第10講 クラス入門

10.1 クラスの基本

本節では、オブジェクト指向プログラミングの中核概念である「クラス（Class）」について紹介します。クラスは、C++で独自のデータ型を作成するための基本的な構成要素であり、データとそのデータを操作する関数を一つにまとめます。

---

1. クラスの基本的な考え方

ばらばらのデータの扱いから始める

プログラミングでは、関連する一連のデータを扱うことがよくあります。例えば、銀行口座を管理するには、口座名義、口座番号、預金残高が必要です。最初は、これらのデータを個別の変数で表現するかもしれません。

List 10-1のように、鈴木さんと武田さんの銀行口座に対してそれぞれ変数を定義します。

```C/C++
// List 10-1: 個別の変数で銀行口座を扱う
#include <string>
#include <iostream>

using namespace std;

int main()
{
	string suzuki_name    = "鈴木龍一";             // 鈴木君の名義氏名
	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";             // 武田君の名義氏名
	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";
}
```

データ処理の限界

この方法には問題があります。変数間の関連性がプログラムの構造上で表現されていないことです。suzuki_name、suzuki_number、suzuki_balanceは同じ口座に属しますが、この関係は命名規則によって暗示されているに過ぎません。プログラマが誤ってsuzuki_nameとtakeda_numberを関連付けてしまう可能性があり、コンパイラはそのような論理エラーを発見できません。

データを「オブジェクト」として統合する

この問題を解決するために、関連するデータを一つにまとめる構造が必要です。これがクラスの基本的な考え方です。クラスを使うと、現実世界の「オブジェクト」（例えば銀行口座）とその属性（口座名義、口座番号、残高）を一つのまとまりとしてモデル化できます。

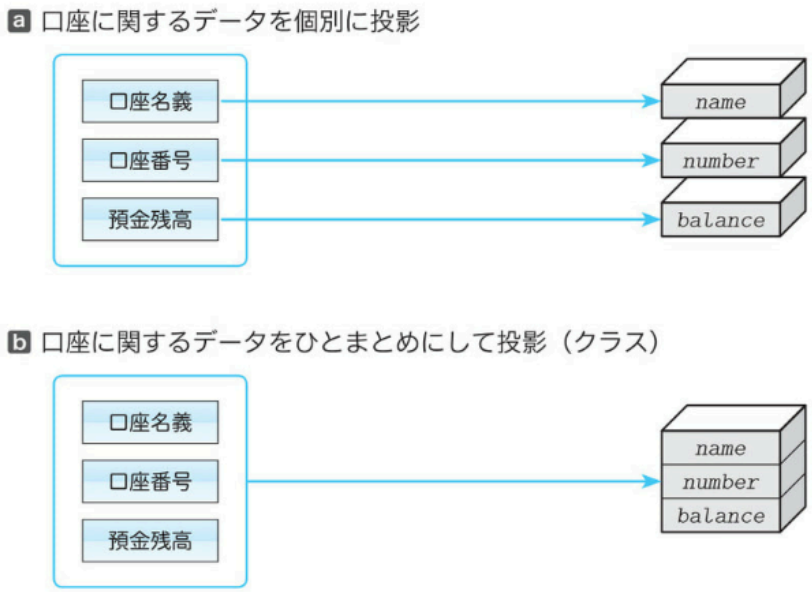


Fig.10-1 オブジェクトの投影とクラス

Fig. 10-1が示すように、ばらばらのデータ(a)を、構造化された単一のエンティティ(b)に統合できます。このエンティティがオブジェクトであり、このオブジェクトを作成するための設計図がクラスです。

2. クラスの定義と使用

List 10-2は、classキーワードを使って銀行口座をクラスとして書き直したものです。

C/C++

```
// List 10-2: 銀行口座クラス（第1版）とその使用
#include <string>
#include <iostream>

using namespace std;

// 銀行口座クラスの定義
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;  // Account classのオブジェクト suzuki(鈴木君の口座)を生成
    Account takeda;  // Account classのオブジェクト takeda(武田君の口座)を生成

    // suzukiオブジェクトのデータを設定
    suzuki.name = "鈴木龍一";
    suzuki.number = "12345678";
    suzuki.balance = 1000;

    // takedaオブジェクトのデータを設定
    takeda.name = "武田浩文";
    takeda.number = "87654321";
    takeda.balance = 200;

    // 入出金操作
    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 << "■鈴木さんの口座：\"" << suzuki.name << "\" (" << suzuki.number << ")
" << suzuki.balance << "円\n";
    cout << "■武田さんの口座：\"" << takeda.name << "\" (" << takeda.number << ")
" << takeda.balance << "円\n";
}
```

classキーワード：クラスを定義する

`class Account { ... };` の部分がクラス定義（**class definition**）です。これは設計図のように、**Account**型のオブジェクトがどのようなデータを含むべきかを記述します。定義の末尾のセミコロン**(;)**は必須です。

