第13講 クラス：静的メンバ・STL

13.1 静的メンバとその応用

1. 静的データメンバ

これまでに扱ってきたメンバ変数は、各オブジェクトに固有のものでした。本節では、クラスのすべてのオブジェクトで共有される静的データメンバ **(static data member)** について学びます。

問題提起：オブジェクトへの一意なIDの割り当て

生成される各オブジェクトに、一意の識別番号（例：1, 2, 3, ...）を割り当てたいとします。もし各オブジェクトが自身のデータメンバ（例えば `id_no`）しか持たない場合、他にどれだけのオブジェクトが生成されたかを知ることができず、自身の番号を決定できません。

この問題を解決するためには、すべてのオブジェクトで共有される情報、つまりオブジェクトの総生成数を記録するカウンタが必要です。

`static`キーワード：クラスレベルの共有データ

C++は、この機能を実現するために静的データメンバ(static data member)を提供します。静的データメンバは、個々のオブジェクトに属するのではなく、クラス自体に属します。オブジェクトがいくつ生成されても（たとえ一つも生成されなくても）、静的データメンバはただ一つのインスタンスしか存在しません。
例：

```C/C++
class IdNo {
    static int counter; // 静的データメンバ：カウント用の共有変数
    int id_no;          // 非静的データメンバ：各オブジェクト固有の識別番号
};
```

`static`メンバ`counter`は一つだけで、オブジェクト`a`と`b`で共有される。一方、`id_no`はそれぞれ独立している。

静的メンバの宣言と定義

静的データメンバの宣言と定義は分離されています。

- 宣言 **(Declaration)**: クラス定義の内部で、`static`キーワードを用いて宣言します。
- 定義 **(Definition)**: クラス定義の外部（通常は対応する`.cpp`ソースファイル内）で、静的データメンバを定義し、初期化する必要があります。

重要: クラスの外部で静的メンバを定義し忘れると、リンク段階でエラーが発生します。

---

2. IDジェネレータクラスの実装

それでは、新しいオブジェクトに一意のIDを自動的に割り当てる`IdNo`クラスを実装してみましょう。

`IdNo.h` (`List 13-1`)
ヘッダファイルで、クラスとそのメンバを宣言します。

```C/C++
// List 13-1: IdNo.h
#ifndef ___Class_IdNo
#define ___Class_IdNo

class IdNo {
    static int counter; // 静的データメンバの宣言
    int id_no;          // 非静的データメンバ：各オブジェクトの識別番号

public:
    IdNo(); // コンストラクタ
    int id() const; // IDを取得するメンバ関数
};

#endif
```

`IdNo.cpp` (`List 13-2`)
ソースファイルで、静的メンバとクラスのメソッドを定義します。

```C/C++
// List 13-2: IdNo.cpp
#include "IdNo.h"

// 静的データメンバの定義と初期化
// クラスの外部で静的メンバを定義し忘れると、リンク段階でエラーが発生します。
int IdNo::counter = 0;
```

```
// コンストラクタ
IdNo::IdNo() {
    id_no = ++counter; // 共有カウンタをインクリメントし、オブジェクトの識別番号に割り当てる
}

// 識別番号の取得
int IdNo::id() const {
    return id_no;
}
```

コード解説:

- `int IdNo::counter = 0;`: これは静的メンバ`counter`の定義です。クラス名とスコープ解決演算子`::`を用いて、それが属するクラスを指定し、0に初期化します。
- `IdNo::IdNo()`: コンストラクタでは、新しい`IdNo`オブジェクトが生成されるたびに、共有の`counter`がインクリメントされ、その値が新しいオブジェクトの`id_no`に割り当てられます。これにより、各オブジェクトがユニークで連番の識別番号を持つことが保証されます。

テストプログラム **IdNoTest.cpp** (**List 13-3**)
このプログラムは`IdNo`クラスの機能を検証します。

```C/C++
// List 13-3: IdNoTest.cpp
#include <iostream>
#include "IdNo.h"

using namespace std;

int main()
{
    IdNo a;      // 識別番号1番
    IdNo b;      // 識別番号2番
    IdNo c[4];   // 識別番号3番～6番

    cout << "aの識別番号：" << a.id() << '\n';
    cout << "bの識別番号：" << b.id() << '\n';
    for (int i = 0; i < 4; i++)
        cout << "c[" << i << "]の識別番号：" << c[i].id() << '\n';
}
```

