1　構造体

構造体とは、変数を一つにまとめて定義されるユーザー定義の型になります。

① ユーザー定義の型

ユーザー定義の型とは構造体やクラスのことを言います。C 言語や C++で扱える変数の型は大きく区別すると下記の二つがあります。

組み込み型

int ,short, char, long など

ユーザー定義型

構造体、クラス、enum など

② なぜ構造体を使うのか？

変数を意味のある一つの塊にまとめると可読性が上がり、コードの保守性も向上するため。

③ 構造体の書き方。

では、ゲームでよくあるプレイヤーに関する変数を構造体に変更する方法を記述します。

変更前

```
int main(){
      int playerHp = 100;                    //プレイヤーの HP
      int playerMp = 200;                    //プレイヤーの MP
      int playerLv = 20;            //プレイヤーのレベル
       std::cout << playerHp;
       std::cout << playerMp;
       std::cout << playerLv;
}
```

変更後

```
//Player という新しいユーザー型を定義。
//これが構造体。
struct Player{
   int hp;      //HP
   int mp;     //MP
   int lv;      //レベル
};
int main()
{
      //Player 型の変数の player を定義。
      //定義の仕方は組み込み型と同じ。
      Player player;
      std::cout << player.hp;
      std::cout << player.mp;
      std::cout << player.lv;

}
```

グローバル変数とローカル変数とスタティック変数

C 言語にはグローバル変数とローカル変数とスタティック変数というものがあります。これらの変数の違いは何かというと、変数のスコープが異なります。
(スタティック変数について今回は説明しません。スタティック変数の説明は後日 C++Ⅲで解説予定のファイル分割の時に説明します。)

① スコープ

スコープとはその変数の有効範囲を意味します。例えば下記の playerHp という変数を考えてみましょう。

```
void FuncA()
{
    int playerHp;      //プレイヤーの HP を定義。これはローカル変数。
    playerHp = 100;   //プレイヤーの HP に 100 を設定。
    std::cout << playerHp;
}
void FuncB()
{
    std::cout << playerHp;   //コンパイルエラー！！！playerHp は定義されていない。！！
}
```

playerHp は関数内で定義されているため、ローカル変数と言われる変数になります。ローカル変数とは定義された関数ないでのみ有効なため、playerHp は FuncB で使用することはできず、playerHp は定義されていない！というコンパイルエラーが発生します。

ローカル変数だけではプレイヤーの HP のような、ゲーム中いたるところで使用しそうな変数の扱いに困ってしまいます。そのため、すべての関数でアクセス可能な変数、グローバル変数という変数が存在します。では playerHp をグローバル変数に変更してみましょう。

```
int playerHp;     //プレイヤーの HP を定義。これはグローバル変数。
void FuncA()
{
    playerHp = 100;            //プレイヤーの HP に 100 を設定。
    std::cout << playerHp;   //プレイヤーの HP を表示。
}
void FuncB()
{
```

```
    std::cout << playerHp;  //グローバル変数なのでアクセスできる。
}
```

このように、playerHp はグローバル変数になったため、プログラムのどこからでも参照できるようになりました。

また、ローカル変数は関数を抜けると破棄されてしまうため、関数を抜けた後は値を保持しておくことができません。ゲームプレイ中など永続的に値を保持していたい場合はグローバル変数を使用することになるでしょう。

Tips

実はグローバル変数の乱用は決して褒められたことではありません。極端な話、ローカル変数を一切使わずに、すべてグローバル変数を使用してプログラムを書くことも可能です。しかし、そのようなコードを書いていた場合プログラムが巨大になってきたときに、あなたは必ず後悔します。そのため、シングルトンパターンなどのようにグローバル変数をもう少しマシに使えるようにするためのテクニックが存在します。ただし、今はグローバル変数の扱いに慣れるため、積極的にグローバル変数を使用してかまいません。慣れたころにこの tips の話を思い出してください。

２ 関数

関数とは複数の処理を一つにまとめて記述するもので、保守性、再利用性、拡張性を高めます。ソフトウェアを開発するうえで欠かすことのできない非常に重要な要素となります。