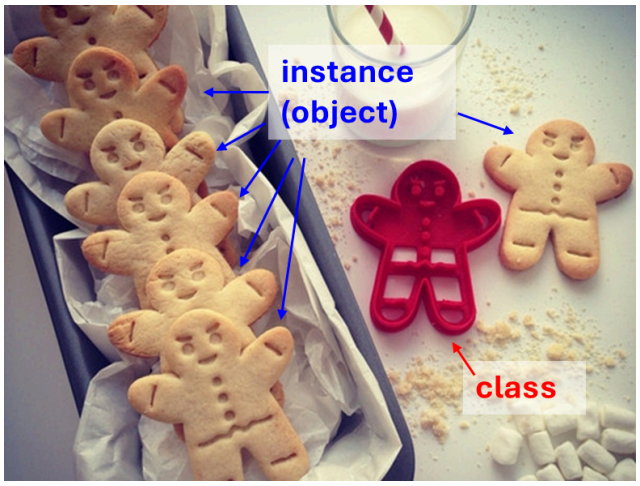
データメンバ

クラス定義の内部で宣言された変数（`name`、`number`、`balance`など）はデータメンバ（**data members**）と呼ばれます。これらがそのクラスのオブジェクトの状態を構成します。

オブジェクト：クラスのインスタンス

`main`関数の中で、`Account suzuki;`という行は`Account`クラスのオブジェクト（**object**）、またはインスタンス（**instance**）を生成します。これは`int`型の変数を宣言するのと同じです。

クラスをたこ焼きの型、オブジェクトをその型から作られた個々のたこ焼きに例えることができます。



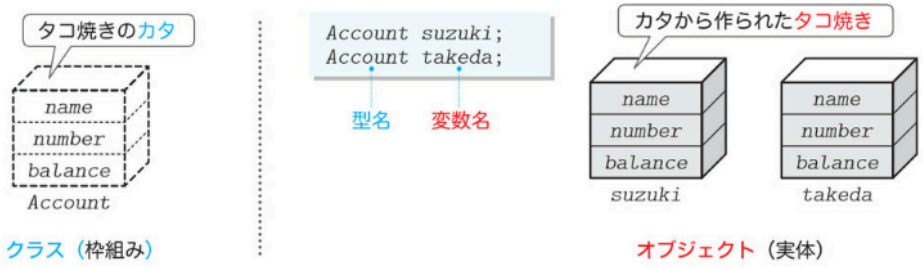


Fig.10-3　クラスとオブジェクト

メンバへのアクセス:ドット演算子(`.`)

オブジェクトのメンバにアクセスするには、ドット演算子(`.`)、別名クラスメンバアクセス演算子を使用します。例えば、`suzuki.name`は`suzuki`オブジェクトの`name`メンバにアクセスすることを意味します。

ユーザー定義型

`int`や`double`のような型は言語に組み込まれています。`class`を使うことで、私たちは`Account`というユーザー定義型(**user-defined type**)を作成したことになります。

---

3. クラスの改良:カプセル化とデータ隠蔽

`List 10-2`の構造は`List 10-1`よりも優れていますが、まだ2つの主要な問題が残っています。

データメンバへの直接アクセスの問題点

問題1:初期化が保証されない

`suzuki`オブジェクトを生成した際、そのメンバは自動的に初期化されません。後のコードで手動で値を代入する必要があります。もし初期化を忘れると、メンバは不定な値を持ち、プログラムのエラーを引き起こす可能性があります。

問題2:データが保護されていない

`Account`クラスのすべてのデータメンバは公開(**public**)されており、どのコードからでも
`suzuki.balance = 0;`
のように自由に書き換えることができます。これは現実世界では非論理的で非常に危険です。銀行口座の残高は外部から直接改ざんされるべきではなく、明確な「預け入れ」や「引き出し」といった操作を通じて変更されるべきです。
多くの場合、関連データには特定の業務ロジックが伴います。外部からの直接的なデータ改変を許してしまうと、ロジックの混乱やデータの不整合を招く危険性があります。そのため、データを隠蔽しつつ、必要なインターフェースを提供することで、機能性と正確なデータロジックの両立を図るべきです。

`private`と`public`アクセス指定子

これらの問題を解決するために、C++はアクセス指定子(**access specifiers**)、主に`public`と`private`を導入しました。

- **`public`**: `public`で修飾されたメンバーは、クラスの外部からアクセスできます。

- **`private`**: `private`で修飾されたメンバーは、クラスの内部（つまりクラスのメンバ関数の中）からのみアクセスできます。

データ隠蔽はオブジェクト指向プログラミングの核心的な原則の一つで、クラスのデータメンバは**`private`**として宣言すべきであるという考え方です。これにより、外部からの直接アクセスを防ぎ、代わりに一連の`public`なメンバ関数を外部との唯一のインターフェースとして提供します。

