# 第１章

# クラスの基本

本章では、クラスの基本を学習します。

- クラスとデータ隠蔽とカプセル化とユーザ定義型
- データメンバと状態
- メンバ関数とメッセージと振舞いと this ポインタ
- コンストラクタとコンストラクタ初期化子
- アクセス指定子（private ／ protected ／ public）
- セッタとゲッタとアクセッサ
- クラス有効範囲
- 静的データメンバと静的メンバ関数
- 演算子の多重定義
- ヘッダと using 指令
- const メンバ関数と const データメンバ
- デフォルトコンストラクタ
- 変換コンストラクタと変換関数／ユーザ定義変換
- コピーコンストラクタ／明示的コンストラクタ
- デストラクタ
- フレンド関数

注意：

本章は、基礎的な知識を前提として、『入門編』の第 10 章以降の内容の一部を圧縮して復習する内容となっています。もし、本章の内容を難しく感じられるのであれば、ぜひ『入門編』に戻って学習してください。

# 1–1 クラスによるカプセル化



『入門編』では第 10 章以降、クラスについて学習しました。本章では、クラスに関する基本的なことがらを、ざっと学習（復習）します。

## クラスによるカプセル化

私たちは、プログラム作成時に、現実世界の物や概念をプログラムの世界に投影します。その際に、ひとまとまりのデータとそれを処理する関数（手続き）をまとめて投影を行う、というのが、**クラス**（*class*）の考え方の基本です。

**List 1-1** に示すのは、人間クラス *Human* のプログラムです。点線で囲んだ部分が、人間クラス *Human* の**クラス定義**（*class definition*）です。クラス定義は、キーワード **class** で始まって、} の後ろの ; まで続きます。

▶ Java とは異なり、クラス定義の末尾には必ずセミコロン ; が必要です。また、キーワード **class** の代わりに **struct** を使って定義することもできます。

青網部は**データメンバ**（*data member*）の宣言で、黒網部は**メンバ関数**（*member function*）の宣言です。データメンバとメンバ関数は、クラスを構成する代表的な要素であり、その総称は**メンバ**（*member*）です。

データメンバに先立つ **private:** は、それ以降に宣言するメンバを、クラスの外部に対して《非公開》にするための指示です。クラス *Human* では、氏名を表す *full_name*、身長を表す *height*、体重を表す *weight* のすべてのデータメンバが非公開となっています。

みなさんは、各種のパスワードを秘密にしているはずです。データを外部から隠して不正なアクセスから守ることを**データ隠蔽**（*data hiding*）といいます。**すべてのデータメンバは、非公開とするのが原則です。**

メンバ関数に先立つ **public:** は、それ以降に宣言するメンバを、クラスの外部に対して《公開》するための指示です。クラス *Human* では、6 個のメンバ関数のすべてが公開されています。

キーワード **public** と **private** は、**アクセス指定子**（*access specifier*）と呼ばれます。この他に、第 4 章で学習する **protected** というアクセス指定子があります。なお、**public:** や **private:** の指定がない場合、メンバは原則として《非公開》となります。

▶ ただし、キーワード **struct** を使って定義されたクラスでは、アクセス指定のないメンバのアクセス性は《公開》となります。

*

クラス名と同じ名前のメンバ関数が**コンストラクタ**（*constructor*）です。コンストラクタの役目は、オブジェクトを適切に初期化することです。その性格上、コンストラクタに返却値型を与えることはできません（**void** と宣言することもできません）。

List 1-1 [A] Human1/Human.cpp

```
// 人間クラスHuman（第１版）とその利用例

#include <string>
#include <iostream>

using namespace std;

//===== 人間クラス =====//
class Human {
private:
    string full_name;   // 氏名
    int height;         // 身長
    int weight;         // 体重

public:
    //--- コンストラクタ ---//
    Human(const string& name, int h, int w) {
        full_name = name;       // 氏名
        height = h;             // 身長
        weight = w;             // 体重
    }

    //--- 氏名を調べる ---//
    string name() {
        return full_name;
    }

    //--- 身長を調べる ---//
    int get_height() {
        return height;
    }

    //--- 体重を調べる ---//
    int get_weight() {
        return weight;
    }

    //--- 太る ---//
    void grow_fat(int dw) {
        weight += dw;               // 体重がdwだけ増える
    }

    //--- やせる ---//
    void slim_off(int dw) {
        weight -= dw;               // 体重がdwだけ減る
    }
};
```

クラス定義

データメンバ

メンバ関数

▶ この記号は、プログラムリストに“続き”があることを示しています。

construct は『構築する』という意味です。そのため、コンストラクタは**構築子**と呼ばれることもあります。

さて、クラス *Human* には、コンストラクタ以外に五つのメンバ関数があります。

- *name* ：氏名を **string** 型で返す。
- *get_height* ：身長を **int** 型で返す。
- *get_weight* ：体重を **int** 型で返す。
- *grow_fat* ：太る（体重が増える）。
- *slim_off* ：やせる（体重が減る）。

クラス *Human* が、3 個のデータと 6 個の関数をまとめたものであることが分かりました。ただし、クラスは、たこ焼きでいうところの「カタ」に相当します。食べることのできる本当のたこ焼きであるオブジェクトは、カタから作らなければなりません。

## コンストラクタとメンバ関数

クラス *Human* のオブジェクトを生成するのが、**main** 関数冒頭に置かれた **1** と **2** の宣言文です。プログラムの流れが宣言文を通過して、網かけ部の式が評価される際に、コンストラクタが呼び出されて実行されます。

**Fig.1-1** に示すのは、宣言 **1** によってオブジェクト *nakano* に対して呼び出されたコンストラクタの動作イメージです。仮引数 *name*, *h*, *w* に受け取った三つの値を、それぞれデータメンバ *full_name*, *height*, *weight* に代入します。

コンストラクタを含むメンバ関数は、そのクラスにとって内輪（うちわ）の存在ですから、非公開のデータメンバにも自由にアクセスできます。

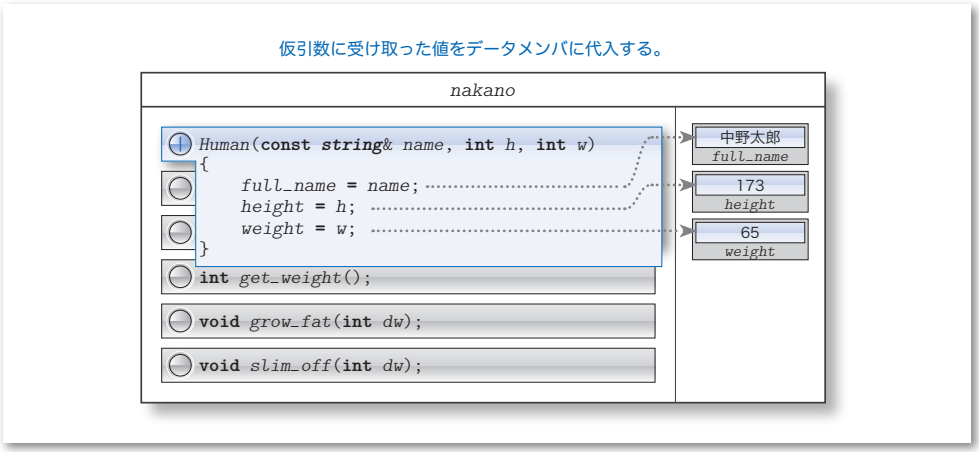


**Fig.1-1**　コンストラクタによるオブジェクトの初期化

**main** 関数に戻りましょう。オブジェクト *nakano* と *morita* の生成後は、中野君が 3kg やせる処理と森田君が 7kg 太る処理を行い、それから、二人のデータを調べて表示する処理を行っています。いずれの処理も、メンバ関数の呼出しによって行われています。

メンバ関数の呼出しは、**ドット演算子**（*dot operator*）**.** を適用した以下の形式です。

```
変数名 . メンバ関数名 ( 実引数 )
```

なお、（本プログラムにはありませんが）通常の変数でなくポインタであれば、ドット演算子 **.** の代わりに**アロー演算子**（*arrow operator*）**->** を利用した以下の形式となります。

```
ポインタ名 -> メンバ関数名 ( 実引数 )
```

二つの演算子の総称が**クラスメンバアクセス演算子**（*class member access operator*）です。

▶ ドット演算子とアロー演算子は、データメンバに対しても適用できます。もし、クラス *Human* のデータメンバが公開されていれば、森田君の身長は式 *morita.height* でアクセスできますし、*Human** 型のポインタ *p* が *morita* を指していれば、森田君の体重は式 *p->weight* でアクセスできます（冗長となりますが、式 *(*p).weight* でもアクセスできます）。

**List 1-1 [B]** Human1/Human.cpp

```
int main()
{
    Human nakano("中野太郎", 173, 65);   // 中野君
    Human morita("森田孝司", 168, 71);   // 森田君

    nakano.slim_off(3);      // 中野君が3kgやせる
    morita.grow_fat(7);      // 森田君が7kg太る

    cout << "nakano = " << nakano.name() << "  " << nakano.get_height() << "cm  "
         << nakano.get_weight() << "kg\n";

    cout << "morita = " << morita.name() << "  " << morita.get_height() << "cm  "
         << morita.get_weight() << "kg\n";
}
```

実行結果

```
nakano = 中野太郎  173cm  62kg
morita = 森田孝司  168cm  78kg
```

たとえば、3では、*morita* に対してメンバ関数 *get_weight* を呼び出しています。呼び出された関数の動作イメージを示したのが、**Fig.1-2** です。メンバ関数 *get_weight* は、オブジェクト *morita* のデータメンバ *weight* の値を調べて、そのまま返却します。

クラスの外部から直接アクセスできない身長や体重などのデータも、メンバ関数を通じて間接的にアクセスできるわけです。

なお、*get_height* や *get_weight* のように、単一のデータメンバの値を取得して返却するメンバ関数は、**ゲッタ**（*getter*）と呼ばれます。

なお、ゲッタとは逆に、データメンバに特定の値を設定するメンバ関数は**セッタ**（*setter*）と呼ばれます。また、ゲッタとセッタの総称が**アクセッサ**（*accessor*）です。

▶ データメンバとメンバ関数が同一名となることは許されないため、データメンバ名と、そのセッタを同じ名前にすることはできません。なお、クラス *Human* には、ゲッタはありますが、セッタはありません。

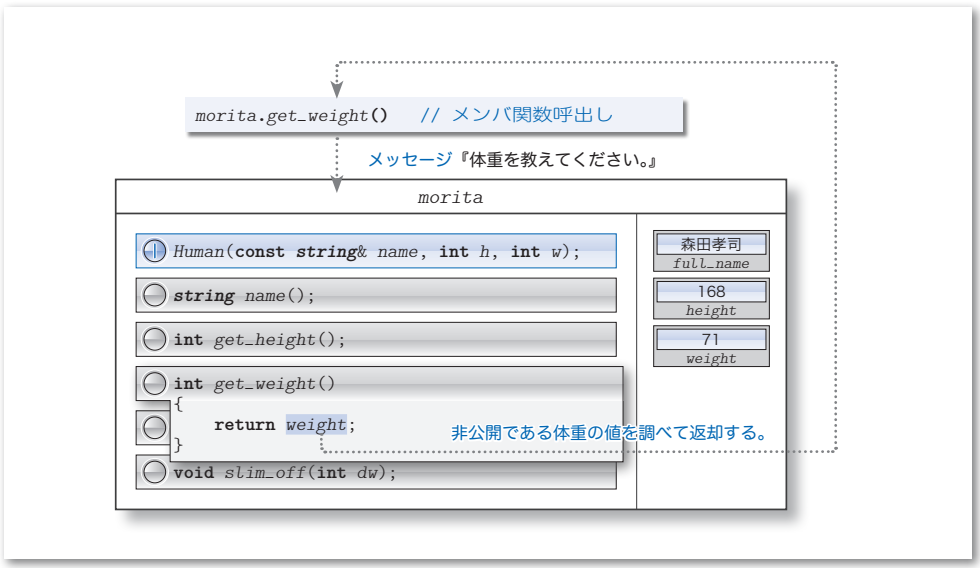


**Fig.1-2**　メンバ関数の呼出し

## メッセージ

**Fig.1-1** と **Fig.1-2** の二つの図が示すように、オブジェクト *nakano* と *morita* のそれぞれが、3 個のデータメンバと 6 個のメンバ関数を有しています。

▶ コンストラクタを含むメンバ関数が個々のオブジェクトごとに作られるというのは、あくまでも概念上のことです。コンパイルの結果作られる内部的なコードは 1 個のみです。

