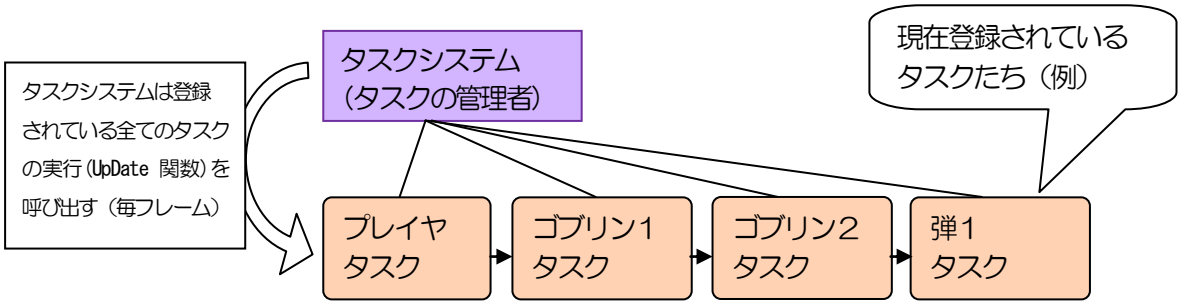
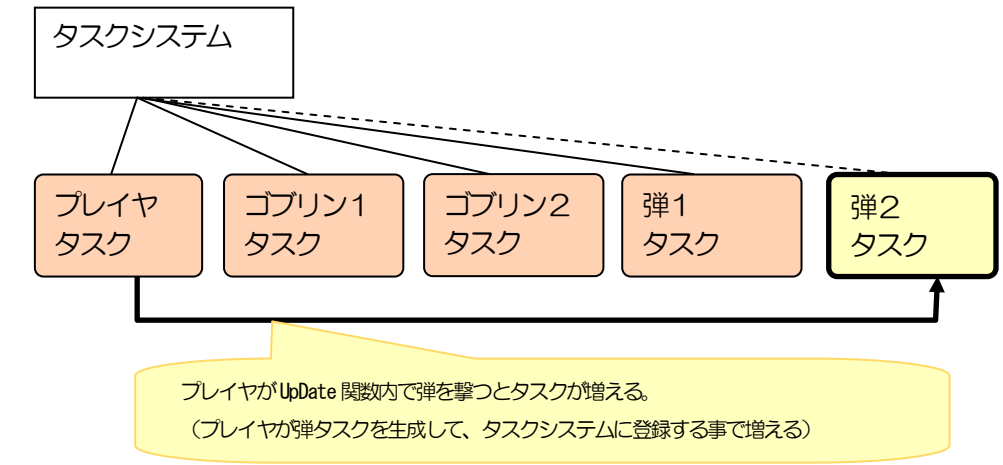
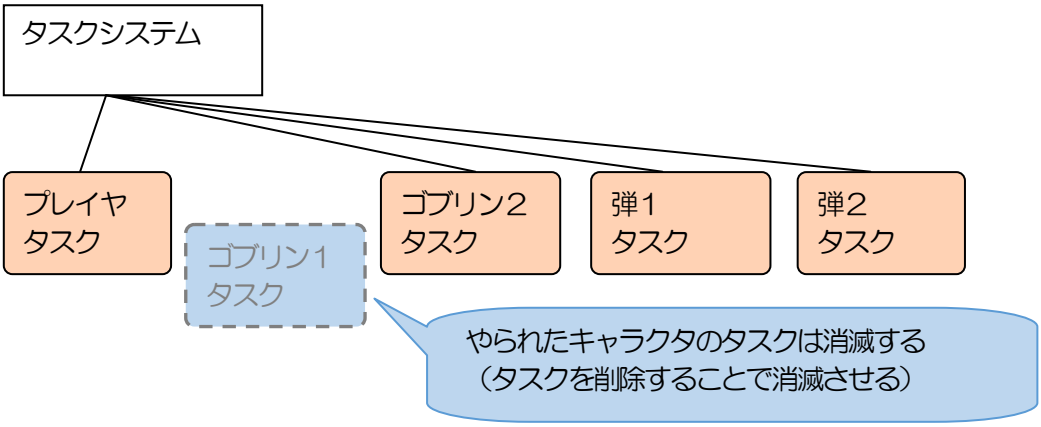
12. タスクシステムを用いたスケルトンプログラム

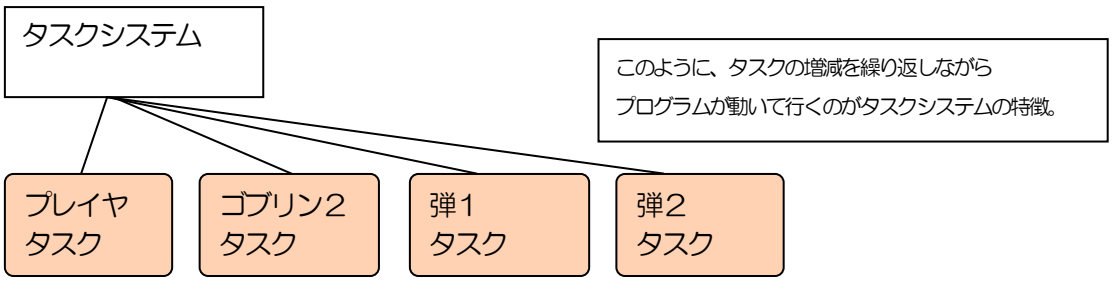
12. 1. タスクシステム

タスクシステムとは、タスクという単位で作られたプログラムが必要に応じて生成・実行・消滅しながら動くプログラムです。
アクションゲームのキャラクタで例えてみましょう。
何かのきっかけで敵キャラクタは出現します。（ある地点に近づいた時・ボスから生み出された時など）
これらのキャラクタは自身で行動し、プレイヤに倒された・ボスの死亡に巻き込まれた等、何らかの理由で消滅します。
このキャラクター体一体が１つのタスクだと考えるとわかりやすいでしょう。









消滅すると言っても、キャラクタを制御するプログラム自体が消えてなくなるわけではありません。
キャラクタのパラメータ情報を入れている変数（構造体）が、プログラム実行中に増えたり減ったりすると言った方が正しいです。

例えば、敵キャラクタを管理する為に、３０件分の配列を用意したのを覚えていますか？（GPG1 16 章）
配列はサイズが固定ですから、キャラクタに有効•無効のフラグを持たせて存在しない事にしていました。

色々端折って説明すると、メモリの動的確保と開放という仕組みを習得する事で、この配列のサイズをプログラム実行中に変更できるようになります。
例えば、生存している敵キャラクタが２４体だったとすると、２４体分のデータを残して配列を小さくしたり、新たに出現する敵の為に配列のサイズを大きくしたりといった事ができるようになります。

このタスクシステムの仕組みを理解するには少なくとも「メモリの動的確保」「クラス」「クラスの継承」「リスト構造」を理解する必要があります。　現時点では勉強不足です。

タスクシステムを利用してゲームのプログラムを作成するだけならそれほど難しくはありません。

12．2．タスク

１つ１つのタスクは、１１章までのプログラムで作ってきたタスク（シーン）と同様に、個々に「初期化」「更新」「描画」「終了」のプログラムを実装していきます。

今回のタスクシステムおよびタスクにはＣ＋＋のクラスという仕組みを使用しています。
クラスは「構造体に関数を追加して出来たもの」程度に捉えておいてください。

新しいタスクは、あらかじめ用意された「ひな形」のソースファイルをコピーして、それを書き換える事で作成できます。

中身を理解するのは現状では困難ですが、ひな形を利用してタスクを追加していくだけならさほど難しくはありません。

まずは、用意されているひな形のファイルを確認しましょう。
（網掛けの部分にのみ注目し、わからないところは読み飛ばして構いません。

Task_ひな形（2D_BTask）．h

```
#pragma warning(disable:4996)
#pragma once
//-----------------------------------------------------------------------
//
//-----------------------------------------------------------------------
#include "GameEngine_Ver3_83.h"

namespace 「ネームスペース名」
{
    //タスクに割り当てるグループ名と固有名
    const  string  defGroupName(      「グループ名」);    //グループ名
    const  string  defName(           「タスク名」);      //タスク名
    //-------------------------------------------------------------------
    class  Resource : public BResource
    {
        bool  Initialize()override;
        bool  Finalize()  override;
        Resource();
    public:
        ~Resource();
        typedef  shared_ptr<Resource>    SP;
        typedef  weak_ptr<Resource>      WP;
        static   WP  instance;
        static  Resource::SP  Create();
        //共有する変数はここに追加する
        「変数宣言を書く」（共有）
    };
    //-------------------------------------------------------------------
    class  Object : public  BTask
    {
    //変更不可◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
    public:
        virtual  ~Object();
        typedef  shared_ptr<Object>      SP;
        typedef  weak_ptr<Object>        WP;
        //生成窓口 引数はtrueでタスクシステムへ自動登録
        static  Object::SP  Create(bool flagGameEnginePushBack_);
        Resource::SP   res;
    private:
        Object();
        bool  B_Initialize();
        bool  B_Finalize();
        bool  Initialize();              //「初期化」タスク生成時に１回だけ行う処理
        void  UpDate()        override;  //「実行」１フレーム毎に行う処理
        void  Render2D_AF()   override;  //「2D描画」１フレーム毎に行う処理
        bool  Finalize();                //「終了」タスク消滅時に１回だけ行う処理
    //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
```

```
        //追加したい変数・メソッドはここに追加する
        「変数宣言を書く」
        「追加メソッドを書く」（専有）
    };
}
```

ここでは、クラスの型定義を行っています。
見慣れない構文に面食らうかもしれませんが、今必要なのは網掛け部分だけです。

タスク（キャラクタ）に必要なメンバ変数を「変数宣言を書く」の部分に記述するので、構造体の型定義と同じようなものと考えておけばよいでしょう。
上側の欄には同一のタスクが複数存在しているとき、その全員が共有する情報を、下の欄には個々のタスクがそれぞれに持つ情報を記述していきます。
ゲームプログラミングで使用するタスクは２つのクラスにより作られています。
共有情報を管理するのが「Resorce（リソース）」クラス、
専有情報を管理するのが「Object（オブジェクト）」クラスです。
その内容を理解するにはC++の勉強が必要なので、取りあえず使い方を身に付けましょう。

Task_ひな形（2D_BTask）．cpp

```
//-------------------------------------------------------------------
//
//-------------------------------------------------------------------
#include  "MyPG.h"
#include  "該当する.h"

namespace  「ネームスペース名」
{
    Resource::WP  Resource::instance;
    //-------------------------------------------------------------------
    //リソースの初期化
    bool  Resource::Initialize()
    {
        return true;
    }
    //-------------------------------------------------------------------
    //リソースの解放
    bool  Resource::Finalize()
    {
        return true;
    }
    //-------------------------------------------------------------------
    //「初期化」タスク生成時に１回だけ行う処理
    bool  Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, true);
```

```
        //リソースクラス生成 or リソース共有
        this->res = Resource::Create();

        //★データ初期化

        //★タスクの生成

        return  true;
    }
    //-----------------------------------------------------------------------
    //「終了」タスク消滅時に１回だけ行う処理
    bool  Object::Finalize()
    {
        //★データ＆タスク解放


        if (!ge->QuitFlag() && this->nextTaskCreate) {
            //★引き継ぎタスクの生成
        }

        return  true;
    }
    //-----------------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
    }
    //-----------------------------------------------------------------------
    //「２D描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
    }
    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
    //以下は基本的に変更不要なメソッド
    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
以下省略
```

タスクの.cpp ファイルには「初期化」「更新」「描画」「終了」の４種類の関数が用意されています。
これらの４つの関数の役割は１１章以前のものと同じです。

Initialize( ) と Finalize( )は２つありますが、前者は Resource クラスの機能で「共有情報」
の準備するために使用し、後者は Object クラスの機能で「専有情報」の準備をするためのものです。

このひな形を利用して、タイトル画面を表示するタスクを実装してみましょう。

12. 3. タイトルタスクの実装

ひな形を使用して、タイトル画面の処理を行うタスクを実現します。
使用するひな形は「2D_BTask」と書かれた方の.hと.cppファイルです。（BCharaではありません）
プロジェクトフォルダ内にある「タスクひな形」フォルダの中から、該当するファイルをコピーして、
プロジェクトフォルダ内にTask_Title.cppとTask_Title.hファイルを作成します。
＊コピーしたファイルを置く場所は「タスクひな形」フォルダではありません。
　このフォルダの外（１つ前のフォルダ）に置いて下さい。
次は、VisualStudioのソリューションエクスプローラーから、MyPG/Tasks/Titleフィルタを開きます。
（該当するフィルタが無い場合は作って下さい）
＊VisualStudioのソリューションエクスプローラ上に表示されるフォルダのようなものは「フィルタ」です。
これはVisualStudio上でのみ機能する分類わけの機能です。

Titleフィルタ上にマウスカーソルを置き、右クリックでメニューを表示します。
メニューから「追加→既存の項目」を選び、先ほど作成したTask_Title.cppとTask_Title.hファイルをプロジェクトに追加します。
（ここまでの手順に間違いがあると、以降のプログラムを正しく入力しても実行出来ません）

Task_Title．h

```
#pragma warning(disable:4996)
#pragma once
//-----------------------------------------------------------------------
//タイトル画面
//-----------------------------------------------------------------------
#include "GameEngine_Ver3_83.h"

namespace  Title
{
    //タスクに割り当てるグループ名と固有名
    const  string  defGroupName("タイトル");     //グループ名
    const  string  defName("NoName");           //タスク名
    //-------------------------------------------------------------------
    class  Resource : public BResource
    {
        bool  Initialize()override;
        bool  Finalize()  override;
        Resource();
    public:
        ~Resource();
        typedef  shared_ptr<Resource>    SP;
        typedef  weak_ptr<Resource>      WP;
        static   WP  instance;
        static  Resource::SP  Create();
        //共有する変数はここに追加する
        DG::Image::SP  img;
    };
    //-------------------------------------------------------------------
    class  Object : public  BTask
    {
```

```
    //変更不可◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
    public:
        virtual  ~Object();
        typedef  shared_ptr<Object>      SP;
        typedef  weak_ptr<Object>        WP;
        //生成窓口 引数はtrueでタスクシステムへ自動登録
        static  Object::SP  Create(bool flagGameEnginePushBack_);
        Resource::SP   res;
    private:
        Object();
        bool  B_Initialize();
        bool  B_Finalize();
        bool  Initialize();              //「初期化」タスク生成時に１回だけ行う処理
        void  UpDate()        override;  //「実行」１フレーム毎に行う処理
        void  Render2D_AF()   override;  //「2D描画」１フレーム毎に行う処理
        bool  Finalize();                //「終了」タスク消滅時に１回だけ行う処理
    //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
        //追加したい変数・メソッドはここに追加する
        int         logoPosY;       //タイトル画像スクロール用カウンタ
    };
}
```

namespaceにTitleという名前を付けたのは、１１章までのタスク（シーン）と同じです。
タスクが行う処理に適した名前を付けてください。

タスクには「グループ名」と「タスク名」を付けることが出来ます。
これは、タスクを検出する時に必要になるものです。
ゲームを作っていくと、タスクを検出する仕組みが必ず必要になります。
（例えば、弾と敵のあたり判定をする時、全ての敵を検出する必要が生じます）

今回は、タスクが使用するオフスクリーン用の変数「DG::Image::SP　img;」と、タイトル画面のロゴを画面外からスライドさせるために使用する変数「int　 logoPosY;」の２つの変数が追加されています。

この２つの変数は、それぞれ別々のクラスに宣言を書いています。Resourceクラス（共有情報）とObjectクラス（専有情報）のどちらに加えるべきかの判断は
「そのタスクが異なるタイミングで複数体生成された時、共有できる情報か否か」
により判断できます。

タイトル画面のロゴ画像は、共有が可能なので、変数「DG::Image::SP　img;」はResourceクラス側に、時間によりロゴ画像の表示位置を変化させる変数「int　 logoPosY;」は共有が出来ないので
Objectクラス側に宣言を書きます。
新しい変数を追加する時も、同様の条件でどちらに宣言すべきか判断してください。

続けて、.cppファイルの実装を行います。

Task_Title . cpp

```
//-------------------------------------------------------------------
//タイトル画面
//-------------------------------------------------------------------
#include  "MyPG.h"
#include  "Task_Title.h"

namespace  Title
{
    Resource::WP  Resource::instance;
    //-------------------------------------------------------------------
    //リソースの初期化
    bool  Resource::Initialize()
    {
        this->img = DG::Image::Create("./data/image/Title.bmp");
        return true;
    }
    //-------------------------------------------------------------------
    //リソースの解放
    bool  Resource::Finalize()
    {
        this->img.reset( );
        return true;
    }
    //-------------------------------------------------------------------
    //「初期化」タスク生成時に１回だけ行う処理
    bool  Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, true);
        //リソースクラス生成orリソース共有
        this->res = Resource::Create();
        //★データ初期化
        this->logoPosY = -270;
        //★タスクの生成

        return  true;
    }
    //-------------------------------------------------------------------
    //「終了」タスク消滅時に１回だけ行う処理
    bool  Object::Finalize()
    {
        //★データ＆タスク解放

        if (!ge->QuitFlag() && this->nextTaskCreate) {
            //★引き継ぎタスクの生成
        }
        return  true;
    }
```

```
    //-------------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
        auto inp = ge->in1->GetState();

        this->logoPosY += 9;
        if (this->logoPosY >= 0) {
            this->logoPosY = 0;
        }

        if (this->logoPosY == 0) {
            if (inp.ST.down) {
                //自身に消滅要請
                this->Kill();
            }
        }
    }
    //-------------------------------------------------------------------
    //「２D描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
        ML::Box2D  draw(0, 0, 480, 270);
        ML::Box2D  src(0, 0, 240, 135);
        draw.Offset(0, this->logoPosY);
        this->res->img->Draw(draw, src);
    }

    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
    //以下は基本的に変更不要なメソッド
    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
以下省略（変更が不要なだけで、不要なわけではありません、消してはいけません！）
```

実装した処理は、ゲームプログラミング１の１６章で実装したタイトル画面とほぼ同じものです。
Resource クラス側の初期化の処理ではタイトルロゴ用の画像を読み込み、
Object クラス側の初期化の処理では、タイトルロゴの位置を管理する変数を１画面分上の座標に
設定しています。

.h ファイルに宣言を書いた変数にアクセスする時、this->という記述が加わっています。
これは、クラスが自分の持っている変数（メンバ変数）や関数（メソッド）にアクセスする際に使用する
ものです。（this は自分自身を指すポインタです）

省略は可能ですが、意味を理解するまでは記述するように心がけましょう。
（気になる人は、クラスを勉強して下さい）

とりあえず、.h ファイル上に宣言を書いた変数にアクセスするときは、this->を付けると覚えておけば良いでしょう。
これで、タイトルタスクの実装は終わりです。
ただし、これはいわゆる「構造体の型定義を行った」と「関数を作った」に相当するものなので、
これだけではタイトルタスクは機能しません。

タスクの生成と、タスクシステムへの登録を行って、初めてタスクは動き始めます。

12．4．タイトルタスクの生成とタスクシステムへの登録

タスクはプログラム上で「生成」し、タスクシステムへ登録をすることで初めて機能するようになります。

以下のコードを追加して、タイトルタスクを機能させましょう。

MyPG．cpp　　（ファイル先頭部分と、１００行目前後）

ファイルの先頭付近

```
#include "MyPG.h"
#include "Task_Title.h"

namespace MyPG
{
```

100 行目程度の位置↓コメントを参考に探す

```
        //初期実行タスク生成&ゲームエンジン（タスクシステム）に登録
        auto  firstTask  =  Title::Object::Create(true);
```

補足
タスクを生成し、タスクシステムへの登録を行うだけなら、

```
Title::Object::Create(true);
```

と書くだけで可能です。

Title::Object::Create( )関数が、タスクの生成を行います。
この引数部分に true を指定することで、生成したタスクがタスクシステムに自動で登録されます。
false を指定した場合、自動登録が行われないため、別途自分で登録をする必要があります。

```
auto  firstTask  =  Title::Object::Create(false);   //タスクを生成
ge->PushBack(firstTask);                //ゲームエンジン内のタスクシステムに登録
```

＊授業の中でこの使い方をする事はありません。

タスクを生成し、タスクシステムに登録すると、以後１フレーム毎にタスクの UpDate( )と
Render2D_AF( )メソッド（メンバ関数）が呼び出されるようになります。
（タスクシステムの処理内部で、これらのメソッドが呼び出されます）
プログラムを実行して動作確認しましょう。

12．5．タスクの消滅

一度生成・登録されたタスクはタスク自身、もしくは別のタスクから消滅の指示を受けない限り
存在し続けます。

Ｐ９５のObject::UpDate()メソッド内で、「//自身に消滅要請」と書かれているところを見てください。
タイトル画面の実行中にSTARTボタンを押すと、タイトルタスクが自分に対して消滅を要請します。

```
this->Kill();
```