簡単な関数の記述法

では、簡単なサンプルコードを見ながら簡単な関数の記述の仕方を見ていきましょう。

関数化する前のコード

```
int main()
{
   //hoge を 10 回インクリメントして、表示するだけの処理。
   int hoge = 0;
   for(int i = 0; i < 10; i++){
      hoge++;
   }
   std::cout << “hoge=” << hoge << “¥n”;
}
```

では、hoge を 10 回インクリメントして、表示する部分を関数化してみましょう。

```
//hoge を 10 回インクリメントして、表示するだけの関数。
void IncrementAndDispHoge()
{
   int hoge = 0;
   for(int i = 0; i < 10; i++){
      hoge++;
   }
   std::cout << “hoge=” << hoge << “¥n”;
}
//メイン関数。
int main()
{
   //IncrementAndDispHoge を呼び出す。
   IncrementAndDispHoge();
}
```

これが関数化です。関数とは大きく分けて３つの構成要素で成り立っています。

では、次の節からは関数の構成要素を詳しく見ていきましょう。

引数

関数には引数を渡すことができます。引数とは関数に渡すことができるパラメータのことです。

では具体的にプログラムを見てみましょう。先ほどの IncrementAndDispHoge 関数で hoge を 10 回インクリメントするのではなく、任意の数インクリメントするようにしてほしいという仕様変更が来たとします。その変更に応えるために IncrementAndDispHoge を下記のように改造してみましょう。

```
//hoge を incrementCount 回インクリメントして、表示するだけの関数。
void IncrementAndDispHoge( int incrementCount )
{
   int hoge = 0;
   for(int i = 0; i < incrementCount; i++){
      hoge++;
   }
   std::cout << "hoge=" << hoge << "¥n";
}
//メイン関数。
int main()
{
   IncrementAndDispHoge( 10 );     //hoge を 10 回インクリメントして表示する。
   IncrementAndDispHoge( 6 );      //hoge を 6 回インクリメントして表示する。
   IncrementAndDispHoge( 4 );      //hoge を 4 回インクリメントして表示する。
   IncrementAndDispHoge( 1000 ); //hoge を 1000 回インクリメントして表示する。
}
```

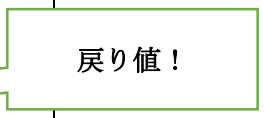
これが引数！！！

これが引数です。IncrementAndDispHoge に引数を渡すようにしただけで、関数というものの有効性が少し見えてきたのではないかと思います。

戻り値

戻り値とは関数が返してくる結果のことを言います。では、具体的なコードを見てみましょう。

```
//平均点を計算する関数。
float CalcAvg( int score0, int score1, int score2, int score3 )
{
	float value = 0;
	value += score0;
	value += score1;
	value += score2;
	value += score3;
	value /= 4;
	return value;		//計算した平均点を返す。
}
int main()
{
	//成績
	int score[4] = {60, 40, 20, 30};
	float avg = CalcAvg(score[0], score[1], score[2], score[3]);
	std::cout << "平均点=" << avg << "¥n";
}
```