オブジェクト指向プログラミングの世界では、メンバ関数は**メソッド**（*method*）とも呼ばれます。また、メンバ関数を呼び出すことは、次のように表現されます。

> **オブジェクトに“メッセージを送る”。**

たとえば、プログラム3のメンバ関数の呼出し式 *morita.get_weight*() は、オブジェクト *morita* に対して『体重を教えてください。』というメッセージを送っています。

その結果、オブジェクト *morita* は「体重を返却してあげればいいのだな。」と能動的に意思決定を行って、『○○ kg ですよ。』と返答処理を行います。

なお、クラス定義の中で定義されたメンバ関数は、内部結合をもった**インライン関数**（*inline function*）となります。そのため、関数呼出しの負荷は（原則として）ありません。

▶ インライン関数が、必ずしもインラインに展開されるとは限らないことは『入門編』の第 6 章で学習しました。メンバ関数も同様です。

**Fig.1-1** 内のメンバ関数はオブジェクト *nakano* に所属し、**Fig.1-2** 内のメンバ関数はオブジェクト *morita* に所属しています。一般に、メンバ関数が行う処理は、自分が所属するオブジェクトのデータメンバの値をもとに処理を行ったり、データメンバの値を更新したりすることです。メンバ関数とデータメンバは、緻密に連携しているわけです。

▶ たとえば、*nakano.get_weight*() は、オブジェクト *nakano* のデータメンバ *weight* の値を調べて返却し、*morita.grow_fat*(7) は、*morita* のデータメンバ *weight* の値を 7 だけ増やします。

クラスを「回路」の《設計図》であると考えましょう。そうすると、その設計図に基づいて作られた実体としての《回路》が、クラス型のオブジェクトです。

回路であるオブジェクトのパワーを起動するとともに、受け取った氏名と身長と体重をデータメンバにセットするのがコンストラクタです。コンストラクタは、回路を起動するための《電源ボタン》によって呼び出されるチップ＝小型の回路と考えられます。

そして、データメンバの値は、その回路＝オブジェクトの現在の状態を表します。そのため、データメンバは、**ステート**（*state*）とも呼ばれます。

▶ state は『状態』という意味です。たとえば、データメンバ *weight* は、現在の体重が何 kg なのか、という状態を **int** 型の整数値として表します。

一方、メンバ関数は回路の**振舞い**（*behavior*）を表します。各メンバ関数は、回路の現在のステート（状態）を調べたり、変更するためのチップです。

整数や実数などの数値を表現する **int** 型や **double** 型などの、プログラミング言語 C++ によって提供される型は、**組込み型**（*built-in type*）と呼ばれます。それに対して、クラス

*Human* のような型は、**ユーザ定義型**（*user-defined type*）と呼ばれます。

なお、データメンバを非公開として外部から保護した上で、そのデータに対する処理を行うメンバ関数と連携させることを、**カプセル化**（*encapsulation*）といいます。

▶ 成分を詰めて、それが有効に働くようにカプセル薬を作ること、と考えればいいでしょう。

## ヘッダとソースの分離

第 1 版の人間クラス *Human* では、すべてのメンバ関数の定義が、クラス定義の中に埋め込まれています。しかし、大規模なクラスであれば、単一のソースファイルでクラスのすべてを管理するのは困難です。そのため、コンストラクタを含めたメンバ関数の定義は、クラス定義の外でも行えるようになっています。ただし、クラス定義の外でメンバ関数の定義を行う場合でも、宣言だけはクラス定義の中に必要です。

コンストラクタを含め、クラス定義の外で定義するメンバ関数は、以下に示す形式で定義します。

```
返却値型 クラス名 :: メンバ関数名 ( 仮引数宣言節 ) { /* … */ }
```

名前の前に“クラス名 ::”を付けるのは、宣言するメンバ関数が**クラス有効範囲**（*class scope*）の中にあることを示すためです。

たとえば、メンバ関数 *get_weight* をクラス定義の外で定義するのであれば、その宣言・定義は、右のようになります。

```
class Human {
   //...
   int get_weight();  // 宣言
   //...
};

// 定義
int Human::get_weight()
{
   return weight;
}
```

なお、クラス定義の外で定義したメンバ関数は、インライン関数として扱われなくなり、外部結合が与えられます。

*

ある程度以上の規模のクラスであれば、**クラス定義をヘッダとして実現して、メンバ関数の定義を別のソースプログラムとして実現するのが一般的です。**

コンストラクタを含む全メンバ関数をクラス定義から分離して、独立したソースプログラムとして実現してみましょう。それが、次ページ以降に示す **List 1-2** と **List 1-3** のプログラムです。本書では、前者を**ヘッダ部**と呼び、後者を**ソース部**と呼ぶことにします。

なお、本プログラムでは、ソース部を独立したこと以外にも、いくつかの変更・修正を加えています。それらの点を学習していきましょう。

▶ ヘッダ部のほぼ全体を `#ifndef` 指令と `#endif` 指令で囲んでいるのは、複数回インクルードしてもクラスの重複定義とならないようにするための工夫です。詳細は、**Column 2-6**（p.81）で学習します。

List 1-2 Human2/Human.h

```
// 人間クラスHuman（第２版）ヘッダ部
#ifndef ___Class_Human
#define ___Class_Human

#include <string>

//===== 人間クラス =====//
class Human {
private:
    std::string full_name;  // 氏名
    int height;             // 身長
    int weight;             // 体重

public:
    Human(const std::string& full_name, int height, int weight);  // コンストラクタ
    std::string name() const;       // 氏名を調べる
    int get_height() const;         // 身長を調べる
    int get_weight() const;         // 体重を調べる
    void grow_fat(int dw);          // 太る
    void slim_off(int dw);          // やせる
};

#endif
```

### ヘッダ部と using 指令

ヘッダ部には“**using namespace** std;”の**using**指令がありません。そのため、文字列を表す***string***型を、std**::*****string***として表しています。

というのも、もし**using**指令をヘッダ中に記述すると、そのヘッダをインクルードするソースファイルで、**using**指令が（勝手に）有効になってしまうからです。クラスのすべての利用者が、必ずしも、そのような状況を好むわけではありません。**ヘッダの中には、using指令を置かないのが原則です。**

そのため、ヘッダ内の標準ライブラリは、std::***string***やstd::coutなどのフルネームで表すことになります。

▶ std::***string***とstd::coutについては、『入門編』で簡単に学習しました。本書では、それぞれ第11章と第12章でさらに詳しく学習します。

## this ポインタ

コンストラクタ本体の網かけ部には、“**this ->** メンバ名”という形式の式があります。**this とは、コンストラクタやメンバ関数が所属するオブジェクトを指すポインタです。**

クラス*C*型のオブジェクトのメンバ関数における**this**の型は*C**です。

▶ ただし、メンバ関数が**const**宣言されていれば**this**の型は**const** *C**、**volatile**宣言されていれば**volatile** *C**、**const volatile**宣言されていれば**const volatile** *C**となります。

アロー演算子**->**の左オペランドである**this**は自分自身のオブジェクトを指すポインタですから、**this->***full_name*は、自分自身のオブジェクトに所属するデータメンバ*full_name*を表します。もちろん、**this->***height*と**this->***weight*も同様です。

List 1-3 Human2/Human.cpp

```
// 人間クラスHuman（第２版）ソース部

#include <string>
#include <iostream>
#include "Human.h"

using namespace std;

//--- コンストラクタ ---//
Human::Human(const string& full_name, int height, int weight)
{
    this->full_name = full_name;          // 氏名
    this->height = height;                // 身長
    this->weight = weight;                // 体重
}

//--- 氏名を調べる ---//
string Human::name() const
{
    return full_name;
}

//--- 身長を調べる ---//
int Human::get_height() const
{
    return height;
}

//--- 体重を調べる ---//
int Human::get_weight() const
{
    return weight;
}

//--- 太る ---//
void Human::grow_fat(int dw)
{
    weight += dw;                 // 体重がdwだけ増える
}

//--- やせる ---//
void Human::slim_off(int dw)
{
    weight -= dw;                 // 体重がdwだけ減る
}
```

さて、コンストラクタが受け取っている三つの仮引数 *full_name*, *height*, *weight* は、データメンバと同一の名前です。**データメンバと同じ名前のものが、仮引数として、あるいは、メンバ関数中の局所的な変数として宣言されると、データメンバの名前が隠されて、宣言されたほうの変数の名前が見えることになっています。**

本コンストラクタでは、**this->** の有無で同一の名前を以下のように使い分けています。

- *full_name* … 仮引数として受け取ったもの。
- **this->***full_name* … 所属するオブジェクト内のデータメンバ。

▶ このテクニックを利用すると、プログラム開発者は、データメンバと一対一に対応する仮引数をどのように命名すべきかを悩まなくて済むことになります。その一方で、コンストラクタやメンバ関数内において、データメンバをアクセスする式に **this->** を付け忘れることで生じるバグを導く可能性が発生します。

## const メンバ関数

メンバ関数 *name*，*get_height*，*get_weight* は、ヘッダ部の宣言とソース部の定義の両方の関数頭部の後ろに、キーワード **const** が付いています。このように宣言されたメンバ関数は、**const メンバ関数**（*const member function*）と呼ばれます。

あるメンバ関数が、所属オブジェクトの状態（データメンバの値）を変更するかどうかの判断は、関数の中身を調べ尽くさない限り不可能です。その判断をコンパイル時や実行時に行うと、コストが高くつきます。そのため、オブジェクトの状態を変更する可能性のある通常の（すなわち **const** でない）メンバ関数は、原則として定値オブジェクトに対しては、呼び出せないようになっています。

一方、**const** メンバ関数は、状態を変更しないことを表明したメンバ関数です。そのため、定値オブジェクトとそうでないオブジェクトの両方に対しても呼び出せます。

一般に、クラスの利用者が **const** なオブジェクトを作るかどうかは、クラスの作成者には分かりません。そのため、**オブジェクトの状態（データメンバの値）を変更しないメンバ関数は、const メンバ関数として実現するのが原則です**。

以上のことを、第２版の人間クラス *Human* を利用するプログラム例である **List 1-4** で確認しましょう。

**List 1-4** Human2/HumanTest.cpp

```
// 人間クラスHuman（第２版）とその利用例

#include <iostream>
#include <string>
#include "Human.h"

using namespace std;

int main()
{
    Human nakano("中野太郎", 173, 65);          // 中野君（普通のオブジェクト）
    const Human morita("森田孝司", 168, 71);    // 森田君（定値オブジェクト）

    nakano.slim_off(3);             // 中野君が3kgやせる
//  morita.grow_fat(7);             // エラー：constな森田君は太れない

    cout << "nakano = " << nakano.name() << "  " << nakano.get_height() << "cm  "
         << nakano.get_weight() << "kg\n";

    cout << "morita = " << morita.name() << "  " << morita.get_height() << "cm  "
         << morita.get_weight() << "kg\n";
}
```

**実行結果**

```
nakano = 中野太郎  173cm  62kg
morita = 森田孝司  168cm  71kg
```

第１版と同様、**const** でない *nakano* に対しては、すべてのメンバ関数が適用可能です。その一方で、定値オブジェクト *morita* には、**const** メンバ関数である *name*，*get_height*，*get_weight* は適用可能ですが、**const** でないメンバ関数 *grow_fat* を適用する式はコンパイルエラーとなります。

