第11講 クラスの作成

11.1 日付クラスの作成

本節では、西暦年・月・日の三つのデータで構成される日付クラスを作成しながら、クラスに対する理解を深めていきます。

---

1. Dateクラス：標準的な実践

年、月、日の情報を格納するためのDateクラスを作成します。

クラスの設計

- データメンバ: privateなint型のメンバ y, m, d。日付データをカプセル化し、保護するために使用します。
- コンストラクタ: publicなコンストラクタDate(int yy, int mm, int dd)。オブジェクト作成時に完全な日付を提供することを保証し、オブジェクトの有効性を確保します。
- ゲッター関数: publicなyear(), month(), day()関数。日付データへの読み取り専用アクセスを提供します。

クラスの宣言と実装を別々のファイルに分離します。
Date.h (List 11-1)

```C/C++
// Date.h: Date クラスのヘッダファイル
class Date {
    int y; // 年
    int m; // 月
    int d; // 日
public:
    // コンストラクタ (宣言)
    Date(int yy, int mm, int dd);

    // ゲッタ (クラス定義内で定義、暗黙的インライン)
    int year() { return y; }
    int month() { return m; }
    int day() { return d; }
};
```

Date.cpp (List 11-2)

```C/C++
// Date.cpp: Date クラスのソースファイル
#include "Date.h"

// コンストラクタ（定義）
Date::Date(int yy, int mm, int dd) {
    y = yy;
    m = mm;
    d = dd;
}
```

この構造はインターフェースと実装を明確に分離し、C++プロジェクトの標準的な実践です。

オブジェクトの初期化

オブジェクトの初期化（生成時に初期値を与えること）には、いくつかの方法があります。`List 11-3`は、3つの一般的な初期化形式を示しています。

`main01.cpp` (List 11-3)

```C/C++
// Date オブジェクトの初期化
#include <iostream>
#include "Date.h"
using namespace std;

int main()
{
    Date a(2025, 11, 18); // 1. 直接初期化
    Date b = a;           // 2. コピー初期化
    Date c(a);            // 3. コピー初期化
}
```

```Shell
g++ main01.cpp Date.cpp -o main01; if($?) { .\main01 }
```

1. 直接初期化: `Date a(...)`は最も直接的な形式で、引数リストに一致するコンストラクタを呼び出します。
2. コピー初期化: `Date b = a;`は、既存の同型オブジェクト`a`を使用して新しいオブジェクト`b`を生成します。

コピー初期化では、C++のデフォルトの振る舞いはメンバごとのコピー **(Member-wise Copy)**です。下図のように、オブジェクト`a`のすべてのデータメンバ（`y`, `m`, `d`）の値が、新しいオブジェクト`b`の対応するメンバに一つずつコピーされます。

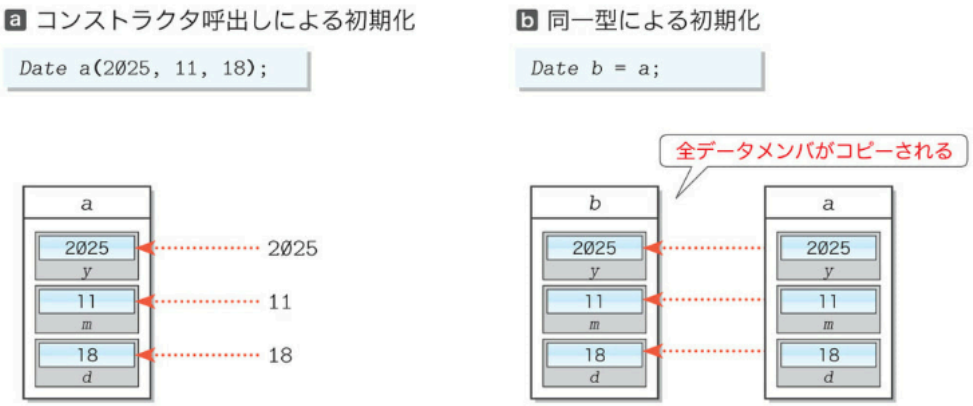

Fig. 11-2: コンストラクタとコピー初期化

---

2. コピーコンストラクタ (Copy Constructor)