---

4. コンストラクタ

「初期化が保証されない」問題を解決するために、C++はコンストラクタ（**Constructor**）を提供します。

コンストラクタは、オブジェクトが生成される際に自動的に呼び出される特殊なメンバ関数です。その名前はクラス名と完全に同じで、戻り値の型はありません。

```C/C++
class Account {
private:
    string full_name;
    string number;
    long crnt_balance;

public:
    // Constructor
    Account(string name, string num, long amnt) {
        full_name = name;
        number = num;
        crnt_balance = amnt;
    }
    // ... 他のメンバ関数
};
```

このコンストラクタがあれば、オブジェクトの生成と同時に初期化を一度に行うことができます。

```C/C++
// オブジェクト生成時に、直接初期値を提供する
Account suzuki("鈴木龍一", "12345678", 1000);
```

これにより、`Account`オブジェクトは生成された瞬間から、有効で既知の状態にあることが保証されます。

5. メンバ関数

「データが保護されていない」問題を解決するために、データを`private`にし、データへのアクセスを制御するための`public`なメンバ関数（**member functions**）を提供します。

`List 10-3`は、改良された最終版です。

```C/C++
// List 10-3: 銀行口座クラス（第2版）とその使用

#include <string>
#include <iostream>

using namespace std;

class Account {
private:
	string full_name;		// 名義氏名
	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;		// 名義氏名
		number = num;			// 口座番号
		crnt_balance = amnt;		// 預金残高
	}

	//--- 名義氏名を調べる ---//
	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";
}
```

このバージョンでは：

- データメンバ`full_name`、`number`、`crnt_balance`は`private`であり、`main`関数から直接アクセスすることはできません（例えば`suzuki.crnt_balance = 0;`はコンパイルエラーになります）。
- ゲッター（**Getter**）関数である`name()`や`balance()`は、データへの読み取り専用アクセスを提供します。
- セッター（**Setter**）関数（ここではより意味の明確なミューテータ）である`deposit()`や`withdraw()`は、制御された方法でデータを変更する手段を提供します。

このようにデータとそれを操作する関数を一つにまとめ、実装の詳細を外部から隠蔽する手法をカプセル化（**Encapsulation**）と呼びます。これはオブジェクト指向プログラミングの基礎であり、コードの安全性、保守性、明確さを大幅に向上させます。

---

6. コンストラクタの深い理解

コンストラクタ（Constructor）の責務は、オブジェクトを確実かつ正しく初期化することです。

コンストラクタの呼び出し

コンストラクタはオブジェクトが生成されるときに呼び出されます。以下の宣言文は、オブジェクトを生成する際に`Account`クラスのコンストラクタを呼び出します。

```C/C++
Account suzuki("鈴木龍一", "12345678", 1000);
Account takeda("武田浩文", "87654321", 200);
```

この構文は、`int x(5);`のように、通常の変数を括弧で初期化する形式に似ています。

コンストラクタは特定のオブジェクトに属する

概念的には、コンストラクタが呼び出されるたびに、それは生成中のそのオブジェクト専用のものとなります。

- `suzuki`オブジェクトが生成されるとき、コンストラクタは`suzuki`のデータメンバを操作します。
- `takeda`オブジェクトが生成されるとき、コンストラクタは`takeda`のデータメンバを操作します。

