# **Chapter 1 デバッガの使い方。**

このチャプターではC++の話は行いません。C++の学習を始める前にプログラムを学習、作成するうえで非常に強力な武器になるデバッガの使い方を説明します。VisualStudioを使用して、代表的なデバッガの機能を紹介します。ツールの使い方の説明だけですので難しいところはありませんので、頑張って習得してください。

## **1.1 デバッガ**

プログラムを作成していく過程で、切っても切り離せない問題にバグ(bug)の存在があります。バグとはプログラム、データの間違いのことで、世の中のどんなソフトウェアでも発生する厄介なものです。ソフトウェアの開発では、コードを書く時間よりバグを潰す時間の方が長くなるということもザラにあります。そのバグを修正する作業をデバッグ(Debug)といいます。そしてデバッグ作業を強力に手助けしてくれるツールのことをデバッガ(Debugger)と言います。

## **1.2 デバッガありで実行(F5)**

今まで皆さんはctrl+F5でプログラムを実行していたのではないでしょうか？ctrl+F5での実行は「デバッガなしで実行」というコマンドです。実はこのコマンドはプロの開発者はほとんど使いません。デバッガという強力なツールを使わない理由がないからです。

デバッガありで実行するにはF5キーを押すだけです。Ctrlキーを押さなくていいので操作もスムーズです。今後の開発ではF5キーでの実行を基本としましょう。

## **1.3 代表的なデバッガの機能**

ここでは、このチャプターの本題となるデバッガの代表的な機能を紹介します。どれも難しいものではないので頑張ってマスターしてください。また、今回の機能の紹介は全てショートカットキーのみを教えます。GUIを使った操作より効率的に開発でき、ストレスも軽減されるため、こちらをマスターしましょう。

**1.3.1 ブレイクポイント(F9)**

プログラムの実行を停止して調査したいときに設定するものです。処理を止めたい箇所にカーソルを合わせてF9キーを押して見てください。ブレイクポイントの設置ができたはずです。このブレイクポイントを削除したい場合もF9キーを押せば削除できます。

**1.3.2 ステップオーバー(F10)、ステップイン(F11)、ステップアウト(shift+F11)**

ここではブレイクポイントで停止させたプログラムを１行ずつトレースするための３つの機能を紹介します。

**1.3.3 ステップオーバー(F10)**

ブレイクポイントでプログラムを停止させたら、そこからの処理を一行ずつトレースしたくなると思います。その時に使用されるデバッガの機能がステップオーバーです。ブレイクポイントで停止させたらF10キーを押して見てください。プログラムが一行だけ進むはずです。

**1.3.4 ステップイン(F11)**

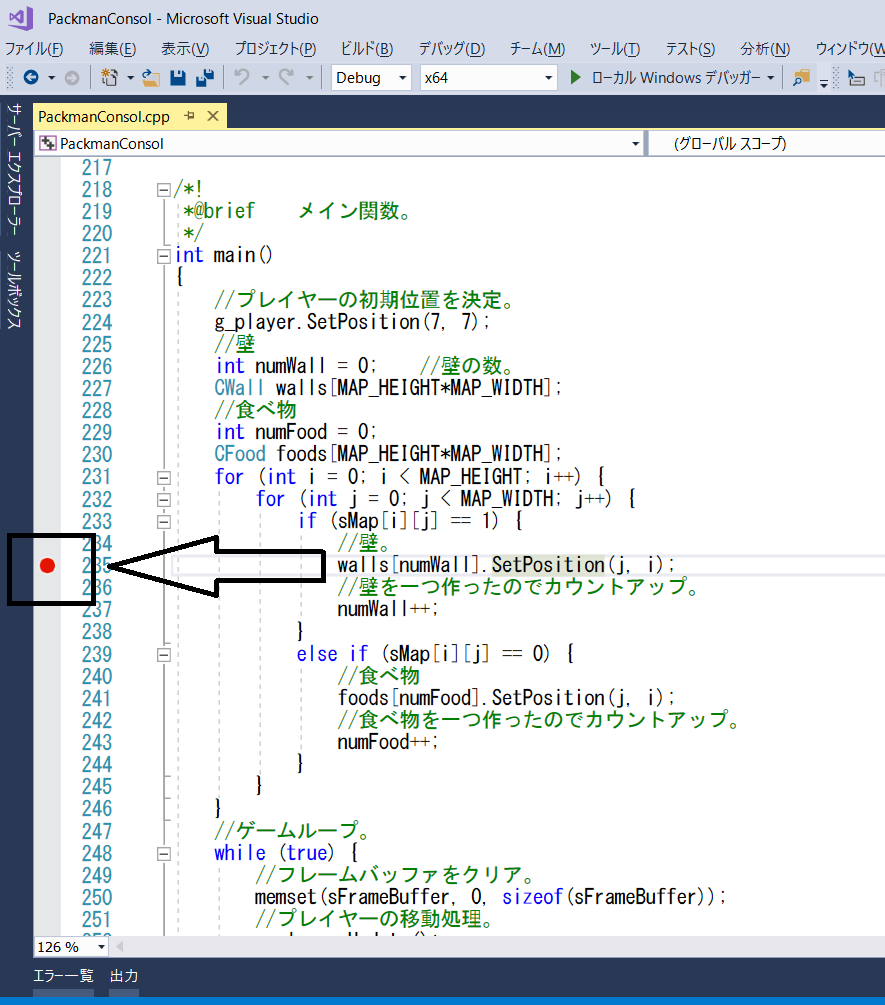
先ほどのステップオーバーとほぼ挙動は同じです。違う点はこのコマンドを使用すると関数の中に入っていけます。関数の中に入るためステップ”イン”という名前になっています。