「メンバごとのコピー」という振る舞いは、実際には特別なコンストラクタであるコピーコンストラクタによって実行されます。

コピーコンストラクタとは？

コピーコンストラクタは、ただ一つの引数（同型オブジェクトの（通常は`const`）参照）を受け取る特殊なコンストラクタです。その役割は、渡されたオブジェクトを使用して新しく作成されるオブジェクトを初期化することです。

もし自分でコピーコンストラクタを定義しない場合、コンパイラが自動的に一つ生成します。この暗黙的に生成されたコピーコンストラクタが、上記の「メンバごとのコピー」を実行します。

私たちの`Date`クラスに対して、コンパイラが生成するデフォルトコピーコンストラクタは以下と等価です：

```C/C++
// コンパイラによって Date クラスのために暗黙的に生成されたコピーコンストラクタ
public:
    Date(const Date& x) {
```

```
        y = x.y;
        m = x.m;
        d = x.d;
    }
```

したがって、`Date b = a;`というコードは、実際にはこのコンパイラが提供したコピーコンストラクタを呼び出しています。
コンパイラから見ると、これは `Date b(a);` と全く同じ意味になります。

カスタムコピーコンストラクタが必要な場合

`Date`のような単純なクラスでは、デフォルトのメンバごとのコピーはうまく機能します。しかし、クラスが外部リソース（動的に割り当てられたメモリ、ファイルハンドル、ネットワーク接続など）を管理している場合、デフォルトのコピーコンストラクタは問題を引き起こします（例えば、2つのオブジェクトが同じメモリブロックを指してしまい、二重解放を引き起こすなど）。このような場合、リソースのコピーを正しく処理するために、必ず自分でコピーコンストラクタを定義する必要があります。

---

3. オブジェクトの代入：コピー代入演算子

初期化と代入は全く異なる概念です。

- 初期化 **(Initialization)**: オブジェクトの作成時に初期値を与えること。
- 代入 **(Assignment)**: 既に存在するオブジェクトに新しい値を与えること。

`List 11-4`は代入操作を示しています：

```C/C++
// List 11-4: Date オブジェクトの代入
#include <iostream>
#include "Date.h"
using namespace std;

int main() {
    Date a(2025, 11, 18);
    Date b(1999, 12, 31); // b はここで初期化される
    Date c(1999, 12, 31); // c はここで初期化される

    b = a;                  // ここは代入であり、初期化ではない！
    c = Date(2000, 12, 31); // ここは代入であり、初期化ではない！
```

```
}
```

この代入操作は、もう一つの特別なメンバ関数であるコピー代入演算子 (**operator=**)によって実行されます。

コピーコンストラクタと同様に、もし自分で定義しなければ、コンパイラがデフォルトのコピー代入演算子を生成し、これもまたメンバごとのコピーを実行します。

同様に、外部リソースを管理する複雑なクラスでは、自分でコピー代入演算子を定義する必要があります。

---

4. オブジェクト間の比較：演算子のオーバーロード

オブジェクトが=で代入できるなら、==で比較できるのでしょうか？

```C/C++
if (b == c) { // コンパイルエラー！
    cout << "bとcは同じ日です。\n";
}
```

答えは「いいえ」です。コンパイラは2つのDateオブジェクトが等しいかどうかを比較する方法を知りません。自動的にメンバを比較することはありません。==や!=のような演算子をカスタムクラスのオブジェクトに使えるようにするには、これらの演算子をオーバーロード **(Overload)**する、つまりカスタムの実装を提供する必要があります。これについては次講で詳しく学びます。

---

5. デフォルトコンストラクタ (Default Constructor)

引数を提供せずにオブジェクトを生成する場合、例えばDateオブジェクトの配列

```
Date darray[3];
```

を宣言する場合、コンパイラは配列の各要素を初期化するために引数なしのコンストラクタを必要とします。クラスに引数付きのコンストラクタしか存在しない場合、このような宣言はコンパイルエラーを引き起こします。

このような「引数なしの初期化」機能を提供するためには、デフォルトコンストラクタが必要です。

デフォルトコンストラクタの実装

デフォルトコンストラクタは、引数を取らないコンストラクタです。データメンバを合理的なデフォルト値に初期化することができます。

例えば、`Date`クラスにデフォルトコンストラクタを追加して、日付を西暦1年1月1日に初期化することができます:

```C/C++
// Date.h (一部)
class Date {
    // ...
public:
    Date(); // デフォルトコンストラクタの宣言
    Date(int yy, int mm, int dd);
    // ...
};

// Date.cpp (一部)
Date::Date() { // デフォルトコンストラクタの定義
    y = 1;
    m = 1;
    d = 1;
}

// main関数 (一部)
int main()
{
    // ...
    Date x;
    Date darray[3];
    // ...
}
```

