInstance`した時に、ついでに、インスタンスのアドレスを表示するようにしてみた。 つまり、同じインスタンスが参照されていれば、同じアドレスが毎回帰ってくるし、毎回作っちゃっていれば、毎回違うアドレスになるはずだよね！



2回目以降の`GetInstance`では、初めに作られたインスタンスが参照されて返されていることがわかる！

でもね

一部では、結合性が云々とかでやたら嫌われてたりもするので、自分の属しているグループの流儀に習って使いましょう！

共有したくなる技術や知識

singletonのサンプル？

シングルトンパターン比較

シングルトンパターンじゃなく作ったPlayerクラスをマウスクリックするたびにnewする動画
staticなメンバ変数`playerCount`がクリックするたびに増えて、自分の番号＝playercountになるよ。

つまり、クリックするたびに、theMain.cppで宣言されている`vector<Hero*> heroes`に新しく生成されたPlayerのオブジェクトが追加されている。

https://www.youtube.com/embed/Q1kJWBAbn7E?si=gY9Rfz0x15K6wo7V

適当なソースコード

```
//Main.cpp
//省略
namespace
{
 const int BGCOLOR[3] = {0, 0, 0}; // 背景色{ 255, 250, 205 }; // 背景色
 int crrTime;
 int prevTime;
 vector<Hero*> heroes; // プレイヤーのポインタを格納するベクター
}

//省略
//ここにやりたい処理を書く
if ((oldMouseInput & MOUSE_INPUT_LEFT) == 0 && (mouseInput & MOUSE_INPUT_LEFT) != 0)
{
 int x, y;
 GetMousePoint(&x, &y);
 Hero *p=new Hero();
 p->SetPosition(x, y);
 p->SetPosition(x, y);
 heroes.push_back(p);
}


for (auto& hero : heroes)
{
 hero->Update();
 hero->Draw();
}
ScreenFlip();
//省略
```

```
#pragma once
class Hero
{
private:
 int posx_, posy_; // プレイヤーの座標
 int myNumber_;// プレイヤーの番号
 int hImage_; // プレイヤーの画像
 static int playerCount_; // プレイヤーの数
public:
 Hero();
 ~Hero();
 void SetPosition(int x, int y);
 void Update();
```

```
 void Draw();
};

Hero::Hero()
 : posx_(0), posy_(0), myNumber_(0), hImage_(-1)
{// プレイヤーの画像を読み込む
 hImage_ = LoadGraph("Assets/tiny_ship5.png");
 if (hImage_ == -1)
 {
  // 画像の読み込みに失敗した場合の処理
 }
 playerCount_++;
 // プレイヤーの番号を設定
 myNumber_ = playerCount_;
}

Hero::~Hero()
{
}

void Hero::SetPosition(int x, int y)
{
 posx_ = x;
 posy_ = y;
}

void Hero::Update()
{
}

void Hero::Draw()
{
 DrawGraph(posx_, posy_, hImage_, TRUE);
 DrawFormatString(posx_+32+2, posy_-2, GetColor(255, 255, 255), "Player %d", myNumber_);
}
```

singletonパターンで書いてみる

singletonにすると、うまくプログラムが動いていれば、実行プログラム内でPlayerのインスタンスは一度newされたらその一つだけになる。（はず）
上の通常の実装と同じように、クリックするたびに、インスタンスのポインタをリストに入れていくとインスタンスがプログラム内で一つしかないなら、同一のアドレスが何度もリストに登録されているはずである。

またまた適当なソースコード（singleton風）

```
//theMain.cpp
//省略
namespace
{
 const int BGCOLOR[3] = {0, 0, 0}; // 背景色{ 255, 250, 205 }; // 背景色
 int crrTime;
 int prevTime;
 vector<Hero*> heroes; // プレイヤーのポインタを格納するベクター
}
//省略
 while (true)
 {
  oldMouseInput = mouseInput;
  mouseInput = GetMouseInput();

//省略
  //ここにやりたい処理を書く
  if ((oldMouseInput & MOUSE_INPUT_LEFT) == 0 && (mouseInput & MOUSE_INPUT_LEFT) != 0)
  {
   int x, y;
   GetMousePoint(&x, &y);
   Hero* p = &(Hero::GetInstance());
   p->SetPosition(x, y);
```

```
   heroes.push_back(p); // ベクターにポインタを追加（シングルトンなら同じアドレスしか入っていかないはず）
  }
//省略
  for (auto& hero : heroes)
  {
   hero->Update();
   hero->Draw();
  }
  ScreenFlip();

 }
  //省略
```