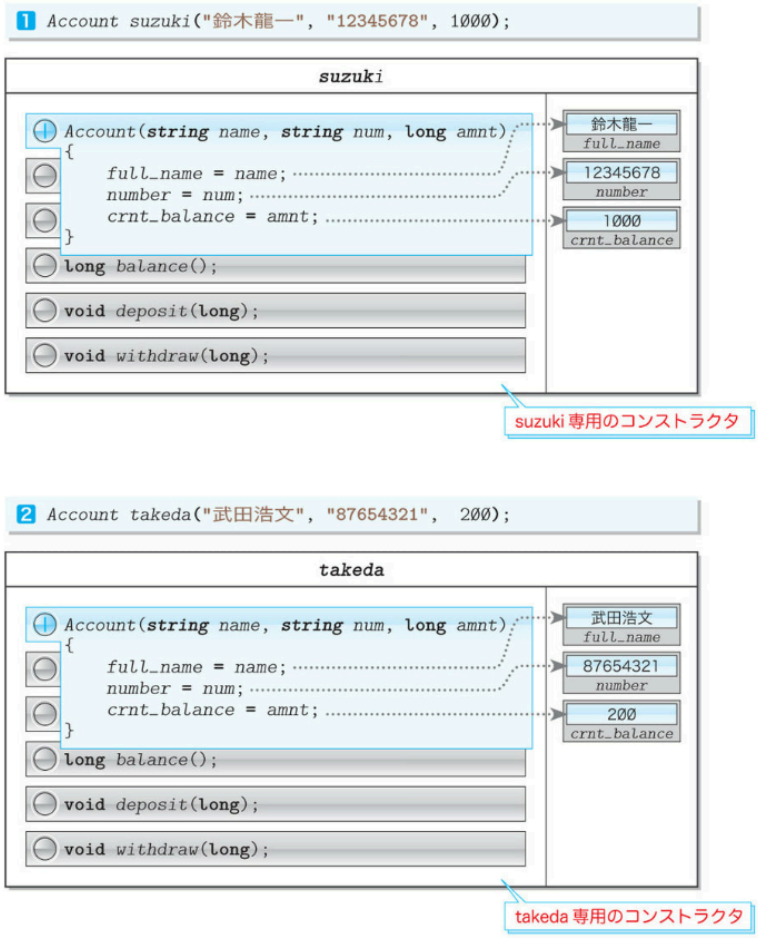


Fig.10-7 オブジェクトとコンストラクタ

コード上には一つのコンストラクタ定義しかありませんが、概念的には、各オブジェクトが自身を初期化するための「専用」コンストラクタを持っていると考えることができます。

強制的な初期化の重要性

引数を必要とするコンストラクタをクラスに定義すると、コンパイラはオブジェクト生成時にそれらの引数を提供するよう強制します。引数が一致しない試みはすべてコンパイルエラーになります。

```C/C++
Account my_account; // コンパイルエラー：引数が提供されていない
Account another("武田"); // コンパイルエラー：引数の数が足りない
```

これこそがコンストラクタの強力な点です。オブジェクトが生まれた瞬間から完全で有効な状態にあることを保証し、不完全または誤った初期化を防ぎます。

デフォルトコンストラクタ

`List 10-2`（第1版の`Account`クラス）を振り返ると、私たちはコンストラクタを一切定義していませんでしたが、`Account suzuki;`のようなコードは正常にコンパイルされました。これはなぜでしょうか？

C++には次のようなルールがあるからです：クラスにコンストラクタを一つも定義しなかった場合、コンパイラは自動的に、引数がなく、中身が空の**public**なコンストラクタを生成する。このコンパイラによって生成されるコンストラクタを**デフォルトコンストラクタ（default constructor）**と呼びます。

第1版の`Account`クラスに対して、コンパイラは実質的に以下を追加していました：

```C/C++
public:
    Account() { } // コンパイラによって自動生成されたデフォルトコンストラクタ
```

重要：一度でもクラスに何らかのコンストラクタ（例えば第2版の`Account(string, string, long)`）を定義すると、コンパイラはもはやデフォルトコンストラクタを自動生成しません。この時点で、もし`Account my_account;`のような構文でオブジェクトを生成したい場合は、引数なしのコンストラクタを自分で明示的に提供する必要があります。

---

7. メンバ関数と「メッセージパッシング」

メンバ関数も特定のオブジェクトに属する

コンストラクタと同様に、メンバ関数も概念的には特定のオブジェクトに属します。`suzuki.withdraw(200);`を呼び出すと、`withdraw`関数は`suzuki`オブジェクトの`crnt_balance`を操作します。メンバ関数の内部では、オブジェクト名とドットを使わずに、そのオブジェクトの他のメンバー（プライベートメンバーを含む）に直接アクセスできます。

オブジェクトに「メッセージを送る」

オブジェクト指向プログラミングでは、メンバ関数の呼び出しはオブジェクトに「メッセージを送る」ことに例えられます。

`suzuki.balance()`というコードは、「`suzuki`オブジェクトよ、あなたの残高を教えてください」というメッセージを送っていると解釈できます。オブジェクトはメッセージを受け取ると、対応するメンバ関数（`balance`）を実行し、結果を返します。

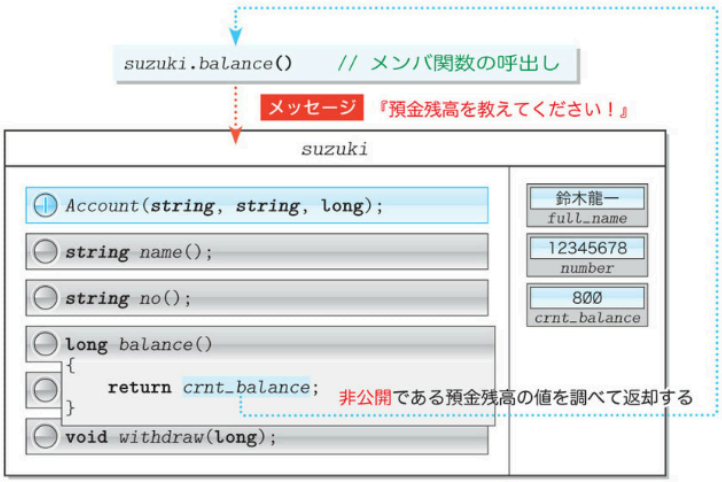


Fig.10-8　メンバ関数の呼出しとメッセージ

インラインメンバ関数によるパフォーマンス最適化

値を取得するためだけに関数を呼び出すこと（例：`balance()`）が、変数を直接アクセスする（`crnt_balance`）よりも効率が悪いのではないかと心配する人もいるかもしれません。

