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。

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)

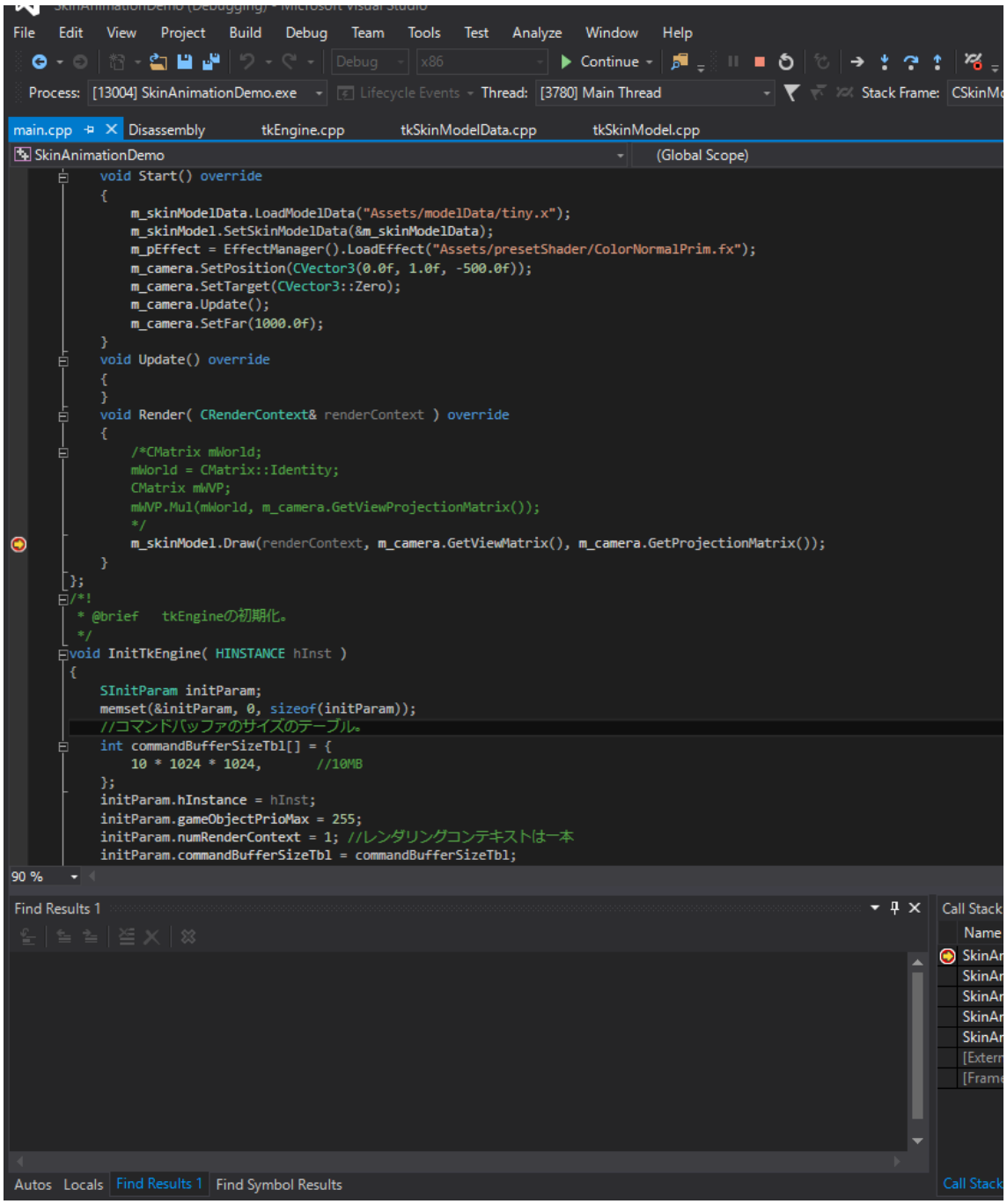
先ほどのステップオーバーとほぼ挙動は同じなのです。違う点はこのコマンドを使用すると関数の中に入っていける点です。関数の中に入るためステップ”イン”という名前になっています。

3.2.3 ステップアウト(Shift+F11)