このデフォルトコンストラクタがあれば、`Date darray[3];`のような宣言が正当になり、配列の各`Date`オブジェクトは西暦1年1月1日に初期化されます。

コンストラクタのオーバーロードとデフォルト引数

C++ではコンストラクタのオーバーロードが許可されており、一つのクラスが同じ名前で異なる引数リストを持つ複数のコンストラクタを持つことができます。これにより、オブジェクトの作成に大きな柔軟性がもたらされます。

オーバーロードに加えて、デフォルト引数を使ってコンストラクタの記述を簡素化することもできます。例えば、

`Date(int yy, int mm, int dd)` を

`Date(int yy = 1, int mm = 1, int dd = 1)`

に変更すると、一つのコンストラクタで多種の初期化方法を実現できます：

```C/C++
// Date.h (一部)
class Date {
    // ...
public:
    // デフォルト引数を持つコンストラクタ
    Date(int yy = 1, int mm = 1, int dd = 1);
    // ...
};

// 呼び出し例
Date p;          // Date(1, 1, 1) と等価
Date q(2021);    // Date(2021, 1, 1) と等価
Date r(2022, 2); // Date(2022, 2, 1) と等価
Date a(2023, 3, 5); // Date(2023, 3, 5) と等価
```

注意：`Date()`と`Date(int yy = 1, ...)`のようなコンストラクタが両方存在する場合、`Date p;` の呼び出しは曖昧さを生じさせます。コンパイラはどちらを呼び出すべきか判断できません。したがって、通常はどちらか一方の方法を選択します。

---

6. `const`メンバ関数 (Const Member Functions)

`const`オブジェクト

オブジェクトを`const`として宣言する場合、例えば

`const Date birthday(1963, 11, 18);`

これはそのオブジェクトの内部状態（データメンバ）が初期化後に変更できないことを意味します。`const`オブジェクトを変更しようとする試みはすべてコンパイルエラーになります。

constメンバ関数の宣言と定義

constオブジェクトに対してメンバ関数を呼び出すためには、これらのメンバ関数がconstメンバ関数として宣言されている必要があります。constメンバ関数は、オブジェクトの状態を変更しないことを約束します。

constメンバ関数を宣言するには、引数リストの後、関数本体（または宣言の末尾のセミコロン）の前にconstキーワードを追加します：

```C/C++
// クラス定義内部での宣言と定義
// Date.h (一部)
class Date {
    // ...
    int year() const { return y; } // constメンバ関数
    // ...
};
```

```C/C++
// クラス定義外部での宣言と定義
// Date.h (一部)
class Date {
    // ...
    int year() const; // 宣言
    // ...
};

// Date.cpp (一部)
int Date::year() const { // 定義
    return y;
}
```

ベストプラクティス：オブジェクトの状態を変更しないすべてのメンバ関数はconstとして宣言すべきです。これによりコードの安全性が向上し、意図がより明確になります。

---

7. thisポインタ

thisポインタの意味と型

C++では、すべての非静的メンバ関数にthisという名前の暗黙的で特別なポインタが含まれています。thisポインタは、そのメンバ関数を呼び出したオブジェクト自体を指します。

- thisの型はClassName* constです。これはthisが定数ポインタであり、常に同じオブジェクトを指すことを意味しますが、それが指すオブジェクト(*this)の値は変更可能です(thisがconstオブジェクトを指している場合を除く)。
- メンバ関数がconstの場合、thisの型はconst ClassName* constとなり、それが指すオブジェクトもconstであることを意味し、thisを通じてオブジェクトの状態を変更することはできません。

thisポインタの役割

thisポインタにより、メンバ関数は所属するオブジェクトのメンバにアクセスできます。例えば、preceding_day()関数では、*thisを通じて現在のオブジェクトの値を取得できます。

```C/C++
// List 11-5: Date.h (第2版：ヘッダ部)
#include <string>
#include <iostream>

class Date
{
    int y; // 西暦年 (年)
    int m; // 月
    int d; // 日

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

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

    // 前日の日付を返却 (閏年非対応)
    Date preceding_day() const;

    // 文字列表現を返却 (返回字符串表示)
    std::string to_string() const;
};
```

