第９章

クラスの基本

オブジェクト指向プログラミングにおいて最も基礎的で重要な概念が、クラスです。本章では、銀行口座クラスと自動車クラスを例に、クラスに関する基本を学習します。

9-1 クラスの考え方

本節では、オブジェクト指向の基礎であるクラスの考え方について学習します。

■ データの扱い

List 9-1 に示すプログラムは、足立君と仲田君の銀行口座のデータを扱うプログラムです。変数に値を設定して、表示するだけの単純なものです。

List 9-1 Chap09/list0901.cpp

```
// 銀行口座
#include <string>
#include <iostream>
using namespace std;

int main()
{
    string adachi_account_name = "足立幸一";     // 足立君の口座名義
    string adachi_account_no = "12345678";       //   〃   の口座番号
    long  adachi_account_balance = 1000;         //   〃   の預金額

    string nakata_account_name = "仲田真二";     // 仲田君の口座名義
    string nakata_account_no = "87654321";       //   〃   の口座番号
    long  nakata_account_balance = 200;          //   〃   の預金額

    adachi_account_balance -= 200;               // 足立君が200円おろす
    nakata_account_balance += 100;               // 仲田君が100円預ける

    cout << "■足立君の口座：\"" << adachi_account_name << "\" ("
         << adachi_account_no << ") " << adachi_account_balance << "円\n";

    cout << "■仲田君の口座：\"" << nakata_account_name << "\" ("
         << nakata_account_no << ") " << nakata_account_balance << "円\n";

    return 0;
}
```

実行結果

```
■足立君の口座："足立幸一" (12345678) 800円
■仲田君の口座："仲田真二" (87654321) 300円
```

2 人分の口座に関するデータを表すために 6 個の変数を利用しています。

たとえば、変数 *adachi_account_name* は**口座名義**で、*adachi_account_no* は**口座番号**で、*adachi_account_balance* は**預金額**です。

> 名前が *adachi_* で始まる変数は**足立君の銀行口座に関するものである**。

という“関係”は、変数名やコメントから推測できます。

しかし、*nakata_account_no* を足立君の口座名義とすることや、*adachi_account_name* を仲田君の口座番号とすることも可能です。すなわち、変数間の“関係”を変数名から**推測**することはできますが、100%確定することは**不可能**です。

口座名義・口座番号・預金額を表す変数がバラバラに宣言・定義されており、ある一つの銀行口座に関するものであるという関係はプログラム中のどこにも現れません。

また、このようなスタイルの命名法を続けていくと、プログラムは莫大な変数名で溢れかえることになります。

■ オブジェクトとクラス

私たちがプログラムを作るときは、**現実世界のオブジェクト（物）を、プログラムの世界のオブジェクト（変数）に投影します。**

このプログラムでは、**Fig.9-1** a に示すように、一つの“口座”に関する口座名義・口座番号・預金額のデータが個別の変数へと投影されているわけです。

▶ この図は一般化して表したものです。足立君の口座・仲田君の口座に対して、三つのデータが別々の変数として投影されます。

口座の 1 側面ではなく、複数の側面に着目しましょう。そして、口座名義・口座番号・預金額をバラバラに扱うのではなく、図 b に示すように、ひとまとめにしたオブジェクトとして投影します。

このような投影を行うのが**クラス**（*class*）の考え方の基本です。

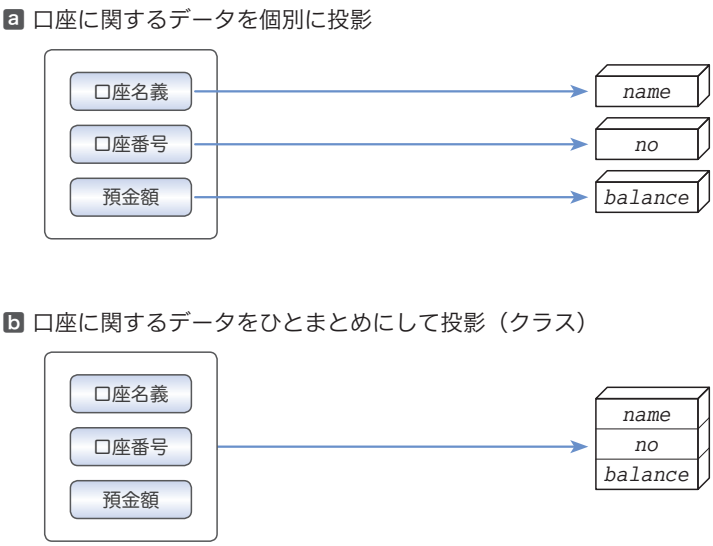


Fig.9-1　オブジェクトの投影とクラス

プログラムで扱う問題の種類や範囲によっても異なりますが、現実の世界からプログラムの世界への投影は、『まとめるべきものは、まとめる。』『本来まとまっているものは、そのままにする。』といった方針にのっとると、より自然で素直なものとなります。

*

クラスを用いて **List 9-1** を書き直すことにします。そのプログラムを **List 9-2**（次ページ）に示します。

List 9-2 Chap09/list0902.cpp

```
// 銀行口座クラス（第１版）

#include <string>
#include <iostream>
using namespace std;

class Account {                          // クラス定義
public:
    string name;                         // 口座名義
    string no;                           // 口座番号
    long balance;                        // 預金額
};

int main()
{
    Account adachi;                      // 足立君の口座
    Account nakata;                      // 仲田君の口座

    adachi.name = "足立幸一";            // 足立君の口座名義
    adachi.no = "12345678";              //   〃   の口座番号
    adachi.balance = 1000;               //   〃   の預金額

    nakata.name = "仲田真二";            // 仲田君の口座名義
    nakata.no = "87654321";              //   〃   の口座番号
    nakata.balance = 200;                //   〃   の預金額

    adachi.balance -= 200;               // 足立君が200円おろす
    nakata.balance += 100;               // 仲田君が100円預ける

    cout << "■足立君の口座：\"" << adachi.name << "\" ("
         << adachi.no << ") " << adachi.balance << "円\n";

    cout << "■仲田君の口座：\"" << nakata.name << "\" ("
         << nakata.no << ") " << nakata.balance << "円\n";

    return 0;
}
```