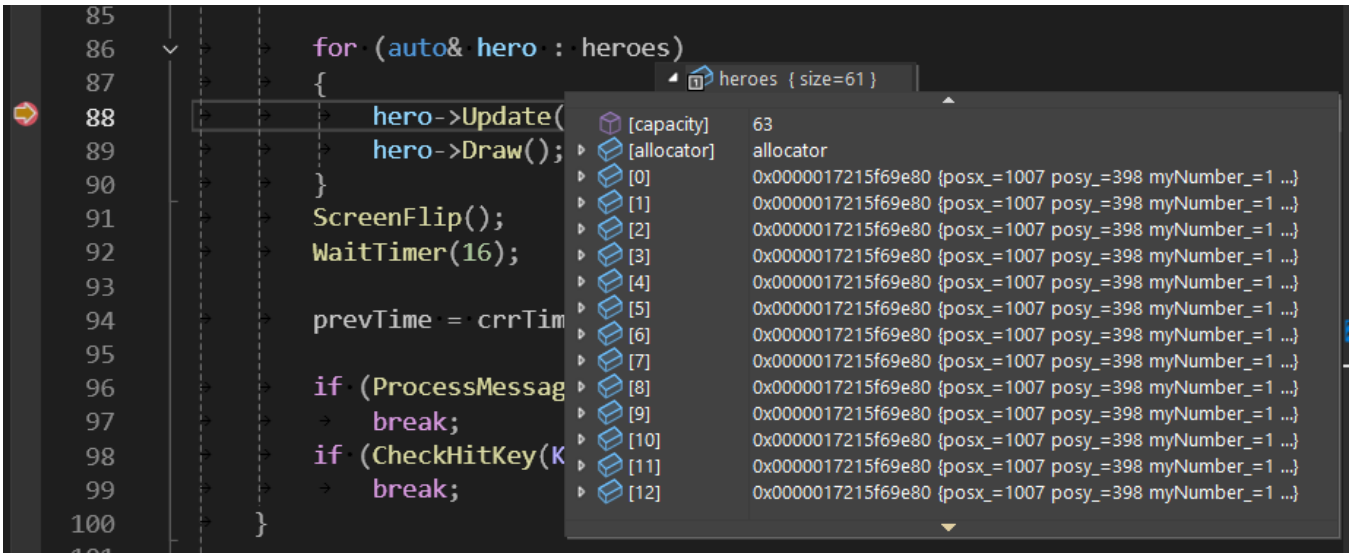
```
#pragma once
class Hero
{
private:
 int posx_, posy_; // プレイヤーの座標
 int myNumber_;// プレイヤーの番号　ずっと0のまま（シングルトンだから）
 int hImage_; // プレイヤーの画像
 static inline int playerCount_ = 0; // プレイヤーの数（シングルトンならふえないはず）
 static inline Hero* instance_ = nullptr; // 唯一のインスタンス
 Hero(); // コンストラクタをprivateにする
 ~Hero();
public:
 // シングルトンパターンを使用して、唯一のインスタンスを取得する
 static Hero& GetInstance();
 //インスタンスを削除する
 static void DeleteInstance();
 void SetPosition(int x, int y);
 void Update();
 void Draw();
 //コピーと代入を禁止する
 Hero(const Hero&) = delete;
 Hero& operator=(const Hero&) = delete;
};
Hero::Hero()
 : posx_(0), posy_(0), myNumber_(0), hImage_(-1)
{// プレイヤーの画像を読み込む
 hImage_ = LoadGraph("Assets/tiny_ship5.png");
 if (hImage_ == -1)
 {
  // 画像の読み込みに失敗した場合の処理
  //おサボりでかかない（みんなは書いてね）
 }
 playerCount_++;
 myNumber_ = playerCount_;  // インスタンス番号として記録
}

Hero::~Hero()
{
 // 画像の解放
 if (hImage_ != -1)
 {
  DeleteGraph(hImage_);
  hImage_ = -1;
 }
}

Hero& Hero::GetInstance()
{
 if (instance_ == nullptr)
 {
  instance_ = new Hero();
 }
 return *instance_;
}
```

```
void Hero::DeleteInstance()
{
}

void Hero::SetPosition(int x, int y)
{
 posx_ = x;
 posy_ = y;
}

void Hero::Update()
{
}

void Hero::Draw()
{
 DrawGraph(posx_, posy_, hImage_, TRUE);
 DrawFormatString(posx_+32+2, posy_-2, GetColor(255, 255, 255), "Player %d", myNumber_);
}
```

結果

https://www.youtube.com/embed/OCuqXxRyNhk?si=Pk0AE4Uf6pa8CR0e

一応、念のため、何度かクリックしたあとのリストの中身を表示してみると。。。



全部同じアドレスが入ってる。無駄な描画してるね笑
でも、クリックしたときにGetInstanceを呼んでも一度インスタンスが生成されてしまえば、同じインスタンスのアドレスを返してくれることがわかった（singletonとして動いているっぽい）
めでたしめでたし。

関数ポインタについて(by チョコミント)

関数ポインタについて

初めまして、チョコミントと申します。 これからゲーム制作の際に使えそうな便利なものなど、勉強感覚で記事にアップしていくので良ければ赤ちゃんを見る気持ちで見ていただけると嬉しいです...!!!