▶ 本プログラムでは、コンパイルエラーの発生を抑止するために、該当部をコメントアウトしています。

## コンストラクタ初期化子

クラス *Human* のコンストラクタは、氏名、身長、体重の三つの値を受け取って、それらを各メンバに《代入》します。そのため、コンストラクタの実行時は、

- データメンバ *full_name* は、いったん空の文字列で初期化されて、その後、仮引数 *full_name* に受け取った文字列が代入される。
- データメンバ *height* と *weight* は、いったん不定値で初期化されて、その後、仮引数 *height* と *weight* に受け取った整数値の代入が行われる。

と、《初期化》と《代入》という２段階での値の設定が行われてしまいます。

これを避けるコンストラクタの宣言を以下に示します。

▶ この例では、第１版と同様に、クラス定義の中でコンストラクタを定義しています。

```
class Human {
    // …
public:
    //--- コンストラクタ ---//
    Human(const string& name, int h, int w)
        : full_name(name), height(h), weight(w)
    { }
    // …
};
```

コロン：で始まる網かけ部は、**コンストラクタ初期化子**（*constructor initializer*）と呼ばれます。そして、コンストラクタ初期化子内の *full_name*(*name*) と *height*(*h*) と *weight*(*w*) の部分は、**メンバ初期化子**（*member initializer*）と呼ばれます。メンバ初期化子はコンマ , で区切らなければなりません。

さて、メンバ初期化子“メンバ名（仮引数名）”を指定すると、仮引数に受け取った値でデータメンバを初期化する作業がコンストラクタ実行開始時に行われます。そのため、データメンバ *full_name* は *name* で初期化され、*height* は *h* で初期化され、*weight* は *w* で初期化されます。各データメンバへの値の設定作業は、１回の《初期化》で済みます。

コンストラクタ初期化子による初期化の処理は、コンストラクタ本体の実行が開始される前に完了します。この例では、コンストラクタ本体で行うべきことがなくなってしまっているため、コンストラクタ本体を空にしています。

なお、**データメンバは、クラス定義中のデータメンバの宣言順で初期化されます**。コンストラクタ初期化子におけるメンバ初期化子の並びの順序とは無関係です。

▶ たとえば、コンストラクタ初期化子が“: *height*(*h*), *full_name*(*name*), *weight*(*w*)”となっていても、データメンバの初期化は *full_name* ➡ *height* ➡ *weight* の順で行われます。

### 演習 1–1

クラス *Human* のコンストラクタを本ページに示したように書きかえたプログラムを作成し、初期化が期待どおりに行われることを確認せよ。

# 1–2 日付クラス



本節では、日付クラスを例に、静的メンバ、デフォルトコンストラクタ、変換コンストラクタ、コピーコンストラクタ、演算子の多重定義、フレンドなどを学習（復習）します。

## 静的メンバ

次に学習するのは、日付クラス *Date* です。このクラスのヘッダ部は **List 1-5** で、ソース部は **List 1-6**（p.14）です。

クラス *Date* の定義冒頭では、データメンバが宣言されています。これらのデータメンバ *y*, *m*, *d* は、それぞれ西暦年・月・日を表します。

### 静的データメンバ

日付に関する種々の計算では、

- 任意の年の日数（365 日なのか 366 日なのか）
- 任意の年月の日数（28 日、29 日、30 日、31 日のいずれなのか）

を知る必要があります。その実現のために、本クラスでは、配列 *dmax* と、いくつかの関数を利用しています。

配列 *dmax* の宣言にはキーワード **static** が付いています。そのため、*dmax* は、**静的データメンバ**（*static data member*）と呼ばれるデータメンバとなります。クラス *Date* 型を利用するプログラム内で、*Date* 型のオブジェクトがいくつ生成されても（たとえ 1 個も生成されなくても）、**そのクラスに所属する静的データメンバの実体は、1 個だけ作られます。**

▶ もし、*dmax* の宣言に **static** を付けずに非静的データメンバとすると、*Date* 型の全オブジェクトがまったく同じ配列 *dmax* を重複して含むことになり、各オブジェクトが肥大化します。

原則として、静的データメンバの実体は、“クラス名 :: データメンバ名”という形式で、クラス定義の外で **static** を付けずに定義することになっています。配列 *dmax* は、ソース部で以下のように定義されています（p.14）。

```
int Date::dmax[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

クラス定義の中の静的データメンバの宣言には **static** が必要ですが、クラス定義の外の定義では **static** は不要です。もし付けると、コンパイルエラーとなります。

なお、配列 *dmax* は、非公開ですから、クラスの外部からのアクセスは不可能です。

▶ もし、配列 *dmax* が公開されていれば、クラスの外部からは “*Date*::*dmax*” としてアクセスできることになります。

*

**特定のオブジェクトではなく、クラスに共通するデータは静的データメンバとして実現するのが原則です。**

List 1-5 Date/Date.h

```
// 日付クラスDate（ヘッダ部）

#ifndef ___Class_Date
#define ___Class_Date

#include <string>
#include <iostream>

//===== 日付クラス =====//
class Date {
    int y;  // 西暦年
    int m;  // 月
    int d;  // 日

    static int dmax[];
    static int days_of_year(int year);              // year年の日数
    static int days_of_month(int year, int month);  // year年month月の日数
public:
    //--- year年は閏年か？ ---//
    static bool leap_year(int year) {
        return year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
    }

    Date();                                     // デフォルトコンストラクタ
    Date(int yy, int mm = 1, int dd = 1);       // コンストラクタ

    bool leap_year() const { return leap_year(y); }     // 閏年か？

    int year()  const { return y; }         // 年を返却
    int month() const { return m; }         // 月を返却
    int day()   const { return d; }         // 日を返却

    void set_year( int yy) { y = yy; }      // 年をyyに設定
    void set_month(int mm) { m = mm; }      // 月をmmに設定
    void set_day(  int dd) { d = dd; };     // 日をddに設定
    void set(int yy, int mm, int dd);       // 日付をyy年mm月dd日に設定

    Date preceding_day() const;             // 前日の日付を返却
    Date following_day() const;             // 翌日の日付を返却

    int day_of_year() const;                // 年内の経過日数を返却

    int day_of_week() const;                // 曜日を返却

    operator long() const;                  // 1970年1月1日からの日数

    Date& operator++();                         // １日進める（前置増分）
    Date  operator++(int);                      // １日進める（後置増分）

    Date& operator--();                         // １日戻す　（前置減分）
    Date  operator--(int);                      // １日戻す　（後置減分）

    Date& operator+=(int dn);                   // dn日進める（Date += int）
    Date& operator-=(int dn);                   // dn日戻す　（Date -= int）

    Date operator+(int dn) const;                   // dn日後を求める（Date + int）
    friend Date operator+(int dn, const Date& day); // dn日後を求める（int + Date）

    Date operator-(int dn) const;               // dn日前を求める　（Date - int）

    long operator-(const Date& day) const;      // 日付の差を求める（Date - Date）

    bool operator==(const Date& day) const;     // dayと同じ日か？
    bool operator!=(const Date& day) const;     // dayと違う日か？

    bool operator> (const Date& day) const;     // dayより後か？
    bool operator>=(const Date& day) const;     // day以降か？
    bool operator< (const Date& day) const;     // dayより前か？
    bool operator<=(const Date& day) const;     // day以前か？

    std::string to_string() const;              // 文字列表現を返却
};

std::ostream& operator<<(std::ostream& s, const Date& x);   // 挿入子
std::istream& operator>>(std::istream& s,       Date& x);   // 抽出子

#endif
```

List 1-6 [A] Date/Date.cpp

```
// 日付クラスDate（ソース部）

#include <ctime>
#include <sstream>
#include <iostream>
#include "Date.h"

using namespace std;

// 平年の各月の日数
int Date::dmax[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

//--- year年の日数（平年…365／閏年…366）---//
int Date::days_of_year(int year)
{
    return 365 + leap_year(year);
}

//--- y年m月の日数を求める ---//
int Date::days_of_month(int year, int month)
{
    return dmax[month - 1] + (month == 2 && leap_year(year));
}
```

## 静的メンバ関数

配列 *dmax* を利用して、年の日数と年月の日数の計算を行うのが、*days_of_year*、*days_of_month*、*leap_year* の三つのメンバ関数です。

最初の二つの関数は、他のメンバ関数の下請け的な位置づけです。外部から呼び出せないように、非公開のアクセス性を与えています。

最後の *leap_year* は、ある年が閏年であるかどうかを判定する関数です。こちらは、公開のアクセス性が与えられており、外部からも呼び出せるようになっています。

なお、これら三つの関数は、いずれも **static** 付きで宣言されており、**静的メンバ関数**（*static member function*）と呼ばれるメンバ関数となります。

静的メンバ関数の定義は、クラス定義の中と外のどちらでも行えます（この点は、静的データメンバとは違います）。クラス *Date* では、*days_of_year* と *days_of_month* はクラス定義の外で定義され、*leap_year* はクラス定義の中で定義されています。

なお、静的メンバ関数をクラス定義の外で定義する場合は、“クラス名 :: メンバ関数名”という形式で、**static** を付けずに定義する必要があります。

*

さて、三つの静的メンバ関数は、特定の日付クラスオブジェクトには所属しておらず、仮引数に受け取った値をもとに処理を行います。

ヘッダ部のクラス定義内で宣言されている順に、各メンバ関数を理解していきましょう。

▪ **static int** *days_of_year*(**int** *year*);

*year* 年の日数を返却する静的メンバ関数です。閏年であるかどうかを判定する静的メンバ関数 *leap_year* の返却値を利用した計算を行います。*year* が平年であれば 365 を返却し、閏年であれば 366 を返却します。

▪ **static int** *days_of_month*(**int** *year*, **int** *month*);

*year* 年 *month* 月の日数（その月が何日まであるのか）を求めるメンバ関数です。各月の日数を格納する配列 *dmax* と、メンバ関数 *leap_year* の返却値を利用した計算を行って、28，29，30，31 のいずれかの値を返却します。

▪ **static bool** *leap_year*(**int** *year*);

仮引数に受け取った *year* 年が閏年であるかどうかを調べ、閏年であれば **true** を返却し、そうでなければ **false** を返却する静的メンバ関数です。

▶ この関数の本体は、ヘッダ部のクラス定義の中で定義されています。

なお、クラスの外部から、公開属性の静的メンバ関数を呼び出す式の形式は、“クラス名 :: メンバ関数名 (...)” です。そのため、以下の式で、西暦 *y* 年が閏年であるかどうかの判定が行えることになります。

```
Date::leap_year(y)                    // y年は閏年か？
```

**特定のオブジェクトではなく、クラスに共通するデータは静的データメンバとして実現し、クラス全体に関わる手続きや、そのクラスに所属する個々のオブジェクトの状態とは無関係な手続きは、静的メンバ関数として実現します。**

▶ 静的メンバ関数は、“オブジェクト名 . メンバ関数名 (...)” の形式でも呼び出せます。たとえば、*d1* や *d2* が *Date* 型オブジェクトであれば、*d1*.*leap*(*year*) や *d2*.*leap*(*year*) と呼び出しても、*Date*::*leap_year*(*y*) と同じ結果が得られます。

静的データメンバも同様です。公開属性をもつ静的データメンバは、“クラス名 :: データメンバ名” と “オブジェクト名 . データメンバ名” の両方でアクセスできます。

ただし、静的メンバ関数と静的データメンバのアクセスに “オブジェクト名 .” 形式を使うと紛らわしくなりますので、“クラス名 ::” 形式を利用するのが原則です。

なお、静的メンバ関数は、特定のオブジェクトに所属しないという性質をもっていますので、当然のことながら **this** ポインタをもちません。

## Column 1-1 const 静的データメンバの初期化子

本文でも学習したとおり、静的データメンバの実体は、“クラス名 :: データメンバ名” という形式で、クラス定義の外で **static** を付けずに定義するのが原則です。

この規則には、例外があります。**静的データメンバが、const 付き汎整数型、あるいは const 付き列挙型の場合に限り、クラス定義の中のデータメンバの宣言で汎整数定数式の初期化子を与えてもよいことになっています。**