**1.3.5 ステップアウト(Shift+F11)**

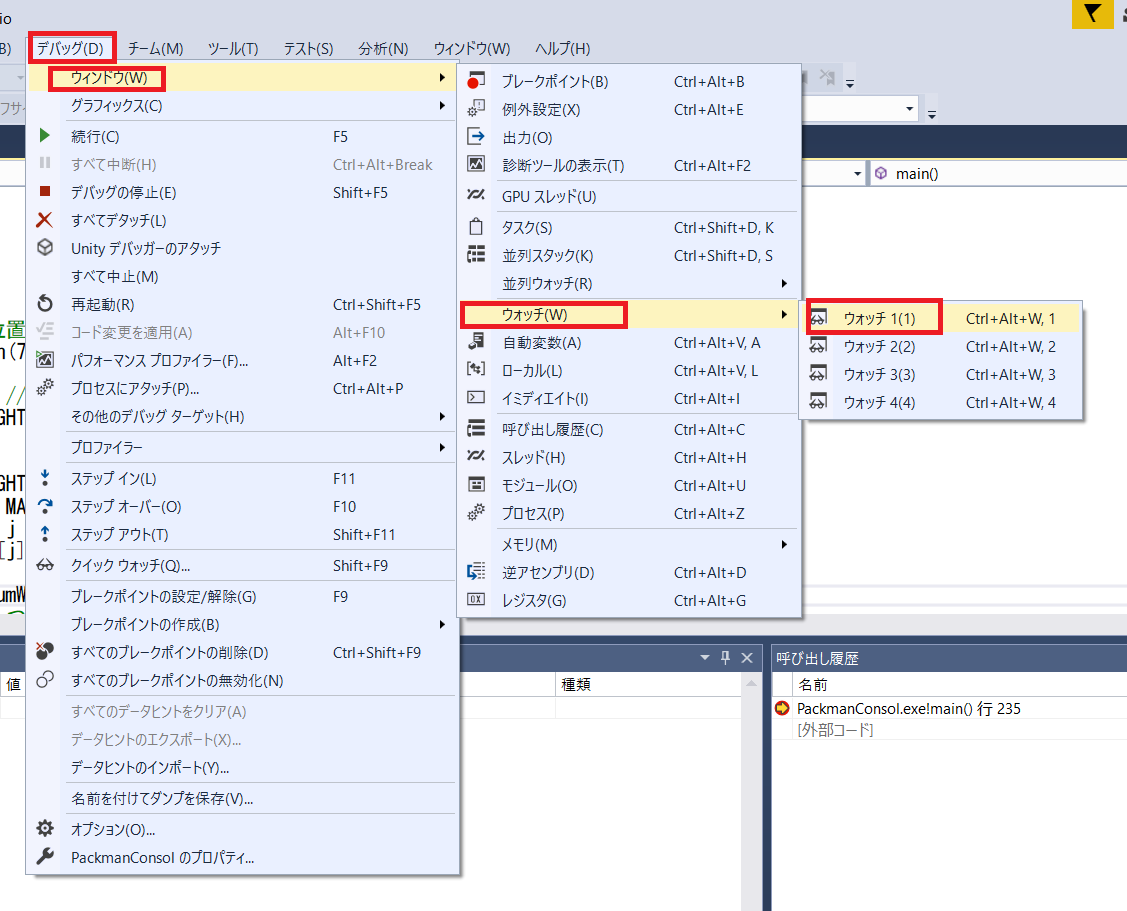
こちらはステップインとは逆の挙動になり、関数から抜ける機能になります。

**1.3.6 ウォッチ**

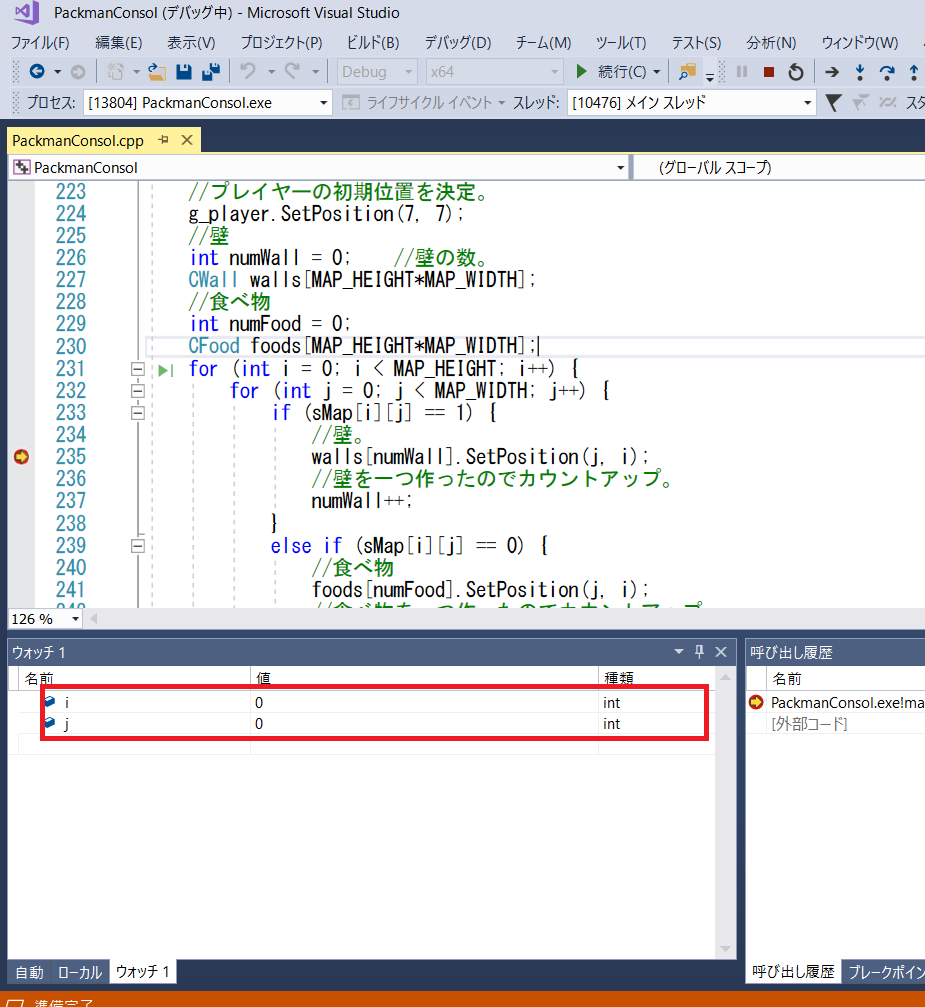
　ブレイクポイントでプログラムを停止させると、その時の変数の中身を調べたくなることがあるかと思います。その時に使える機能がウォッチという機能です。ではウォッチを使ってみましょう。まず、Lesson1/PackmanConsol/PackmanConsol.slnを起動して、下記の図のように235行目にブレイクポイントを設置(F9)してください



ブレイクポイントが設置できたら、F5を押して、デバッガありで実行を行ってください。ブレイクポイントでプログラムを停止することができたら、ウォッチでループ変数のiとjの値を確認してみましょう。ウォッチのウィンドウが表示されていない人はメニューバーのデバッグ->ウィンドウ->ウォッチ->ウォッチ１を選択してウォッチウィンドウを表示させてください(下記の図を参照)。

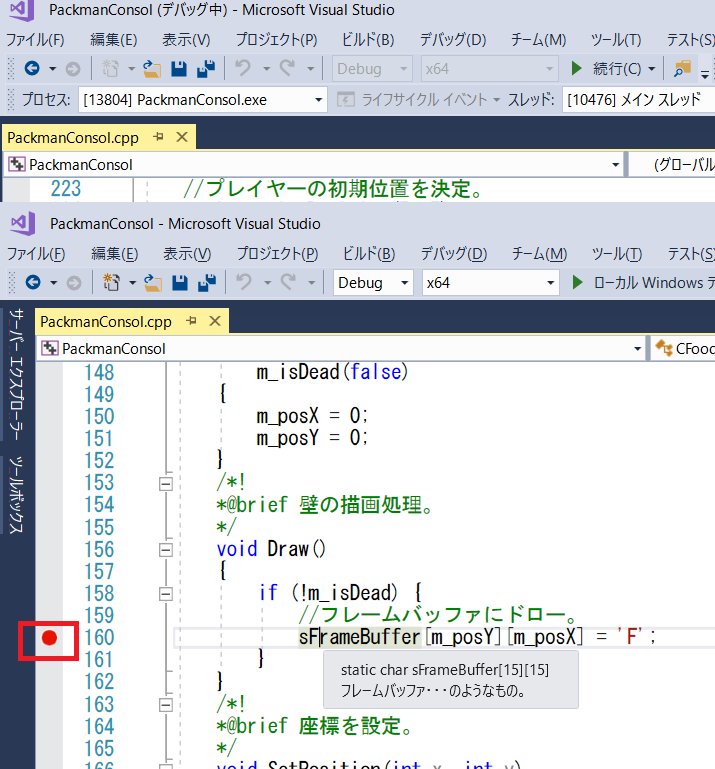


ウォッチが表示されたら、中身を見たい変数名をウォッチウィンドウに追加してみましょう。



**1.3.7 呼び出し履歴**

関数の呼び出し履歴が確認できる機能です。ブレイクポイントで停止させた時に、その関数がどこから呼ばれているかを遡ることができます。では、下記の図のようにPackmanConsole.cppの160行目にブレイクポイントを設定してF5で実行してください。



ブレイクポイントでプログラムを停止出来たら、CFood::Draw関数がどこから呼び出されているのか、呼び出し履歴で確認してみましょう。呼び出し履歴が表示されていない人は次の操作で表示させてください。メニューバーからデバッグ->ウィンドウ->呼び出し履歴



## **1.4 まとめ**

　ここまで紹介したものが非常に使用頻度の高いデバッガの機能になります。他にもたくさん機能はあるのですが、まずは基本となるこれらの機能をマスターして次のステップに進みましょう。

## **1.５ 章末テスト**

下記のURLのテストを行いなさい。

<https://docs.google.com/forms/d/e/1FAIpQLScQw2x7TgbIvKf77dL8A6RMKRN4KhSFekb4F9BYCogYVGgu2g/viewform?usp=sf_link>

# **Chapter 2 クラス**

C++とCの大きな違いはクラスという概念をサポートしているかどうかになります。このチャプターではオブジェクト指向プログラムの肝となるクラスについて見ていきます。

## **2.1 クラスの概念**

　クラスというのは、プログラムで実装する必要のある機能をモデル化してプログラムの可読性、保守性、再利用性を向上させるために用いられる概念です。昔はクラスとは現実世界のモノをモデル化するものだという説明が入門書などに記述されていたのですが、これは間違った説明で、これがクラスを理解できない原因になっているとまで言われていたものです。このような話を聞いたことがある人は頭から叩き出してください。