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

プログラム出力:

```None
aの識別番号：1
bの識別番号：2
c[0]の識別番号：3
c[1]の識別番号：4
c[2]の識別番号：5
c[3]の識別番号：6
```

出力結果から、オブジェクトが生成されるたびに、共有の`counter`から一意の連番が割り当てられていることがわかります。

---

3. 静的データメンバへのアクセス

静的データメンバが`public`であれば、クラスの外部からアクセスできます。アクセス方法は2種類あります。

- クラス名によるアクセス

これは最も直接的で明確な方法で、そのメンバが特定のオブジェクトではなくクラスに属することを示します。
`ClassName::StaticMemberName`

- オブジェクト名によるアクセス

クラスのオブジェクトを介して静的メンバにアクセスすることも可能です。
`objectName.StaticMemberName`

この方法は可能ですが、メンバがオブジェクト固有のものであるかのような誤解を招きやすいです。したがって、クラス名によるアクセスを推奨します。

検証例

`IdNo`というクラスを用いて、これら2つのアクセス方法を検証します。

C/C++

```
// List 13-4: IdNo.h
#ifndef ___Class_IdNo
#define ___Class_IdNo

class IdNo {
    int id_no;          // 非静的データメンバ：各オブジェクトの識別番号

public:
    static int counter; // <== 静的データメンバの宣言 (private から public に変更)

    IdNo(); // コンストラクタ
    int id() const; // IDを取得するメンバ関数
};

#endif
```

C/C++

```
// List 13-5: IdNoTest.cpp
#include <iostream>
#include "IdNo.h"

using namespace std;

int main()
{
    IdNo a;    // 識別番号1番
    IdNo b;    // 識別番号2番
    IdNo c[4]; // 識別番号3番～6番

    cout << "aの識別番号：" << a.id() << '\n';
    cout << "bの識別番号：" << b.id() << '\n';
    for (int i = 0; i < 4; i++)
        cout << "c[" << i << "]の識別番号：" << c[i].id() << '\n';

    cout << "IdNo::counter = " << IdNo::counter << '\n';
    cout << "a.counter = " << a.counter << '\n';
    cout << "b.counter = " << b.counter << '\n';
}
```

プログラム出力:

```
None
aの識別番号：1
bの識別番号：2
c[0]の識別番号：3
c[1]の識別番号：4
c[2]の識別番号：5
c[3]の識別番号：6
IdNo::counter = 6
a.counter = 6
b.counter = 6
```

---

4. 静的メンバ関数

これまでに、クラスに属しオブジェクトには属さない共有データである静的データメンバを学びました。本節ではさらに、静的メンバ関数 **(static member function)** について学習します。

静的メンバ関数：クラスレベルの操作

静的メンバ関数は、特定のオブジェクトに束縛されません。`this`ポインタを持たないため、クラスの非静的データメンバや非静的メンバ関数に直接アクセスすることはできません。

- 主に静的データメンバを操作したり、クラス自体に関連するが特定のオブジェクトの状態を必要としない操作を実行したりするために使用されます。
- 静的メンバ関数は、クラス定義内で`static`キーワードを使って宣言しますが、クラス定義外で実装する際には`static`キーワードは不要です。
- 静的メンバ関数と非静的メンバ関数は、それらの引数リスト（関数シグネチャ）が異なれば、同じ名前を持つことができます。

非`static`メンバ関数：オブジェクトレベルの操作

非静的メンバ関数は、クラスの特定のオブジェクトに束縛されます。`this`ポインタを持ち、クラスのすべてのデータメンバ（静的および非静的）とすべてのメンバ関数（静的および非静的）にアクセスできます。主にオブジェクト自身の状態を操作するために使用されます。