1

実行結果

```
■足立君の口座："足立幸一" (12345678) 800円
■仲田君の口座："仲田真二" (87654321) 300円
```

■ クラス定義

口座名義・口座番号・預金額をセットにしたクラスを宣言するのが網かけ部です。このような宣言を**クラス定義**（*class definition*）と呼びます。

先頭の **class** *Account* { がクラス定義の開始であり、その定義は }; まで続くことになります。関数とは違って、末尾にセミコロンが必要です。

{ } 間に置かれているのは**データメンバ**（*data member*）の宣言です。クラス *Account* は、以下の三つのメンバ＝要素から構成されることになります（**Fig.9-2**）。

- 口座名義を表す ***string*** 型の *name*
- 口座番号を表す ***string*** 型の *no*
- 預金額を表す **long** 型の *balance*

配列は同一型の要素を組み合わせて作られる型でした。任意の型の要素を組み合わせて作られる型が**クラス**であるといえますね。

データメンバに先立つ **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**ことの指示です。

Fig.9-2　クラス定義

▶ もし公開しなければ、クラスの外（たとえば **main** 関数）から、そのデータメンバの名前や型はおろか、その存在を知ることすらできなくなります（p.302）。

■ クラス型のオブジェクト

クラス定義は《**型**》の**宣言**であって、具体的な実体であるオブジェクト＝変数の定義ではありません。クラス *Account* 型のオブジェクトを宣言・定義するのが**1**です。

以下のように **int** 型オブジェクトの定義と並べると分かりやすくなります。*Account* が型名で、*adachi* や *nakata* がオブジェクトの名前です。

```
Account adachi;          // Account型のadachiの宣言・定義
Account nakata;          // Account型のnakataの宣言・定義
int     x;               // int     型のx     の宣言・定義
型名    変数名;
```

クラスは、タコ焼きを焼くための「カタ」のようなものです。上のように宣言・定義することによって、カタから作られた本物のタコ焼きができるわけです（**Fig.9-3**）。

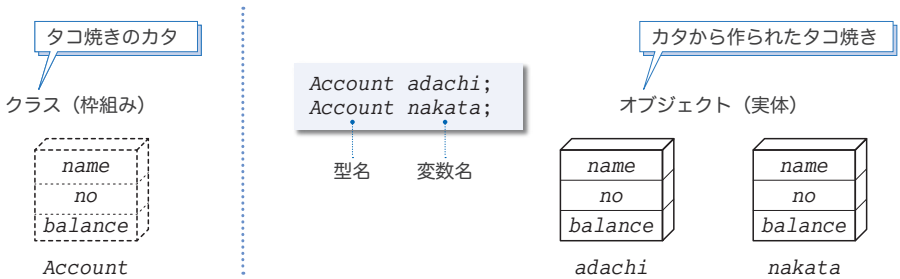


Fig.9-3　クラスとオブジェクト

int 型や **double** 型は、整数や実数などの数値を表現するのに都合のよい型です。このような型は、C++ というプログラミング言語に最初から用意されるものであり、**組込み型**（*built-in type*）と呼ばれます。

これに対して、銀行口座の口座番号や預金額をひとまとめにしたクラス *Account* は、プログラマが自分で作成する型です（もちろん、誰かが作ったものを利用することもあります）。このような型を、**ユーザ定義型**（*user-defined type*）と呼びます。

■ メンバのアクセス

プログラムでは足立君と仲田君の各メンバに値を代入して表示しています。

クラス型オブジェクト内のメンバをアクセスするために利用するのが、**ドット演算子**と呼ばれる **. 演算子**（*.operator*）です（**Table 9-1**）。

■Table 9-1　メンバアクセス演算子（ドット演算子）

x.y	*x* のメンバ *y* をアクセスする。

▶　ドットは点という意味です。“ドット演算子”は俗称です。

たとえば、足立君の口座の各データメンバをアクセスする式は次のようになります。

```
adachi.name          // 足立君の口座名義
adachi.no            //   〃   の口座番号
adachi.balance       //   〃   の預金額
```

仲田君の口座を表すオブジェクト *nakata* も同様です（**Fig.9-4**）。

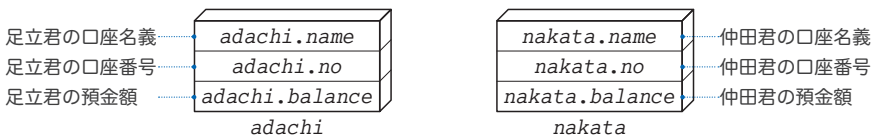


Fig.9-4　データメンバのアクセス

■ 問題点

クラスの導入によってプログラムはすっきりしたものの、まだ問題が残っています。

1　確実な初期化に対する無保証

プログラムでは、口座オブジェクトのメンバが**初期化されていません**。オブジェクトを作った後に値を**代入**しているだけです。

値を設定するかどうかがプログラマに委ねられている状態となっていますので、初期化を忘れた場合は、思いもよらぬ結果が生じる危険性があります。

“すべてのオブジェクトを初期化すべきかどうか”という一般論はさておき、初期化すべきオブジェクトは、初期化を強制したほうがよいでしょう。

2　データの保護に対する無保証

足立君の預金額である *adachi.balance* は、誰もが自由に扱うことができます。このことを現実の世界に置きかえると、足立君でなくても（通帳や印鑑がなくても）、自由に足立君の口座からお金をおろしたりすることができるということです。