## **2.1.1 可読性**

可読性とはコードの読みやすさということです。ソフトウェアの開発というのは複数人で行われるもので、一人で開発を行うということは稀です。そのため、他人のソースコードを読むということが必ず発生します。そのためコードの読みやすさというのはソフトウェア開発を潤滑に進めるために重要なファクターになります。また、ソースコートというものは他人が読むだけではありません。間違いなく自分の書いたコードを一番多く読むのは自分自身です。昨日の自分は他人という言葉がソフトウェア開発の世界ではよく使われます。可読性の高いプログラムを書くということは、他人のためというよりも自分のためという側面の方が強くなります。

　ではなぜクラスを使うと可読性が上がるのでしょうか？例えば、あなたがゲーム開発でプレイヤーの処理を実装していると考えてみて下さい。クラスを扱うことに少し慣れている人ならclass Player{}などのようにプレイヤーのクラスを作ることをすんなりと考えられるはずです。なぜなら、Playerをクラス化しておけば、プレイヤーの処理は基本的にはPlayerクラスのソースを読めばいいし、そこに書けばいいことが分かるからです。大規模な開発ならソースコードがウン十万行あることは珍しくありません。そこでクラスの概念がない言語(C言語とか)で開発を行うことを考えてみてください。恐らくあなたはプレイヤーの処理がどこに書かれているのかを把握することすら困難になるでしょう。

## **2.1.2 保守性**

保守性というのはプログラムのメンテナス、拡張のしやすさという意味です。例えばアクションゲームを作っているケースを考えてみてください。そのゲームでは当初はAボタンが押されるとプレイヤーがジャンプするという仕様でした。しかし、開発が半年ほど経過して中盤に差し掛かった頃にクライアントから、ジャンプ中にAボタンが押されたら２段ジャンプができるようにして欲しいという要望が来ました。そのためプレイヤーのプログラムを変更する必要に迫られます。プレイヤーのジャンプのプログラムを作成したのは数ヶ月前になり、コードの場所の記憶も薄れてきています。しかし、このプロジェクトがC++の

ようなクラスが使えるプログラム言語で作成されているのであれば、すぐにPlayerクラスが見つかることでしょう。そしてすぐにプログラマはPlayerクラスのJump関数を見つけ出して、その関数のプログラムを変更すればいいことを思い出すはずです。もしこれがC言語のようなクラスが使えないプログラム言語で作成されていたのであれば、プログラマはウン十万という膨大なソースの中からプレイヤーの処理を探すことになるでしょう。

もちろんC言語を使っていても、ソースファイル名にPlayer.cppなどのような分かりやすい名前をつけていれば該当する処理を探し出すことは容易です。しかし、クラスという機能をモデル化する概念がない言語では往々にして、プレイヤーの処理はいたるところに記述されていきます。そしていつかそれを把握するのが困難になります。とくに記憶が薄れてきたころに。

## **2.1.3 再利用性**

オブジェクト指向言語で言われる再利用とはプログラムを再利用と、設計の再度利用をさします。オブジェクト指向がもてはやされた時に、クラスや継承を使用するとプログラムの再利用性が向上して開発効率が高くなると言われていました。しかし現実問題、作成したクラスの再利用は一部のライブラリなどを作成するプログラマが考えることで、アプリケーション層のプログラムを書いている場合は再利用性をそこまで考える必要はありません。再利用されるかどうかも分からないプログラムで、再利用されたときのことを考えても無駄になります。オブジェクト指向のキモは可読性と保守性の向上です。ただし、設計の再利用は重要です。これがデザインパターンと呼ばれるものです。デザインパターンとは先人の考えた優れた設計をカタログ化して再利用しようという考えです。

## **2.2 C++でのクラスの作成の仕方**

　では3Dアクションゲームでプレイヤークラスを作成する場合を考えてクラスを作成してみましょう。まず、クラスの名前はPlayerなどといった名前にすることが思いつくかと思います。

　class Player{

};

次はメンバ変数を考えてみましょう。3Dアクションゲームなので、当然プレイヤーは3D空間場で位置を表すための変数を保持しているはずです。そのため、プレイヤーに位置を表す変数を保持させてみましょう。

class Player{

private:

float m\_positionX; //X座標

float m\_positionY; //Y座標

float m\_positionZ; //Z座標

};

また、このゲームは体力という概念があり、敵から攻撃を受けると体力が減るという仕様があります。そのためプレイヤーは体力というメンバ変数を保持しているはずです。

class Player{

private:

float m\_positionX; //X座標

float m\_positionY; //Y座標

float m\_positionZ; //Z座標

int m\_hitPoint; //体力

};

また、画面にプレイヤーを描画する必要があるため、Drawというメンバ関数も必要なはずです。

そして、ユーザーのキー入力によりプレイヤーは移動するため、Moveというメンバ関数も必要になるでしょう。

class Player{

float m\_positionX; //X座標

float m\_positionY; //Y座標

float m\_positionZ; //Z座標

int m\_hitPoint; //体力

public:

void Move()

{

//移動処理。

}

void Draw()

{

//描画する処理を記述する。

}

};

ではこのPlayerクラスを使用して、簡単なゲームプログラムを書いてみましょう。

int main()

{

Enemy enemy; //敵

　 Player player; //プレイヤー

　//ゲームループ

while(true){

　　enemy.Move();　//敵の移動処理。

player.Move(); //プレイヤの移動処理

　　enemy.Draw(); //敵の描画処理。

　　player.Draw();　 //プレイヤの描画処理。

　　WaitVSync(); //垂直同期待ち。おまじない。

}

}

このクラスの作成の仕方の流れは私の思考をトレースした一例でしかありません。このような流れでクラスを作成する必要があるわけではないので注意してください。

## **2.3　実習**

下記の実習を行いなさい。

実習 1

Lesson2はクラスを使用せずにプレイヤーの処理を記述しています。このプログラムのリファクタリングを行い、プレイヤークラスを作成して、同様の処理を行えるようにしなさい。

　プレイヤークラスは下記の仕様を最低限、満たすようにしなさい。

　・プレイヤーのX座標を表す、int型のメンバ変数を保持している。

　・プレイヤーのY座標を表す、int型のメンバ変数を保持している。

　・プレイヤーを描画するためのメンバ関数、Draw関数を実装している。

　・プレイヤーを移動させるためのメンバ関数、Move関数を実装している。

実習２

　Lesson2を改造して、下記の動画のような挙動をするエネミーを実装しなさい。この実習でのプログラムは下記の仕様を最低限、満たすものとする。

・エネミークラスを作成している。

・エネミークラスはX座標を表す、int型のメンバ変数を保持している。

・エネミークラスはY座標を表す、int型のメンバ変数を保持している。

・エネミーを描画するためのメンバ関数、Draw関数を実装している。

・エネミーを移動させるためのメンバ関数、Move関数を実装している。

・エネミーのインスタンスを10個用意しているものとする。

・エネミーの初期位置に特に決まりはない。動画の通りでなくてもよい。ただし、エネミー同

士かぶってはいけないものとする。



## **2.4 メンバ変数**

クラスは構造体と同じように変数を記述することができます。そして下記のように構造体と同じように使うことができます。

|  |
| --- |
| //Playerクラスを定義。  class Player{  public: //アクセス指定子。後ほど説明する。  float posX; //X座標。  float posY; //Y座標。  float posZ; //Z座標。  };  int main()  {  Player pl; //Player型のplという変数を定義。  　 pl.posX = 10.0f;  pl.posY = 20.0f;  pl.posZ = 30.0f;  } |

このように構造体と全く同じように扱うことができます。

## **2.5 メンバ関数**

クラスには変数だけではなく、関数も定義することができます。クラス定義の中で関数宣言を記述すると、その関数はメンバ関数になります。では先ほどのPlayerクラスにX方向に移動するMoveX関数を追加してみます。

|  |
| --- |
| //Playerクラスを定義。  class Player{  public:  float posX; //X座標。  float posY; //Y座標。  float posZ; //Z座標。  //X方向に移動するMove関数の宣言。  void MoveX();  };  //PlayerクラスのMove関数の定義。  void Player::MoveX()  {  //X方向に1移動。  posX += 1.0f;  }  int main()  {  Player pl; //Player型のplという変数を定義。  　pl.posX = 10.0f;  pl.posY = 20.0f;  pl.posZ = 30.0f;  pl.MoveX(); //X方向に移動させる。//これでpl.posXは11.0fになる。  } |