静的メンバ関数のさらなる応用

以前の`IdNo`クラスに戻り、これまでに割り当てられた最大のIDを取得する機能を追加してみましょう。この新しい機能`get_max_id`は、静的データメンバ`counter`にのみ依存するため、静的メンバ関数として実装するのに最適です。

`IdNo`クラスの第2版

C/C++

```
// List 13-10: IdNo2.h
#ifndef ___Class_IdNo2
#define ___Class_IdNo2

class IdNo2
{
    static int counter; // 静的データメンバの宣言
    int id_no;          // 非静的データメンバ：各オブジェクトの識別番号

public:
    IdNo2();
    int id() const;
    static int get_max_id(); // 最大IDを取得する静的メンバ関数
};
#endif
```

C/C++

```
// List 13-11: IdNo2.cpp
#include "IdNo2.h"
int IdNo2::counter = 0; // 静的データメンバの定義と初期化
IdNo2::IdNo2()
{
    id_no = ++counter;
}
int IdNo2::id() const
{
    return id_no;
}

// 静的メンバ関数は、クラス定義内で static キーワードを使って宣言しますが、
// クラス定義外で実装する際には staticキーワードは不要です。
int IdNo2::get_max_id()
{
    return counter;
}
```

C/C++

```
// List 13-12: IdNo2Test.cpp
#include <iostream>
#include "IdNo2.h"
```

```
using namespace std;

int main()
{
    IdNo2 a;    // 識別番号1番
    IdNo2 b;    // 識別番号2番
    IdNo2 c[4]; // 識別番号3番〜6番

    cout << "aの識別番号：" << a.id() << '\n';
    cout << "bの識別番号：" << b.id() << '\n';
    for (int i = 0; i < 4; i++)
        cout << "c[" << i << "]の識別番号：" << c[i].id() << '\n';

    cout << "割り当て済みの最大識別番号：" << IdNo2::get_max_id() << '\n';
}
```

```
Shell
g++ IdNo2Test.cpp IdNo2.cpp -o IdNo2Test; if($?) { .\IdNo2Test}
```

プログラム出力:

```
None
aの識別番号：1
bの識別番号：2
c[0]の識別番号：3
c[1]の識別番号：4
c[2]の識別番号：5
c[3]の識別番号：6
割り当て済みの最大識別番号：6
```

`get_max_id`関数は特定のオブジェクトに依存せず、クラスレベルの`counter`にのみ関連するため、静的メンバ関数として正しく設計されています。

---

5. 静的メンバの応用例：シングルトン・パターン

静的メンバの最も強力な応用例の一つに、シングルトン・パターン **(Singleton Pattern)** があります。これは、ソフトウェア設計で頻繁に用いられるデザインパターンの一つです。

デザインパターンとは、ソフトウェア開発で繰り返し現れる典型的な設計上の課題に対する、再利用可能な解決策の「型」です。実装の細部ではなく、クラスやオブジェクトの役割分担・構造を整理し、保守性や拡張性を高めることを目的とします。

シングルトン・パターンとは？

シングルトン・パターンは、あるクラスのインスタンスがプログラム全体で一つしか存在しないことを保証し、その唯一のインスタンスへのグローバルなアクセスポイントを提供するデザインパターンです。

