ESライブラリ＆＆
ゲームプログラミング

○　ゲーム制作編 - 第６回　定数・関数・構造体　○

衝突検出

・よく使う数値や意味のある数値は、定数として宣言
・よく使う処理は、関数としてまとめる
・プログラムのメンテナンス性向上のため、１つの関数が長くなる場合も複数の関数に分割する
・関連のある変数は構造体にまとめる → 「関連性のある変数」ということがわかりやすくなる

定数

プログラムで記述される具体的な数値、たとえば「893」「3103」といったものは、そのプログラムの製作者は数値の意図を把握していますが、他のプログラマーまたは製作者本人が数値の意図を忘れたときに閲覧すると「数字の意味がわからない」となってしまいます。そのため、このような数値をマジックナンバーと呼ぶことがあります。

マジックナンバーは、

・その数値の持つ意味がわかりづらい
・数値を変更する場合に、複数の箇所を変更しなければならない可能性がある

といった理由から、できるだけ含まない方がよいとされています。

```
zeikomi = kakaku * 1.05;
```

これは、kakakuに商品の消費税抜き価格を代入しておけば、消費税込みの価格が計算されるプログラムです。しかし、「1.05」という数値の意図が不明瞭で、マジックナンバーとみなされてしまいます。また、将来税率が変更された場合、すべての「1.05」を探しだし、訂正しなければなりません。１つでも書き換え忘れると、不具合の原因となり、根の深い問題となりかねません。

これを定数を用いたプログラムに変更すると、税率が変更されても、定数の変更だけですみます。訂正が１カ所ですむので不具合が入り込む可能性が非常に低くなります。Ｃ言語では、定数は#defineで定義します。

```
#define ZEIRITSU 0.05
```

Ｃ＋＋では、定数はenumやconst修飾子で宣言します。enumは整数しか定義できないので、0.05といった小数の場合は、constをつけた変数として定義します。

```
const double ZEIRITSU = 0.05;
```

const修飾子は、書き換えができない変数の宣言です。Ｃ＋＋のプログラムでは、よく使われます。定数を宣言したことにより、税込みの価格を計算するプログラムは、以下のように変わります。

```
zeikomi = kakaku * (1.0 + ZEIRITSU);
```

式に意図が示されるため、可読性が上がり、修正も容易になります。定数の宣言は、名前の選択など、幾分負荷がかかりますが、将来の変更に備える以外にも、デバッグ時に数値の変更がしやすいなど、利点も多くあります。ただし、すべての数値を定数として宣言する必要はありません。

```
#define ZERO 0
#define ONE  1
#define TWO  2
    ⋮
```

のように宣言しても、かえって読みにくく、わかりにくいプログラムとなります。単に定数として宣言しただけでは、定数の意図が伝わらず、マジックナンバーのままとなってしまいます。

「#define」「enum」「(static) const修飾子」の使い分けですが、グローバルな宣言では、どれを使ってもかまいません。クラス宣言内の整数ではenum、関数内では「const修飾子」が適しています。また、「enum」「const修飾子」で定数を宣言すると、コンパイル時に型チェックが行われますので、思わぬ不具合を防ぐことができます(enumの型チェックは使い方によります)。

関数

高速化をねらう場合は除きますが、よく使う処理は関数にまとめます。また、１つの関数が長くなった場合も、複数の関数に分割した方が、可読性が上がり、メンテナンス向上に貢献できます。

前回までの課題で作成したMeteoは、内部処理や描画処理がこのまま拡張を続けると、非常に長いものになってしまいます。プログラムを修正したくなった場合、該当箇所を探すのが手間になってきます。

これをたとえば「プレイヤー移動処理」「隕石移動処理」「プレイヤー描画処理」「隕石描画処理」といった感じで大まかに関数に分割すると、内部処理と描画処理が非常に簡略化され、見渡しが良くなります。見渡しが良くなると、関数の全体像が見やすくなり、メンテナンスもしやすくなります。また、機能毎に担当も割り当てやすくなります(本来は設計段階でしっかりとクラスまたは関数に分割しておきます)。

```
int CGameMain::Update()
{
    // スコア加算
    score++;

    // プレイヤー処理
        ：
        ：

    // 隕石処理
        ：
        ：

    // 衝突検出
        ：
        ：

    return 0;
}
```

関数化すると…

```
int CGameMain::Update()
{
    Score();        // スコア処理
    Player();       // プレイヤー処理
    Meteo();        // 隕石処理
    Collision();    // 衝突検出

    return 0;
}
```