さて、初めての記事は何を書こうか悩んだのですが、汎用的に使えそうな関数ポインタについて触れていこうかなと思います。

関数ポインタとは

簡単に説明すると「関数」を参照している「ポインタ」のことです。 まあ変数のポインタと同じようなものですね！

```
void Test() { std::cout << "Hello"; }
int main()
{
 void (*func)() = Test;
 func();
}
```

上のコードではTest関数のポインタつまりアドレスをfuncに格納し funcを呼ぶことでHelloが表示されます。

ラムダ式を使って関数ポインタに登録してみる

みなさんは「ラムダ式」という言葉を聞いたことがありますか? 私も完璧には理解していないのですが、「疑似的に関数を作れる」みたいなことです。

```
int main()
{
 void (*func)() = []() { std::cout << "Hello"; };
 func();
}
```

上のコードではラムダ式 ［キャプチャ］（引数） {...} で関数を作成しfuncに登録しています。 funcを呼ぶことでHelloが表示されます。

ラムダ式の[]の部分にはキャプチャリストを記述できます。 つまりラムダ式のスコープ外にあるものを参照できるわけです。便利ですね...

```
int main()
{
 std::string str = "ぴえん";

 std::function<void()> func = [str]() { std::cout << str; };
 func();
}
```

上のコードではstr変数をキャプチャすることでstrを参照し ぴえんを表示しています。 キャプチャしたものはconstなので値は変更できません。

ここで「std::function<void()>」という新しいものがでてきました。 std::functionというものは関数ポインタ、関数オブジェクト、メンバ関数ポインタ、メンバ変数ポインタを保持できるクラスです。 えええ！！！！　ってことはstd::functionを使うことで、メンバ関数ポインタを保持できるってことは多種多様な機能が実現できるかも！！！

```
class Collider
{
 std::function<void()> hitfunc_; //コライダー同士がヒットした時に呼ばれる
 std::function<void()> exitfunc_; //コライダー同士が離れた時に呼ばれる

 bool isbeforeHit_;
```

```
public:

 Collider(std::function<void()> hit, std::function<void()> exit)
  :isbeforeHit_(false),hitfunc_(hit),exitfunc_(exit){}

 //当たり判定
 void CollisionDetection()
 {
  bool isHit = true;

  //当たり判定を行う
  //...
  //...

  if (isHit && !isbeforeHit_)
   hitfunc_();
  else if (!isHit && isbeforeHit_)
   exitfunc_();

  isbeforeHit_ = isHit;
 };
};


class GameObject
{
public:

 std::unique_ptr<Collider> collider_;

 GameObject()
 {
  collider_ = std::make_unique<Collider>([&]() { return HitFunc(); }, [&]() { return ExitFunc(); });
 }

 void HitFunc() { std::cout << "Hit"; };
 void ExitFunc() { std::cout << "Exit"; };
};

int main()
{
 std::unique_ptr<GameObject> obj = std::make_unique<GameObject>();
 obj->collider_->CollisionDetection();
}
```

上のコードではコライダークラスにあらかじめヒットした時、離れた時用の関数ポインタなどを保持するstd::function<void()>型の変数を用意しておきます。 ゲームオブジェクトクラスで好きな関数をラムダ式の中で呼び出しそれを登録します。 このコードを実行してみるとhitfunc_();が呼ばれHitが表示されます。

コライダーごとに当たった後、離れた後の挙動を変更したい時などとてもおすすめなんです！！ みなさんもぜひ活用してみたください！

最後に

初めて記事を書いたので、至らない点などが多かったと思いますが、最後まで見ていただきありがとうございます！！ 質問やわからなかったことなどがあればぜひ気軽にコメントしてください！