口座番号を公開することはあっても、預金額を操作できるような状態で公開するといったことは、現実の世界ではあり得ません。

ここで掲げた問題点を解決するようにクラスを定義・利用することができます。そのように改良したプログラムが **List 9-3** です。

List 9-3 Chap09/list0903.cpp

```
// 銀行口座クラス（第２版）

#include <string>
#include <iostream>
using namespace std;

class Account {
private:
    string name;            // 口座名義
    string no;              // 口座番号
    long balance;           // 預金額

public:
    Account(string n, string num, long z) {     // コンストラクタ
        name = n;           // 口座名義
        no = num;           // 口座番号
        balance = z;        // 預金額
    }

    string GetName() {          // 口座名義を調べる
        return name;
    }

    string GetNo() {            // 口座番号を調べる
        return no;
    }

    long CheckBalance() {       // 預金額を調べる
        return balance;
    }

    void Deposit(long k) {      // 預ける
        balance += k;
    }

    void Withdraw(long k) {     // おろす
        balance -= k;
    }
};

int main()
{
    Account adachi("足立幸一", "12345678", 1000);       // 足立君の口座
    Account nakata("仲田真二", "87654321",  200);       // 仲田君の口座

    adachi.Withdraw(200);           // 足立君が200円おろす
    nakata.Deposit(100);            // 仲田君が100円預ける

    cout << "■足立君の口座：\"" << adachi.GetName() << "\" ("
         << adachi.GetNo() << ") " << adachi.CheckBalance() << "円\n";

    cout << "■仲田君の口座：\"" << nakata.GetName() << "\" ("
         << nakata.GetNo() << ") " << nakata.CheckBalance() << "円\n";

    return 0;
}
```

実行結果

```
■足立君の口座："足立幸一" (12345678) 800円
■仲田君の口座："仲田真二" (87654321) 300円
```

■ 非公開メンバと公開メンバ

第1版ではすべてのデータメンバを**公開**していました。今回の第2版ではすべて**非公開**にしています。非公開を指示するのが **private:** です。

非公開メンバは、クラスの外部に対して存在が隠されます。

クラス *Account* のイメージを表したのが **Fig.9-5** です。

```
class Account {
private:
    string name;     // 口座名義        非公開
    string no;       // 口座番号
    long balance;    // 預金額
public:
    Account(string n, string num, long z)
    { /* … */ }                          公 開
    string GetName()        { /* … */ }
    string GetNo()          { /* … */ }
    long CheckBalance()     { /* … */ }
    void Deposit(long k)    { /* … */ }
    void Withdraw(long k)   { /* … */ }
};
```

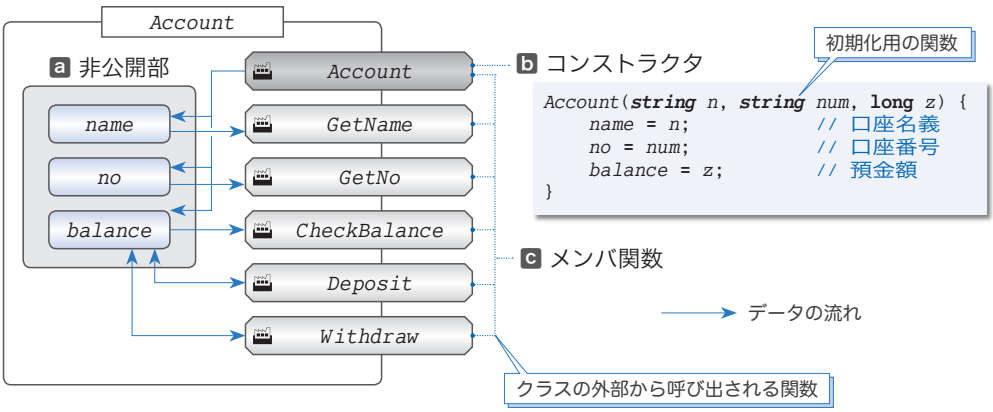


Fig.9-5　クラスAccount

a部の枠内に入っている *name*, *no*, *balance* は非公開です。クラスの外部である **main** 関数からは、非公開メンバにアクセスすることはできません。そのため、もし **main** 関数に以下のようなプログラムがあれば、すべてコンパイル時にエラーとなります。

```
adachi.name = "柴田望洋";    // 足立君の口座名義を書きかえる
adachi.no = "99999999";      // 足立君の口座番号を書きかえる
cout << adachi.balance;      // 足立君の預金額を表示
```

メンバを非公開とすれば、**データの保護性・隠蔽性だけでなく、プログラムの保守性も向上する**ことが期待できます。

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。

なお、**public:** や **private:** がなければ、メンバは非公開となります。したがって、本クラスの **private:** は削除できます。

また、**public:** や **private:** は、クラス定義中に何度でてきても構いません（**Fig.9-6**）。

```
class X {
    int a;          非公開
public:
    int b;          公開
    int c;
private:
    int d;          非公開
};
```

Fig.9-6　アクセス指定

■ コンストラクタ

クラス名と同じ名前の関数（**Fig.9-5 b**）が**コンストラクタ**（*constructor*）です。クラス型のオブジェクトの生成時に呼び出される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。

Account 型のオブジェクトを宣言している、**main** 関数の網かけ部に着目しましょう。

```
Account adachi("足立幸一", "12345678", 1000);   // 足立君の口座   ←1
Account nakata("仲田真二", "87654321",  200);   // 仲田君の口座   ←2
```

プログラムの流れがこれらの宣言文を通過してオブジェクト *adachi* と *nakata* が生成される際に、コンストラクタが呼び出されます。コンストラクタは、仮引数 *n*, *num*, *z* に受け取った値を各データメンバ *name*, *no*, *balance* にセットします。

オブジェクトに対して呼び出されるコンストラクタは、**自分自身のオブジェクトが何であるかを知っています**。したがって、1で呼び出されたコンストラクタの中で *name* といえば *adachi.name* のことですし、2で呼び出されたコンストラクタの中で *name* といえば *nakata.name* のことです。

▶ コンストラクタを呼び出すクラスオブジェクトの宣言は、**int** 型の整数 *x* を 5 で初期化する宣言（p.233）と同じ形であることに注意しましょう。

```
int x(5);          // int x = 5;と同じ
```

なお、1の宣言を以下のように書きかえるとエラーになります。

```
Account adachi;                  // エラー：引数がない
Account adachi("足立幸一");      // エラー：引数が不足
```