構造体

構造体は、関連のあるデータを１つにまとめて扱うものです。構造体にまとめると、関数の引数など１つの変数で複数のデータを渡すことができるなどの利点があります。

Meteoでは、たとえばプレイヤーのデータは「モデル」「座標」程度しかないため、構造体にまとめる利点は現時点では感じませんが、将来的に「回転角」「ライフ」「残機」といったデータが増える可能性も考えられ、また、関数やファイルを超えてプレイヤーのさまざまなデータを受け渡しされる可能性を考えると、関連のあるデータは構造体にまとめておいた方が良いと考えられます(これも本来は設計段階でクラスまたは構造体にまとめるか検討しておきます)。

構造体の定義は、以下のように行います(Ｃ＋＋の場合)。

```
struct PLAYER {
    MODEL     model;     // モデル
    Vector3   pos;       // 座標
    Vector3   rot;       // 回転角度
        ：
};  // ←セミコロンを忘れない(VC++2010ではエラー表示してくれます)

// 使用例
PLAYER   player;
player.model = GraphicsDevice.CreateModelFromX( TEXT("Model\\Fighter.x") );
```

課　　題

数値を定数、いくつかの処理を関数として定義しましょう。

(１)隕石の数を定数として宣言します。

隕石の数は「100」といったように、プログラム中のいたるところで記述されていますが、100の意図が伝わりにくく、マジックナンバーとなっています。これを定数に変更します。
以下のプログラムを"GameMain.h"の適切な場所に記述しましょう。

```
enum { METEO_MAX = 100 };    // 隕石最大数
```

(２)プログラム中の隕石関連のすべての「100」を「METEO_MAX」に変更しましょう。

(３)「METEO_MAX」に設定している数値を「150」や「200」など、適当な数値に変更し、プログラムが正しく動作するか確認しましょう。

(４)隕石を初期化する処理を関数にしましょう。

「隕石を初期化する処理」はプログラム中に２ヵ所あります。２ヵ所とは、ゲーム起動時に配列すべての隕石を初期化しているところ(Initialize関数内)と、画面外に消えた隕石を再登場させるところ(Update関数内)です。

仕様変更があり(注：本来はプログラム作成前にしっかりと設計しておくべきですが)、隕石初期化の部分を訂正する場合、２ヵ所同じ作業をしなければなりません。修正忘れは不具合の原因となります。よって、関数として１つにまとめます。

関数を定義するには、まずは関数のプロトタイプを宣言します。以下のプログラムを"GameMain.h"の適切な場所に記述しましょう。

```
void InitMeteo(int idx);
```

引数は初期化を行う隕石のインデックス(配列の添字)です。指定された配列要素の隕石の座標、速度を初期化し、再登場に備えます。

(５)関数本体を定義します。"GameMain.cpp"の適切な場所に、以下のプログラムを記述しましょう。

```
//-----------------------------------------------------------------------------
//  指定された添字の隕石を初期化する
//-----------------------------------------------------------------------------
void CGameMain::InitMeteo(int idx)
{
    ※関数の内容は各自考えてください
    ※ヒント：これまでの初期化部分をコピー、添字の変数名の変更
}
```

(６)InitMeteo関数が完成したので、使用するようにプログラムを変更します。Initialize関数の隕石初期化を以下のように変更しましょう。

```
// 隕石初期化
for(int i = 0; i < METEO_MAX; i++) {
    InitMeteo(i);
}
```

(７)Update関数で隕石移動後、画面外に消えた隕石を再登場させるための初期化処理を以下のように変更しましょう。

```
    // 隕石移動
    for(int i = 0; i < METEO_MAX; i++) {
        meteoPos[i].z -= meteoSpd[i];       // 速度の分だけ移動させる
        if(meteoPos[i].z < -2.0f) {         // 隕石が画面外に出たら
            InitMeteo(i);                   // 再登場させるために初期化する
        }
    }
```

応用問題１　プレイヤーの移動処理を関数にしましょう。
応用問題２　プレイヤーの描画処理を関数にしましょう。
応用問題３　隕石の移動処理を関数にしましょう。
応用問題４　隕石の描画処理を関数にしましょう。
応用問題５　衝突判定を関数にしましょう。
応用問題６　プログラム中にマジックナンバーがいくつか残っています。定数として宣言しましょう。
(プレイヤー、隕石の大きさ・移動速度・範囲など)