演算子のオーバーロードしてみようz

そのままいきなりソースコード

```
#include <iostream>

//オペレータ　演算子　+ - * / % sizeof () = * &
// int a = 1 + 2 二項演算子　１とか２のことをオペランド（右オペランド、左オペランド）
// int *b = &a; オペランド１個のやつ＝単項演算子

using std::endl;
using std::cout;

class Vec2
{
public:
    int x;
    int y;
    Vec2(int _x, int _y):x(_x),y(_y){}
    Vec2() {}
    ~Vec2() {};
    void PrintVal() { cout << "(x, y) = (" << x << ", " << y << ")" << endl; }
};
//
//Vec2 PlusVec2AndVec2(const Vec2 &_v1, const Vec2 &_v2);
//
//Vec2 PlusVec2AndVec2(const Vec2 &_v1, const Vec2 &_v2)
//{
//    return(Vec2(_v1.x + _v2.x, _v1.y + _v2.y));
//}


//プロトタイプ宣言
Vec2& operator+(const Vec2& _v1, const Vec2& _v2);

//定義
Vec2& operator+(const Vec2& _v1, const Vec2& _v2) {
    Vec2 ret;

    ret = Vec2(_v1.x + _v2.x, _v1.y + _v2.y);

    return (ret);
}


int main()
{
    Vec2 pos1 = { 10,10 };
    pos1.PrintVal();
    Vec2 pos2(20, 30);
    pos2.PrintVal();

    //int a = 10;
    //int b = 20;
    //int c = a + b;
    //cout << c << endl;
    //res.x = pos1.x + pos2.x;
    //res.y = pos1.y + pos2.y;

    Vec2 res = pos1 + pos2;
```

```
    //Vec2 res;
    //res = PlusVec2AndVec2(pos1, pos2);
    res.PrintVal();
}
```

共有したくなる技術や知識

Visual Studioでimport stdする

概要

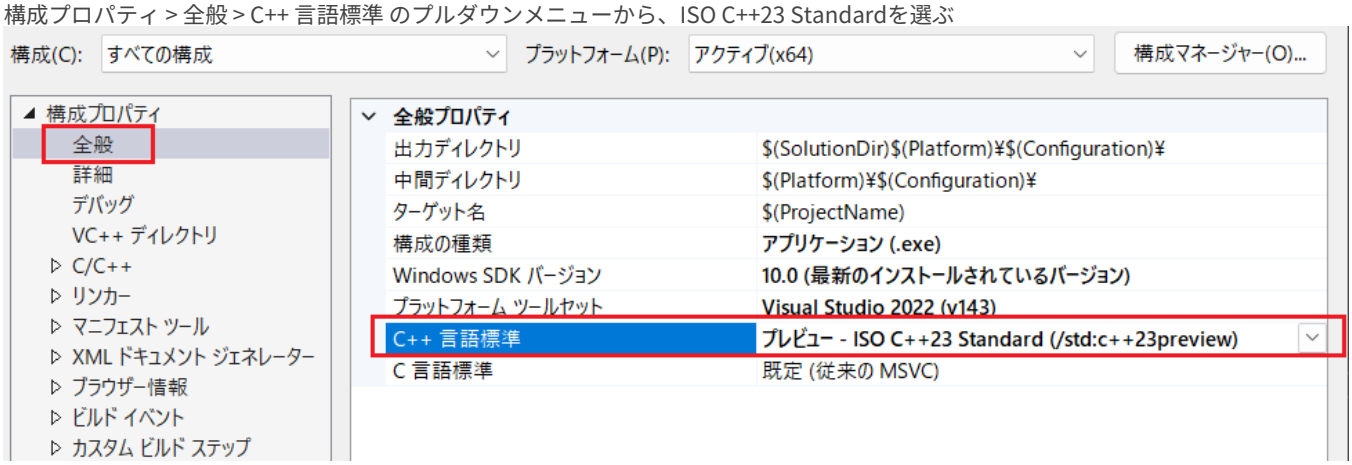
c++23から、`import std;` することで、c++及びcの標準ライブラリが使えるようになる。
これを使うと、コンパイル時間が短縮されたり、コーディング作業がほんのちょっと楽になったりする。
しかしながら、そのまま書いてもすぐ使えるわけではない。
本ページでは、これをどのようにして使えるようにするのかを解説していく。

環境

- Windows11
- Visual Studio 2022 version 17.13.7

やっていこう

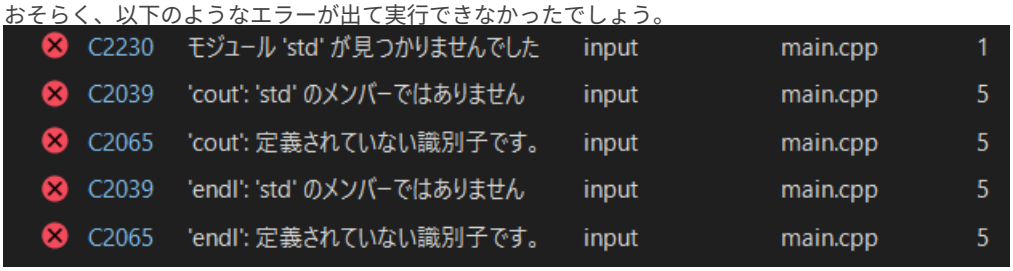
Visual Studioを開いて、適当なc++プロジェクトを作成します。
そうしたら、適当なソースファイルを作成した後、プロジェクトのプロパティを開き、
構成プロパティ > 全般 > C++ 言語標準 のプルダウンメニューから、ISO C++23 Standardを選ぶ



