第12講 クラスの発展編

12.1 変換関数と演算子関数

本節では、整数をカウントするための `Counter` クラスをゼロから作成することを通して、C++におけるクラスの設計、メンバ関数、コンストラクタを深く学び、特に型変換関数と演算子のオーバーロードによって、カスタムクラスの振る舞いを `int` のような組み込み型のように自然にする方法を探求します。

---

1. カウンタクラス

本節では、整数をカウントするための `Counter` クラスを設計し、実装します。

クラスの基本構想と機能定義

`Counter` オブジェクトには、以下の4つの基本能力を持たせたいと考えます：

1. 初期化：オブジェクト作成時に、カウンタを自動的に0に初期化する。
2. インクリメント：カウンタに1を加える操作。
3. デクリメント：カウンタから1を引く操作。
4. 値の取得：カウンタの現在の値を読み取る。

`Counter` クラスの規模は比較的小さいため、すべてのメンバ関数をインライン関数として実装し、クラスの完全な定義をヘッダファイル `Counter.h` に配置します。

`Counter` 内部でカウントに使用するメンバ `cnt` は `unsigned int` 型です。これは符号なし整数で、その表現範囲は 0 から `UINT_MAX`（`<climits>` で定義される定数、通常は 4294967295）です。これは `cnt` が決して負数にならないことを意味します。

ヘッダファイル `Counter.h` の実装 (`List 12-1`)

`List 12-1` は `Counter` クラスの最初のバージョンを示しています。

```C/C++
// List 12-1: Counter.h
// Counter クラス（第1版）
#ifndef ___Class_Counter
#define ___Class_Counter
```

```
#include <climits> // UINT_MAX を使用するため

//===== カウンタクラス =====//
class Counter
{
    unsigned int cnt; // カウンタ（符号なし整数）

public:
    //--- コンストラクタ ---
    Counter() : cnt(0) {}

    //--- カウンタアップ ---
    void increment()
    {
        // オーバーフローを防止
        if (cnt < UINT_MAX)
            cnt++;
    }

    //--- カウンタダウン ---
    void decrement()
    {
        // アンダーフローを防止
        if (cnt > 0)
            cnt--;
    }

    //--- カウンタを返す ---
    unsigned int value() const
    {
        return cnt;
    }
};

#endif
```

コンストラクタ (Constructor)

`Counter() : cnt(0) { }` はクラスのコンストラクタです。

- これは引数を受け取らないコンストラクタです。メンバ初期化子リスト **(`: cnt(0)`)** を通じて、オブジェクト作成時に `cnt` メンバを効率的に0に初期化します。
- `Counter c;` のようにオブジェクトを宣言すると、このコンストラクタが呼び出され、`c` のカウント値`cnt`が0に初期化されます。

メンバ関数 (Member Functions)

- `increment()`: `cnt` のインクリメント操作を実行します。
  - 安全対策：オーバーフロー（`unsigned int` が表現できる最大値 `UINT_MAX` を超えること）を防ぐため、チェックを加えています。`
- `decrement()`: `cnt` のデクリメント操作を実行します。
  - 安全対策：アンダーフロー（0から非常に大きな正数に変わること）を防ぐため、`cnt > 0` をチェックします。
- `value()`: `cnt` の現在の値を返します。
  - `const`修飾子：この関数はクラスのデータメンバを変更しない「読み取り専用」関数であるため、`const` として宣言されています。

---

2. `Counter` クラスの使用と分析

使用例 (`List 12-2`)

`List 12-2` の `main` 関数は `Counter` オブジェクトを作成し、ユーザーの入力に応じてカウント値を増減させ、その振る舞いを検証します。

```C/C++
// List 12-2: CounterTest.cpp
// Counterクラス（第1版）の利用例
#include <iostream>
#include "Counter.h"

using namespace std;

int main()
{
    Counter x; // オブジェクトxを作成。コンストラクタが呼ばれ、x.cntが0に初期化される

    cout << "現在のカウンタ：" << x.value() << "\n";

    int no;
    cout << "カウントアップ回数：";
    cin >> no;
    for (int i = 0; i < no; i++) {
        x.increment(); // メンバ関数を呼び出してインクリメント
    }
    cout << "現在のカウンタ：" << x.value() << "\n";
```

```
    cout << "カウントダウン回数：";
    cin >> no;
    for (int i = 0; i < no; i++) {
        x.decrement(); // メンバ関数を呼び出してデクリメント
    }
    cout << "現在のカウンタ：" << x.value() << "\n";
}
```