## **2.6 アクセス指定子**

クラスにはアクセス指定子を記述することでメンバ変数や、メンバ関数をどこからアクセス可能なのかを指定できる機能があります。2.3で出てきたpublic:がアクセス指定子の一つです。C++にはpublic、private、protectedの三つのアクセス指定子があります。今回はpublicとprivateを説明します。protectedについてここで説明をすると混乱を招きますので説明しません。まずpublicとprivateをマスターしましょう。

## **2.6.1 public(パブリック)**

publicを指定されたメンバ変数、メンバ関数はどこからでもアクセス可能になります。構造体と同じだと考えて下さい。では具体的にプログラムを見ていきましょう。

|  |
| --- |
| class Player{  public: //パブリックを指定する。  float posX; //X座標。このメンバ変数はパブリックになる。  float posY; //Y座標。このメンバ変数はパブリックになる。  float posZ; //Z座標。このメンバ変数はパブリックになる。  void MoveX(); //X方向に移動するMove関数の宣言。このメンバ関数はパブリックになる。  };  //PlayerクラスのMove関数の定義。  void Player::MoveX()  {  //X方向に1移動。  posX += 1.0f;  }  int main()  {  Player pl; //Player型のplという変数を定義。  　pl.posX = 10.0f; //パブリックなのでアクセスできる。  pl.posY = 20.0f; //パブリックなのでアクセスできる。  pl.posZ = 30.0f; //パブリックなのでアクセスできる。  pl.MoveX(); //X方向に移動させる。パプリックなのでアクセスできる。  } |

実はこのコードは2.6で見たサンプルコードと全く同じです。2.4のサンプルコードもpublicを指定しているため、変更する必要がありませんでした。

## **2.6.2 private(プライベート)**

privateはクラスのカプセル化(後述します)といわれるものの肝となるアクセス指定子になります。非常に重要なものになりますので、なんとかマスターしてください。では説明をしていきます。

privateが指定されたメンバ変数、メンバ関数はクラスの外部からはアクセスができなくなります。では具体的にprivateを指定するとどうなるか見ていきましょう。

|  |
| --- |
| class Player{  public: //パブリックを指定する。  float posX; 　　　 //X座標。このメンバ変数はパブリックになる。  float posY; 　　　//Y座標。このメンバ変数はパブリックになる。  float posZ; 　　　//Z座標。このメンバ変数はパブリックになる。  void MoveX (); 　　//プレイヤーを移動させる関数。このメンバ関数はパブリックになる。  private: //プライベートを指定する。  float moveSpeedX; //X方向への移動速度。このメンバ変数はプライベートになる。  };  //PlayerクラスのMove関数の定義。  void Player::MoveX()  {  //X方向に1移動。  **//MoveXはメンバ関数なので、moveSpeedXにアクセスできる。OK!!**  posX += moveSpeedX;  }  int main()  {  Player pl; //Player型のplという変数を定義。  　pl.posX = 10.0f; //パブリックなのでアクセスできる。  pl.posY = 20.0f; //パブリックなのでアクセスできる。  pl.posZ = 30.0f; //パブリックなのでアクセスできる。  pl.posX += pl. moveSpeedX; **//moveSpeedXはプライベートなので外部からアクセスできない。**  **//コンパイルエラーが発生する！！！！**  pl.MoveX(); //X方向に移動させる。パプリックなのでアクセスできる。  } |

上のコードのコメントに書いてある通り、moveSpeedXはprivateが指定されているためクラス内部からしかアクセスできなくなっています。そのため、Playerクラスのメンバ関数であればアクセスできますが、外部関数のmain関数からアクセスするとコンパイルエラーが発生するようになります。さて、なぜprivate指定子のようなものがあるのでしょうか？すべてpublicでアクセスできる方がよほど分かりやすいと思いませんか？ではこの話は次のカプセル化の節でお話しします。

## **2.7 カプセル化**

　カプセル化とはクラスのメンバ変数をクラス内で隠蔽して外部からアクセスできないように保護することです。2.6.2のサンプルコードで、Playerクラスのメンバ変数のmoveSpeedXのアクセス指定子をprivateにしていました。これがカプセル化です。

　カプセル化とはクラスの内部情報を外部に見せないことによって、クラスのインスタンスを保護して、デバッグ、拡張を容易することができます。

## **2.7.1 getter**

　プレイヤーの座標を表すメンバ変数が下記のコードのようにprivateになっていると、外部からアクセスすることができなくなります。

|  |
| --- |
| class Player{  　　・  　　・  　省略  　　・  　　・  private:  Vector3 m\_position; //座標を表すメンバ変数。  }; |

しかし、これでは下記のコードのように、コンパイルエラーが出てしまって、ゲームを製作するのがとても困難になります。プレイヤーとエネミーの距離計算を行うこともできなくなりますし、マリオのようにコインを取得する処理も書けなくなるでしょう。

|  |
| --- |
| Player g\_player; //Playerクラスのインスタンス。  　・  　・  　省略  　・  　・  //EnemyクラスのUpdate関数。  void Enemy::Update()  {  　　//プレイヤーとの距離計算をしたいのだけど・・・、m\_positionはprivateなので  　 //外部のクラスからはアクセスできない！！！  Vector3 diff = m\_position – **g\_player.m\_position**;  　　・  　　　　・  　　　　・  　　　省略  　　　　・  　　　　・  　　　　・  } |

　このように、privateのメンバ変数の値を使用したい場合は、下記のようにgetterを作成するのが一般的です。

|  |
| --- |
| class Player{  public:  Vector3 GetPosition()  {  return m\_position;  }  　　・  　　・  　省略  　　・  　　・  private:  Vector3 m\_position; //座標を表すメンバ変数。  }; |

そして、外部のクラスでは下記のようにgetterを使用して、Playerクラスのm\_positionの値を読み取ります。

|  |
| --- |
| //EnemyクラスのUpdate関数。  void Enemy::Update()  {  　　//getterを使用して、プレイヤーの座標をplayerPosにコピーする。  　　Vector3 playerPos = g\_player.GetPosition();  Vector3 diff = m\_position – **playerPos**;  　　　・  　　　　・  　　　　・  　　　省略  　　　　・  　　　　・  　　　　・  } |

## **2.7.2 setter**

続いて、下記のように、プレイヤーの座標を設定する場合を考えていきましょう。

|  |
| --- |
| //プレイヤーを瞬間移動させるギミックの更新関数。  void TeleportationGimmick::Update()  {  Vector3 newPlayerPos;  ・  　　・  　　・  　　省略  　　・  　　・  　　・  **g\_player.m\_position** = newPlayerPos; //m\_positionはprivateなのでコンパイルエラー！  } |

このように、privateのメンバ変数に値を代入したい場合は、下記のようにsetterを作成します。

|  |
| --- |
| class Player{  public:  Vector3 GetPosition()  {  return m\_position;  }  Void SetPosition(Vector3 pos)  {  m\_position = pos;  }  　　・  　　・  　省略  　　・  　　・  private:  Vector3 m\_position; //座標を表すメンバ変数。  }; |

値を代入するコードは下記のようになります。

|  |
| --- |
| //プレイヤーを瞬間移動させるギミックの更新関数。  void TeleportationGimmick::Update()  {  Vector3 newPlayerPos;  ・  　　・  　　・  　　省略  　　・  　　・  　　・  **g\_player.SetPosition( newPlayerPos ); //m\_positionはprivateなのでコンパイルエラー！**  } |

## **2.7.3 getter/setter**

続いて、getterとsetterを使用して、プレイヤーを動かす方法を見ていきましょう。下記のようにプレイヤーに風力を与えて、動かすギミックがあると考えてください。

|  |
| --- |
| //プレイヤーに風力を与えるギミック。  void FanGimmick::Update()  {  Vector3 addPower;  　　　・  　　　・  　　風力を計算。  　　　・  　　　・  　　　・  　 g\_player.m\_position += addPower; //m\_positionはアクセスできない。コンパイルエラー！！  } |

　このような処理を行う場合には、下記のようにgetterとsetterの両方を使います。

|  |
| --- |
| //プレイヤーに風力を与えるギミック。  void FanGimmick::Update()  {  Vector3 addPower;  　　　・  　　　・  　　風力を計算。  　　　・  　　　・  　　　・  　 Vector3 pos = g\_player.GetPosition();  pos += addPower;  g\_player.SetPosition(pos);  } |

getterを使用して、ローカル変数のposに値を代入して、posに風力を加算して、新しい座標を計算して、それをプレイヤーに書き戻しています。少し面倒ですが、これがgetterとsetterを使用する場合の典型的なパターンです。