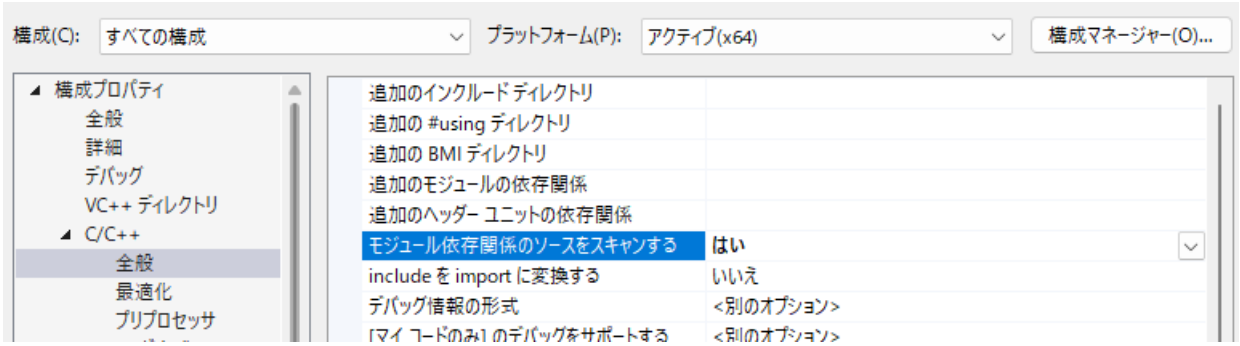
そして、以下のコードを入力し、実行します。

```
import std;

int main()
{
 std::cout << "Hello, Module!." << std::endl;
 return 0;
}
```

おそらく、以下のようなエラーが出て実行できなかったでしょう。



そしたら、再度プロジェクトのプロパティを開き、
更生プロパティ > C/C++ > 全般 > モジュール依存関係のソースをスキャンするのプルダウンメニューから、
はい を選び適用する。



そしてもう一度実行してみると…

ほらできた。

以上。

stl(vector)の３種類のループの仕方

std::vectorの要素を繰り返し処理（ループ）する代表的な3つの方法（範囲ベースfor文、イテレータ、インデックス）

1. 範囲ベースfor文 (Range-based for loop) - C++11以降

最も現代的で、簡潔かつ安全な方法です。要素を直接参照したい場合に最適です。

```
std::vector<int> vec = {10, 20, 30};
// 参照(&)を使って値を変更可能にする、またはコピーを避ける
for (const auto& value : vec) {
    std::cout << value << std::endl;
}
```

- **メリット:** コードが非常に読みやすい。イテレータの begin/end を意識しなくて良い。
- **用途:** 全要素に対する単純な処理。

2. イテレータ (Iterator)

伝統的な手法で、コンテナの特定の範囲や、要素を削除・挿入しながらのループに有用です。

```
std::vector<int> vec = {10, 20, 30};
for (auto it = vec.begin(); it != vec.end(); ++it) {
    std::cout << *it << std::endl; // ポインタのようにデリファレンスしてアクセス
}
```

- **メリット:** 高機能。ループ途中で要素の削除 (`erase`) が可能。
- **用途:** 要素を削除しながらの走査、STLアルゴリズムとの併用。

3. インデックスによるループ (Index-based)

配列の添え字（0, 1, 2...）を使ってアクセスする方法です。

```
std::vector<int> vec = {10, 20, 30};
for (std::size_t i = 0; i < vec.size(); ++i) {
    std::cout << vec[i] << std::endl; // vec.at(i) も使用可能
}
```

- **メリット:** インデックス（位置情報）自体が必要な場合に便利。
- **用途:** インデックス番号を利用した計算、特定の範囲や飛び飛びの要素のアクセス。

まとめと選び方

方法	特徴・メリット	おすすめの用途
範囲ベースfor	最も簡単、安全、読みやすい	基本的な全要素ループ
イテレータ	柔軟性（要素の削除が可能）	削除・挿入が発生するループ
インデックス	位置（i）を直接扱える	i番目を計算に使う場合

通常は **1. 範囲ベースfor文** を使い、要素削除が必要なら **2. イテレータ** を選ぶのが一般的です。

C++よもやま話

C++にまつわる小話や、都市伝説など、ホラー要素満載の読み物

C++よもやま話

C++とWindowsと非同期処理

Windows向けのゲーム開発において、**非同期処理や排他制御**（mutex、semaphore、critical sectionなど）を行う場合、以下のように目的や状況に応じて **C++標準ライブラリとWindows API** を使い分けるのが実践的です。

