オブジェクト指向とゲームプログラミング

○　C++編 － 第４回　オブジェクトの生成と破棄　○

オブジェクトの生成

クラスは、定義しただけでは使用できません。というのも、クラスがどのような属性や動作を持つのかを定義しただけであり、属性の内容といったことについては、まだなにも記述していないからです。そこで必要になるのがオブジェクトの生成という作業です。

たとえば、前回のCharacterクラスを利用して、実際にプログラム上で１つずつキャラクターをメモリ上に作成していきます。すると、その１つずつに座標やアニメーション番号を設定することができるようになります。

このように、あるクラスからプログラム上でメモリに生成されるもののことをオブジェクト(object)またはインスタンス(instance)と呼びます。Characterクラスをもとに生成されるオブジェクトは、Characterクラスのオブジェクトと呼ぶことができます。

実際に、プログラムでオブジェクトを生成するには、次の２つの方法があります。

１．オブジェクトを格納する変数を宣言する
２．オブジェクトのポインタ型の変数を宣言し、必要なときにその変数で扱えるようにする

１の方法は、通常のint型やdouble型の変数を宣言するのと同じように、型名の部分にクラス名を使えばよいのです。

```
Character    player;      // Characterクラスのplayerオブジェクトを生成
```

これがCharacterクラスのオブジェクトを扱うための変数playerを用意する文です。変数playerは、Character型の変数と呼ぶことができます。

２の方法は、ポインタ型の変数と後述で示すnew演算子を使い、任意のタイミングでオブジェクトを生成する方法です。

メンバへのアクセス

オブジェクトのメンバへのアクセスは、構造体と同じように、直接メンバ演算子"."で行います。

```
void func()
{
    Character    player;          // Characterクラスのplayerオブジェクトを生成

    // playerオブジェクトのメンバ関数Moveを呼び出す
    player.Move(3.0, 0.0);        // 右に3.0移動
}
```

ポインタがオブジェクトを指している場合は､以下のように間接メンバ演算子"->"でアクセスします。これも構造体と同じです。

```
void func2(Character* pChara)
{
    // ポインタが指すオブジェクトのメンバ関数Moveを呼び出す
    pChara->Move(3.0, 0.0);       // 右に3.0移動
}
```

newとdeleteによるオブジェクトの動的な生成と破棄

C++では、newという演算子でオブジェクトを動的に(プログラマの任意のタイミングで)生成することができます。この演算子は次のように使います。

```
クラス名* 変数名 = new クラス名(あれば引数);
```

newはオブジェクトを生成し、そのオブジェクトが配置されたアドレスを返す演算子です。また、後ろに丸カッコをつけて引数を与えると、その引数の数や型に応じたコンストラクタが呼び出されます。引数をとらないコンストラクタを呼び出したいときは、カッコをつけないか、カッコの中を空にします。

```
Character*   pChara1 = new Chara;                    // デフォルトコンストラクタ
Character*   pChara2 = new Chara(320.0, 240.0);     // 引数付きコンストラクタ
Character*   pChara3 = NULL;    // ポインタの宣言だけしておき、
pChara3 = new Chara(0.0, 0.0);  // 必要なときにnewでオブジェクトを生成することもできる
```

上記の例では、Characterクラスのオブジェクトが３つ生成され、pChara1, pChara2, pChara3はそれらが配置されているアドレスが格納されます。

newで動的に生成したオブジェクトは、delete演算子で破棄します。deleteが実行されると、そのオブジェクトが持つデストラクタが呼び出されます。

```
delete pChara1;     // pChara1の破棄
```

また、NULLを指すポインタに対してdeleteしても何も起こりません。したがって、以下のようなNULLポインタのチェックは不要です。

```
if(pChara1 != NULL)
    delete pChara1;
```

一般には、プログラム終了時にＯＳによって変数などが使用していた領域は解放されますが、newで動的に生成した領域は解放されません(されるＯＳもあります)。動的に確保した領域は、プログラマがその寿命を管理することになっているので、適切なタイミングで破棄します。

newやmalloc関数などによって動的に確保したメモリが解放されない現象を、メモリリークと呼びます。メモリリークはしばしば根の深いバグの原因になるので注意しましょう。これを防ぐ方法の１つに、デストラクタにメモリ解放処理を記述しておくことが挙げられます。

オブジェクトを動的に生成するメリットは、不要になったオブジェクトをいつでも破棄できるというようなコードが簡単に書けるところにあります。たとえば、シューティングゲームなどで、プレイヤーや敵キャラクターのオブジェクトを動的に生成しておき、ゲーム上で破壊されたキャラクターを動的に破棄することにより、メモリ上からもそのオブジェクトを消すことができます。その結果、消されたオブジェクトのメモリが空き、消されたオブジェクトに関する処理も完全に省くことができるようになります。

動的な配列の生成

newを使うと、配列も動的に生成することができます。この場合は以下のように書きます。

```
クラス名* 変数名 = new クラス名[要素数];

Character*   pEnemy = new Character[100];
```

このように生成すると、Characterオブジェクトが100個、配列と同じように連続した領域に生成され、pEnemyにその先頭アドレスが格納されます。
配列を生成するときは、デフォルトコンストラクタ(引数をとらないコンストラクタ)が呼び出されるので、デフォルトコンストラクタを定義しておく必要があります。
このように生成した領域は、普通の配列と同じように扱えます。たとえば第i番目の要素はpEnemy[i]でアクセスすることができます。そのオブジェクトのメンバには、「pEnemy[i].Animation()」というように、直接メンバ演算子"."でアクセスします。

動的に生成された配列の破棄は、delete[]を使って解放します。[]をつけ忘れると、先頭のオブジェクトしか解放されないといったメモリリークが起こるので注意しましょう。

```
delete[] pEnemy;     // 配列pEnemyの解放
```

練習問題

1　以下のプログラムを入力し、コンストラクタとデストラクタがどのように実行されるかを確認しましょう。

```
#include <iostream>

using namespace std;

class CTest {
public:
    CTest()  { cout << "CTestのコンストラクタが呼ばれました" << endl; }
    ~CTest() { cout << "CTestのデストラクタが呼ばれました"   << endl; }
};

void main()
{
    cout << "*** main関数実行" << endl;
    CTest*   pTest;
    cout << "*** ポインタptestが宣言されたました" << endl;
    pTest = new CTest;
    cout << "*** CTestオブジェクトが生成されたました" << endl;
    delete pTest;
    cout << "*** CTestオブジェクトが破棄されたました" << endl;

    cout << endl;
    pTest = new CTest[5];
    cout << "*** CTestオブジェクトの配列が生成されたました" << endl;
    delete[] pTest;
    cout << "*** CTestオブジェクトの配列が破棄されたました" << endl;
}
```

2　newで確保した領域をdeleteしないでプログラムを終了した場合に起こる不具合を説明しましょう。また、そのような症状を起こさないための策を１つ挙げましょう。

3　new[]で確保した領域をdeleteで解放したときに起こる不具合を説明しましょう。

4　newとmalloc関数の違いを説明しましょう。また、deleteとfree関数の違いも説明しましょう。