## **2.7.4 getterで内部ハンドルを返してはいけない。**

　2.7.3のgetter/setterを使用して、内部データを操作するパターンですが、実は下記のような実装にすると、コードがシンプルになります。

|  |
| --- |
| class Player{  public;  //m\_positionの参照を返している！！！  **Vector3&** GetPosition()  {  return m\_position;  }  　　・  　　・  　省略  　　・  　　・  Private:  Vector3 m\_position; //座標を表すメンバ変数。  }; |

　GetPositionがm\_positionの参照を返しています。では、このPlayerクラスを使って、2.7.3のFanGimmick::Update関数を書き換えてみましょう。

|  |
| --- |
| //プレイヤーに風力を与えるギミック。  void FanGimmick::Update()  {  Vector3 addPower;  　　　・  　　　・  　　風力を計算。  　　　・  　　　・  　　　・  **//g\_player.m\_positionの参照を受け取る！**  **Vector3& pos = g\_player.GetPosition();**  **//posはg\_player.m\_positionの参照なので、**  **//posの値を変更すると、g\_player.m\_positionの値も変更される。**  **pos += addPower;**  } |

これで2.7.3のようなまどろっこしいコードを書く必要はなくなりました。しかし、このコードはメンバ変数がpublicになっていることと何ら変わらない状態になっています。これではカプセル化は行えていないことと同じです。このようなコードは記述しないように注意しましょう。

## **2.9 カプセル化のメリット**

　さて、ここまでの説明ではカプセル化を行うメリットはあまり感じられなかったのではないでしょうか。コードを記述する量は増え、無駄に複雑になっただけになったように感じます。では、カプセル化のメリットについて見ていきましょう。

クラスのメンバ変数にpublic指定されていたら、その変数はプログラムどこからでもアクセス可能になってしまいます。ウン万行もソースコードがあるような、大きなゲームを作っていると考えてみて下さい。そして、ある時Playerクラスのメンバ変数のhpに仕様上あり得ない数字(例えばマイナスの数字とか)が代入されている不具合が発生したことを考えてみて下さい。もしこのメンバ変数にpublicが指定されていたらあなたはウン万行あるソースコードの中から、この不具合を発生させている箇所を探しださなくてはいけなくなります。ではhpがprivateに指定されていたらどうなるでしょうか？privateに指定されている場合は、その変数に値を代入するためにはメンバ関数を記述する必要があります。つまり、あなたのPlayerクラスにはSetHPなどのようなメンバ関数があるはずです。

|  |
| --- |
| class Player{  private:  int hp;  public:  //メンバ変数のhpに値をセットするためのメンバ関数。  void SetHP( int value );  };  void Player::SetHP( int value )  {  hp = value;  } |

hpがカプセル化されている場合、あなたは短いプログラムを記述するだけで不具合を起こしている箇所を特定することができます。

|  |
| --- |
| void Player::SetHP( int value )  {  　if(value < 0){  std::cout << “valueの値が不正！！！”;  std::abort(); //プログラムを停止させる命令。  }  hp = value;  } |

## **2.10 実習**

　① 　Lesson\_3\_1のゲームはあるタイミングで表示がおかしくなる不具合があります。この不具合の原因はtkEngine/graphics/tkCamera.hのCCameraクラスのメンバ変数m\_positionに非数(0除算などが行われている)が設定されていることが原因です。この不具合が発生している箇所を突き止めなさい。

　②　 Lesson\_3\_2のtkEngine/graphics/tkCamera.hのCCameraクラスのメンバ変数m\_positionはアクセス指定子がprivateになっており、SetPosition関数を使って、値が変更されます。SetPositionの関数を改造して、不具合が発生している箇所を特定しなさい。なお、isnanというC言語の標準関数を使用すれば非数判定を行うことができます。

　③　Lesson\_3\_3のPlayerクラスとEnemyクラスのリファクタリングを行い、メンバ変数のアクセス指定子をすべてprivateに変更しなさい。

**2.9 コンストラクタとデストラクタ**

　クラスにはコンストラクタとデストラクタという特殊なメンバ関数を記述することができます。コンストラクタとデストラクタの記述の仕方は下記のようになります。

|  |
| --- |
| //プレイヤークラス。  class Player{  クラスの名前と同じ名前で宣言する。戻り値は記述できない。  public:  //コンストラクタの宣言。  Player();  //デストラクタの宣言。  ~Player();  };  クラス名の前に~(チルダ)を追加して宣言する。戻り値と引数は記述できない。  //コンストラクタの定義。  Player::Player()  {  }  //デストラクタの定義。  Player::~Player()  {  } |

では、次の節からコンストラクタとデストラクタについて具体的に見ていきます。

**2.9.1 コンストラクタ**

コンストラクタはクラスのインスタンスが生成されたときに自動で呼び出される関数で、メンバ変数などの初期化を行うためのものとなります。では先ほどのPlayerクラスのコンストラクタにメンバ変数の初期化のコードを記入してみましょう。

|  |
| --- |
| //プレイヤークラス。  class Player{  private:  D3DXVECTOR3 position; //座標。  D3DXQUATERNION rotation; //回転。  int hp; //ヒットポイント。  int attack; //攻撃力。  int deffence; //防御力。  public:  //コンストラクタの宣言。  Player();  //デストラクタの宣言。  ~Player();  //HPを取得  　　int GetHP();  };  **//コンストラクタの定義。**  **Player::Player()**  **{**  **//座標を0.0f, 0.0f, 0.0fに初期化。**  **position.x = 0.0f;**  **position.y = 0.0f;**  **position.z = 0.0f;**    **//回転を単位クォータニオンで初期化。**  **rotation.x = 0.0f;**  **rotation.y = 0.0f;**  **rotation.z = 0.0f;**  **rotation.w = 1.0f;**    **hp = 0; //ヒットポイントを0で初期化。**  **attack = 0; //攻撃力を0で初期化。**  **deffence = 0; //防御力を0で初期化。**  **}**  //デストラクタの定義。  Player::~Player()  {  }  //HPを取得  int Player::GetHP()  {  return hp;  } |

このようにコンストラクタで初期化を行うコードを追加しました。ではコンストラクタがどのように呼び出されるのか、Playerクラスのインスタンスを生成するコードを見て確認してみましょう。

|  |
| --- |
| int main()  {  Player p; //プレイヤークラスのpというインスタンスを作成する。  int hp = p.GetHP();  　std::cout << “プレイヤーのHP = ” << hp;  return 0;  } |

このプログラムをコンパイルすると、コンパイラが自動でコンストラクタを呼び出すコードを追加します。ではそれを確認するために、コンパイラが生成するこのプログラムのアセンブラコードを見てみましょう。

|  |
| --- |
| Player p; //プレイヤークラスのpというインスタンスを作成する。  00DB2448 lea ecx,[p]  コンストラクタ呼び出しが追加されている。  00DB244B call Player::Player (0DB10AAh)  int hp = p.GetHP();  00DB2450 lea ecx,[p]  00DB2453 call Player::GetHP (0DB114Ah)  00DB2458 mov dword ptr [hp],eax |

このように、コンパイラによってPlayerのインスタンスが生成された直後にコンストラクタ呼び出しが追加されます。

**2.9.1 デストラクタ**

デストラクタはインスタンスが破棄されるときに呼び出される関数です。インスタンスを破棄する際に後始末を行う必要がある場合にデストラクタに記述を行います。では先ほどのPlayerクラスに後始末の必要があるメンバ変数を追加して、デストラクタの中身を記述してみます。

|  |
| --- |
| class Player{  private:  D3DXVECTOR3 position; //座標。  D3DXQUATERNION rotation; //回転。  int 　　hp; //ヒットポイント。  int attack; //攻撃力。  int deffence; //防御力。  SkinModel\* skinModel; //モデル  public:  //コンストラクタの宣言。  Player();  //デストラクタの宣言。  ~Player();  };  //コンストラクタの定義。  Player::Player()  {  //座標を0.0f, 0.0f, 0.0fに初期化。  position.x = 0.0f;  position.y = 0.0f;  position.z = 0.0f;  //回転を単位クォータニオンで初期化。  rotation.x = 0.0f;  rotation.y = 0.0f;  rotation.z = 0.0f;  rotation.w = 1.0f;  hp = 0; //ヒットポイントを0で初期化。  attack = 0; //攻撃力を0で初期化。  deffence = 0; //防御力を0で初期化。  **skinModel = new SkinModel; //モデルクラスのインスタンスを生成。**  後始末  }  **//デストラクタの定義。**  **Player::~Player()**  **{**  **delete skinModel; //newされたのでdeleteを行う。**  **}** |