☒ 結論（簡潔版）

用途	推奨
移植性が欲しい （Windows以外も視野）	`std::mutex` など **C++標準ライブラリ**

パフォーマンス最優先
・
細かい制御が必要
	`CRITICAL_SECTION`
	や
	`WaitForSingleObject`
	など
	WinAPI

軽量なロック を大量に使う	`CRITICAL_SECTION` （WinAPIの軽量mutex）
スレッド間通信 （セマフォ）	WinAPIの `CreateSemaphore` などが高機能

簡単・安全に実装したい	`std::thread`
	、
	`std::mutex`
	、
	`std::condition_variable`

☒ 詳細比較

比較項目	C++ 標準（`std::mutex`、`std::thread`、`std::condition_variable`）	Windows API（`CRITICAL_SECTION`、`HANDLE`、`WaitFor...`）
可搬性	☒ Linux/Mac含むどこでも動く	☒ Windows限定
学習コスト	☒ 高レベルで簡単	☒ APIの種類が多く煩雑
速度（軽さ）	☒ `std::mutex` は重め（内部的にカーネルオブジェクト使用）	☒ `CRITICAL_SECTION` はユーザーモードで高速
機能の豊富さ	☒ ローレベルな制御は難しい	☒ タイムアウト・イベント・スレッド優先度など豊富
デバッグしやすさ	☒ STLのトレースが可能な環境も多い	☒ WinAPIデバッグはやや面倒

☒ ゲーム開発の実践的指針

◆ エンジン/ゲームループ系の並列化

→ パフォーマンス重視なので、`CRITICAL_SECTION` や `SRWLock`（Slim Reader/Writer Lock）など **WinAPI系** の軽量同期がおすすめ。

◆ アセット読み込み/非同期IO（ロード中にスレッド使用など）

→ `std::thread`, `std::async`, `std::mutex` など **標準ライブラリ** で十分。

◆ 条件通知（ロード完了したら通知、など）

→ `std::condition_variable` or `CreateEvent()`（WinAPI）どちらもOK
簡単に済ませたいなら **標準ライブラリ**、細かく制御したいなら **WinAPI**。

☒ よく使う組み合わせ例

軽量な同期（ゲームループ内で高速な排他が必要）

```
// Windows専用、軽量で高速
CRITICAL_SECTION cs;
InitializeCriticalSection(&cs);
EnterCriticalSection(&cs);
// 共有リソース操作
LeaveCriticalSection(&cs);
DeleteCriticalSection(&cs);
```

汎用的な排他制御（ポータブルかつ安全）

```
// 標準C++
std::mutex mtx;
std::lock_guard<std::mutex> lock(mtx);
// 共有リソース操作
```

☒ まとめ

ケース	推奨API
ちょっとした非同期	`std::thread` `std::mutex`
超高速同期処理	`CRITICAL_SECTION`
イベント通知	`std::condition_variable`（簡単） or `CreateEvent()`（高機能）
移植性を確保したい	C++標準ライブラリ
Windows専用かつパフォーマンス重視	Windows API（特に `CRITICAL_SECTION` `SRWLOCK` `WaitFor...` 系）

✔ **「C++の標準ライブラリの同期処理（`std::mutex` など）を使うと Windows 上で不具合が出るか？」**という点については、**通常は問題なく動作しますが、以下のようなケースでは注意が必要です。**

☒ 結論（要点）

問題	起こる可能性	備考
標準ライブラリのバグ	低い	MSVCのSTL実装は成熟している
パフォーマンス劣化	**あり得る**	`std::mutex` は `CRITICAL_SECTION` より重い
デッドロック	**設計ミスで起こる**	C++標準/WinAPIどちらでも同じ
UIスレッド（WndProc）との相性	**注意が必要**	`std::mutex` でブロック → フリーズの原因に
DLLや静的初期化子との相性	**ごく一部で注意**	`std::call_once` との絡みなど

☒ 詳細な注意点

1. ☒ パフォーマンスの問題（WinAPIのほうが軽い）

- `std::mutex` は実装上、**Windowsのミューテックス（`HANDLE`）を内部で使う**ケースがあります。

- この `HANDLE` は**カーネルオブジェクト**であり、ユーザーモードからカーネルモードへの切り替えが発生します。
- 一方、`CRITICAL_SECTION` は**ユーザーモードのみで完結**しており、遥かに軽量です。

☒ **ゲームループや頻繁なロック解除が発生する処理には向きません。**

---

2. ⚠ デッドロックは標準でも普通に起こる

```
std::mutex m1, m2;

void thread1() {
    std::lock_guard<std::mutex> lock1(m1);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock2(m2); //  デッドロックの可能性
}

void thread2() {
    std::lock_guard<std::mutex> lock2(m2);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock1(m1); //
}
```