コードの振る舞い分析

このテストプログラムを通じて、主に2つの点を確認しました：

- オブジェクト生成時、その `cnt` メンバは確かにコンストラクタによって正しく0に初期化されています。
- `cnt` メンバはクラスの内部にカプセル化 **(encapsulated)** されています。外部コードは `cnt` を直接読み書きできず、`increment`, `decrement` などのメンバ関数を介してのみ操作できます。これにより、データの完全性が保証され、例えば外部コードが負の値を `cnt` に代入することはできません。

`Counter` クラスの使用方法は、`int` などの組み込み型とは明らかに異なり、やや煩雑に感じられます：

- 操作方法: `int` 型の変数 `n` に対しては、`n++` や `n--` を直接使用できます。しかし、`Counter` オブジェクト `x` に対しては、メンバ関数 `x.increment()` や `x.decrement()` を呼び出す必要があります。
- 値の取得: `int` 変数 `n` の値を取得するには、単に `n` を使用します。しかし、`Counter` オブジェクト `x` の値を取得するには、`x.value()` を呼び出す必要があります。

この違いはコードの直感性を低下させます。目標は、`Counter` クラスを `int` のように使いやすくすることです。

---

3. 型変換関数：他型への変換を実現

`Counter` オブジェクトを `unsigned int` のように直接使用できるようにするため、型変換関数 **(conversion function)** を定義することができます。

`operator Type` の定義と実装

型変換関数の名前は `operator Type` で、ここで `Type` はターゲット型です。通常、オブジェクトの状態を変更すべきではないため、`const` として宣言されます。

```C/C++
// Counter クラスの public 部分に以下の関数を追加
public:
    // ... 他のメンバ ...

    operator unsigned() const { return cnt; } // unsigned 型への変換
```

この関数は、「`Counter` オブジェクトをどのように `unsigned` 型に変換するか」、すなわち内部の `cnt` の値を返す方法を定義します。

型変換関数の呼び出し

型変換関数を定義すると、`unsigned` が必要な文脈で、コンパイラがそれを呼び出します（明示的型変換・暗黙の型変換）。

```C/C++
unsigned x;
Counter cnt;

x = unsigned(cnt);              // 明示的キャスト：変換関数が呼び出される
x = (unsigned)cnt;              // 明示的キャスト
x = static_cast<unsigned>(cnt); // 明示的キャスト
x = cnt;                        // 暗黙的キャスト
```

---

4. 演算子のオーバーロード

演算子のオーバーロード **(Operator Overload)** はC++の強力な機能で、カスタムクラスに対してほとんどのC++演算子の振る舞いを再定義することができます。

演算子のオーバーロードは、演算子関数を定義することによって実現されます。その名前は `operator☆` で、☆ はオーバーロードする演算子です。

論理否定演算子 ! のオーバーロード

論理否定演算子 `!` をオーバーロードして、カウンタが0かどうかを判断できるようにします。

```C/C++
// Counter クラスの public 部分に以下の関数を追加
```

```
public:
    // ... 他のメンバ ...

    // 論理否定演算子
    bool operator!() const { return cnt == 0; }
```

- 定義: `bool operator!() const` は `bool` 値を返す演算子関数を定義します。
- ロジック: `cnt` が0の時に `true` を返し、それ以外の場合は `false` を返します。
- 使用: この定義により、`if (!c)` のようなコードを書いてカウンタ `c` が0かどうかをチェックでき、`if (c.value() == 0)` よりも簡潔で直感的です。

```
C/C++
    if (!cnt)
        cout << "!cnt = True \n";
```

---

5. インクリメント・デクリメント演算子のオーバーロード

次に、`"++"` と `"--"` 演算子のオーバーロードを実装します。これらの演算子には前置 **(prefix)** と後置 **(postfix)** の2つの形式があります。

形式	例	意味
前置形式	`++c`	まず `c` をインクリメントし、インクリメント後の `c` の値を返す。
後置形式	`c++`	まず `c` をインクリメントし、インクリメント前の `c` の値を返す。

C++でこれら2つの形式を区別するため、後置形式の演算子関数は `int` 型のダミー引数を必要とします。

前置インクリメント演算子 (`operator++()`)