C++では、この問題は通常インライン関数（**inline function**）によって解決されます。デフォルトでは、クラス定義の内部で実装されたメンバ関数は自動的にインライン関数と見なされます。

これは、コンパイラが関数呼び出しを関数本体のコードそのものに置き換えようと試みることを意味し、それによって関数呼び出しのオーバーヘッドがなくなります。したがって、

```
long b = suzuki.balance();
```

はコンパイル後、

```
long b = suzuki.crnt_balance;
```

と同じ効果になる可能性が高く、安全性と高性能を両立します。

---

8. 命名規則とアクセサ

データメンバとメンバ関数の命名衝突

C++では、同じクラス内のデータメンバとメンバ関数は同じ名前を持つことができません。

これが、私たちの`Account`クラスで、口座名義を表すプライベートデータメンバが`full_name`であり、それを取得するパブリックメンバ関数が`name()`である理由です。

アクセサ：ゲッターとセッター

- ゲッター（**Getter**）：プライベートデータメンバの値を読み取るためのメンバ関数。例：`name()`、`balance()`。
- セッター（**Setter**）：プライベートデータメンバの値を書き込むためのメンバ関数。私たちの`Account`クラスには汎用的なセッターはなく、代わりに`deposit()`や`withdraw()`のような、意図がより明確なミューテータを提供しています。
- アクセサ（**Accessor**）：ゲッターとセッターの総称。

一般的な命名スタイル

データメンバとアクセサを区別するために、業界にはいくつかの一般的な命名スタイルがあります。

1. 異なる名前：データメンバとゲッターに全く異なる名前を使用する（例：データメンバ`crnt_balance`、ゲッター`balance()`）。これは私たちの例で採用したスタイルで、内部実装の詳細を効果的に隠蔽します。
2. 接尾辞/接頭辞：プライベートデータメンバに統一された接尾辞や接頭辞を付ける（例：`balance_`や`m_balance`）、そしてゲッターは接辞なしの名前（`balance()`）を使用する。
3. **`get_`**接頭辞：ゲッターに`get_`接頭辞を付ける（例：`get_balance()`）。このスタイルはJavaなどの言語で非常に一般的ですが、C++の標準ライブラリではあまり見られません。

---

9. クラスとオブジェクト

回路設計図の比喩

クラスとオブジェクトの関係を、生き生きとした比喩でまとめることができます。

- クラス（**Class**）：回路設計図のようなもの。回路の全構成（どの部品があるか）と機能（どのボタンがあるか）を定義します。
- インスタンス（**instance**）/ オブジェクト（**Object**）：その設計図に基づいて製造された実体の回路。各実体回路は、設計図に定義されたすべての部品と機能を持ちます。

State（状態）とBehavior（振る舞い）

- データメンバ：オブジェクトの**状態（State）**を表します。例えば、`crnt_balance`の値は口座オブジェクトの現在の状態の一つです。
- メンバ関数：オブジェクトの**振る舞い（Behavior）**を表します。これらは外部がオブジェクトと対話し、その状態を問い合わせたり変更したりするための「ボタン」です。例えば、`deposit`という振る舞いは`crnt_balance`の状態を変更します。

カプセル化を通じて、オブジェクトの状態を保護し、必要な振る舞いのインターフェースのみを公開します。これこそがオブジェクト指向設計の神髄です。

10.2 クラスの実装

本節では、C++におけるクラスと大規模プロジェクト開発の標準的な実践方法、すなわちクラスの宣言と実装をどのように分離し、それらを異なるファイルにどのように整理するかについて深く探ります。これは、モジュール化され、再利用可能で、保守しやすいコードを書くための鍵となります。

---

1. クラス定義の外部でのメンバ関数の定義

これまでの例では、すべてのメンバ関数の定義が`class`の波括弧内に直接書かれていました。大規模なクラスの場合、これはクラスの定義を肥大化させ、読みにくくします。

より明確で規範的な方法は、関数の宣言と定義を分離することです。

- 宣言（**Declaration**）：`class`定義の内部に残り、コンパイラに関数の存在、名前、引数、戻り値の型を伝えます。
- 定義（**Definition**）：関数の実際のコードブロックで、`class`定義の外部に移動できます。

スコープ解決演算子 `::`

メンバ関数の定義をクラスの外部に移動する場合、その関数がどのクラスに属しているかをコンパイラに明示的に伝える必要があります。これはスコープ解決演算子 `::` を使って行います。