タスクに消滅要請を行うと、そのフレームの処理が終わった後（全てのタスクのUpDate( )が実行された後）で、消滅処理が行われます。

（消滅要請をした時点ではタスクは消滅しません）
消滅処理の最中に各タスクのFinalize( )処理が実行されます。

12．6．本編タスクを追加する

今度は、ゲームの本編タスクを追加します。
本編タスクは背景画像を表示し、方向キーの入力に応じて動く黄色いスライムが１体出るものとして
作ります。（ゲームプログラミング１の１６章と同様）

今回は１つのタスクではなく「本編統括」「背景表示」「スライム」の３つのタスクに分けて実装します。

まず「背景表示」タスクを実装します。

Task_GameBG．h

```
#pragma warning(disable:4996)
#pragma once
//-------------------------------------------------------------------
//ゲーム本編背景
//-------------------------------------------------------------------
#include "GameEngine_Ver3_83.h"

namespace  GameBG
{
    //タスクに割り当てるグループ名と固有名
    const  string  defGroupName( "本編");    //グループ名
    const  string  defName(      "背景");    //タスク名
    //-------------------------------------------------------------------
    class  Resource : public BResource
    {
        bool  Initialize()override;
        bool  Finalize()  override;
        Resource();
    public:
        ~Resource();
        typedef  shared_ptr<Resource>    SP;
        typedef  weak_ptr<Resource>      WP;
```

```
        static   WP  instance;
        static  Resource::SP  Create();
        //共有する変数はここに追加する
        DG::Image::SP  img;
    };
    //-------------------------------------------------------------------
    class  Object : public  BTask
    {
        //変更不可◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
    public:
        virtual  ~Object();
        typedef  shared_ptr<Object>      SP;
        typedef  weak_ptr<Object>        WP;
        //生成窓口 引数はtrueでタスクシステムへ自動登録
        static  Object::SP  Create(bool flagGameEnginePushBack_);
        Resource::SP   res;
    private:
        Object();
        bool  B_Initialize();
        bool  B_Finalize();
        bool  Initialize();             //「初期化」タスク生成時に１回だけ行う処理
        void  UpDate()        override;  //「実行」１フレーム毎に行う処理
        void  Render2D_AF()   override;  //「2D描画」１フレーム毎に行う処理
        bool  Finalize();               //「終了」タスク消滅時に１回だけ行う処理
        //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
        //追加したい変数・メソッドはここに追加する
    };
}
```

Task_GameBG . cpp

```
//-------------------------------------------------------------------
//ゲーム本編背景
//-------------------------------------------------------------------
#include  "MyPG.h"
#include  "Task_GameBG.h"

namespace  GameBG
{
    Resource::WP  Resource::instance;
    //-------------------------------------------------------------------
    //リソースの初期化
    bool  Resource::Initialize()
    {
        this->img = DG::Image::Create("./data/Image/GameBG.bmp");
        return true;
    }
```

```
    //-------------------------------------------------------------------
    //リソースの解放
    bool  Resource::Finalize()
    {
        this->img.reset();
        return true;
    }
    //-------------------------------------------------------------------
    //「初期化」タスク生成時に１回だけ行う処理
    bool  Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, true);
        //リソースクラス生成orリソース共有
        this->res = Resource::Create();
        //★データ初期化
        this->render2D_Priority[1] = 1.0f;
        //★タスクの生成
        return  true;
    }
    //-------------------------------------------------------------------
    //「終了」タスク消滅時に１回だけ行う処理
    bool  Object::Finalize()
    {
        //★データ＆タスク解放

        if (!ge->QuitFlag() && this->nextTaskCreate) {
            //★引き継ぎタスクの生成
        }
        return  true;
    }
    //-------------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
    }
    //-------------------------------------------------------------------
    //「２D描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
        //背景描画
        ML::Box2D  draw(0, 0, 480, 270);
        ML::Box2D  src(0, 0, 480, 270);
        this->res->img->Draw(draw, src);
    }
    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
    //以下は基本的に変更不要なメソッド
    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
以下省略
```

背景表示タスクは背景画像を表示するだけのタスクなので、実装する処理はほとんどありません。
次に「本編統括」タスクを実装します。

Task_Game . h

```
#pragma warning(disable:4996)
#pragma once
//-----------------------------------------------------------------------
//ゲーム本編
//-----------------------------------------------------------------------
#include "GameEngine_Ver3_83.h"

namespace  Game
{
    //タスクに割り当てるグループ名と固有名
    const  string  defGroupName( "本編");    //グループ名
    const  string  defName(      "統括");    //タスク名
    //-------------------------------------------------------------------
    class  Resource : public BResource
    {
        bool  Initialize()override;
        bool  Finalize()  override;
        Resource();
    public:
        ~Resource();
        typedef  shared_ptr<Resource>    SP;
        typedef  weak_ptr<Resource>      WP;
        static   WP  instance;
        static  Resource::SP  Create();
        //共有する変数はここに追加する
    };
    //-------------------------------------------------------------------
    class  Object : public  BTask
    {
        //変更不可◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
    public:
        virtual  ~Object();
        typedef  shared_ptr<Object>      SP;
        typedef  weak_ptr<Object>        WP;
        //生成窓口 引数はtrue でタスクシステムへ自動登録
        static  Object::SP  Create(bool flagGameEnginePushBack_);
        Resource::SP   res;
    private:
        Object();
        bool  B_Initialize();
        bool  B_Finalize();
        bool  Initialize();              //「初期化」タスク生成時に１回だけ行う処理
        void  UpDate()        override;  //「実行」１フレーム毎に行う処理
        void  Render2D_AF()   override;  //「2D 描画」１フレーム毎に行う処理
        bool  Finalize();                //「終了」タスク消滅時に１回だけ行う処理
```

```
        //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
        //追加したい変数・メソッドはここに追加する
    };
}
```

Task_Game．cpp

```
//-------------------------------------------------------------------
//ゲーム本編
//-------------------------------------------------------------------
#include  "MyPG.h"
#include  "Task_Game.h"
#include  "Task_GameBG.h"

namespace  Game
{
    Resource::WP  Resource::instance;
    //-------------------------------------------------------------------
    //リソースの初期化
    bool  Resource::Initialize()
    {
        return true;
    }
    //-------------------------------------------------------------------
    //リソースの解放
    bool  Resource::Finalize()
    {
        return true;
    }
    //-------------------------------------------------------------------
    //「初期化」タスク生成時に１回だけ行う処理
    bool  Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, true);
        //リソースクラス生成orリソース共有
        this->res = Resource::Create();

        //★データ初期化

        //★タスクの生成
        //背景タスク
        auto  bg = GameBG::Object::Create(true);

        return  true;
    }
    //-------------------------------------------------------------------
```

```
    //「終了」タスク消滅時に１回だけ行う処理
    bool  Object::Finalize()
    {
        //★データ＆タスク解放
        ge->KillAll_G("本編");
        if (!ge->QuitFlag() && this->nextTaskCreate) {
            //★引き継ぎタスクの生成
        }
        return  true;
    }
    //-------------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
        auto inp = ge->in1->GetState( );
        if (inp.ST.down) {
            //自身に消滅要請
            this->Kill();
        }
    }
    //-------------------------------------------------------------------
    //「２Ｄ描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
    }
    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
    //以下は基本的に変更不要なメソッド
    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
以下省略
```

本編統括タスクは、本編開始時・終了時に必要な処理を執り行う統括役と考えてください。
本編が開始されるとき「背景表示タスク」と「プレイヤタスク」を生成・登録し、本編終了時に
「背景表示タスク」と「プレイヤタスク」を消滅させます。
（プレイヤタスクは現在実装されていません）

タイトル画面の処理が終了した時、次に実行されるタスク（引き継ぎタスク）として本編タスクを
生成・登録することで「タイトル画面→ゲーム本編」への流れを実現します。

Task_Title．cpp　ファイル先頭部分および、Object::Finalize( )メソッド内の該当箇所に追加

ファイル先頭部分に追加

```
#include  "Task_Game.h"
```

Object::Finalize( )メソッド内に追加

```
if (!ge->QuitFlag() && this->nextTaskCreate) {
    //★引き継ぎタスクの生成
    auto nextTask = Game::Object::Create(true);
}
```

本編タスクが開始でき、背景画像が表示されることを確認したらプレイヤタスクを追加します。

Task_Player . h

```
#pragma warning(disable:4996)
#pragma once
//-------------------------------------------------------------------
//プレイヤ（仮
//-------------------------------------------------------------------
#include "GameEngine_Ver3_83.h"

namespace  Player
{
    //タスクに割り当てるグループ名と固有名
    const  string  defGroupName( "プレイヤ");    //グループ名
    const  string  defName(      "NoName");     //タスク名
    //-------------------------------------------------------------------
    class  Resource : public BResource
    {
        bool  Initialize()override;
        bool  Finalize()  override;
        Resource();
    public:
        ~Resource();
        typedef  shared_ptr<Resource>    SP;
        typedef  weak_ptr<Resource>      WP;
        static   WP  instance;
        static  Resource::SP  Create();
        //共有する変数はここに追加する
        DG::Image::SP  img;
    };
    //-------------------------------------------------------------------
    class  Object : public  BTask
    {
        //変更不可◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
    public:
        virtual  ~Object();
        typedef  shared_ptr<Object>      SP;
        typedef  weak_ptr<Object>        WP;
        //生成窓口 引数はtrueでタスクシステムへ自動登録
        static  Object::SP  Create(bool flagGameEnginePushBack_);
        Resource::SP   res;
    private:
        Object();
        bool  B_Initialize();
        bool  B_Finalize();
        bool  Initialize();              //「初期化」タスク生成時に１回だけ行う処理
        void  UpDate()        override;  //「実行」１フレーム毎に行う処理
        void  Render2D_AF()   override;  //「2D描画」１フレーム毎に行う処理
        bool  Finalize();                //「終了」タスク消滅時に１回だけ行う処理
        //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
```

```
        //追加したい変数・メソッドはここに追加する
        XI::GamePad::SP  controller;
        ML::Point      pos;
    };
}
```

Task_Player . cpp

```
//-------------------------------------------------------------------
//プレイヤ（仮
//-------------------------------------------------------------------
#include  "MyPG.h"
#include  "Task_Player.h"

namespace  Player
{
    Resource::WP  Resource::instance;
    //-------------------------------------------------------------------
    //リソースの初期化
    bool  Resource::Initialize()
    {
        this->img = DG::Image::Create("./data/Image/Chara00.png");
        return true;
    }
    //-------------------------------------------------------------------
    //リソースの解放
    bool  Resource::Finalize()
    {
        this->img.reset();
        return true;
    }
    //-------------------------------------------------------------------
    //「初期化」タスク生成時に１回だけ行う処理
    bool  Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, true);
        //リソースクラス生成orリソース共有
        this->res = Resource::Create();
        //★データ初期化
        this->controller = ge->in1;
        this->render2D_Priority[1] = 0.5f;
        this->pos.x = 480 / 2;
        this->pos.y = 270 * 2 / 3;
        //★タスクの生成
        return  true;
    }
```

```
    //-------------------------------------------------------------------
    //「終了」タスク消滅時に１回だけ行う処理
    bool  Object::Finalize()
    {
        //★データ＆タスク解放

        if (!ge->QuitFlag() && this->nextTaskCreate) {
            //★引き継ぎタスクの生成
        }
        return  true;
    }
    //-------------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
        if (this->controller) {
            auto  inp = this->controller->GetState();
            if (inp.LStick.BL.on) { this->pos.x -= 3; }
            if (inp.LStick.BR.on) { this->pos.x += 3; }
            if (inp.LStick.BU.on) { this->pos.y -= 3; }
            if (inp.LStick.BD.on) { this->pos.y += 3; }
        }
    }
    //-------------------------------------------------------------------
    //「２Ｄ描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
        //キャラクタ描画
        ML::Box2D  draw(-32, -16, 64, 32);
        draw.Offset(this->pos);
        ML::Box2D  src(0, 0, 64, 32);
        this->res->img->Draw(draw, src);
    }

    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
    //以下は基本的に変更不要なメソッド
    //★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★★
以下省略
```

プレイヤタスクが実装できたら、背景表示タスクと同様に、Task_Game.cpp の Object::Initialize( ) メソッド内に、プレイヤタスクの生成・登録の処理を追加します。

Task_Game . cpp　ファイル先頭部分および、Object::Initialize( )メソッド内の該当箇所に追加

ファイル先頭部分に追加

```
#include  "Task_Player.h"
```

Object::Initialize( )メソッド内に追加

```
    //プレイヤタスク
    auto  pl = Player::Object::Create(true);
```

これで、ゲーム本編中にプレイヤが出現するようになります。
実行結果を確認してください。

プログラム実行中に BackSpace キーを２回押すと、画面に現在実行中のタスクの一覧が表示されます。

ゲーム本編では、「本編＞統括」「本編＞背景」「プレイヤ＞NoName」が表示されるはずです。

その状態でＳＴＡＲＴボタンを押すと、「本編」タスクが自滅し、「背景表示」タスクもつられて消滅します。
今は、プレイヤのタスクを消滅させる処理を入れ忘れているので、プレイヤタスクだけがとり残されます。
（生きているタスクが残っている限り、プログラムは終了しません）

このようにタスクを消し忘れると、シーンが切り替わったとき、前のシーンのタスクが残ってしまいます。
本編の終了時の処理に、「プレイヤ」を消滅させる処理を追加して、再度動作確認してください。

Task_Game . cpp　該当箇所のみ抜粋

```
    //-----------------------------------------------------------------------
    //「終了」タスク消滅時に１回だけ行う処理
    bool  Object::Finalize()
    {
        //★データ＆タスク解放
        ge->KillAll_G("本編");
        ge->KillAll_G("プレイヤ");
        if (!ge->QuitFlag() && this->nextTaskCreate) {
            //★引き継ぎタスクの生成
        }
        return  true;
    }
```

12. 7. 倍々ゲーム

プレイヤキャラクタのObject::UpDate( )処理内に以下のコードを実装し、動作確認してください。

```
if (inp.B1.down) {
    auto  pl = Player::Object::Create(true);
    pl->pos = this->pos;
    pl->pos.x += (rand() % 100) - 50;
    pl->pos.y += (rand() % 100) - 50;
}
```

本編処理中にＢ１ボタンを押すと、プレイヤキャラクタが増殖します。
増殖したプレイヤキャラクタも元のキャラクタと同じく、ユーザーの入力に合わせて動くので、
ボタンを押すたびにプレイヤが倍に増えていきます。（1→2→4→8→１６→３２→・・・・・）

追加したコードはプレイヤが自分の周辺に１体のプレイヤを生成する処理です。
２体に増えたプレイヤはそれぞれが１体ずつプレイヤを生成するので、ボタンを押すたびに倍増していきます。

私がテストに使用した環境では、Debugモードで実行した時はプレイヤが３２体出た時点で処理落ちが発生しました。
同じプログラムをReleaseモードで実行した場合は少なくとも４０９６体までは処理落ちが発生しませんでした。

Debugモードの実行ファイルは、プログラムを途中で停止させ、変数の中身を見たり、プログラムの流れを追えるように、デバッグ用の様々な機能が追加された状態で作られています。

スポーツに例えてみましょう。同じ選手でも、体中にセンサーや重い計測機器を取り付けられた状態では普段通りのパフォーマンスは発揮できません。それと同じです。

だから、プログラムを商品としてリリース（提供）するプログラムは基本的にReleaseモードで生成した実行ファイルです。
これを知らないのはプログラマとしては恥ずかしいことなので、これを機に覚えてください。

課題 12-1

エンディングタスクを実装し、スタートボタンを押すごとに「タイトル」→「本編」→「エンディング」→「タイトル」・・・と繰り返しシーンが遷移するように作り替えよ。
（エンディングタスクの動作は、ゲームプログラミング１の１６章の時と同じでよい）

12. 8. 描画のプライオリティ

複数のタスクがそれぞれ画像を表示するとき、その順番をコントロールするのが「描画のプライオリティ」です。

ゲーム本編では、画面全体を覆う背景画像と、プレイヤとして黄色いスライムが表示されます。
この時、黄色いスライムを先に描画してから背景を描画すると、黄色いスライムの画像は背景画像に上書きされて見えなくなってしまいます。

これを回避するために、タスクにはそれぞれ描画順を調整するための機能が実装されています。
この機能で順番を決めるのに使用されるのがプライオリティ変数です。

背景タスクのObject::Initialize( )メソッド内より抜粋

```
this->render2D_Priority[1] = 1.0f;
```

プレイヤタスクのObject::Initialize( )メソッド内より抜粋

```
this->render2D_Priority[1] = 0.5f;
```

値が大きいほど、遠くにあるものと考えて先に描画が行われます。
有効範囲は0.0～1.0までです。（それを超える範囲の値も一応指定は可能です）

プレイヤより背景の方がプライオリティの値が大きいので、先に描画されます。
プライオリティを設定しなかった場合、0.0が指定されたことになります。

背景とプレイヤのプライオリティを調整して、描画順が調整されている事を確認しましょう。
（確認したら元に戻してください）

１３．旧プログラムから下地を移植

タスクシステム導入前に実現したマップ表示やあたり判定、弾の発射などの仕組みをタスクに
置き換えて移植し、ゲームを作る下地を実装しましょう。

まず、配布フォルダから「１３章_初期状態」のプロジェクトをコピーし、プロジェクトを開いてください。
（１２章のプロジェクトからゲーム本編の背景とプレイヤタスクを取り除いただけのものです）

13．1．マップの移植

２次元配列と、３２＊３２ドットのマップチップを使ったマップを１つのタスクとして移植します。
ここからは、プログラムの全文は記載しません、ひな形からの変更内容のみ記載します。

追加タスク

ファイル名:　Task_Map2D.h　及び.cpp
ファイルタイトル：　２次元配列マップ
ネームスペース名：　Map2D
デフォルトグループ名：　"フィールド"
デフォルトタスク名：　"マップ"

追加変数（Resorce)：　なし（*マップはマップごとに違う画像を使うと考えられる為）

追加変数（Object)：
```
DG::Image::SP     img;
int               arr[8][15];
ML::Box2D         chip[16];
```

追加メソッド：
```
bool  Load(const  string&  fpath_);//マップ読み込み
bool  CheckHit(const  ML::Box2D&  hit_);//あたり判定
```

cpp 追加・変更部分のみ抜粋（以下に記載のないメソッドはひな型をそのまま残せばよい）

```
//-------------------------------------------------------------------
//「初期化」タスク生成時に１回だけ行う処理
bool  Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成orリソース共有
    this->res = Resource::Create();

    //★データ初期化
    this->render2D_Priority[1] = 0.9f;
    //マップのゼロクリア
    for (int y = 0; y < 8; ++y) {
        for (int x = 0; x < 15; ++x) {
            this->arr[y][x] = 0;
        }
    }
    //マップチップ情報の初期化
```

```
        for (int c = 0; c < 16; ++c) {
            int  x = (c % 8);
            int  y = (c / 8);
            this->chip[c] = ML::Box2D(x * 32, y * 32, 32, 32);
        }
        //★タスクの生成

        return  true;
    }
    //-------------------------------------------------------------------
    //「終了」タスク消滅時に１回だけ行う処理
    bool  Object::Finalize()
    {
        //★データ＆タスク解放
        this->img.reset();

        if (!ge->QuitFlag() && this->nextTaskCreate) {
            //★引き継ぎタスクの生成
        }
        return  true;
    }
    //-------------------------------------------------------------------
    //「２Ｄ描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
        for (int y = 0; y < 8; ++y) {
            for (int x = 0; x < 15; ++x) {
                ML::Box2D  draw(0, 0, 32, 32);
                draw.Offset(x * 32, y * 32); //表示位置を調整
                this->img->Draw(draw, this->chip[this->arr[y][x]]);
            }
        }
    }
    //-------------------------------------------------------------------
    //マップ読み込み
    bool  Object::Load(const  string&  fpath_)
    {
        //取りあえずファイルからは読み込まずに固定データをセット
        int  w_map[8][15] = {
            { 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8 },
            { 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0 },
            { 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 8 },
            { 8, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 8 },
            { 8, 0, 0, 8, 8, 0, 8, 0, 0, 8, 8, 8, 0, 0, 8 },
            { 8, 0, 0, 8, 0, 8, 0, 0, 0, 0, 8, 0, 0, 0, 0 },
            { 8, 0, 0, 8, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0 },
            { 8, 8, 0, 8, 8, 8, 8, 8, 8, 8, 8, 8, 0, 8 },
        };
        //データのコピー
```

```
        for (int y = 0; y < 8; ++y) {
            for (int x = 0; x < 15; ++x) {
                this->arr[y][x] = w_map[y][x];
            }
        }
        //マップチップ画像の読み込み
        this->img = DG::Image::Create("./data/image/MapChip01.bmp");
        return true;
    }
    //-----------------------------------------------------------------------
    //あたり判定
    bool  Object::CheckHit(const  ML::Box2D&  hit_)
    {
        ML::Rect  r = {   hit_.x,
                          hit_.y,
                          hit_.x + hit_.w,
                          hit_.y + hit_.h };
        //矩形がマップ外に出ていたらサイズを変更する
        ML::Rect  m = { 0, 0, 32 * 15, 32 * 8 };
        if (r.left < m.left) { r.left = m.left; }
        if (r.top < m.top) { r.top = m.top; }
        if (r.right > m.right) { r.right = m.right; }
        if (r.bottom > m.bottom) { r.bottom = m.bottom; }

        //ループ範囲調整
        int sx, sy, ex, ey;
        sx = r.left / 32;
        sy = r.top / 32;
        ex = (r.right - 1) / 32;
        ey = (r.bottom - 1) / 32;

        //範囲内の障害物を探す
        for (int y = sy; y <= ey; ++y) {
            for (int x = sx; x <= ex; ++x) {
                if (8 <= this->arr[y][x]) {
                    return true;
                }
            }
        }
        return false;
    }
```