このことは、コンストラクタが**不完全あるいは不正な初期化を防止する**ことを示しています。必ず初期化を行わなければならないという手間が発生しますが、確実な初期化が行えるというメリットに比べると、ちっぽけな問題に過ぎません。

重要 クラス型を宣言するときは、必ずコンストラクタを用意し、オブジェクトを確実に初期化せよ。

なお、コンストラクタは**値を返却できません**。

▶ すなわち、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** とすることもできません）。

■ メンバ関数

コンストラクタを含め、クラスの内部に存在して非公開のメンバにもアクセスできる特権をもった関数を**メンバ関数**（*member function*）と呼びます（**Fig.9-5 c**）。

▶ コンストラクタは、オブジェクト生成時に呼び出される特殊なメンバ関数です。

クラス *Account* 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。

- *GetName*　：口座名義を ***string*** 型で返す。
- *GetNo*　：口座番号を ***string*** 型で返す。
- *CheckBalance*　：預金額を **long** 型で返す。
- *Deposit*　：お金を預ける（預金額を増やす）。
- *Withdraw*　：お金をおろす（預金額を減らす）。

メンバ関数はクラスにとって**よそ者ではない“内輪”**の存在です。そのため、メンバ関数の内部では、公開データメンバにも非公開データメンバにもアクセスできます。

▶ たとえばメンバ関数 *GetName* は、非公開のデータメンバである *name* の値を調べて返却します。

メンバ関数の呼出しには、**Table 9-1**（p.300）に示したメンバアクセス演算子 **.** を利用します。以下に例を示します。

```
adachi.CheckBalance()          // 足立君の預金額を調べる
adachi.Withdraw(200)           // 足立君の口座から200円おろす
nakata.Deposit(100)            // 仲田君の口座に100円預ける
```

クラスの外部から直接アクセスできない口座番号や預金額などのデータも、**メンバ関数を通じて間接的にアクセスできる**わけです。

そうすると、次のような疑問がわき上がってくるでしょう。

> データメンバの値を設定したり調べたりするだけのために、わざわざ関数を呼び出していると、実行効率が低下するのではないか？

もっともな疑問ですが**心配は無用です。クラス定義の中に埋め込まれたメンバ関数はインライン関数（p.206）となる**からです。そのため、

```
x = adachi.CheckBalance();  // 預金額を返却するメンバ関数を呼び出す
```

は、次に示すのと同等なコードに置きかえられた上でコンパイルされます。

```
x = adachi.balance;         // 実質的なコード
```

ただし、必ずしもインラインに展開されるとは限りません（p.207）。

▶ 以下のように、生成済みのオブジェクトに対してコンストラクタを呼び出すことは**できません**（この点で、コンストラクタは普通のメンバ関数とは異なります）。

```
adachi.adachi("足立幸一", "12345678", 5000);   // エラー
```

■ メソッドとメッセージ

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、*adachi.CheckBalance()* は、オブジェクト *adachi* に対して『預金額を教えてください。』というメッセージを送っているわけです（**Fig.9-7**）。

その結果、オブジェクト *adachi* が「預金額を返却してあげればいいのだな。」と能動的に意志決定を行って、『○○円ですよ。』と返答処理を行います。

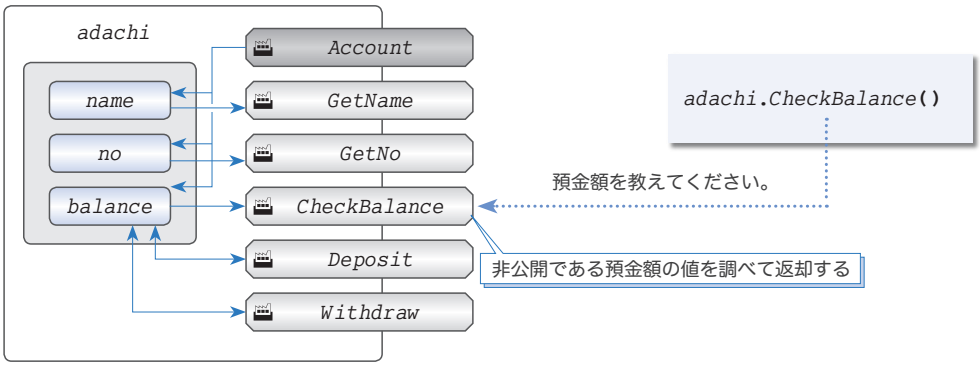


Fig.9-7　メンバ関数の呼出しとメッセージ

■ クラスとオブジェクト

メンバ関数は、データメンバの値をもとに処理を行い、必要に応じてデータメンバの値を更新します。メンバ関数とデータメンバは、緻密に結合されているわけです。

クラスは「プログラムの集積回路」の設計図であると考えられます。そして、その設計図に基づいて作られた実体としての集積回路がクラスのオブジェクトです。

C 言語のプログラムや、本書の第 8 章までのプログラムは、（実質的には）**関数の集合**ですが、クラスを多用する C++ のプログラムは、（理想的には）**クラスの集合**となります。

集積回路の設計図＝クラスを優れたものとすれば、C++ がもつ強大なパワーを発揮するプログラムとなります。

■ Column 9-1　インラインメンバ関数と前方参照

通常の非メンバ関数は、関数原型宣言が与えられない限り、前方参照（自分より後ろ側で定義された関数を呼び出すこと）ができません（p.170）。

しかし、メンバ関数では、その制限から解放されます。以下のクラスを考えましょう。

```
class X {
    int x;
public:
    int func1() { return func2(); } // 後方で定義されている関数の呼出し
    int func2() { return x; }
};
```

メンバ関数 *func1*，*func2* ともにインライン関数です。関数 *func1* から、それより後方で定義されている *func2* を呼び出していますが、エラーとなることなく正しくコンパイルできます。

コンパイラは、クラス定義を最後まで読んだ後で、インラインメンバ関数の解析を始めるのです。

9-2 クラスの実装

本節では、クラスの実装法について学習します。

■ クラス定義の外部でのメンバ関数の定義

ここまでの口座クラス *Account* では、すべてのメンバ関数の定義をクラス定義中に埋め込んでいます。クラスが大規模になると、そのような実装は現実的ではなくなってきます。単一のソースファイルで一つのクラスを管理するのは困難になるからです。