前置インクリメント演算子 `++c` の関数定義は以下の通りです。引数を受け取らず、左辺値として使用できるように、自身への参照を返します。

```
C/C++
//--- 前置インクリメント演算子 ---
```

```
Counter& operator++()
{
    if (cnt < UINT_MAX) {
        cnt++;
    }
    return *this; // 返却するのは自分自身への参照
}
```

- ロジック: まず `cnt` の値を増やします（同様にオーバーフローチェックあり）。
- 戻り値: `return *this;` は、この関数を呼び出したオブジェクト自身の参照を返します。参照を返すことで、`++(++c)` のような連続呼び出しが可能になり、その結果を代入することもできます（左辺値として）。

後置インクリメント演算子 (`operator++(int)`)

後置インクリメント演算子 `c++` の関数定義は、前置バージョンと区別するために `int` 型の引数を必要とします。この `int` 引数は単なる識別子であり、実際には使用されません。

`++c` と書くと、コンパイラは `c.operator++()` に変換します。`c++` と書くと、コンパイラは `c.operator++(0)` に変換し、値0の整数リテラルを渡して後置バージョンを呼び出します。

```C/C++
//--- 後置インクリメント演算子 ---
Counter operator++(int) {
    Counter x = *this; // 1. インクリメント前のオブジェクトの状態を保存
    if (cnt < UINT_MAX) {
        cnt++;         // 2. 現在のオブジェクトをインクリメント
    }
    return x;          // 3. 保存しておいた状態を返す
}
```

- ロジック:
    1. 一時オブジェクト `x` を作成し、現在のオブジェクト (`*this`) の状態で初期化します。これはインクリメント前のコピーを保存するのと同じです。
    2. 現在のオブジェクトの `cnt` メンバをインクリメントします。
    3. 以前に作成した一時オブジェクト **`x`** を返します。`x` にはインクリメント前の値が保存されています。
- 実装戦略: 通常、後置演算子の実装は、コードの重複を避け効率を上げるために、前置演算子を呼び出します。

インクリメント/デクリメント前の値が必要ない場合は、前置形式 (**++c**) を優先して使用すべきです。なぜなら、一時オブジェクトの作成と返却のオーバーヘッドを避けられるからです。

Counter クラスの第2版

すべての新しい演算子関数（インクリメント/デクリメントの前置・後置バージョン）、および以前の型変換と論理否定演算子を Counter クラスに統合し、第2版の Counter クラスを作成します。

List 12-3 は更新されたヘッダファイル Counter2.h です。

```C/C++
// List 12-3: Counter2.h
// カウンタクラス（第２版）
#ifndef ___Class_Counter
#define ___Class_Counter

#include <climits>

//===== カウンタクラス =====//
class Counter {
    unsigned cnt;

public:
    //--- コンストラクタ ---
    Counter() : cnt(0) { }

    //--- unsignedへの変換関数 ---
    operator unsigned() const { return cnt; }

    //--- 論理否定演算子！ ---
    bool operator!() const { return cnt == 0; }

    //--- 前置インクリメント演算子++ ---
    Counter& operator++() {
        if (cnt < UINT_MAX) cnt++;
        return *this;
    }

    //--- 後置インクリメント演算子++ ---
    Counter operator++(int) {
        Counter x = *this;
        if (cnt < UINT_MAX) cnt++;
        return x;
    }

    //--- 前置デクリメント演算子-- ---
```

```
    Counter& operator--() {
        if (cnt > 0) cnt--;
        return *this;
    }

    //--- 後置デクリメント演算子-- ---
    Counter operator--(int) {
        Counter x = *this;
        if (cnt > 0) cnt--;
        return x;
    }
};

#endif
```

List 12-4 は第2版 Counter クラスを使用するテストプログラムです。

C/C++
```
// List 12-4: Counter2Test.cpp
// カウンタクラス（第２版）の利用例
#include <iostream>
#include "Counter2.h"

using namespace std;

int main()
{
    int no;
    Counter x;
    Counter y;

    // カウントアップ回数:
    cout << "カウントアップ回数：";
    cin >> no;
    for (int i = 0; i < no; i++)
        // xは後置でyは前置
        cout << x++ << ' ' << ++y << '\n';

    // カウントダウン回数:
    cout << "カウントダウン回数：";
    cin >> no;
    for (int i = 0; i < no; i++)
        // xは後置でyは前置
        cout << x-- << ' ' << --y << '\n';
```

```
    // 論理否定演算子による判定
    if (!x)
        cout << "xは0です。\n";
    else
        cout << "xは0ではありません。\n";
}
```

