

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 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、世の中の全てのプログラマの頭を悩ませ、睡眠時間を削り、精神力、体力ともに疲弊させる厄介なものです。プロとして働き出すと、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。デバッガの使い方を習得して、定時に帰宅できるプログラマになりましょう。

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のでしょうか？`ctrl+F5` の実行は「デバッグなしで実行」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。

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。

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。

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。

3.2.1 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。

3.2.2 ステップイン(F11)

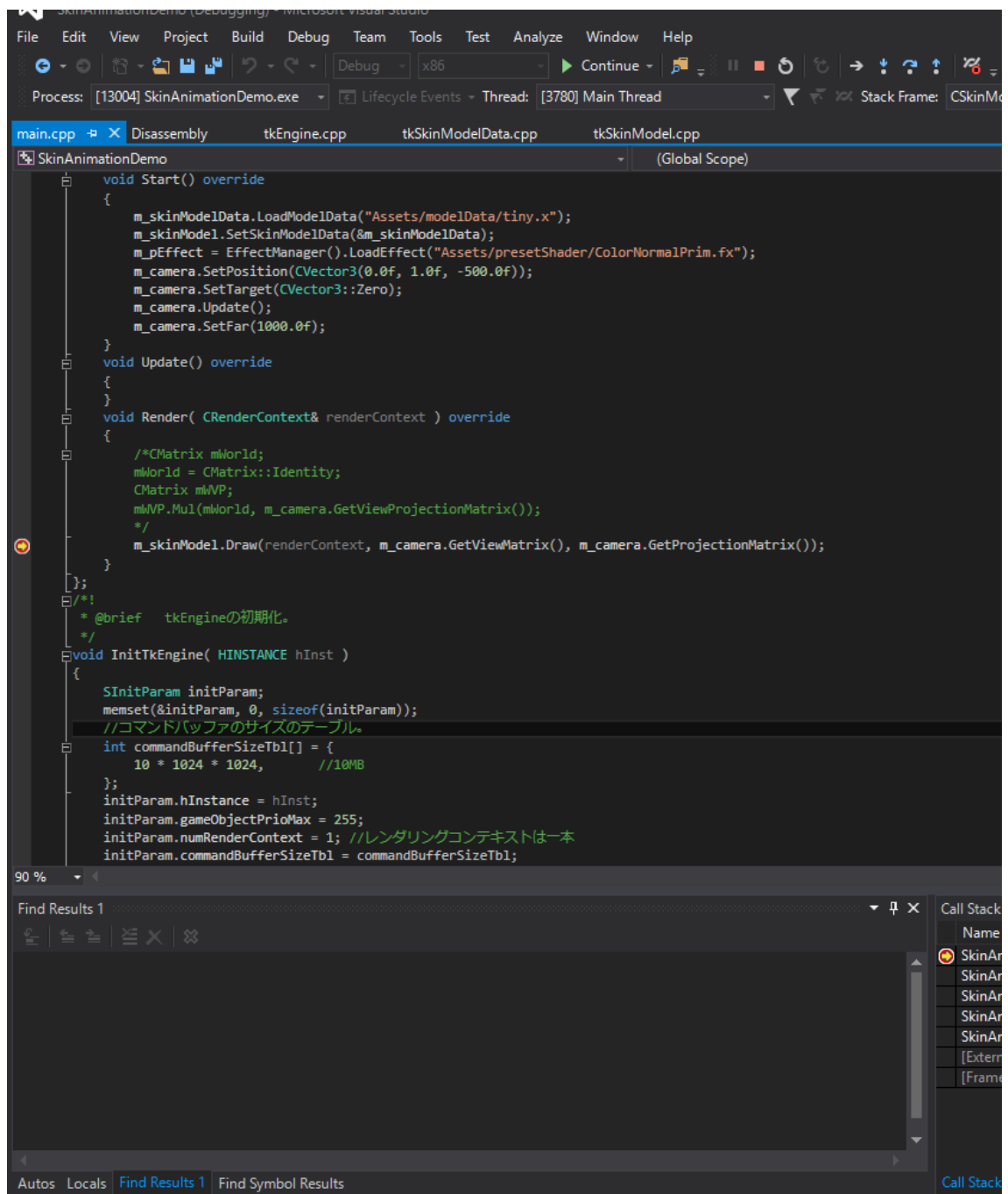
先ほどのステップオーバーとほぼ挙動は同じなのです。違う点はこのコマンドを使用すると関数の中に入っていきける点です。関数の中に入るためステップ”イン”という名前になっています。