そのため、クラス定義の外部でメンバ関数を定義できるようになっています。そのように実装したプログラムが **List 9-4** です。

▶ メンバは原則非公開ですから、クラス定義の冒頭にあった **private:** は削除しました。

List 9-4 Chap09/list0904.cpp

```
// 銀行口座クラス（第３版：メンバ関数の定義を分離）

#include <string>
#include <iostream>
using namespace std;

class Account {
    string name;            // 口座名義
    string no;              // 口座番号
    long balance;           // 預金額

public:
    Account(string n, string num, long z);      // コンストラクタ
    string GetName();                           // 口座名義を調べる
    string GetNo();                             // 口座番号を調べる
    long CheckBalance();                        // 預金額を調べる
    void Deposit(long k);                       // 預ける
    void Withdraw(long k);                      // おろす
};

//--- コンストラクタ ---//
Account::Account(string n, string num, long z)
{
    name = n;           // 口座名義
    no = num;           // 口座番号
    balance = z;        // 預金額
}

//--- 口座名義を調べる ---//
string Account::GetName()
{ return name; }

//--- 口座番号を調べる ---//
string Account::GetNo()
{ return no; }

//--- 預金額を調べる ---//
long Account::CheckBalance()
{ return balance; }
```

1 2

実行結果

```
■足立君の口座："足立幸一"（12345678）800円
■仲田君の口座："仲田真二"（87654321）300円
```

```
//--- 預ける ---//
void Account::Deposit(long k)
{
    balance += k;
}

//--- おろす ---//
void Account::Withdraw(long k)
{
    balance -= k;
}

int main()
{
    Account adachi("足立幸一", "12345678", 1000);       // 足立君の口座
    Account nakata("仲田真二", "87654321",  200);       // 仲田君の口座

    adachi.Withdraw(200);           // 足立君が200円おろす
    nakata.Deposit(100);            // 仲田君が100円預ける

    cout << "■足立君の口座：\"" << adachi.GetName() << "\" ("
         << adachi.GetNo() << ") " << adachi.CheckBalance() << "円\n";

    cout << "■仲田君の口座：\"" << nakata.GetName() << "\" ("
         << nakata.GetNo() << ") " << nakata.CheckBalance() << "円\n";

    return 0;
}
```


クラス定義の外部でメンバ関数の**定義**を行う場合でも、関数の**宣言だけ**はクラス定義中に必要です。すなわち、以下のように宣言・定義します。

1. クラス定義の中ではメンバ関数の**関数原型宣言**を行う。
2. クラス定義の外ではメンバ関数の**定義**を行う。

コンストラクタを含め、どのメンバ関数も、名前が以下のようになっています。

クラス名 :: メンバ関数名

:: 演算子は**有効範囲解決演算子**でしたね。“クラス名 ::”が付いた名前が**クラス有効範囲**（*class scope*）中にあることを示しています。たとえば預金額を調べるためのメンバ関数 *CheckBalance* は、

クラス *Account* に属する *CheckBalance*

であることを示すために、*Account*:: という前置きが必要となります。

▶ 単項の :: 演算子は p.197 で、2項の :: 演算子は p.283 で学習しました。

重要 クラス *X* のメンバ関数 *func* は、クラス定義の外部では以下のように定義する。
返却値型 *X*::*func*(仮引数宣言節) { /* … */ }

▶ 本章以降では、“プログラムを一目で見渡せるように”という観点から、プログラムやコメントの表記をぎっしり詰めています。実際にプログラムを作る際は、もっとスペース・タブ・改行を入れて、ゆとりある表記をするようにしてください。

■ インタフェース部と実装部の分離

前ページのプログラムは、クラス *Account* の宣言・定義と、それを利用する部分（この場合は **main** 関数）を一つのソースファイルで実現しています。

しかし、よほど小規模なものでない限り、クラスの設計・開発から利用までのすべてを一人で行って、しかもそれが 1 個のソースファイルに収まることはありません。

クラスを利用しやすくするためには、独立したファイルとして提供すべきです。また、クラスの利用者にとって、クラス定義は必ず必要ですが、具体的な関数定義の中身までが必要になるとは限りません。

保守の点などから考えても、クラス定義とメンバ関数の定義は別々のファイルとして実装すべきです。したがって一般的な構成は **Fig.9-8** のようになります。

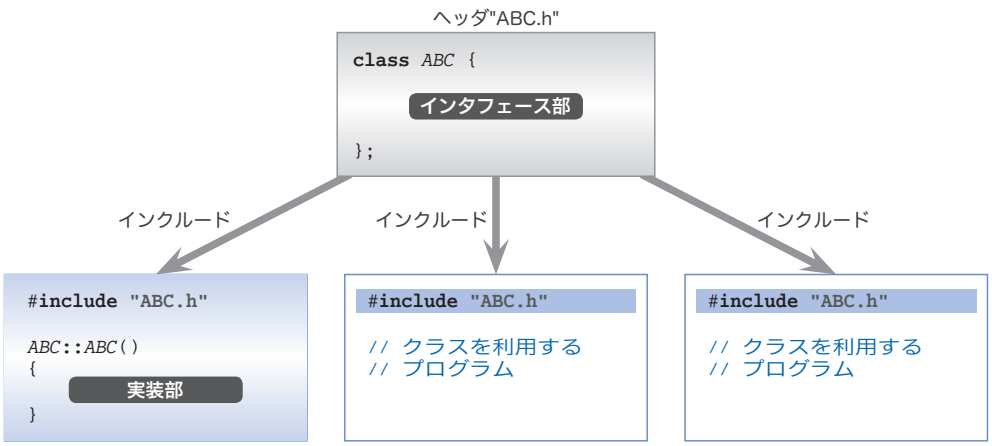


Fig.9-8　クラスの実装

クラスの開発者は、クラス定義である**インタフェース部**をヘッダとして作成し、関数定義などの**実装部**をソースプログラムとして作成します。

▶ 図に示すのは、実装部を一つのソースファイルとして実装した例です。大規模なクラスであれば、実装部を分割して複数のソースファイルで実現することになります。

このように実現した銀行口座クラスを以下に示します。

- **List 9-5**　クラス *Account* のインタフェース部（ヘッダ）
- **List 9-6**　クラス *Account* の実装部（メンバ関数の定義）