これが戻り値と言われるものです。Calc 関数の中で計算した結果を return 文を使用して返しています。

関数を使用するメリット

　では関数を使用するメリットを具体的なプログラムで見ていきましょう。

４クラスの平均点を求めて表示するプログラムを関数化していない場合。

```
int main()
{
	//A クラスの平均点を求める。
	int AclassScore[3];
	for (int i = 0; i < 3; i++) {
		std::cout << "A クラスの成績を入力してください。";
		std::cin >> AclassScore[i] ;
		std::cout << "¥n";
	}
	//平均を求める。
	int totalScore = 0;
	for (int i = 0; i < 2; i++) {
		totalScore += AclassScore[i];
	}
	std::cout << "A クラスの平均点は" << totalScore / 2 << "です¥n¥n";


	//B クラスの平均点を求める。
	totalScore = 0;
	int BClassscore[2];
	for (int i = 0; i < 2; i++) {
		std::cout << "B クラスの成績を入力してください。";
		std::cin >> BClassscore[i];
		std::cout << "¥n";
	}
	//平均を求める。
	totalScore = 0;
	for (int i = 0; i < 1; i++) {
		totalScore += BClassscore[i];
	}
	std::cout << "B クラスの平均点は" << totalScore / 1 << "です¥n¥n";

	//C クラスの平均点を求める。
	int CClassscore[3];
	for (int i = 0; i < 3; i++) {
		std::cout << "C クラスの成績を入力してください。";
		std::cin >> CClassscore[i];
		std::cout << "¥n";
	}
	//平均を求める。
	totalScore = 0;
	for (int i = 0; i <2; i++) {
		totalScore += CClassscore[i];
	}
	std::cout << "C クラスの平均点は" << totalScore / 2 << "です¥n¥n";

	//D クラスの平均点を求める。
	int DClassscore[4];
	for (int i = 0; i < 4; i++) {
		std::cout << "D クラスの国語の成績を入力してください。";
```

```
                std::cin >> DClassscore[i];
                std::cout << "¥n";
        }
        //平均を求める。
        totalScore = 0;
        for (int i = 0; i < 3; i++) {
                totalScore += DClassscore[i];
        }
        std::cout << "D クラスの平均点は" << totalScore / 3 << "です¥n¥n";
    return 0;
}
```

このプログラムにはプログラムの間違いが存在していて。このバグを修正するためには８箇所の修正が必要になり、また見つけるのも困難です。

では平均を求める処理と表示する処理を関数化したプログラムを見てみましょう。

```
//クラスの成績の入力と表示を行う関数。
//className  クラス名
//numStudent  生徒の数。
void InputAndDispClassScore( const char* className, int numStudent)
{
    int classScore[256];
    for (int i = 0; i < numStudent; i++) {
        std::cout << className;
        std::cout << "の成績を入力してください。";
        std::cin >> classScore[i];
        std::cout << "¥n";
    }
    //平均を求める。
    int totalScore = 0;
    for (int i = 0; i < numStudent-1; i++) {
        totalScore += classScore[i];
    }
    std::cout << className << "の平均点は" << totalScore / (numStudent-1) << "です
¥n¥n";
}
int main()
{
    //A クラスの成績入力と表示。
    InputAndDispClassScore("A クラス", 3);
    //B クラスの成績入力と表示。
    InputAndDispClassScore("B クラス", 2);
    //C クラスの平均点を求める。
    InputAndDispClassScore("C クラス", 3);
    //D クラスの平均点を求める。
    InputAndDispClassScore("D クラス", 4);
    return 0;
}
```

こちらのプログラムにも先ほどのコードと同様の不具合が存在しています。しかしこちらは２か所の修正だけでバグが解消します。このように処理をまとめて、再利用性を高めることにより、不具合の修正や仕様の拡張が容易に行えるようになります。

4 ポインタ

ポインタはC言語、C++の大きな壁の一つと言われています。ポインタを理解するということはメモリを理解するということとほぼ同じ意味になります。

4.1 メモリ

ポインタを理解するためにはメモリの理解が必須になりますので、そもそもメモリとは何なのか考えていきましょう。

メモリと言われると皆さん下記のようなキーワードを思い浮かべるのではないでしょうか。

- USBメモリ
- ハードディスク
- SDカード
- PlayStationやPlayStation2のメモリーカード
- メインメモリ

この他にも色々と思い浮かぶと思いますが、プログラマがメモリというと基本的にはメインメモリのことを差します。ではメインメモリとその他のメモリの違いとはなんでしょうか？

最も大きな違いは揮発性メモリかどうか、ということです。

4.2 揮発性メモリと不揮発性メモリ

揮発性メモリとは一体なんなのか？これは電力を供給しないと記録している内容が失われてしまうメモリのことを言います。例えばハードディスクはパソコンの電源を切ってもデータが失われることがありません。そのため、ハードディスクは不揮発性のメモリということになります。同じようにUSBメモリ、SDカード、PlayStationのメモリーカードも不揮発性のメモリです。一方メインメモリはPCの電源を落とすとメモリの内容はすべて失われてしまいます。プログラムで使用されるメモリは基本的にメインメモリとなります。もちろんゲームの進行状況などは不揮発性のメモリに記録しておかないと、電源を落とすたびに最初から始めることになります、そのためハードディスクに書き込みを行うこともありますが、基本的にメモリというとメインメモリのことを差します。