構文: 戻り値の型 クラス名::関数名(引数リスト) { /* ...関数本体... */ }

`List 10-4`は、`Account`クラスの第3版を示しており、コンストラクタ、`deposit`、`withdraw`の定義がクラス宣言の外部に移動されています。

```
C/C++
// List 10-4: 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 full_name;          // 名義氏名
        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

        string name()  { return full_name; }                // 名義氏名を調べる
        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
        void withdraw(long amnt);                                   // おろす
};

//--- コンストラクタ ---//
Account::Account(string name, string num, long amnt)
{
        full_name = name;          // 名義氏名
        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;
}

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";
}
```

inlineキーワードの意味の変化

非常に重要な違いがあります：

- クラス定義の内部で実装されたメンバ関数は、コンパイラによって自動的に**inline**と見なされます。
- クラス定義の外部で実装されたメンバ関数は、デフォルトでは**inline**ではありません。

外部で定義された関数をインラインにしたい場合は、その定義の前に明示的にinlineキーワードを追加する必要があります。

---

2. ヘッダファイルとソースファイルの分離

宣言と定義を分離したら、次のステップは、C++の標準的な慣習に従って、それらを異なる種類のファイルに配置することです。

- ヘッダファイル（**Header File, .h**または**.hpp**）：クラスの定義（データメンバとメンバ関数の宣言を含む）を格納します。ヘッダファイルはクラスの「公開インターフェース」または「仕様書」です。
- ソースファイル（**Source File, .cpp**）：メンバ関数の定義（実装）を格納します。ソースファイルはクラスの内部実装の詳細です。

【重要】宣言（インターフェース）をヘッダファイルに、定義（実装）をソースファイルに分けることは、C++のソフトウェアエンジニアリングにおける中心的な慣習であり、主に以下の3つの技術的な理由に基づいています：

- コンパイル効率：ヘッダファイルには必要な宣言（クラス構造、関数プロトタイプ、メンバ変数）のみが含まれます。コンパイラは、あるクラスや関数がどのようなものかを知る必要はあっても、その内部がどのように機能するかを知る必要はありません。この分離はコンパイル依存関係を最小限に抑え、コンパイルにかかる時間を大幅に短縮します。
- 重複定義エラーの回避：非インライン（non-inline）関数の完全な定義（実装）をヘッダファイルに記述した場合、このヘッダファイルが2つ以上の `.cpp` ファイルに含まれると、コンパイラはそれをインクルードする各 `.cpp` ファイルに対してその関数のコードのコピーを生成します。リンク段階で、リンカは同じ関数（または変数）に対して複数の実装が存在することを発見し、致命的な「再定義」**(Redefinition)** エラーを報告します。
- カプセル化と情報隠蔽

- ヘッダファイルはインターフェースを公開する: ヘッダファイルは、クラスの公開契約（Public Contract）です。ユーザーが必要とする情報（公開メンバ関数とデータ構造）のみを公開します。
- ソースファイルは実装を隠蔽する: ソースファイル (`.cpp`) は、すべての実装の詳細を隠します。ユーザーはこれらの詳細を見ることも、変更することもできませんし、そうすべきでもありません。場合によっては、ソフトウェア会社はヘッダファイル（インターフェース）とプリコンパイルされたライブラリファイル（実装）のみを配布し、それによって中核となるアルゴリズムや知的財産が漏洩するのを防ぎます。

`List 10-5` と `List 10-6` は、`Account`クラスを `Account.h` と `Account.cpp` に分離する方法を示しています。

`Account.h` (List 10-5)

```
C/C++
// Account.h - 銀行口座クラスのヘッダファイル
#include <string>

class Account {
private:
    std::string full_name;
    std::string number;
    long crnt_balance;

public:
    Account(std::string name, std::string num, long amnt);

    std::string name() { return full_name; }
    std::string no() { return number; }
    long balance() { return crnt_balance; }

    void deposit(long amnt);
    void withdraw(long amnt);
};
```