テストコードが非常に直感的になり、ほとんど組み込みの `int` 型を操作するのと同じになったことがわかります。

12.2 複素数クラス

本節では、完全な複素数クラス `Complex` を作成することを通じて、演算子オーバーロードの詳細、特にメンバ関数と非メンバ（フレンド）関数で演算子を実装する際の区別と選択について深く理解します。

---

1. 複素数クラスの基本設計

`Complex` 複素数クラスの作成は、演算子オーバーロードをマスターするための優れた練習です。

複素数の数学的背景

- 複素数は `a + bi` の形式で表され、`a` は実部 (real part)、`b` は虚部 (imaginary part) 、`i` は虚数単位で `i² = -1` と定義されます。
- 例えば、`(x + yi)` の二乗は `(x² - y²) + 2xyi` です。これらの複素数演算のルールが、演算子をオーバーロードする際のロジックの基礎となります。

`Complex` クラスの定義

複素数は実部 と虚部から構成されます。`Complex` クラスは、それぞれを表すために2つの `double` 型のプライベートメンバ `re` と `im` を含みます。

```C/C++
// Complex.h (初期バージョン)
class Complex {
```

```
    double re; // 実部
    double im; // 虚部

public:
    // コンストラクタ
    Complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }

    // 実部を取得
    double real() const { return re; }

    // 虚部を取得
    double imag() const { return im; }
};
```

コンストラクタ

`Complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }`

これはデフォルト引数を持つコンストラクタです。この設計は非常に柔軟で、1つの関数が複数のコンストラクタの役割を果たします：

- デフォルトコンストラクタ: `Complex c1;` ⇒ `Complex(0.0, 0.0)` を呼び出す
- 変換コンストラクタ: `Complex c2(1.2);` ⇒ `Complex(1.2, 0.0)` を呼び出す
- 通常コンストラクタ: `Complex c3(1.2, 3.7);` ⇒ `Complex(1.2, 3.7)` を呼び出す
- デフォルトコンストラクタが存在するため、`Complex` オブジェクトの配列を作成できます（例：`Complex a[10];`）。その場合、各要素は (0, 0) に初期化されます。

単項演算子 + と -

`Complex` クラスに対して、単項の `+`（正号）と `-`（負号）演算子をオーバーロードできます。これらの演算子は単一のオブジェクトに作用するため、メンバ関数として適しています。

```C/C++
// Complex クラスの public 部分に追加
public:
    // ...

    // 単項 + 演算子
    Complex operator+() const {
        return *this;
    }

    // 単項 - 演算子
    Complex operator-() const {
```

```
        return Complex(-re, -im);
    }
```

- `operator+()`: オブジェクト自身のコピーを返し、何も変更しません。
- `operator-()`: 実部と虚部が元のオブジェクトの反対符号である新しい `Complex` オブジェクトを返します。

使用例:

```C/C++
// ComplexTest.cpp
#include <iostream>
#include "Complex.h"
using namespace std;

int main()
{
    Complex x(1.0, 0.5);
    Complex y = -x;
    Complex z(+x);
}
```

---

2. 二項演算子と非メンバ関数

次に、二項加算演算子 `+` を考えてみましょう。自然な発想は、これをメンバ関数として実装することです。

```C/C++
// メンバ関数版のoperator+（問題あり）
public:
    Complex operator+(const Complex& x) const {
        return Complex(re + x.re, im + x.im);
    }
```

`z = x + y;`（ここで `x`, `y`, `z` はすべて `Complex` オブジェクト）を実行すると、このコードはコンパイラによって `z = x.operator+(y);` と解釈され、うまく機能します。

メンバ関数 operator+ の限界

しかし、オペランドに他の型（例えば double）が含まれると問題が発生します。

- `z = x + 7.5;` // **OK**
    - これは `z = x.operator+(Complex(7.5));` と解釈されます。コンパイラは変換コンストラクタ `Complex(double)` を使用して `7.5` を一時的な `Complex` オブジェクト `(7.5, 0.0)` に暗黙的に変換し、その後メンバ関数を呼び出します。
- `z = 7.5 + x;` // コンパイルエラー!
    - コンパイラはこの行を `z = (7.5).operator+(x);` と解釈しようとします。しかし、`7.5` は `double` 型の組み込み型であり、クラスオブジェクトではないため、`operator+` というメンバ関数を持ちません。そのため、コンパイルは失敗します。

friend 関数：解決策

`7.5 + x` のような式を機能させるためには、二項演算子 `+` を非メンバ関数として定義する必要があります。そうすれば、コンパイラは両方のオペランドに型変換ルールを適用できます。