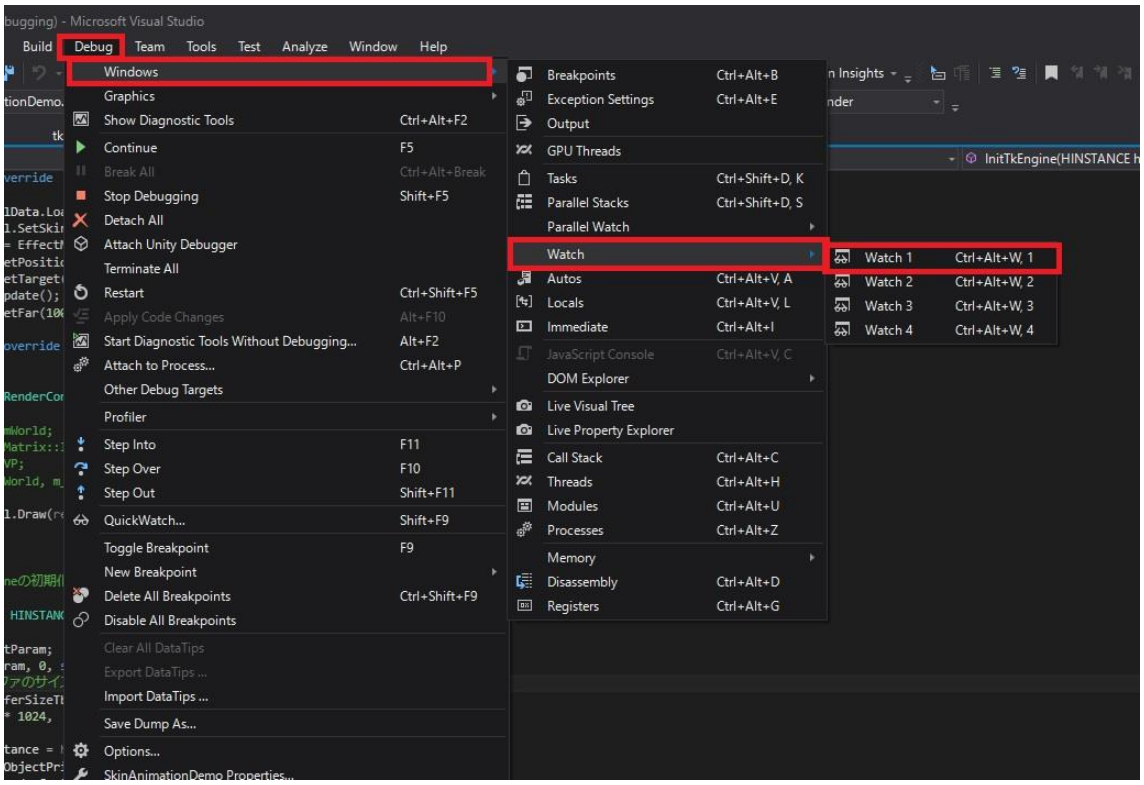
こちらはステップインとは逆の挙動になり、関数から抜ける機能になります。

3.3 ウォッチ

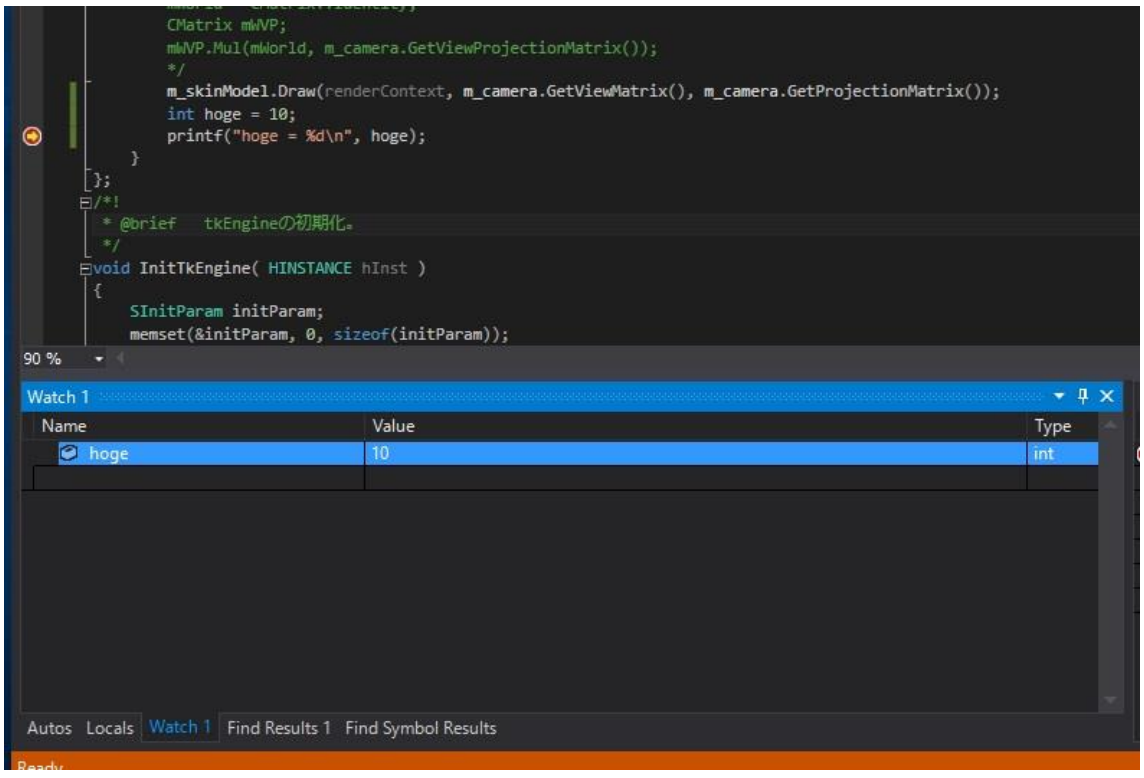
ブレイクポイントでプログラムを停止させると、その時の変数の中身を調べたくなることがあるかと思います。その時に使える機能がウォッチという機能です。ではウォッチを使ってみましょう。まず、適当なプログラムにブレイクポイントを設置して F5 キーを使って実行してプログラムを下記の図のように停止させてみてください。