```
class A {
    const static int a = 0;          // OK!
    static int b = 0;                // コンパイルエラー：constでない
    const static double pi = 3.14;   // コンパイルエラー：整数でない
};
```

List 1-6 [B]　Date/Date.cpp

```
//--- デフォルトコンストラクタ（今日の日付に設定）---//
Date::Date()
{
    time_t current = time(NULL);                    // 現在の暦時刻を取得
    struct tm* local = localtime(&current);         // 要素別の時刻に変換

    y = local->tm_year + 1900;          // 年：tm_yearは西暦年-1900
    m = local->tm_mon + 1;              // 月：tm_monは0～11
    d = local->tm_mday;                 // 日
}

//--- コンストラクタ（指定された年月日に設定）---//
Date::Date(int yy, int mm, int dd)
{
    set(yy, mm, dd);                    // 日付をyy年mm月dd日に設定
}

//--- 日付をyy年mm月dd日に設定 ---//
void Date::set(int yy, int mm, int dd)
{
    y = yy;     // 年
    m = mm;     // 月
    d = dd;     // 日
}
```

## デフォルトコンストラクタ

クラス Date には、二つの形式のコンストラクタが多重定義されています。ここでは、便宜上、宣言順にコンストラクタⒶ、コンストラクタⒷと呼ぶことにします。

▪ Date::Date();

コンストラクタⒶは、引数を受け取らないコンストラクタです。現在（プログラム実行時）の日付となるように初期化を行います。なお、本コンストラクタのように、**実引数を与えずに呼び出せるコンストラクタは、デフォルトコンストラクタ**（*default constructor*）**と呼ばれます。**

## 変換コンストラクタ

▪ Date::Date(int yy, int mm = 1, int dd = 1);

コンストラクタⒷは、日付を yy 年 mm 月 dd 日に設定するコンストラクタです。データメンバへの値の代入は、この後で解説するメンバ関数 set に委ねています。

ヘッダ部では、仮引数 mm と dd に**デフォルト実引数** 1 が設定されています。そのため、コンストラクタⒶとⒷを利用すると、以下の四つの形式で Date 型オブジェクトの初期化を行えることになります。

```
Date p;                 // コンストラクタA：今日の日付
Date q(2017);           // コンストラクタB：2017年1月1日
Date r(2018, 2);        // コンストラクタB：2018年2月1日
Date s(2019, 3, 5);     // コンストラクタB：2019年3月5日
```

さて、2 番目の呼出しは、実引数が一つです。このように、**単一の実引数によって呼び**

**出せるコンストラクタは、変換コンストラクタ**（*conversion constructor*）**と呼ばれます。**引数の型をクラスの型に変換する働きをする、とみなせるからです。

**単一の実引数で呼び出せる変換コンストラクタは、( ) 形式に加えて、= 形式でも呼び出せることになっています。**したがって、*q* は以下のようにも宣言できます。

```
Date q = 2017;      // Date q(2017);と同じ
```

▶ 何だか“日付を整数で初期化している”ように見えるため、ちょっと違和感を覚えるかもしれません。このような初期化を抑制する方法は、次節で学習します。

一般に、単一の *Type* 型の実引数で呼び出せる、クラス *C* のコンストラクタは、*Type* 型から *C* 型への型変換を行う変換コンストラクタとなります。

上記の例の場合、**int** 型の“2017”から、*Date* 型の“2017 年 1 月 1 日”への変換を行っているわけです。

㊟：以下に解説する *leap_year* から *set_day* まで 7 個のメンバ関数は、ソース部ではなくヘッダ部で定義されています。

▪ **bool** *leap_year*() **const**;

本関数は、所属するオブジェクトの日付が閏年であるかどうかを調べ、閏年であれば **true** を返却し、そうでなければ **false** を返却する、非静的メンバ関数です。

**同一の名前をもつ静的メンバ関数と非静的メンバ関数を多重定義している**わけです。

▪ **int** *year*() **const**;

西暦年を取得するためのゲッタです。データメンバ *y* の値をそのまま返却します。

▪ **int** *month*() **const**;

月を取得するためのゲッタです。データメンバ *m* の値をそのまま返却します。

▪ **int** *day*() **const**;

日を取得するためのゲッタです。データメンバ *d* の値をそのまま返却します。

▪ **void** *set_year*(**int** *yy*);

年を設定するためのセッタです。受け取った *yy* の値をデータメンバ *y* に代入します。

▪ **void** *set_month*(**int** *mm*);

月を設定するためのセッタです。受け取った *mm* の値をデータメンバ *m* に代入します。

▪ **void** *set_day*(**int** *dd*);

日を設定するためのセッタです。受け取った *dd* の値をデータメンバ *d* に代入します。

▪ **void** *set*(**int** *yy*, **int** *mm*, **int** *dd*);

年・月・日の三値を一括して設定するためのセッタです。受け取った *yy*，*mm*，*dd* の値をデータメンバ *y*，*m*，*d* に代入します。

List 1-6 [C]　Date/Date.cpp

```
//--- 前日の日付を返却 ---//
Date Date::preceding_day() const
{
    Date temp(*this);                   // *thisのコピーを作成
    return --temp;                      // コピーの前日を求めて返却
}

//--- 翌日の日付を返却 ---//
Date Date::following_day() const
{
    Date temp(*this);                   // *thisのコピーを作成
    return ++temp;                      // コピーの翌日を求めて返却
}

//--- 年内の経過日数を返却 ---//
int Date::day_of_year() const
{
    int days = d;   // 年内の経過日数

    for (int i = 1; i < m; i++)         // 1月～(m-1)月の日数を加える
        days += days_of_month(y, i);
    return days;
}

//--- 曜日を返却（日曜～土曜が0～6に対応）---//
int Date::day_of_week() const
{
    int yy = y;
    int mm = m;
    if (mm == 1 || mm == 2) {
        yy--;
        mm += 12;
    }
    return (yy + yy / 4 - yy / 100 + yy / 400 + (13 * mm + 8) / 5 + d) % 7;
}

//--- 1970年1月1日からの経過日数を返却（long型への変換関数）---//
Date::operator long() const
{
    return *this - Date(1970, 1, 1);
}
```

## コピーコンストラクタ

▪ *Date preceding_day*() **const**;

このメンバ関数は、前日の日付を求めて返却する関数です。この後に定義されている前置減分演算子 **--** を利用して処理を行います。

メンバ関数本体の *temp* の宣言に着目しましょう。

キーワード **this** は、メンバ関数が所属するオブジェクトを指すポインタでした。したがって、***this は、メンバ関数が所属するオブジェクトそのものを表す式です**（ポインタに対して間接演算子 * を適用した式は、そのポインタが指すオブジェクトそのものを表すからです）。

この関数では、“*Date* 型の値を受け取って *Date* 型の値を初期化するコンストラクタ”を呼び出して、*temp* を ***this** と同じ日付で初期化しています。

**受け取った同一型の引数をもとに初期化を行うコンストラクタは、コピーコンストラクタ**（*copy constructor*）**と呼ばれます。コピーコンストラクタを明示的に定義していないクラスに対しては、全データメンバの値を単純にコピーするコピーコンストラクタが、コンパイラによって自動的に提供されることになっています。**そのため、クラス *Date* では、コピーコンストラクタの定義は行っていません。

本関数は、コピーコンストラクタで自分自身の日付のコピーを *temp* として作っておき、その前日の日付を前置減分演算子 **--** で求めて返却する、という仕組みになっています。

▶ コピーコンストラクタを明示的に定義する方法は、次節で学習します。

▪ *Date following_day*() **const**;

翌日の日付を求めて返却します。前置減分演算子 **--** ではなく前置増分演算子 **++** を利用していることを除くと、日付を求める手順は *preceding_day* と同様です。

*

引き続き、二つのメンバ関数が定義されています。

▪ **int** *day_of_year*() **const**;

年内の経過日数（1 月 1 日から順に、1，2，… と数えた値）を返却します。

▪ **int** *day_of_week*() **const**;

曜日を返却します。ツェラーの公式と呼ばれる式を利用して、日曜、月曜日、…、土曜を 0，1，…，6 の整数値として求めて返却します。

## 変換関数

▪ *Date*::**operator long**() **const**;

1970 年 1 月 1 日からの経過日数を返却するメンバ関数です（日付が 1970 年 1 月 1 日より古い場合の返却値は、負値となります）。経過日数の計算には、この後で定義されている 2 項 **-** 演算子を利用しています。

本関数のように、“**operator** 型名”と名付けられたメンバ関数は、**変換関数**（*conversion function*）と呼ばれます。**変換関数は、所属するオブジェクトの値を、任意の型に変換して返却するメンバ関数です。**

一般に、Type 型への変換関数の名前は、“**operator** Type”です。**関数名そのものが返却値型を表す**ため、返却値型は指定できません。また、引数を受け取ることもできません。

▶ 変換関数は、**const** メンバ関数として実現するのが一般的です（本クラスもそうなっています）。

**long** 型への変換関数 **operator long** を提供しているため、*Date* から **long** への型変換は、明示的キャスト・暗黙裏のキャストのいずれでも行えるようになります。

*

変換コンストラクタと変換関数の総称が**ユーザ定義変換**（*user-defined conversion*）です。両者がそろうと、二つの型間の型変換が相互に可能になる、というわけです。

List 1-6 [D]　Date/Date.cpp

```
//--- １日進める（前置増分：++(*this)）---//
Date& Date::operator++()
{
    if (d < days_of_month(y, m))     // 月末より前であれば
        d++;                         //     日をインクリメントするだけ
    else {                           // 翌月に繰り上がる
        if (++m > 12) {              //     12月を超えるのであれば
            y++;                     //         翌年の…
            m = 1;                   //         １月に繰り上がる
        }
        d = 1;                       //     次の月の１日となる
    }
    return *this;
}

//--- １日進める（後置増分：(*this)++）---//
Date Date::operator++(int)
{
    Date temp(*this);                // インクリメント前の値をコピー
    ++(*this);                       // 前置増分演算子++によってインクリメント
    return temp;                     // コピーを返却
}

//--- １日戻す（前置減分：--(*this)）---//
Date& Date::operator--()
{
    if (d > 1)                       // 月始めでなければ
        d--;                         //     日をデクリメントするだけ
    else {                           // 前月に繰り下がる
        if (--m <= 1) {              //     １月を超えるのであれば
            y--;                     //         前年の…
            m = 12;                  //         12月に繰り下がる
        }
        d = days_of_month(y, m);     //     前月の月末となる
    }
    return *this;
}

//--- １日戻す（後置減分：(*this)--）---//
Date Date::operator--(int)
{
    Date temp(*this);                // デクリメント前の値をコピー
    --(*this);                       // 前置減分演算子--によってデクリメント
    return temp;                     // コピーを返却
}
```

## 演算子の多重定義

ここで定義されているメンバ関数は、いずれも**演算子関数**（*operator function*）と呼ばれる関数です。一般に、“☆演算子”は、“**operator** ☆”という名前のメンバ関数、あるいは非メンバ関数として定義します。この名前の関数が定義されると、クラス型オブジェクトに対して、☆演算子が適用できるようになります。

なお、増分演算子 **++** と減分演算子 **--** の演算子関数は、前置形式と後置形式を区別して多重定義することになっています。

各関数の返却値型は任意ですが、クラス名が *C* であれば、右のようにするのが一般的です。

- 前置演算子 … *C*& 型
- 後置演算子 … *C* 型

▶ 本クラスも、これに準じています。

なお、前置形式は引数を受け取らない形式としなければならず、後置形式は **int** 型引数を受け取る形式としなければなりません。

さて、クラス *Date* で定義されているのは、日付を 1 日進めたり戻したりするための前置あるいは後置の増分／減分演算子です。

### ▪ 前置演算子

日付を 1 日進めたり戻したりするための計算を行っています。日付をインクリメント／デクリメントした後の《自分自身》への参照を ***this** として返却しています。

### ▪ 後置演算子

日付を進めたり戻したりするための計算を行うのは前置版と同じですが、変更前の日付を返却する必要があるため、処理の手順は前置版よりも複雑です。

① 自分自身である ***this** のコピーを作業用の変数 *temp* に保存しておく。
② 前置増分／減分演算子を利用して日付をインクリメント／デクリメントする。
③ 保存しておいたインクリメント／デクリメント前の値 *temp* を返却する。

このように、後置形式の増分／減分演算子では、いったん ***this** をコピーしておき、インクリメント／デクリメントを行った後に、コピーを返却する必要があります。そのため、**前置形式よりも後置形式のほうが、高コストとなるのが一般的です。**

**このことから、前置形式と後置形式のいずれを呼び出してもよい文脈では、前置形式を呼び出したほうがよい、という教訓が得られます。**

### ▪ 演算子関数の呼出し

定義された演算子は、クラスオブジェクトに対して適用できます。以下に示すのが、その一例です（*d1* と *d2* は *Date* 型であるとします）。

