後期課題１

３日で分かるC++言語での基本プログラミング①

■はじめに

１）C++言語でプログラミングするという事

いわゆるオブジェクト指向のプログラミング言語ですが、個々の機能を持ったオブジェクト同士を組み合わせてプログラムを動かす考え方で、高い保守性を持つ事ができてシステムの拡張や移行を容易に行う事ができます。特に大きな規模のシステムや多人数での開発で威力を発揮します。

２）C++言語のバージョン

C++の歴史は古く、前進の「C with Classes」が１９７９年に開発が始まっています。これまでにアップデートを繰り返しており、バージョンにより機能が異なる場合があるので注意が必要です。これからであれば最新版の「C++17」を学びましょう。

規格設定日	C++標準規格	通称
1998年 9月 1日	ISO/IEC 14882:1998	C++98
2003年10月16日	ISO/IEC 14882:2003	C++03
2007年11月15日	ISO/IEC TR 19768:2007	C++TR1
2011年 9月 1日	ISO/IEC 14882:2011	C++11
2014年12月15日	ISO/IEC 14882:2014	C++14
2017年12月	ISO/IEC 14882:2017	C++17

3)C++言語の長所と短所

C言語と違いオブジェクト指向の概念を理解する必要があるため、C言語の延長と考えてしまうと言語構造に戸惑う事もあるかもしれません。書き方がやや複雑になる傾向がありますので「完全に理解する」という考えではなく、長く付き合っていこうと割り切る事が大事です。ポインタなどの概念も当たり前の様に出てきますし、C言語では単純だった処理も手間がかかったり自前で準備したり、かえって面倒じゃない？と思う事も度々あるかもしれませんが上手に受け流していきましょう。

□それでも学びたい

とはいえ、特別な魅力のあるC++言語ですから是非とも使いこなしてみたいものです。とにかく一つのスタイルでゲームを作れる様になれば、自ずとスキルアップも可能です。

これからの人も行き詰った人も、習うより慣れろの精神でとにかくC++言語で一本ゲームを作ってみましょう。

■クラスとは

オブジェクト単位でプログラミングする上で欠かかせないのが「クラス」です。今回のゲーム開発も、基本的にはクラス単位で構築していきます。クラスはC言語でいう構造体とよく似ており、構造体の中に使用できる関数を追加したものだとイメージしてください。これにより、単なるデータの塊から機能を持った塊に生まれ変わる事ができるのです。

尚、クラスの中の変数を「メンバ変数」、関数を「メンバ関数（クラスメソッド）」と言います。

```
// ----- クラス
class クラスの型名
{
  int  x;                  } メンバ変数
  int  y;

  void  Init(void);        } メンバ関数
  void  Update(void);        （クラスメソッド）

}; ※セミコロンあり
```

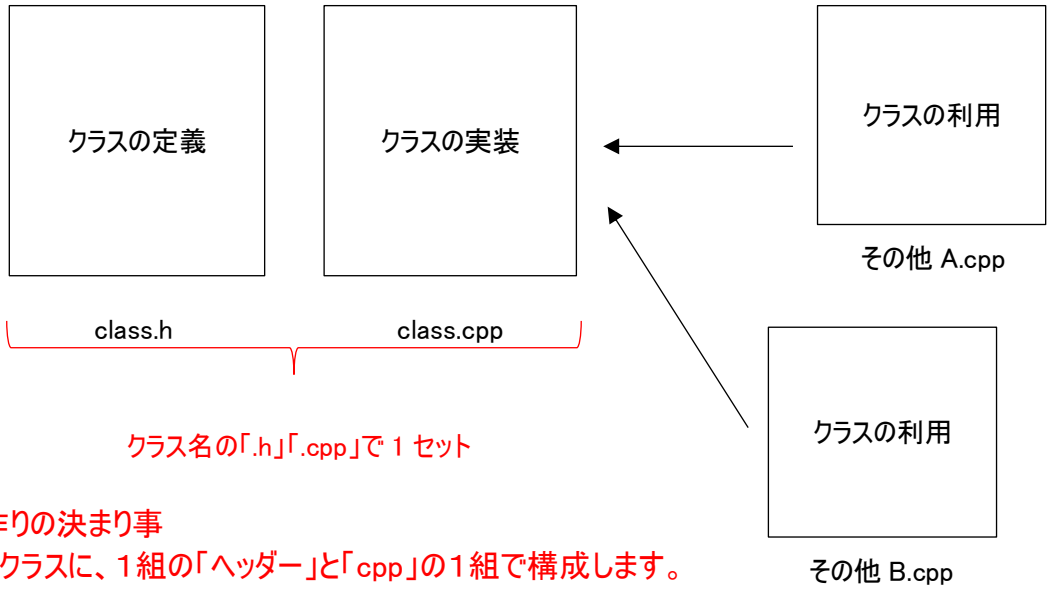
```
// ----- 構造体
struct 構造体の型名
{
  int  x;          } メンバ変数
  int  y;

}; ※セミコロンあり
```