4.3 メインメモリ(主記憶)

ではメインメモリについて見ていきましょう。皆さんのPCはメインメモリが8GBほどあると思いますが、ではプログラムがそのメモリをすべて使用できるのか？というとそういうわけではありません。例えばあなたの作成したプログラムがすべてのメモリを独占してしまうと、同時に起動しているVisualStudioやブラウザなどと言った他のプログラムが

強制終了してしまいます。そのため、プログラムが起動すると OS がそのプログラムが使用できるメモリを割り当てます。

例えば、512 バイトのメモリが割り当てられたケースを考えてみましょう。メモリのイメージは下記のようになります。

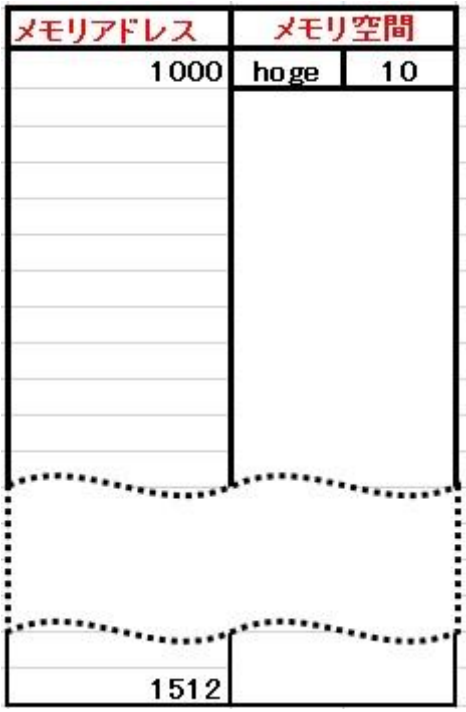
メモリアドレス	メモリ空間
1000	未使用
1512	

では、このようなメモリ空間が割り当てられたとして、具体的にプログラムを書いてみてメモリがどのようになるか見ていきましょう。

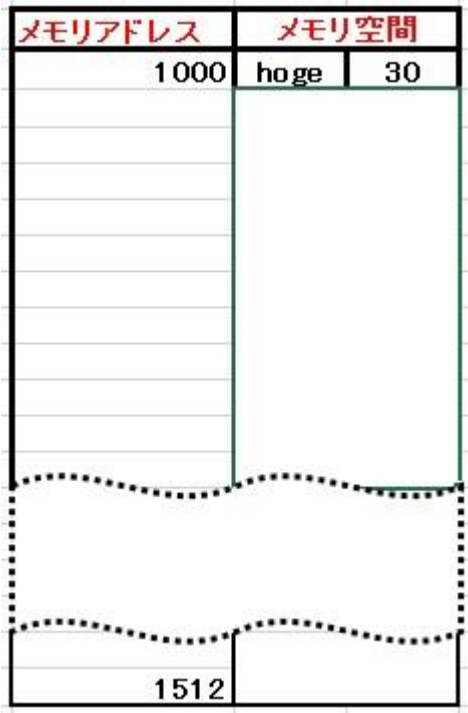
```
int main()
{
    int hoge = 10;              //変数は値を記録できる。つまりメモリに領域が確保される。    ①
    hoge += 20;                                                                          ②
    int hogehoge = 20;          //変数は値を記録できる。つまりメモリに領域が確保される。    ③
    std::cout << hoge << hogehoge;
    return 0;
}
```