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

C/C++

```
// List 11-6A: Date.cpp
// 日付クラスDate（第2版：ソース部）
#include <ctime>
#include <sstream>
#include <iostream>
#include "Date.h"

using namespace std;

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

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

// --- Dateのコンストラクタ（指定された年月日に設定）---
Date::Date(int yy, int mm, int dd)
{
    y = yy;
    m = mm;
    d = dd;
}

// --- 前日の日付を返却（閏年非対応）---
Date Date::preceding_day() const
{
    int dmax[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    Date temp = *this; // 同一日付

    if (temp.d > 1)
    {
        temp.d--;
    }
    else
    {
```

```
        if (--(temp.m) < 1)
        {
            temp.y--;
            temp.m = 12;
        }
        temp.d = dmax[temp.m - 1];
    }
    return temp;
}

// --- 文字列表現を返却 ---
string Date::to_string() const
{
    ostringstream s; // 出力文字列ストリームオブジェクトを作成

    // データをストリームに挿入
    s << y << "年" << m << "月" << d << "日";

    // s.str()で、ストリームs内部の文字列をコピーして返す
    return s.str();
}

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

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

    // 期待される入力形式: yy [任意の文字] mm [任意の文字] dd
    // 例: 2025/11/26
    s >> yy >> ch >> mm >> ch >> dd;
    x = Date(yy, mm, dd);
    return s;
}
```

C/C++

```
// List 11-7: main02.cpp
// 日付クラスDate（第2版）の利用例（メンバ関数preceding_dayの働きを確認）
#include <iostream>
```

```
#include "Date.h"

using namespace std;

int main()
{
    Date today; // 今日

    cout << "今日は" << today << "です。\n";
    cout << "昨日は" << today.preceding_day() << "です。\n";
    cout << "一昨日は" << today.preceding_day().preceding_day() << "です。\n";
}
```

Shell
```
g++ main02.cpp Date.cpp -o main02; if($?) { .\main02 }
```

クラス型のオブジェクトを返す

C++の関数はクラス型のオブジェクトを返すことができます。`preceding_day()`関数が`return temp;`を実行すると、`temp`オブジェクトのコピーが作成され、関数の戻り値として返されます。このプロセスには通常、コピーコンストラクタが関与します。

`this->`を使用したメンバアクセスによる名前の隠蔽回避

メンバ関数の引数やローカル変数がデータメンバと同じ名前を持つ場合、名前の隠蔽 **(Name Hiding)**が発生します。この状況では、引数やローカル変数がデータメンバを「覆い隠し」ます。データメンバを明示的に参照するには、`this->`を使用する必要があります。

例えば、コンストラクタで引数名がデータメンバ名と同じ場合：

C/C++
```
class Human {
    int height; // データメンバ
public:
    Human(int height) { // 引数heightがデータメンバheightを隠蔽
        this->height = height; // this->を使用してデータメンバを参照する必要がある
    }
};
```

---

8. 文字列ストリーム

`to_string()`メンバ関数

`Date`クラスの`to_string()`メンバ関数は、フォーマットされた日付文字列（例：「2025年12月18日」）を返すことを目的としています。オブジェクトの状態を変更しないため、`const`として宣言されます。

```C/C++
// Date.cpp (一部)
#include <string>
#include <sstream>
std::string Date::to_string() const {
    std::ostringstream s; // 出力文字列ストリームオブジェクトを作成
    s << y << "年" << m << "月" << d << "日"; // データをストリームに挿入
    return s.str(); // フォーマットされた文字列を返す
}
```

文字列ストリームの用途

C++の文字列ストリーム **(String Streams)**（`<sstream>`ヘッダで定義）は、文字列を入力/出力ストリームとして扱うことを可能にし、データのフォーマットや文字列の解析に非常に便利です。

- **`std::ostringstream`**: 様々な型のデータを文字列にフォーマットして出力するために使用します。
- **`std::istringstream`**: 文字列から様々な型のデータを解析（抽出）するために使用します。
- **`std::stringstream`**: 入力と出力の両方をサポートします。

