第2课 STL课程

C++ 库的概念

库是指一组**预编译的代码**，这些代码可以被程序重用，而不需要重复编写相同的逻辑。C++ 提供了广泛的标准库，这些库帮助开发者完成许多常见的任务，如输入输出、数学运算、数据结构等。库分为两种主要类型：

1. **静态库**（`.lib` 或 `.a` 文件）：在编译时被链接到程序中，生成的可执行文件会包含库的代码。
2. **动态库**（`.dll`、`.so` 文件）：在运行时被加载，程序只在需要时访问库的代码。

C++ 库的分类

C++ 库通常可以分为以下几类：

1. **标准库（C++ Standard Library）**：包括输入输出（如 `iostream`）、字符串处理（如 `string`）、数学函数（如 `cmath`）、容器类（如 `vector`）、算法（如 `sort`）等。
2. **第三方库**：由其他开发者或公司创建的库，如 Boost、Qt、OpenCV 等。

如何在 C++ 中使用库

使用库的过程主要包括以下几个步骤：

1. **包含头文件**：头文件（`.h` 或 `.hpp`）提供了函数、类、类型的声明。你可以通过 `#include` 来引入头文件。
2. **链接库文件**：编译器需要知道库的代码文件在哪，静态库或动态库的路径需要在编译时指定。

基本步骤

- 步骤1：包含头文件

在代码中使用库的第一步是包含对应的头文件。例如，C++ 的标准库头文件通常以 `<header>` 的形式包含。

```
#include <iostream>
#include <cmath>
```

- 步骤2：编译时链接库

如果你使用的是标准库，大多数情况下编译器会自动链接这些库。如果是第三方库，你可能需要显式指定链接的库。例如，在使用 g++ 时：

```
g++ main.cpp -o main -l<library_name>
```

其中 `-l<library_name>` 用于指定要链接的库。

标准库的使用示例

```
#include <iostream>   // 引入标准输入输出库
#include <cmath>      // 引入数学函数库

int main() {
```

```cpp
    double x = 9.0;

    // 使用标准库函数 sqrt 来计算平方根
    std::cout << "The square root of " << x << " is " << std::sqrt(x) << std::endl;

    return 0;
}
```

- `<iostream>`：提供了 `std::cout` 和 `std::cin` 等输入输出功能。
- `<cmath>`：提供了数学函数 `std::sqrt` 用于计算平方根。

编译这段代码：

```
g++ main.cpp -o main
```

在 C++ 以及其他编程语言中，**库的链接**是指将程序的各个部分（源代码、库文件等）组合在一起，以生成一个完整的可执行文件的过程。链接的本质是**将多个独立编译的代码模块整合在一起**，使得程序能够正确找到并使用它们所依赖的函数、变量、类等符号。

链接的本质

链接的本质是将程序中的**符号引用**（如函数调用、变量访问等）与它们的**定义**关联起来。C++ 程序通常分为多个源文件或模块，每个模块可能会调用其他模块中的函数或使用其他模块中的变量。链接过程确保每个符号都有正确的定义和实现，并将它们整合成一个最终的可执行文件或动态库。

链接通常分为两种：

1. **静态链接**：在编译时将库的代码嵌入到可执行文件中，生成的可执行文件不需要依赖外部的库文件。

2. **动态链接**：在运行时加载外部的库文件（如 `.dll` 或 `.so` 文件），可执行文件只包含库的引用，库的实际代码存储在外部的动态库中。

链接的原理

C++ 的编译过程通常分为几个步骤，其中**链接**是非常重要的一个环节。整个流程可以分为以下几个步骤：

1. **预处理**（Preprocessing）：对源代码进行宏展开、头文件包含等预处理操作。
2. **编译**（Compilation）：将预处理后的 C++ 源代码转换为目标代码（机器码），生成 `.o` 或 `.obj` 文件（目标文件）。
3. **链接**（Linking）：将所有目标文件与库文件进行链接，生成最终的可执行文件。