このプログラムを実行した場合、メモリの内容がどうなるか見ていきましょう。

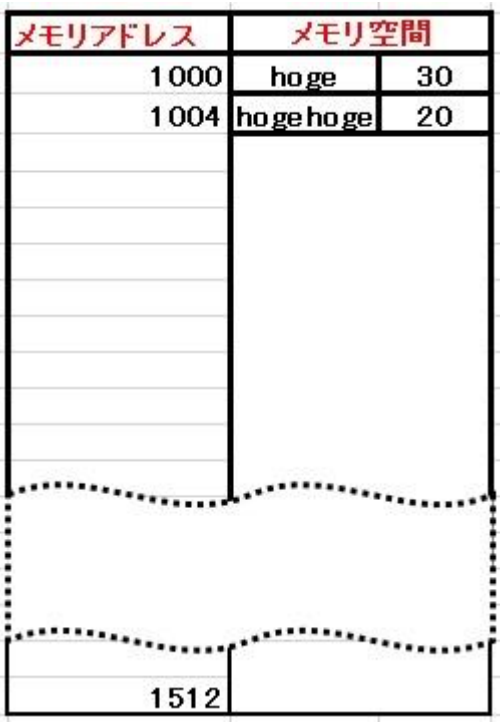
① を実行したら hoge の領域が使用されて、そこに 10 という値が記録される。



② を実行すると hoge の値が+20 される。



③ を実行すると hogehoge の領域が使用されて、そこに 20 が記録される。



メモリのイメージはこのようになります。

4.4 ポインタ型

さて、では本題のポインタについて見ていきましょう。まず、ポインタについて多くの書籍や、多くの人が誤解を招く下記のような言い方をします。(これは私もします。)

ポインタとはアドレスを記録する変数である。

実はこの説明は正しい説明ではありません。この説明を聞くとまるでポインタが変数であるかのように思えると思いますが、必ずしもそうではありません。ここでは正しい説明をしようと思います。

int* hoge;

上記のような変数がある場合、hoge の正しい説明は下記のようになります。

int のポインタ型の変数

分かりますでしょうか？ここで理解してほしいのは **int*** というのは**ポインタ型**であるということです。そして hoge というのは**ポインタ型の変数**であるということなのです。

色々な書籍や、プログラマがポインタというキーワードを使うときは、文脈によって、ポインタ型の変数であったり、ポインタ型のことであったりします。では下記に型の一覧を上げてみましょう。

Int	整数型	整数の値を記録できる。
Float	浮動小数点型	小数点の値を記録できる。
Char	文字型	文字を記録できる。
int*	**int のポインタ型**	**int 型の変数のアドレスを記録できる。**
float*	**float のポインタ型**	**float 型の変数のアドレスを記録できる。**
char*	**char のポインタ型**	**char 型の変数のアドレスを記録できる。**

つまり、int*とか float*とか char*はポインタ型と言われるもので、int や float などと同じ型でしかないということです。

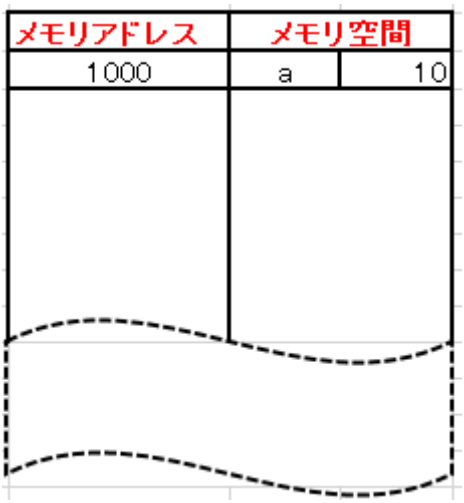
4.5 ポインタ型の変数

ではポインタ型の変数について見ていきましょう。前節で説明したように、ポインタ型の変数というのは、変数のアドレスを記録できる変数でした。そして、変数のアドレスというのは&を用いて取得することができます。