▶ 本書では、クラス定義を含むヘッダを**インタフェース部**と呼び、メンバ関数の定義を含むソースファイルを**実装部**と呼ぶことにします（ヘッダ中に関数定義が含まれることもありますので、これらの語句が 100%適切というわけではありません）。

List 9-5 Chap09/Account/Account.h

```
// 銀行口座クラス（第４版：インタフェース部）

#include <string>
using namespace std;

class Account {
    string name;            // 口座名義
    string no;              // 口座番号
    long balance;           // 預金額

public:
    Account(string n, string num, long z);      // コンストラクタ

    string GetName()    { return name; }        // 口座名義を調べる
    string GetNo()      { return no; }          // 口座番号を調べる
    long CheckBalance() { return balance; }     // 預金額を調べる

    void Deposit(long k);                       // 預ける
    void Withdraw(long k);                      // おろす
};
```

List 9-6 Chap09/Account/Account.cpp

```
// 銀行口座クラス（第４版：実装部）

#include <string>
#include <iostream>
#include "Account.h"
using namespace std;

//--- コンストラクタ ---//
Account::Account(string n, string num, long z)
{
    name = n;           // 口座名義
    no = num;           // 口座番号
    balance = z;        // 預金額
}

//--- 預ける ---//
void Account::Deposit(long k)
{
    balance += k;
}

//--- おろす ---//
void Account::Withdraw(long k)
{
    balance -= k;
}
```


クラス定義を含むヘッダは、クラスを利用するプログラムにとっての《**窓口**》となります。窓口を "Account.h" というヘッダで供給するのですから、そのクラスの実装部からも、そのクラスを利用するプログラムからも、

```
#include "Account.h"
```

とインクルードすることによって、クラス Account の宣言を取り込んでいます。

クラスの利用者にとって大事なのは**インタフェース部**です。実装部に関しては、たとえソースプログラムが入手できなくても、コンパイル済みのオブジェクトファイルが用意されていれば、そのクラスを利用することができます。

事実、C++ の標準ライブラリは、そのような形で提供されます。

クラス定義にコメントを適切に記入しておけば、立派なドキュメントにもなります。

> **重要** クラスはブラックボックスである。クラスの宣言であるインタフェース部は原則としてヘッダに記述する。それはクラスの“仕様書”となる。

さて、クラス *Account* では、メンバ関数 *GetName*, *GetNo*, *CheckBalance* がクラス定義中で定義されています。このことは、以下のことを示しています。

> ① インライン関数となるため、効率のよい処理が期待できる。
> ② クラスの利用者に対して、非公開部の詳細までをも暴露している。

①は好ましいことですが、②はどうでしょう。実は、C++ のクラス定義は、**非公開部の中身が見えてしまう状態で利用者に提供する**仕様となっています。

メンバ関数のインライン関数化によるプログラムの実行効率を向上させる努力を完全に放棄するのであれば、クラスの利用者に対して“公開部”のみを示すことができます。

しかし、そうすると、コンパイルの結果作成される実行プログラムは、実行速度という点での“**品質**”が低下することになります。C++ のクラスは、『C 言語と同程度の実行効率をもたなければならない』という使命を与えられているがゆえの中途半端な仕様となっているのです。C++ の本音は、

> **効率のためだったら、他人さまに見せるべきではないものを見られてもいいや！**

といったところのようです。

■ -> 演算子によるメンバのアクセス

クラス *Account* 第 4 版を利用するプログラム例を **List 9-7** に示します。

adachi と *nakata* がクラス *Account* 型のオブジェクトではなく *Account** 型のポインタとなっている点が、これまでのプログラムと違います。

new 演算子によってオブジェクトを生成していますので、ポインタ *adachi* と *nakata* が指す **adachi* や **nakata* が銀行口座のオブジェクトとなります（p.231）。

▶ **int** 型のオブジェクトを動的に生成して **x* を 5 で初期化する（p.232）のと同じ形式です。

```
int* x = new int(5);
```

一般にポインタ *p* が指すオブジェクトは **p* と表せます（p.217）から、*p* が指すクラスオブジェクト **p* のメンバ *m* は、以下の式でアクセスできることになります（**Fig.9-9**）。

```
(*p).m                          // pが指すオブジェクト*pのメンバm
```

List 9-7 Chap09/Account/AccountTester.cpp

```
// 銀行口座（第４版）の利用例
#include <string>
#include <iostream>
#include "Account.h"
using namespace std;

int main()
{
    Account* adachi = new Account("足立幸一", "12345678", 1000);
    Account* nakata = new Account("仲田真二", "87654321",  200);

    adachi->Withdraw(200);       // 足立君が200円おろす
    nakata->Deposit(100);        // 仲田君が100円預ける

    cout << "■足立君の口座：\"" << adachi->GetName() << "\" ("
         << adachi->GetNo() << ") " << adachi->CheckBalance() << "円\n";

    cout << "■仲田君の口座：\"" << nakata->GetName() << "\" ("
         << nakata->GetNo() << ") " << nakata->CheckBalance() << "円\n";

    return 0;
}
```

実行結果

```
■足立君の口座："足立幸一" (12345678) 800円
■仲田君の口座："仲田真二" (87654321) 300円
```

▶ *p を囲む（ ）を省略することはできません。アドレス演算子 * よりもメンバアクセス演算子 . の優先度のほうが高いからです。

ただし、この式は煩雑ですから、以下のように簡略表記できることになっています。

```
p->m
```

一般に**アロー演算子**（*arrow operator*）と呼ばれる **->** は、ポインタが指すオブジェクトのメンバをアクセスするための演算子です（**Table 9-2**）。