`Account.cpp` (List 10-6)

```
C/C++
// Account.cpp - 銀行口座クラスのソースファイル
#include <string>
#include <iostream>
#include "Account.h" // 自身のヘッダファイルをインクルード
```

```cpp
using namespace std;

// コンストラクタの定義
Account::Account(string name, string num, long amnt) {
    full_name = name;
    number = num;
    crnt_balance = amnt;
}

// 他のメンバ関数の定義...
void Account::deposit(long amnt) {
    crnt_balance += amnt;
}

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

`Account`クラスを使用する必要があるファイル（例えば`main.cpp`）は、今や`#include "Account.h"`とするだけで済みます。

この分離の利点は次のとおりです：

1. カプセル化: ユーザーはヘッダファイルを見るだけでよく、内部実装を気にする必要がありません。
2. 効率: `.cpp`ファイルが変更された場合、そのファイルのみを再コンパイルすればよく、それを使用する他のファイルは再コンパイルする必要がありません。
3. 保守性: コードの構造がより明確になり、管理しやすくなります。

---

3. ヘッダファイル内での`using`指令：悪い実践

`Account.h`で`string`の代わりに`std::string`を使用したことにお気づきかもしれません。これは非常に重要なコーディング規約です。

重要：原則として、ヘッダファイル内で **`using namespace`** や **`using std::string;`** のような指令を使用してはなりません。

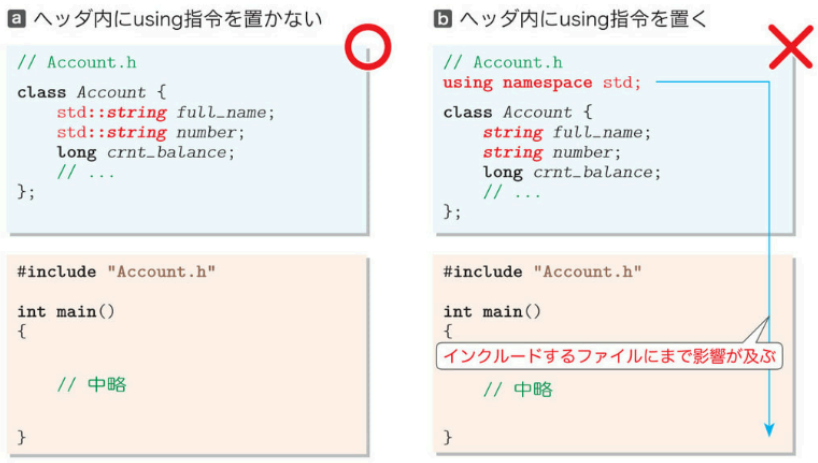


Fig. 10-11: ヘッダファイル内の*using*指令の問題

理由: `using`指令は、そのヘッダファイルをインクルードするすべてのソースファイルに「漏洩」します。あるプロジェクトがあなたの`Account.h`をインクルードすると、そのプロジェクトは`std`名前空間全体を強制的に取り込むことになり、自身のコードと名前の衝突を引き起こし、コンパイルエラーにつながる可能性があります。これはコードのカプセル化と独立性を破壊します。

正しいやり方:

- ヘッダファイル(`.h`)では、常に`std::string`のように完全修飾名を使用します。
- ソースファイル(`.cpp`)では、`using namespace std;`を安全に使用できます。そのスコープはそのファイルに限定され、他のファイルに影響を与えないからです。

---

4. メンバ関数のリンケージ

リンケージは、コンパイラとリンカが異なるファイルに現れる同名の関数をどのように扱うかを決定します。

- 内部リンケージ(**Internal Linkage**): 関数の実体は、それが存在するファイルに対して「プライベート」です。複数のファイルが同名の内部リンケージ関数を持つことができ、それらはそれぞれ独立したコピーであり、リンク時に衝突しません。
- 外部リンケージ(**External Linkage**): 関数の定義はプログラム全体で一意でなければなりません。複数のファイルが同名の外部リンケージ関数を定義した場合、リンカは「重複定義」エラーを報告します。

メンバ関数に関するルールは以下の通りです:

1. クラス定義の内部で実装されたメンバ関数（つまり暗黙的に**inline**なもの）は、内部リンケージを持ちます。

    - `Account::balance()`関数の定義は`Account.h`内にあります。`Account.h`をインクルードする各`.cpp`ファイルは、この関数のコピーを一つずつ取得します。これは内部リンケージであるため、リンクエラーを引き起こしません。

2. クラス定義の外部で実装されたメンバ関数（**inline**でないもの）は、外部リンケージを持ちます。

    - `Account::deposit()`関数の定義は`Account.cpp`内にあります。この定義はプログラム全体で一意です。`deposit()`を呼び出す他のすべての場所から、リンカはこの唯一の実装を指します。

これが、外部リンケージを持つ関数の定義を`.cpp`ファイルに置き、`.h`ファイルに置いてはならない理由です。もし`.h`ファイルに置くと、それをインクルードする各`.cpp`ファイルが定義を生成しようとし、結果としてリンクエラーが発生します。

使用する際は、一般的に以下のルールに従えば十分です：
- クラスの宣言と**inline** メンバ関数の実装は、ヘッダファイルに記述します。
- それ以外の関数の実装は、ソースファイルに記述します。

---

5. アロー演算子 `->`

ポインタを介してオブジェクトのメンバにアクセスする必要がある場合、C++は特別な演算子を提供します：アロー演算子（**->**）です。

ポインタを介したメンバアクセスの構文

`Account`オブジェクトを指すポインタ`p`があるとします。`p`が指すオブジェクトの`name()`メンバ関数にアクセスするには、`(*p).name()`と書くことができます。ここで、`*p`はまずポインタをデリファレンスして、それが指すオブジェクトを取得し、次に`.name()`でそのオブジェクトのメンバにアクセスします。

アロー演算子（`->`）の役割

この一般的な操作を簡略化するために、C++はアロー演算子`->`を提供します。式`p->name()`は`(*p).name()`と等価です。

`List 10-7`は、`Account`オブジェクトへのポインタを引数として受け取り、アロー演算子を使ってそのメンバにアクセスする`print_Account`関数を示しています。

Main.cpp (List 10-7)

