まとめ

- クラス型のオブジェクトが、同一型のオブジェクトの値で初期化されるときは、**コピーコンストラクタ**によって、すべてのデータメンバの値がコピーされる。この働きを行うコピーコンストラクタは、コンパイラによって自動的に作られて提供される。
- クラス型のオブジェクトに、同一型のオブジェクトの値が代入されるときは、すべてのデータメンバの値がコピーされる。この働きを行う**代入演算子**は、コンパイラによって自動的に作られて提供される。
- コンストラクタを明示的に呼び出す文脈などで、名前をもたない**一時オブジェクト**が生成されることがある。一時オブジェクトは、不要になった時点で自動的に破棄される。
- コンストラクタを含めたメンバ関数は、多重定義できる。
- 引数を与えずに呼び出せるコンストラクタを、**デフォルトコンストラクタ**と呼ぶ。
- 単一の実引数で呼び出せるコンストラクタは、( ) 形式だけでなく、= 形式でも起動できる。
- 等価演算子 **==** あるいは **!=** によって、同一クラス型オブジェクトの状態（全データメンバの値）が等しいかどうかの判定を行うことはできない。
- **イミュータブル**な（const な）オブジェクトに対して、通常のメンバ関数を起動することはできない。そのようなオブジェクトに対しても起動する必要のあるメンバ関数は、**const メンバ関数**として実現する。
- クラスの利用者にとっての定値性とは無関係であり、オブジェクトの内部的な状態を表すようなデータメンバは、**mutable メンバ**とするとよい。
- **文字列ストリーム**は文字列に接続されたストリームであり、挿入子 **<<** と抽出子 **>>** によって、文字の挿入・抽出が可能である。
- クラスに対して**挿入子 <<** や**抽出子 >>** を多重定義すると、入出力のコードが簡潔になる。
- クラスのデータメンバ型が他のクラス型であるとき、**has-A の関係**が成立する。
- オブジェクトに含まれるメンバとしてのオブジェクトのことを、**メンバ部分オブジェクト**と呼ぶ。
- メンバ関数は、自身が所属するオブジェクトを指す **this ポインタ**をもっている。そのため、所属するオブジェクトそのものは、式 ***this** で表せる。
- 関数は、配列を返却することはできないが、クラス型の値は返却できる。

- **コンストラクタ初期化子**によるデータメンバの初期化は、コンストラクタ本体の実行に先立って行われる。
- コンストラクタ本体内でのデータメンバへの値の設定は、初期化ではなく代入である。クラス型のメンバは、コンストラクタ本体内で値を代入するのではなく、コンストラクタ初期化子によって初期化すべきである。
- クラス定義を含むヘッダは、何度インクルードされてもコンパイルエラーとならないように**インクルードガード**を施さなければならない。

chap02/Point2D.h

```
#ifndef ___Point2D
#define ___Point2D
//--- ２次元座標クラス ---//
class Point2D {
    int xp, yp;      // Ｘ座標とＹ座標
public:
    Point2D(int x = 0, int y = 0) : xp(x), yp(y) { }
    int x() const { return xp; }                  // Ｘ座標
    int y() const { return yp; }                  // Ｙ座標
    void print() const { std::cout << "(" << xp << "," << yp << ")"; } // 表示
};
#endif
```

コンストラクタ初期化子

chap02/Circle.h

```
#ifndef ___Circle
#define ___Circle
#include "Point2D.h"
//--- 円クラス ---//
class Circle {
    Point2D center;      // 中心座標
    int radius;          // 半径
public:
    Circle(const Point2D& c, int r) : center(c), radius(r) { }
    Point2D get_center() const { return center; }   // 中心座標
    int get_radius() const { return radius; }       // 半径
    void print() const {                            // 表示
        std::cout << "半径[" << radius << "] 中心座標";  center.print();
    }
};
#endif
```

コンストラクタ初期化子

Circle has a Point2D.

chap02/CircleTest.cpp

```
#include <iostream>
#include "Point2D.h"
#include "Circle.h"
using namespace std;
int main()
{
    Point2D origin(0, 0);            // 原点
    Circle c1(Point2D(3, 5), 7);     // 中心座標(3, 5) 半径7の円
    Circle c2(Point2D(), 8);         // 中心座標(0, 0) 半径8の円
    Circle c3(origin, 9);            // 中心座標(0, 0) 半径9の円
    cout << "c1 = ";   c1.print();   cout << '\n';
    cout << "c2 = ";   c2.print();   cout << '\n';
    cout << "c3 = ";   c3.print();   cout << '\n';
}
```