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)。

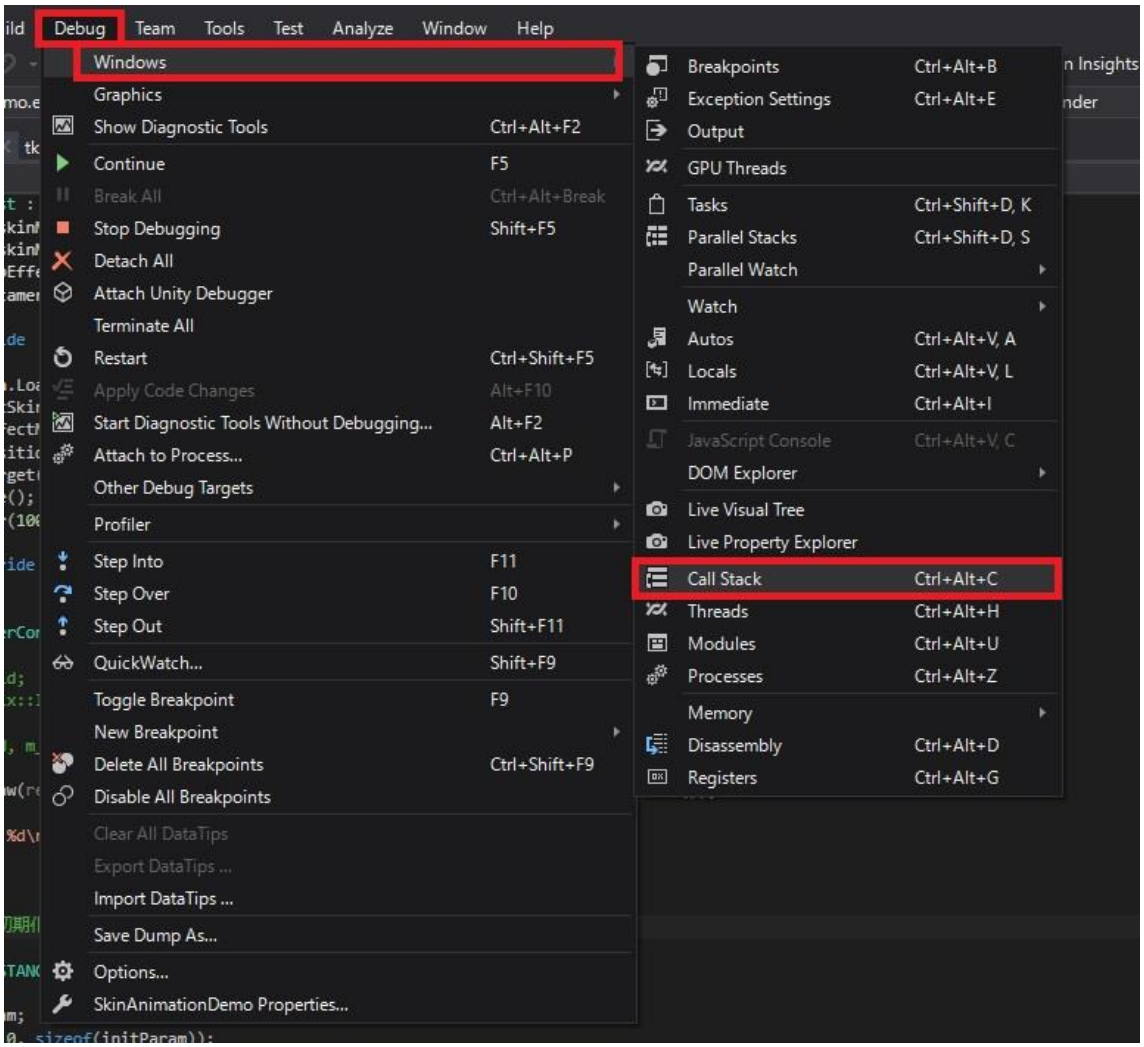


ウォッチが表示されたら、中身を見たい変数名をウォッチウィンドウに追加してみましょう。



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。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 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;
}
```