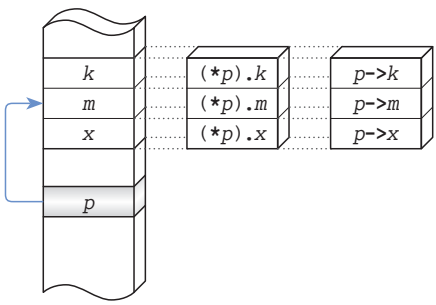


Fig.9-9 クラスオブジェクトとポインタ

本プログラムでは、メンバ関数の呼出しで、この演算子を利用しています。

Table 9-2 メンバアクセス演算子（アロー演算子）

x->*y*	*x* が指すオブジェクトのメンバ *y* をアクセスする（すなわち (*x).*y* と同じ）。

▶ アロー演算子という俗称は -> の形が**矢印**（*arrow*）に似ていることに由来します。

■ 自動車クラス

クラス定義中ですべてのメンバ関数を定義すれば、**そのクラスはヘッダだけで提供できる**ことになります。そのように定義した自動車クラスを **List 9-8**（次ページ）に示します。

自動車には現在位置を表す座標と燃料のデータがあります。燃料が残っている限り、自由に移動できるようになっています。

List 9-8 Chap09/Car.h

```
// 自動車クラス

#include <cmath>
#include <string>
#include <iostream>
using namespace std;

class Car {
    string name;                // 名前
    int width, length, height;  // 車幅・車長・車高
    double x, y;                // 現在位置座標
    double fuel;                // 残り燃料

public:
    //--- コンストラクタ ---//
    Car(string n, int w, int l, int h, double f) {
        name = n; width = w; length = l; height = h; fuel = f; x = y = 0.0;
    }

    double X() { return x; }            // 現在位置Ｘ座標を返す
    double Y() { return y; }            // 現在位置Ｙ座標を返す
    double Fuel() { return fuel; }      // 残り燃料を返す

    void PutSpec() {                                    // スペック表示
        cout << "名前：" << name << "\n";
        cout << "車幅：" << width << "mm\n";
        cout << "車長：" << length << "mm\n";
        cout << "車高：" << height << "mm\n";
    }

    bool move(double dx, double dy) {   // Ｘ方向にdx・Ｙ方向にdy移動
        double dist = sqrt(dx * dx + dy * dy);      // 移動距離

        if (dist > fuel)
            return false;               // 燃料不足
        else {
            fuel -= dist;               // 移動距離の分だけ燃料が減る
            x += dx;  y += dy;
            return true;
        }
    }
};
```


クラス *Car* には７個のデータメンバがあります。

name は自動車の名前で、*width*, *length*, *height* は車幅・車長・車高です。*x* と *y* は現在位置のＸ座標とＹ座標で、*fuel* は残り燃料です。

・コンストラクタ

座標を除いた五つのデータを受け取って各メンバにセットします。*x* と *y* の値を 0.0 とすることによって、自動車の位置を原点（0.0, 0.0）とします。

・メンバ関数 *X*, *Y*, *Fuel*

これらのメンバ関数は、Ｘ座標の値 *x*、Ｙ座標の値 *y*、残り燃料の値 *fuel* をそのまま返します。

・メンバ関数 *PutSpec*

車の名前と車幅・車長・車高を表示します。

・メンバ関数 *move*

自動車を X 方向に *dx*、Y 方向に *dy* だけ移動させます。移動する距離 *dist* は **Fig.9-10** に示す計算によって求めます。

▶ **double** 型実数値の平方根を求めるライブラリが <cmath> で提供される ***sqrt*** です。関数の形式は以下の通りです。

```
double sqrt(double);
```

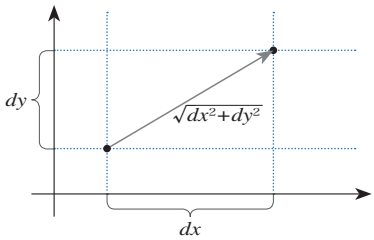


Fig.9-10 自動車の移動と距離

残り燃料 *fuel* が移動距離 *dist* に満たなければ移動不能と判断して **false** を返します。移動に必要な燃料がある場合は現在位置と残り燃料を更新して **true** を返します。

▶ 燃費を 1 と考えています。すなわち距離 1 を移動すると燃料も 1 減ります。

自動車クラスを利用するプログラム例を **List 9-9** に示します。

List 9-9 Chap09/CarTester.cpp

```
// 自動車クラスの利用例

#include <iostream>
#include "Car.h"
using namespace std;

int main()
{
    string name;
    int width, length, height;
    double gas;
    cout << "車のデータを入力せよ。\n";
    cout << "名前は：";  cin >> name;
    cout << "車幅は：";  cin >> width;
    cout << "車長は：";  cin >> length;
    cout << "車高は：";  cin >> height;
    cout << "ガソリン量は：";  cin >> gas;

    Car myCar(name, width, length, height, gas);
    myCar.PutSpec();          // スペック表示

    while (true) {
        cout << "現在地(" << myCar.X() << ", " << myCar.Y() << ")\n";
        cout << "残り燃料：" << myCar.Fuel() << '\n';
        cout << "移動[0…No／1…Yes]：";
        int move;
        cin >> move;
        if (move == 0) break;

        double dx, dy;
        cout << "X方向の移動距離：";  cin >> dx;
        cout << "Y方向の移動距離：";  cin >> dy;
        if (!myCar.move(dx, dy))
            cout << "\a燃料が足りません！\n";
    }
}
```

実行例

```
車のデータを入力せよ。
名前は：僕の愛車
車幅は：1885
車長は：5022
車高は：1490
ガソリン量は：90
名前：僕の愛車
車幅：1885mm
車長：5022mm
車高：1490mm
現在地(0, 0)
残り燃料：90
移動[0…No／1…Yes]：1
X方向の移動距離：5.5
Y方向の移動距離：12.3
現在地(5.5, 12.3)
残り燃料：76.5263
移動[0…No／1…Yes]：0
```