実行結果

```
c1 = 半径[7] 中心座標(3,5)
c2 = 半径[8] 中心座標(0,0)
c3 = 半径[9] 中心座標(0,0)
```

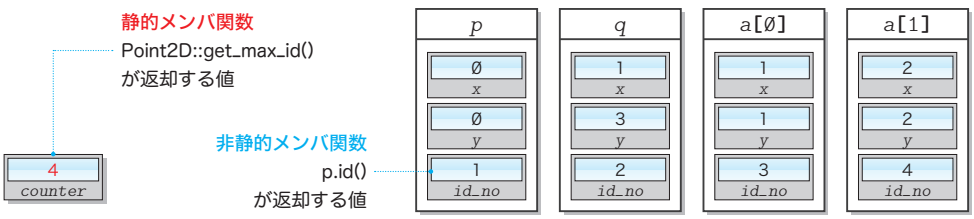
- クラス定義の中で **static** を付けて宣言されたデータメンバは、**静的データメンバ**となる。
- クラス定義の中での静的データメンバの宣言は、実体の定義ではない。実体の定義は、クラス定義の外で、**static** を付けずに行う。
- 個々のオブジェクトに所属する**非静的データメンバ**は、個々のオブジェクトの状態（ステート）を表すのに適している。
  それに対して、**静的データメンバ**は、そのクラスに所属している全オブジェクトで共有するデータを表すのに適している。
- 静的データメンバは、そのクラス型のオブジェクトの個数とは無関係に（たとえオブジェクトが存在しなくても）、１個のみが存在する。
- 静的データメンバのアクセスは、“オブジェクト名 **.** データメンバ名”によっても行えるが、“クラス名 **::** データメンバ名”で行うべきである。
- クラス定義の中で **static** を付けて宣言されたメンバ関数は、**静的メンバ関数**となる。
- 静的メンバ関数の定義をクラス定義の外に置く場合は、**static** を付けてはならない。
- 個々のオブジェクトに所属する**非静的メンバ関数**は、個々のオブジェクトの振舞いを表すのに適している。
  それに対して、**静的メンバ関数**は、クラス全体に関わる処理や、クラスのオブジェクトの状態とは無関係な処理を実現するのに適している。
- 静的メンバ関数の呼出しは、“オブジェクト名 **.** メンバ関数名 **(...)**”によっても行えるが、“クラス名 **::** メンバ関数名 **(...)**”で行うべきである。
- 静的メンバ関数は、特定のオブジェクトに所属しないため、**this** ポインタをもたない。
- 静的データメンバの初期化は、それを定義するソースファイルの中で初めて利用される時点までに完了することになっており、**main** 関数の実行前に初期化が完了する保証はない。
- 静的データメンバを初めてアクセスする箇所が、静的データメンバの定義を含むソースファイル以外のソースファイルである場合、静的データメンバが未初期化の状態でのアクセスを行う危険性がある。
- 静的データメンバの定義と、それをアクセスするすべてのメンバ関数の定義は、単一のソースファイルにまとめるべきである。
- 同一名のメンバ関数を定義する多重定義は、静的メンバ関数と非静的メンバ関数とにまたがって行える。

chap02/static/Point2D.h

```
#ifndef ___Point2D
#define ___Point2D
#include <iostream>
//--- 識別番号付き２次元座標クラス ---//
class Point2D {
    int xp, yp;                     // Ｘ座標とＹ座標
    int id_no;                      // 識別番号
    static int counter;             // 何番までの識別番号を与えたか【宣言】
public:
    Point2D(int x = 0, int y = 0);      // コンストラクタ【宣言】
    int id() const { return id_no; }    // 識別番号
    void print() const {                // 座標の表示
        std::cout << "(" << xp << "," << yp << ")";
    }
    static int get_max_id();            // 識別番号の最大値を返却【宣言】
};
#endif
```

chap02/static/Point2D.cpp

```
#include "Point2D.h"
int Point2D::counter = 0;       // 何番までの識別番号を与えたか【定義】
//--- コンストラクタ【定義】---//
Point2D::Point2D(int x, int y) : xp(x), yp(y) {
    id_no = ++counter;          // 識別番号を与える
}
//--- 識別番号の最大値を調べる【定義】---//
int Point2D::get_max_id() {
    return counter;             // 識別番号の最大値を返却
}
```



静的データメンバ
オブジェクトとは無関係に１個のみ存在

非静的データメンバ
個々のオブジェクトに１個ずつ存在

chap02/static/Point2DTest.cpp

```
#include <iostream>
#include "Point2D.h"
using namespace std;
int main()
{
    Point2D p;
    Point2D q(1, 3);
    Point2D a[] = {Point2D(1, 1), Point2D(2, 2)};
    cout << "最後に与えた識別番号：" << Point2D::get_max_id() << '\n';
    cout << "p    = ";   p.print();   cout << "  識別番号：" << p.id() << '\n';
    cout << "q    = ";   q.print();   cout << "  識別番号：" << q.id() << '\n';
    for (int i = 0; i < sizeof(a) / sizeof(a[0]); i++) {
        cout << "a[" << i << "] = ";   a[i].print();
        cout << "  識別番号：" << a[i].id() << '\n';
    }
}
```

実行結果

```
最後に与えた識別番号：4
p    = (0,0)  識別番号：1
q    = (1,3)  識別番号：2
a[0] = (1,1)  識別番号：3
a[1] = (2,2)  識別番号：4
```