多少構文は変化していますが、あたり判定の処理は８章で作成した「マップ外は障害物のない空間」である時の仕組みと同じものです。

タスクを実装したら本編統括タスクに生成処理を組み込んでマップを表示しましょう。

Task_Game．cpp 該当部分のみ抜粋

```
#include  "Task_Map2D.h"      ←ファイル先頭部分に追加
＊タスクの生成・アクセスの際には対象となるタスクのヘッダファイルの組み込みが必要です。
  以後、ヘッダファイルの組み込み指示は記載しないので各自で行ってください。
    //-------------------------------------------------------------------
    //「初期化」タスク生成時に１回だけ行う処理
    bool  Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, true);
        //リソースクラス生成 or リソース共有
        this->res = Resource::Create();

        //★データ初期化

        //★タスクの生成
        auto  map = Map2D::Object::Create(true);
        map->Load("./data/Map/map2.txt");
        return  true;
    }

    //-------------------------------------------------------------------
    //「終了」タスク消滅時に１回だけ行う処理
    bool  Object::Finalize()
    {
        //★データ＆タスク解放
        ge->KillAll_G("本編");
        ge->KillAll_G("フィールド");
        if (!ge->QuitFlag() && this->nextTaskCreate) {
            //★引き継ぎタスクの生成
            auto next = Ending::Object::Create(true);
        }

        return  true;
    }
```

これでマップが表示されます。
現時点ではマップファイルからデータは読み込んでいないので勘違いしないようにしてください。

13. 2. プレイヤの移植

プレイヤと言っても四角を表示し操作に合わせて動くだけのものですが、これも１つのタスクとして移植します。

追加タスク

ファイル名:	Task_Player.h 及び.cpp	
ファイルタイトル：	プレイヤ	
ネームスペース名：	Player	
デフォルトグループ名：	"プレイヤ"	
デフォルトタスク名：	"仮"	
追加変数（Resorce)：	DG::Image::SP	img;
追加変数（Object)：	XI::GamePad::SP	controller;
	ML::Vec2	pos; //キャラクタ位置
	ML::Box2D	hitBase; //あたり判定範囲
追加メソッド：	void CheckMove(ML::Vec2& e_); //めり込まない移動	

cpp 追加・変更部分のみ抜粋

```
//-------------------------------------------------------------------
//リソースの初期化
bool  Resource::Initialize()
{
    this->img = DG::Image::Create("./data/image/HitTest.bmp");
    return true;
}

//-------------------------------------------------------------------
//リソースの解放
bool  Resource::Finalize()
{
    this->img.reset();
    return true;
}

//-------------------------------------------------------------------
//「初期化」タスク生成時に１回だけ行う処理
bool  Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成orリソース共有
    this->res = Resource::Create();
    //★データ初期化
    this->render2D_Priority[1] = 0.5f;
    this->controller = ge->in1;
```

```
        this->pos.x = 0;
        this->pos.y = 0;
        this->hitBase = ML::Box2D(-15, -24, 30, 48);
        //★タスクの生成

        return  true;
    }

    //-------------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
        ML::Vec2  est(0, 0);
        if (this->controller) {
            auto  inp = this->controller->GetState();
            if (inp.LStick.BL.on) { est.x -= 3; }
            if (inp.LStick.BR.on) { est.x += 3; }
            if (inp.LStick.BU.on) { est.y -= 3; }
            if (inp.LStick.BD.on) { est.y += 3; }
        }
        this->CheckMove(est);
    }

    //-------------------------------------------------------------------
    //「２D描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
        ML::Box2D  draw = this->hitBase.OffsetCopy(this->pos);
        ML::Box2D  src(0, 0, 100, 100);

        this->res->img->Draw(draw, src);
    }

    //-------------------------------------------------------------------
    //めり込まない移動処理
    void Object::CheckMove(ML::Vec2&  e_)
    {
        //未実装
        this->pos += e_;//仮処理
    }
```

ここまで実装したらマップと同様にゲーム本編統括タスクにプレイヤの生成・登録を追加しましょう。

Task_Game . cpp 該当部分のみ抜粋

```
//-------------------------------------------------------------------
//「初期化」タスク生成時に１回だけ行う処理
bool  Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成 or リソース共有
    this->res = Resource::Create();

    //★データ初期化

    //★タスクの生成
    auto  map = Map2D::Object::Create(true);
    map->Load("./data/Map/map2.txt");
    //プレイヤの生成
    auto  pl = Player::Object::Create(true);
    pl->pos.x = 480 / 2;
    pl->pos.y = 270 * 2 / 3;

    return  true;
}
//-------------------------------------------------------------------
//「終了」タスク消滅時に１回だけ行う処理

bool  Object::Finalize()
{
    //★データ＆タスク解放
    ge->KillAll_G("本編");
    ge->KillAll_G("フィールド");
    ge->KillAll_G("プレイヤ");

    if (!ge->QuitFlag() && this->nextTaskCreate) {
        //★引き継ぎタスクの生成
        auto next = Ending::Object::Create(true);
    }

    return  true;
}
```

動作を確認して、本編中にプレイヤが登場し、操作できることを確認してください。
＊実行可能にするには、マップの時と同様に、Task_Game.cpp の先頭部分に「Task_Player.h」を
組み込む必要があります。忘れていませんか？

13. 3. めり込まない移動の実装

未実装状態のめり込まない移動を実装します。
これも過去の章で作ったものをタスク用に作り変えたものなので、内容はほぼ同じです。

Task_Player . cpp　該当部分のみ抜粋（既存の仮処理になっているメソッドを書き換える）

```
//-------------------------------------------------------------------
//めり込まない移動処理
void Object::CheckMove(ML::Vec2&  e_)
{
    //マップが存在するか調べてからアクセス
    auto   map = ge->GetTask <Map2D::Object>("フィールド", "マップ");
    if (nullptr == map) { return; }//マップが無ければ判定しない(出来ない)

   //横軸に対する移動
    while (e_.x != 0) {
        float  preX = this->pos.x;
        if (e_.x >= 1) {         this->pos.x += 1;        e_.x -= 1; }
        else if (e_.x <= -1) {   this->pos.x -= 1;        e_.x += 1; }
        else {                   this->pos.x += e_.x;     e_.x = 0; }
        ML::Box2D  hit = this->hitBase.OffsetCopy(this->pos);
        if (true == map->CheckHit(hit)) {
            this->pos.x = preX;       //移動をキャンセル
            break;
        }
    }
    //縦軸に対する移動
    while (e_.y != 0) {
        float  preY = this->pos.y;
        if (e_.y >= 1) {         this->pos.y += 1;        e_.y -= 1; }
        else if (e_.y <= -1) {   this->pos.y -= 1;        e_.y += 1; }
        else {                   this->pos.y += e_.y;     e_.y = 0; }
        ML::Box2D  hit = this->hitBase.OffsetCopy(this->pos);
        if (true == map->CheckHit(hit)) {
            this->pos.y = preY;       //移動をキャンセル
            break;
        }
    }
}
```

プログラムを入力したら、めり込まない移動が正しく機能することを確認してください。

以下のコードによりタスクシステムに登録されているマップを、グループ名・タスク名を使って検出しています。

```
auto   map = ge->GetTask<Map2D::Object>("フィールド", "マップ");
if (nullptr == map) { return; }//マップが無ければ判定しない(出来ない)
```

タスクが検出できなかった場合、nullptr が返るので、それを検知した場合接触判定を取りやめます。

13. 4. 着地判定の実装

着地判定を行うメソッドも実装しましょう。

Task_Player . h　Object クラス内　該当部分のみ抜粋

```
    //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
        //追加したい変数・メソッドはここに追加する
        XI::GamePad::SP    controller;
        ML::Vec2           pos;        //キャラクタ位置
        ML::Box2D          hitBase;    //あたり判定範囲
        bool               hitFlag;    //足元判定確認用

        //めり込まない移動処理
        void  CheckMove(ML::Vec2&  e_);
        //足元接触判定
        bool  CheckFoot();
    };
}
```

メソッドを追加する際には、対応したプロトタイプをヘッダに加える必要があります。
今後、この部分は教科書上では省略していくので自身で追加してください。

Task_Player . cpp　該当部分のみ抜粋

```
    //-------------------------------------------------------------------------
    //足元接触判定
    bool  Object::CheckFoot()
    {
        //あたり判定を基にして足元矩形を生成
        ML::Box2D  foot(
            this->hitBase.x,
            this->hitBase.y + this->hitBase.h,
            this->hitBase.w,
            1);
        foot.Offset(this->pos);

        auto   map = ge->GetTask<Map2D::Object>("フィールド", "マップ");
        if (nullptr == map) { return  false; }//マップが無ければ判定しない(出来ない)
        //マップと接触判定
        return map->CheckHit(foot);
    }
```

通常のあたり判定矩形から、足元に高さ１ドットの矩形を作り、マップと接触判定しています。
（やっていることは、タスク化以前のものと同じです）

これを動作確認するにあたって、新しいタスクとして「エフェクト」を作り、飛ばしてみます。

エフェクトの実装

追加タスク

ファイル名: Task_Effect00.h 及び.cpp
ファイルタイトル： エフェクト00
ネームスペース名： Effect00
デフォルトグループ名： "エフェクト"
デフォルトタスク名： "NoName"

追加変数（Resorce)： DG::Image::SP img;

追加変数（Object)： ML::Vec2 pos; //キャラクタ位置
ML::Vec2 moveVec; //移動方向ベクトル
float life; //寿命兼不透明度
float angle; //向き

追加メソッド： 無し

cpp 追加・変更部分のみ抜粋

```
//-------------------------------------------------------------------
//リソースの初期化
bool  Resource::Initialize()
{
    this->img = DG::Image::Create("./data/image/星-01.png");
    return true;
}
//-------------------------------------------------------------------
//リソースの解放
bool  Resource::Finalize()
{
    this->img.reset();
    return true;
}
//「初期化」タスク生成時に１回だけ行う処理
bool  Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成orリソース共有
    this->res = Resource::Create();

    //★データ初期化
    this->render2D_Priority[1] = 0.6f;
    this->pos = ML::Vec2(0, 0);
    this->angle = ML::ToRadian((float)(rand() % 360));
    this->moveVec = ML::Vec2(cos(angle) * 4, sin(angle) * 4);
```

```
        this->life = 1.0f;
        //★タスクの生成

        return  true;
    }
    //-----------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
        this->life -= 0.01f;
        this->pos += moveVec;
        this->moveVec.y += 0.08f;
        if (this->life < 0) {
            this->Kill();
        }
    }
    //-----------------------------------------------------------------
    //「２Ｄ描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
        ML::Box2D  draw(-32, -32, 64, 64);
        draw.Offset(this->pos);
        ML::Box2D  src(0, 0, 256, 256);
        this->res->img->Rotation(this->angle, ML::Vec2(draw.w / 2.0f, draw.h / 2.0f));
        this->res->img->Draw(draw, src, ML::Color(this->life, 1, 1, 1));
    }
```

実装出来たら、プレイヤのObject::UpDate()処理に着地判定とエフェクトの生成を組み込みます。

Task_Player . cpp 該当部分のみ抜粋

```
    //-----------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
        ML::Vec2  est(0, 0);
        if (this->controller) {
            auto  inp = this->controller->GetState();
            if (inp.LStick.BL.on) { est.x -= 3; }
            if (inp.LStick.BR.on) { est.x += 3; }
            if (inp.LStick.BU.on) { est.y -= 3; }
            if (inp.LStick.BD.on) { est.y += 3; }
        }
        this->CheckMove(est);
        //足元接触判定
        this->hitFlag = this->CheckFoot();
        if (this->hitFlag) {
```

```
            //意味もなく星をばらまいてみる
            auto  eff = Effect00::Object::Create(true);
            eff->pos = this->pos;
        }
    }
```

プレイヤの下部（足元）が障害物と接触している間、プレイヤから星のエフェクトがあふれ出すように出現し続ける事が確認できるでしょう。（１フレーム毎に１個）

タスクシステムでは、いちいち構造体の配列を作って管理する必要がありません。
好きなときにタスクを生成でき、登録するだけで各タスクが動いてくれるので、派手なエフェクトがあふれるように出るゲームも今までよりずっと作りやすくなります。

13．5．任意サイズマップに変更

マップ配列のサイズを１００×１００に拡張し、「最小１×１」から「最大１００×１００」までの任意サイズを扱えるように変更します。

Task_Map2D．h 該当部分のみ抜粋

```
    //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
        //追加したい変数・メソッドはここに追加する
        DG::Image::SP     img;
        int               arr[100][100];
        int               sizeY, sizeX;
        ML::Box2D         hitBase;//ピクセル単位のマップサイズを持つ
        ML::Box2D         chip[16];
        bool  Load(const  string&  fpath_);//マップ読み込み
        bool  CheckHit(const  ML::Box2D&  hit_);//あたり判定
    };
}
```

Task_Map2D．cpp 該当部分のみ抜粋

```
    //-------------------------------------------------------------------------
    //「初期化」タスク生成時に１回だけ行う処理
    bool  Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, true);
        //リソースクラス生成orリソース共有
        this->res = Resource::Create();
        //★データ初期化
        this->render2D_Priority[1] = 0.9f;
        //マップのゼロクリア
```

```
        for (int y = 0; y < 100; ++y) {
            for (int x = 0; x < 100; ++x) {
                this->arr[y][x] = 0;
            }
        }
        this->sizeX = 0;
        this->sizeY = 0;
        this->hitBase = ML::Box2D(0, 0, 0, 0);
        //マップチップ情報の初期化
        for (int c = 0; c < 16; ++c) {
            int  x = (c % 8);
            int  y = (c / 8);
            this->chip[c] = ML::Box2D(x * 32, y * 32, 32, 32);
        }
        //★タスクの生成

        return  true;
    }
    //-------------------------------------------------------------------
    //「２D描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
        for (int y = 0; y < this->sizeY; ++y) {
            for (int x = 0; x < this->sizeX; ++x) {
                ML::Box2D  draw(0, 0, 32, 32);
                draw.Offset(x * 32, y * 32); //表示位置を調整
                this->img->Draw(draw, this->chip[this->arr[y][x]]);
            }
        }
    }
    //-------------------------------------------------------------------
    //マップ読み込み
    bool  Object::Load(const  string&  fpath_)
    {
        //取りあえずファイルからは読み込まずに固定データをセット
        int  w_map[8][15] = {
            { 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8 },
            { 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 0 },
            { 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 8 },
            { 8, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0, 0, 0, 0, 8 },
            { 8, 0, 0, 8, 8, 0, 8, 0, 0, 8, 8, 8, 0, 0, 8 },
            { 8, 0, 0, 8, 0, 8, 0, 0, 0, 0, 8, 0, 0, 0, 0 },
            { 8, 0, 0, 8, 0, 0, 0, 0, 0, 0, 8, 0, 0, 0, 0 },
            { 8, 8, 0, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 0, 8 },
        };
        //データのコピー
        for (int y = 0; y < 8; ++y) {
            for (int x = 0; x < 15; ++x) {
                this->arr[y][x] = w_map[y][x];
```

```
            }
        }
        this->sizeX = 15;
        this->sizeY = 8;
        this->hitBase = ML::Box2D(0, 0, this->sizeX * 32, this->sizeY * 32);
        //マップチップ画像の読み込み
        this->img = DG::Image::Create("./data/image/MapChip01.bmp");
        return true;
    }
    //-------------------------------------------------------------------
    //あたり判定
    bool  Object::CheckHit(const  ML::Box2D&  hit_)
    {
        ML::Rect  r = {
            hit_.x,
            hit_.y,
            hit_.x + hit_.w,
            hit_.y + hit_.h };
        //矩形がマップ外に出ていたらサイズを変更する
        ML::Rect  m = {
            this->hitBase.x,
            this->hitBase.y,
            this->hitBase.x + this->hitBase.w,
            this->hitBase.y + this->hitBase.h
        };
        if (r.left   < m.left) { r.left = m.left; }
        if (r.top    < m.top) { r.top = m.top; }
        if (r.right  > m.right) { r.right = m.right; }
        if (r.bottom > m.bottom) { r.bottom = m.bottom; }
        //ループ範囲調整
        int sx, sy, ex, ey;
        sx = r.left / 32;
        sy = r.top / 32;
        ex = (r.right - 1) / 32;
        ey = (r.bottom - 1) / 32;
        //範囲内の障害物を探す
        for (int y = sy; y <= ey; ++y) {
            for (int x = sx; x <= ex; ++x) {
                if (8 <= this->arr[y][x]) {
                    return true;
                }
            }
        }
        return false;
    }
```

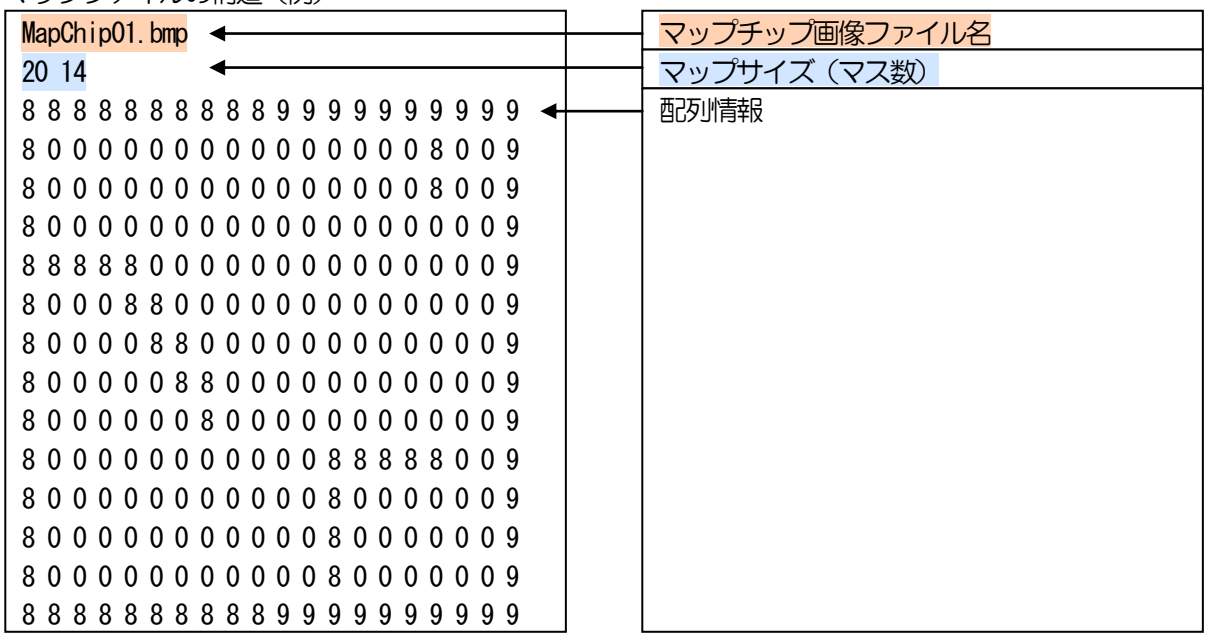
この時点では見た目が全く変わりません。

マップタスクの持つhitBaseはドット単位のマップ全体を覆う矩形です。
マップのサイズが１０＊８マスなら、hitBaseのwは３２０、hは２５６として設定されます。
（Load( )関数を実行した時設定される）

13. 6. マップファイルからデータを読み込む

外部ファイルから任意サイズのマップデータを読み込めるようにします。
そのマップで使用するマップチップ画像ファイルも、マップファイルから読み込んだファイル名を使って読み込みを行います。