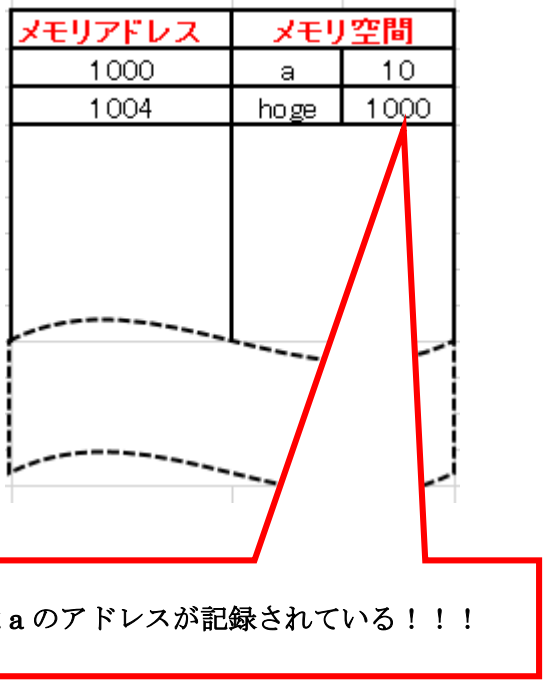
```
int a = 20;          //int 型の変数の a に 20 という値を記録する。
int* hoge = &a;  //a のアドレスを hoge に記録する。
```

ではこのコードが実行されると、メモリ空間上でどのようなことが起きているのか見てみましょう。

① int 型の変数 a の領域が用意されて 10 が記録される。



② int のポインタ型の変数 hoge の領域が用意されて a のアドレスが記録される。



これがポインタ型の変数にアドレスを代入したときのメモリ上の動作になります。
また、ポインタ型の変数に*をつけるとアドレスの差している先の値を変更できるということも習ったかと思います。

```
*hoge = 20;  //a の値が 20 に変わる！！！
```

ポインタ型の変数というのはこのような使い方をします。

4.6 関数の引数でのポインタ型の変数の使用

プログラムを書いていると、関数の中で行った計算などを呼び出し元の変数に格納したい場合が多々あります。例えば下記のようなコード。

```
//テストの成績の配列。グローバル変数。
int score[5] = {
   10, 40, 30, 20, 50
};
//平均点と総得点を計算する関数。
//引数の avg に平均点、total に総得点を入れるつもりなのだが・・・
void CalcAverageAndTotal( int avg, int total )
{
      avg = 0;
      total = 0;
      for(int i = 0; i < 5; i++){
         total += score[i];
      }
      avg = total / 5;
}
int main()
{
      int avg, total;
      avg = 0;
      total = 0;
      CalcAverageAndTotal(avg, total);   //avg と total に平均点と総得点を入れるつもり・・・
      std::cout << "平均点" << avg << "¥n";       0 が出力される！！！
      std::cout << "総得点" << total << "¥n"       0 が出力される！！！
      return 0;
}
```

プログラムを書いた人は CalcAverageAndTotal 関数を呼ぶことで、引数に平均点と総得点が代入されることを期待してプログラムを記述していますが、この関数の引数は値渡しになっていて、期待通りの結果を得ることはできません。

4.6.1 値渡しと参照渡し

下記のような関数の引数を値渡しといいます。

```
void CalcAverageAndTotal( int avg, int total )
{
    略
}
int main()
{
    int avg, total;
    avg = 0;
    total = 0;
    CalcAverageAndTotal(avg, total); //値渡し
}
```

そして下記のような関数の引数を参照渡しといいます。

```
void CalcAverageAndTotal( int* avg, int* total )
{
    略
}
int main()
{
    int avg, total;
    avg = 0;
    total = 0;
    CalcAverageAndTotal(&avg, &total); //参照渡し
}
```

値渡しの場合、関数には**変数のコピー**が渡されます。そのため、コピーをいくら編集しようがオリジナルに影響は一切ありません。ファイルのコピーを考えてみて下さい。コピーしたファイルをいくら編集しようとも、オリジナルには影響がないはずです。

参照渡しの場合、関数には変数のアドレスが渡されます。こちらはオリジナルの変数の中身を変えることができます。こちらはファイルのショートカットリンクのようなものです。ショートカットリンクをダブルクリックしたファイルを編集するとオリジナルのファイルも編集できます。