```
++d1;          // 日付d1を1日進める（このような文脈では前置を使う）
d2 = ++d1;     // 日付d1を1日進めた後にd1をd2に代入
d3 = d1++;     // 日付d1をd3に代入してからd1を1日進める
```

なお、クラスオブジェクトに対して演算子を適用する式を見つけたコンパイラは、以下のように、メンバ関数である演算子関数を呼び出すコードを生成します。

```
++x  ➡  x.operator++()      // 前置増分演算子（引数無し）
x++  ➡  x.operator++(0)     // 後置増分演算子（ダミーの引数 0 が渡される）
```

規定により、後置演算子の適用時は、ダミーの値として 0 が渡されます。

▶ 後置増分演算子の関数頭部は、“*Date* **operator++(int)**” となっています。宣言されている **int** 型の仮引数には、名前すら与えられていません（後置減分演算子も同様です）。

List 1-6 [E] Date/Date.cpp

```
//--- 日付をdn日進める（複合代入：*this += dn）---//
Date& Date::operator+=(int dn)
{
    if (dn < 0)                             // dnが負であれば
        return *this -= -dn;                // 演算子-=に処理を委ねる

    d += dn;                                // 日にdnを加える

    while (d > days_of_month(y, m)) {       // 日が月の日数内に収まるように年月を調整
        d -= days_of_month(y, m);
        if (++m > 12) {
            y++;
            m = 1;
        }
    }
    return *this;
}

//--- 日付をdn日戻す（複合代入：*this -= int）---//
Date& Date::operator-=(int dn)
{
    if (dn < 0)                             // dnが負であれば
        return *this += -dn;                // 演算子+=に処理を委ねる

    d -= dn;                                // 日からdnを減じる

    while (d < 1) {                         // 日が正になるように年月を調整
        if (--m < 1) {
            y--;
            m = 12;
        }
        d += days_of_month(y, m);
    }
    return *this;
}

//--- dn日後を求める（加算：*this + dn）---//
Date Date::operator+(int dn) const
{
    Date temp(*this);
    return temp += dn;                      // 演算子+=を利用
}

//--- dayのdn日後を求める（加算：dn + day）---//
Date operator+(int dn, const Date& day)
{
    return day + dn;                        // Date + intの演算子+に処理を委ねる
}

//--- dateのdn日前を求める（減算：*this - dn）---//
Date Date::operator-(int dn) const
{
    Date temp(*this);
    return temp -= dn;                      // 演算子-=を利用
}
```

- `Date& operator+=(int dn);`

  日付を *dn* 日進めるための複合代入演算子です（*Date* 型に **int** を加えます）。

- `Date& operator-=(int dn);`

  日付を *dn* 日戻すための複合代入演算子です（*Date* 型から **int** を減じます）。

- `Date operator+(int dn) const;`

  日付の *dn* 日後を求める加算演算子です（*Date* 型に **int** を加えます）。

以上の三つの関数は、２項演算子です。**メンバ関数として定義された２項演算子は、左オペランドに対してその演算子関数が呼び出され、右オペランドが引数として渡されることになっています。**

## フレンド関数

▫ `friend Date operator+(int dn, const Date& date);`

日付 *date* の *dn* 日後を求める加算演算子です（**int** 型に *Date* 型を加えます）。

▶ この関数の本体は、以下のように１行だけで実現されています。

```
return day + dn;
```

すなわち、*Date* 型に **int** を加えるメンバ関数版の **operator+** に実質的な処理を委ねています。

ヘッダ部の関数宣言の冒頭には、キーワード **friend** が与えられています（ソース部の関数定義には付いていません）。このように宣言された関数は、メンバ関数ではなく、**随伴関数＝フレンド関数**（*friend function*）となります。

クラス定義の中で **friend** を付けて『この関数は、私の友達ですよ！』と宣言すると、その“お友達関数”には、**そのクラスの非公開メンバを自由にアクセスする権限が与えられる**のです。なお、フレンド関数になれるのは、クラスの外部で定義される通常の関数＝非メンバ関数か、別のクラスに所属するメンバ関数です。

一般に、クラス *C* のフレンド関数 *mem* は、そのクラス *C* 型のオブジェクトに対して起動されるわけではないという点で、通常の関数と同じです。そのため、ドット演算子 **.** を適用して、クラス *C* 型のオブジェクト *x* に対して *x.mem*(...) として呼び出すことはできませんし、**this** ポインタをもちません。メンバでないにもかかわらず、“クラス内部にこっそりアクセスできる”特別な許可が与えられた関数です。

▶ もし **+** 演算子をフレンドでない関数として実装するのであれば、以下の点に注意する必要があります。

- **非公開データメンバにアクセスできない。**
  もし非公開データメンバにアクセスする必要があれば、データメンバの値を返すメンバ関数（ゲッタ）を呼び出さなければなりません。
- **自動的にインライン関数とならない。**
  関数の定義をヘッダ部に置くのであれば、インライン関数にして内部結合を与えなければなりません。キーワード **inline** を指定してインライン関数となるように明示的に宣言する必要があります。

**非メンバ関数として定義された２項演算子では、左オペランドが関数の第１引数として渡されて、右オペランドが第２引数として渡されることになっています。**

▪ `Date operator-(int dn) const;`

日付の *dn* 日前を求める減算演算子です（*Date* 型から **int** を減じます）。

▶ この演算子は、メンバ関数として実現された２項演算子です。

List 1-6 [F]　Date/Date.cpp

```
//--- 日付の差を求める（減算：*this - day）---//
long Date::operator-(const Date& day) const
{
    long count;
    long count1 = this->day_of_year();  // *thisの年内経過日数
    long count2 = day.day_of_year();    // day  の年内経過日数

    if (y == day.y)                                     // *thisとdayは同じ年
        count = count1 - count2;
    else if (y > day.y) {                               // *thisのほうが新しい年
        count = days_of_year(day.y) - count2 + count1;
        for (int yy = day.y + 1; yy < y; yy++)
            count += days_of_year(yy);
    } else {                                            // *thisのほうが古い年
        count = -(days_of_year(y) - count1 + count2);
        for (int yy = y + 1; yy < day.y; yy++)
            count -= days_of_year(yy);
    }

    return count;
}

//--- dayと同じ日付か？（等価：*this == day）---//
bool Date::operator==(const Date& day) const
{
    return y == day.y && m == day.m && d == day.d;
}

//--- dayと違う日付か？（等価：*this != day）---//
bool Date::operator!=(const Date& day) const
{
    return !(*this == day);          // 演算子==を利用
}

//--- dayより後の新しい日付か？（関係：*this > day）---//
bool Date::operator>(const Date& day) const
{
    if (y > day.y) return true;      // 年が異なる（新しい）
    if (y < day.y) return false;     //     〃     （古い）

    if (m > day.m) return true;      // 年が等しい ― 月が異なる（新しい）
    if (m < day.m) return false;     //     〃                （古い）

    return d > day.d;                //     〃     ― 月も等しい（日を比較）
}

//--- day以降の日付か？（関係：*this >= day）---//
bool Date::operator>=(const Date& day) const
{
    return !(*this < day);           // 演算子<を利用
}

//--- dayより前の古い日付か？（関係：*this < day）---//
bool Date::operator<(const Date& day) const
{
    if (y < day.y) return true;      // 年が異なる（古い）
    if (y > day.y) return false;     //     〃     （新しい）

    if (m < day.m) return true;      // 年が等しい ― 月が異なる（古い）
    if (m > day.m) return false;     //     〃                （新しい）

    return d < day.d;                //     〃     ― 月も等しい（日を比較）
}

//--- day以前の日付か？（関係：*this <= day）---//
bool Date::operator<=(const Date& day) const
{
    return !(*this > day);           // 演算子>を利用
}
```

▪ **long operator-(const** *Date& day*) **const;**

日付 *day* との差の日数を、**long** 型の値として求める減算演算子です（*Date* 型である ***this** の日付から、引数として受け取った *Date* 型の日付 *day* を減じます）。

本メンバ関数が所属する日付が、引数 *day* よりも新しければ正の値を、*day* と等しければ 0 を、*day* よりも古ければ負の値を返却します。

▪ **bool operator==(const** *Date& day*) **const;**

*day* と同じ日付かどうかを判定する等価演算子です。

▪ **bool operator!=(const** *Date& day*) **const;**

*day* と違う日付かどうかを判定する等価演算子です。

**等価演算子 == と != とがクラスに対して自動的に提供されることはありません。オブジェクトの値を比較する必要があるクラスでは、自分で定義する必要があります。**

▪ **bool operator>(const** *Date& day*) **const;**

*day* より後の日付であるかどうかを判定する関係演算子です。

▪ **bool operator>=(const** *Date& day*) **const;**

*day* 以降の日付であるかどうかを判定する関係演算子です。

▶ この関数の本体は、以下のように 1 行だけで実現されています。

```
return !(*this < day);
```

すなわち、実質的な処理を演算子 < に委ねています（演算子関数 **operator<** を呼び出します）。

なお、

```
return *this > day || *this == day;
```

と実現すると、効率が落ちることに注意しましょう。というのも、左オペランドが **false** となる場合に、二つの演算子関数 **operator>** と **operator==** の両方が呼び出されるからです。

▪ **bool operator<(const** *Date& day*) **const;**

*day* より前の日付であるかどうかを判定する関係演算子です。

▪ **bool operator<=(const** *Date& day*) **const;**

*day* 以前の日付であるかどうかを判定する関係演算子です。

代入演算子が定義されていなければ、クラスオブジェクトの値が同じ型のクラスオブジェクトに代入される際は、全データメンバの値がコピーされることになっています。

代入操作によって、全データメンバをコピーすればよい（コピーしても構わない）クラスには、代入演算子を定義する必要はありません。

そのため、本クラスでは、代入演算子は定義されていません。

▶ 代入演算子を明示的に定義する方法は、次節で学習します。

**List 1-6 [G]** Date/Date.cpp

```
//--- 文字列表現を返却 ---//
string Date::to_string() const
{
    ostringstream s;
    s << y << "年" << m << "月" << d << "日";
    return s.str();
}

//--- 出力ストリームsに日付xを挿入 ---//
ostream& operator<<(ostream& s, const Date& x)
{
    return s << x.to_string();
}

//--- 入力ストリームsから日付を抽出してxに格納 ---//
istream& operator>>(istream& s, Date& x)
{
    int yy, mm, dd;
    char ch;

    s >> yy >> ch >> mm >> ch >> dd;
    x = Date(yy, mm, dd);
    return s;
}
```

▪ std::**string** to_string() **const**;

文字列表現を返却します。"y年m月d日" 形式の文字列を生成して返却します。

▶ 本メンバ関数では、文字列生成のために **ostringstream** クラスを利用しています。このクラスは、『入門編』の第 11 章で簡単に学習しました。

▫ std::**ostream**& **operator<<**(std::**ostream**& s, **const** Date& x);

日付 x を出力ストリーム s に挿入する挿入子です。クラス Date のメンバ関数ではなく、非メンバ関数として定義されています。

▫ std::**istream**& **operator>>**(std::**istream**& s, Date& x);

入力ストリーム s から読み込んだ日付を x に格納する抽出子です。挿入子と同様に、非メンバ関数として定義されています。

▶ 挿入子と抽出子の定義方法は、『入門編』の第 11 章で簡単に学習しました。

日付クラス Date を利用するプログラム例を **List 1-7**（p.28）に示します。クラス Date のいろいろな機能が試せるようになっています。実行してみましょう。

▶ 右ページに示すのは、プログラムの実行例です。

### 演習 1–2

本文で学習したクラス Date は、たとえば、2015 年 2 月 30 日といった不正な日付での初期化や代入などを許してしまう。コンストラクタなどに与えられた日付が不正であれば、正しい日付に補正するように書きかえたクラスを作成せよ。

## 実行例