マップファイルの構造（例）



配布フォルダからMap2.txtをコピーし内容を確認したのち、
./data/Map/フォルダ内に追加してください。（フォルダが無ければ作ってください）

ファイルを追加したら、Load( )メソッドを変更し、外部ファイルからの読み込み機能を実装します。

Task_Map2D．cpp 該当部分のみ抜粋

```
//-----------------------------------------------------------------
//マップ読み込み
bool  Object::Load(const  string&  fpath_)
{
    関数内の処理をすべて消して以下のコードを入力する
    //ファイルを開く（読み込み）
    ifstream   fin(fpath_);
    if (!fin) { return  false; }//読み込み失敗

    //チップファイル名の読み込みと、画像のロード
    string   chipFileName, chipFilePath;
    fin >> chipFileName;
    chipFilePath = "./data/image/" + chipFileName;
    this->img = DG::Image::Create(chipFilePath);

    //マップ配列サイズの読み込み
    fin >> this->sizeX >> this->sizeY;
    this->hitBase = ML::Box2D(0, 0, this->sizeX * 32, this->sizeY * 32);

    //マップ配列データの読み込み
    for (int y = 0; y < this->sizeY; ++y) {
        for (int x = 0; x < this->sizeX; ++x) {
            fin >> this->arr[y][x];
        }
    }
    fin.close();
    return true;
}
```

仮処理を削除し、ファイルからのデータ読み込み処理を実装しました。
読み込む対象のマップファイルは４０×２８マスのデータを持っているので、これを表示しても画面には収まりきりません。

ファイルからのデータ読み込みには ifstrem を使用しています。　データの読み出しには＞＞を使用します。（詳しく知りたい人は C++を学習してください）

動作確認をして Map2.txt ファイルから読み込んだマップが表示されることを確認してください。

14. スクロール

14. 1. ドット単位スクロール

キャラクタを画面の中心に据えて、キャラクタが動くことによってステージ上の表示範囲が変化するスクロールの仕組みは自分で考えて実現するのはかなり難しいものですが、きちんと準備をしたうえで段階を経て実装すればそれほど難しくはありません。

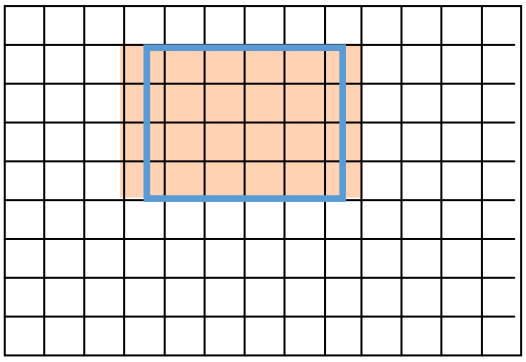
まず「ゲーム空間」と「画面」を分けて考える事に慣れましょう。
キャラクタは「ゲーム空間」内にいます。
キャラクタだけでなく、アイテムやエフェクト、マップ（ステージ）も全てゲーム空間内にあると考えます。

画面を通して映る映像は、カメラを介してゲーム空間の一部が映し出されているものと考えられます。
カメラを動かして、写す範囲を変えたとしても、ゲーム空間にあるものは何も影響を受けません。

カメラを右に動かしたら止まっているキャラクタは画面上を左に移動していくでしょうが、
それはあくまでもカメラを動かしただけの話で、そのキャラクタは左には動いていないのです。

「カメラの動きはゲーム空間上にあるものに一切の影響を与えない」これは重要なポイントです。

まず、画面の表示範囲に対応する矩形を用意します。これをカメラが映し出す範囲と考えます。
画面の大きさが500×400あったとするなら、この矩形の大きさも500×400にします。
この矩形の左上が、座標（350, 100）を刺していた場合、以下の図のような範囲を指していると考えられます。



１マスあたり
100×100 ドット
であるとした場合の、
ML::Box2D(350, 100, 500, 400);
が指す範囲

この範囲に含まれるものが画面上に表示されれば良いのです。
例えば、ゲーム空間上の座標（400, 200）にキャラクタがいたとしましょう。
そのキャラクタは今の状況だと画面上のどこにいることになりますか？

画面の左上（0, 0）にゲーム空間の座標（350, 100）にあるものが表示されるのであれば、
そのキャラクタは画面上の座標（50, 100）にいると考えられます。

同様に、ゲーム空間の座標(600, 340)にいるキャラクタは、画面上の座標では（250, 240）にいると考えられます。

つまり、ゲーム空間上の座標からカメラ矩形の左上座標を引けば、スクロールに対応した画面上の表示位置を求めることができるということです。

この表示位置を求める対象となるものは「ゲーム空間上に存在するすべてのモノ」です。
プレイヤキャラクタ、敵キャラクタ、アイテム、エフェクト、マップ、etc..
ゲーム空間上にあるすべてのモノが対象ですから、マップも対象です。
画面上の固定位置に表示される得点やライフゲージ等は、ゲーム空間上には存在しないと考えます。
（つまり、スクロールの対象にはならない）

勘違いしないでほしいのですが、ゲーム空間上の座標から画面座標に変換するのはキャラクタそのものの座標ではなく「画像の表示座標」だけです。

つまり、表示処理の中で宣言している変数 draw を書き換えるだけです。

まず、強制スクロールを実現して、ここまでの内容を確認していきましょう。

MyPG . h 該当部分のみ抜粋　（該当箇所はファイルの終端）

```
    //ゲームエンジンに追加したいものは下に加える
    //-------------------------------------------
        MyPG::Camera::SP          camera[4];     //  カメラ
        XI::Mouse::SP             mouse;
        XI::GamePad::SP           in1, in2, in3, in4;   //取り合えず4本
        //2Dカメラ矩形
        ML::Box2D                 camera2D;  //  2Dスクロール制御用
    //-------------------------------------------
    };
}
extern MyPG::MyGameEngine* ge;
```

Task_Game . cpp　該当部分のみ抜粋（Object::Initialize( )メソッド内）

```
    //★データ初期化
    ge->camera2D = ML::Box2D(-200, -100, 480, 270);//取りあえず初期値設定
```

Task_Game . cpp　該当部分のみ抜粋（Object::UpDate ( )メソッド内）

```
    //とりあえず強制スクロール
    ge->camera2D.x += 2;
    ge->camera2D.y += 1;
```

ここまでで、カメラの枠矩形と、それを時間の経過に合わせて移動させる仕組みは実装しました。
あとは、キャラクタやマップの表示処理の中に draw 矩形を書き換える処理を加えるだけです。

Task_Player . cpp 該当部分のみ抜粋

```
//-----------------------------------------------------------------
//「２D描画」１フレーム毎に行う処理
void  Object::Render2D_AF()
{
    ML::Box2D  draw = this->hitBase.OffsetCopy(this->pos);
    ML::Box2D  src(0, 0, 100, 100);
    //スクロール対応
    draw.Offset(-ge->camera2D.x, -ge->camera2D.y);
    this->res->img->Draw(draw, src);
}
```

Task_Map2D . cpp　該当部分のみ抜粋

```
//-----------------------------------------------------------------
//「２D描画」１フレーム毎に行う処理
void  Object::Render2D_AF()
{
    for (int y = 0; y < this->sizeY; ++y) {
        for (int x = 0; x < this->sizeX; ++x) {
            ML::Box2D  draw(0, 0, 32, 32);
            draw.Offset(x * 32, y * 32); //表示位置を調整
            //スクロール対応
            draw.Offset(-ge->camera2D.x, -ge->camera2D.y);
            this->img->Draw(draw, this->chip[this->arr[y][x]]);
        }
    }
}
```

課題 14-1

動作確認しやすいようにマップを編集したのち、着地した時に足元から出る星のエフェクトの位置がズレている事を確認せよ。
そのうえで、星のエフェクトもスクロールに対応するように作り替えたプログラムを提出せよ。

14．２．画面外へのマップチップの描画処理を防止する

マップ描画のループを調整することで画面外にチップを描こうとする無駄な処理を防止します。
画面サイズが４８０×２７０なので、３２×３２ドットのチップは、横に１５枚、縦に8枚（実質7．５枚）だから、１５×８回の２重ループにすればよいという考え方は正解ではありません。

１５枚というのは、画面枠とマップチップがぴったりそろったときの枚数であり、画面端に表示されるチップが一部分だけ画面内にある時は１６枚表示する必要があるからです。

では１６回の繰り返しにすればよいかというと、そういう話でもありません。
重要なのは「カメラ枠の矩形が重なっているマップ配列の範囲を特定すること」そして「その範囲だけでループを回してマップチップを描画する事」です。

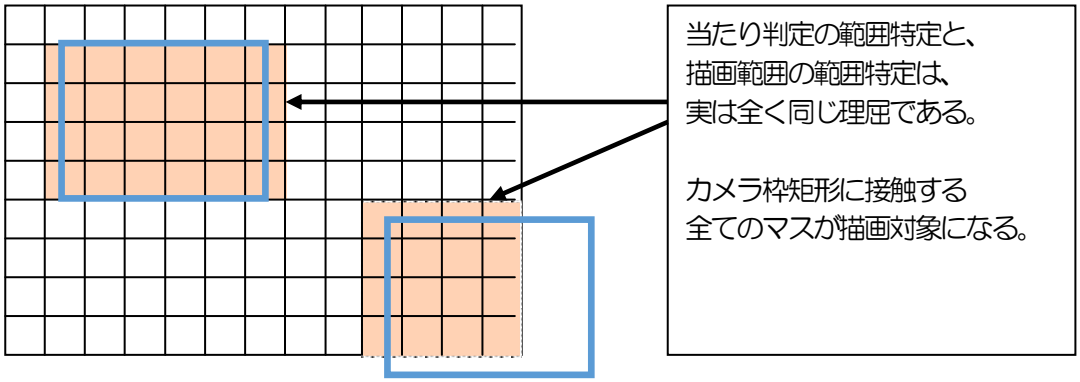
では、どうすれば「カメラ枠の矩形が重なっているマップ配列の範囲を特定すること」ができるでしょうか？

・・・

どうすれば「矩形とマップ配列が重なっているところを特定すること」ができますか？

・・・

矩形とマップ配列の接触判定は8章ですでに実現しています。
その仕組みを使えば、カメラ枠の矩形とマップ配列の重なる範囲を特定することができます。



この仕組みをマップの表示部分にも組み込み、ループの範囲を限定することで無駄な描画を極力減らすことができます。

Task_Map2D．cpp　該当部分のみ抜粋（既存の関数を作り替える、変更部分が多いので注意）

```
//-----------------------------------------------------------------
//「２D描画」１フレーム毎に行う処理
void  Object::Render2D_AF()
{
    //カメラが完全にマップ外を指しているか調べる
    if (false == this->hitBase.Hit(ge->camera2D)) {
        return; //完全に外に出ていたらその時点で描画処理を取りやめる
    }
```

```
        //カメラとマップが重なっている範囲だけの矩形を作る
        ML::Rect  c = {
            ge->camera2D.x,
            ge->camera2D.y,
            ge->camera2D.x + ge->camera2D.w,
            ge->camera2D.y + ge->camera2D.h };
        ML::Rect  m = {
            this->hitBase.x,
            this->hitBase.y,
            this->hitBase.x + this->hitBase.w,
            this->hitBase.y + this->hitBase.h };

        //２つの矩形の重なっている範囲だけの矩形を求める
        ML::Rect  isr;
        isr.left = max(c.left, m.left);
        isr.top = max(c.top, m.top);
        isr.right = min(c.right, m.right);
        isr.bottom = min(c.bottom, m.bottom);

        //ループ範囲を決定
        int sx, sy, ex, ey;
        sx = isr.left / 32;
        sy = isr.top / 32;
        ex = (isr.right - 1) / 32;
        ey = (isr.bottom - 1) / 32;

        //画面内の範囲だけ描画
        for (int y = sy; y <= ey; ++y) {
            for (int x = sx; x <= ex; ++x) {
                ML::Box2D  draw(0, 0, 32, 32);
                draw.Offset(x * 32, y * 32); //表示位置を調整
                //スクロール対応
                draw.Offset(-ge->camera2D.x, -ge->camera2D.y);
                this->img->Draw(draw, this->chip[this->arr[y][x]]);
            }
        }
    }
```

見た目に変化が無いので確認するのは難しいですが、これにより大分無駄が解消されます。

14．3．プレイヤキャラクタを中心としたスクロール

強制スクロールを取りやめ、プレイヤキャラクタを画面中央に据えたスクロールを実装します。
まずは壁際のスクロール停止機能を含まない状態を作ります。

Task_Game . cpp 該当部分のみ抜粋　（以下のプログラムを削除する）

```
    //とりあえず強制スクロール
    ge->camera2D.x += 2;
    ge->camera2D.y += 1;
```

Task_Player . cpp 該当部分のみ抜粋

```
    //-----------------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
        ML::Vec2  est(0, 0);
        if (this->controller) {
            auto  inp = this->controller->GetState();
            if (inp.LStick.BL.on) { est.x -= 3; }
            if (inp.LStick.BR.on) { est.x += 3; }
            if (inp.LStick.BU.on) { est.y -= 3; }
            if (inp.LStick.BD.on) { est.y += 3; }
        }
        this->CheckMove(est);
以下は削除する
        //足元接触判定
        this->hitFlag = this->CheckFoot();

        if (this->hitFlag) {
            //意味もなく星をばらまいてみる
            auto  eff = Effect00::Object::Create(true);
            eff->pos = this->pos;
        }
        //カメラの位置を再調整
        {
            //プレイヤを画面の何処に置くか（今回は画面中央）
            int  px = ge->camera2D.w / 2;
            int  py = ge->camera2D.h / 2;
            //プレイヤを画面中央に置いた時のカメラの左上座標を求める
            int  cpx = int(this->pos.x) - px;
            int  cpy = int(this->pos.y) - py;
            //カメラの座標を更新
            ge->camera2D.x = cpx;
            ge->camera2D.y = cpy;
        }
    }
```

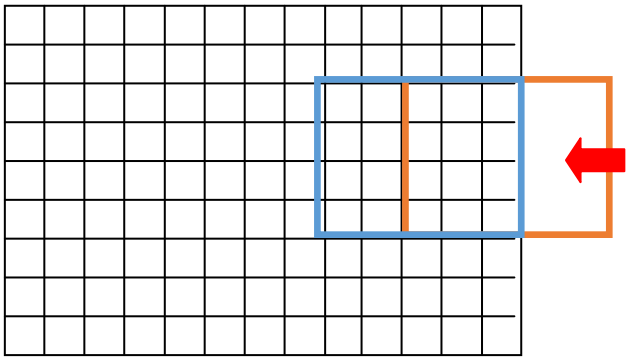
プレイヤキャラクタを画面の中央に据えるためには、カメラ枠の座標をプレイヤキャラクタの座標から、(-240, -135) した座標に設定する必要があります。（画面サイズの半分を引いた座標）

プログラムを実行して、プレイヤキャラクタの移動にあわせてスクロールする事を確認してください。

14．4．マップ端のスクロール停止

プレイヤキャラクタが移動してマップ端に近づいたとき、スクロールを停止させる仕組みを追加します。
この制御を加えれば、ドット単位のスクロール制御も終了です。

理屈は簡単で、カメラ枠の矩形がマップから飛び出したら内側に強制移動して戻すだけです。



マップにカメラ枠の矩形位置を調整する仕組みを追加します。

Task_Map2D．cpp 追加メソッドのみ記載

```
//-------------------------------------------------------------------
//マップ外を見せないようにカメラを位置調整する
void  Object::AdjustCameraPos()
{
    //カメラとマップの範囲を用意
    ML::Rect  c = {
        ge->camera2D.x,
        ge->camera2D.y,
        ge->camera2D.x + ge->camera2D.w,
        ge->camera2D.y + ge->camera2D.h };
    ML::Rect  m = {
        this->hitBase.x,
        this->hitBase.y,
        this->hitBase.x + this->hitBase.w,
        this->hitBase.y + this->hitBase.h };

    //カメラの位置を調整
    if (c.right > m.right) { ge->camera2D.x = m.right - ge->camera2D.w; }
    if (c.bottom > m.bottom) { ge->camera2D.y = m.bottom - ge->camera2D.h; }
    if (c.left < m.left) { ge->camera2D.x = m.left; }
    if (c.top < m.top) { ge->camera2D.y = m.top; }
    //マップがカメラより小さい場合
    if (this->hitBase.w < ge->camera2D.w) { ge->camera2D.x = m.left; }
    if (this->hitBase.h < ge->camera2D.h) { ge->camera2D.y = m.top; }
}
```

メソッドの追加ですから、.hファイルへのプロトタイプ追加を忘れないでください。
先にプロトタイプを追加することを心掛けておけば、VisuakStudioのインテリセンスが機能して
メンバ変数が自動表示されるので、コードが書きやすくなる筈です。

追加したメソッドをプレイヤの移動後に使用することで、カメラ位置の調整が機能するようになり、
マップの外が見えないようになります。

Task_Player . cpp 該当部分のみ抜粋

```
    //カメラの位置を再調整
    {
        //プレイヤを画面の何処に置くか（今回は画面中央）
        int  px = ge->camera2D.w / 2;
        int  py = ge->camera2D.h / 2;
        //プレイヤを画面中央に置いた時のカメラの左上座標を求める
        int  cpx = int(this->pos.x) - px;
        int  cpy = int(this->pos.y) - py;
        //カメラの座標を更新
        ge->camera2D.x = cpx;
        ge->camera2D.y = cpy;
        //マップの外側が映らないようにカメラを調整する
        if (auto map = ge->GetTask<Map2D::Object>("フィールド", "マップ")) {
            map->AdjustCameraPos();
        }
    }
```

これでドット単位スクロールは終了です。

課題 14-2

ここまでの内容を実装したプログラムを提出せよ。

＊タスク検出の際に、グループ名やタスク名に対して文字列を直接書く事に抵抗があり、対象の名称は
デフォルト名から変更されていない事もわかっている場合、以下のように書くこともできる。

```
if (auto map = ge->GetTask<Map2D::Object>(Map2D::defGroupName, Map2D::defName)) {
```

この方法であれば「微妙に名前が違って検出できなかった」「デフォルト名を変更したら検出できなく
なった」等の問題が起こらないため、不都合がないなら積極的に使用したほうが良い。

15. 弾を撃つ

15. 1. 弾タスク

タスクシステムを使ったゲームでは、弾１発も１つのタスクとして作ります。
まずは、発生させると自動で飛んでいき、壁に当たるか一定時間で自滅する「弾タスク」を作ります。

追加タスク
ファイル名: Task_Shot00.h 及び.cpp
ファイルタイトル： プレイヤの出す弾
ネームスペース名： Shot00
デフォルトグループ名： "弾（プレイヤ）"
デフォルトタスク名： "NoName"

追加変数（Resorce)： DG::Image::SP img;

追加変数（Object)：
```
ML::Vec2        pos;      //キャラクタ位置
ML::Box2D       hitBase;  //あたり判定範囲
ML::Vec2        moveVec;  //移動ベクトル
int             moveCnt;  //行動カウンタ
```

追加メソッド： 無し

cpp 追加・変更部分のみ抜粋

```
//-------------------------------------------------------------------
//リソースの初期化
bool  Resource::Initialize()
{
    this->img = DG::Image::Create("./data/image/Shot00.png");
    return true;
}

//-------------------------------------------------------------------
//リソースの解放
bool  Resource::Finalize()
{
    this->img.reset();
    return true;
}

//-------------------------------------------------------------------
//「初期化」タスク生成時に１回だけ行う処理
bool  Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成orリソース共有
```

```
        this->res = Resource::Create();
        //★データ初期化
        this->render2D_Priority[1] = 0.4f;
        this->pos.x = 0;
        this->pos.y = 0;
        this->hitBase = ML::Box2D(-8, -8, 16, 16);
        this->moveVec = ML::Vec2(0, 0);
        this->moveCnt = 0;
        //★タスクの生成

        return  true;
    }

    //-------------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
        this->moveCnt++;
        //限界の時間を迎えたら消滅
        if (this->moveCnt >= 30) {
            //消滅申請
            this->Kill();
            return;
        }

        //移動
        this->pos += this->moveVec;
        //移動先で障害物に接触したら消滅
        //マップが存在するか調べてからアクセス
        if (auto  map = ge->GetTask<Map2D::Object>("フィールド", "マップ")) {
            ML::Box2D  hit = this->hitBase.OffsetCopy(this->pos);
            if (true == map->CheckHit(hit)) {
                //消滅申請
                this->Kill();
                //とりあえず星はばら撒くよ
                for (int c = 0; c < 4; ++c) {
                    auto  eff = Effect00::Object::Create(true);
                    eff->pos = this->pos;
                }
                return;
            }
        }

        //敵対象と衝突判定＆ダメージを与える処理
        //＊未実装＊
    }
```

```
//-------------------------------------------------------------------
//「２D描画」１フレーム毎に行う処理
void  Object::Render2D_AF()
{
    ML::Box2D  draw(-8, -8, 16, 16);
    draw.Offset(this->pos);
    ML::Box2D  src(0, 0, 32, 32);

    //スクロール対応
    draw.Offset(-ge->camera2D.x, -ge->camera2D.y);
    this->res->img->Draw(draw, src);
}
```