では CalcAverateAndTotal の引数を参照渡しに変更してみましょう。

```
//テストの成績の配列。グローバル変数。
int score[5] = {
   10, 40, 30, 20, 50
};
//平均点と総得点を計算する関数。
void CalcAverageAndTotal( int* avg, int* total )
{
    *avg = 0;
    *total = 0;
    for(int i = 0; i < 5; i++){
       *total += score[i];
    }
    *avg = *total / 5;
}
int main()
{
    int avg, total;
    avg = 0;
    total = 0;
    CalcAverageAndTotal(&avg, &total);
    std::cout << "平均点" << avg << "¥n";      期待通り出力される！！！
    std::cout << "総得点" << total << "¥n"      期待通り出力される！！！
    return 0;
}
```

4.7 ポインタと配列

ポインタと配列には密接な関係があります。配列は下記のようにポインタを介してアクセスすることができます。

```
int hogeArray[4] = { 0, 1, 2, 3 };
int* pHoge = hogeArray;　//hogeArray の先頭アドレスを pHoge に代入！
pHoge[0] = 10;　//hogeArray[0]の値が変わる！！！
pHoge[1] = 20;　//hogeArray[1]の値が変わる！！！
pHoge[2] = 30;　//hogeArray[2]の値が変わる！！！
pHoge[3] = 40;　//hogeArray[3]の値が変わる！！！
```

あたかもポインタが配列であるかのように扱えていますね。

実は hogeArray は添え字演算子の[]を使わずに記述した場合、ポインタ型の変数と全く同じになります。hogeArray は配列の先頭アドレスが入っているポインタ変数となります。

ですので、下記のようなコードを記述することもできます。

```
int hogeArray[4] = { 0, 1, 2, 3 };
*( hogeArray) = 10;　　//hogeArray[0]の値が変わる！！！
*( hogeArray +1) = 20;　//hogeArray[1]の値が変わる！！！
*( hogeArray +2) = 30;　//hogeArray[2]の値が変わる！！！
*( hogeArray *3) = 40;　//hogeArray[3]の値が変わる！！！
```

まるでポインタ変数のようにアクセスできています。これは hogeArray が添え字演算子なしで記述されると正真正銘ポインタとして扱われるためです。

つまり、配列を関数の引数に渡したい場合は下記のようなコードを書けばいいことが分かります。

```
void FuncDisp(int* pArray)
{
    std::cout << pArray[0] << "¥n";　　//10 と表示される。
    std::cout << pArray[1] << "¥n";　　//20 と表示される。
}
int main()
{
    int array[2] = { 10, 20 };
    FuncDisp(array);
    return 0;
}
```

tips

配列の添え字はシンタックスシュガーと言われているものです。この節で説明した通り、C 言語において配列というのはポインタとほぼ同じ挙動をします。

そのため、配列の要素はすでに説明したように下記のようにアクセスすることも可能です。

```
int array[2] { 10, 20 };
*(array + 1) = 20;   //array[1]の値が変わる。
```

しかし、このようなコードを書くのは面倒くさくて分かりにくいですね。そのため、プログラマに分かりやすいように下記のように書けるようになっているのです。

```
array[1] = 20;
```

このようなプログラマが分かりやすくするためだけに導入された機能のことを、人間にとって甘く(とっつきやすく)するという意味でシンタックスシュガーと呼ぶことがあります。

4.6 ポインタ型の変数を使用する理由

恐らくここまでの説明だけでは、ポインタ型の変数の挙動は理解できたが、何故こんなものを使うのかよく分からないのではないかと思います。しかし、ポインタ型の変数というものが存在しないとプログラムを書くのは非常に困難になります。

ポインタ型の変数を使用する理由は、それはもう少し規模の大きいプログラムを実装するようにならないと理解することは難しいはずです。(恐らく 1 年の最後に行う個人製作あたりの規模は必要)。ですので、今は使う理由はよく分からないが、とにかく沢山プログラムを書いて、どんどんポインタ型の変数を使用してみて下さい。そして使い方に慣れていってください。

5 ファイル分割

それなりにプログラムの規模が大きくなってくると、ファイルを分割した方が開発しやすくなってきます。C++ですと、1 ファイル 1 クラスが基準になると言われています(厳守する必要はありません)。しかし、C++でのファイル分割はそこまで簡単ではなく、少々やっかいな問題がいくつかあります。今回はそれを見ていきましょう。