- **friend** キーワード: クラス外の関数や他クラスに対して、そのクラスの`private`メンバへのアクセス権を例外的に付与するための宣言です。

この非メンバ関数が `Complex` クラスのプライベートメンバ `re` と `im` にアクセスできるようにするためには、クラス定義内でそれをフレンド (**friend**) として宣言する必要があります。

```C/C++
// Complex クラスの定義内でフレンド関数を宣言
class Complex {
    // ...
    friend Complex operator+(const Complex& x, const Complex& y);
};

// クラス外部で非メンバ関数を定義
inline Complex operator+(const Complex& x, const Complex& y) {
    return Complex(x.re + y.re, x.im + y.im);
}
```

- 動作原理: コンパイラが `z = 7.5 + x;` を見ると、`operator+(double, Complex)` に一致する関数を探します。定義した `operator+(const Complex&, const Complex&)` を見つけ、変換コンストラクタを介して `7.5` を `Complex` オブジェクトに変換で

きることを発見します。したがって、この式は最終的に `operator+(Complex(7.5), x)` として正常に解釈されます。

上記の例では、引数の型は `const Complex&` と宣言されています。

- `&` (参照): 関数呼び出し時の `Complex` オブジェクトのコピーを避け、パフォーマンスを向上させます。
- `const`: 関数が渡された引数オブジェクトを変更しないことを保証します。これは安全性の保証です。

メンバ関数 vs. `friend` 関数

特性	メンバ関数	フレンド(非メンバ)関数
呼び出し方	`x.func(y)`	`func(x, y)`
アクセス権	`private` メンバに直接アクセス可	`private` メンバにアクセスするには `friend` 宣言が必要
`this` ポインタ	あり	なし

結論: `+`, `-`, `*`, `/` のような二項算術演算子は、`double` などの他の型との混合演算をサポートしたい場合、通常は非メンバの `friend` 関数として定義すべきです。

---

3. 演算子のさらなるオーバーロード

`Complex` クラスをより完全で使いやすくするために、さらに多くの演算子をオーバーロードする必要があります。

加法演算子のさらなるオーバーロード

`y = 5 + x;` のような(`int` + `Complex`)式が、複数回の型変換(`int` -> `double` -> `Complex`)によって効率が低下するのを避けるため、一般的な型の組み合わせに対して直接オーバーロードバージョンを提供することができます。

```C/C++
// doubleとComplexの組み合わせのためにoperator+をオーバーロード
friend Complex operator+(double d, const Complex& y) {
    return Complex(d + y.re, y.im);
```

```
}

friend Complex operator+(const Complex& x, double d) {
    return Complex(x.re + d, x.im);
}
```

複合代入演算子 (+=)

+=, -=, *= のような複合代入演算子は、左オペランドのオブジェクトの状態を変更します。そのため、これらはメンバ関数として実装されるべきです。

```C/C++
// Complex クラスの public 部分に追加
public:
    // ...

    // 複合加算代入演算子
    Complex& operator+=(const Complex& x) {
        re += x.re;
        im += x.im;
        return *this; // 自身への参照を返す
    }
```

等価/不等価演算子 (==, !=)

等価性の判断はオブジェクトの状態を変更すべきではなく、対称性（x == y と y == x は等価であるべき）のため、これらは通常非メンバ関数（多くは friend）として実装されます。

```C/C++
// Complexクラス定義内でフレンドを宣言
friend bool operator==(const Complex& x, const Complex& y);
friend bool operator!=(const Complex& x, const Complex& y);

// 外部で定義
inline bool operator==(const Complex& x, const Complex& y) {
    return x.re == y.re && x.im == y.im;
}

inline bool operator!=(const Complex& x, const Complex& y) {
    return !(x == y); // 定義済みの==演算子を利用
}
```