クラスや構造体の記述は「あくまでも定義」なので実体がありません。定義したクラスを実際に使用する為には、クラスの変数を宣言する必要があります。この実体を生成する事を「インスタンス化」「インスタンスの生成」「実体化」と言います。

□クラス利用の手順

基本的に「クラス定義」→「クラスの実装」→「クラスの利用」の3つの段階で利用していきます。

1）クラスの定義・・・・クラス専用のヘッダーファイルでクラス定義を行う。
2）クラスの実装・・・・クラス専用の cpp ファイルに記述する。クラス関数の中身を記述していきます。
3）クラスの利用・・・・クラスオブジェクトを使用したい関数などからインスタンスを生成して利用していきます。

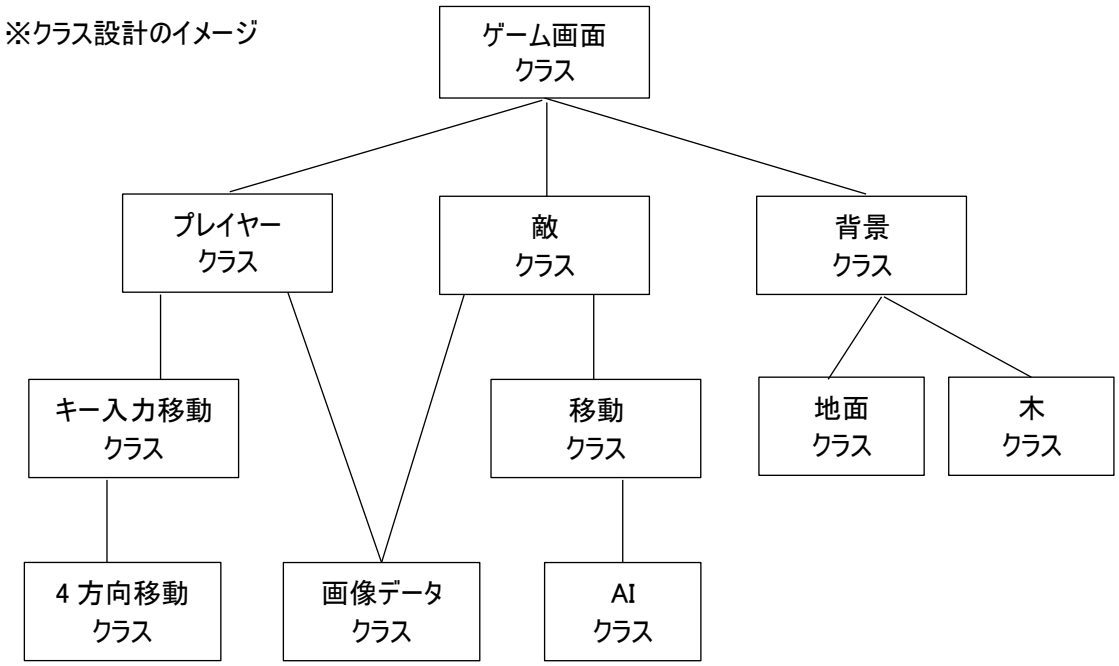


ファイル作りの決まり事
・1つのクラスに、1組の「ヘッダー」と「cpp」の1組で構成します。
・ファイル名は、原則クラス名と同じにします。

□クラスはどの様な単位で作る？

クラスは「個々の機能を持ったオブジェクト」の１つの塊として作成していきます。個々の機能と言っても「ある計算だけを行う」「グラフィックデータを管理する」「タイトル画面を管理する」「プレイヤーそのもの」「ボスキャラそのもの」など、内部機能として動かすものから、プレイヤーなどの主役級の機能を持つものなどまで様々あります。