例えば、プログラム全体で共有すべき設定情報、データベース接続、ログ管理機能など、複数存在すると逆に問題を引き起こすようなオブジェクトに適用されます。

静的メンバを用いた実装

静的メンバは、このパターンを実現するための鍵となります。

C/C++

```
// SingletonTest.cpp
#include <iostream>
#include <string>

// データベース接続を管理するシングルトンクラス
class Database {
public:
    // 唯一のインスタンスを取得するためのグローバルなアクセスポイント
    static Database& getInstance() {
        // 静的関数内の静的変数としてインスタンスを宣言
        // このコードは最初に getInstance() が呼ばれたときに一度だけ実行される
        static Database instance;
        return instance;
    }

    // クラスが提供するビジネス機能
    void query(std::string sql) {
        std::cout << "SQL実行: " << sql << std::endl;
    }

private:
    // --- 【重要ポイント1】コンストラクタを private にする ---
    // これにより、外部からのインスタンス生成を防ぐ（例: Database db; はコンパイルエラーになる）
    Database() {
        std::cout << ">>> 初期化処理：データベースへの接続を確立中..." << std::endl;
        // 実際の接続処理などをここに記述
    }
```

```
    // --- デストラクタ ---
    // プログラム終了時に自動的に呼ばれ、リソースを解放する
    ~Database() {
        std::cout << ">>> クリーンアップ処理：データベース接続を閉じ、リソースを解放中...
" << std::endl;
        // 実際の切断処理などをここに記述
    }

    // --- 【重要ポイント2】コピーと代入を禁止する ---
    // これにより、唯一のインスタンスがコピーされるのを防ぐ
    Database(const Database&) = delete;
    Database& operator=(const Database&) = delete;
};

int main() {
    std::cout << "プログラム開始..." << std::endl;

    // 1回目の呼び出し：ここで初めてインスタンスが生成され、コンストラクタが呼ばれる
    std::cout << "1回目のgetInstance()呼び出し\n";
    Database::getInstance().query("SELECT * FROM users");

    // 2回目の呼び出し：インスタンスはすでに存在するため、コンストラクタは呼ばれない
    std::cout << "\n2回目のgetInstance()呼び出し\n";
    Database::getInstance().query("UPDATE users SET name='Tom'");

    std::cout << "\nプログラム終了..." << std::endl;
    return 0;
} // <--- main関数が終了すると、静的インスタンスが破棄され、デストラクタが自動的に呼ばれる
```

ポイント解説

1. なぜ **`getInstance()`** を使うのか？

    - `Database db;` のようにオブジェクトを自由に作成できてしまうと、「インスタンスは一つだけ」というルールが破られてしまいます。
    - そこで、コンストラクタを `private` にして外部から直接オブジェクトを作れないようにします。
    - その代わり、唯一の公式な窓口として `public` な静的メンバ関数 `getInstance()` を用意します。オブジェクトが欲しければ、必ずこの関数を通す必要があります。

2. **`static Database instance;`** の魔法

    - `getInstance()` 関数の中にある `static Database instance;` がこのパターンの核心です。

- 関数内で宣言された`static`変数は、その関数が最初に呼ばれたときに一度だけ生成・初期化され、プログラムが終了するまで存在し続けます。
- 2回目以降に `getInstance()` を呼んでも、新しい `instance` は作られず、最初に作られたものがそのまま返されます。これにより、インスタンスが一つだけであることが保証されます。

3. コピーの禁止 (**`= delete;`**)
   C++11から導入された **`= delete`** 指定子は、コンパイラが自動生成する関数（デフォルトコンストラクタ、コピーコンストラクタ、代入演算子など）を明示的に禁止するために使用されます。シングルトンパターンにおいては、インスタンスの「唯一性」を保証するための極めて重要な機能です。
   - もし `Database db2 = Database::getInstance();` のようにオブジェクトをコピーできてしまうと、やはりインスタンスが複数になってしまいます。
   - それを防ぐために、コピーコンストラクタと代入演算子を `= delete;` と記述して、コンパイラに「これらの操作は禁止」と明示的に伝えます。

実行結果と流れ

```
None
プログラム開始...
1回目のgetInstance()呼び出し
>>> 初期化処理：データベースへの接続を確立中...
SQL実行: SELECT * FROM users

2回目のgetInstance()呼び出し
SQL実行: UPDATE users SET name='Tom'

プログラム終了...
>>> クリーンアップ処理：データベース接続を閉じ、リソースを解放中...
```

上記の実行結果から、`getInstance()`が最初に呼ばれたときだけ「初期化処理」が走り、2回目には走らないことがわかります。そして、`main`関数が終了する直前に「クリーンアップ処理」が自動的に実行され、後片付けが行われています。これがシングルトン・パターンの基本的な動作です。

13.2 標準テンプレートライブラリ（STL）

これまではクラスを自作してきましたが、C++には非常に強力で便利な部品群が予め用意されています。それが標準テンプレートライブラリ（**Standard Template Library, STL**）です。本節では、その中でも特に利用頻度の高いコンテナについて紹介します。