3.2.3 ステップアウト(Shift+F11)

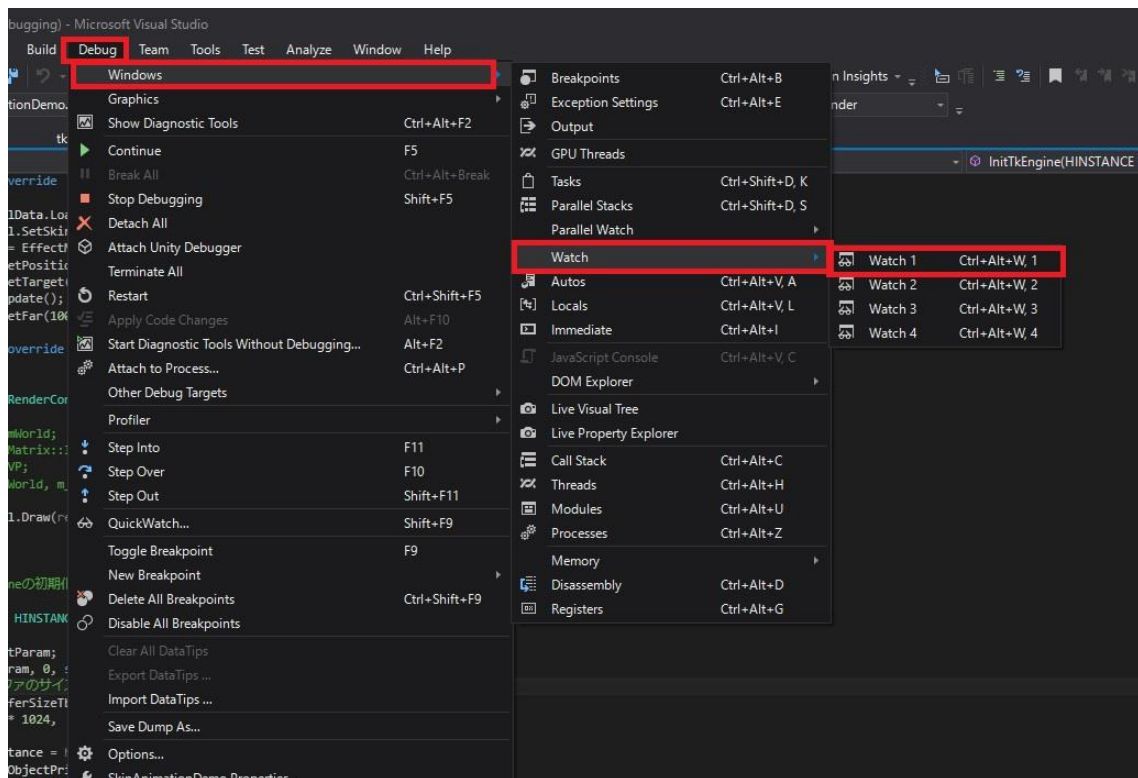
こちらはステップインとは逆の挙動になり、関数から抜ける機能になります。

3.3 ウォッチ

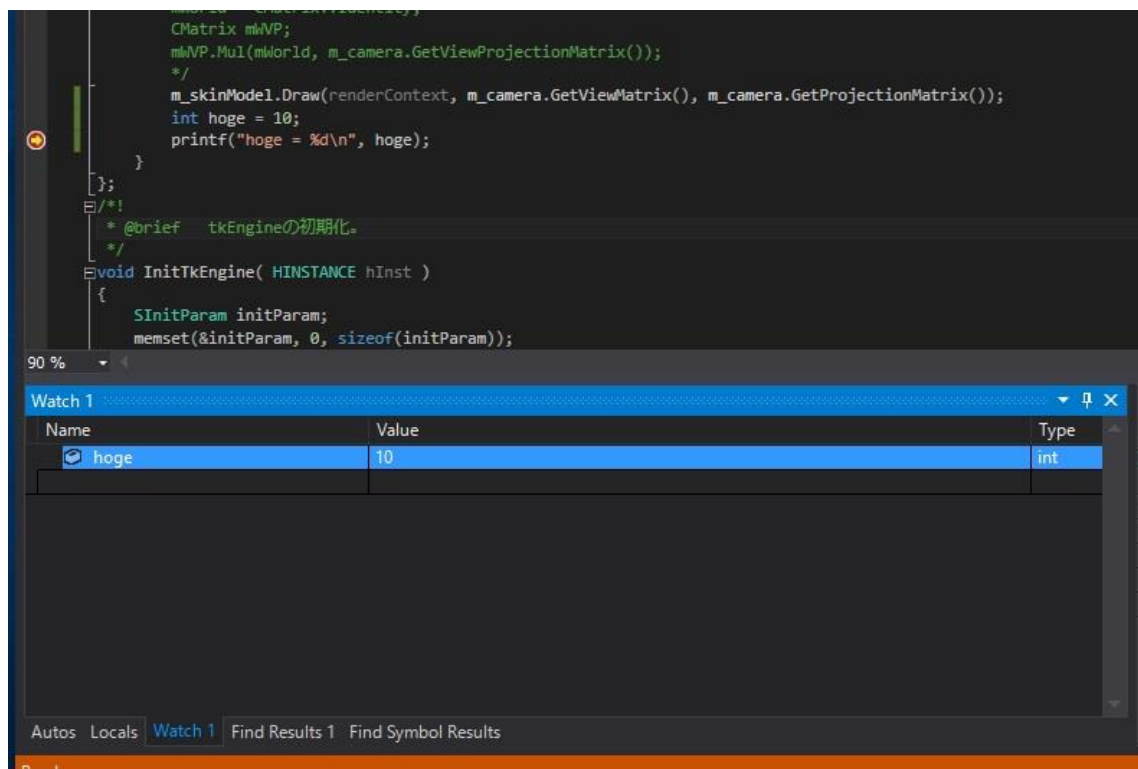
ブレークポイントでプログラムを停止させると、その時の変数の中身を調べたくなることがあるかと思います。その時に使える機能がウォッチという機能です。ではウォッチを使ってみましょう。まず、適当なプログラムにブレークポイントを設置して F5 キーを使って実行してプログラムを下記の図のように停止させてみてください。



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 Debug->Windows->Watch->Watch1 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)。

