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。

- 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

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。

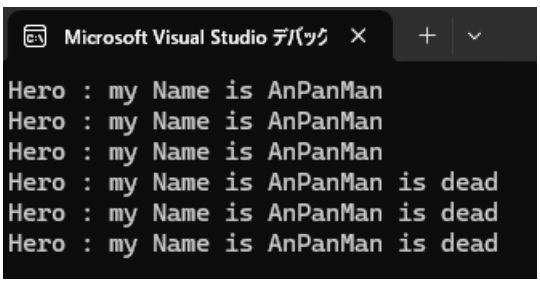
作ってみるよ

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

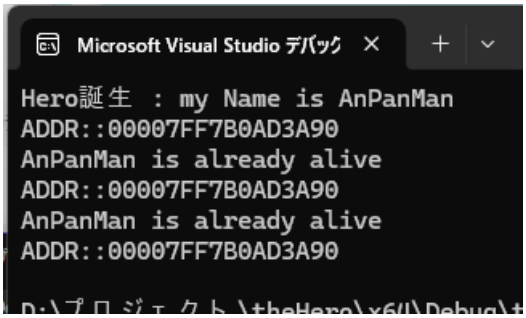
プログラムの全体像と、実行のためのmain関数を以下に示す。

```
#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！

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
```

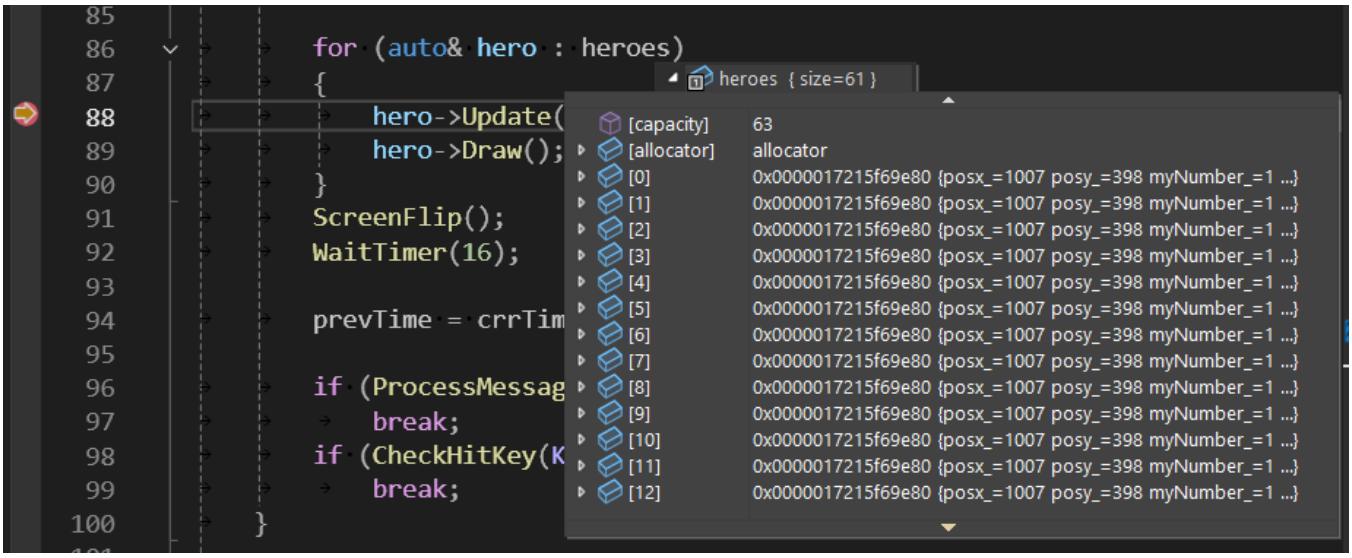
```
#pragma once
class Hero
{
private:
 int posx_, posy_; // プレイヤーの座標
 int myNumber_;// プレイヤーの番号　ずっと0のまま（シングルトンだから）
 int hImage_; // プレイヤーの画像
 static inline int playerCount_ = 0; // プレイヤーの数（シングルトンならふえないはず）
 static inline Hero* instance_ = nullptr; // 唯一のインスタンス
 Hero(); // コンストラクタをprivateにする
 ~Hero();
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();
}
```

