C++の標準ライブラリコンテナ

C++の標準ライブラリには、データを効率的に格納して操作できるさまざまなコンテナが用意されています。コンテナは、オブジェクトのコレクションを格納して管理できるクラスです。各コンテナには固有のプロパティがあり、個々のユースケースに対応できるように設計されています。

C++の標準ライブラリコンテナは、主に次の3つに分類できます。

- シーケンスコンテナ
- 連想コンテナ
- 非順序連想コンテナ

シーケンスコンテナ

シーケンスコンテナには、線形シーケンス(直線的配列)で要素が格納されます。各要素に高速でアクセスできるほか、要素の追加と削除を効率的に実行できます。主なシーケンスコンテナは次のとおりです。

- `std::vector`: サイズを拡大、縮小できる動的な配列です。高速なランダムアクセスが可能で、メモリを効率的に使用します。
- `std::list`: 双方向連結リストです。リスト内の任意の位置に迅速に要素を追加、削除できます。
- `std::deque`: 両端キューです。コンテナの両端で迅速に要素を追加、削除できます。

連想コンテナ

連想コンテナにはソートされた状態で要素が格納されるため、迅速に検索、追加、削除できます。要素はキーでソートされ、各要素には一意のキーがあります。主な連想コンテナは次のとおりです。

- `std::set`: 一意のキーのコレクションです。キーでソートされます。
- `std::map`: キーと値のペアのコレクションです。一意のキーを持ち、キーでソートされます。
- `std::multiset`: キーのコレクションです。キーでソートされており、同じキーを複数格納できます。
- `std::multimap`: キーと値のペアのコレクションです。キーでソートされており、同じキーを持つキーと値のペアを複数格納できます。

非順序連想コンテナ

非順序連想コンテナには、ソートされていない状態で要素が格納されます。ハッシュ関数を使用して各要素に高速でアクセスできます。主な非順序連想コンテナは次のとおりです。

- `std::unordered_set`: 一意のキーのコレクションです。キーでハッシュ化されています。
- `std::unordered_map`: キーと値のペアのコレクションです。一意のキーを持ち、キーでハッシュ化されています。
- `std::unordered_multiset`: キーのコレクションです。キーでハッシュ化されており、同じキーを複数格納できます。
- `std::unordered_multimap`: キーと値のペアのコレクションです。キーでハッシュ化されており、同じキーを持つキーと値のペアを複数格納できます。

コンテナの使用

コンテナは、ユーザー定義型を含む任意のデータ型で使用できます。提供されているメンバ関数を使用して、コンテナ内のデータを操作できます。また、イテレータを使用してコンテナ内の要素を走査できます。

std::vectorを使用して整数値を格納、操作する例を次に示します。

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers;

    for (int i = 1; i <= 5; ++i) {
        numbers.push_back(i);
    }

    for (size_t i = 0; i < numbers.size(); ++i) {
        std::cout << numbers[i] << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

出力:

```
1 2 3 4 5
```

この簡単な例では、整数のベクターを作成する方法、`push_back()`を使用してベクターに要素を追加する方法、配列インデックス演算子`[]`を使用して要素にアクセスする方法を示しています。

`std::map`を使用して、単語とその頻度のリストを格納して操作する例を次に示します。

```
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, int> word_frequency;

    word_frequency["apple"] = 3;
    word_frequency["banana"] = 2;
    word_frequency["orange"] = 1;

    std::cout << "Frequency of 'apple': " << word_frequency["apple"] <<
'\n';

    for (const auto &entry : word_frequency) {
        std::cout << entry.first << ": " << entry.second << '\n';
```

```
    }

    return 0;
}
```

出力:

```
Frequency of 'apple': 3
apple: 3
banana: 2
orange: 1
```

この例では、文字列(単語)と整数(頻度)のマップを作成する方法、キーと値のペアをマップに追加する方法、キーを使用して値にアクセスする方法、forループで範囲を指定してマップを反復処理する方法を示しています。

適切なコンテナの選択

特定のユースケースに最適なコンテナを選択するには、コンテナの各種類の長所と短所を理解することが重要です。ここでは、さまざまなシナリオについて調べ、最適なコンテナを判断します。

両端での迅速な追加と削除

シナリオ: 受信した要求のリストを保持するシンプルなWebサーバーを構築しているとします。サーバーは先入れ先出し(FIFO)方式で要求を処理し、新しい要求はキューの末尾に追加されます。

ソリューション: `std::deque`はキューの末尾への追加と先頭からの取り出しを迅速に実行できるため、このユースケースに最適なコンテナです。

```
#include <iostream>
#include <deque>

int main() {
    std::deque<std::string> request_queue;

    request_queue.push_back("Request 1");
    request_queue.push_back("Request 2");
    request_queue.push_back("Request 3");

    while (!request_queue.empty()) {
        std::cout << "Processing: " << request_queue.front() << '\n';
        request_queue.pop_front();
    }

    return 0;
}
```