それらの機能を持ったパーツをどう作ってどう組み合わせるか？がゲーム開発において重要なテーマになります。



□クラス定義をやってみる

実際に3つの手順に基づきクラスの利用をやってみましょう。

1. クラスの定義

player.h

```
class Player
{
public:
    int pos_x;
    int pos_y;

public:
    void Init(void);
    void Update(int, int);
};
```

public:はアクセス指定子と言います。
後で説明がでてきますので、今回は無視してください。

メンバ変数

メンバ関数

「player.h」に、作成するクラスの定義を行います。C 言語でいう構造体の型宣言と関数プロトタイプを、クラス定義時に同時に行うイメージです。

2. クラスの実装

player.cpp

```
#include “DxLib.h”
#include “player.h”

void Player::Init()    // Player クラスの Init 関数の中身
{
   pos_x = 0;
   pos_y = 0;
}

void Player::Update(int a, int b)    // Player クラスの Update 関数の中身
{
    pos_x = a;
    pos_y = b;
}
```

「player.cpp」に関数の実態を記述します。記述時に、定義ファイルの“player.h“をインクルードしておきます。
ここでは実際に実行される部分になりますので、例えば DxLib の関数を使用する場合などは”DxLib.h“もインクルードしておきます。

```
戻り値の型　クラス名 : : 関数名 ( 引数 )
{
    // 処理
}
```

3. クラスの利用（オブジェクトの生成と利用）

main.cpp

```
// ----- player オブジェクトを利用する処理
#include “DxLib.h”
#include “player.h”

void main()
{
   Player  p_obj;    // Player をインスタンス化して p_obj に持っておく

   p_obj.pos_x = 100;    // Player クラス内の pos_x に数値を代入
   p_obj.pos_y = 50;     // Player クラス内の pos_y に数値を代入


   p_obj.Init();    // Player クラス内の Init 関数を実行


   p_obj.Update(500, 100);    // Player クラス内の Update 関数を実行


}
```

実際の利用については、オブジェクトの生成が必要です。使用する関数「例：main.cpp」で実体を定義します。この事を「インスタンス」と言います。インスタンスは型名と変数の名前を記述しますが、この手順は構造体と同じです。

```
Player  p_obj;
```

```
型名　実体化したものを管理する変数名;
```

インスタンス化したオブジェクトのメンバ変数とメンバ関数へのアクセス方法は、構造体と同じで「ドット演算子（.）」を使います。※ポインタを経由する場合は「アロー演算子（－＞）」を使用します(別途記載)。

```
p_obj.pos_x = 100;          // p_obj のメンバー変数 pos_x に 100 を代入する

p_obj.Init();               // p_obj の Init 関数を実行する

p_obj.Update(500, 100);     // p_obj の Update 関数に値渡しで実行する
```

□クラス定義時に処理も書く

クラスメンバ関数（クラスメソッド）の定義は、クラスの定義内に宣言を記述しますが、同時に処理内容も実装する事が可能です。

```
class クラス名
{
public:

   int x;
   int y;

   void Init(){
       x = 0;
       y = 0;
   }

   void Update(int a, int b){
       x = a;
       y = b;
   }

};
```

実際、この書き方でも大丈夫です。
まずは、一般的な書き方として「宣言はヘッダー側で」「実装は cpp 側で」を徹底していきましょう。

□宣言はヘッダー、実装は cpp

先ほどの宣言と実装は以下の様に記述します。

ヘッダー側

```
class クラス名
{
public:

   int x;
   int y;

   void Init();

   void Update(int, int);
};
```

cpp 側

```
#include “クラス名.h”

void クラス名::Init()
{
   x = 0;
   y = 0;
}

void クラス名 :Update(int a, int b)
{
    x = a;
    y = b;
}
```