弾は壁に衝突すると消滅するため、プレイヤのようにめり込まない移動処理は行いません。

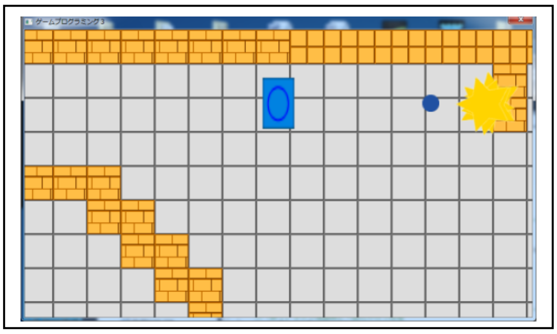
弾タスクが実装出来たらプレイヤの行動処理の中に B3 ボタン（C キー）の入力に応じて弾を出現させる処理を追加します。

Task_Player . cpp 該当部分のみ抜粋

```
//-------------------------------------------------------------------
//「更新」１フレーム毎に行う処理
void  Object::UpDate()
{
    ML::Vec2  est(0, 0);
    if (this->controller) {
        auto  inp = this->controller->GetState();
        if (inp.LStick.BL.on) { est.x -= 3; }
        if (inp.LStick.BR.on) { est.x += 3; }
        if (inp.LStick.BU.on) { est.y -= 3; }
        if (inp.LStick.BD.on) { est.y += 3; }
        //弾を撃つ
        if (inp.B3.down) {
            //弾を生成
            auto  shot = Shot00::Object::Create(true);
            shot->pos = this->pos;
            shot->moveVec = ML::Vec2(8, 0);
        }
    }
    this->CheckMove(est);
    //足元接触判定
    this->hitFlag = this->CheckFoot();

    //カメラの位置を再調整
    {
        以下省略
```

Ｂ３ボタン（Ｃキー）を押すと弾が発射されます。

壁にぶつかるか、時間経過で消滅することを確認してください。



＊弾が壁にぶつかって消滅する時のみ、星を４つ出現させます。

15. 2. 向きに合わせて弾を飛ばす

プレイヤキャラクタの向きに合わせて、弾を飛ばす方向を変えてみましょう。
とは言っても、現時点でプレイヤキャラクタはそもそも向きの情報を持っていないので、向きの追加も含めて実装します。

Task_Player．h 該当部分のみ抜粋

```
    //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
        //追加したい変数・メソッドはここに追加する
        XI::GamePad::SP    controller;
        ML::Vec2           pos;        //キャラクタ位置
        ML::Box2D          hitBase;    //あたり判定範囲
        bool               hitFlag;    //足元判定確認用
        //左右の向き（2D 横視点ゲーム専用）
        enum class Angle_LR { Left, Right };
        Angle_LR   angle_LR;
        //めり込まない移動処理
        void  CheckMove(ML::Vec2&  e_);
        //足元接触判定
        bool  CheckFoot();
    };
}
```

Task_Player．cpp 該当部分のみ抜粋

```
Object::Initialize()  メソッド内に追加
    this->angle_LR = Angle_LR::Right;
```

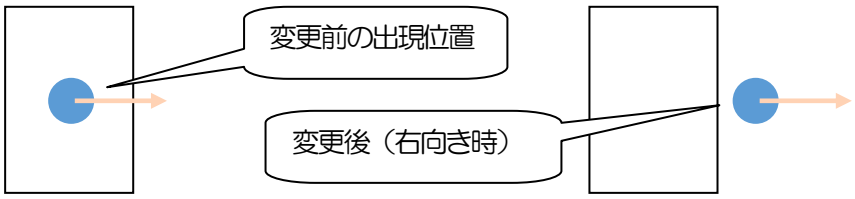
```
//-------------------------------------------------------------------
//「更新」１フレーム毎に行う処理
void  Object::UpDate()
{
    ML::Vec2  est(0, 0);
    if (this->controller) {
```

```
        auto  inp = this->controller->GetState();
        if (inp.LStick.BL.on) { est.x -= 3; this->angle_LR = Angle_LR::Left;     }
        if (inp.LStick.BR.on) { est.x += 3; this->angle_LR = Angle_LR::Right;    }
        if (inp.LStick.BU.on) { est.y -= 3; }
        if (inp.LStick.BD.on) { est.y += 3; }
        //弾を撃つ
        if (inp.B3.down) {
            //弾を生成
            auto  shot = Shot00::Object::Create(true);
            shot->pos = this->pos;
            if (this->angle_LR == Angle_LR::Right) {
                shot->moveVec = ML::Vec2(8, 0);
            }
            else {
                shot->moveVec = ML::Vec2(-8, 0);
            }
        }
    }
    以下省略
```

プレイヤの向きに合わせて弾の移動ベクトルを変更することで、弾の飛ぶ方向を変更しています。
（今のプレイヤには見た目の向きはありません）

課題 15-1

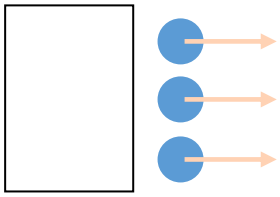
弾の出現位置をプレイヤの中心ではなく、向きに合わせて３０ドット離れた位置に変更せよ。



＊ここで悩んでしまう者は、まず右向き限定で弾の出る位置を調整する方法を考えよ。

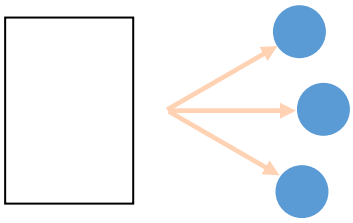
課題 15-2

弾を一度に３発発射するように変更せよ。（左右どちらにも撃てること）



課題 15-3

3発の弾を3方向に飛ぶように変更せよ。（左右どちらにも撃てること）
（3方向に分かれれば、弾の速度は同じでなくとも良い）

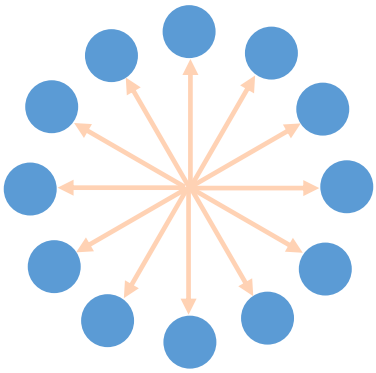


課題 15-SP1

３０度刻みで、１２方向に弾が飛ぶように変更せよ。　cos( )　sin( )を使用する必要あり

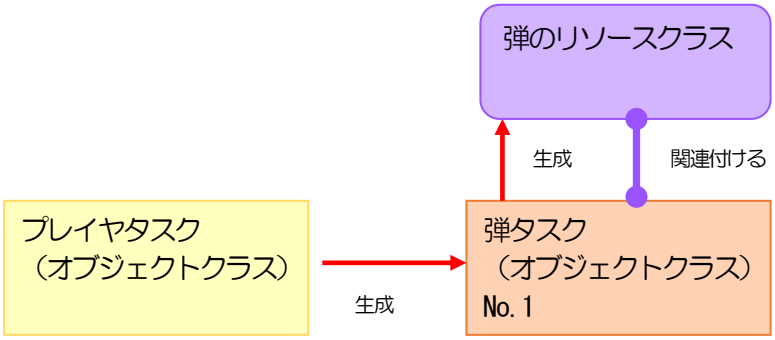
- 弾の速度は全て同じで円が広がるように飛ばすこと。
- 弾の速度は１フレーム当たり４ドット分以上とする。

＊この課題は、弾の移動ベクトルの設定だけで対応できる。Task_Shit00 系ファイルを変更する必要はない。

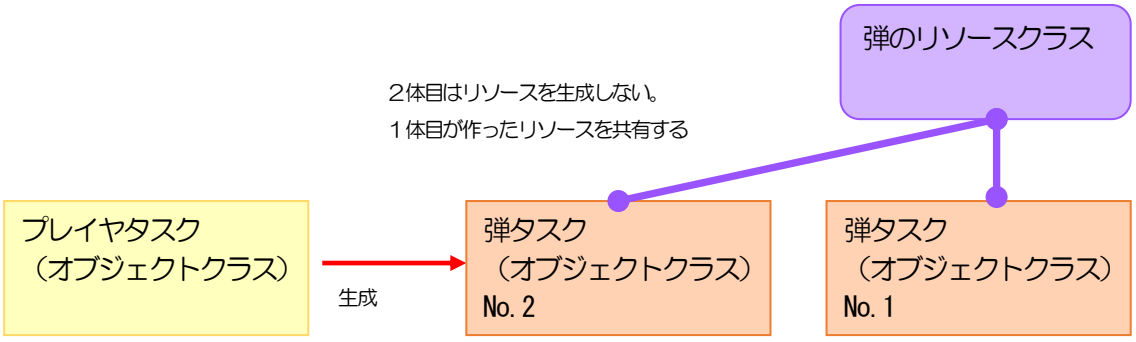


15．3．頻繁に生成・消滅を繰り返すタスクのリソースを常駐させる（処理の効率化①）

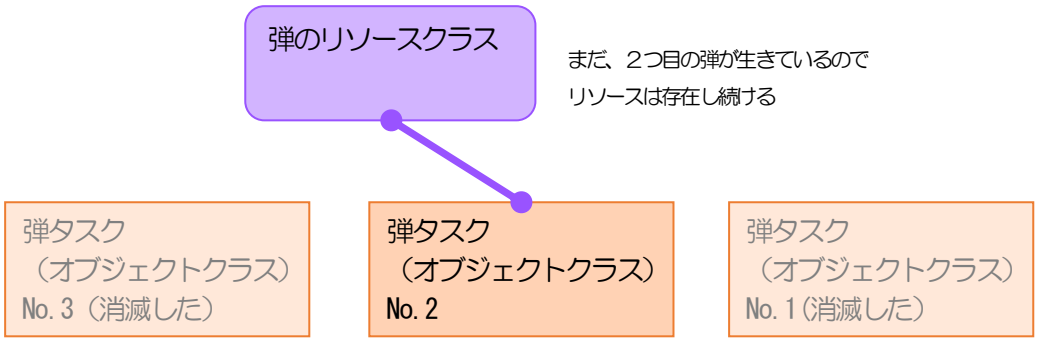
１つ目のタスク（オブジェクトクラス）が生成されたとき、リソースクラスも生成されます。



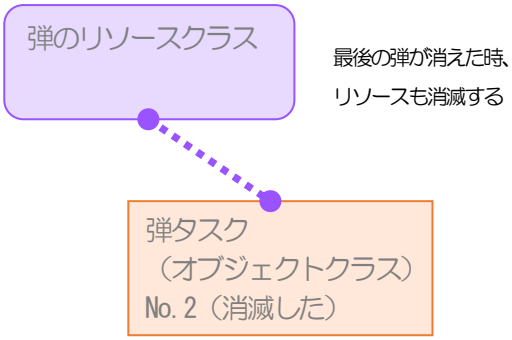
２つ目のタスクが生成された時には、１つ目のタスクが生成したリソースクラスが２つ目のタスクにも共有（シェア）されます。
以降何体タスクが生成されてもリソースクラスは最初の１体が生成したものが共有されます。



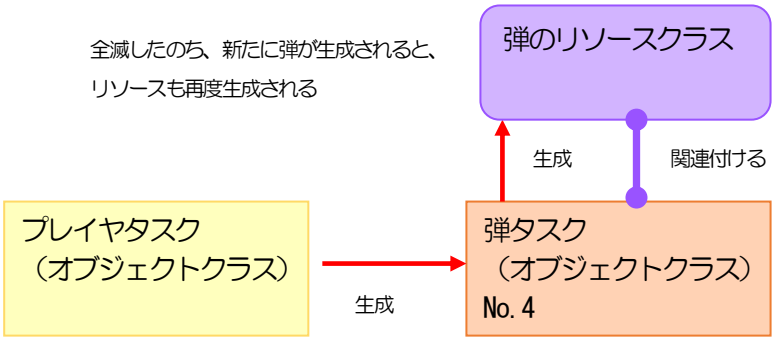
同種のタスクが１つでも生き残っていれば、それが何番目に生成されたものであれ、
リソースクラスは存在し続けます。



同種のタスクが全て消滅した時、リソースクラスも消滅します。

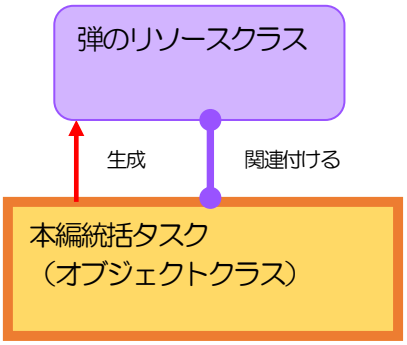


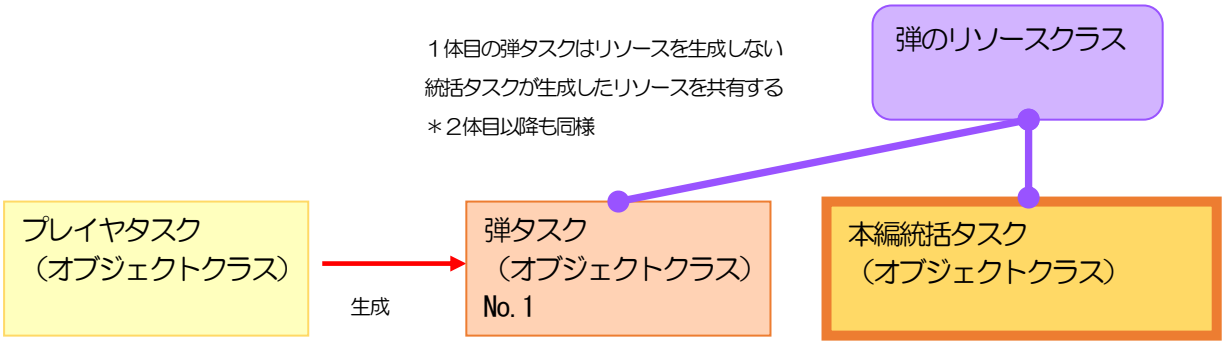
一度同種のタスクが全滅したのち、再度タスクが生成されると、リソースクラスも再生成されます。



リソースを使用するタスクがいなくなった時、リソースを解放してくれるので、これはこれで理に適った仕組みなのですが、弾のように頻繁に生成され、頻繁に全消滅が起こるタスクの場合、リソースの準備（ファイルからの読み込み等、処理に時間がかかるもの）も頻繁に行われるようになるため、ゲームを遅延させてしまうなど、不都合を生じる場合があります。
（現状特に遅延を感じないのはデータ量が少ないのと、キャッシュが機能している影響だと思います）

このような状況になる事がわかっている場合、あらかじめゲーム本編統括タスク（シーンの統括タスク）に対象のリソースクラスを生成させ、関連付けしておくことで、ゲーム本編が終了するまでリソースが開放されなくなるようにする事が出来ます。





本編統括タスクはゲーム本編が終了するまで消滅しないので、その間リソースクラスの生存は保証されます。

今回は「プレイヤの弾タスク」と、「星のエフェクトタスク」のリソースクラスをこの対象としてみます。

Task_Game . h　　該当箇所のみ抜粋

```
        //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
        //追加したい変数・メソッドはここに追加する
        vector<BResource::SP>  residentResorce;
    };
}
```

追加した宣言は、「BResorce クラスへのシェアポインタ型」の可変長配列を管理する変数です。
BResource クラスは、BTask クラスと同様に、各タスクのリソースクラスの親クラスです。
Vector を使用し可変長配列を使用することで、あとから同様にリソース常駐させたいものが出てきたとき、追記するコードを減らすことができます。

この変数（residentResorce）が消滅するまで登録したリソースクラスは存在し続けます。
（residentResorce が消滅するのは、ゲーム本編が終了したときです）

＊シェアポインタを理解するには、先に「ポインタ」「クラス」「テンプレート」の理解が必要です。

cpp 側に、リソースを常駐させる処理を組み込みます。

Task_Game . cpp 該当部分のみ抜粋

ファイル先頭部分に追加

```
#include  "Task_Effect00.h"
#include "Task_Shot00.h"
```

```
    //-----------------------------------------------------------------------
    //「初期化」タスク生成時に１回だけ行う処理
    bool  Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, true);
```

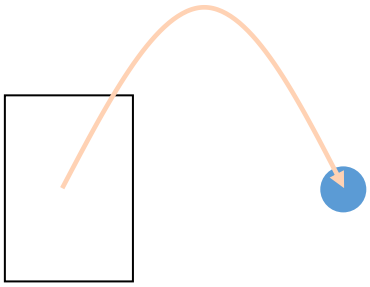
```
        //リソースクラス生成 or リソース共有
        this->res = Resource::Create();

        //★データ初期化
        ge->camera2D = ML::Box2D(-200, -100, 480, 270);//取りあえず初期値設定
        //リソースを常駐させる
        this->residentResorce.push_back(Shot00::Resource::Create());
        this->residentResorce.push_back(Effect00::Resource::Create());

        //★タスクの生成
        auto  map = Map2D::Object::Create(true);
        map->Load("./data/Map/map2.txt");

        auto  pl = Player::Object::Create(true);
        pl->pos.x = 480 / 2;
        pl->pos.y = 270 * 2 / 3;

        return  true;
    }
//---------------------------------------------------------------------
    //「終了」タスク消滅時に１回だけ行う処理
    bool  Object::Finalize()
    {
        //★データ＆タスク解放
        ge->KillAll_G("本編");
        ge->KillAll_G("フィールド");
        //リソースの常駐を解除する（書かなくても勝手に解除される）
        this->residentResorce.clear();
        if (!ge->QuitFlag() && this->nextTaskCreate) {
            //★引き継ぎタスクの生成
            auto next = Ending::Object::Create(true);
        }
        return  true;
    }
```

```
this->residentResorce.push_back(Shot00::Resource::Create());
                              |                 └リソースクラスの生成（戻り値でシェアポインタ①）
                              |
                              └可変長配列のサイズを増やし、その配列の終端に①を追加する。
```

これで「プレイヤの弾」と「星のエフェクト」のタスクのリソースは、ゲーム本編実行中は常に常駐するようになります。

15. 4. 放物線を描く弾を飛ばす

放物線を描きながら飛ぶ弾を作ります。
まずは、Shot00 をそのままコピーした
Shot01 タスクを作成・追加してください。



B4 ボタン（V キー）を押したとき、Shot01 を生成する
プログラムを実装し、弾の発射時に与えるベクトルを
(±5,-6)にします。
（Player::Object::UpDate 内）

そのままでは、弾は斜め上に飛んでいくだけです。
この弾タスクに重力加速の処理を加えて、放物線を描くように変更します。

Task_Shot01 . cpp 該当部分のみ抜粋

```
    //-------------------------------------------------------------------
    //「更新」１フレーム毎に行う処理
    void  Object::UpDate()
    {
        this->moveCnt++;
        //限界の時間を迎えたら消滅
        if (this->moveCnt >= 100) {
            //消滅申請
            this->Kill();
            return;
        }
        //移動
        this->pos += this->moveVec;
        //重力加速
        this->moveVec.y += ML::Gravity(32) * 5;

        //移動先で障害物に接触したら消滅
        //マップが存在するか調べてからアクセス
以下省略
```

課題 15-4

ここまでの内容を実装したプログラムを提出せよ。

15. 5. テスト用の敵キャラクタを出現させる

キャラクタ同士の接触判定を確認するために、テスト用の敵キャラクタを出現させます。
まず、接触判定は組み込まず、敵キャラクタの実装と配置のみを行います。

追加タスク

ファイル名:	Task_EnemyTest.h 及び.cpp		
ファイルタイトル：	敵キャラクタ（あたり判定テスト用		
ネームスペース名：	EnemyTest		
デフォルトグループ名：	"敵"		
デフォルトタスク名：	"NoName"		
追加変数（Resorce)：	DG::Image::SP	img;	
追加変数（Object)：	ML::Vec2	pos;	//キャラクタ位置
	ML::Box2D	hitBase;	//あたり判定範囲
追加メソッド：	無し		

cpp 追加・変更部分のみ抜粋

```
//-------------------------------------------------------------------
//リソースの初期化
bool  Resource::Initialize()
{
    this->img = DG::Image::Create("./data/image/HitTest.bmp");
    return true;
}
//-------------------------------------------------------------------
//リソースの解放
bool  Resource::Finalize()
{
    this->img.reset();
    return true;
}
//-------------------------------------------------------------------
//「初期化」タスク生成時に１回だけ行う処理
bool  Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成orリソース共有
    this->res = Resource::Create();

    //★データ初期化
    this->render2D_Priority[1] = 0.4f;
    this->pos.x = 0;
    this->pos.y = 0;
    this->hitBase = ML::Box2D(-16, -16, 32, 32);
```