STLは、C++標準ライブラリの一部で、テンプレートという技術を核とした汎用的なコンポーネントの集まりです。一言で言えば、「よく使われるデータ構造とアルゴリズムをまとめた道具箱」のようなものです。

STLの主な利点：

- ジェネリック（汎用性）: 同じコンテナやアルゴリズムを、`int`、`double`、自作クラスなど、様々なデータ型に対して再利用できます。
- 高性能: 多くの実装は長年にわたって最適化されており、効率的です。
- 安全な自動メモリ管理（**RAII**）：`std::vector` や `std::string` などのコンテナは必要に応じて動的にメモリを確保・解放し、スコープを抜けると自動的に後始末されます。
- 組み合わせ可能: 「コンテナ」「イテレータ」「アルゴリズム」という部品を自由に組み合わせることで、複雑な処理を簡潔に記述できます。

1. STLの三大要素

STLは主に以下の3つの要素から構成されます。

(1) コンテナ（Containers）

データを格納するための「入れ物」です。格納方式によっていくつかの種類に分かれます。

- シーケンスコンテナ（順序付きで格納）
    - `std::vector`: 動的配列。ランダムアクセスがO(1)と高速。末尾への追加も効率的。
    - `std::list`: 双方向リスト。どの位置でも高速に挿入・削除が可能ですが、ランダムアクセスは苦手。
    - `std::deque`: 両端キュー。先頭と末尾の両方で高速に追加・削除が可能。
- 連想コンテナ（キーに基づいて自動でソート）
    - `std::map` / `std::set`: 通常、赤黒木というデータ構造で実装されており、要素は常にキーでソートされています。検索、挿入、削除はO(log n)です。
- 非順序連想コンテナ（ハッシュテーブルを利用）
    - `std::unordered_map` / `std::unordered_set`: 要素の順序は保証されませんが、平均的にO(1)で高速に検索、挿入、削除が可能です。
- コンテナアダプタ

- `std::stack`, `std::queue`, `std::priority_queue`: 既存のコンテナ（`vector`や`deque`など）のインターフェースを制限し、特定の機能（スタック、キューなど）を提供します。

(2) イテレータ（Iterators）

イテレータは、コンテナの要素を指し示し、その上を移動するためのオブジェクトです。ポインタのような役割を果たし、コンテナとアルゴリズムの橋渡しをします。

`std::sort`や`std::find`といったアルゴリズムが、`vector`や`list`などコンテナの種類を問わず動作できるのは、このイテレータという共通のインターフェースのおかげです。

(3) アルゴリズム（Algorithms）

ソート、検索、データ変換など、コンテナ上の要素に対して行う様々な操作を提供します。ほとんどは`<algorithm>`ヘッダで定義されています。

- 特徴: アルゴリズムはコンテナそのものではなく、イテレータが指す範囲（例：`begin()`から`end()`まで）に対して処理を行います。
- よく使われる例:
  - 検索: `find`, `binary_search`
  - ソート: `sort`, `stable_sort`, `reverse`
  - 計数・集計: `count`, `accumulate`
  - 条件判定: `all_of`, `any_of`, `transform`

---

2. C++ コンテナクラス: `vector`

`std::vector`は、最も基本的で最もよく使われるコンテナです。「可変長の配列」と考えると分かりやすいでしょう。

- 特性: **`vector`**の要素は、メモリ上で連続して配置されます。これにより、インデックスを使ったアクセスが非常に高速です。
- ヘッダファイル: `#include <vector>`

利点:
- 動的にサイズが変更できる
- ランダムアクセスが高速 (`v[i]`)
- メモリが連続しているためキャッシュに乗りやすく、走査が速い

欠点:
- 中間への挿入や削除が遅い (後続要素の移動が発生)