---

9. 挿入演算子（`operator<<`）のオーバーロード

演算子オーバーロード **(Operator Overloading)**は、カスタム型に対してC++演算子の振る舞いを再定義することを可能にします。挿入演算子`<<`をオーバーロードすることで、組み込み型のように`std::cout << myDate;`と直接`Date`オブジェクトを印字できるようになります。

`operator<<`は通常、非メンバ関数として実装され、privateメンバにアクセスする必要がある場合はクラスの`friend`関数として宣言されます。

シグネチャ: `std::ostream& operator<<(std::ostream& os, const Date& d)`

- 最初の引数`std::ostream& os`は出力ストリームへの参照で、データを書き込み、連鎖呼び出しを可能にします。
  例: `cout << Date(2025, 1, 1) << Date(2025, 1, 2) << endl;`
- 2番目の引数`const Date& d`は出力する`Date`オブジェクトへの定数参照で、元のオブジェクトを変更せず、不要なコピーを避けます。

実装例:

```
C/C++
// Date.cpp (一部)
#include <ostream> // ostreamをインクルード
// ...

std::ostream& operator<<(std::ostream& os, const Date& d) {
    return os << d.to_string(); // to_string()メンバ関数を利用してフォーマット出力
}
```

---

10. 抽出演算子(`operator>>`)のオーバーロード

同様に、抽出演算子`>>`をオーバーロードすることで、`std::cin >> myDate;`と直接入力ストリームからデータを読み込み、`Date`オブジェクトの値を設定できます。

シグネチャ: `std::istream& operator>>(std::istream& is, Date& d)`

- 最初の引数`std::istream& is`は入力ストリームへの参照で、データを読み込み、連鎖呼び出しを可能にします。
  例:
  ```
  Date x, y;
  cin >> x >> y;
  ```
- 2番目の引数`Date& d`は変更する`Date`オブジェクトへの非`const`参照で、入力操作がオブジェクトの状態を変更するためです。

実装例:

```
C/C++
// Date.cpp (一部)
#include <istream> // istreamをインクルード
// ...
```

```
std::istream &operator>>(std::istream &s, Date &x)
{
    int yy, mm, dd;
    char ch;

    // 期待される入力形式: yy [任意の文字] mm [任意の文字] dd
    // 例: 2025/11/26
    s >> yy >> ch >> mm >> ch >> dd;
    x = Date(yy, mm, dd);  // 読み取った値で一時Dateオブジェクトを構築し、xに代入
    return s;
}
```

11.2 クラスの組み合わせ

本講では、あるクラスのオブジェクトを別のクラスのメンバとすることで、より複雑なクラスを構築する方法、すなわちコンポジション **(Composition)**について学びます。また、コンストラクタにおけるメンバ初期化子リスト **(Member Initializer List)**の重要性とその動作原理（メンバの初期化順序を含む）についても深く探ります。

---

1. メンバとしてのクラス：コンポジション (Composition)

C++では、あるクラスが別のクラスのオブジェクトをデータメンバとして含むことができます。この関係はコンポジションと呼ばれ、「has-a」（〜を持つ）関係を体現します。例えば、`Account`（銀行口座）オブジェクトは、その開設日として`Date`（日付）オブジェクトを「持ち」ます。

“has-a”関係

クラス`A`のオブジェクトがクラス`B`のオブジェクトを含む場合、`A`クラスと`B`クラスの間には「has-a」関係が存在すると言います。これは`B`が`A`の一部であることを示す強い関連です。

メンバサブオブジェクト

別のクラスに含まれるオブジェクトはメンバサブオブジェクト **(Member Sub-object)**と呼ばれます。例えば、`Account`クラスでは、`Date`型の`open`オブジェクトが`Account`のメンバサブオブジェクトです。

AccountクラスとDateクラスの統合

Accountクラスを修正し、口座の開設日を表すDate型のメンバopenを追加します。これがAccountクラスの第5版となります。

Account.h (List 11-9)

```
C/C++
// Account.h: 銀行口座クラス（第5版）のヘッダファイル
#include <string>
#include "Date.h"

class Account
{
private:
    std::string full_name; // 口座名義
    std::string number;    // 口座番号
    long crnt_balance;     // 預金残高
    Date open;             // 口座開設日

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

    void deposit(long amnt);        // 預け入れる
    void withdraw(long amnt);       // おろす

    std::string name() const { return full_name; }  // 口座名義を調べる
    std::string no() const { return number; }       // 口座番号を調べる
    long balance() const { return crnt_balance; }   // 預金残高を調べる
    Date opening_date() const { return open; }      // 口座開設日を調べる
};
```