これは標準・WinAPI問わず「ロック順序が違う」とアウト。

---

3. ⚠ UIスレッドでのmutex使用は注意

`WndProc()` や `WM_PAINT`、`WM_TIMER` など**WindowsのUIスレッド**で `std::mutex::lock()` を呼ぶと、

- **デッドロック**
- **メッセージポンプ停止**
- **アプリがフリーズ**

といった副作用が起きやすいです。

☒ UIスレッド内では `try_lock()` や `std::atomic_flag` など**非ブロッキングな同期方法**を使うべき。

---

4. ⚠ DLLの初期化タイミングでの罠（static変数・`std::call_once`）

- `std::mutex` を使っている `static` ローカル変数の初期化や `std::call_once()` を DLL 内で使っていると、
  DLLロード順やタイミング依存のクラッシュやデッドロック が起こることがあります。

特に `DllMain` や `static グローバル変数` の初期化で重い処理をすると危険。

---

5. ☒ Visual Studio/MSVCの不具合は基本的にない

- MSVCの `std::mutex`, `std::condition_variable` は **Windowsのネイティブ機能を使って安定実装されています**。
- なので、**Windows公式コンパイラ（MSVC）での「機能的な不具合」は稀**です。

☒ 安全に使うためのヒント

状況	推奨対応
高速処理が必要	`CRITICAL_SECTION` か `std::atomic_flag` を検討
UIスレッド内	`try_lock()` か非同期通知（メッセージポスト）を使う
複数ロックあり	`std::scoped_lock` で複数同時ロック順序を統一
DLL初期化時	`std::call_once` の使用タイミングに注意

☒ まとめ

観点	C++標準ライブラリの同期処理
Windowsでの動作	基本的に安定して動く
パフォーマンス	`CRITICAL_SECTION` より重い
UIとの相性	ブロックするとフリーズに注意
典型的不具合	デッドロック、DLL初期化でのタイミング依存など
おすすめ用途	軽い並列タスク、非UIスレッドの排他制御

あいつは自分から見て右にいるの？左にいるの？

☒目的

自分の向いている方向から見て、「ターゲットが右にいるのか、左にいるのか」を判断し、その方向に回転させたい。

☒前提知識

1. `atan2(y, x)` の意味

ベクトル `(x, y)` の**角度（ラジアン）を、原点から見たときのx軸との角度**として返します。範囲は `[-π, π]`。

2. 基準ベクトル

自分が向いている方向をベクトル `forward` とし、右方向を基準にする場合は、通常 `right = {1, 0}` などとします（右向き単位ベクトル）。

☒方法1：`atan2`による角度の差で判断

```
// 自分の向きとターゲット方向
Vec2 forward = 自分の向きベクトル;     // 例: {1, 0} → 右向き
Vec2 toTarget = targetPos - selfPos;   // 自分からターゲットへのベクトル

// それぞれの角度を求める
float angleForward = atan2(forward.y, forward.x);
float angleToTarget = atan2(toTarget.y, toTarget.x);

// 差を求める（-π〜π の範囲に自動でなる）
float delta = angleToTarget - angleForward;

// 回転方向を判断
if (delta > 0) {
    // 左にターゲットがある（左回りに回転すべき）
} else if (delta < 0) {
    // 右にターゲットがある（右回りに回転すべき）
}
```

☒この方法のメリット：

角度を厳密に扱える。滑らかに回転アニメーションしたいときに使いやすい。

☒方法2：右向きベクトルとの **外積 or 内積** を使って右左判定

2Dにおける**外積（cross product）**で符号を利用する方法：

```
float cross = forward.x * toTarget.y - forward.y * toTarget.x;
```

- `cross > 0` → 左側
- `cross < 0` → 右側
- `cross == 0` → 同一直線上

または、**右向きベクトルと角度比較**

基準として右向きのベクトル `right = {1, 0}` を使って、ターゲットとの角度を見る：

```
Vec2 right = {1, 0};
float angleToTarget = atan2(toTarget.y, toTarget.x);
float angleRight = atan2(right.y, right.x); // = 0

// 結果的に angleToTarget だけ見れば良い
if (angleToTarget > 0) {
    // ターゲットは上（左回り）
} else {
    // ターゲットは下（右回り）
}
```

☒補足：`atan2` と外積の使い分け

方法	特徴	回転方向判定	回転量取得
`atan2` 差分	正確な角度取得。平滑回転に◎	◎	◎（角度そのもの）
外積（2D）	軽量。方向判定だけなら高速	◎（符号）	×（角度は出せない）