効率的なソートと重複の削除

シナリオ: サイズの大きいテキストファイルを分析し、ファイルに含まれる単語(重複は削除)をソートしたリストを作るプログラムを作成しているとします。

ソリューション: `std::set`は要素を効率的にソートして重複を削除するため、このユースケースに最適なコンテナです。

```
#include <iostream>
#include <set>
#include <sstream>

int main() {
    std::string text = "apple banana apple orange orange banana grape";
    std::istringstream iss(text);
    std::set<std::string> unique_words;

    for (std::string word; iss >> word;) {
        unique_words.insert(word);
    }

    for (const std::string& word : unique_words) {
        std::cout << word << " ";
    }

    return 0;
}
```

要素への高速なランダムアクセス

シナリオ: センサーデータを分析するアプリケーションを作成しているとします。センサーは一定の間隔でデータポイントを記録します。アプリケーションではこれらのデータポイントの格納、平均などの統計値の計算、特定の位置の値の取得を実行する必要があります。

ソリューション: `std::vector`は効率的かつ連続的にメモリを割り当て、インデックスを使用して各要素に高速でアクセスできます。また、必要に応じて動的にサイズを変更できるため、このユースケースに最適なコンテナです。

```
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>

double calculate_average(const std::vector<double>& data) {
    if (data.empty()) return 0.0;
    double sum = std::accumulate(data.begin(), data.end(), 0.0);
    return sum / data.size();
}

int main() {
```

```
    std::vector<double> sensor_data;

    sensor_data.push_back(20.5);
    sensor_data.push_back(21.3);
    sensor_data.push_back(19.8);
    sensor_data.push_back(22.1);

    double average = calculate_average(sensor_data);
    std::cout << "Average sensor data value: " << average << std::endl;

    double third_data_point = sensor_data[2];
    std::cout << "Third data point: " << third_data_point << std::endl;

    std::sort(sensor_data.begin(), sensor_data.end());

    std::cout << "Sorted sensor data: ";
    for (double data_point : sensor_data) {
        std::cout << data_point << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

シーケンス内の任意の位置への効率的な追加と削除 - std::list

シナリオ: 曲のプレイリストを管理するプログラムを作成しているとします。ユーザーがプレイリスト内の任意の位置に曲を追加したり、削除したりできるようにします。

ソリューション: `std::list`はシーケンス内の任意の位置に効率的に要素を追加、削除できるため、このユースケースに最適なコンテナです。

```
#include <iostream>
#include <list>

int main() {
    std::list<std::string> playlist = {"Song A", "Song B", "Song C"};

    playlist.insert(std::next(playlist.begin()), "Song D");

    playlist.erase(std::next(playlist.begin()));

    for (const std::string& song : playlist) {
        std::cout << song << '\n';
    }

    return 0;
}
```

キーと値の対応付け

シナリオ: 店舗の在庫を管理するアプリケーションを作成しているとします。製品名とそれぞれの在庫数を対応付ける必要があります。

ソリューション: `std::map`は一意のキーを使用してキーと値のペアを効率的に保管できるため、このユースケースに最適なコンテナです。

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> inventory{
        {"Apples", 10},
        {"Oranges", 5},
        {"Bananas", 7}
    };

    inventory["Grapes"] = 15;

    inventory["Apples"] = 12;

    for (const auto& [product, stock] : inventory) {  // structured
bindings
        std::cout << product << ": " << stock << '\n';
    }

    return 0;
}
```

演習

演習1

次のコードの出力はどのようになりますか。

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers{ 10, 20, 30, 40, 50 };

    for (int i = numbers.size() - 1; i >= 0; --i) {
        std::cout << numbers[i] << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

```
A: 10 20 30 40 50

B: 50 40 30 20 10

C: 50 40 30 20

D: Compiler Error
```

演習2

次のコードの出力はどのようになりますか。

```cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    std::map<std::string, int> ages;
    ages["Alice"] = 28;
    ages["Bob"] = 35;
    ages["Charlie"] = 22;

    std::cout << ages["Diana"] << '\n';

    return 0;
}
```

```
A: 0

B: 1

C: -1

D: Compiler Error
```

演習3

整数型の`std::set`を作成し、このセットに1から10の数字を追加するプログラムを作成します。次に、このセットからすべての偶数を削除し、残りの要素を出力します。

演習4

整数型の`std::list`を使用する次のコードについて考えてみましょう。出力はどのようになりますか。

```cpp
#include <iostream>
#include <list>

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

    numbers.insert(iter, 10);
    numbers.reverse();

    for (const auto &num : numbers) {
        std::cout << num << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

```
A: 1 2 3 4 5 10

B: 1 2 10 3 4 5

C: 5 4 3 10 2 1

D: 5 4 10 3 2 1
```

演習5

整数型の`std::deque`を作成し、この両端キューに1から5の数字を追加するプログラムを作成します。次に、この両端キューからすべての奇数を削除し、残りの要素を出力します。

演習6

学生の名前とテストの点数のリストを管理するプログラムを作成しているとします。学生の名前で点数をすばやく検索したり、点数を更新したりできるようにする必要があります。このユースケースに最も適したコンテナはどれですか。

```
A. std::vector

B. std::list

C. std::deque

D. std::set

E. std::map
```