---

2. メンバ初期化子リスト (Member Initializer List)

クラスが他のクラスのオブジェクトをメンバとして含む場合、これらのメンバサブオブジェクトを正しく効率的に初期化することが非常に重要です。C++は、この作業を遂行するためにメンバ初期化子リスト **(Member Initializer List)**を提供します。

なぜメンバ初期化子リストを使うのか？

`Account`クラスのコンストラクタを考えてみましょう。`open`メンバの値を設定するには2つの方法があります：

1. コンストラクタ本体での代入 (非推奨):

```
C/C++
Account::Account(..., int y, int m, int d) {
    full_name = name;
    // ... 他のメンバの代入
    open = Date(y, m, d); // ここは代入操作
}
```

    この方法の欠点は、`open`メンバがまずデフォルトコンストラクタで初期化され、次に`Date(y, m, d)`が一時**`Date`**オブジェクトを作成し、最後にコピー代入演算子によって一時オブジェクトの値が`open`に代入されることです。これにより、不要なデフォルトコンストラクタ呼び出しと代入操作が発生し、非効率です。

2. メンバ初期化子リストの使用 (推奨):

```
C/C++
Account::Account(std::string name, std::string num, long amnt, int y, int m,
int d)
    : full_name(name), number(num), crnt_balance(amnt), open(y, m, d) {
    // コンストラクタ本体は空でもよい
}
```

ここで、`open(y, m, d)`は`Date`クラスのコンストラクタを直接呼び出して`open`メンバを初期化します。これは直接初期化であり、効率が良いです。

メンバ初期化子リストの構文

メンバ初期化子リストは、コンストラクタの引数リストの後、関数本体の前に位置し、コロン`:`で始まり、各メンバの初期化式はカンマ`","`で区切られます。

構文: `Constructor(...) : member1(arg1), member2(arg2), ... { /* コンストラクタ本体 */ }

効率と正確性

- 効率: クラス型のメンバに対してメンバ初期化子リストを使用すると、不要なデフォルトコンストラクタ呼び出しと代入操作を避け、引数付きコンストラクタを直接呼び出すことができるため、効率が向上します。
- 正確性: 以下の型のメンバは、必ずメンバ初期化子リストで初期化する必要があります:
    - `const`データメンバ
    - 参照データメンバ
    - デフォルトコンストラクタを持たないクラス型のメンバ

重要: 組み込み型を含むすべてのデータメンバに対して、一貫したコードスタイルとベストプラクティスを維持するために、メンバ初期化子リストで初期化することを推奨します。

---

3. `Account`クラスと`Date`クラスの組み合わせ例

`List 11-10`は、`Account`クラスのコンストラクタの実装を示しており、メンバ初期化子リストを使用して`Date`型の`open`メンバを初期化しています。

```C/C++
// List 11-10: Account.cpp
// 銀行口座クラス（第5版）のソースファイル
#include <iostream>
#include "Account.h"

// コンストラクタ定義、メンバ初期化子リストでopenメンバを初期化
Account::Account(std::string name, std::string num, long amnt, int y, int m,
int d)
    : full_name(name), number(num), crnt_balance(amnt), open(y, m, d)
{
    // 全メンバが初期化子リストで初期化されるため、コンストラクタ本体は空でよい
}

void Account::deposit(long amnt)  { crnt_balance += amnt; } // 預け入れる
void Account::withdraw(long amnt) { crnt_balance -= amnt; } // おろす
```