- 容量拡張時に全要素の移動が発生することがある
- 容量拡張や挿入・削除でイテレータが無効になることがある

よく使われるメンバ関数

関数	説明
`push_back(val)`	末尾に要素を追加する
`pop_back()`	末尾の要素を削除する
`at(pos)`	指定位置の要素を返す（境界チェックあり）
`operator[]`	指定位置の要素を返す（境界チェックなし）
`front()`	最初の要素を返す
`back()`	最後の要素を返す
`size()`	現在の要素数を返す
`capacity()`	現在確保されている容量を返す
`reserve(n)`	少なくともn個の要素を格納できるメモリを予約する
`resize(n)`	要素数をnに変更する
`clear()`	全ての要素を削除する
`insert(it, val)`	イテレータ it が指す位置（の直前）に要素 val を挿入する
`erase(it)`	イテレータ it が指す要素を削除し、削除した要素の次を指すイテレータを返す
`begin()` / `end()`	先頭/末尾の次を指すイテレータを返す

使用例

1. 基本的な操作

```C/C++
// VectorTest1.cpp
#include <iostream>
#include <vector>
```

```cpp
int main() {
    // 初期化
    std::vector<double> vec = {1.2, 2.0, 3.5, 4, 5};

    // forループと operator[] を使って全要素を出力
    std::cout << "Vectorの要素: ";
    for (std::size_t i = 0; i < vec.size(); ++i) {
        std::cout << vec[i] << " ";
    }
    std::cout << std::endl;

    // 先頭と末尾の要素にアクセス
    std::cout << "最初の要素: " << vec.front() << std::endl;
    std::cout << "最後の要素: " << vec.back() << std::endl;

    // サイズを10に変更（新しい要素は0で初期化される）
    vec.resize(10);
    std::cout << "リサイズ後のVectorの要素: ";
    for (int i = 0; i < vec.size(); ++i) {
        std::cout << vec[i] << " ";
    }
    std::cout << std::endl;
}
```

このコードの最後で手動で（`delete` などにより）リソースを解放する必要がないのは、`std::vector` が RAII（Resource Acquisition Is Initialization）に基づく資源管理を行っているためです。

- `std::vector` は要素を保持するために内部でヒープ領域に動的メモリを確保しますが、そのメモリは vector オブジェクト自身が管理します。
- `main` の実行が終わると、ローカル変数 `vec` はスコープを抜け、コンパイラによって `vec` のデストラクタ が自動的に呼び出されます。
- このデストラクタが、内部で確保した動的メモリを自動的に解放し（同時に要素も破棄し）ます。

そのため、`std::vector`（および `std::list` などの標準コンテナ）を使用する場合、通常は内部資源を手動で解放する必要はなく、むしろ手動で解放しようとすべきではありません。これは、**STL** コンテナが提供する「安全な自動メモリ管理」の重要な利点の一つです。

2. 動的な要素の追加と削除

C/C++
```cpp
// VectorTest2.cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec;
    std::cout << "初期容量: " << vec.capacity() << std::endl;

    // 末尾に要素を追加
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);

    std::cout << "現在の要素数: " << vec.size() << std::endl;
    std::cout << "現在の容量: " << vec.capacity() << std::endl;

    // 末尾の要素を削除
    vec.pop_back();
    std::cout << "pop_back後の要素数: " << vec.size() << std::endl;

    // 全要素を出力
    std::cout << "Vectorの要素: ";
    for (int i = 0; i < vec.size(); ++i)
    {
        std::cout << vec[i] << " ";
    }
    std::cout << std::endl;
}
```

3. 境界チェックと安全なアクセス

C/C++
```cpp
// VectorTest3.cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vec = {1, 2, 3};
```

```
    // 安全なアクセス (at)
    vec.at(2) = 100;
    std::cout << "vec.at(2) の値: " << vec.at(2) << std::endl; // 正常に出力
    std::cout << "vec.at(5) にアクセス..." << std::endl;
    std::cout << vec.at(5) << std::endl; // 範囲外のため、例外をスロー

    // operator[] は境界チェックを行わないため、範囲外アクセスは未定義動作を引き起こす（危険）
    // std::cout << vec[5] << std::endl;
}
```

- operator[] は境界チェックを行わないため、範囲外アクセスをすると未定義動作になります。
- 一方、at() は範囲チェックを行い、範囲外アクセスの場合は例外（`std::out_of_range`）を送出します。

