第 10 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のが、クラスの概念です。本章では、クラスの基本を学習します。

- クラスとは
- クラス定義
- ユーザ定義型
- データメンバとステート（状態）
- メンバ関数と振舞い／メッセージ
- コンストラクタ
- クラス型オブジェクトの初期化
- アクセス指定子（公開 public と非公開 private）
- 情報隠蔽
- カプセル化
- アクセッサ（ゲッタとセッタ）
- クラス有効範囲
- クラス定義の外でのメンバ関数の定義
- メンバ関数の結合性
- クラスメンバアクセス演算子（ドット演算子 . とアロー演算子 ->）
- クラスと構造体と共用体（class ／ struct ／ union）
- ヘッダ部とソース部
- ヘッダと using 指令
- string クラス

10–1 クラスの考え方

プログラムの部品である関数と、その処理対象となるデータを組み合わせた構造を表すのが、クラスです。関数より一回り大きな単位の部品であるクラスは、オブジェクト指向プログラミングを支える最も根幹的で基礎的な技術です。本節では、クラスの基本を学習します。

データの扱い

List 10-1 は、鈴木君と武田君の銀行口座に関するデータを扱うプログラムです。変数に値を設定した上で、それらの値を表示します。

List 10-1 chap10/list1001.cpp

```
// 鈴木君と武田君の銀行口座
#include <string>
#include <iostream>

using namespace std;

int main()
{
    string suzuki_name    = "鈴木龍一";      // 鈴木君の口座名義
    string suzuki_number  = "12345678";      //   〃   の口座番号
    long   suzuki_balance = 1000;            //   〃   の預金残高

    string takeda_name    = "武田浩文";      // 武田君の口座名義
    string takeda_number  = "87654321";      //   〃   の口座番号
    long   takeda_balance = 200;             //   〃   の預金残高

    suzuki_balance -= 200;                   // 鈴木君が200円おろす
    takeda_balance += 100;                   // 武田君が100円預ける

    cout << "■鈴木君の口座：\"" << suzuki_name << "\" (" << suzuki_number
         << ") " << suzuki_balance << "円\n";

    cout << "■武田君の口座：\"" << takeda_name << "\" (" << takeda_number
         << ") " << takeda_balance << "円\n";
}
```

実行結果

```
■鈴木君の口座："鈴木龍一" (12345678) 800円
■武田君の口座："武田浩文" (87654321) 300円
```

２人分の銀行口座に関するデータを６個の変数で表しています。口座名義と口座番号は ***string*** 型で、預金残高は **long** 型です。たとえば、変数 *suzuki_name* は**口座名義**で、*suzuki_number* は**口座番号**で、*suzuki_balance* は**預金残高**です。

名前が *suzuki_* で始まる変数は**鈴木君の銀行口座に関するデータである。**

ということは、変数名やコメントから推測できます。

しかし、鈴木君の口座名義を *takeda_number* としたり、武田君の口座番号を *suzuki_name* とすることも可能です。

問題は、変数間の《関係》を、変数名から**推測**できるものの**確定ができない**ことです。

バラバラに宣言された口座名義・口座番号・預金残高の変数が一つの銀行口座に関するものであるという関係は、プログラム上で表現されていません。

クラス

私たちは、プログラム作成時に、**現実世界のオブジェクト（物）や概念を、プログラムの世界のオブジェクト（変数）に投影します。**

本プログラムでは、**Fig.10-1** a に示すように、一つの《口座》の口座名義・口座番号・預金残高のデータが個別の変数へと投影されています。

▶ この図は、一般化して表したものです。鈴木君の口座・武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。

口座の一つの側面ではなく、複数の側面に着目しましょう。そうすると、図 b に示すように、口座のデータは、口座名義・口座番号・預金残高をまとめたオブジェクトとして投影できます。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。

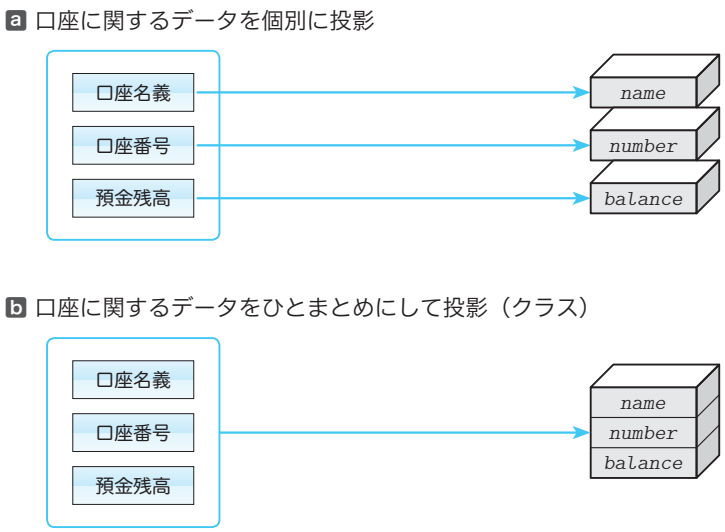

Fig.10-1　オブジェクトの投影とクラス

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、より自然で素直なプログラムとなります。

*

クラスを用いて書き直すことにしましょう。そのプログラムを **List 10-2**（次ページ）に示します。

List 10-2 chap10/list1002.cpp

```
// 銀行口座クラス（第１版）とその利用例
#include <string>
#include <iostream>

using namespace std;

class Account {
public:
    string name;    // 口座名義
    string number;  // 口座番号
    long balance;   // 預金残高
};

int main()
{
    Account suzuki;     // 鈴木君の口座
    Account takeda;     // 武田君の口座

    suzuki.name    = "鈴木龍一";    // 鈴木君の口座名義
    suzuki.number  = "12345678";    //   〃   の口座番号
    suzuki.balance = 1000;          //   〃   の預金残高

    takeda.name    = "武田浩文";    // 武田君の口座名義
    takeda.number  = "87654321";    //   〃   の口座番号
    takeda.balance = 200;           //   〃   の預金残高

    suzuki.balance -= 200;          // 鈴木君が200円おろす
    takeda.balance += 100;          // 武田君が100円預ける

    cout << "■鈴木君の口座：\"" << suzuki.name << "\" (" << suzuki.number
         << ") " << suzuki.balance << "円\n";

    cout << "■武田君の口座：\"" << takeda.name << "\" (" << takeda.number
         << ") " << takeda.balance << "円\n";
}
```

実行結果

```
■鈴木君の口座："鈴木龍一" (12345678) 800円
■武田君の口座："武田浩文" (87654321) 300円
```

1 クラス定義

2 クラス型オブジェクトの定義

クラス定義

まずは、1に着目しましょう。これは、クラス *Account* が“口座名義・口座番号・預金残高をまとめたクラス”であることを表す宣言です。

なお、この宣言は、**クラス定義**（*class definition*）と呼ばれます。

先頭の“**class** *Account* {”がクラス定義の開始であり、そのクラス定義は“};”まで続きます。関数定義とは違い、**クラス定義の末尾にはセミコロンが必要です。**

{ } の中は、クラスの構成要素である**メンバ**（*member*）の宣言です。クラス *Account* を構成するのは、右に示す三つのメンバです。

- 口座名義を表す ***string*** 型の *name*
- 口座番号を表す ***string*** 型の *number*
- 預金残高を表す **long** 型の *balance*

いずれも値をもつ変数であって、このようなメンバは、**データメンバ**（*data member*）と呼ばれます。

クラス *Account* の定義とその構造を示したのが、**Fig.10-2** です。**配列**は同一型の要素を組み合わせて作る型でしたが、**クラス**は、任意の型の要素を組み合わせて作る型です。

なお、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。

▶ **public** とコロン：のあいだには空白類を入れても構いません。

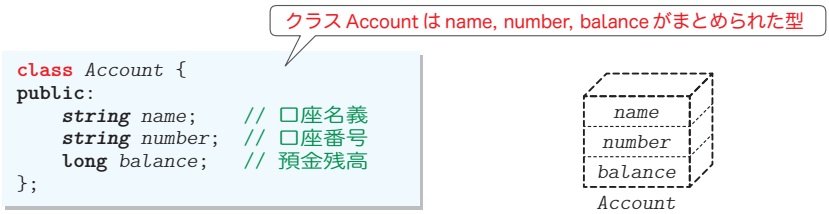

Fig.10-2　クラス定義とデータメンバの構成

クラス型のオブジェクト

クラス定義は、(名前は定義であるものの) 単なる《**型**》の**宣言**です。クラス *Account* 型の**実体**である《**オブジェクト**》を**宣言・定義**しているのが、**2**の箇所です。