```
日付dayを入力せよ。
年：2015
月：10
日：25
[1]情報表示 [2]日付の変更 [3]増減分演算子 [4]前後の日付 [5]比較 [0]終了：1
・日付2015年10月25日に関する情報
  曜日は日曜日   年内経過日数は298日   1970年1月1日の16733日後   その年は閏年ではない。
[1]情報表示 [2]日付の変更 [3]増減分演算子 [4]前後の日付 [5]比較 [0]終了：2
[1]年変更 [2]月変更 [3]日変更 [4]年月日変更 [5]n日進める [6]n日戻す [0]戻る：1
年：2017
2017年10月25日に更新されました。
[1]年変更 [2]月変更 [3]日変更 [4]年月日変更 [5]n日進める [6]n日戻す [0]戻る：2
月：11
2017年11月25日に更新されました。
[1]年変更 [2]月変更 [3]日変更 [4]年月日変更 [5]n日進める [6]n日戻す [0]戻る：3
日：18
2017年11月18日に更新されました。
[1]年変更 [2]月変更 [3]日変更 [4]年月日変更 [5]n日進める [6]n日戻す [0]戻る：4
年：2018
月：9
日：27
2018年9月27日に更新されました。
[1]年変更 [2]月変更 [3]日変更 [4]年月日変更 [5]n日進める [6]n日戻す [0]戻る：5
日数：30
2018年10月27日に更新されました。
[1]年変更 [2]月変更 [3]日変更 [4]年月日変更 [5]n日進める [6]n日戻す [0]戻る：6
日数：40
2018年9月17日に更新されました。
[1]年変更 [2]月変更 [3]日変更 [4]年月日変更 [5]n日進める [6]n日戻す [0]戻る：0
[1]情報表示 [2]日付の変更 [3]増減分演算子 [4]前後の日付 [5]比較 [0]終了：3
[1]++day  [2]day++  [3]--day  [4]day--  [0]戻る：1
++day = 2018年9月18日
day   = 2018年9月18日
[1]++day  [2]day++  [3]--day  [4]day--  [0]戻る：2
day++ = 2018年9月18日
day   = 2018年9月19日
[1]++day  [2]day++  [3]--day  [4]day--  [0]戻る：3
--day = 2018年9月18日
day   = 2018年9月18日
[1]++day  [2]day++  [3]--day  [4]day--  [0]戻る：4
day-- = 2018年9月18日
day   = 2018年9月17日
[1]++day  [2]day++  [3]--day  [4]day--  [0]戻る：0
[1]情報表示 [2]日付の変更 [3]増減分演算子 [4]前後の日付 [5]比較 [0]終了：4
[1]翌日 [2]前日 [3]n日後(day+n) [4]n日後(n+day)[5]n日前 [0]戻る：1
それは2018年9月18日です。
[1]翌日 [2]前日 [3]n日後(day+n) [4]n日後(n+day)[5]n日前 [0]戻る：2
それは2018年9月16日です。
[1]翌日 [2]前日 [3]n日後(day+n) [4]n日後(n+day)[5]n日前 [0]戻る：3
日数：15
それは2018年10月2日です。
[1]翌日 [2]前日 [3]n日後(day+n) [4]n日後(n+day)[5]n日前 [0]戻る：4
日数：15
それは2018年10月2日です。
[1]翌日 [2]前日 [3]n日後(day+n) [4]n日後(n+day)[5]n日前 [0]戻る：5
日数：20
それは2018年8月28日です。
[1]翌日 [2]前日 [3]n日後(day+n) [4]n日後(n+day)[5]n日前 [0]戻る：0
[1]情報表示 [2]日付の変更 [3]増減分演算子 [4]前後の日付 [5]比較 [0]終了：5
比較対象の日付day2を入力せよ。
年：2020
月：11
日：18
day  = 2018年9月17日
day2 = 2020年11月18日
day  -  day2  = -793
day2 -  day   = 793
day  == day2  = false
day  != day2  = true
day  >  day2  = false
day  >= day2  = false
day  <  day2  = true
day  <= day2  = true
[1]情報表示 [2]日付の変更 [3]増減分演算子 [4]前後の日付 [5]比較 [0]終了：0
```

List 1-7 Date/DateTest.cpp

```
// 日付クラスDateの利用例

#include <iostream>
#include "Date.h"

using namespace std;

//--- 日付に関する情報を表示 ---//
void display(const Date& day)
{
    string dw[] = {"日", "月", "火", "水", "木", "金", "土"};
    cout << "・日付" << day << "に関する情報\n";
    cout << "  曜日は" << dw[day.day_of_week()] << "曜日";
    cout << "  年内経過日数は" << day.day_of_year() << "日";
    cout << "  1970年1月1日の" << long(day) << "日後";
    cout << "  その年は閏年で" << (day.leap_year() ? "ある" : "はない。") << '\n';
}

//--- 日付を変更 ---//
void change(Date& day)
{
    while (true) {
        cout << "[1]年変更 [2]月変更 [3]日変更 [4]年月日変更 "
             << "[5]n日進める [6]n日戻す [0]戻る：";
        int selected;
        cin >> selected;

        if (selected == 0) return;

        int y, m, d, n;
        if (selected == 1 || selected == 4) { cout << "年："; cin >> y; }
        if (selected == 2 || selected == 4) { cout << "月："; cin >> m; }
        if (selected == 3 || selected == 4) { cout << "日："; cin >> d; }
        if (selected == 5 || selected == 6) { cout << "日数："; cin >> n; }

        switch (selected) {
         case 1: day.set_year(y);   break;      // y年に設定
         case 2: day.set_month(m);  break;      // m月に設定
         case 3: day.set_day(d);    break;      // d日に設定
         case 4: day.set(y, m, d);  break;      // y年m月d日に設定
         case 5: day += n;          break;      // n日進める
         case 6: day -= n;          break;      // n日戻す
        }
        cout << day << "に更新されました。\n";
    }
}

//--- 増分および減分演算子を適用 ---//
void inc_dec(Date& day)
{
    while (true) {
        cout << "[1]++day  [2]day++  [3]--day  [4]day-- [0]戻る：";
        int selected;
        cin >> selected;

        if (selected == 0) return;

        switch (selected) {
         case 1: cout << "++day = " << ++day << '\n';  break;   // 前置増分
         case 2: cout << "day++ = " << day++ << '\n';  break;   // 後置増分
         case 3: cout << "--day = " << --day << '\n';  break;   // 前置減分
         case 4: cout << "day-- = " << day-- << '\n';  break;   // 後置減分
        }
        cout << "day   = " << day << '\n';
    }
}

//--- 前後の日付を求める ---//
void before_after(Date& day)
{
    while (true) {
        cout << "[1]翌日 [2]前日 [3]n日後(day+n) [4]n日後(n+day)"
             << "[5]n日前 [0]戻る：";
```

```
        int selected;
        cin >> selected;
        if (selected == 0) return;
        int n;
        if (selected >= 3 && selected <= 5) {
            cout << "日数：";  cin >> n;
        }
        cout << "それは";
        switch (selected) {
         case 1: cout << day.following_day();   break;  // 翌日
         case 2: cout << day.preceding_day();   break;  // 前日
         case 3: cout << day + n;               break;  // n日後 (Date + int )
         case 4: cout << n   + day;             break;  // n日後 (int  + Date)
         case 5: cout << day - n;               break;  // n日前 (Date - int )
        }
        cout << "です。\n";
    }
}
//--- 他の日付との比較 ---//
void compare(const Date& day)
{
    int y, m, d;
    cout << "比較対象の日付day2を入力せよ。\n";
    cout << "年：";  cin >> y;
    cout << "月：";  cin >> m;
    cout << "日：";  cin >> d;
    Date day2(y, m, d); // 比較対象の日付
    cout << boolalpha;
    cout << "day  = " << day  << '\n';
    cout << "day2 = " << day2 << '\n';
    cout << "day  -  day2  = " << (day  -  day2) << '\n';
    cout << "day2 -  day   = " << (day2 -  day ) << '\n';
    cout << "day  == day2  = " << (day  == day2) << '\n';
    cout << "day  != day2  = " << (day  != day2) << '\n';
    cout << "day  >  day2  = " << (day  >  day2) << '\n';
    cout << "day  >= day2  = " << (day  >= day2) << '\n';
    cout << "day  <  day2  = " << (day  <  day2) << '\n';
    cout << "day  <= day2  = " << (day  <= day2) << '\n';
}
int main()
{
    int y, m, d;
    cout << "日付dayを入力せよ。\n";
    cout << "年：";  cin >> y;
    cout << "月：";  cin >> m;
    cout << "日：";  cin >> d;
    Date day(y, m, d);  // 読み込んだ日付
    while (true) {
        cout << "[1]情報表示 [2]日付の変更 [3]増減分演算子 [4]前後の日付 "
             << "[5]比較 [0]終了：";
        int selected;
        cin >> selected;
        if (selected == 0) break;
        switch (selected) {
         case 1: display(day);      break;  // 日付に関する情報を表示
         case 2: change(day);       break;  // 日付を変更
         case 3: inc_dec(day);      break;  // 増分演算子・減分演算子
         case 4: before_after(day); break;  // 前後の日付を求める
         case 5: compare(day);      break;  // 他の日付との比較
        }
    }
}
```

## Column 1-2 演算子の多重定義

本節では、クラス *Date* に対して、いくつかの演算子関数を定義しました。ここでは、演算子関数に関する規則をまとめます。

### ▪ 定義可能な演算子

すべての演算子を多重定義できるわけではありません。また、単項版と２項版の両方を多重定義できる演算子があります。また、非メンバ関数として定義できる演算子とそうでない演算子とがあります。その規則をまとめたのが、**Fig.1C-1** です。すべてを覚えなくてもよいので、必要に応じて参照しましょう。


**Fig.1C-1** 演算子と多重定義

演算子関数は、自由に定義できます。加算を行うはずの + 演算子に対して、減算の意味をもたせることも可能です。しかし、そのように定義された演算子が使いやすいはずがありません。演算子の本来の仕様と可能な限り同一あるいは類似した仕様となるように定義するのが原則です。

なお、プログラマが新しい演算子を作り出すことはできません。たとえば、FORTRAN を模して、べき乗を求めるための ** 演算子を定義するといったことは不可能です。

また、優先度や結合規則を変更することもできません。たとえば、加算を行う２項 + 演算子の優先度を、乗算を行う２項 * 演算子よりも高くするようなことは不可能です。

### ▪ 単純代入演算子

単純代入演算子 = を非メンバ関数として定義できないことは、必ず覚えておく必要があります。

▪ 複合代入演算子

ある演算子☆を定義することによって、それに対応する複合代入演算子☆= が自動的に定義されることはありません。

また、**int** 型や **double** 型などの組込み型に対して成立する、“本質的には “a = a + b” と “a += b” は同等である。”といった規則が、ユーザ定義型に対して自動的に成立することもありません。

▪ 論理演算子

組込み型に対する論理演算子 **&&** と || を用いた論理演算では、**短絡評価**が行われることを『入門編』の第２章で学習しました。**&&** 演算子の左オペランドが **false** であれば右オペランドの評価が省略され、|| 演算子の左オペランドが **true** であれば右オペランドの評価が省略されます。

クラス型に対して **&&** 演算子と || 演算子を定義することはできますが、それらの演算で**短絡評価が行われるように定義することはできません。**

そのため、**&&** 演算子の左オペランドの評価結果が **false** であっても右オペランドの評価は必ず行われます。また、|| 演算子の左オペランドの評価結果が **true** であっても右オペランドの評価は必ず行われます。

組込み型に対する論理演算子の挙動と、クラス型に対する論理演算子の挙動とを一致させることはできないわけです。したがって、**クラス型に論理演算子を定義することは推奨されません。**

▪ 演算子の定義と呼出し

演算子の定義の形式は、単項であるか２項であるかによって、大きく異なります。

※ ３項演算子 **? :** は定義できないため、定義できるのは単項演算子と２項演算子のみです。

▫ 単項演算子

単項演算子は、引数を受け取らないメンバ関数または引数１個の非メンバ関数として定義します。以下に示すのが、その形式の一例です。

- 引数０個のメンバ関数　　返却値型 *C*::**operator** ☆ **()** { /* 中略 */ }
- 引数１個の非メンバ関数　　返却値型 **operator** ☆ **(const** *C* **&)** { /* 中略 */ }

一般に、“☆演算子” を適用した式 “☆a” は、以下のように解釈されます。

- 引数０個のメンバ関数　　a**.operator** ☆ **()**
- 引数１個の非メンバ関数　　**operator** ☆ **(**a**)**

ただし、本文で学習したとおり、後置形式の増分演算子 **++** と減分演算子 **--** だけは、例外的にダミーの **int** 型引数を受け取ります。

▫ ２項演算子

２項演算子は、引数１個のメンバ関数または引数２個の非メンバ関数として定義します。以下に示すのが、その形式の一例です。

- 引数１個のメンバ関数　　返却値型 *C*::**operator** ☆ **(***Type* **&)** { /* 中略 */ }
- 引数２個の非メンバ関数　　返却値型 **operator** ☆ **(const** *C* **&,** *Type* **&)** { /* 中略 */ }

一般に、“☆演算子” を適用した式 “a ☆ b” は、以下のように解釈されます。

- 引数１個のメンバ関数　　a**.operator** ☆ **(**b**)**
- 引数２個の非メンバ関数　　**operator** ☆ **(**a**,** b**)**

# 1–3 整数配列クラス



本章では、動的に確保する記憶域を扱う整数配列クラスを通じて、コンストラクタやデストラクタなどについて学習します。

## 整数配列クラス

C++ が言語レベルで提供する配列は、宣言時に要素数を指定する必要があります。ここでは、生成時に自由に要素数を指定できる配列クラスを作っていくことにします。

**List 1-8** と **List 1-9**（p.35）に示すのが、**int** 型の配列を扱うクラス *IntArray* のヘッダ部とソース部です。クラス *IntArray* には、2 個のデータメンバがあります。要素数を表す *nelem* と、配列本体の先頭要素を指すポインタ *vec* です。なお、要素数 *nelem* の値は、そのゲッタであるメンバ関数 *size* で調べられます。

## 明示的コンストラクタ

本クラスでは、二つのコンストラクタが多重定義されています。まずは、最初のコンストラクタに注目しましょう。

コンストラクタの定義の先頭に、キーワード **explicit** が付けられています。これは、宣言するコンストラクタを、**明示的コンストラクタ**（*explicit constructor*）にするための関数指定子です。明示的コンストラクタとは、**暗黙の型変換を抑止するコンストラクタです**。以下の例で考えていきましょう。