4. 複素数クラス

C/C++

```
// Complex.h

#ifndef __CLASS_Complex
#define __CLASS_Complex

#include <iostream>

// -------- 複素数クラス --------
class Complex
{
private:
    double re; // 実部
    double im; // 虚部

public:
    // コンストラクタ
    Complex(double r = 0, double i = 0) : re(r), im(i) {}

    double real() const { return re; }  // 実部を返す
    double imag() const { return im; }  // 虚部を返す

    Complex operator+() const { return *this; }             // 単項+演算子
    Complex operator-() const { return Complex(-re, -im); } // 単項-演算子

    // -------- 複合代入演算子+= --------
    Complex &operator+=(const Complex &x)
    {
        re += x.re;
        im += x.im;
        return *this;
    }

    // -------- 複合代入演算子-= --------
    Complex &operator-=(const Complex &x)
    {
        re -= x.re;
        im -= x.im;
        return *this;
    }

    // -------- 等価演算子== --------
    friend bool operator==(const Complex &x, const Complex &y)
    {
        return x.re == y.re && x.im == y.im;
```

```
    }

    // -------- 等価演算子!= --------
    friend bool operator!=(const Complex &x, const Complex &y)
    {
        return !(x == y);
    }

    // -------- 2項 + 演算子 (Complex + Complex) --------
    friend Complex operator+(const Complex &x, const Complex &y)
    {
        return Complex(x.re + y.re, x.im + y.im);
    }

    //--- 2項+演算子 (double + Complex) ---
    friend Complex operator+(double x, const Complex &y)
    {
        return Complex(x + y.re, y.im);
    }

    // -------- 2項 + 演算子 (Complex + double) --------
    friend Complex operator+(const Complex &x, double y)
    {
        return Complex(x.re + y, x.im);
    }

};

// -------- 非メンバ関数（挿入演算子）：出力ストリームに << を挿入 --------
// friend宣言が不要な理由：privateメンバに直接アクセスせず、
// パブリックなメンバ関数real()とimag()を通じて値を取得しているから。
inline std::ostream &operator<<(std::ostream &s, const Complex &x)
{
    return s << "(" << x.real() << ", " << x.imag() << ")";
}

#endif
```

C/C++

```
// ComplexTest.cpp

#include <iostream>
#include "Complex.h"
```

```
using namespace std;

int main()
{
    double re, im;

    cout << "A部の実部: ";
    cin >> re;
    cout << "A部の虚部: ";
    cin >> im;
    Complex a(re, im);

    cout << "B部の実部: ";
    cin >> re;
    cout << "B部の虚部: ";
    cin >> im;
    Complex b(re, im);

    Complex c = -a + b;

    b += 2.0;                       // bに (2.0, 0.0)を加える
    c -= Complex(1.0, 1.0);         // cから(1.0, 1.0)を減じる
    Complex d(b.imag(), c.real()); // dを(bの虚部, cの実部)とする

    cout << "a = " << a << '\n';
    cout << "b = " << b << '\n';
    cout << "c = " << c << '\n';
    cout << "d = " << d << '\n';
}
```

12.3 コンストラクタとデストラクタ

本節では、C++におけるコンストラクタ **(constructor)** とデストラクタ **(destructor)**、そしてそれらがクラス内で動的に割り当てられたメモリやその他のリソースを管理する上で果たす重要な役割について深く学びます。

---

1. 整数配列クラス `IntArray`

カスタムの整数配列クラス `IntArray` を通じて、コンストラクタとデストラクタを深く理解します。

クラスの基本構造

`IntArray` クラスは、整数配列へのポインタ `vec` と、配列のサイズを表す整数 `size` を含みます。

```C/C++
// List 14-1: IntArray.h (第1版)
#ifndef ___Class_IntArray
#define ___Class_IntArray

class IntArray
{
    int *vec; // 配列へのポインタ
    int size; // 配列のサイズ

public:
    // コンストラクタ
    IntArray(int sz)
    {
        size = sz;
        vec = new int[sz]; // 動的にメモリを割り当てる
    }

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

    // サイズを取得
    int get_size() const { return size; }
};

#endif
```

コンストラクタ：動的メモリ割り当て

`IntArray(int sz)` は `IntArray` クラスのコンストラクタです。

- 配列のサイズとして整数 `sz` を受け取ります。
- コンストラクタ内部で `new int[sz]` を使用してメモリを動的に割り当て、そのメモリアドレスを `vec` ポインタに代入します。

使用例 (`List 14-2`)