メンバ変数のskinModelはコンストラクタでnewを行っているため、デストラクタで確実にdeleteするようなコードを追加しました。デストラクタもコンストラクタと同様にインスタンスの破棄のタイミングで、コンパイラによって自動で呼び出すコードが生成されます。

**2.9.2 引数付きコンストラクタ**

　コンストラクタは引数を渡すことができます。ではプレイヤークラスに初期座標を受け取るコンストラクタを追加してみましょう。

|  |
| --- |
| //プレイヤークラス。  class Player{  private:  D3DXVECTOR3 position; //座標。  D3DXQUATERNION rotation; //回転。  int hp; //ヒットポイント。  int attack; //攻撃力。  int deffence; //防御力。  public:  //コンストラクタの宣言。  Player( int x, int y, int z );  //デストラクタの宣言。  ~Player();  //HPを取得  　　int GetHP();  };  座標の引数を受け取る  //コンストラクタの定義。  **Player::Player( int x, int y, int z )**  {  //引数を使用して座標を初期化。  **position.x = x;**  **position.y = y;**  **position.z = z;**    //回転を単位クォータニオンで初期化。  rotation.x = 0.0f;  rotation.y = 0.0f;  rotation.z = 0.0f;  rotation.w = 1.0f;    hp = 0; //ヒットポイントを0で初期化。  attack = 0; //攻撃力を0で初期化。  deffence = 0; //防御力を0で初期化。  }  //デストラクタの定義。  Player::~Player()  {  } |
| tip  コンストラクタにはコピーコンストラクタという特殊なコンストラクタがあります。下記のような記述をするとコピーコンストラクタを実装できます。  class Player{  public:  Player(const Player& player); //コピーコンストラクタ！！！  };  コピーコンストラクタは定義を行わない場合、コンパイラがデフォルトの実装のコピーコンストラクタの実装を勝手に行います。このコピーコンストラクタは浅いコピーが行われます。コンパイラが実装するコピーコンストラクタは予期せぬ不具合を発生させることがあるため、これを削除するNoncopyableイディオムと言われるテクニックも存在します。会社のコーディングルールによってはデフォルトのコピーコンストラクタは使用禁止というものまであります。 |

**章末テスト(60分)**

①　２次元のベクトルクラス、Vector2を作成しなさい。Vector2の要求仕様は下記にな

ります。(20点)

・メンバ変数にint型のm\_xとm\_yを保持する。アクセス指定子はprivate。

・デフォルトコンストラクタでm\_xとm\_yを0で初期化する。

・m\_xの値を返すGetX、m\_yの値を返すGetYというメンバ関数を追加する。アクセス指定子はpublic。

・m\_xに値を設定するSetX、m\_yに値を設定するSetYというメンバ関数を追加する。アクセス指定子はpublic。

②　Playerクラスの座標を表すメンバ変数のm\_posX、m\_posYを削除して、①で作成したVector2型のm\_pos変数を追加しなさい。そしてコンパイルエラーをすべて修正してきちんとゲームが元通り動作するようにしなさい。(20点)

③　Foodクラスの座標を表すメンバ変数のm\_posX、m\_posYを削除して、①で作成したVector2型のm\_pos変数を追加しなさい。そしてコンパイルエラーをすべて修正してきちんとゲームが元通り動作するようにしなさい。(20点)

④ Wallクラスの座標を表すメンバ変数のm\_posX、m\_posYを削除して、①で作成したVector2型のm\_pos変数を追加しなさい。そしてコンパイルエラーをすべて修正してきちんとゲームが元通り動作するようにしなさい。(20点)

⑤　Food.cppのFood::Update関数を改造して食べ物とプレイヤーが接触したら食べ物が消えるようしなさい。(20点)

**Chapter 3**

**継承**

C++の強力な機能の一つに継承というものがあります。この継承という考え方はオブジェクト指向を用いた設計、デザインパターンを学ぶ上で非常に重要な概念になります。このチャプターでは継承について見ていきましょう。

**3.1 メンバ変数の継承**

　まずは、簡単な例で継承を見ていきましょう。例えばレースゲームを作成していることを考えてみて下さい。車には色々な車種があります。ワゴンＲ、フィット、ヴィッツ、フェラーリ、ポルシェなどなど。これらは当然車種ごとに、ステアリング性能、加速性能、燃費、車体フレームなど異なる点が多数存在します。しかし、どの車種も車であることに違いはありません。そのため共通点がいくつか存在します。タイヤは４つ付いていますし、ハンドルも付いています。アクセル、ブレーキなども付いているはずです。これらの共通部分を抽出して下記のようなクラスを作成します。

|  |
| --- |
| //車の基底クラス。  class CarBase{  private:  Tire tire[4]; //タイヤ  Handle handle; //ハンドル  BrakePedal brakePedal; //ブレーキペダル  　 AxellPedal axellPedal; //アクセルペダル。  }; |

そして、各車種はCarBaseクラスを継承して実装します。

|  |
| --- |
| //フィット  class Fit : **public CarBase**{  Model model; //フィットの車体モデル。  };  //ワゴンＲ  class WagonR : **public CarBase**{  Model model; //ワゴンＲの車体モデル。  };  //フェラーリ  class Ferrari : **public CarBase**{  Model model; //フェラーリの車体モデル。  }; |

このように記述を行うことで、各車種はCarBaseクラスを継承することができます。そして、各クラスはメンバ変数として、tire[4]、handle、brakePedal、axellPedalを保持することになります。

**3.2 メンバ関数の継承**

　継承はメンバ変数のみではなく、メンバ関数も継承することができます。先ほどの車を例にして見ていきましょう。車には走る処理のRun関数、窓を開けるOpenWindow関数、ドアを開けるOpenDoor関数などなど、いくつも共通の処理が存在するはずです。C++では、このような処理を基底クラスのメンバ関数として記述することで、どの車でも共通の処理として定義することができます。

**派生クラスでアクセスしたい場合はprotected!!!**

車の基底クラスは下記のようになるでしょう。

|  |
| --- |
| class CarBase{  **protected**:  Tire tire[4]; //タイヤ  Handle handle; //ハンドル  BrakePedal brakePedal; //ブレーキペダル  　 AxellPedal axellPedal; //アクセルペダル。  public:  //走る処理  void Run();  　　//窓開ける。  void OpenWIndow();  　　//ドア開ける  void OpenDoor();  }; |

この基底クラスを継承した派生クラスはメンバ関数として、Run、OpenWIndow、OpenDoorを保持するようになります。

**3.3 継承すべきか委譲すべきか**

オブジェクト指向のクラス設計において、車の例のような共通処理のクラス化は継承の他に委譲というテクニックが存在します。では継承と委譲の使い分けはどのようにすればいいのか？ この指針としてよく言われるものに下記のようなものがある。

**クラス間の関係がis-aの場合は継承、has-aの場合は委譲。**

is-aとは、「フェラーリは車である」のように、派生クラス=基底クラスが成り立つ場合のことを言います。

has-aの場合は「フェラーリはブレーキペダルを持っている」という場合になります。

is-aの場合は継承を行うことを検討する、has-aの場合は委譲を行うことを検討してみることが設計の指針になります。

**3.4 仮想関数**

　さて、先ほどのドアを開ける処理ですが、車種によってはスライドドアのものがあれば、従来の引手のドアもあるでしょう。そのため基底クラスのOpenDoor関数に引手のドアの処理を記述している場合、問題が出てきます。もちろんif文などで処理を分けてもいいでしょう。しかしC++であればこれを仮想関数というものを使用することによって、スマートに解決することができます。仮想関数とは派生クラスで実装を変更できる関数になります。仮想関数の実装は下記のようになります。

|  |
| --- |
| class CarBase{  private:  Tire tire[4]; //タイヤ  Handle handle; //ハンドル  BrakePedal brakePedal; //ブレーキペダル  AxellPedal axellPedal; //アクセルペダル。  public:  //走る処理  **これが仮想関数！！**  **デフォルトの実装は引きドア**  void Run();  //窓開ける。  void OpenWIndow();  //ドア開ける  **virtual** void OpenDoor();  };  //デリカ  class Delica : public CarBase{  Model model; //デリカの車体モデル。  **void OpenDoor(); //デリカはスライドドアなので、オーバーライドする！**  };  //ワゴンＲ  class WagonR : public CarBase{  Model model; //ワゴンＲの車体モデル。  };  //フィット  class Fit : public CarBase{  Model model; //フェラーリの車体モデル。  }; |