```
1 IntArray a = 5;       // コンパイルエラー
2 IntArray x(5);        // ＯＫ
```

コンストラクタに **explicit** の指定がなければ、1 と 2 の両方の形式の宣言が許されます。しかし、コンストラクタが明示的コンストラクタとなっていると、宣言 2 だけが許されて、宣言 1 はコンパイルエラーとなります。“配列を整数で初期化している”と誤解されかねないような、紛らわしい形式の宣言を、**explicit** が抑止するわけです。

**単一引数のコンストラクタが = 形式で起動されるのを抑止するには、( ) 形式での起動のみを許す明示的コンストラクタとして定義するのが原則です。**

さて、コンストラクタの本体では、配列本体の動的生成、すなわち配列用の記憶域の確保を行います。生成する配列の要素数は、仮引数 *size* に受け取った値です。

先ほどの 2 によってオブジェクトが宣言・定義された場合のコンストラクタの動作を考えましょう。

まず、メンバ初期化子 *nelem*(*size*) の働きによって、データメンバ *nelem* が 5 で初期化されます。

List 1-8 IntArray1/IntArray.h

```
// 整数配列クラスIntArray（第１版：ヘッダ部）

#ifndef ___Class_IntArray
#define ___Class_IntArray

//===== 整数配列クラス ======//
class IntArray {
    int nelem;      // 配列の要素数
    int* vec;       // 先頭要素へのポインタ

public:
    //--- 明示的コンストラクタ ---//
    explicit IntArray(int size) : nelem(size) { vec = new int[nelem]; }

    //--- コピーコンストラクタ ---//
    IntArray(const IntArray& x);

    //--- デストラクタ ---//
    ~IntArray() { delete[] vec; }

    //--- 要素数を返す ---//
    int size() const { return nelem; }

    //--- 代入演算子= ---//
    IntArray& operator=(const IntArray& x);

    //--- 添字演算子[] ---//
    int& operator[](int i) { return vec[i]; }

    //--- const版添字演算子[] ---//
    const int& operator[](int i) const { return vec[i]; }
};

#endif
```

その後に実行されるコンストラクタ本体では、**new** 演算子で確保した *nelem* 個分の記憶域の先頭要素へのポインタが *vec* に代入されます（**Fig.1-3**）。

ポインタが配列の先頭要素を指すとき、そのポインタはあたかも配列であるかのように振る舞うのでしたね。そのため、クラス *IntArray* の内部では、生成した配列の各要素を先頭から順に *vec*[0]，*vec*[1]，…，*vec*[4] でアクセスできることになります。

扱える配列の要素数が可変であるとともに、オブジェクト（この場合は *x*）自体の大きさが要素数に依存することなく一定であることが、*IntArray* クラスの特徴です。

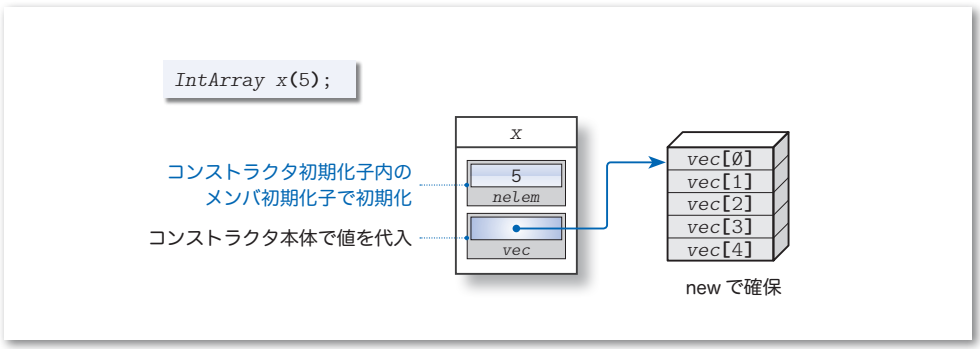


**Fig.1-3** コンストラクタによる配列の生成

## コピーコンストラクタの多重定義

次に学習するのは、２番目に宣言されたコンストラクタです（定義は右ページのソース部にあります）。仮引数 *x* の型が *IntArray* への参照ですから、p.18 で学習したコピーコンストラクタです。

**このコピーコンストラクタが定義されていなかったらどうなるか**を、右のように宣言された配列 *x*, *y* を例に考えましょう。

```
IntArray x(12);
IntArray y = x;    // yをxで初期化
```

コンパイラによって暗黙のうちに提供される、“全データメンバの値をメンバ単位でコピーする”コピーコンストラクタの働きによって、*x* のデータメンバ *nelem* と *vec* の値が *y* のメンバ *nelem* と *vec* にコピーされます。その結果、二つのポインタ *x.vec* と *y.vec* が同一領域を指すことになってしまうという不都合が生じます。

コンパイラによってコピーコンストラクタが暗黙のうちに提供されるのを抑止する必要があるクラスでは、コピーコンストラクタの明示的な多重定義が必要です。

クラス名が *C* であれば、右に示す形式とするのが原則です。

```
// コピーコンストラクタの一般的な形式
C::C(const C&);
```

さて、本クラスのコピーコンストラクタの **if** 文では、引数として受け取ったオブジェクトへのポインタ **&***x* と、自分自身へのポインタ **this** の等価性を判定しています。もし両者が等しければ“未初期化の自分自身で初期化する”ということです。データメンバの値が不定値になるのを避けるために、要素数 *nelem* を 0 にして、ポインタ *vec* に空ポインタを代入しています。

両者が等しくない場合は、引数に受け取った配列 *x* と同じ要素数で、全要素の値が同一となるように配列を作って初期化します。

## デストラクタ

本クラスでは、**デストラクタ**（*destructor*）と呼ばれる特別なメンバ関数がヘッダ部で定義されています。デストラクタの名前は、クラス名の前にチルダ ~ が付いたものです。デストラクタは、そのクラスの**オブジェクトの生存期間が終了しそうになったときに自動的に呼び出されるメンバ関数**です。

返却値をもたない点はコンストラクタと同じですが、自動的に呼び出されるという性質上、**引数を受け取らない**点と**多重定義できない**点がコンストラクタとは異なります。

クラス *IntArray* 型のオブジェクトの生存期間が尽きたときは、オブジェクトそのものの領域は解放されますが、コンストラクタで確保した配列領域が自動的に解放されることはありません。

本クラス *IntArray* のデストラクタが行うのは、ポインタ *vec* が指す配列領域の解放です。コンストラクタで行った仕事の後始末をデストラクタが行うわけです。

List 1-9 IntArray1/IntArray.cpp

```
// 整数配列クラスIntArray（第１版：ソース部）

#include <cstdlib>
#include "IntArray.h"

//--- コピーコンストラクタ ---//
IntArray::IntArray(const IntArray& x)
{
    if (&x == this) {                       // 初期化子が自分自身であれば…
        nelem = 0;
        vec = NULL;
    } else {
        nelem = x.nelem;                    // 要素数をxと同じにする
        vec = new int[nelem];               // 配列本体を確保
        for (int i = 0; i < nelem; i++)     // 全要素をコピー
            vec[i] = x.vec[i];
    }
}

//--- 代入演算子 ---//
IntArray& IntArray::operator=(const IntArray& x)
{
    if (&x != this) {                   // 代入元が自分自身でなければ…
        if (nelem != x.nelem) {         // 代入前後の要素数が異なれば…
            delete[] vec;               // もともと確保していた領域を解放
            nelem = x.nelem;            // 新しい要素数
            vec = new int[nelem];       // 新たに領域を確保
        }
        for (int i = 0; i < nelem; i++) // 全要素をコピー
            vec[i] = x.vec[i];
    }
    return *this;
}
```

## 代入演算子の多重定義

代入演算子 = は、仮引数 x に受け取った配列を代入する（要素数と全要素の値が同一となるように、配列を更新する）演算子として定義されています。

**この代入演算子が定義されておらず、右に示すコードが実行されたら、どうなるでしょう。**

```
IntArray a(2);  // aは要素数2
IntArray b(5);  // bは要素数5
a = b;
```

同じクラス型をもったオブジェクトどうしの代入が行われると、**全データメンバがコピーされます**。したがって、コピーコンストラクタを多重定義しない場合と同じ問題が発生します（**Column 1-3**：p.37）。

一般に、同一クラスのオブジェクトの値を代入する際に、全メンバのコピーを行うべきでないクラス C には、代入演算子を以下の形式で多重定義するのが原則です。

```
// 代入演算子の一般的な形式
C& C::operator=(const C&)
{
    // ...
    return *this;
}
```

もちろん、この演算子関数が返却するのは、メンバ関数が所属するオブジェクトへの参照です。

## 添字演算子の多重定義

配列の各要素を手軽にアクセスできるように定義しているのが、添字演算子 [ ] です。その演算子関数 operator[] の返却値型は、int ではなく int& となっています。というのも、