この宣言を理解しやすくするために、**int** 型オブジェクトの宣言と並べてみます。

```
型名      オブジェクト名;
int       x;              // int      型のオブジェクトx       の宣言・定義
Account suzuki;           // Account型のオブジェクトsuzukiの宣言・定義
Account takeda;           // Account型のオブジェクトtakedaの宣言・定義
```

Account が型名で、*suzuki* や *takeda* がオブジェクト名であることがはっきりします。

さて、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・定義すると、カタから作られた本物の“**タコ焼き**”ができます（**Fig.10-3**）。

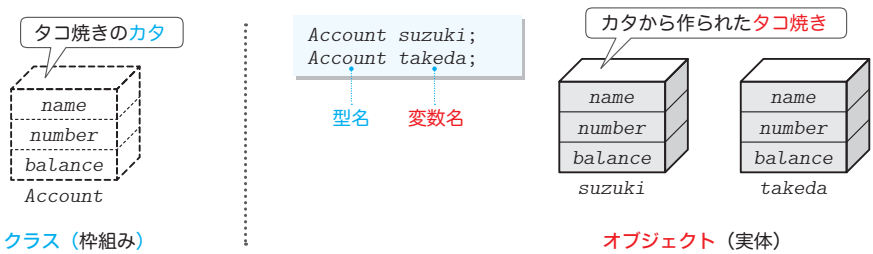

Fig.10-3　クラスとオブジェクト

ユーザ定義型

整数や実数などの数値を表現する **int** 型や **double** 型などの、言語が提供する型が**組込み型**と呼ばれることは、第4章で学習しました（p.132）。

これに対して、銀行口座のデータ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）。このような型は、**ユーザ定義型**（*user-defined type*）と呼ばれます。

メンバのアクセス

プログラムでは、鈴木君と武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のアクセスに使うのが、**Table 10-1** に示す**クラスメンバアクセス演算子**（*class member access operator*）です。この演算子は、**. 演算子**（*.operator*）や**ドット演算子**（*dot operator*）とも呼ばれます。

Table 10-1　クラス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（dot）は『点』という意味です。『クラスメンバアクセス演算子』という名称は、ドット演算子 . と、**Table 10-2**（p.359）で学習するアロー演算子 **->** の総称です。

たとえば、鈴木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。

```
suzuki.name          // 鈴木君の口座名義
suzuki.number        //   〃   の口座番号
suzuki.balance       //   〃   の預金残高
```

武田君の口座を表すオブジェクト *takeda* も同様です（**Fig.10-4**）。

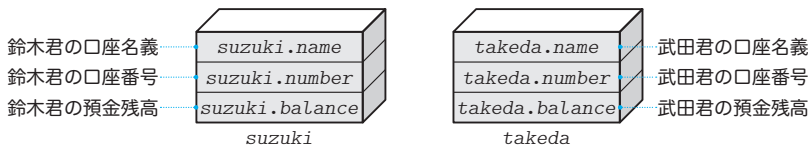

Fig.10-4　データメンバのアクセス

問題点

クラスの導入によって、口座のデータを表す変数間の関係がプログラム中に明確に埋め込まれました。しかし、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ため、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。初期化すべきオブジェクトは、初期化を強制するとよさそうです。

② データの保護に対する無保証

鈴木君の預金残高である *suzuki.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、鈴木君でなくても（通帳や印鑑がなくても）、鈴木君の口座から自由にお金をおろせるということです。

口座番号を公開することはあっても、預金残高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改良したのが、**List 10-3** のプログラムです。
クラス *Account* が複雑になった一方で、それを利用する **main** 関数が簡潔になっています。

List 10-3 chap10/list1003.cpp

```
// 銀行口座クラス（第２版）とその利用例
#include <string>
#include <iostream>

using namespace std;

class Account {
private:
    string full_name;        // 口座名義
    string number;           // 口座番号
    long crnt_balance;       // 預金残高

public:
    //--- コンストラクタ ---//
    Account(string name, string num, long amnt) {
        full_name = name;        // 口座名義
        number = num;            // 口座番号
        crnt_balance = amnt;     // 預金残高
    }
    //--- 口座名義を調べる ---//
    string name() {
        return full_name;
    }
    //--- 口座番号を調べる ---//
    string no() {
        return number;
    }
    //--- 預金残高を調べる ---//
    long balance() {
        return crnt_balance;
    }
    //--- 預ける ---//
    void deposit(long amnt) {
        crnt_balance += amnt;
    }
    //--- おろす ---//
    void withdraw(long amnt) {
        crnt_balance -= amnt;
    }
};

int main()
{
    Account suzuki("鈴木龍一", "12345678", 1000);        // 鈴木君の口座
    Account takeda("武田浩文", "87654321",  200);        // 武田君の口座

    suzuki.withdraw(200);            // 鈴木君が200円おろす
    takeda.deposit(100);             // 武田君が100円預ける

    cout << "■鈴木君の口座：\"" << suzuki.name() << "\" (" << suzuki.no()
         << ") " << suzuki.balance() << "円\n";

    cout << "■武田君の口座：\"" << takeda.name() << "\" (" << takeda.no()
         << ") " << takeda.balance() << "円\n";
}
```

実行結果

```
■鈴木君の口座："鈴木龍一" (12345678) 800円
■武田君の口座："武田浩文" (87654321) 300円
```

▶ 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。ご自身でプログラムを作る際は、スペース・タブ・改行を入れるとともに、詳細なコメントを記入して、ゆとりある表記をするようにしましょう。

クラス *Account* 第２版の骨格を右に示しています。第１版と異なるのは、主として以下の３点です。

- データメンバの宣言の前に置かれていた **public:** が **private:** に変更されている。
- **public:** 以降で関数が定義されている。
- 口座名義と預金残高のデータメンバの変数名が変更されている。

```
class Account {
private:                                  非公開
    string full_name;    // 口座名義
    string number;       // 口座番号
    long crnt_balance;   // 預金残高
public:                                   公 開
    Account(string , string, long)
                              { /*…*/ }
    string name()             { /*…*/ }
    string no()               { /*…*/ }
    long balance()            { /*…*/ }
    void deposit(long)        { /*…*/ }
    void withdraw(long)       { /*…*/ }
};
```

第２版のイメージを表した **Fig.10-5** を見ながら、これらの点を理解していきましょう。

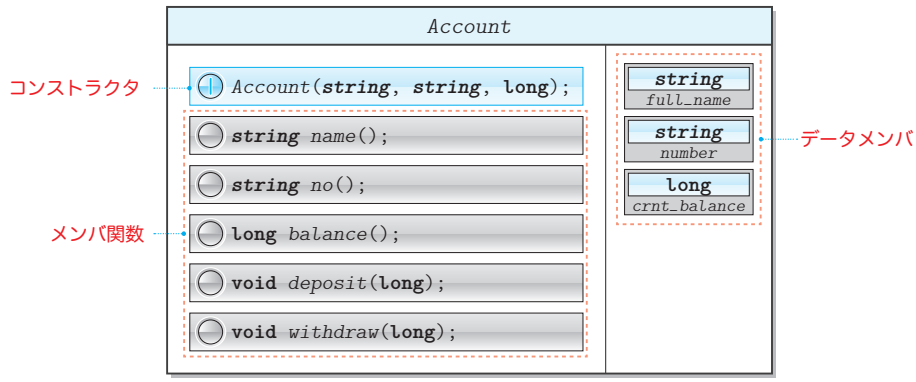

Fig.10-5　クラス Account の構造

非公開メンバと公開メンバ

クラス *Account* の第１版では、**public:** によって全データメンバを**公開**していました。第２版では、全データメンバを**非公開**にしています。**非公開メンバは、クラスの外部に対して存在を隠します**。非公開を指示するのが、**private:** です。