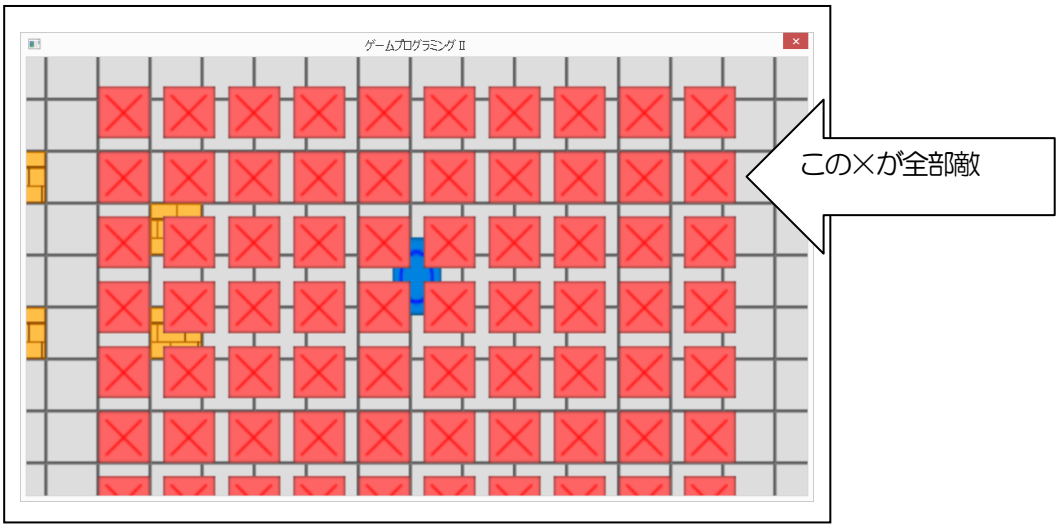
```
        //★タスクの生成

        return  true;
    }
    //-------------------------------------------------------------------
    //「2D描画」1フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
        ML::Box2D  draw(-16, -16, 32, 32);
        draw.Offset(this->pos);
        ML::Box2D  src(100, 0, 100, 100);
        //スクロール対応
        draw.Offset(-ge->camera2D.x, -ge->camera2D.y);
        this->res->img->Draw(draw, src);
    }
```

この敵タスクを900体出現させます。

Task_Game．cpp 追加内容のみ記載　　（タスクの初期化処理に追加する）

```
    //敵キャラクタの配置
    for (int y = 0; y < 30; ++y) {
        for (int x = 0; x < 30; ++x) {
            int  px = x * 40 + 400;
            int  py = y * 40 + 200;
            auto  enemy = EnemyTest::Object::Create(true);
            enemy->pos.x = float(px);
            enemy->pos.y = float(py);
        }
    }
```

縦横３０個、合計９００体の敵が出現する事を確認してください。



敵の出現が確認できたらあたり判定を実装します。
（極端に動作が遅くなる場合、敵数を減らして調整してください）

以前（ゲームプログラミングＩの１２章）でも説明した通り、あたり判定は攻撃する側が主体となり
執り行います。

プレイヤの弾と敵のあたり判定であれば、「プレイヤの弾」が主体となります。
今回は、ダメージ計算などは行わず、接触した弾と敵の両方が消えるだけのものとして処理を実装します。

Task_Shot00．cpp と Task_Shot01．cpp　追加内容のみ記載（UpDate( )メソッドの終端に追加）

```
//敵対象と衝突判定＆ダメージを与える処理
ML::Box2D  me = this->hitBase.OffsetCopy(this->pos);
//敵を全て抽出する
auto  targets = ge->GetTasks<EnemyTest::Object>("敵");
for (auto it = targets->begin();
    it != targets->end();
    ++it)
{
    //敵キャラクタのあたり判定矩形を用意
    ML::Box2D  you = (*it)->hitBase.OffsetCopy((*it)->pos);
    //重なりを判定
    if (true == you.Hit(me)) {
        (*it)->Kill();
        this->Kill();
        //とりあえず星はばら撒くよ
        for (int c = 0; c < 4; ++c) {
            auto  eff = Effect00::Object::Create(true);
            eff->pos = this->pos;
        }
        return;
    }
}
```

攻撃を加える側を me、受ける側を you として、あたり判定用の矩形を用意する事は以前と変わりません。
敵キャラクタすべてに対してループを回す為に、タスクシステムから対象のタスクを全て検出します。

複数のタスクを検出するには、
auto targets = ge->GetTasks<EnemyTest::Object>("敵");
を使用します。
タスクを１つだけ検出する GetTask と記述こそほとんど同じですが、戻り値の型は全く異なります。
この部分を間違えると、その先で変数 targets を使おうとした時、エラーが発生します。
端折って言うなら、GetTask が返すのは「変数」で、GetTasks が返すのは「配列」です。
変数と配列ではアクセスする方法が異なりますよね？

検出したタスク全てに対して処理を行うので、ループを使用しています。
for 文の中身を理解するには STL コンテナの「vector」を学習する必要があります。

(*it)が検出した敵キャラクタのうちの１体を指すポインタだと考えましょう。
これも正しく理解するには STL コンテナを学習したうえで「イテレータ」を学習する必要があります。
利用するだけなら、以上のプログラムを参考にすれば、複数対象を検出して処理する事は可能です。

15. 6. 検出不可の設定を行う（処理の効率化②）

タスクシステムに登録されているタスクの数が増えると、タスクの検出にも相応の負荷がかかるようになります。

例えば、登録されている中から該当する数件のタスクを検出しようとしたとしましょう。

タスクの登録数が１００件である場合、１００件の中から探します。
タスクの登録数が１０００件であるばあい、１００件の時の１０倍のコストがかかります。

しかも、タスクが増えているので、１フレームの処理内でタスク検出を求められる回数も増加します。
すべてのタスクが1回ずつタスク検出を要求すると、登録件数が１０倍になる度に、検出処理のコストは１００倍に増えていきます。
タスク検出の処理は基本コストが高めなので、これが１００倍になるということは、処理落ちの原因になります。

そこで、接触判定の対象にならないタスクなどは、初めから検出不可（不要）の設定をしておくことでコストの増加を抑えることが出来ます。

現時点でこの対象になるのは「エフェクト」系のタスクです。
エフェクトは単なる画面効果なので、別のタスクから接触判定を求められる事が無いと考えられます。

検出の可不可の設定は、タスク生成時にのみ可能です。

各タスクの．cpp 該当部分のみ抜粋

```
    //------------------------------------------------------------------------
    //「初期化」タスク生成時に１回だけ行う処理
    bool  Object::Initialize()
    {
        //スーパークラス初期化
        __super::Initialize(defGroupName, defName, false);
        //                                          ↑検出不可に指定する場合

        //リソースクラス生成 or リソース共有
        this->res = Resource::Create();
以下略
```

スーパークラス初期化と書かれたコメントの次の行の３番目の引数を true にすれば「検出可能」に、false にすれば「検出不可能」になります。（ひな形は基本 true になっています）

じつは、このタスクシステム内には２つの配列があり、検出の可不可により配属される配列が決まります。

GetTask 系の検出は、「可」の側の配列に対してしか行われないので、フラグを使い分ければ検出コストを抑えられるようになります。（検出ができなくなるだけで、KillAll_G( )等の機能は利用できます）

「弾」タスクが検出不可、「プレイヤ」「敵」タスクが検出可の状態を例にして考えてみましょう。

- 「敵」タスクが「敵」タスクの検出を要求したら、敵タスクの一覧を手にすることができます。
- 「敵」タスクが「ﾌﾟﾚｲﾔ」タスクの検出を要求したら、ﾌﾟﾚｲﾔタスクの一覧を手にすることができます。
- 「敵」タスクが「弾」タスクの検出を要求しても、弾タスクの一覧はもらえません。

- 「ﾌﾟﾚｲﾔ」タスクが「敵」タスクの検出を要求したら、敵タスクの一覧を手にすることができます。
- 「ﾌﾟﾚｲﾔ」タスクが「ﾌﾟﾚｲﾔ」タスクの検出を要求したら、ﾌﾟﾚｲﾔタスクの一覧を手にすることができます。
- 「ﾌﾟﾚｲﾔ」タスクが「弾」タスクの検出を要求しても、弾タスクの一覧はもらえません。

- 「弾」タスクが「敵」タスクの検出を要求したら、敵タスクの一覧を手にすることができます。
- 「弾」タスクが「ﾌﾟﾚｲﾔ」タスクの検出を要求したら、ﾌﾟﾚｲﾔタスクの一覧を手にすることができます。
- 「弾」タスクが「弾」タスクの検出を要求しても、弾タスクの一覧はもらえません。

だれであろうと、検出不可のタスクの一覧をもらうことはできません。
（検出不可のタスク単体の検出もできません）

検出を要求するタスクが、検出可なのか、不可なのかは、検出の成否には影響しません。

現在のプログラムでは、この方法による速度面の改善は効果が薄いです。
エフェクトが大量に飛び回るようなゲームであれば、大きな効果が期待できます。

16. 汎用キャラクラス

16. 1. 汎用キャラクタクラスの利点

ゲームを作り、複数のキャラクタを作っていくと、各キャラクタには共通している情報や機能を
持っていることがわかります。
それらをいちいち各キャラクタに実装していくのではなく、「汎用キャラクタクラス」（基底クラス）を
作り、それを各キャラクタタスクに「継承」させる事でプログラム作成に多くの利益が得られます。

「汎用キャラクタクラス」はミニ四駆のシャーシに例えることが出来ます。
シャーシは同じもの（同じ規格）でも外装のフレームやタイヤなどを変えることで新たな車種を
作ることが出来ます。
場合によっては、その他の部品を付け替えたり、増設したりも出来ます。

つまり、共通部分はいちいち作り直さなくても良いということです。
ほかにも利点はありますが、まずは「継承」によって可能になる「共通部分」を活用する方法を
学習しましょう。

16. 2. 汎用キャラクタクラスの作成

１５章のプログラムをベースにして、汎用キャラクタクラスの組み込みを行います。
まず、プレイヤと弾、それから今後追加される敵などに対する汎用キャラクタクラスを作成します。

共通化する情報（変数やメソッド）は、それを引き継ぐ全員が必ず使うものでなくても構いません。
複数の者が使う可能性のある情報なら共通化の対象と考えます。
（つまり、弾には必要なくても、プレイヤと敵に必要な情報なら共通化すると考えてよいです）

BChara . h　＊タスクのひな型は使用せず、新規にファイルを作成する

```
#pragma once
#pragma warning(disable:4996)
#pragma once
//-----------------------------------------------------------------------------
//キャラクタ汎用スーパークラス
//-----------------------------------------------------------------------------
#include "GameEngine_Ver3_83.h"

class BChara : public BTask
{
    //変更不可◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
public:
    typedef shared_ptr<BChara>      SP;
    typedef weak_ptr<BChara>        WP;
public:
    //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    //キャラクタ共通メンバ変数
    ML::Vec2        pos;        //キャラクタ位置
    ML::Box2D       hitBase;    //あたり判定範囲
    ML::Vec2        moveVec;    //移動ベクトル
    int             moveCnt;    //行動カウンタ
```

```
    //左右の向き（2D 横視点ゲーム専用）
    enum class Angle_LR { Left, Right };
    Angle_LR       angle_LR;
    //メンバ変数に最低限の初期化を行う
    //★★メンバ変数を追加したら必ず初期化も追加する事★★
    BChara()
        : pos(0, 0)
        , hitBase(0, 0, 0, 0)
        , moveVec(0, 0)
        , moveCnt(0)
        , angle_LR(Angle_LR::Right)
    {
    }
    virtual  ~BChara() {}

    //キャラクタ共通メソッド
    //めり込まない移動処理
    void  CheckMove(ML::Vec2&  est_);
    //足元接触判定
    bool  CheckFoot();
};
```

「汎用キャラクタクラス」（基底クラス）は BTask を継承して作ります。
「継承」とは、それが持つ能力を引き継いだクラスを作ることです。

今まで作ってきたプログラムがタスクシステムに登録でき、自動で UpDate( )メソッドなどが呼び出されていたのも、各タスクに BTask を継承させていたからです。

```
class BChara : public  BTask
```

ここで、BChara クラスは、BTask クラスを「継承」することを示しています。

BChara クラスには、現在実装されている３種類のキャラクタが持っていた変数のほぼすべて持たせました。

```
ML::Vec2        pos;     //キャラクタ位置
ML::Box2D       hitBase;//あたり判定範囲
ML::Vec2        moveVec;//移動ベクトル
int             moveCnt;//行動カウンタ
//左右の向き（2D 横視点ゲーム専用）
enum class  Angle_LR { Left, Right };
Angle_LR        angle_LR;
```

これらの変数の初期化（最低限の初期値設定やゼロクリア）はコンストラクタに記述します。
（クラス名と同名のメソッドの事をコンストラクタと呼びます）

```
BChara()
    : pos(0, 0)
    , hitBase(0, 0, 0, 0)
    以下略
```

BChara . cpp　＊タスクのひな型は使用せず、新規にファイルを作成する

```
//-------------------------------------------------------------------------------
//キャラクタ汎用スーパークラス
//-------------------------------------------------------------------------------
#include "BChara.h"
#include  "MyPG.h"
#include  "Task_Map2D.h"

//-------------------------------------------------------------------------------
//めり込まない移動処理
void BChara::CheckMove(ML::Vec2&  e_)
{
    中身はプレイヤに実装してあるものと全く同じ（コピペで移植可能）
}
//-------------------------------------------------------------------------------
//足元接触判定
bool  BChara::CheckFoot()
{
    中身はプレイヤに実装してあるものと全く同じ（コピペで移植可能）
}
```

cpp ファイルには、共通化の可能なメソッドを書きます。
関数内に書く処理は、すでにプレイヤタスクに実装されている処理と全く同じです。

これでとりあえず、現時点で共通化できるものをまとめた BChara クラスは作成できました。

16. 3. プレイヤに汎用キャラクタクラスを継承させる

すでに BTask を継承した BChara は作りました。
次は、Player タスクが継承しているクラスを、BTask から BChara に変更します。

Task_Player . h

```
#pragma warning(disable:4996)
#pragma once
//-------------------------------------------------------------------
//プレイヤ
//-------------------------------------------------------------------
#include "BChara.h"    ＊元は GameEngine_Ver3_83.h

namespace  Player
{
    //タスクに割り当てるグループ名と固有名
    const  string  defGroupName( "プレイヤ");    //グループ名
    const  string  defName(      "仮");          //タスク名
    //-------------------------------------------------------------------
    class  Resource : public BResource
```

```
    {
        bool  Initialize()override;
        bool  Finalize()  override;
        Resource();
    public:
        ~Resource();
        typedef  shared_ptr<Resource>    SP;
        typedef  weak_ptr<Resource>      WP;
        static   WP  instance;
        static  Resource::SP  Create();
        //共有する変数はここに追加する
        DG::Image::SP  img;
    };
    //-------------------------------------------------------------------
    class  Object : public  BChara       ＊元はBTask
    {
    //変更不可◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆
    public:
        virtual  ~Object();
        typedef  shared_ptr<Object>      SP;
        typedef  weak_ptr<Object>        WP;
        //生成窓口 引数はtrueでタスクシステムへ自動登録
        static  Object::SP  Create(bool flagGameEnginePushBack_);
        Resource::SP   res;
    private:
        Object();
        bool  B_Initialize();
        bool  B_Finalize();
        bool  Initialize();   //「初期化」タスク生成時に１回だけ行う処理
        void  UpDate()        override;//「実行」１フレーム毎に行う処理
        void  Render2D_AF()       override;//「2D描画」１フレーム毎に行う処理
        bool  Finalize();     //「終了」タスク消滅時に１回だけ行う処理
    //変更可◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇◇
    public:
        //追加したい変数・メソッドはここに追加する
        //BCharaに持たせていない変数・メソッドのみここに書く
        XI::GamePad::SP    controller;     ＊これ以外の変数は削除　メソッドも削除
    };
}
```

BCharaに持たせた変数やメソッドは、それを継承したクラスに持たせて（宣言を書いては）はいけません。
宣言を消し忘れると、１つのクラス内に同時に２つの変数が存在することになります。

Ｃ＋＋の仕様を理解していない者がそれをすれば、バグを生み出す要因にしかならないので確実に
削除しておきましょう。

続けて.cppファイル側にも変更を加えましょう。

Task_Player . cpp 該当部分のみ抜粋

```
//-------------------------------------------------------------------
//「初期化」タスク生成時に１回だけ行う処理
bool  Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成 or リソース共有
    this->res = Resource::Create();

    //★データ初期化
    this->pos.x = 0;      ←削除する
    this->pos.y = 0;      ←削除する
    this->render2D_Priority[1] = 0.5f;
    this->controller = ge->in1;
    this->hitBase = ML::Box2D(-15, -24, 30, 48);
    this->angle_LR = Angle_LR::Right;

    //★タスクの生成

    return  true;
}
```

BChara で行う初期化とは別の値を指定する必要がある変数のみ初期化を書く

以下のメソッドは削除する

```
//-------------------------------------------------------------------
//めり込まない移動処理
void Object::CheckMove(ML::Vec2&  e_)
{
    省略・・・・
}
//-------------------------------------------------------------------
//足元接触判定
bool  Object::CheckFoot()
{
    省略・・・・
}
```

これでプレイヤへの変更は終了です。
プレイヤは BChara を継承する形に変更されました。
実行結果に変化が無いことを確認しましょう。

プレイヤの Object クラスには、CheckMove()や、CheckFoot()メソッドはなくなってしまいました。
しかし、プレイヤは今や BChara の能力を引き継いでいるので、代わりに BChara が持っている CheckMove()や、CheckFoot()を使うことができます。
だから、プレイヤの Object::UpData( )メソッドの処理を全く変更しなくてもそのままプログラムは動きます。

課題 16-1

弾タスク（２種類）と、敵タスクも BChara を継承する形に作り替えよ。

16. 4. 複数種のタスクを一括で検出する

現時点で敵は１種類しかいません。（EnemyTest だけ）
しかし、ゲームを作っていくと、弾と同じように敵の種類も増えていきます。

例えば、EnemyA EnemyB EnemyC の３種類のタスクがあるとしましょう。

これらのタスクが BChara を継承していない場合、タスクのグループ名も「敵 A」「敵 B」「敵 C」と分けておき、それぞれのグループ名で検出する必要があります。
つまり、当たり判定などで敵のリストが欲しいとき、３度も検出処理を要求することになります。

グループ名を「敵」に統一した場合、１回の検出要求で３種類全部の敵を検出できますが、
敵のデータにアクセスしたときに異常なアクセスが起こり、データが破損します。

```
auto  tg = ge->GetTasks<EnemyA::Object>("敵");
```

指定できる型は１種だけ

３種類のタスクのアドレスを、すべてを EnemyA 型のポインタで受け取ってしまったリスト

EnemyA 型のポインタで EnemyB 型の敵にアクセスすると、間違ったメモリアクセスを起こすのです。

BChara を継承したタスクであれば、この問題を回避することができます。

EnemyA EnemyB EnemyC の３種類のタスクが皆 BChara を継承していて、グループ名が「敵」で統一されていれば、以下の方法で３種類全部の敵を検出し、安全にアクセスできるようになります。

```
auto  tg = ge->GetTasks <BChara>("敵");
```

３種類のタスクの BChara の部分のアドレスを BChara 型のポインタで受け取ったリスト

この方法であれば、複数種類のタスクを１度に検出して、扱うことができます。
ただし、この方法でアクセスできるのは BChara の持つ変数とメソッドだけです。

例えば、Player タスクの Object クラスには、XI::GamePad::SP　controller; という変数がありますが、BChara 型のポインタでは、この変数にはアクセスできません。
＊学習を進めれば、それを可能にする方法も得られます。
当面、この制約は問題にならないので無視しておきましょう。

課題 16-1+　(提出不要・先の内容に進むために変更は必須)

弾と敵の当たり判定で敵タスクを検出する際の型を BChara に変更して、今までと変わりなく動作することを確認せよ。

16. 5. 敵検出の頻度を減らす（処理の効率化③）

現在、弾は敵キャラクタとあたり判定するとき、弾１発毎に敵キャラクタの検出処理をしています。
前にも示した通り、タスクの検出処理はコストの高い処理です。

弾が３０発出現しているならば、３０個の弾タスクがそれぞれの処理の中ですべての敵の検出処理を
することになるので、処理の効率は格段に悪くなります。

そこで、１フレーム毎の処理で敵を検出する回数を１回に固定することでコストを低減させます。

Task_Game の Object::UpDate( )処理の中で敵を検出させ、それ（敵のテーブル）を持たせておきます。
弾はそれを「借りて」あたり判定を行うように変更します。

MyPG . h 該当部分のみ抜粋

ファイル先頭部分に追加

```
#include "BChara.h"
```

ファイル下方「ゲームエンジンに追加したいものは下に加える」の部位に追加

```
    //敵の検出処理を少なくするために
    shared_ptr<vector<BChara::SP>>      qa_Enemys;
```

Task_Game . cpp 該当部分のみ抜粋　Object::UpDate()メソッド内に追加する

```
    //アクセス効率化の為(敵の検出を減らす)
    ge->qa_Enemys = ge->GetTasks<BChara>("敵");
```

Task_Shot00 . cpp および、Task_Shot01 . cpp　該当箇所のみ抜粋

変更前

```
//敵を全て抽出して当たり判定する
auto  targets = ge->GetTasks<BChara>("敵");                    ①
for (auto it = targets->begin();
    it != targets->end();
    ++it)
{
    以下略
```