最初に名前や車幅などのデータを読み込んで、その値をもとにクラス *Car* 型のオブジェクト *myCar* を構築します。それからメンバ関数 *PutSpec* によってスペックを表示します。その後、現在位置の移動を対話的に繰り返します。

■ Column 9-2　C言語の構造体と共用体

C言語の**構造体**（*structure*）と**共用体**（*union*）について簡単に紹介します。

■ 構造体

C言語の構造体を用いると、日付（第 10 章）は以下のように宣言できます。

```
struct date {
    int year;       /* 西暦年 */
    int month;      /* 月 */
    int day;        /* 日 */
};
```

C++ のクラスと比べると、C言語の構造体には以下のような制限があります。

① 独立した型とならない

上のように *date* が宣言されても、“*date* 型”ができるのではなく、“構造体 *date* 型”が作られるだけです。構造体名“*date*”単独では型とはならず、“**struct** *date*”が型となります。したがって、“構造体 *date* 型”のオブジェクト *day* の定義は以下のようになります。

```
struct date day;
```

struct は省略できません。

② すべてのメンバは公開される

構造体のメンバはすべて公開されます。すなわち、全メンバが“公開部”で宣言されているものとして扱われます。したがって、クラス外部からメンバを保護することはできません。

③ メンバ関数をもつことができない

構造体のメンバとしては、データメンバのみが許されます。メンバ関数をもつことはできません。もちろん、メンバ関数の一種である“コンストラクタ”をもつこともできませんので、確実な初期化の手段を提供することはできません。初期化を行う際は、

```
struct date day = {2010, 11, 18};
```

と配列風の { } を用いた初期化子を与えなければなりません（各メンバに対する初期化子をコンマで区切ったものを { } で囲みます）。

いくつかの制限をあげましたが、次章以降で解説する“クラス”特有の機能は、ほとんどすべて使えません。

※実際には、“C++ のクラスを制限したものがC言語の構造体”ではなく、“C言語の構造体をもとにして大幅に拡張して作られたものが C++ のクラス”です。

■ 共用体

構造体が直列的なデータ構造を実現するのに対し、共用体は並列的かつ選択的なデータ構造を実現します。構造体と共用体のイメージを対比したのが **Fig.9C-1** です。

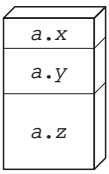
共用体であることを宣言するためのキーワードが **union** です。メンバの宣言法や、**.** 演算子と **->** 演算子によってメンバをアクセスできる点は、クラスや構造体と同じです。

なお、構造体と同様に、メンバ関数をもったり、非公開とすることはできません。

a 構造体

```
struct xyz {
    int    x;
    long   y;
    double z;
};

struct xyz a;
```



b 共用体

```
union xyz {
    int    x;
    long   y;
    double z;
};

union xyz b;
```

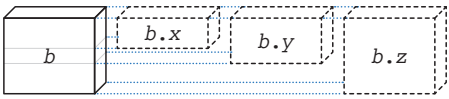


Fig.9C-1　構造体と共用体

共用体ではすべてのメンバが同一アドレス上に並びます。そのため、“全メンバが同時に存在する”というよりも、“同時には一つのメンバだけが意味をもつ”といった性格となります。

たとえば、

```
b.x = 5;        /* int型のメンバxに5を代入 */
```

という代入を行って、その直後に、

```
c = b.z;        /* double型のメンバzとして値を取り出す */
```

としても意味のない値が得られることになります。共用体は、

『メンバ *x*, *y*, *z* を同時に使うことはないから、一緒の領域に閉じこめてしまえ。』

といった目的で利用されるのです。

なお、C++ では共用体も大幅に機能が拡張されています。

＊

class, **struct**, **union** というキーワードは、《クラス》、《構造体》、《共用体》という概念と一対一で対応するものでは**ありません**。

C++ でのキーワード **struct** は、キーワード **class** とほとんど同じ意味が与えられており、クラスの宣言に使うことができます。唯一の相違点は、**struct** によってクラスを宣言すると、アクセス権を指定しないメンバが（非公開でなく）公開となるということだけです。

演習 9-1

自動車クラス *Car* にデータメンバやメンバ関数を自由に追加せよ（ナンバーを表すデータメンバを追加する、燃費を表すデータメンバを追加する、移動による燃料残量の計算に燃費を反映させる、タンク容量を表すデータメンバを追加する、給油のためのメンバ関数を追加する etc…）。

演習 9-2

名前・身長・体重などをメンバとしてもつ《人間クラス》を自由に定義せよ。

まとめ

- プログラムを作る際は、現実世界のオブジェクトをプログラムの世界のオブジェクトに投影する。
- その投影に際しては「まとめるべきものは、まとめる。」「本来まとまっているものは、そのままにする。」といった方針をとったほうが自然で素直なプログラムとなる。
- クラスはプログラムの集積回路の設計図である。任意の型の要素をデータメンバとしてもつことができる。メンバ関数をもつこともできる。
- クラス *A* のクラス定義は **class** *A* { /* … */ }; とする。末尾にはセミコロンが必要である。{ } の中はデータメンバとメンバ関数の宣言である。
- オブジェクト *x* のメンバ *m* は、*x*.*m* としてアクセスできる。
- ポインタ *p* が指すオブジェクトのメンバ *m* は、(*p).*m* あるいは *p*->*m* としてアクセスできる。
- クラスのメンバは、そのクラスの有効範囲の中に入る。したがって、クラス *C* のメンバ *m* の名前は *C*::*m* となる。
- データメンバとメンバ関数は、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。
- データ隠蔽を実現するために、原則としてデータメンバは非公開にするとよい。
- オブジェクトの生成時に呼び出される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。
- コンストラクタの名前はクラス名と同一である。返却値型はなく **void** とすら定義することはできない。
- 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。
- 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。

- クラス定義中で定義されたメンバ関数はインライン関数となる。メンバ関数は前方参照ができる。

- クラス定義は独立したヘッダとして実装する。そのヘッダをインタフェース部と呼ぶ。

- クラス定義中で定義しないメンバ関数の定義をソースファイルとして実装する。そのソースファイルを実装部と呼ぶ。