重要 **private** 宣言されたメンバは、クラスの外部に対して**非公開**となる。

そのため、クラス *Account* にとって外部の存在である **main** 関数では、非公開のデータメンバ *full_name*, *number*, *crnt_balance* のアクセスは不可能です。もし **main** 関数に以下のようなコードがあれば、いずれもコンパイルエラーとなります。

```
suzuki.full_name = "柴田望洋";   // エラー：鈴木君の口座名義を書きかえる
suzuki.number = "99999999";      // エラー：鈴木君の口座番号を書きかえる
cout << suzuki.crnt_balance;     // エラー：鈴木君の預金残高を表示
```

情報を公開するかどうかを決定するのはクラス側です。『お願いですから、このデータ

を特別に見せてください。』と、クラスの外部から依頼することはできません。

みなさんは、各種のパスワードや暗証番号を秘密にしているでしょう。それと同じです。

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。データメンバを非公開としてデータ隠蔽を行えば、**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。**

すべてのデータメンバは、非公開とするのが原則です。

> **重要** データ隠蔽を実現してプログラムの品質を向上させるために、クラス内のデータメンバは、原則として非公開にすべきである。

▶ この後で学習するように、データメンバの値は、コンストラクタとメンバ関数を通じて間接的に読み書きできます。そのため、データメンバを非公開とすることによって不都合が生じることは基本的にありません。

`public:` も `private:` も指定されていなければ、メンバは**非公開**となります。また、クラス定義中には、`public:` や `private:` は何度出てきても構いません。再び `public:` や `private:` が現れるまで、指定は有効です。以上の規則をまとめたのが、**Fig.10-6** です。

なお、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、**限定公開**を指定する `protected` があります。

クラス *Account* の三つのデータメンバは、クラス定義中の先頭で宣言されています。そのため、仮に `private:` の指定を取り除いたとしても、これらは非公開となります。

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

- アクセス指定がなければ非公開
- public: 以降は公開となる
- private: 以降は非公開となる
- public: や private: の順序は任意であって何度現れても構わない

Fig.10-6　アクセス指定とメンバの宣言

コンストラクタとメンバ関数

第 1 版の *Account* は、データメンバだけで構成されていました。第 2 版では、データメンバ以外に、《コンストラクタ》と《メンバ関数》と呼ばれる関数が追加されています。それぞれについて学習していきましょう。

▶ 第 2 版のクラス *Account* の定義では、“データメンバ➡コンストラクタ➡メンバ関数”の順に宣言が並んでいます。それぞれは、まとまっている必要もありませんし、順序も任意です。

コンストラクタ

クラス名と同じ名前の関数が**コンストラクタ**（*constructor*）です。コンストラクタの役割は、**オブジェクトを確実かつ適切に初期化する**ことです。

▶ 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**とも呼ばれます。

コンストラクタが呼び出されるのは、クラス型オブジェクトの生成時です。すなわち、プログラムの流れが以下の宣言文を通過して、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。

```
1 Account suzuki("鈴木龍一", "12345678", 1000);     // 鈴木君の口座
2 Account takeda("武田浩文", "87654321",  200);     // 武田君の口座
```

これらの宣言の形式は、次のようになっています。

```
クラス名 変数名 ( 実引数の並び );
```

▶ これは、組込み型の変数を ( ) 形式の初期化子で初期化する宣言（p.271）と同じ形式です。

```
int x(5);          // int型変数xを5で初期化：int x = 5;と同じ
```

コンストラクタの働きを示したのが、**Fig.10-7** です。呼び出されたコンストラクタは、仮引数 *name*，*num*，*amnt* に受け取った三つの値を、それぞれデータメンバ *full_name*，*number*，*crnt_balance* に代入します。

代入先の式は、*suzuki.number* や *takeda.number* ではなく、“単なる *number*”です。1 で呼び出されたコンストラクタでの *number* は *suzuki.number* のことであり、2 で呼び出されたコンストラクタでの *number* は *takeda.number* のことです。

このようにデータメンバの名前だけで表せるのは、コンストラクタが、**自分自身のオブジェクトが何であるかを知っているからです**。図に示すように、個々のオブジェクトに対して、専用のコンストラクタが存在します。

換言すると、**“コンストラクタは、特定のオブジェクトに所属する”**のです。たとえば、1 で呼び出されたコンストラクタは *suzuki* に所属して、2 で呼び出されたコンストラクタは *takeda* に所属します。

▶ 個々のオブジェクトに専用のコンストラクタを用意するのは、現実には不可能です。“コンストラクタが特定のオブジェクトに所属する”というのは、概念上のものであって、物理的にそうなっているわけではありません。コンパイルの結果生成されるコンストラクタ用の内部的なコードは、実際は 1 個だけです。

コンストラクタは、非公開データメンバ *full_name*，*number*，*crnt_balance* に自由にアクセスする権利をもっています。というのも、コンストラクタはクラス *Account* にとって内輪（うちわ）の存在だからです。

*

さて、1 と 2 の宣言を以下のように書きかえてみましょう。そうすると、コンパイルエラーが発生します。

```
Account suzuki;                  // エラー：引数がない
Account takeda("武田浩文");      // エラー：引数が不足
```

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**ことが分かります。

重要 クラス型を作るときは、コンストラクタを用意して、オブジェクトを確実かつ適切に初期化する手段を提供しよう。

なお、コンストラクタは**値を返却できません**。

▶ すなわち、コンストラクタの宣言で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）。

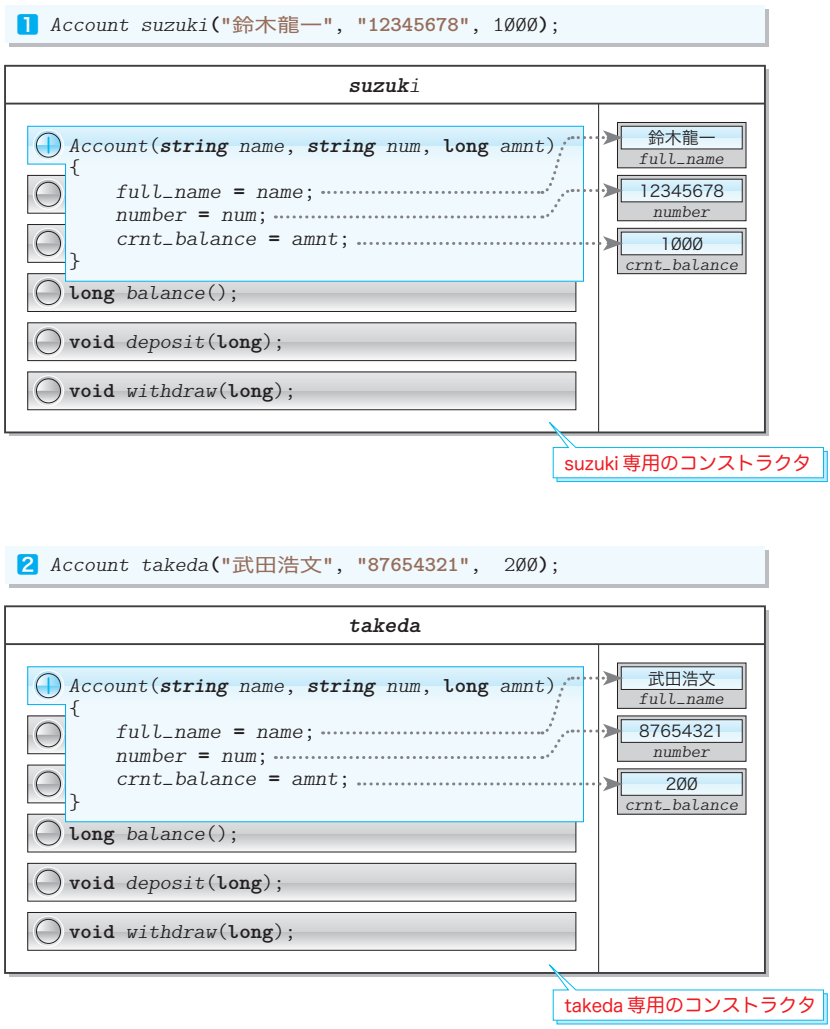

Fig.10-7 オブジェクトとコンストラクタ

メンバ関数とメッセージ

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が**メンバ関数**（*member function*）です。

▶ 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５個のメンバ関数があります。

- *name*　：口座名義を ***string*** 型で返す。
- *no*　：口座番号を ***string*** 型で返す。
- *balance*　：預金残高を **long** 型で返す。
- *deposit*　：お金を預ける（預金残高を増やす）。
- *withdraw*　：お金をおろす（預金残高を減らす）。