優先度付きキューの話（メロリンキュー）

queue と priority_queue の違い

― FIFO と「優先度」の正体 ―

1. このページの目的

C++ STL には複数の「キュー系コンテナ」が存在します。
本ページでは、次のコードを題材に、

- `std::queue`（通常のキュー）
- `std::priority_queue`（優先度付きキュー）

の **動作原理と使いどころの違い** を整理します。

対象は **C++ が書けるゲームエンジニア科 2 年生**です。
STL の仕様寄りの説明を行います。

---

2. 使用するサンプルプログラム

2.1 プログラム全体

```
#include <iostream>
#include <queue>

namespace {
    std::priority_queue<int, std::vector<int>, std::greater<int>> myQueue;
    std::queue<int> ituQueue;
}

int main()
{
    ituQueue.push(10);
    ituQueue.push(3);
    ituQueue.push(5);
    ituQueue.push(8);
    ituQueue.push(2);

    for(int i = 0; i < 5; ++i) {
        int val = ituQueue.front();
        ituQueue.pop();
        std::cout << "Pop: " << val << std::endl;
    }

    std::cout << "-----------------------------" << std::endl;

    myQueue.push(10);
    myQueue.push(3);
    myQueue.push(5);
    myQueue.push(8);
    myQueue.push(2);

    for (int i = 0; i < 5; ++i) {
```

```
        int val = myQueue.top();
        myQueue.pop();
        std::cout << "Pop: " << val << std::endl;
    }

    return 0;
}
```

---

3. std::queue の挙動（FIFO）

3.1 std::queue とは

`std::queue` は **FIFO（First In, First Out）** のキューです。

- 先に入れたものが
- 先に出てくる

という **順番固定** のデータ構造です。

3.2 実行結果（queue）

```
Pop: 10
Pop: 3
Pop: 5
Pop: 8
Pop: 2
```

3.3 なぜこの順番になるか

```
ituQueue.push(10);
ituQueue.push(3);
ituQueue.push(5);
ituQueue.push(8);
ituQueue.push(2);
```

投入順そのままに、

```
10 → 3 → 5 → 8 → 2
```

が `front()` → `pop()` で取り出されます。

3.4 queue の用途

- イベントキュー
- 入力処理
- 通信パケット処理
- BFS（幅優先探索）

「**順番を守ること**」が最優先の処理向きです。

---

4. std::priority_queue の挙動（優先度付き）

4.1 priority_queue とは

`std::priority_queue` は、

> **「順番」ではなく「優先度」で取り出されるキュー**

です。

投入順は一切保証されません。

4.2 宣言の意味

```
std::priority_queue<int, std::vector<int>, std::greater<int>> myQueue;
```

各要素の意味は以下です。

要素	意味
`int`	格納する型
`std::vector<int>`	内部コンテナ（ヒープ用）
`std::greater<int>`	比較規約

4.3 比較規約の重要ルール

`priority_queue` では次の規約が使われます。

> **Compare(a, b) == true のとき、a は b より優先度が低い**

4.4 std::greater<int> の意味

```
std::greater<int>()(a, b)  // a > b
```

- `a > b == true`
- → **a は後回し**
- → **小さい値ほど優先**

つまり、

☒ **最小値が常に top() に来る**

4.5 実行結果（priority_queue）

```
----------------------------
Pop: 2
Pop: 3
Pop: 5
Pop: 8
Pop: 10
```

4.6 なぜこの順番になるか

投入順：

```
10, 3, 5, 8, 2
```

内部では **ヒープ構造** が作られ、

```
常に最小値が top()
```

になります。

---

5. queue と priority_queue の本質的違い

項目	std::queue	std::priority_queue
取り出し基準	入れた順	優先度
順序保証	あり	なし
内部構造	deque	heap（vector）
top / front	front()	top()

6. ゲーム開発での典型用途

6.1 std::queue

- 入力イベント処理
- メッセージキュー
- BFS（迷路探索など）

6.2 std::priority_queue

- A* / ダイクストラ法
- タスクスケジューラ
- 距離・コスト最小探索
- AI 思考順制御

7. 自作構造体での使用例（探索系）

```
struct Node {
    int cost;
};

// cost が小さいほど優先
bool operator>(const Node& a, const Node& b)
{
    return a.cost > b.cost;
}

std::priority_queue<Node, std::vector<Node>, std::greater<Node>> open;
```

```
open.push({10});
open.push({3});
open.push({7});

open.top().cost; // 3
```

8. まとめ

- `std::queue`
  - **順番重視**
  - FIFO
- `std::priority_queue`
  - **優先度重視**
  - 最小 or 最大ヒープ

探索アルゴリズムでは
「次に処理すべきもの」を選ぶために priority_queue が使われる
という点を押さえておくと、実装の意味が見えてきます。