---

3. C++ コンテナクラス: `list`

`std::list`は、双方向リンクリストで実装されたコンテナです。`vector`とは対照的に、要素はメモリ上で離散的に配置され、各要素が前後の要素へのポインタを持っています。

- 基本特性: どの位置でも要素の挿入・削除が高速に行えます。しかし、特定の位置へのランダムアクセスはできません。
- ヘッダファイル: `#include <list>`

利点:

- 挿入・削除位置のイテレータが既に得られている場合、挿入・削除が高速（O(1)）
- 容量の拡張という概念がない

欠点:

- 特定の要素へのランダムアクセスが遅い (O(n))
- 各要素がポインタを持つため、`vector`よりメモリ消費が大きい
- メモリが連続していないため、キャッシュ効率は`vector`に劣る

よく使われるメンバ関数

関数	説明
`push_front(val)`	先頭に要素を追加する
`pop_front()`	先頭の要素を削除する

関数	説明
`push_back(val)`	末尾に要素を追加する
`pop_back()`	末尾の要素を削除する
`sort()`	要素をソートする
`remove(val)`	指定された値を持つ全ての要素を削除する
`unique()`	隣接する重複要素を削除する（要ソート）
`merge(other)`	ソート済みの別のリストをマージする
`reverse()`	要素の順序を反転させる

使用例

1. 基本的な操作

```C/C++
// ListTest1.cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> lst = {10, 20, 30};

    // 先頭と末尾に要素を追加・削除
    lst.push_front(5);   // {5, 10, 20, 30}
    lst.push_back(40);   // {5, 10, 20, 30, 40}
    lst.pop_front();     // {10, 20, 30, 40}
    lst.pop_back();      // {10, 20, 30}

    // イテレータを使って全要素を出力
    std::cout << "Listの要素: ";
    for (auto it = lst.begin(); it != lst.end(); ++it) {
        std::cout << *it << " "; // *it でイテレータが指す要素の値を取得
    }
    std::cout << std::endl;
}
```

it は std::list<int> の イテレータ（**iterator**）で、イメージとしては「連結リストのあるノード（要素）を指すポインタのようなもの」です。

- lst.begin()：先頭要素を指すイテレータを返します。
- lst.end()：末尾の次（終端）を指すイテレータを返します（有効な要素は指さず、ループ終了判定に使います）。
- ++it：イテレータを次のノード（要素）へ進めます。
- *it：イテレータを参照（デリファレンス）して、指している要素の値（ここでは int）を取得します。

なお、std::list は連結リストなので ランダムアクセスができません。そのため、イテレータは `++it` / `--it` で順に移動するのが基本で、`it + 5` のような操作はできません。

イテレータを使って全要素を出力: `lst.begin()`は先頭要素を指すイテレータを返し、`lst.end()`は「末尾の次」（有効な要素を指さない）を指すイテレータを返します。`++it`でイテレータを次に進め、`*it`でその場所の要素の値を得ます。`list`は**`list[i]`**のようなアクセスができないため、このイテレータによる走査が基本となります。

`auto` は C++ の 型推論のためのキーワードで、右辺の式からコンパイラが変数の型を自動的に決定します。

`auto it = lst.begin();`
は、次の宣言と等価です（lst が std::list<int> の場合）：
`std::list<int>::iterator it = lst.begin();`

2. 特定の位置への要素の挿入と削除

C/C++

```
// ListTest2.cpp
#include <iostream>
#include <list>

int main()
{
    std::list<int> lst = {1, 2, 3, 4, 5};
    auto it = lst.begin();

    // イテレータを3番目の要素（値は3）まで進める
    std::advance(it, 2);

    // 3番目の要素の前に10を挿入
    lst.insert(it, 10); // {1, 2, 10, 3, 4, 5}

    // it は依然として要素`3`を指している。これを削除する。
    // eraseは削除された要素の次の要素を指すイテレータを返す
    it = lst.erase(it); // {1, 2, 10, 4, 5}

    // list内の「5の倍数」をすべて削除する
```

```
    for (auto it = lst.begin(); it != lst.end();)
    {
        if ((*it) % 5 == 0)
        {
            // eraseは削除された要素の次の要素を指すイテレータを返す
            it = lst.erase(it);
        }
        else
        {
            ++it;
        }
    }

    // 出力
    std::cout << "編集後のListの要素: ";
    for (auto it = lst.begin(); it != lst.end(); ++it)
    {
        std::cout << *it << " "; // *it でイテレータが指す要素の値を取得
    }
    std::cout << std::endl;
}
```