ではまずコードが分かれていないプログラムを見てみましょう。

```
//main.cpp
#include <iostream>
//クラス定義
class Player{
   int hp;
public:
   //HP を取得。
   int GetHP();
   //HP を設定。
   void SetHP(int hp);
};
//HP を取得。
int Player::GetHP()
{
    return hp;
}
//HP を設定。
void Player::SetHP(int _hp)
{
    hp = _hp;
}
int main()
{
   Player pl;
   pl.SetHP(100);
   std::cout << pl.GetHP();
   return 0;
}
```

では、このプログラムをファイル分割してみましょう。Player.h と Player.cpp を追加します。

まず、ヘッダーファイルを見てみましょう。

```
//Player.h
#pragma once //必ず書く！！！
//クラス定義
class Player{
   int hp;    //HP
public:
   //HP を取得。
   int GetHP();
   //HP を設定。
   void SetHP(int _hp);
};
```

クラス定義はヘッダーファイルに記述しましょう。そしてヘッダーファイルの先頭には必ず#pragma once を記述しましょう。理由は後述します。

では続いて player.cpp を見ていきましょう。

```
//Player.cpp
#include “Player.h”

//HP を取得する関数の定義。
int Player::GetHP()
{
   return hp;
}
//HP を設定する関数の定義。
void Player::SetHP(int _hp)
{
    hp = _hp;
}
```

では最後に main.cpp のソースを見てみましょう。

```
//main.cpp
#include <iostream>
int main()
{
   Player pl;          //コンパイルエラー！！
   pl.SetHP(100);
   std::cout << pl.GetHP();
   return 0;
}
```

さて、上記のコードでは main.cpp で Player.h がインクルードされていないため、コンパイルエラーが発生します。何故コンパイルエラーが出たのかを詳しく見ていきましょう。

コンパイラというのはソースファイル単位でコンパイルを実行します。つまり main.cpp をコンパイルしている時は Player.cpp や Player.h に記述されている内容をコンパイラは一切知りません。そのため、main.cpp をコンパイルしているコンパイラは Player というものがどんなメンバ関数があるの、そもそも Player はクラスなのか構造体なのか、など一切分からないわけです。そのため、Player クラスの内容をコンパイラに教えてやる必要があります。それが#include になります。

Player クラスの定義が記述されている Player.h を main.cpp をインクルードすることでコンパイラは Player クラスの内容を知ることができ、コンパイルが実行できるのです。

5.1 #include

では、そもそも#include とはなんなのか詳しく見ていきましょう。

まず、先ほどの main.cpp に Player.h をインクルードしてコンパイルエラーを解決しましょう。

```
//main.cpp
#include <iostream>
#include “Player.h”
int main()
{
   Player pl;
   pl.SetHP(100);
   std::cout << pl.GetHP();
   return 0;
}
```

これでコンパイルエラーは発生しなくなります。

では#include が何をしているのか見てみましょう。

5.1.2 プリプロセッサ

実は VisualStudio などのコンパイラはコンパイルを行う前にソースコードの変形を行います。これがプリプロセスと言われるもので、プリプロセスを行うソフトウェアをプリプロセッサと言います。そして#から始まる構文、#include や#pragma や#ifdef などはプリプロセッサに対する命令になります。

では#include 命令が記述されているとプリプロセッサはどのようにソースコードを変形するのでしょうか？

実はプリプロセッサは#include を見つけると、そこに指定されたファイルの内容をコピーアンドペーストします。

つまり、先ほどの main.cpp はプリプロセッサによって、下記のようにソースを変形されます。

```
//main.cpp
#include <iostream>
//クラス定義
class Player{
   int hp;     //HP
public:
   //HP を取得。
   int GetHP();
   //HP を設定。
   void SetHP(int _hp);
};
int main()
{
   Player pl;
   pl.SetHP(100);
   std::cout << pl.GetHP();
   return 0;
}
```

このように変形されます。これによって、main.cpp をコンパイルしているコンパイラは Player クラスが分かるようになります。