■複数のインスタンス(クラスの実態)

1. 一つのクラスを使って複数のインスタンスを作る事で、機能は同じで全く別のオブジェクトを作る事ができます。キャラクタ1体を作っておいて、それを元に 100 体のザコキャラを作る様なイメージです。構造体でも同様な事ができたと思いますが、それと同じ事です。

```
Player  p_objA;         // インスタンス:p_objA
Player  p_objB;         // インスタンス:p_objB
Player  p_obj[100];     // インスタンス:p_obj[100 体]
```

インスタンスが違えば、それぞれの「メンバ変数」「メンバ関数」は全く別物として扱われます。

■アクセス指定子

1. アクセス指定子とは、メンバ変数やメンバ関数にアクセスする範囲を指定する事ができるもので、クラスの宣言時にそれぞれの属性として指定する事ができます。

アクセス指定子	意味
public	全ての範囲からアクセス(読み込み・書き込み)可能
private	同一クラス、インスタンス内でのみアクセス可能
protected	同一クラス、インスタンスに加え、継承されたサブクラスでもアクセス可能

2. まずは「public」と「private」の属性について動きを見てみます。(protected は後ほど説明します)

player.h

```
class Player
{
public:
    int pos_x;    // public のメンバ変数
    int pos_y;
private:
    int x;        // private のメンバ変数
    int y;

public:
    void Set(int, int);    // public のメンバ関数
private:
    void Init(void);       // private のメンバ関数
};
```

player.cpp

```
#include  “player.h”

void Player::Set(int a, int b)
{
    x = a;    // x は private だが自分自身の定義された変数なのでアクセスできる
    y = b;    // y は private だが自分自身の定義された変数なのでアクセスできる
};
```

3. インスタンス化して main.cpp からアクセスしてみましょう。

main.cpp

```
// ----- player オブジェクトを利用する処理
#include  “player.h”

void main()
{
   Player  p_obj;     // Player をインスタンス化して p_obj に持っておく

   // ①
   p_obj.pos_x = 100;     // pos_x は public なので main.cpp からアクセス OK。
   p_obj.pos_y = 50;      // pos_y は public なので main.cpp からアクセス OK。

   // ②
   p_obj.x ※エラー      // x は private なので main.cpp からアクセス不可。
   p_obj.y ※エラー      // y は private なので main.cpp からアクセス不可。

   // ③
   p_obj.Set(10, 20);     // Set 関数は public なので main.cpp からアクセス OK。

   p_obj.Init() ※エラー   // Init 関数は private なので main.cpp からアクセス不可。
}
```

例にある様に、public なメンバ変数やメンバ関数は main.cpp からアクセスができます。 ※エラーの行は private なので実体（player.cpp）以外の main.cpp ではアクセスできないのでビルドエラーとなります。

なぜこの様にするかというと、オブジェクト指向ではそのクラスのメンバ関数しかアクセスできない様に、可能な限り制限をかけていきます。その様な手法をカプセル化（隠蔽）と言います。単純に public を減らして private を増やすイメージです。

ただ、private の変数の値を入れたり、値を見たりしたい場合もあります。その時はどうするのか？その時は、private の変数の値を設定したり取得したりする public なメンバ関数を利用してアクセスします。それをアクセスメソッドと呼びますが、「ゲッター」「ゲット関数」「セッター」「セット関数」と読んだりもします。

4. 図で表すと下記の様になります。

main.cpp

```
// ----- メイン関数
void main()
{
  Player p_obj;
  // playerｵﾌﾞｼﾞｪｸﾄへのｱｸｾｽ
  // ①
  p_obj.pos_x = 100;
  p_obj.pos_y =  50;


  // ②
  p_obj.x = 0;
  p_obj.y = 0;


  // ③
  p_obj.Set(10, 20);
      ※x, yにｱｸｾｽする様なｾｯﾀｰ
}
```

```
// ----- Player クラス定義
class Player
{

public:

  int pos_x;
  int pos_y;


private:
  int x;
  int y;


public:
  p_obj.Set(int a, int b){
      x = a;
      y = b;
};
```