前章までに学習してきた関数とは異なり、クラスのメンバ関数は、そのクラスの個々のオブジェクトごとに作られます。そのため、*suzuki* も *takeda* も、自分専用のメンバ関数 *name*, *no*, *balance*, … をもちます。

換言すると、**“メンバ関数は、特定のオブジェクトに所属する”**のです。

重要 **メンバ関数**は、概念的には、個々のオブジェクトごとに作られて、そのオブジェクトに所属する。

▶ 個々のオブジェクトごとにメンバ関数が作られるというのは、あくまでも概念上のものです。コンストラクタと同様に、コンパイルの結果作られる内部的なコードは１個です。

メンバ関数は、クラスにとって内輪の存在ですから、非公開のデータメンバに自由にアクセスできます。この点はコンストラクタと共通です。

また、メンバ関数の中では、*suzuki.number* や *takeda.number* ではなく、単なる *number* によって、自分が所属するオブジェクトの口座番号のデータメンバにアクセスできます。この点も、コンストラクタと同じです。

メンバ関数の呼出しには、**Table 10-1**（p.338）で学習した**ドット演算子 .** を利用します。以下に示すのが、メンバ関数呼出し式の一例です。

```
1 suzuki.balance()          // 鈴木君の預金残高を調べる
2 suzuki.withdraw(200)      // 鈴木君の口座から200円おろす
3 takeda.deposit(100)       // 武田君の口座に100円預ける
```

1 の呼出しによって、鈴木君の預金残高を調べる様子を示したのが、**Fig.10-8** です。*suzuki* に対して呼び出されたメンバ関数 *balance* は、データメンバ *crnt_balance* の値を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口座番号や預金残高などの非公開のデータメンバも、**メンバ関数を通じて間接的にアクセスできます。**

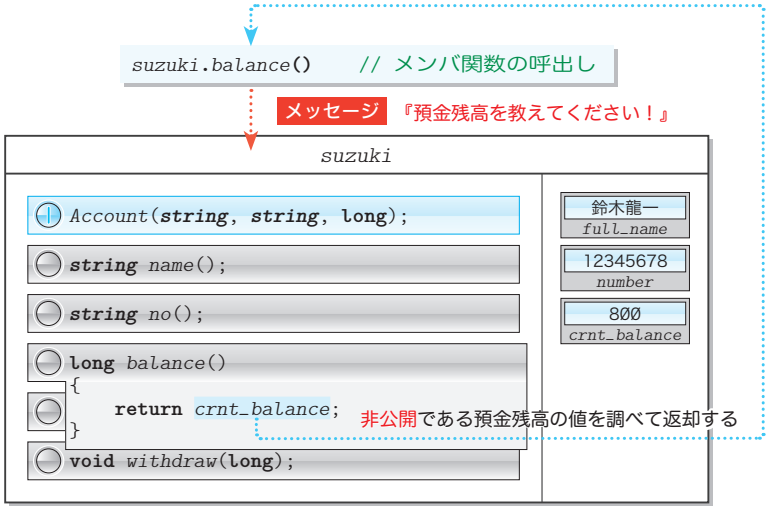

Fig.10-8　メンバ関数の呼出しとメッセージ

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”。

図に示すように、メンバ関数の呼出し式 *suzuki*.balance() は、オブジェクト *suzuki* に対して『預金残高を教えてください！』というメッセージを送ります。

その結果、オブジェクト *suzuki* は『預金残高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、『○○円ですよ。』と返答します。

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ては、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、自動的にインライン関数**（p.232）**となる**からです。そのため、

```
suzuki.balance()
```

は、次に示すのと同等なコードに変換された上でコンパイルされます。

```
x = suzuki.crnt_balance;      // 実質的なコード
```

▶　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を第６章で学習しました。メンバ関数も同様です。

データメンバとアクセッサ

クラス *Account* の第２版への改良にあたっては、データメンバの名前を変更しています。口座名義を *name* から *full_name* に変更し、預金残高を *balance* から *crnt_balance* に変更しています。そして、口座名義を調べるメンバ関数名を *name* にして、預金残高を調べるメンバ関数名を *balance* にしています。

『データメンバと、その値を調べるメンバ関数は、同一名にすればよいのではないか。』と疑問をもたれたかもしれません。しかし、以下に示す制限があります。

重要 同じクラスに所属するデータメンバとメンバ関数とが同一の名前をもつことは許されない。

さて、メンバ関数 *name*，*no*，*balance* は、それぞれ、データメンバ *full_name*，*number*，*crnt_balance* の値を調べて返却するだけの働きをします。このように、**特定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。**

▶ クラス *Account* には、ゲッタはありますが、セッタはありません。

メンバ関数とコンストラクタ

コンストラクタは、特殊なメンバ関数です。そのため、次のように、生成ずみ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。

```
suzuki.suzuki("鈴木龍一", "12345678", 5000);    // コンパイルエラー
```

＊

さて、第 i 版のクラス *Account* では、コンストラクタを定義していないにもかかわらず、オブジェクトの生成が可能でした。

実は、コンストラクタを定義していないクラスには、本体が空であって引数を受け取らないコンストラクタが、コンパイラによって自動的に作られます。なお、そのコンストラクタは、公開アクセス性をもつインライン関数です。

重要 コン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 **public** で **inline** のコンストラクタが、コンパイラによって自動的に定義される。

すなわち、第 1 版のクラス *Account* には、以下のコンストラクタがコンパイラによって自動的に作られていたのです。

```
class Account {
public:
    Account() { }    // コンパイラによって自動的に作られたコンストラクタ
};
```

Column 10-1 データメンバとゲッタの命名

データメンバと、そのゲッタの名前を同一にできないため、それらの命名に関して、いくつかのスタイルが考案されています。

① データメンバとゲッタ名を異なる名前とする

データメンバとゲッタに、まったく異なる名前を与えます。

```
class C {
    int number;
    string full_name;
public:
    int no() { return number; }              // numberのゲッタ
    string name() { return full_name; }      // full_nameのゲッタ
};
```

クラス Account 第２版で採用したスタイルです。データメンバとゲッタの名前の両方を考案した上で、名前を使い分ける必要があるという点で、クラス開発者に負担がかかります。ただし、このことを裏返すと、クラス利用者に公開されたゲッタ名から、非公開のデータメンバ名を推測される危険性がなくなる、という長所につながります。

② データメンバ名に下線を付ける

データメンバ名の末尾に下線 _ を付けておき、ゲッタの名前は、下線を外したものとします。

```
class C {
    int no_;
    string name_;
public:
    int no() { return no_; }                 // no_のゲッタ
    string name() { return name_; }          // name_のゲッタ
};
```

データメンバ名に下線が付くため、プログラムの記述や解読が行いにくくなり、クラス開発者に負担がかかります。また、非公開であるデータメンバ名が、クラス利用者に推測されてしまう可能性があります。

なお、下線をデータメンバ名の後ろでなく、前に付けるスタイルもあります。

③ ゲッタの先頭に get_ を付ける

データメンバ名の前に get_ を付けたものをゲッタの名前とします。

```
class C {
    int no;
    string name;
public:
    int get_no() { return no; }              // noのゲッタ
    string get_name() { return name; }       // nameのゲッタ
};
```

命名規則がシンプルであるため、クラス開発者がメンバ関数の命名に迷うこともありませんし、プログラムの記述も楽に行えます。その一方で、メンバ関数名が長くなり、非公開のデータメンバ名がクラス利用者にバレてしまいます。

なお、Java では、フィールド（データメンバ）abc のゲッタ名を getAbc とし、セッタ名を setAbc とするスタイルが広く使われています。

C++ が提供する標準ライブラリのメンバ関数名は、シンプルなものが多く、①や②のようなスタイルで命名が行われています。③のように、get_ が付けられたメンバ関数は存在しません。

クラスとオブジェクト

一般に、メンバ関数は、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します。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、*suzuki*.balance() は、オブジェクト *suzuki* の預金残高の値を調べて返却します。また、*takeda*.deposit(100) は、オブジェクト *takeda* の預金残高の値を 100 だけ増やします。

データメンバを非公開として外部から保護した上で、メンバ関数とうまく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。

Fig.10-9 に示すのは、クラス *Account* 型と、その型の二つのオブジェクトです。

図**a**のクラスを「回路」の《**設計図**》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、図**b**に示す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、受け取った口座名義と口座番号と預金残高を各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、《**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ね。