ワゴンＲとFITは引きドアなので、CarBaseに実装されているデフォルトのOpenDoor関数を使用しています。しかしデリカはスライドドアのため、OpenDoorを**オーバーライド**しています。基底クラスの仮想関数の実装を派生クラスで再定義することをオーバーライドといいます。

さて、ここまでの説明だと仮想関数がなぜ必要なのかピンと来ないのではないかと思います。なぜなら仮想関数なんて使わなくても下記のようにしてやれば引き戸の処理は実装できるからです。

|  |
| --- |
| //デリカ  class Delica : public CarBase{  Model model; //デリカの車体モデル。  **void OpenDoorSlideDoor(); //スライドドア用のドアを開く関数を別名定義。**  }; |

仮想関数が真価を発揮するのはポリモーフィズム(多態性)と言われる機能を使うときになります。では次の節では多態性について見ていきましょう。

**3.5 ポリモーフィズム(多態性)**

**「基底クラスのポインタ型の変数に、派生クラスのインスタンスのアドレスを代入すると、あたかも派生クラスのインスタンスであるかのように振舞う」**ことをいいます。ではサンプルコードを見てみましょう。

|  |
| --- |
| class HogeBase{  public:  //仮想関数版のPrint関数  virtual void Print()  {  std::cout << “HogeBase\n”;  }  };  class Hoge : public HogeBase{  public:  void Print()  {  std::cout << “Hoge\n”;  }  };  int main()  {  Hoge hoge;  HogeBase\* hogeBase = &hoge; //HogeBase型のポインタ変数にhogeのアドレスを代入。  **hogeBase->Print(); //Hogeと表示される。これがポリモーフィズム。**  return 0;  } |

このように、HogeBaseを継承しているHogeクラスのインスタンスは、基底クラスのHogeBaseのポインタ型の変数にアドレスを代入できます。そして、HogeBase型のポインタはPrint関数を呼び出すと、あたかもHogeであるかのように振る舞います。Print関数が仮想関数でない場合は、HogeBaseと表示されます。

**3.5.1 ポリモーフィズムを使う理由。**

では、前節のレースゲームを例にして考えてみましょう。レースゲームではユーザーがレースを始める前に自分が操作する車を選択します。そして、ゲーム中のアップデート関数では選択した車に対する操作(ブレーキやアクセルやドアを開くなど)が実行されるはずです。では、ポリモーフィズムを知らない不幸なコードを見てみましょう。

|  |
| --- |
| //car.h  //車の基底クラス  class CarBase{  private:  Tire tire[4]; //タイヤ  Handle handle; //ハンドル  BrakePedal brakePedal; //ブレーキペダル  AxellPedal axellPedal; //アクセルペダル。  public:  //ブレーキをかける処理  virtual void Brake();  //アクセル  virtual void Accell();  //走る処理  virtual void Run();  };  //デリカ  class Delica : public CarBase{  //ブレーキをかける処理  void Brake();  //アクセル  void Accell();  };  //ワゴンＲ  class WagonR : public CarBase{  //ブレーキをかける処理  void Brake();  };  //フィット  class Fit : public CarBase{  //アクセル  void Access();  }; |

|  |
| --- |
| //car.cpp  #include “car.h”  Int selectCarType; //0だとデリカ、1だとワゴンR、２だとFIT  Delica delica;  WagonR wagonR  Fit fit;  //ブレーキの処理。  void Brake()  {  If(selectCarType == 0){  //デリカ  delica.Brake();  }else if(selectCarType == 1){  //ワゴンR  wagonR.Brake();  }else if(selectCarType == 2){  //フィット  fit.Brake();  }  }  //アクセルの処理。  void Accell()  {  If(selectCarType == 0){  //デリカ  delica.Accell();  }else if(selectCarType == 1){  //ワゴンR  wagonR.Accell();  }else if(selectCarType == 2){  //フィット  fit.Accell();  }  }  //走る処理。  void Run()  {  If(selectCarType == 0){  delica.Run();  }else if(selectCarType == 1){  wagonR.Run();  }else if(selectCarType == 2){  fit.Run();  }  }  //メイン関数  int main()  {  　std::cin >> selectCarType;  　while(true){ //ゲームループ。  //ブレーキの処理  　 Brake();  //アクセルの処理。  　 Accell();  //走る処理。  　 Run();  }  } |

ポリモーフィズムを知らないプログラマはこのようなコードを書くと思います。実際のレースゲームであれば、車種はもっと多いはずなのでselectCarTypeを使用したif文の数は100を軽く超えることになるでしょう。そして。この不幸なプログラマは下記のような仕様変更が発生した時に定時で帰ることはできなくなるでしょう。

**「クライアントから車のドアを開けられるようにして欲しいという要望が来たので対応してください。」**

このコードを書いたプログラマはOpenDoorという処理を記述して、また新しくselectCarTypeの条件文を追加して、OpenDoorという関数の呼び出しを100箇所以上記述することになります。

また、次のような要望が来た時も悲しいことになります。

**「クライアントから新しい車種を追加して欲しいという要望が来たので対応してください。」**

あなたはAccell関数、Brake関数、Run関数に新しいselectCarTypeを使用する条件文を記述して、各種メンバ関数の呼び出しコードを記述することになります。そして、ある日このプログラマは、また新しい車種が追加されたときにRun関数だけコードを追加することを忘れてしまって、不具合に頭を悩ませることになるでしょう。

では、このプログラムをポリモーフィズムを使用するプログラムで書き換えてみましょう。ヘッダーファイルに変更点はありません。

|  |
| --- |
| //car.cpp  #include “car.h”  Int selectCarType; //0だとデリカ、1だとワゴンR、２だとFIT  Delica delica;  WagonR wagonR  Fit fit;  **CarBase\* carBaseArray[3]; //CarBaseのポインタ型の配列**  //ブレーキの処理。  void Brake()  {  **carBaseArray[selectCarType]->Brake(); //これがポリモーフィズム！！！**  }  //アクセルの処理。  void Accell()  {  **carBaseArray[selectCarType]->Accell(); //これがポリモーフィズム！！！**  }  //走る処理。  void Run()  {  **carBaseArray[selectCarType]->Run(); //これがポリモーフィズム！！！**  }  //メイン関数  int main()  {  **carBaseArray[0] = & delica; //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。**  **carBaseArray[1] = & WagonR; //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。**  **carBaseArray[2] = &Fit; //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。**  　std::cin >> selectCarType;  　while(true){ //ゲームループ。  //ブレーキの処理  　 Brake();  //アクセルの処理。  　 Accell();  //走る処理。  　 Run();  }  } |

非常にシンプルな短いコードになりました。これがポリモーフズムを活用したプログラムになります。ポリモーフズムとは**同じ操作で、異なる動作をするもの**となります。このようなコードを記述したプログラマであれば、新しい車種が追加された場合、追加で記入するプログラムは下記のたった一行になります(もちろん新しい車種のクラスは作成しますが)。

|  |
| --- |
| int main()  {  carBaseArray[0] = & delica; //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。  carBaseArray[1] = & WagonR; //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。  carBaseArray[2] = &Fit; //派生クラスのインスタンスアドレスを基底クラスのポインタ型の変数に代入。  **carBaseArray[3] = &vitz //VITZを追加！**  　std::cin >> selectCarType;  　while(true){ //ゲームループ。  //ブレーキの処理  　 Brake();  //アクセルの処理。  　 Accell();  //走る処理。  　 Run();  }  } |

Brake関数、Accell関数、Run関数に新しい条件文を記述する必要は全くありません。なぜならば、vitzのインスタンスのアドレスを代入されたcarBaseArrayはあたかもでvitzであるかのように振舞うからです。