□アクセス指定子にまつわるクラスと構造体の違い

1. private と public の定義時に指示をしなかった場合は、初期設定が逆になっています。
   クラスはデフォルトで private(非公開)、構造体はデフォルトで public(公開)です。

```
// ----- クラスの場合

class クラス名
{
    // ①変数とか関数とか

private:
    // ②変数とか関数とか

public:
    // ③変数とか関数とか

};
```

```
// ----- 構造体の場合

struct 構造体名
{
    // ④変数とか関数とか

private:
    // ⑤変数とか関数とか

public:
    // ⑥変数とか関数とか

}
```

上記の設定の場合、それぞれの属性は下記の様になります。

①private
②private
③public

④public
⑤private
⑥public

■コンストラクタとデストラクタ

クラスの中の関数を「メンバ関数」と言いますが、そのメンバ関数の中には「コンストラクタ」「デストラクタ」と呼ばれる特殊な関数があります。

■コンストラクタ

そのクラスをインスタンス化した時に、自動的に呼び出されるメンバ関数です。
特徴は
- ・コンストラクタ名は、その型名と同じ名前（の関数）
- ・戻り値がない
- ・引数を持たせる事ができる
- ・初期化子というのを利用する事ができる

です。コンストラクタ内では、通常では初期化などの作業をします。

コンストラクタの定義①

```
例）Player::Player();

クラス名::クラス名();     // 関数の実体は他の関数同様 cpp ファイルで記述する。
                              ※クラス定義時に h ファイルにも記述できます（後ほど説明）
```

コンストラクタの定義②（引数も可能）

```
例）Player::Player(引数);

クラス名::クラス名(引数);     // 通常の関数同様、引数を持つ事もできます。（複数可）
```

コンストラクタの定義③（初期化子）

```
例）
class Sample
{
public:
   const int NUM_MAX;
   int num;

   Sample :: Sample() : NUM_MAX(100)     // 定数 NUM_MAX に 100 をセットして実行
   {
      num = NUM_MAX;
   }

};
```

コンストラクタの定義時に、:で区切って()に値を入れると、メンバ変数をその値を初期化する事ができます。通常の初期化であれば、引数で処理する事もできますが、その変数が定数（const）の時は、「const の変数が初期化されていないとコンパイルエラーになってしまいます。その場合は、コンストラクタと同時（前）に const の変数を初期化する必要があるので、初期化子を利用して値を設定するのです。

■デストラクタ

クラスのインスタンスが解放される時に、解放直前に自動的に呼び出されます。簡単にいうと「オブジェクトが削除される時」「main()の処理などが終了する時などで、消える寸前に後始末をする様なイメージです。

デストラクタの定義

```
例）Player::~Player();

クラス名::~クラス名();     // デストラクタ名はクラス名の先頭に~（チルダ）を付けます。
```

終了処理というのは、動的に確保されたメモリの解放など主にメモリ関係の処理になります。デストラクタのタイミングを逃すと、メモリの解放を行う事ができなくなります。

□コンストラクタとデストラクタは省略可能

コンストラクタとデストラクタは、必要がなければ作る必要はありません。省略してもコンパイルされます。

コンストラクタとデストラクタの定義の例①

```
// ----- Player クラス定義
class Player
{
public:

  // ｺﾝｽﾄﾗｸﾀ
  Player();

  // ﾃﾞｽﾄﾗｸﾀ
  ~Player();

};
```

```
// ----- Player 実体
#include”Player.h”

// ｺﾝｽﾄﾗｸﾀ
Player::Player()
{
    ※なんらかの処理
}

// ﾃﾞｽﾄﾗｸﾀ
Player::~Player()
{
    ※なんらかの処理
}
```

コンストラクタとデストラクタの定義の例②

```
// ----- Player クラス定義
class Player
{
public:

  // ｺﾝｽﾄﾗｸﾀ
  Player(){
      ※なんらかの処理
  }

  // ﾃﾞｽﾄﾗｸﾀ
  ~Player(){
      ※なんらかの処理
  }

};
```