```
a = x[2];        // 代入演算子の右オペランド：intでもint&でも可
```

と値を取り出せるだけでなく、以下に示すように、代入式の左辺に置いて値を代入できるようにするためです。

```
x[3] = 10;       // 代入演算子の左オペランド：intでは不可でint&では可
```

なお、const オブジェクトに対して添字演算子が適用される場合を考慮して、const 版の添字演算子も多重定義されています。

*

**List 1-10** は、クラス *IntArray* を利用するプログラム例です。

**List 1-10** IntArray1/IntArrayTest.cpp

```
// 整数配列クラスIntArray（第１版）の利用例

#include <iostream>
#include "IntArray.h"

using namespace std;

int main()
{
    IntArray v1(5);              // v1は要素数5の配列
    for (int i = 0; i < v1.size(); i++)
        v1[i] = i + 1;
    const IntArray v2 = v1;      // v2はv1のコピー
//  v2[0] = 10;                  // コンパイルエラー
    // v1の全要素を表示
    for (int i = 0; i < v1.size(); i++)
        cout << "v1[" << i << "] = " << v1[i] << '\n';
    // v2の全要素を表示
    for (int i = 0; i < v2.size(); i++)
        cout << "v2[" << i << "] = " << v2[i] << '\n';
}
```

**実行結果**

```
v1[0] = 1
v1[1] = 2
v1[2] = 3
v1[3] = 4
v1[4] = 5
v2[0] = 1
v2[1] = 2
v2[2] = 3
v2[3] = 4
v2[4] = 5
```

本プログラムでは、二つの配列が定義されています。

最初の *v1* は、明示的コンストラクタで要素数 5 の配列として初期化されます。もう一つの *v2* は、コピーコンストラクタで、配列 *v1* のコピーとして初期化されます。

配列 *v1* の要素のアクセスでは、非 const 版の演算子 operator[] が利用されます。また、const 宣言されている *v2* の要素のアクセスには、const 版の演算子 operator[] が利用されます。

## Column 1-3 代入演算子の多重定義

クラス *IntArray* で定義されている代入演算子 **operator=** は非常に複雑な構造です。以下に示すように、シンプルに定義してもよさそうですね。

```
void IntArray::operator=(const IntArray& x)
{
    delete[] vec;                       // もともと確保していた領域を解放
    nelem = x.nelem;                    // 新しい要素数
    vec = new int[nelem];               // 新たに領域を確保
    for (int i = 0; i < nelem; i++)     // 全要素をコピー
        vec[i] = x.vec[i];
}
```

実は、この代入演算子には、以下に示す三つの問題点が含まれています。

### ▪ 記憶域の不要な解放・確保を行うこと

代入元の配列 *x* と代入先の配列 **this** の要素数が一致していれば、代入先の配列領域をそのまま流用できます。上記のコードは、二つの配列の要素数とは無関係に、代入先の配列をいったん解放して再確保しています。

**List 1-9**（p.35）では、記憶域の解放と再確保を行うのを、代入元と代入先の配列要素数が異なる場合に限定しています。

### ▪ 返却値型が void であること

上記のコードでは、メンバ関数 **operator=** の返却値型が **void** であるため、組込み型オペランドに対する代入演算子と同じような使い方ができません。たとえば、以下に示す代入では、**A**はエラーとならないものの、**B**がコンパイルエラーとなります。

```
A  x = y;          // x.operator=(y);                  ＯＫ
B  x = y = z;      // x.operator=(y.operator=(z));     エラー
```

それぞれの代入は、コメントに書かれているように解釈されます。**B**の網かけ部の型が **void** であるため、**const** *IntArray&* を受け取る関数の引数として渡すことは不可能です。

代入式を評価すると、代入後の左オペランドの型と値が得られます。また、関数の返却値を参照としておけば、代入式の左辺にも右辺にも置ける左辺値式となることは、『入門編』の第 6 章で学習しました。

代入演算子の返却値型は、そのクラスへの参照型にすべきであるという原則にしたがって定義すべきです。

### ▪ 自己代入に対応していないこと

自分自身の値を代入することを“自己代入”といいます。*IntArray* 型オブジェクトの自己代入を行ったらどうなるでしょうか。

```
x = x;      // 左辺xの配列を解放した後で右辺xの配列の要素をコピー（？）
```

上記のコードは、うまくいきません。というのも、関数冒頭の **delete[]** で配列を解放してしまい、全要素が消滅してしまうからです。末尾の **for** 文では、消滅済み配列からのコピーを試みることになってしまいます。もちろん、そのようなことは不可能であり、実行時エラーとなります。

＊

上記三つの問題点に対処するために、**List 1-9** の代入演算子の定義は複雑になっているのです。

# まとめ

- クラス *C* のクラス定義は、**class** *C* { /* … */ }; という形式で行う。末尾のセミコロンは省略できない。{ } の中は、データメンバ・メンバ関数などのメンバの宣言である。データメンバは、オブジェクトの状態（ステート）を表し、メンバ関数はオブジェクトの振舞いを表し、それらがカプセル化される。

- クラス型は、**int** 型や **double** 型などの組込み型と対比して、ユーザ定義型と呼ばれる。

- クラスのメンバは、クラスの外部に対して公開することもできるし、非公開にすることもできる。公開を指示するのが **public**: であり、非公開を指示するのが **private**: である。非公開のメンバをクラスの外部からアクセスすることはできない。データ隠蔽を実現するために、データメンバは原則として非公開にすべきである。

- データメンバはオブジェクトの一部である。すなわち、データメンバはオブジェクトに所属する。同様に、メンバ関数も（論理的には）オブジェクトに所属する。メンバ関数の中では、公開メンバにも非公開メンバにも自由にアクセスできる。

- オブジェクト *x* のメンバ *m* は、ドット演算子 **.** を用いた *x*.*m* によってアクセスできる。また、ポインタ *p* が指すオブジェクトのメンバ *m* は、(***p*).*m* によってアクセスできるが、アロー演算子 **->** を用いた *p*->*m* のほうが簡潔な表記である。

- クラス *C* の内部で定義された列挙などの識別子 *m* は、そのクラスの有効範囲の中に入る。識別子 *m* が公開属性をもっている場合、クラスの外部からは、有効範囲解決演算子 **::** を用いた *C*::*m* によってアクセスできる。

- オブジェクトの生成時に呼び出される特別なメンバ関数が、コンストラクタである。コンストラクタの目的は、オブジェクトを適切に初期化することである。コンストラクタは、名前がクラス名と同一であって、返却値をもたない。

- コンストラクタを定義しないクラスには、引数を受け取らず、本体が空であるコンストラクタが、コンパイラによって自動的に定義される。

- データメンバは、クラス定義内において宣言された順に初期化される。

- 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。

- メンバ関数を呼び出すことは、オブジェクトに対してメッセージを送ることである。メッセージを受け取ったオブジェクトは能動的に処理を行う。

- データメンバの値を取得して返却するメンバ関数をゲッタと呼び、データメンバに値を設定するメンバ関数をセッタと呼ぶ。両者の総称がアクセッサである。

- メンバ関数は、自分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、メンバ関数が所属するオブジェクトそのものは、式 ***this** によって表せる。

- データメンバ *m* と同一名の仮引数あるいは局所的な変数がメンバ関数で宣言された場合、宣言されたほうの名前が見えて、データメンバの名前が隠される。データメンバは **this->***m* としてアクセスでき、宣言された引数・変数は *m* としてアクセスできる。

- クラス定義は、独立したヘッダとして実現するとよい。本書では、これをヘッダ部と呼ぶ。なお、ヘッダ部には、**using** 指令を置くべきでない。

- クラス定義の中で定義されたメンバ関数は、内部結合をもつインライン関数となる。

- クラス定義の外で定義されたメンバ関数は、明示的な指定のない限り、外部結合をもつ非インライン関数となる。このような関数の定義は、ヘッダ部とは別に、独立したソースファイルとして実現するとよい。本書では、これをソース部と呼ぶ。

- ヘッダ内で定義する非メンバ関数には、内部結合を与えなければならない。

- 同一型のオブジェクトの値による初期化と代入は、まったく異なる。前者はコピーコンストラクタによって行われ、後者は代入演算子によって行われる。

- クラス型のオブジェクトが、同一クラス型のオブジェクトの値で初期化される際は、全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る。

- 同一型のクラスオブジェクトの値によるオブジェクトの構築・初期化において、全データメンバを単純にコピーすべきでないクラスに対しては、コピーコンストラクタを定義しなければならない。

- 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際は、全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る。

- 同一型のクラスオブジェクト間の代入において、全データメンバを単純にコピーすべきでないクラスに対しては、代入演算子を定義しなければならない。

- 代入演算子を定義する場合、自己代入に対する対処を行う必要がある。自己代入であるかどうかは、所属するオブジェクトへのポインタ **this** と、代入元オブジェクトへのポインタとの等価性で判定できる。

- フレンド関数には、そのクラスの非公開メンバにアクセスする特権が与えられる。

- 実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、= 形式によっても起動できる。また、この形式のコンストラクタは、変換コンストラクタと呼ばれ、引数型からそのクラス型への変換を行う働きをもつ。
- 単一の実引数でのオブジェクトの構築・初期化を ( ) 形式のみに限定するには、コンストラクタの宣言に **explicit** を付けて、明示的コンストラクタとするとよい。明示的コンストラクタとは、= 形式によるオブジェクトの構築・初期化を抑止するコンストラクタである。
- 定値オブジェクトに対して、通常の（非 **const** の）メンバ関数を起動することはできない。そのため、定値オブジェクトに対して起動される可能性があるメンバ関数は、**const** メンバ関数として実現しておかなければならない。
- 変換関数を定義すると、クラス型のオブジェクトの値を任意の型へと変換できるようになる。Type 型に変換するための変換関数の名前は **operator** Type である。変換関数は、必要に応じて暗黙裏に呼び出される。また、キャストによって明示的に呼び出すこともできる。
- 変換関数による型変換と、変換コンストラクタによる型変換の総称が、ユーザ定義変換である。
- 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。
- 演算子関数を定義すると、その演算子をクラス型オブジェクトに適用できるようになる。演算子 **@** の関数名は **operator @** である。
- 演算子関数は、その演算子の本来の仕様と、可能な限り同一あるいは類似した仕様となるように定義するのが、原則である。
- 増分演算子 **++** と減分演算子 **--** は、前置と後置を区別して定義する。後置形式ではダミーの **int** 型引数を受け取る。一般に後置形式のほうが高コストであるため、両者のどちらを呼び出しても構わない文脈では、前置形式を利用するとよい。
- メンバ関数として定義された 2 項演算子関数の左オペランドの型は、そのメンバ関数が所属するクラス型でなければならない。左オペランドに対して暗黙の型変換を適用すべき 2 項演算子は、非メンバ関数として実現すべきである。
- ある演算子 **@** 用の演算子関数を定義しても、それに対応する複合代入演算子 **@=** 用の演算子関数がコンパイラによって自動的に定義されることはない。
- 論理演算子 **&&** と **||** は多重定義すべきでない。短絡評価が行われないからである。
- クラスに対して挿入子 **<<** や抽出子 **>>** を多重定義すると、入出力の実現が容易になる。

- 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個のみが作られる。
- クラス定義の中で **static** を付けて宣言されたデータメンバが、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、そのメンバの実体を定義しない。実体の定義は、クラス定義の外で、**static** を付けずに行う。
- 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”によって行うべきである。
- 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。
- 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。
- 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 **(**...**)**”によって行うべきである。
- 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。
- デストラクタは、オブジェクトの生存期間が尽きて破棄される直前に、自動的に呼び出される特別なメンバ関数である。
- デストラクタは、引数を受け取らず返却値をもたない。多重定義することもできない。
- デストラクタを定義しないクラスには、本体が空であって引数を受け取らない、デフォルトデストラクタが、コンパイラによって自動的に定義される。
- コンストラクタやメンバ関数の中で **new** 演算子によって確保した動的記憶域期間をもつ領域が、オブジェクト破棄の際に自動的に解放されることはないので、デストラクタで解放するとよい。