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、データメンバの値は、**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 *crnt_balance* は、現在の預金残高がいくらなのか、という状態を long 型の整数値として表します。

Column 10-2 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、宣言されていない変数や関数の**前方参照**（自分より後ろ側で定義された変数や関数をアクセスしたり呼び出したりすること）ができません（p.196）。ところが、クラスのメンバ関数には、そのような制限は課せられません。**同一クラス内であれば、後方で宣言・定義されている変数や関数にアクセスできます。**以下に示すのが、コードの一例です。

```
class C {
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; }        // 後方で宣言されている変数のアクセス
private:
    int x;
};
```

関数 *func1* では、自分より後方で定義されている関数 *func2* を呼び出しています。また、その関数 *func2* では、自分より後方で宣言されている変数（データメンバ）をアクセスしています。コンパイルエラーとならないのは、コンパイラが、クラス定義を一通り最後まで読んだ後で、メンバ関数を含むクラス全体のコンパイル作業を始めるからです。

ここに示すクラス *C* は、公開メンバが先頭側、非公開メンバが末尾側で宣言・定義されています。このように、公開メンバを先頭側で宣言・定義すべきである、という原則を採用するプログラミングスタイルもあります（本書では、データメンバを先頭側に置くスタイルを採用しています）。

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。

▶ たとえば、非公開である預金残高 *crnt_balance* の値（状態）は、外部からは直接見ることができません。その代わり、*balance*() ボタンを押すことによって調べられるようになっています。

C 言語のプログラムや、本書の前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できます。

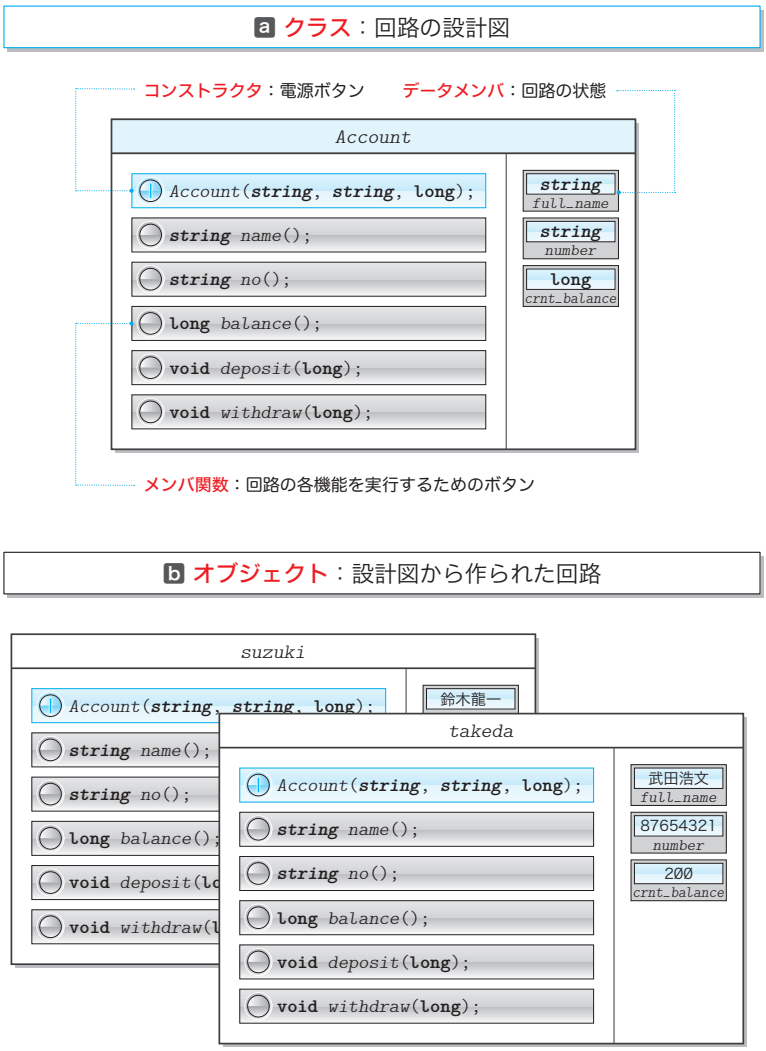

Fig.10-9　クラスとオブジェクト

10–2 クラスの実現

本節では、クラスを記述するための、ソースプログラムやヘッダの実現法などを学習していきます。

クラス定義の外でのメンバ関数の定義

第２版の口座クラス *Account* 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、クラスのすべてを単一のソースファイルで管理するのは困難です。

そのため、コンストラクタを含め、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。**List 10-4** に示すのが、コンストラクタと二つのメンバ関数 *deposit* と *withdraw* の関数定義を、クラス定義の外に移動したプログラムです。

▶ 第２版のクラス定義の冒頭にあった **private:** は削除しています。

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。すなわち、宣言・定義は、以下のように行います。

1. クラス定義の**中**ではメンバ関数の**関数宣言**を行う。
2. クラス定義の**外**ではメンバ関数の**関数定義**を行う。

さて、コンストラクタを含め、クラスの外で定義されたメンバ関数の名前は、以下に示す形式となっています。

クラス名 :: メンバ関数名

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

たとえば、預金残高を調べるメンバ関数 *deposit* は、

クラス *Account* に所属する *deposit*

ですから、*Account*::*deposit* となります。

重要 クラス *C* のメンバ関数 *func* は、クラス定義の外では以下の形式で定義する。

返却値型 *C*::*func*(仮引数宣言節) { /* … */ }

クラス定義の中で定義されたメンバ関数が、自動的にインライン関数とみなされることを p.345 で学習しました。

一方、**クラス定義の外で定義されたメンバ関数は、インライン関数ではありません。**

▶ そのため、コンストラクタとメンバ関数 *deposit* および *withdraw* は、非インライン関数です。

List 10-4 chap10/list1004.cpp

```
// 銀行口座クラス（第３版：メンバ関数の定義を分離）とその利用例

#include <string>
#include <iostream>

using namespace std;

class Account {
    string full_name;       // 口座名義
    string number;          // 口座番号
    long crnt_balance;      // 預金残高

public:
    Account(string name, string num, long amnt);    // コンストラクタ   宣言

    string name()  { return full_name; }            // 口座名義を調べる
    string no()    { return number; }               // 口座番号を調べる
    long balance() { return crnt_balance; }         // 預金残高を調べる

    void deposit(long amnt);                        // 預ける           宣言
    void withdraw(long amnt);                       // おろす           宣言
};

//--- コンストラクタ ---//
Account::Account(string name, string num, long amnt)
{
    full_name = name;       // 口座名義
    number = num;           // 口座番号
    crnt_balance = amnt;    // 預金残高
}                                                   定義

//--- 預ける ---//
void Account::deposit(long amnt)
{
    crnt_balance += amnt;
}                                                   定義

//--- おろす ---//
void Account::withdraw(long amnt)
{
    crnt_balance -= amnt;
}                                                   定義

int main()
{
    Account suzuki("鈴木龍一", "12345678", 1000);       // 鈴木君の口座
    Account takeda("武田浩文", "87654321",  200);       // 武田君の口座

    suzuki.withdraw(200);           // 鈴木君が200円おろす
    takeda.deposit(100);            // 武田君が100円預ける

    cout << "■鈴木君の口座：\"" << suzuki.name() << "\" (" << suzuki.no()
         << ") " << suzuki.balance() << "円\n";

    cout << "■武田君の口座：\"" << takeda.name() << "\" (" << takeda.no()
         << ") " << takeda.balance() << "円\n";
}
```

実行結果

```
■鈴木君の口座："鈴木龍一" (12345678) 800円
■武田君の口座："武田浩文" (87654321) 300円
```

実行効率が重視されるプログラムを開発するには、この点を押さえておかなければなりません。

重要 クラス定義の外で定義されたメンバ関数は、インライン関数ではない。

▶ インライン関数にするためには、明示的に **inline** を付けて定義する必要があります。

ヘッダ部とソース部の分離

これまでのプログラムは、クラスの定義と、それを利用する main 関数を単一のソースファイルで実現しています。

とはいえ、設計・開発から利用までのすべてを一人で行って、しかもそれが単一のソースファイルに収まるのは、小規模なクラスに限られます。

クラスを利用しやすくするには、独立したファイルとすべきです。また、クラスの利用者にとって、メンバ関数の宣言は必要ですが、必ずしも定義は必要ではありません。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現すべきです。そのため、クラスの一般的な構成は **Fig.10-10** のようになります。