新しいOpenDoorという関数が追加されても下記の処理の追加だけで完了します。

|  |
| --- |
| carBaseArray[selectCarType]->OpenDoor(); |

ポリモーフィズムを上手に活用したプログラムは関数呼び出しの追加忘れや仕様変更に非常に強いプログラムになり、ヒューマンエラーの発生を大きく下げてくれます。

Chapter 4 std::vector

4.4 Update Methodパターン

　では、動的配列とポリモーフィズムを学んだので、その二つの活用事例として、ゲームで古くから使われ続けてきた、Update Methodパターンというデザインパターンを見ていきましょう。

　Update Methodパターンとはインスタンスを登録すると、自動的に関数が呼ばれるというものです。これまで皆さん、tkEngineのNewGOを実行すると、勝手にUpdateやStartといった関数が呼ばれていたと思います。基本的にメンバ関数というのは明示的に呼び出すコードを書いてやらないと実行されることはありません。つまり、NewGOを実行してインスタンスを作成した場合、何かがそのインスタンスのUpdateやRender関数を呼び出してくれていたことになります。tkEngineにはゲームオブジェクトマネージャというクラスがあり、NewGOで生成されたインスタンスはゲームオブジェクトマネージャのインスタンスに登録されていたのです。そして、ゲームオブジェクトマネージャが登録されたインスタンスのUpdateやRender関数を呼び出してくれていました。

4.5 ゲームオブジェクトマネージャ

ではLesson6のサンプルプログラムを使って、簡単はゲームオブジェクトマネージャの実装を見ていきましょう。IGameObject.hを開いてください。

IGameObject.h

|  |
| --- |
| #pragma once  class IGameObject  {  public:  IGameObject();  virtual ~IGameObject();  virtual void Update()  {  　　　　　　　printf("IGameObject::Update\n");  }  }; |

IGameObjectクラスは仮想関数のUpdate関数を保持しているクラスです。では、続いてGameObjectManager.hを開いてください。

GameObjectManager.h

|  |
| --- |
| class GameObjectManager  {  public:  GameObjectManager();  ~GameObjectManager();  void AddGameObject(IGameObject\* go);  void Update();  private:  std::vector<IGameObject\*> m\_gameObjects;  }; |

GameObjectManagerはIGameObject\*の動的配列のm\_gameObjectsというメンバ変数を保持しています。

では、続いてGameObjectManager.cppを開いてください。

GameObjectManager.cpp

|  |
| --- |
| void GameObjectManager::AddGameObject(IGameObject\* go)  {  m\_gameObjects.push\_back(go);  }  void GameObjectManager::Update()  {  for (IGameObject\* go : m\_gameObjects) {  go->Update();  }  } |

AddGameObject関数でIGameObjectのインスタンスをm\_gameObjectsに追加しています。Update関数ではforeachを使用して、登録されているIGameObjectのインスタンスのUpdate関数を呼び出しています。

では最後にstl\_vector.cppを開いてください。

|  |
| --- |
| #include<thread>  class Player : public IGameObject  {  public:  void Update()  　　{  printf("Player!!!\n");  　　}  };  class Enemy : public IGameObject  {  public:  void Update()  　　{  printf("Enemy!!!\n");  　　}  };  int main()  {  //ゲームオブジェクトマネージャ。  GameObjectManager goMgr;  Player\* p = new Player(); //プレイヤーのインスタンスを作成して  goMgr.AddGameObject(p); //ゲームオブジェクトマネージャーに登録する。  Enemy\* e = new Enemy(); //エネミーのインスタンスを作成して  goMgr.AddGameObject(e); //ゲームオブジェクトマネージャに登録する。  //ゲームループ。  while (true) {  //ゲームオブジェクトマネージャに登録されている  //オブジェクトのアップデートをすべて呼び出だす！！！  goMgr.Update();  //眠る。  std::this\_thread::sleep\_for(std::chrono::seconds(1));  　　}  return 0;  } |

IGameObjectを継承したPlayerとEnemyクラスのインスタンスを生成して、GameObjectManagerに登録を行っています。GameObjectManagerは登録されたインスタンスの正体がPlayerなのかEnemyなのかということは一切知らずに、IGameObjectのインスタンスということしか知りません。しかし、GameObjectManagerがUpdate関数を呼び出すと、PlayerやEnemyのUpdateが正しく呼ばれます。これがポリモーフィズムです。

**4.6 純粋仮想関数と抽象クラス**

4.5で実装したUpdateMethodパターンをもう少し優れた設計に変更するのを検討してみましょう。4.5で実装したUpdateMethodパターンには、**IGameObjectを継承したPlayerクラスでUpdateをオーバーライドをするのを忘れてしまった**、というヒューマンエラーが発生する可能性があります。

下記のようなコードが書けてしまうのです。

|  |
| --- |
| #include<thread>  class Player : public IGameObject  {  public:  //Update関数をオーバーライドしてない。  };  int main()  {  //ゲームオブジェクトマネージャ。  GameObjectManager goMgr;  Player\* p = new Player(); //プレイヤーのインスタンスを作成して(問題なし)  goMgr.AddGameObject(p); //ゲームオブジェクトマネージャーに登録する。  //ゲームループ。  while (true) {  //ゲームオブジェクトマネージャに登録されている  //オブジェクトのアップデートをすべて呼び出だす！！！  goMgr.Update();  //眠る。  std::this\_thread::sleep\_for(std::chrono::seconds(1));  　　}  return 0;  } |

このように、IGameObjectを継承しているが、Updateをオーバーライドしていない場合、基底クラスのIGameObjectのUpdate関数が呼ばれることになります。もちろんオーバーライドをせずに、基底クラスの提供する処理を呼び出すというのは、なにも問題があることではありません。それを望んでいることもたくさんあるでしょう。しかし今回のUpdateMethodパターンでは、これはおそらく不具合だと思われます。なぜならUpdateMethodパターンは、必ず継承先でUpdate関数をオーバーライドすることを求めているパターンだからです。この欲求を実現するための機能が純粋仮想関数といわれるものです。ではIGameObjectのUpdate関数を下記のように純粋仮想関数に変更してみてください。

|  |
| --- |
| class IGameObject  {  public:  IGameObject();  virtual ~IGameObject();  **virtual void Update() = 0;**  }; |

これでUpdate関数は実装を持たない純粋仮想関数になりました。純粋仮想関数を持っているクラスは抽象クラスと呼ばれるクラスになます。抽象クラスはインスタンスを生成することができない、という制約があります。つまり、**IGameObjectを継承したクラスはUpdate関数をオーバーライドして実装しないとインスタンスを作ることができない**、ということです。

**実習**

1. Lesson6/純粋仮想関数を改造して、IGameObjectにvoid Render()という純粋仮想関数を追加しなさい。そして、それが毎フレーム呼び出されるようにプログラムを改造しなさい。
2. Lesson6/純粋仮想関数を改造して、IGameObjectにvoid Render()という純粋仮想関数を追加しなさい。そして、それが初めてUpdate関数が呼ばれるときに一度だけ実行されるように改造しなさい。

Chapter 5 std::list

5.1 listとvectorの違い

　.１　要素へのランダムアクセスができない。

　　　　->配列の添え字アクセスのこと。

　　　 　例えば、先頭から10番目の要素にアクセスしたい場合、vectorの場合は一発で

　　　　　アクセスすることができるが、listの場合は先頭から走査していく必要がある。

.２　要素の削除と追加

Vectorは要素の削除と追加に時間がかかる。

　　 Listは速い。

5.2 listを使うべきか？vectorを使うべきか？

　listとvectorの特性の違いから、listとvectorの使い分けが見えてきます。一般的にvectorは頻繁に要素の追加や削除が発生しない場合は最もよい選択肢となります。反対に要素の削除や追加が頻繁に発生する場合はlistを使用することを検討すべきかもしれません。ただし、ゲームプログラミングにおいて、頻繁な要素の追加、削除が発生する場合は注意が必要でしょう。多くの場合で、動的に要素が増えるとしても、要素の上限というのは決まっているものです。そのような場合はvectorのreserve関数を使用して、必要なメモリを最初から確保しておけば、要素の追加にかかるコストを減らすことができます。メモリの断片化、キャッシュ効率についてもゲームプログラミングを行う場合は注意を払うべきです。

5.3 vectorとlistのを見てみる。