定義と同時に{}と処理を記載する事も可能です。
その場合 cpp ファイルでの実態定義の記載は不要ですが、ヘッダーファイルでの処理記入はなるべく避けましょう。処理が少ない場合や処理そのものを省略する場合などに留めておく方が良いです。

■newとdelete

コンストラクタとデストラクタは、インスタンスの生成と解放のタイミングで呼び出されますが、ゲームを作っているとそれぞれ呼び出したいタイミングが違ってきたりします（しかるべきタイミングで敵を発生させたい時、画像を読み込みたい時など）。

ある関数でクラスオブジェクトを定義していると、コンパイル時にそれぞれのオブジェクトが生成され（スタック）、その時自動的にコンストラクタも呼ばれます。デストラクタも関数が終了するまで呼ばれないなど、コントロールはOSやコンパイラが行う事になります。

そこで、インスタンスそのものの生成や消去のタイミングをプログラム側がコントロールする方法として「new」「delete」を使用します。保存先としてもヒープ領域となり容量も大きく動的確保が可能となりますので、C 言語でいう“malloc”と同様のものです。

□そこで new 演算子の登場

早速、普通のオブジェクトの定義と new を使った定義をやってみます。

```
※Player クラスがあると想定

void main()
{
   Player  p1;          // ①Player 型のｵﾌﾞｼﾞｪｸﾄを生成し p1 に保持

   Player* p2;          // ②動的確保するｵﾌﾞｼﾞｪｸﾄのｱﾄﾞﾚｽを生成
   p2 = new Player();   // ②Player 型の領域を動的確保

   delete p2;           // ②動的確保した領域を解放
}
```

①と②の内部的な違い

項目	①スタック	②ヒープ ※new
確保の仕方	Player p1	Player* p2 = new Player();
管理	OS/コンパイラ	プログラマ
生存期間	関数終了/デストラクタまで	delete/free まで
メモリ	スタック領域（少ない）	ヒープ領域（多い）
変数サイズ	コンパイル時に固定	実行時に動的
解放	関数終了時に自動	delete p2;

new 演算子の書式

```
new コンストラクタ名
```

注意点としては、new の後に来るのはクラス名でなくコンストラクタ名という事です。記述的には同じ様な感じですが、本来の役割はコンストラクタを呼び出してインスタンスを生成する事となります。従って、コンストラクタに引数が設定されている場合は同時に引数をセットしていきます。また、戻り値は確保したアドレスとなりますので、受け取る変数は、それぞれのオブジェクト型のﾎﾟｲﾝﾀにして置くと良いでしょう。

```
例
   Player* p3;     // Player 型のﾎﾟｲﾝﾀ変数
   p3 = new Player(posX, posY, playerType, playerLevel);     // 引数も同時にｾｯﾄ
```

□次に delete 演算子

delete を実行すると new で確保した領域を示すポインタを開放します。delete されたインスタンスは、デストラクタが呼び出され消去される事になります。

delete 演算子の書式

```
delete インスタンス名
```

```
例
   Player* p3;     // Player 型のﾎﾟｲﾝﾀ変数
   p3 = new Player(posX, posY, playerType, playerLevel);     // 引数も同時にｾｯﾄ

   delete p3;     // 領域確保
```

□「new と delete はセット」、malloc と free との違いは？

「new と delete」の関係は、「malloc と free」に近いイメージです。どちらも動的なメモリの確保を行うものです。だた、C++言語では malloc と free は推奨されません。両者との大きな違いは、malloc と free は、「コンストラクタ、デストラクタを呼び出す事ができない」からです。その為、C++言語ではメモリの生成および消去には new と delete が用いられます。

new して確保したものは必ず delete しましょう！

■まとめ

このドキュメントで出て来た用語の一覧です。各キーワードの理解はいかがでしょうか。

- クラスの定義
- クラスの実装
- クラスの利用
- アクセス指定子（private）
- アクセス指定子(public)
- コンストラクタ
- デストラクタ
- new 演算子
- delete 演算子