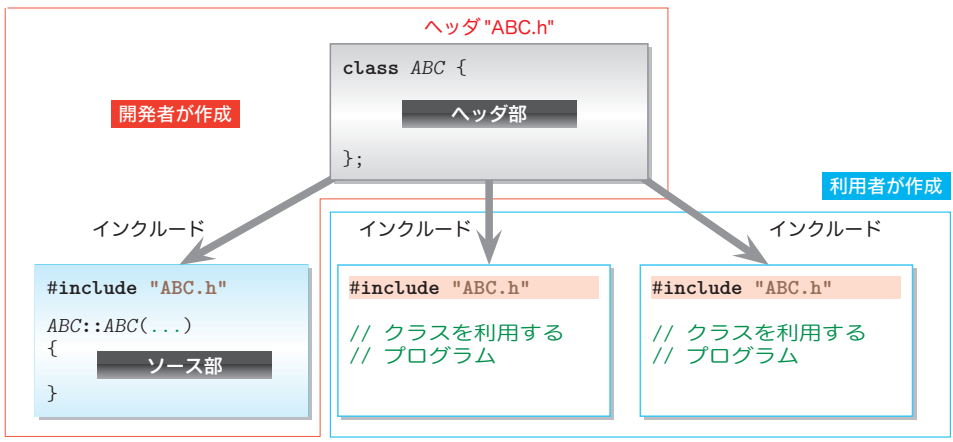

Fig.10-10　クラスの実現

クラスの開発者は、以下の二つのソースファイルを作ります。

- **ヘッダ部** … クラス定義などを含む。
- **ソース部** … メンバ関数の定義などを含む。

▶ 図に示すのは、ソース部が単一のファイルで実現できる例です。大規模なクラスになると、ソース部自体が複数個のファイルに分割されます。なお、本書では、クラスのヘッダ部を保存するファイルには、拡張子 .h を付けます。

クラス定義を含むヘッダ部は、クラスを開発・利用するプログラムにとっての《**窓口**》です。窓口をヘッダ "ABC.h" で供給するのですから、クラスのソース部でも、クラスを利用するプログラムでも、

```
#include "ABC.h"
```

とインクルードすることによって、*ABC* のクラス定義を取り込みます。

クラスの利用者にとって、ヘッダ部は必須です。クラス定義がなければコンパイルが行えないからです。その一方で、ソース部は必須ではありません。というのも、コンパイルずみオブジェクトファイルが用意されていれば、それをリンクすることでクラスを利用できるからです。実際、C++ の標準ライブラリは、ヘッダ部のみが提供され、ソース部はコンパイルずみのライブラリファイルとして提供されるのが一般的です。

*

ヘッダ部とソース部を独立したファイルとして実現した銀行口座クラスのプログラムを作りましょう。**List 10-5** に示すのがヘッダ部で、**List 10-6** に示すのがソース部です。

List 10-5 Account04/Account.h

```
// 銀行口座クラス（第４版：ヘッダ部）

#include <string>

class Account {
    std::string full_name;  // 口座名義
    std::string number;     // 口座番号
    long crnt_balance;      // 預金残高

public:
    Account(std::string name, std::string num, long amnt);  // コンストラクタ

    std::string name()  { return full_name; }       // 口座名義を調べる
    std::string no()    { return number; }          // 口座番号を調べる
    long balance()      { return crnt_balance; }    // 預金残高を調べる

    void deposit(long amnt);                        // 預ける
    void withdraw(long amnt);                       // おろす
};
```

List 10-6 Account04/Account.cpp

```
// 銀行口座クラス（第４版：ソース部）

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

using namespace std;

//--- コンストラクタ ---//
Account::Account(string name, string num, long amnt)
{
    full_name = name;       // 口座名義
    number = num;           // 口座番号
    crnt_balance = amnt;    // 預金残高
}

//--- 預ける ---//
void Account::deposit(long amnt)
{
    crnt_balance += amnt;
}

//--- おろす ---//
void Account::withdraw(long amnt)
{
    crnt_balance -= amnt;
}
```

ヘッダと using 指令

List 10-5 のヘッダ部には、これまでのプログラムとは異なる点があります。それは、文字列を表す ***string*** 型を、“std::***string***”で表していることです（Fig.10-11 a）。

string クラスは、std 名前空間に所属する型です。そのため、図 b に示すように、ヘッダ内に **using** 指令を置けば、単なる ***string*** で表せます。

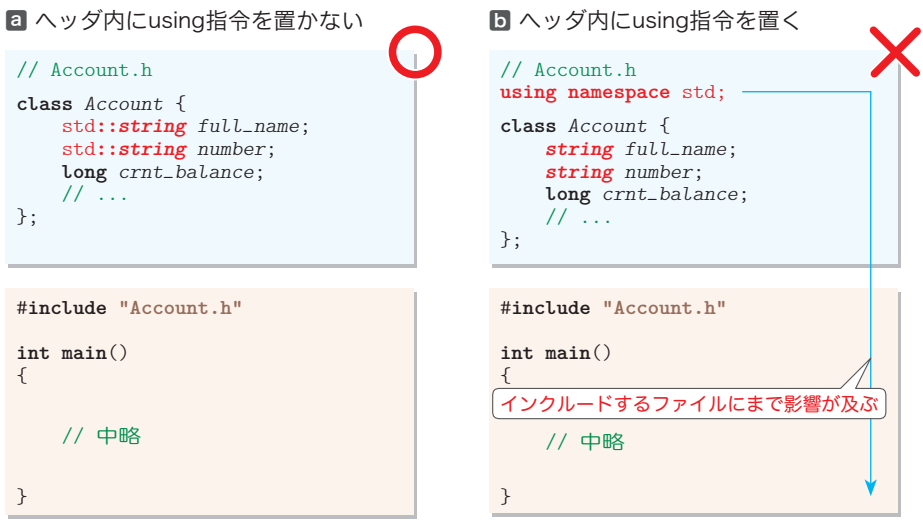

Fig.10-11　ヘッダ内の using 指令の有無

ただし、図 b のように実現されたヘッダには、一つ大きな問題が潜んでいます。それは、そのヘッダをインクルードするソースファイルで、“**using namespace** std;”という **using** 指令が有効になってしまうことです。もちろん、クラスの利用者が、必ずしも、そのような状況を好むわけではありません。そのため、以下の教訓が導かれます。

重要　原則として、ヘッダの中に **using** 指令を置いてはならない。

▶　たとえば、名前空間 *hakata* に所属する博多弁文字列クラス *hakata*::*string* を自作して、標準ライブラリ std::***string*** と使い分けているとします。そのような場合、ヘッダをインクルードするだけで、“**using namespace** std;”の **using** 指令が勝手に有効になると、不都合が生じます。

なお、List 10-6 のソース部には、**using** 指令が置かれています。ここでの **using** 指令は、このソースファイルの中でのみ通用するものであって、他のソースファイルに影響を与えないからです。

▶　もちろん、冒頭の **using** 指令を削除した上で、すべての ***string*** を std::***string*** に書きかえることもできます。

Column 10-3 string クラス

文字列を扱う ***string* クラス**は、第１章からたびたび利用してきました。実は、C++ の標準ライブラリには、“***string***”という名称のクラスは存在しません。その正体は、**クラステンプレート *basic_string* を“明示的に特殊化した”テンプレートクラス**です。

※第９章では、関数テンプレートや特殊化について学習しました。本書では学習しないのですが、テンプレートは、関数だけでなく、クラスにも適用できます。

クラステンプレート ***basic_string*** を文字 **char** 用に特殊化したテンプレートクラスが ***string*** で、ワイド文字 **wchar_t** 用に特殊化したテンプレートクラスが ***wstring*** です。

ここでは、重要な事項のみを簡単に学習します。

▪ 文字列の長さ

文字列の格納先は動的に確保されます。文字列の長さは、非公開データメンバによって管理されています（ナル文字を末尾に配置して終端の目印とするＣ言語の手法は使われていません）。

▪ 容量

文字列の長さに応じて必要な記憶域の大きさが増減するため、格納できる文字数である《**容量**》を指定したり予約したりできるようになっています。長さを調べる size、容量を指定する resize、最低限の容量を予約する reserve といったメンバ関数が提供されます。