組み合わせたオブジェクトのメンバへのアクセス

`List 11-11`は、この更新された`Account`クラスの使い方を示しています。

C/C++

```
// List 11-11: AccountTest.cpp
// 銀行口座クラス（第5版）の使用例
#include <iostream>
#include "Account.h" // Account.hをインクルード、間接的にDate.hもインクルードされる
using namespace std;

int main() {
    // Account オブジェクトを作成し、開設日を提供する
    Account suzuki("鈴木龍一", "12345678", 1000, 2025, 1, 24);
    Account takeda("武田浩文", "87654321", 200, 2023, 7, 15);

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

    cout << "■鈴木君の口座\n";
    cout << "口座名義 = " << suzuki.name() << "\n";
    cout << "口座番号 = " << suzuki.no() << "\n";
    cout << "預金残高 = " << suzuki.balance() << "円\n";
    // Date オブジェクトを直接出力、Dateクラスのoperator<<オーバーロードを利用
    cout << "開設日 = " << suzuki.opening_date() << "\n";

    cout << "■武田君の口座\n";
    cout << "口座名義 = " << takeda.name() << "\n";
    cout << "口座番号 = " << takeda.no() << "\n";
    cout << "預金残高 = " << takeda.balance() << "円\n";
    // 返されたDateオブジェクトのメンバ関数にアクセス
    cout << "開設日 = " << takeda.opening_date().year() << "年"
         << takeda.opening_date().month() << "月"
         << takeda.opening_date().day() << "日\n";
}
```

Shell

```
g++ AccountTest.cpp Account.cpp Date.cpp -o actest; if($?) { .\actest }
```

この例は、`Account`オブジェクトの`opening_date()`メンバ関数を通じて`Date`メンバサブオブジェクトを取得し、さらに`Date`オブジェクトのメンバにアクセスしたり、オーバーロードされた`operator<<`を使用して出力する方法を示しています。

---

4. 引数としてのオブジェクト：コンストラクタの改良

`Account`クラス第5版のコンストラクタ`Account(string, string, long, int, int, int)`は引数リストが長すぎて可読性が低いです。直接`Date`オブジェクトを渡すことでこれを改善できます。

オブジェクト渡し vs. 基本型渡し

複数の関連する基本型引数（`int y, int m, int d`など）を一つのオブジェクト（`Date`）にまとめて渡すことは、優れたオブジェクト指向設計の現れです。これによりインターフェースがよりクリーンになり、意図が明確になります。

Accountクラスの第6版

`Account.h` (List 11-14)

```C/C++
// Account.h (第6版)
// ... (ヘッダガード)
#include "Date.h"

class Account {
    // ...
public:
    // コンストラクタは複数のintではなく、const Date&を受け取る
    Account(std::string name, std::string num, long amnt, const Date& op);
    // ...
};
```

`Account.cpp` (List 11-15)

```C/C++
// Account.cpp (第6版)
// ...

// コンストラクタの実装
Account::Account(std::string name, std::string num, long amnt, const Date& op)
    : full_name(name), number(num), crnt_balance(amnt), open(op) {
    // open(op)はDateクラスのコピーコンストラクタを呼び出す
}
```

使用例 (`List 11-16`)

```C/C++
// main.cpp
int main() {
    // 引数内で直接一時Dateオブジェクトを構築
    Account suzuki("鈴木龍一", "12345678", 1000, Date(2025, 1, 24));

    // ...
}
```

この改良により、`Account`の作成がより直感的で型安全になります。

---

5. C言語の`struct`と`union`

C++の`class`と比較して、C言語の`struct`と`union`の機能は限定的です。

- **`struct` (C言語)**: データメンバしか含めず、メンバ関数や`private`/`public`アクセス制御を持つことはできません。すべてのメンバはpublicです。

```C/C++
// Date構造体の定義
struct Date {
    int year;  // 年
    int month; // 月
    int day;   // 日
};
```

- **`struct` (C++)**: C++では、`struct`は`class`とほぼ同じですが、唯一の違いは`struct`のデフォルトのメンバアクセス権が`public`であるのに対し、`class`は`private`である点です。
- **`union` (共用体)**: すべてのメンバが同じメモリ空間を共有し、一度にそのうちの一つのメンバの値しか格納できません。主にメモリを節約するため、または同じメモリブロックを異なる型で解釈するために使用されます。

```C/C++
union xyz {
    int x;
    long y;
    double z;
```

```
};

union xyz q;
```

a 構造体

```
struct abc {
    int    a;
    long   b;
    double c;
};

struct abc p;
```

b 共用体

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

union xyz q;
```

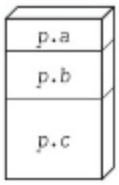

Fig.11C-2　構造体と共用体