Visual Studioでimport stdする

概要

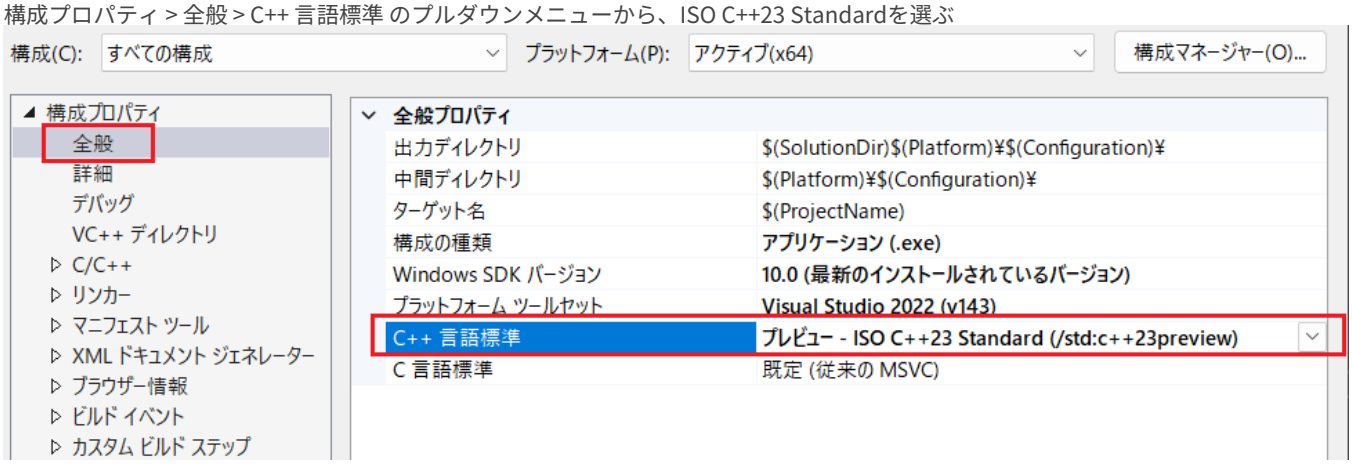
c++23から、`import std;` することで、c++及びcの標準ライブラリが使えるようになる。
これを使うと、コンパイル時間が短縮されたり、コーディング作業がほんのちょっと楽になったりする。
しかしながら、そのまま書いてもすぐ使えるわけではない。
本ページでは、これをどのようにして使えるようにするのかを解説していく。

環境

- Windows11
- Visual Studio 2022 version 17.13.7

やっていこう

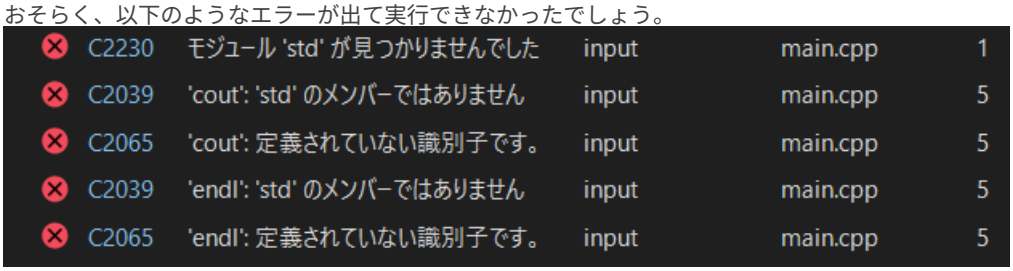
Visual Studioを開いて、適当なc++プロジェクトを作成します。
そうしたら、適当なソースファイルを作成した後、プロジェクトのプロパティを開き、
構成プロパティ > 全般 > C++ 言語標準 のプルダウンメニューから、ISO C++23 Standardを選ぶ



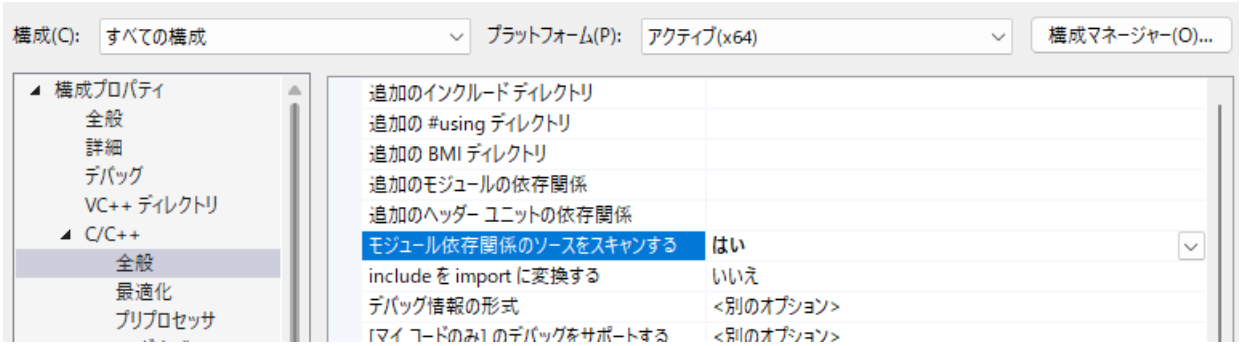
そして、以下のコードを入力し、実行します。

```
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。