▪ 要素のアクセス

文字列内の個々の文字をアクセスする手段として、以下に示す２種類の方法が提供されます。

- 添字の範囲をチェックしない添字演算子 **[]**
- 不正な添字に対して ***out_of_range*** の例外を送出するメンバ関数 at

string クラスを利用するプログラム例を、**List 10C-1** に示します。

List 10C-1 chap10/list10c01.cpp

```
// stringクラスの利用例

#include <string>
#include <iostream>

using namespace std;

int main()
{
    string s1 = "ABC";
    string s2 = "HIJKLMN";
    string digits = "0123456789";

    s1 += "DEF";                              // s1の末尾に"DEF"を連結
    s1 += 'G';                                // s1の末尾に'G'を連結
    s1 += s2;                                 // s1の末尾に"HIJKLMN"を連結
    s1.insert(6, digits.substr(5, 3));        // s1[6]に"567"を挿入

    s2.replace(3, 2, "kl");                   // s2[3]～s2[4]を"kl"に置換
    s2.erase(6);                              // s2[6]を削除

    cout << "s1 = ";
    for (int i = 0; i < s1.length(); i++)
        cout << s1[i];
    cout << '\n';
    cout << "s2 = " << s2 << '\n';
}
```

実行結果

```
s1 = ABCDEF567GHIJKLMN
s2 = HIJklM
```

メンバ関数の結合性

クラス定義の中で定義されたメンバ関数がインライン関数となり、クラス定義の外で定義されたメンバ関数がインライン関数とならないことを、本節で学習しました。

▶ これは、**inline** 指定子を明示的に与えない場合でのことです。クラス定義の外での関数定義に **inline** 指定子を付けると、インライン関数となります。

また、メンバ関数ではない通常の関数について、インライン関数は内部結合をもち、そうでない関数は（**static** を付けて宣言しない限り）外部結合をもつことを、第 6 章で学習しました。メンバ関数の場合も同様であり、次のようになります。

重要 クラス定義の中で定義されたメンバ関数は**内部結合**をもち、クラス定義の外で（明示的に **inline** を指定せずに）定義されたメンバ関数は**外部結合**をもつ。

このことについて、クラスの中で定義されたメンバ関数 *balance* と、クラスの外で定義されたメンバ関数 *deposit* を例に、**Fig.10-12** を見ながら理解していきましょう。

▶ スペースの都合上、これら二つのメンバ関数以外は省略しています。実行プログラムを作成する際は、"Account.cpp" と "func.cpp" と "main.cpp" の 3 個のソースファイルをコンパイルして得られる 3 個のオブジェクトファイルをリンクします。

▪ クラス定義の中で（ヘッダ部で）定義されたメンバ関数 *balance*

ヘッダ "Account.h" をインクルードする、すべてのソースファイルに関数定義が取り込まれます。そのため、"func.cpp" と "main.cpp" の両方に、メンバ関数 *balance* の定義が埋め込まれます。すなわち、定義は 2 個です。

▶ この図では、関数 *balance* をインラインに展開されていない状態で示しています。"func.cpp" で呼び出されている 1 の *balance* は、"func.cpp" に埋め込まれた *balance* であり、"main.cpp" で呼び出されている 3 の *balance* は、"main.cpp" に埋め込まれた *balance* です。

インラインで内部結合をもつため、メンバ関数の識別子は、ソースファイルに特有のものです（他のソースファイルから見えないように隠されています）。

そのため、3 個のソースファイルをコンパイルしたオブジェクトファイルをリンクする際に、識別子重複のリンク時エラーが発生することはありません。

▪ クラス定義の外で（ソース部で）定義されたメンバ関数 *deposit*

このメンバ関数は、ソース部 "Account.cpp" で定義されており、定義は 1 個のみです。その識別子は外部結合をもちますので、他のソースファイルから呼び出せる状態です。

▶ ソースプログラム "func.cpp" と "main.cpp" から呼び出されている 2 と 4 の *deposit* は、"Account.cpp" で定義された関数 *deposit* です。

関数の実体が 1 個だけですから、三つのソースファイルをコンパイルしたオブジェクトファイルのリンク時に、識別子重複のリンク時エラーが発生することはありません。

※スペースの都合上、メンバ関数 balance と deposit 以外は省略しています。

```
// Account.h … クラスAccountのヘッダ部
class Account {
public:
    long balance() { return crnt_balance; }
    void deposit(long amnt);
};
```

内部結合かつインライン

このヘッダをインクルードするすべてのソースファイルに関数定義が埋め込まれる。

```
// Account.cpp … クラスAccountのソース部
#include "Account.h"

void Account::deposit(long amnt)
{
    crnt_balance += amnt;
}
```

外部結合かつ非インライン

関数の定義は 1 個のみ。他のソースプログラムから呼び出せる。

Account.cpp で定義されたメンバ関数 deposit の呼出し

インクルードの結果、同一名の関数の定義が複数のソースファイルに埋め込まれる。
その識別子は内部結合をもつため、ソースファイル内でのみ通用する。
定義は 2 個だが、識別子重複のリンク時エラーが発生することはない。

```
// func.cpp
#include "Account.h"

インクルードの結果 埋め込まれる、内部結合かつインラインである関数 balance の定義
inline long Account::balance() { return crnt_balance; }

void func()
{
    Account x("Mr.X", "99999999", 100);
    long b = x.balance(); 1
    x.deposit(100);
}   2
```

```
// main.cpp
#include "Account.h"

インクルードの結果 埋め込まれる、内部結合かつインラインである関数 balance の定義
inline long Account::balance() { return crnt_balance; }
void func();
int main()
{
    func();
    Account y("Mr.Y", "88888888", 300);
    long c = y.balance(); 3
    y.deposit(100);
}   4
```

Fig.10-12　クラス定義の内外で定義されたメンバ関数

ヘッダとメンバ関数

クラス *Account* 第４版では、三つのメンバ関数 *name*、*no*、*balance* の関数定義が、クラス定義の中にあります。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（それなりに）見えてしまう状態で利用者に提供せざるを得ない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》のみを提供するような言語仕様とすることもできるでしょう。

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**》が低下します。C++ のクラスは、“C 言語と同程度（あるいは、それ以上）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、次のような感じなのではないでしょうか。

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！

なお、ヘッダ部で提供するクラス定義に適切なコメントを記入しておけば、単なるプログラムではなく、立派なドキュメントにもなります。

重要 外部との窓口であり、ブラックボックスでもあるクラス定義は、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。

–> 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 10-7** に示します。

▶ `"Account.cpp"` と `"AccountTest.cpp"` の両方をそれぞれコンパイルした上で、リンクする必要があります（p.363）。

これまでのプログラムとは異なり、口座の情報を表示する処理を、独立した関数 *print_Account* として実現しています。この関数は、文字列の *title* と、*Account* へのポインタ型の *p* を仮引数として受け取って、文字列 *title* と、*p* が指すクラス *Account* 型オブジェクトの口座情報である口座名義・口座番号・預金残高を表示します。

さて、ポインタ *p* が指すオブジェクトを **p* と表せることは、第７章で学習しました。そのため、*p* が指すクラスオブジェクト **p* のメンバ *m* を表す式は以下のようになります。

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

同じ

▶ この式から **p* を囲む ( ) を削除することはできません。アドレス演算子 * よりドット演算子 . の優先度のほうが高いからです。

List 10-7 Account04/AccountTest.cpp

```
// 銀行口座クラス（第４版）の利用例

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

using namespace std;

//--- pが指すAccountの口座情報（口座名義・口座番号・預金残高）を表示 ---//
void print_Account(string title, Account* p)
{
    cout << title
         << p->name() << "\" (" << p->no() << ") " << p->balance() << "円\n";
}

int main()
{
    Account suzuki("鈴木龍一", "12345678", 1000);        // 鈴木君の口座
    Account takeda("武田浩文", "87654321",  200);        // 武田君の口座

    suzuki.withdraw(200);              // 鈴木君が200円おろす
    takeda.deposit(100);               // 武田君が100円預ける

    print_Account("■鈴木君の口座：", &suzuki);

    print_Account("■武田君の口座：", &takeda);
}
```