`List 14-2` は `IntArray` クラスの簡単な使用例を示しています。

```C/C++
// List 14-2: IntArrayTest.cpp
#include <iostream>
#include "IntArray.h"

using namespace std;

int main() {
    IntArray x(5); // サイズ5のIntArrayオブジェクトを作成

    cout << "5個の整数を入力してください：";
    for (int i = 0; i < x.get_size(); i++) {
        cin >> x[i]; // 添字演算子経由で代入
    }

    cout << "配列要素：";
    for (int i = 0; i < x.get_size(); i++) {
        cout << x[i] << " "; // 添字演算子経由でアクセス
    }
    cout << endl;
}
```

コード分析:

- `IntArray x(5);`: `IntArray` オブジェクト `x` を作成し、そのコンストラクタが呼び出されて5つの整数を含む配列が動的に割り当てられます。
- `cin >> x[i];` と `cout << x[i];`: オーバーロードされた添字演算子 `operator[]` を通じて、`IntArray` オブジェクトの要素に通常の配列のようにアクセスできます。

---

2. オブジェクトの生存期間とリソース管理

メモリリーク問題

`IntArray` オブジェクト `x` の生存期間を考えてみましょう。`main` 関数が終了すると、`x` オブジェクトは破棄されます。しかし、`x` 内部の `vec` ポインタが指す動的に割り当てられたメモリは自動的に解放されません。これによりメモリリーク **(memory leak)** が発生します。

デストラクタ:リソースの解放

メモリリーク問題を解決するために、デストラクタ **(destructor)** を導入します。デストラクタは、オブジェクトが破棄されるときに自動的に呼び出される特別なメンバ関数です。

- 命名: デストラクタの名前はクラス名と同じで、前にチルダ ~ が付きます。
- 引数と戻り値: デストラクタは引数を取れず、戻り値もありません。
- 役割: デストラクタの主な役割は、オブジェクトが生存期間中に取得したリソース（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）を解放することです。

`IntArray` クラスには、`vec` ポインタが指すメモリを解放するためのデストラクタを追加する必要があります:

```C/C++
// IntArray クラスの public 部分に追加
public:
    // ...

    // デストラクタ
    ~IntArray()
    {
        delete[] vec; // 動的に割り当てたメモリを解放
    }
```

代入演算子 (=)

代入演算子 = は、右オペランドのオブジェクトの内容で「左オペランドのオブジェクトの状態を上書き」します。そのため、左側のオブジェクトを書き換えるメンバ関数として実装するのが基本です。

- 自分自身への代入に注意し、最後に `*this` への参照を返すのが慣習です。
- すでにリソースを確保している場合は、代入処理を行う前にそれらを適切に解放する必要があります。
- もし代入演算子（=）を自分でオーバーロードしない場合、コンパイラはデフォルトの代入演算子を自動生成します。このときの動作は、各メンバに対して単純にメンバごとのコピー（浅いコピー / **shallow copy**）を行うだけです。

クラスのメンバに生ポインタや、手動で管理すべきリソース（new で確保したメモリ、ファイルハンドル、ソケットなど）が含まれている場合、デフォルトの `operator=` をそのまま使うと、次のような問題が発生します。

- 複数のオブジェクトが同じメモリアドレス（同じポインタ）を共有してしまう。
- 片方のオブジェクトがデストラクタで `delete[]` すると、もう片方のポインタはダングリングポインタになる。その後もう一方のオブジェクトがデストラクタで再び delete[] してしまい、二重解放などの未定義動作を引き起こす。
- 本来は代入前に「古いリソースを解放」すべきだが、デフォルトの `operator=` はそれを行わないため、メモリリークの原因にもなる。

このような理由から、クラス内部でリソース（特に生ポインタ）を自前で管理している場合は、代入演算子 = を必ず自分で定義し、適切なリソース管理を行う必要があります。同時に、コピーコンストラクタとデストラクタも自前で用意する「**Rule of Three**（三つ組の法則）」が重要になります。

```C/C++

public:
    // ...

    // 代入演算子
    IntArray &operator=(const IntArray &x)
    {
        if (this == &x)
            return *this; // 自己代入のときは何もしない

        delete[] vec; // いったん自分が持っている配列を解放

        // サイズをコピーして、新しく配列を確保
        size = x.size;
        vec = new int[size];
        for (int i = 0; i < size; ++i) // 要素をコピー
            vec[i] = x.vec[i];

        return *this; // 自分自身への参照を返す
    }
```

IntArray クラスの第2版 (List 14-3)