3. ソートと重複削除

C/C++

```
// ListTest3.cpp
#include <iostream>
#include <list>

int main() {
    std::list<int> lst = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    // まずソートして、同じ値の要素を隣接させる
    lst.sort();
    std::cout << "ソート後のList: ";
    for (int element : lst) {
        std::cout << element << " ";
    }
    std::cout << std::endl;

    // 隣接する重複要素を削除
    lst.unique();

    // 出力
    std::cout << "ソートして重複削除したList: ";
```

```
    for (int element : lst) {
        std::cout << element << " ";
    }
    std::cout << std::endl;
}
```

注意: `unique()`メンバ関数は、隣接している重複要素のみを削除します。そのため、リスト全体の重複をなくしたい場合は、先に`sort()`を呼び出して同じ値の要素を隣り合わせにしておく必要があります。

---

4. C++ コンテナクラス: `map`

`std::map`は、キー**(key)**と値**(value)**のペアを格納する連想コンテナです。辞書のように、キーを使って高速に対応する値を検索できます。

- 基本特性: `map`の内部では、要素は常にキーによって自動的にソートされています。これにより、順序だったアクセスと高速な検索が可能になります。
- ヘッダファイル: `#include <map>`

利点:

- キーによる検索、挿入、削除が高速 (O(log n))
- 要素は常にキーでソートされている
- キーは一意であることが保証される

欠点:

- `operator[]`はキーが存在しない場合に自動で要素を生成するため注意が必要
- `vector`のようなインデックスによる直接アクセスはできない

よく使われるメンバ関数

関数	説明
`insert({key, val})`	キーと値のペアを挿入する
`erase(key)`	指定されたキーを持つ要素を削除する

関数	説明
`find(key)`	キーを検索し、その要素を指すイテレータを返す（見つからなければ`end()`）
`at(key)`	キーに対応する値を返す（キーが存在しない場合、例外をスロー）
`operator[](key)`	キーに対応する値を返す（キーが存在しない場合、デフォルト値で要素を生成）
`count(key)`	指定されたキーを持つ要素の数を返す（`map`では0か1）
`size()`	要素数を返す
`clear()`	全ての要素を削除する

使用例

C/C++

```cpp
// MapTest.cpp
#include <iostream>
#include <map>
#include <string>

int main()
{
    // string型をキー、int型を値とするmapを生成
    std::map<std::string, int> scores;

    // operator[] を使った挿入
    scores["Alice"] = 90;
    scores["Bob"] = 85;

    // insertを使った挿入
    scores.insert({"Charlie", 92});

    // イテレータを使った走査（キーでソートされている順に出力される）
    std::cout << "全員のスコア:\n";
    for (auto it = scores.begin(); it != scores.end(); ++it)
    {
        // it->first  : キー（ここでは名前）
        // it->second : 値（ここではスコア）
        std::cout << "  " << it->first << ": " << it->second << std::endl;
    }
```

```
    // findを使った検索
    std::cout << "スコア検索:" << std::endl;
    auto it = scores.find("Bob");
    if (it != scores.end())
    {
        std::cout << "  Bobのスコア: " << it->second << std::endl;
    }

    // eraseを使った削除
    scores.erase("Alice");
    std::cout << "削除後の要素数: " << scores.size() << std::endl;
}
```

at()と**operator[]**の違い（注意点）:

- scores.at("David"): scoresに"David"というキーが存在しない場合、プログラムは例外を投げて停止します。安全なアクセス方法です。
- scores["David"]: scoresに"David"というキーが存在しない場合、自動的に**"David"**というキーとデフォルト値（**int**なら**0**）を持つ新しい要素が**map**に挿入されます。意図しない要素の追加に繋がることがあるため、値の参照だけが目的の場合はfindやatの使用が推奨されます。