実行結果

```
■鈴木君の口座："鈴木龍一"（12345678）800円
■武田君の口座："武田浩文"（87654321）300円
```

ただし、この式は煩雑ですから、以下の形式で表記できるようになっています。

```
p->m                                   // pが指すオブジェクト*pのメンバm
```

アロー演算子（*arrow operator*）と呼ばれる **->演算子**（-> *operator*）は、ドット演算子と同様に、クラスオブジェクトのメンバをアクセスする演算子です。**Table 10-2** に示すように、x->y は、(*x).y と同等です。

Table 10-2　クラス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名称は、-> の形状が**矢印**（*arrow*）に似ていることに由来します。

関数 *print_Account* では、アロー演算子 -> を利用して、メンバ関数 *name*、*no*、*balance* を呼び出しています。

重要 ポインタ p が指すオブジェクトのメンバ m である (*p).m は、**アロー演算子 ->** を利用した式 p->m でアクセスできる。

▶ ここでは、関数 *print_Account* の第２引数を《**ポインタの値渡し**》によって実現しています。よりよい方法は、《**const 参照渡し**》で実現することです。その詳細は、第 12 章で学習します。

自動車クラス

クラス定義の中で、すべてのメンバ関数を定義すれば、**そのクラスはヘッダ部だけで提供できます**。そのことを、自動車クラス *Car* を作りながら学習しましょう。

ここでは、自動車クラス *Car* には、以下に示す 7 個のデータを、データメンバとしてもたせることにします（**Fig.10-13**）。

- 名前
- 幅
- 長さ
- 高さ
- 現在位置のＸ座標
- 現在位置のＹ座標
- 残り燃料

右側の三つは、自動車の“移動”に必要なデータです。Ｘ座標とＹ座標は、図 b の平面上の“どこに”自動車が位置しているのかを表します。もちろん、移動に伴って燃料は減ります。そこで、燃料が残っているあいだだけ移動できるものとします。

また、すべてのデータメンバは外部からアクセスできないように《非公開》とします。そのため、たとえば燃料が盗まれて 0 になるということは、なくなります。

a 自動車のデータ　　b 座標

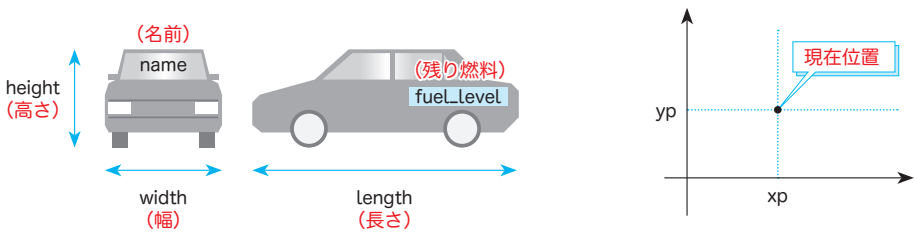

Fig.10-13　自動車クラスのデータ

▶　幅・長さ・高さの単位は mm とし、残り燃料の単位はℓ（リットル）とします。また、座標の単位は km とします。

クラスには、データメンバだけでなく、コンストラクタとメンバ関数が必要です。それらの概要は、以下のようにします。

- コンストラクタ

現在位置の座標を、原点（0.0, 0.0）にセットします。なお、座標以外のデータメンバには、仮引数に受け取った値を設定します。

■ メンバ関数

以下のメンバ関数を作ります。

- 現在位置のX座標を調べる。
- 現在位置のY座標を調べる。
- 残り燃料を調べる。
- 車のスペックを表示する。
- 自動車を移動する。

List 10-8 に示すのが、以上の設計に基づいて作成した自動車クラスです。

List 10-8 Car01/Car.h

```
// 自動車クラス

#include <cmath>
#include <string>
#include <iostream>

class Car {
    std::string name;            // 名前
    int width, length, height;   // 車幅・車長・車高
    double xp, yp;               // 現在位置座標
    double fuel_level;           // 残り燃料

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

    double x() { return xp; }                // 現在位置のX座標を返す
    double y() { return yp; }                // 現在位置のY座標を返す

    double fuel() { return fuel_level; }     // 残り燃料を返す

    void print_spec() {                      // スペック表示
        std::cout << "名前：" << name   << "\n";
        std::cout << "車幅：" << width  << "mm\n";
        std::cout << "車長：" << length << "mm\n";
        std::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_level)
            return false;               // 燃料不足
        else {
            fuel_level -= dist;         // 移動距離の分だけ燃料が減る
            xp += dx;
            yp += dy;
            return true;
        }
    }
};
```

ヘッダ部のみで実現しているため、**すべてのメンバ関数は、内部結合をもつインライン関数となります。**

▶ クラス *string* 型を std::*string* で表している点は、クラス *Account* 第4版と同じです。それと同じ理由によって、cout も std::cout で表しています。

各メンバ関数を理解していきましょう。

▪ コンストラクタ

座標以外の５個のデータを受け取って各メンバにセットします。*xp* と *yp* の値を 0.0 とすることで、自動車の位置を原点（0.0，0.0）に設定します。

▪ メンバ関数 *x*，*y*，*fuel*

これらのメンバ関数は、X 座標 *xp* の値、Y 座標 *yp* の値、残り燃料 *fuel_level* の値をそのまま返します。

▶ すなわち、データメンバ *xp*，*yp*，*fuel_level* のゲッタです。

▪ メンバ関数 *print_spec*

車のスペック（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.10-14** に示す計算によって求めます。

▶ **<cmath> ヘッダ**で関数宣言が提供される ***sqrt* 関数**は、引数に与えられた **double** 型実数値の**平方根**を求めて **double** 型の値として返却します。関数の形式は以下のとおりです。

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

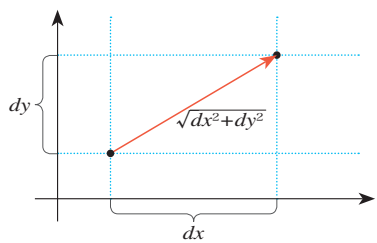

Fig.10-14　移動における座標と距離

なお、燃費は 1 としています。すなわち距離 1 の移動に必要な燃料は 1 です。

残り燃料 *fuel_level* が移動距離 *dist* に満たなければ、移動は不可能であるため、**false** を返します。また、移動に必要な燃料がある場合は、現在位置 *xp*，*yp* と残り燃料 *fuel_level* を更新した上で **true** を返します。

＊

自動車クラスを利用するプログラム例を **List 10-9** に示します。

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 *print_spec* を呼び出してスペックを表示します。その後、車の移動を対話的に繰り返します。

List 10-9 Car01/CarTest.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.print_spec();      // 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：5220
車高は：1490
ガソリン量は：90
名前：僕の愛車
車幅：1885mm
車長：5220mm
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
```

▶ ヘッダ部のみで実現されている車クラス *Car* は、インクルードするだけで利用できます。クラス定義が **#include** 指令によって自動的に取り込まれるため、**List 10-9** の "CarTest.cpp" が、クラス *Car* の定義を含むことになるからです。そのため、"CarTest.cpp" をコンパイル・リンクするだけで、実行ファイルが生成されます。

一方、ヘッダ部とソース部とに分けられて実現されたクラスを利用するプログラムの実行ファイルを作る際は、**クラスを利用するプログラムと、ソース部の両方のコンパイルが必要です。**

たとえば、クラス *Account* 第４版を利用する **List 10-7** の "AccountTest.cpp" の実行ファイルを作るには、そのプログラムと、ソース部である **List 10-6** の "Account.cpp" の両方をコンパイルした上で、その結果作られる二つのオブジェクトファイルをリンクします。

クラス名を *ABC* として、一般的にまとめると、以下のようになります。

▪ クラス ABC がヘッダ部 "ABC.h" のみで実現されているとき

クラス *ABC* を利用するプログラム "test.cpp" では、"ABC.h" をインクルードする。実行ファイルを作る際は、"test.cpp" のみをコンパイルする。

▪ クラス ABC がヘッダ部 "ABC.h" とソース部 "ABC.cpp" とで実現されているとき

クラス *ABC* を利用するプログラム "test.cpp" では、"ABC.h" をインクルードする。実行ファイルを作る際は、"test.cpp" と "ABC.cpp" の両方をコンパイルした上でリンクする。

次章以降も同様です。御自身で適宜判断して、コンパイル・リンクの作業を行いましょう。