変更後

```
//敵すべてと当たり判定を行う
for (auto it = ge->qa_Enemys->begin();                         ②
    it != ge->qa_Enemys->end();
    ++it)
{
    //すでに「タスク状態が死亡」になっている敵は当たり判定から除外する
    if ((*it)->CheckState() == BTask::State::Kill) { continue; }    ③
    以下略
```

①個々の弾タスクが、自分で「敵」の検出を行っていた部分を削除。

②ゲームエンジンに持たせた「敵のテーブル」を借りて、当たり判定のループを回す。

③リスト内の「敵」の中には、同フレーム処理中に「死亡」になったものがいる可能性がある。
「死亡」になった敵と当たり判定をすると問題が生じるので、その敵との当たり判定をキャンセルする。

課題 16-SP1

各タスクが、個別にマップ（Map2D タスク）を検出するのをやめ、敵キャラクタと同様に、１フレームに１回だけ検出する形に作り替えよ。

ヒント　MyPG.h

```
    //敵の検出処理を少なくするために
    shared_ptr<vector<BChara::SP>>      qa_Enemys;
追加↓
    //マップの検出処理を少なくするために
    Map2D::Object::SP                   qa_Map;
```

ヒント　Task_Game．cpp

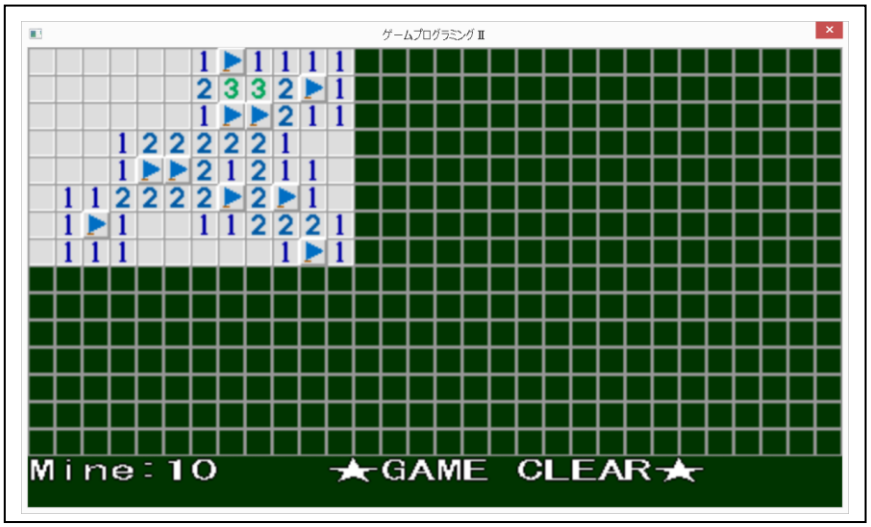
```
    //アクセス効率化の為(敵の検出を減らす)
    ge->qa_Enemys = ge->GetTasks<BChara>("敵");
追加↓
    //アクセス効率化の為(マップの検出を減らす)
    ge->qa_Map = ge->Get？？？？？？？？？？？？？？？？？？；
```

＊マップと接触判定するすべてのタスク・処理を見直す。

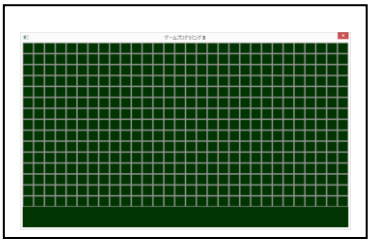
17. マインスイーパー

17. 1. マインスイーパー

フィールド中に隠されたマイン（地雷）の位置を予測しそのマスに旗を立てて、それ以外の全てのマスを開封するとクリアになるゲームです。



配布フォルダから、「１７章マインスイーパー用のスケルトンプログラム」をコピーしてください。
（本編タスクで、マスのガイドラインが書かれた背景画像を表示するだけのプログラムです）



動作確認が出来たらゲームの実装に移りましょう。
ゲームのルールをしっかり把握してから始めてください。（ブラウザ上で無料で遊べるサイトもあります）

17. 2. 地雷数を表示する

まず、地雷数を表示するタスクを追加します。後にゲームの状態（オーバー・クリア）なども
このタスク内で表示する予定です。
また、場合によっては経過時間などを表示するのに使っても良いでしょう。
そういったゲームの情報を表示するタスクを追加します。

追加タスク

ファイル名:	Task_ShowInfo.h 及び.cpp
ファイルタイトル：	ゲーム情報の表示
ネームスペース名：	ShowInfo
デフォルトグループ名：	"情報表示"
デフォルトタスク名：	"NoName"
追加変数（Resorce)：	DG::Font::SP font;
追加変数（Object)：	無し
追加メソッド：	無し

cpp 追加・変更部分のみ抜粋

```
//-----------------------------------------------------------------------
//リソースの初期化
bool  Resource::Initialize()
{
    this->font = DG::Font::Create("ＭＳ ゴシック", 12, 24, 100);
    return true;
}

//-----------------------------------------------------------------------
//リソースの解放
bool  Resource::Finalize()
{
    this->font.reset();
    return true;
}

//-----------------------------------------------------------------------
//「初期化」タスク生成時に１回だけ行う処理
bool  Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成 or リソース共有
    this->res = Resource::Create();
    //★データ初期化
    this->render2D_Priority[1] = 0.1f;
    //★タスクの生成

    return  true;
}

//-----------------------------------------------------------------------
//「２D描画」１フレーム毎に行う処理
void  Object::Render2D_AF()
{
    string  buf;

    buf = "Mine:" + to_string(ge->mineMax);

    ML::Box2D  draw(0, 240, 480, 30);
    this->res->font->DrawF(draw, buf, DG::Font::x4,
                           ML::Color(1, 1, 0, 0), ML::Color(1, 1, 1, 1));
}
```

網掛け部分の「ge->mineMax」は地雷の数を保持する変数です。
現時点ではこの変数は宣言されていないので、ビルドエラーになります。

MyPG．h　該当箇所のみ抜粋

```
    //ゲームエンジンに追加したいものは下に加える
    //-------------------------------------------
        MyPG::Camera::SP      camera[4];     //  カメラ
        XI::Mouse::SP  mouse;
        XI::GamePad::SP  in1, in2, in3, in4;//取り合えず4本
        int  mineMax;
    //-------------------------------------------
```

Task_Game．cpp　該当箇所のみ抜粋（Object::Initialize( )メソッド内）

```
        //★データ初期化
        srand((unsigned int)time(NULL));
        ge->mineMax = 15;

        //★タスクの生成
        //BG 生成
        auto  bg = BackGround::Object::Create(true);
        //情報表示タスク生成
        auto  si = ShowInfo::Object::Create(true);
```

変数の宣言と初期化を追加したので実行確認できるようになったはずです。

ビルドが通ることを確認したら、Task_Game 上で「ゲーム情報の表示タスク」を生成・エンジンに登録して画面左下に地雷数が表示されることを確認してください。

（画面左下）

17. 3. ゲーム盤を追加する

次に、ゲーム盤タスクを追加します。まずは全てのマスを空白マスにして、任意の位置に数字や地雷を表示し、表示処理が正しく機能しているか確認するところまで実装します。

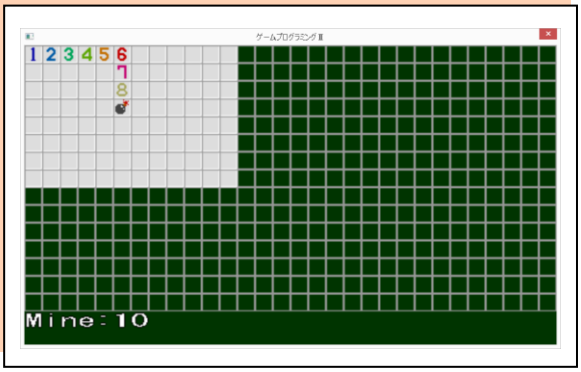
追加タスク

ファイル名:	Task_Board.h 及び.cpp	
ファイルタイトル：	ゲーム盤	
ネームスペース名：	Board	
デフォルトグループ名：	"ゲーム盤"	
デフォルトタスク名：	"NoName"	
追加変数（Resorce)：	DG::Image::SP	imgChip;
追加変数（Object)：	int	arr[15][30];
	ML::Box2D	chip[10];
	int	sizeX;
	int	sizeY;
追加メソッド：	//ボードの広さと地雷数を指定してボードを初期化する	
	void Set(int sx_, int sy_, int mine_);	

cpp 追加・変更部分のみ抜粋

```
//-------------------------------------------------------------------
//リソースの初期化
bool  Resource::Initialize()
{
    this->imgChip = DG::Image::Create("./data/image/Chip.bmp");
    return true;
}
//-------------------------------------------------------------------
//リソースの解放
bool  Resource::Finalize()
{
    this->imgChip.reset();
    return true;
}
//-------------------------------------------------------------------
//「初期化」タスク生成時に１回だけ行う処理
bool  Object::Initialize()
{
    //スーパークラス初期化
    __super::Initialize(defGroupName, defName, true);
    //リソースクラス生成orリソース共有
    this->res = Resource::Create();
    //★データ初期化
    this->render2D_Priority[1] = 0.8f;
    this->sizeX = 0;
    this->sizeY = 0;
```

```
        //チップ情報の初期化
        for (int c = 0; c < 10; ++c) {
            this->chip[c] = ML::Box2D(c * 16, 0, 16, 16);
        }
        //マップ配列情報の初期化
        ZeroMemory(this->arr, sizeof(this->arr));
        //★タスクの生成

        return  true;
    }
    //-------------------------------------------------------------------
    //「２D描画」１フレーム毎に行う処理
    void  Object::Render2D_AF()
    {
        //表示
        for (int y = 0; y < this->sizeY; ++y) {
            for (int x = 0; x < this->sizeX; ++x) {
                ML::Box2D  draw(x * 16, y * 16, 16, 16);
                this->res->imgChip->Draw(draw, this->chip[this->arr[y][x]]);
            }
        }
    }
    //-------------------------------------------------------------------
    //ボードの広さと地雷数を指定してボードを初期化する
    void  Object::Set(int sx_, int  sy_, int  mine_)
    {
        this->sizeX = sx_;
        this->sizeY = sy_;
        //ボードをクリア
        for (int y = 0; y < this->sizeY; ++y) {
            for (int x = 0; x < this->sizeX; ++x) {
                this->arr[y][x] = 0;
            }
        }
        //仮のデータ設定（表示確認用確認後削除する
        this->arr[0][0] = 1;
        this->arr[0][1] = 2;
        this->arr[0][2] = 3;
        this->arr[0][3] = 4;
        this->arr[0][4] = 5;
        this->arr[0][5] = 6;
        this->arr[1][5] = 7;
        this->arr[2][5] = 8;
        this->arr[3][5] = 9;
    }
```



Set()メソッドに注目してください。このメソッドはタスク生成直後に盤面の大きさと地雷数を指定して呼び出す必要があるメソッドです。
このメソッドには今後多くの処理を追加していきます。

タスクを追加したら、その生成及び、Set( )メソッドの呼び出しを追加します。

Task_Game . cpp Object::Initialize( )メソッド内に追加

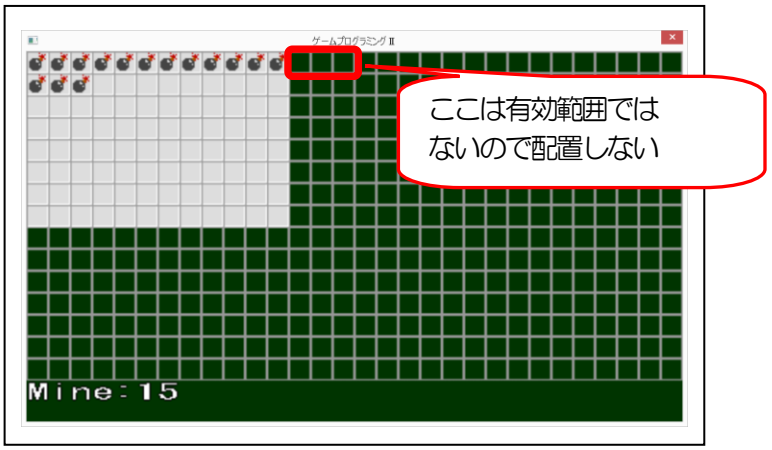
```
//ゲーム盤の生成
auto  board = Board::Object::Create(true);
board->Set(12, 8, ge->mineMax);
```

前ページ下の、仮のデータ設定部は動作確認後削除してください。(ゲーム盤上に何もない状態になります)

課題 17-1

盤上に地雷を指定数分仮配置する処理を追加せよ。(Set( )メソッド内に追加する)

・ゲーム盤の有効範囲内に指定された数だけ地雷を配置する。
例えば、ゲーム盤が１２×８で地雷数が１５個である場合、実行結果は以下のようになる。



・有効範囲を超える数の地雷数が指定された場合の対策はしなくてもよい。
(作り方により必要になる場合もある、その場合の話、出来るならもちろんしたほうが良い)

＊地雷の数は、Set( )メソッドに対して引数で送られてきている。(ゲーム盤のサイズも)
この変数を必ず使うこと。

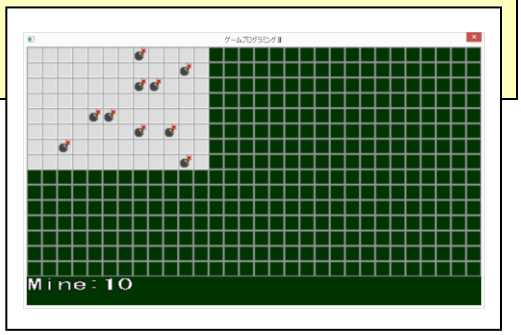
17．4．地雷をシャッフルする

ゲーム盤上の地雷をシャッフルして盤上に散らばせます。もちろんその過程で地雷が増減してしまったりしないように、かつ効率的にシャッフルします。

以下のコードは、２つのマスの情報を入れ替える処理を繰り返すものです。
１つ目のマスは配列の左上から順番に１個ずつ進み選択していきます。２つ目のマスは配列上のランダムな位置を選択します。これにより、すべてのマスが少なくとも１度以上入れ替えの対象となるため、双方のマスをランダムに選んで入れ替えるより効率的にシャッフルが行われます。

Task_Boadr．cpp　Set( )メソッド内、地雷の仮配置処理の後に、別のループを作り追記する。

```
//シャッフルして地雷をばらけさせる
for (int y = 0; y < this->sizeY; ++y) {
    for (int x = 0; x < this->sizeX; ++x) {
        int xr = rand() % this->sizeX;
        int yr = rand() % this->sizeY;
        int temp = this->arr[y][x];
        this->arr[y][x] = this->arr[yr][xr];
        this->arr[yr][xr] = temp;
    }
}
```



17．5．地雷の隣接マスに数値を書き込む

マインスイーパーは地雷の隣接マス（斜めを含む８マス）に地雷数を示す数値が表示されます。
隣接する地雷数を調べてその数を書き込む処理を追加します。

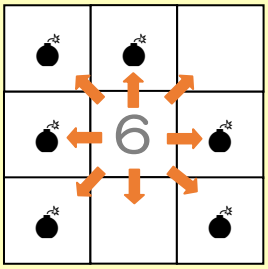
Task_Boadr．cpp　Set( )メソッド内、地雷のシャッフル処理の後に、別のループを作り追記する。

```
//隣接地雷数の書き込みを行う
for (int y = 0; y < this->sizeY; ++y) {
    for (int x = 0; x < this->sizeX; ++x) {
        if (this->arr[y][x] == 0) {
            this->arr[y][x] = this->CntMine(x, y);
        }
    }
}
```

上記のプログラムは、地雷の配置されていない空白マスを見つけては、そのマスに隣接する地雷数を調べてその数をマスに上書きする処理です。
地雷の数はCntMine( )メソッドで調べます。（CntMine( )メソッドの実装は次ページ）

Task_Boadr . cpp　追加メソッドのみ抜粋

```
//-------------------------------------------------------------------
//隣接８マスの地雷数を返す
int  Object::CntMine(int  x_, int  y_)
{
    ML::Point  m[8] = {
        { -1, -1 },    { 0, -1 },   { +1, -1 },
        { -1,  0 },                 { +1,  0 },
        { -1, +1 },    { 0, +1 },   { +1, +1 },
    };
    for (int i = 0; i < 8; ++i) {
        m[i].x += x_;
        m[i].y += y_;
    }
    //地雷数を調べる
    int  cnt = 0;
    for (int i = 0; i < 8; ++i) {
        //範囲外チェック
        if (m[i].x < 0) { continue; }
        if (m[i].y < 0) { continue; }
        if (m[i].x >= this->sizeX) { continue; }
        if (m[i].y >= this->sizeY) { continue; }
        //地雷を見つける毎にカウントアップ
        if (this->arr[m[i].y][m[i].x] == 9) {
            cnt++;
        }
    }
    return cnt;
}
```



特徴のある処理なので、内容をよく確認をしておきましょう。

引数で指定された座標ｘ_，ｙ_の周辺８マスを対象として地雷を探します。
配列ｍはその８マスを効率的に見るために用意したテーブルです。
（テーブルとは処理効率を向上させるための用意する配列の事です）
これに指定された座標ｘ_，ｙ_を足すことで、周辺８マスの座標を用意しています。

その情報を使用して地雷の有無を確認する処理を実施します。
ループ内の４つの if 文は、対象のマスがゲーム盤の有効範囲外であることを調べています。
チェックをパスした（２次元配列内）のマスだけを対象に地雷があるか調べます。
そのマスが地雷である場合、カウンタを＋します。

こうして、0から、最大８個までの地雷数を検出します。
検出した地雷の数を戻り値で返して処理終了です。

これで、ゲームを開始するためのゲーム盤の準備は終了、あとはキャップをかぶせるだけです。

17. 6. キャップを実装し、表示する

現在表示されているゲーム盤の全体を覆うフタ（キャップ）を追加し、上に重ねて表示することで
ゲーム盤を隠します。

Task_Board . h　追加内容のみ抜粋

Resource クラスに追加

```
    DG::Image::SP  imgCap;
```

Object クラスに追加

```
    int            cap[15][30];
    ML::Box2D      capChip[4];
```

Task_Board . cpp 追加内容のみ抜粋

Resource::Initialize()メソッドに追加

```
    this->imgCap  = DG::Image::Create("./data/image/Cap.bmp");
```

Object::Initialize()メソッドに追加

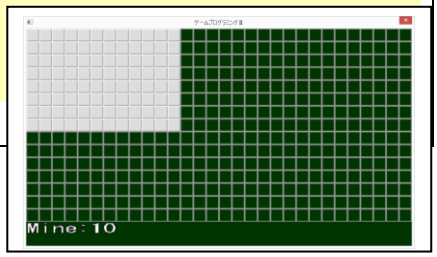
```
    //キャップ情報の初期化
    for (int c = 0; c < 4; ++c) {
        this->capChip[c] = ML::Box2D(c * 16, 0, 16, 16);
    }
    //キャップ配列情報の初期化
    ZeroMemory(this->cap, sizeof(this->cap));
```

Object::Set( )メソッドに追加

```
    //キャップ配列を初期化
    for (int y = 0; y < this->sizeY; ++y) {
        for (int x = 0; x < this->sizeX; ++x) {
            this->cap[y][x] = 1;
        }
    }
```

Object::Render2D_AF()メソッド内に追加

```
    //キャップ（覆い）を重ねて表示
    for (int y = 0; y < this->sizeY; ++y) {
        for (int x = 0; x < this->sizeX; ++x) {
            if (this->cap[y][x] != 0) {
                ML::Box2D  draw(x * 16, y * 16, 16, 16);
                this->res->imgCap->Draw(draw, this->capChip[this->cap[y][x]]);
            }
        }
    }
```



課題 17-2

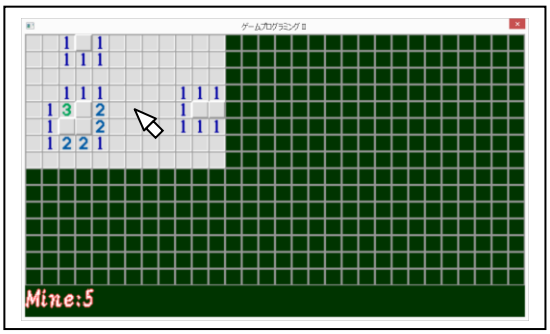
右クリック入力に対応して、キャップを「□」「▶」「?」に入れ替える処理を追加せよ。
既にキャップはこれらの表示に対応しているので、実装が必要なのは右クリックに対する処理だけである。
（キャップは　開封＝0、　□＝1、　▶＝2、　?＝3　に対応している）