List 14-3 は、デストラクタを含む IntArray クラスの第2版を示しています。

```C/C++
// List 14-3: IntArray.h (第2版)
#ifndef ___Class_IntArray
#define ___Class_IntArray

class IntArray
{
    int *vec; // 配列へのポインタ
    int size; // 配列のサイズ

public:
    // コンストラクタ
    IntArray(int sz)
    {
        size = sz;
        vec = new int[sz]; // 動的にメモリを割り当てる
    }
```

```
    // コピーコンストラクタ
    IntArray(const IntArray &x)
    {
        // サイズをコピーして、新しく配列を確保
        size = x.size;
        vec = new int[size];
        for (int i = 0; i < size; ++i) // 要素をコピー
            vec[i] = x.vec[i];
    }

    // デストラクタ
    ~IntArray()
    {
        delete[] vec; // 動的に割り当てたメモリを解放
    }

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

    // 代入演算子
    IntArray &operator=(const IntArray &x)
    {
        if (this == &x)
            return *this; // 自己代入のときは何もしない

        delete[] vec; // いったん自分が持っている配列を解放

        // サイズをコピーして、新しく配列を確保
        size = x.size;
        vec = new int[size];
        for (int i = 0; i < size; ++i) // 要素をコピー
            vec[i] = x.vec[i];

        return *this; // 自分自身への参照を返す
    }

    // サイズを取得
    int get_size() const { return size; }
};

#endif
```

デストラクタの呼び出しタイミング

デストラクタは以下の状況で自動的に呼び出されます：

- ローカルオブジェクトがそのスコープを抜けるとき。
- `new` で作成されたオブジェクトが `delete` で解放されるとき。

- 一時オブジェクトの生存期間が終了するとき。
- プログラムが終了するとき、グローバルオブジェクトと静的オブジェクトのデストラクタが呼び出される。

コンストラクタとデストラクタの比較

特性	コンストラクタ	デストラクタ
呼び出しタイミング	オブジェクト作成時	オブジェクト破棄時
名前	クラス名	~ クラス名
引数	引数を取ることができる	引数を取れない
戻り値	戻り値なし	戻り値なし
役割	オブジェクトの初期化、リソースの取得	オブジェクトのクリーンアップ、リソースの解放

3. RAII 原則

コンストラクタでリソースを取得し、デストラクタでリソースを解放するこの設計パターンは **RAII (Resource Acquisition Is Initialization)** と呼ばれます。これはC++でリソースを管理する中心的な原則であり、リソースが不要になったときに正しくタイムリーに解放されることを保証します。

RAIIの主要な利点（メリット）

1. 自動的なリソース解放（自動解放）
   - 動作: 関数が正常な処理フローで終了する場合でも、あるいは例外(try ... exception...)が投げられたことによって異常終了する場合でも、オブジェクトのデストラクタは必ず呼び出されます。
   - 効果: これにより、リソースが常に正しく解放されることが保証され、リソースリーク（資源漏れ）を回避できます

2. コードの簡潔化
   - 動作: プログラマは、コード内のすべての終了パス（処理の出口）において、リソースを解放するためのコードを手動で記述する必要がなくなります。

例:

```C/C++
// RAII を使わない場合（すべての終了パスで手動でリソースを解放する必要）
// ...
bool func_without_raii(int n)
{
```

```
    // 業務ロジックの都合により、ここで動的配列を確保する必要
    int* data = new int[n];

    if (!init_data(data, n)) {    // 初期化に失敗した場合は、後続処理を行わずに終了
        delete[] data;            // 確保したメモリを解放
        return false;
    }

    if (!process_data(data, n)) {    // 処理に失敗した場合も、ここで終了してメモリを解
放
        delete[] data;               // 確保したメモリを解放
        return false;
    }

    // 正常終了パス
    delete[] data;
    return true;
}

// RAII を使う場合（デストラクタに任せる）
bool func_with_raii(int n)
{
    IntArray arr(n);          // 動的配列を確保（スコープ終了時にデストラクタで自動解放）

    if (!init_data(arr)) {        // 初期化に失敗した場合は、後続処理を行わずに終了
        return false;             // メモリ解放は IntArray のデストラクタに任せる
    }

    if (!process_data(arr)) {    // 処理に失敗した場合も、ここで終了
        return false;            // メモリ解放は IntArray のデストラクタに任せる
    }

    // 正常終了パス（関数を抜けるときに自動的にメモリが解放される）
    return true;
}
```