```C/C++
// List 10-7: Accountオブジェクトへのポインタを使用する
#include <string>
#include <iostream>
#include "Account.h" // Account.hがクラス定義を含むと仮定

using namespace std;

// 関数：Accountの情報を表示（Accountオブジェクトへのポインタを受け取る）
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);
    Account takeda("武田浩文", "87654321", 200);

    suzuki.withdraw(200);
    takeda.deposit(100);

    print_Account("■鈴木さんの口座", &suzuki); // suzukiオブジェクトのアドレスを渡す
    print_Account("■武田さんの口座", &takeda);  // takedaオブジェクトのアドレスを渡す
}
```

---

6. 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しています。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。

```C/C++
// List 10C-1: stringクラスの利用例

#include <string>
#include <cstring>
#include <iostream>

using namespace std;

int main()
{
	string s1 = "ABC";
	string s2 = "HIJKLMN";
	digit s = "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';
}
```

---

7. Carクラスの例

クラスの設計と実装をさらに理解するために、より複雑な例であるCarクラスを見てみましょう。このクラスは自動車をシミュレートし、その属性（データメンバ）と振る舞い（メンバ関数）を含みます。

Carクラスのデータメンバ

Carクラスは、自動車を記述するために以下のデータを含みます：

- `name` (string): 車名
- `width`, `length`, `height` (int): 車の寸法
- `xp`, `yp` (double): 車の現在のX, Y座標（位置）
- `fuel_level` (double): 残り燃料量

Carクラスのメンバ関数

Carクラスは以下のメンバ関数を提供します：

- コンストラクタ: `Car(string n, int w, int l, int h, double f)`。車の各属性を初期化し、初期位置を(0.0, 0.0)に設定します。
- ゲッター関数: `x()`, `y()`, `fuel()`。それぞれ車の現在のX座標、Y座標、残り燃料を返します。
- **`print_spec()`**: 車の詳細なスペック（名前、寸法）を表示します。
- **`move(double dx, double dy)`**: 車の移動をシミュレートします。移動距離を計算し、燃料が十分か確認します。燃料が不足している場合は`false`を返し、そうでなければ車の位置と燃料量を更新して`true`を返します。

`List 10-8`は、Carクラスの完全な定義を示しており、すべてのメンバ関数はクラス内部で実装されています。

```C/C++
// List 10-8: Carクラス（すべてのメンバ関数をヘッダファイルで実装）
#include <cmath> // std::sqrtのために
#include <string>
#include <iostream>

class Car {
    std::string name;
    int width, length, height;
    double xp, yp; // 現在位置
    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; // 初期位置を原点に設定
    }

    // ゲッター関数
    double x() { return xp; }
    double y() { return yp; }
    double fuel() { return fuel_level; }

    // スペックを表示
    void print_spec() {
        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";
    }

    // 車を移動
    bool move(double dx, double dy) {
        double dist = std::sqrt(dx * dx + dy * dy); // 移動距離を計算

        if (dist > fuel_level) { // 燃料不足
            return false;
        } else {
            fuel_level -= dist;
            xp += dx;
            yp += dy;
            return true;
        }
    }
};
```

すべてのメンバ関数をクラス内部で定義（暗黙の`inline`）

この`Car`クラスの実装では、すべてのメンバ関数（コンストラクタ、ゲッター、`move`を含む）がクラス定義の内部で直接実装されています。これは、それらがすべてコンパイラによって暗黙的に**`inline`**関数と見なされることを意味します。

Carクラスの使用例

List 10-9は、Carクラスを使用した完全なプログラムを示しており、ユーザーが車のデータを入力し、その移動をシミュレートできます。

C/C++

```
// List 10-9: Carクラスの使用例
#include <iostream>
#include "Car.h" // Carクラスの定義をインクルード

using namespace std;

int main() {
    string name;
    int width, length, height;
    double gas;

    cout << "--- 車のデータを入力してください ---\n";
    cout << "名前: "; cin >> name;
    cout << "車幅: "; cin >> width;
    cout << "車長: "; cin >> length;
    cout << "車高: "; cin >> height;
    cout << "ガソリン量(L): "; cin >> gas;

    Car myCar(name, width, length, height, gas); // Carオブジェクトを生成

    myCar.print_spec(); // 車のスペックを表示

    while (true) {
        cout << "現在位置: (" << myCar.x() << ", " << myCar.y() << ")\n";
        cout << "残り燃料: " << myCar.fuel() << " L\n";
        cout << "移動しますか？ [0...いいえ/1...はい]: ";
        int move_choice;
        cin >> move_choice;

        if (move_choice == 0) break; // 移動しない場合はループを抜ける

        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)) { // 移動を試みる
            cout << "燃料が足りません！\n";
        }
    }
}
```

このプログラムは、対話的な方法で、オブジェクトの生成、メンバ関数を呼び出してオブジェクトの状態を問い合わせたり、その振る舞いを変更したりする方法をユーザーに体験させます。