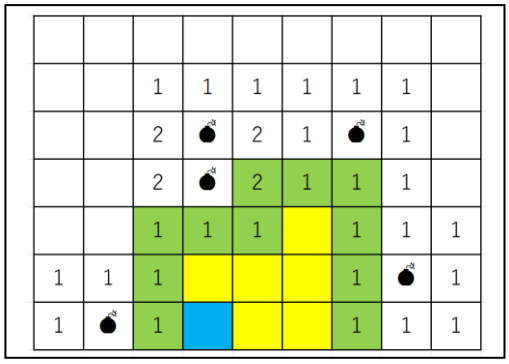
Task_Board . cpp 該当箇所のみ抜粋

```
//-----------------------------------------------------------------------
//「更新」１フレーム毎に行う処理
void  Object::UpDate()
{
    auto  ms = ge->mouse->GetState();
    //マウスの右ボタンが押されたら
    if (ms.RB.down) {
        //画面の入力範囲内であることを確認する
        ML::Point  pos = ms.pos;
        if (pos.x >= 0 && pos.x < 16 * this->sizeX &&
            pos.y >= 0 && pos.y < 16 * this->sizeY) {
            //マス単位の座標に変換する
            ML::Point masu = { pos.x / 16, pos.y / 16 };

            //指定座標のキャップを「空白」「ハタ」「?」でループさせる

                                    ?

            ＊開封状態のマスを未開封状態に戻してしまわないように注意
        }
    }
}
```

17．7．左クリックしたマスを開封する

左クリックしたマスを開封します。





クリックしたマスが「地雷マス」であった場合、そのマスだけが開封されます。（ゲームオーバー）
クリックしたマスが「数字マス」であった場合、そのマスだけが開封されます。（ゲーム継続）
クリックしたマスが空白マスであれば、そのマスに隣接する８マスもつられて全て開封されます。
（この８マスには絶対地雷はありません）
つられて開封されたマスの中に空白マスがあれば、その空白マスの周辺８マスも開封されます。
このようにして、空白マスの開封が連鎖していき、広範囲のマスが開封されます。

最終的に開封した空白マスとそれに隣接したすべての空白マスが開き、それに隣接するすべての数字マスも開きます。
（右上図参照）

青　＝　クリックしたマス
黄　＝　つられて開封される空白マス
緑　＝　つられて開封される数字マス

＊ちなみに、黄色のマスのどれを選んでいたとしても同じ範囲が開封されます。

左クリックに対する処理では、最終的にここまでの機能を実現しますが、
まず、開封処理を呼び出すまでのプログラムを実装しましょう。

Task_Boadr . cpp　UpDate( )メソッド内に追加

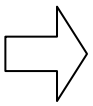
＊注意：右クリックに対する処理はそのまま残しておくこと

```
    //左ボタンが押されたら
    if (ms.LB.down) {
        //画面の入力範囲内であることを確認する
        ML::Point  pos = ms.pos;
        if (pos.x >= 0 && pos.x < 16 * this->sizeX &&
            pos.y >= 0 && pos.y < 16 * this->sizeY) {
            //マス単位の座標に変換する
            ML::Point masu = { pos.x / 16, pos.y / 16 };

            //指定座標のキャップが「通常」なら
            if (this->cap[masu.y][masu.x] == 1) {
                //マスを開放する
                this->SarchAndOpen(masu);
            }
        }
    }
```

未作成のSarchAndOpen( )メソッドで開封処理を行います。
空白マスが開封されたとき、隣接範囲を連鎖的に開封していくプログラムを作る為に、再帰関数という仕組みを利用します。　再帰関数とは、関数が自身の処理内で、自分自身を呼び出す仕組みです。
呼び出した自分の中で、さらに自分を呼び出します。

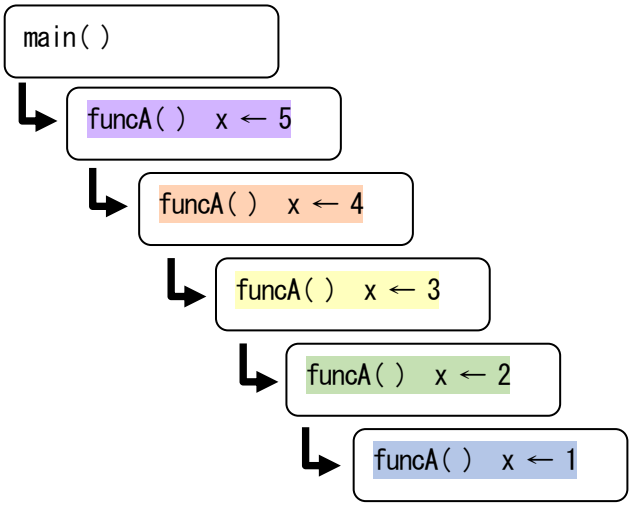
以下に単純な再帰のコードを挙げます。

```
void  main()
{
    funcA(5);
}
void  funcA(int  x)
{
    cout << ">" << x << endl;//出力
    int  a = x - 1;
    if(a > 0){  funcA(a);   }
    cout << "<" << x << endl;//出力
}
```

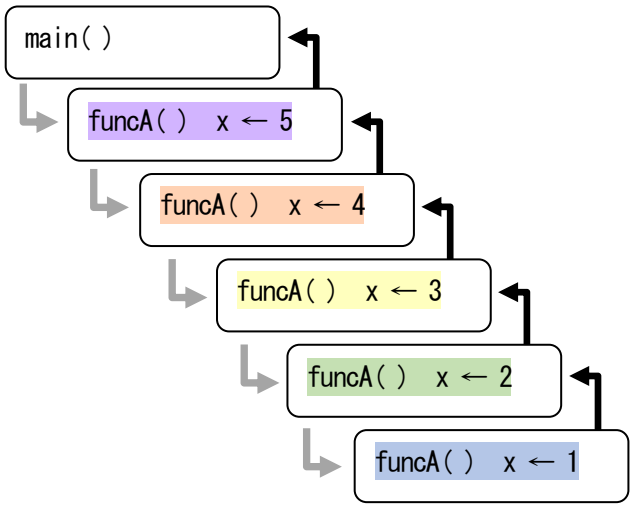


```
>5
>4
>3
>2
>1
<1
<2
<3
<4
<5
```

funcA( )関数の中でfuncA( )関数を呼び出しています。



永遠の呼び出し続けるわけではなく、何らかのきっかけにより呼び出しを終了し、根元に戻っていきます。



同じ処理を呼び出し続けて大きく広がっていきますが、最終的に根元に戻ってくるので「再帰関数」と呼ばれます。
同じ処理を呼び出し続けても、根元に戻ってこないのであれば再帰関数ではありません。

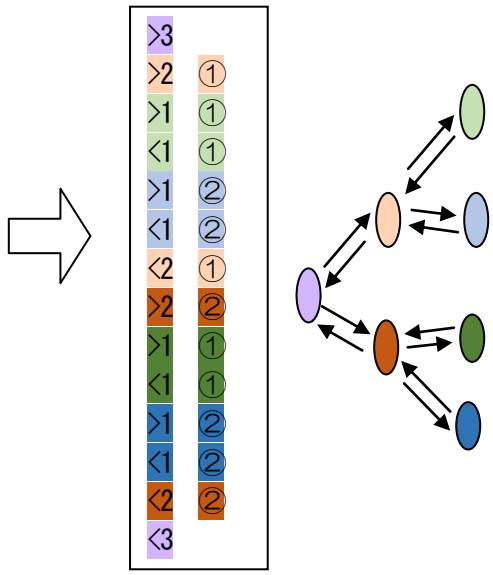
無制限に自分を呼び出し戻ってこないプログラムは「スタックオーバーフロー」という現象を起こし、プログラムが強制終了されます。

再帰関数を正しく理解するには、関数呼び出しとスタックについて理解する必要があります。

今回実装するキャップの開封処理はこの再帰関数を利用しています。

もう少し、再帰について学習しましょう。
以下のコードは２つに増えていく再帰のコードです。

```
void  main()
{
    funcA(3);
}
void  funcA(int  x)
{
    cout << ">" << x << endl;//出力
    int  a = x - 1;
    if(a > 0){  funcA(a);   }①
    if(a > 0){  funcA(a);   }②
    cout << "<" << x << endl;//出力
}
```



丸数字はプログラム上の物ではなく、多少わかりやすくするために追記したものです。
２つに増えていくというのはあくまでもイメージで、同時に兄弟が仕事をするわけではありません。

紙面上ではうまく説明できないので、授業の際に先生の説明をよく聞いてください。

ちなみに、マインスイーパーの再帰処理は２つに増えるどころか８つに増えます。
その動きを解説することは困難なので、再帰関数が１回の処理でどのようなことをしているかだけ説明します。

まずは、ソースコードを確認してください。

Task_Boadr . cpp　追加メソッドのみ抜粋

```
    //隣接マスを開放する
    void Object::SarchAndOpen(ML::Point pos_)
    {
        //探索開始前に行う処理があれば行う
        //（今回は無い）
        //探索開始
        this->SarchAndOpen_Sub(pos_.x, pos_.y);
    }
```

```
//---------------------------------------------------------------
void Object::SarchAndOpen_Sub(int x_, int y_)
{
    //キャップが開放済みなら探索終了
    if (this->cap[y_][x_] == 0) { return; }
    //足元のキャップを開放
    this->cap[y_][x_] = 0;
    //足元が空白以外なら探索終了
    if (this->arr[y_][x_] != 0) { return; }
    //隣接する8マスを再帰で調べる
    ML::Point  m[8] = {
        { -1, -1 },{ 0, -1 },{ +1, -1 },
        { -1,  0 },          { +1,  0 },
        { -1, +1 },{ 0, +1 },{ +1, +1 },
    };
    for (int i = 0; i < 8; i++) {
        m[i].x += x_;
        m[i].y += y_;
    }
    for (int i = 0; i < 8; i++) {
        //範囲外チェック
        if (m[i].x < 0) { continue; }
        if (m[i].y < 0) { continue; }
        if (m[i].x >= this->sizeX) { continue; }
        if (m[i].y >= this->sizeY) { continue; }
        //再帰
        this->SarchAndOpen_Sub(m[i].x, m[i].y);
    }
}
```

SarchAndOpen( )メソッドは再帰処理ではなくその窓口として用意しました。(今回は無くてもよかった)
再帰を行うのはSarchAndOpen_Sub( )メソッドです。

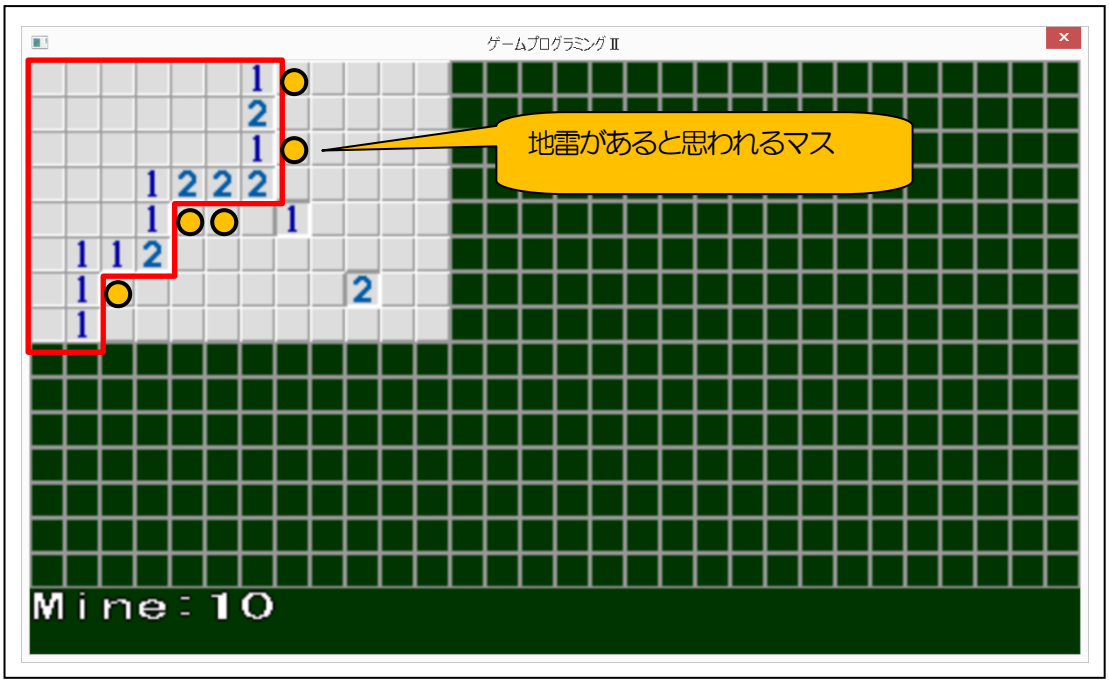
SarchAndOpen_Sub( )メソッドは、引数で指定されたマスのキャップが未開封か調べます。
開封済みなら、そこは以前に調べた事のあるマスなので処理を終了します。
↓
指定されたマスが未開封なら開封を行います。
↓
開封したマスが空白マス以外ならその時点で処理を終了します。

ここまで指定座標が「開封済み」「地雷」「数字」のマスへの処理は終わりです。( )
ここから先は指定座標が「空白」マスだった時だけ行う処理です。( )

複雑そうな処理に見えるかもしれませんが、目的は「周辺8マス」に対して「自分」を呼び出すことだけです。(SarchAndOpen_Sub( )を最大8回呼び出すということ)

そうして呼び出された「自分」は、新しいマスで再度、同手順の処理を実行していきます。

これを繰り返すと、隣接する空白マスおよび、それに隣接する数字のマスが全て開封されます。



17. 8. ゲームオーバー判定と表示を追加する

左クリックで解放したマスに地雷があった場合、ゲームオーバーにする必要があります。
まず、ゲームオーバーフラグを追加して、初期化しておきましょう。

MyPG . h　該当部分のみ抜粋

```
//ゲームエンジンに追加したいものは下に加える
    //-------------------------------------------
        MyPG::Camera::SP      camera[4];     //  カメラ
        XI::Mouse::SP         mouse;
        XI::GamePad::SP  in1, in2, in3, in4;//取り合えず4本
        int                   mineMax;
        bool                  gameOverFlag;  //  ゲームオーバーフラグ
    //-------------------------------------------
```

Task_Game . cpp　Object::Initialize( )メソッド内に追加

```
    ge->gameOverFlag = false;
```

更にフラグが true になった際に画面にメッセージが表示されるように変更を加えます。

Task_ShowInfo．cpp 該当部分のみ抜粋

```
//-------------------------------------------------------------------
//「２D描画」１フレーム毎に行う処理
void  Object::Render2D_AF()
{
     string  buf;

    if (ge->gameOverFlag == true) {
        buf = "Mine:" + to_string(ge->mineMax) + "    ☆GAME OVER☆";
    }
    else {
        buf = "Mine:" + to_string(ge->mineMax);
    }
      ML::Box2D  draw(0, 240, 480, 30);
      this->res->font->DrawF(draw, buf, DG::Font::x4,
                        ML::Color(1, 1, 0, 0), ML::Color(1, 1, 1, 1));
}
```

最後に、左クリックしたとき「指定したマスが地雷だった場合フラグを変更する処理」と、
「フラグがONになったらマウス入力を無効化する処理」をTask_Boardに加えます。

Task_Board．cpp　　Object::UpDate()メソッドのみ抜粋

```
//-------------------------------------------------------------------
//「更新」１フレーム毎に行う処理
void  Object::UpDate()
{
    //既にゲーム終了済みなら入力は無効
    if (ge->gameOverFlag == true) { return; }
    auto  ms = ge->mouse->GetState();
    //マウスの右ボタンが押されたら
    if (ms.RB.down) {
        //画面の入力範囲内であることを確認する
        ML::Point  pos = ms.pos;
        if (pos.x >= 0 && pos.x < 16 * this->sizeX &&
            pos.y >= 0 && pos.y < 16 * this->sizeY) {
            //マス単位の座標に変換する
            ML::Point masu = { pos.x / 16, pos.y / 16 };
            //指定座標のキャップを「空白」「ハタ」「?」でループさせる
            auto &ref = this->cap[masu.y][masu.x];
            switch (ref) {
            case 1:ref = 2;   break;
            case 2:ref = 3;   break;
            case 3:ref = 1;   break;
            }
        }
    }
    //左ボタンが押されたら
```

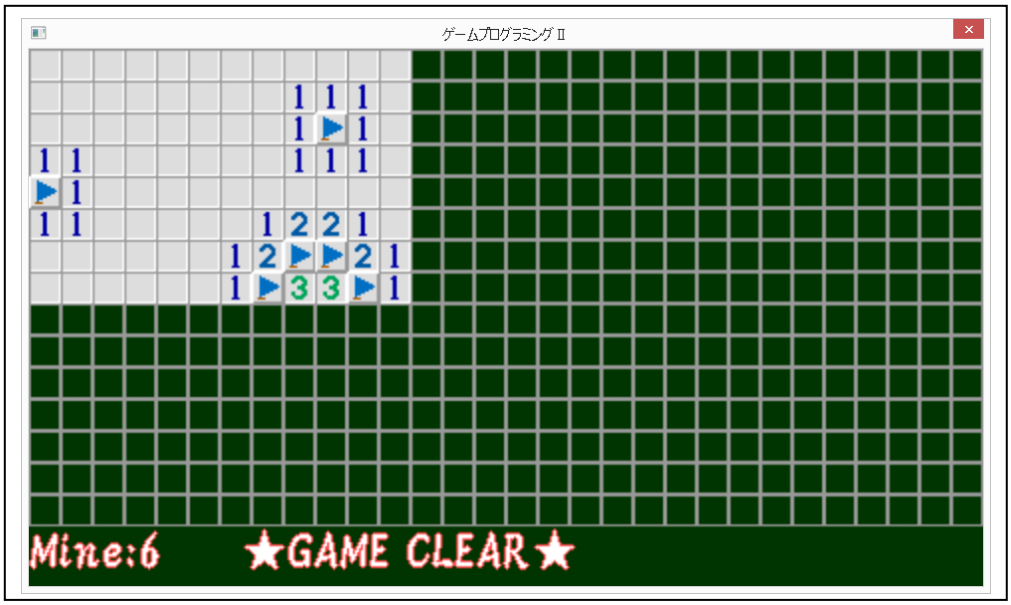
```
        if (ms.LB.down) {
            //画面の入力範囲内であることを確認する
            ML::Point  pos = ms.pos;
            if (pos.x >= 0 && pos.x < 16 * this->sizeX &&
                pos.y >= 0 && pos.y < 16 * this->sizeY) {
                //マス単位の座標に変換する
                ML::Point masu = { pos.x / 16, pos.y / 16 };

                //指定座標のキャップが「通常」なら
                if (this->cap[masu.y][masu.x] == 1) {
                    //マスを開放する
                    this->SarchAndOpen(masu);
                    //地雷を踏んだらゲームオーバー
                    if (this->arr[masu.y][masu.x] == 9) {
                        ge->gameOverFlag = true;
                    }
                }
            }
        }
    }
```

17．9．ゲームクリア判定と表示を追加する

全ての地雷にハタを立てて、それ以外の全てのマスが開封状態になったら、ゲームクリア状態にする必要があります。
以下に、クリア判定を行うメソッドと、クリア後のスクリーンショットを挙げておきますので、
不足分は自分で処理を追加して、クリア判定を機能させてください。（課題１７－３とする）
（ゲームオーバー判定の実装を参考にしてください）

Task_Board．cpp　　追加メソッドのみ抜粋

```
    //-------------------------------------------------------------------
    //クリア判定
    bool  Object::CheckClear()
    {
        //全ての地雷に旗を立て、それ以外のマス全てを開放している場合
        for (int y = 0; y < this->sizeY; ++y) {
            for (int x = 0; x < this->sizeX; ++x) {
                //地雷があり、旗を立ててある①
                if (this->arr[y][x] == 9 && this->cap[y][x] == 2) { continue; }
                //地雷がなく、開封済みである②
                if (this->arr[y][x] != 9 && this->cap[y][x] == 0) { continue; }
                return false;//①でも②でも無い所を見つけた（まだクリアでない）
            }
        }
        return true;
    }
```



課題 17-3

クリア判定を実装せよ、クリア判定は左クリック・右クリック双方で行う必要があるので注意する事。
＊何も操作していない時、クリア判定を行わないように作る事。
＊Sキー（スタートボタン）を押してタイトル画面に戻り、再度ゲームを始めた時きちんと
ゲームが出来るか確認する事。

課題 17-SP1

プレイ時間を表示する処理を追加せよ。

「Time：01:42　　Mine：10　　」　　← 通常時
「Time：03:14　　★GAME　CLEAR★」　　← 終了（クリア）後
「Time：03:14　　☆GAME　OVER☆」　　← 終了（オーバー）後

ゲームプログラミングⅡ（２０２１年度版）
２０２１年１０月１日初版　　第一刷発行

著者　須賀　康之
発行者　学校法人 電子学園 日本電子専門学校
〒１６９－８５２２　東京都新宿区百人町 1-25-4

発行所　株式会社　栄伸
印刷所　有限会社ひまわり印刷