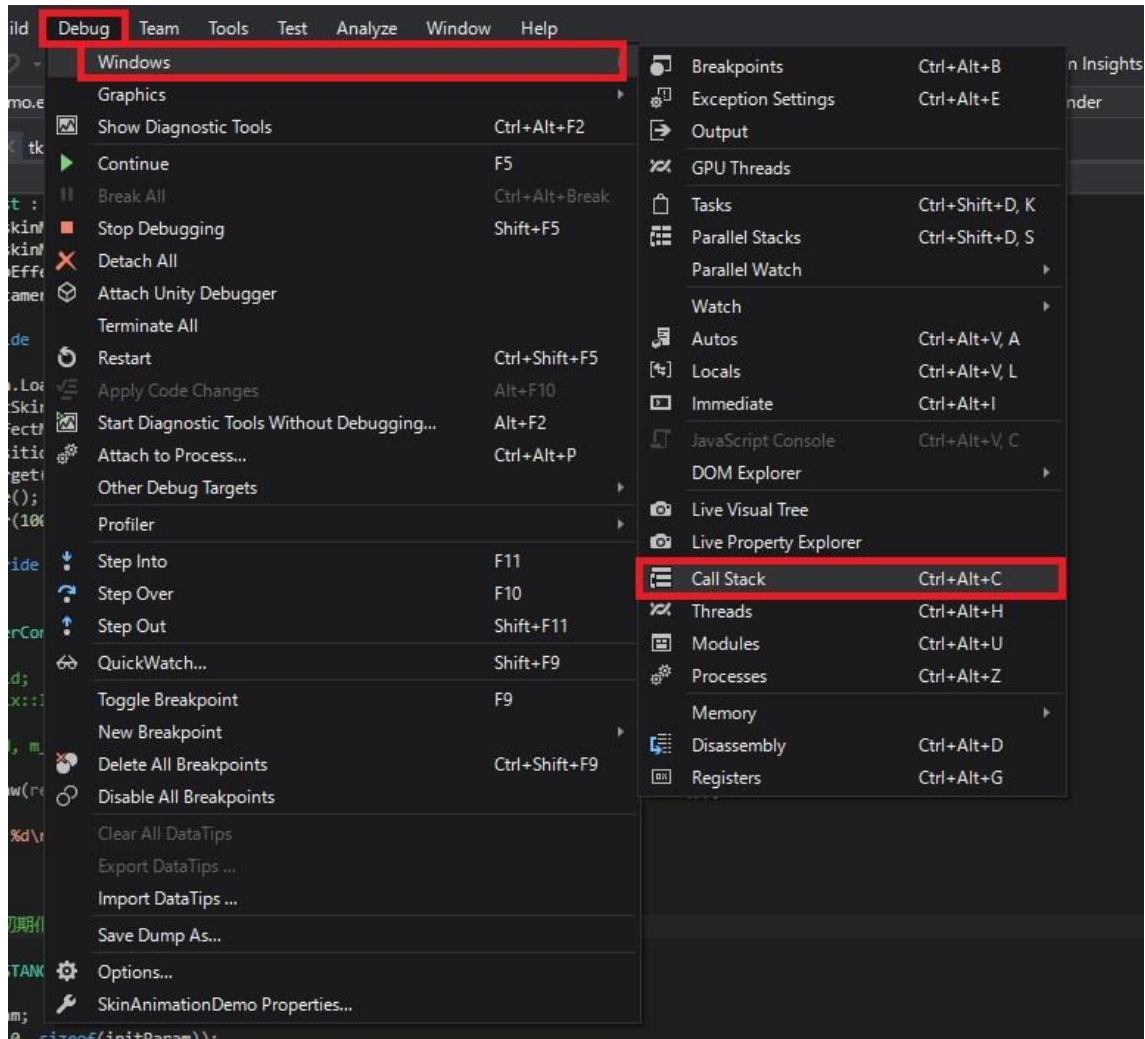


ウォッチが表示されたら、中身を見たい変数名をウォッチウィンドウに追加してみましょう。



3.4 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。ブレー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。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 Debug->Windows->Call Stack(呼び出し履歴)。



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。

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が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、ソースコード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ンス、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

で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 言語を使っている、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のはです。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();  //プレイヤーの移動処理
        enemy.Draw();   //敵の描画処理。
        player.Draw();  //プレイヤーの描画処理。
        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 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。

2.4 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。
}
```

2.5 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。

2.5.1 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。パブリックなのでアクセスできる。
}
```

実はこのコードは2.4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。

2.5.2 private(プライベート)

`private` はクラスのカプセル化(口述します)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。パブ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。

2.6 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つまり、先ほどのメンバ変数に `private` を指定していたことがそれに当たります。ではなぜカプセル化をするのでしょうか？これは 2.1 で話したクラスを使う理由とほぼイコールになります。つまり、可読性、保守性、再利用性を向上させるためです。

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。あなたが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、この不具合が発生させている箇所を探しださなくてはいけません。では `hp` が `private` に指定されていたらどうなるのでしょうか？`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;
}
```

2.6 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{
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()
{
}
//デストラクタの定義。
Player::~Player()
{
}
```

クラスの名前と同じ名前宣言する。戻り値は記述できない。

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。

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。

2.6.1 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]
00DB244B call    Player::Player (0DB10AAh)
        int hp = p.GetHP();
00DB2450 lea     ecx, [p]
00DB2453 call    Player::GetHP (0DB114Ah)
00DB2458 mov     dword ptr [hp], eax

```

コンストラクタ呼び出しが追加されている。

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。

2.6.2 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。
}
//デストラクタの定義。
Player::~Player()
{
    delete skinModel; //new されたので delete を行う。
}
```

後始末

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。

2.6.1 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();
};

//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()
{
}
```

座標の引数を受け取る

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。

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く存在します。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。

車の基底クラスは下記のようなでしょう。

```
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();
};
```

派生クラスでアクセスしたい場合は protected!!!

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いただく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
    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。
};
```

これが仮想関数！！
デフォルトの実装は引きドア

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。これがポリモーフィズムを活用したプログラムになります。ポリモーフィズムとは**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 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。