静态链接与动态链接的区别

（重要！常考！）

静态链接

在静态链接中，库的代码会在**编译时**与程序的目标文件一同链接，生成一个**独立的可执行文件**，该文件包含了所有需要的库代码。因此，静态链接生成的可执行文件通常会比较大，因为它包含了库的所有实现。

工作流程：

1. 编译器将每个源文件编译为目标文件（如 `.o` 文件）。
2. 链接器将目标文件与库文件中的代码进行组合。
3. 最终生成的可执行文件包含了所有程序和库的代码，运行时不再依赖外部库文件。

静态链接的优点：

- 可执行文件不依赖外部库，部署更简单。

- 程序运行时不需要查找和加载库文件，启动速度更快。

静态链接的缺点：

- 生成的可执行文件较大，因为包含了所有库代码。
- 如果库发生了更新，必须重新编译程序以包含更新后的库代码。

动态链接

在动态链接中，库的代码不会被包含到可执行文件中，而是生成对库文件（如 `.dll` 或 `.so` 文件）的引用。在程序运行时，这些库文件会被动态加载到内存中。

工作流程：

4. 编译器将每个源文件编译为目标文件。
5. 链接器将目标文件与动态库的符号表进行链接，生成包含库引用的可执行文件。
6. 程序运行时，操作系统负责加载动态库，将需要的库代码加载到内存中。

动态链接的优点：

- 可执行文件较小，因为不包含库代码。
- 如果库发生更新，无需重新编译程序，只需要替换库文件。
- 多个程序可以共享同一个动态库，节省内存。

动态链接的缺点：

- 程序运行时需要加载动态库，可能会导致程序启动稍慢。
- 如果动态库缺失或版本不匹配，程序可能无法运行。

链接的具体实现

静态链接的工作原理

静态链接在编译期间进行，具体步骤如下：

1. **符号解析**：每个目标文件中可能包含对其他目标文件中定义的函数或变量的引用。静态链接器会解析这些符号，并将它们与目标文件或库文件中定义的符号匹配。
2. **地址分配**：链接器为每个符号分配具体的内存地址，这些地址在程序运行时被用来访问数据或调用函数。
3. **代码和数据合并**：链接器将各个目标文件的代码和数据段合并为一个整体，并处理所有符号的引用。

动态链接的工作原理

动态链接是在程序**运行时**完成的，具体步骤如下：

1. **符号查找**：当程序启动时，操作系统的**动态链接器**（Dynamic Linker）会检查程序的依赖关系，加载所需的动态库。
2. **符号解析**：动态链接器在库中查找程序所需的符号，将它们与程序中的引用关联起来。
3. **内存映射**：动态链接器将动态库映射到程序的内存地址空间中，使得程序可以调用库中的函数或访问库中的变量。

STL库简介

1. STL概述

标准模板库（STL）是C++标准库的一部分，提供了一系列模板类和函数，用于处理数据结构和算法。STL的核心组件包括：

- 容器（Containers）：用于存储数据的各种数据结构。
- 迭代器（Iterators）：类似于指针，用于访问容器中的元素。
- 算法（Algorithms）：各种常见算法，如排序、搜索等。

- 函数对象（Function objects）：可用于算法的泛型函数。

2.优势

通用、高效、可重用的数据结构和算法，以便C++程序员可以更轻松地编写高质量的代码。

通用性：STL的容器和算法可以适用于各种数据类型，使得编写通用代码变得更加容易。**性能**：STL的算法和数据结构经过优化，通常比手动实现的代码更快。

可重用性：STL的组件可以在不同的项目中重复使用，提高了代码的可维护性。

泛型编程

以下是泛型编程的一些关键特点和解释：

1. 通用性：泛型编程的核心思想是编写通用代码，可以处理多种不同类型的数据，而不需要为每种数据类型都编写特定的代码。这使得代码更加灵活和通用。
2. 模板：在C++中，泛型编程主要通过模板来实现。模板是一种将数据类型参数化的方式，允许创建通用的函数、类和数据结构。模板代码可以在编译时根据具体的数据类型生成实际的代码。
3. 代码重用：泛型编程有助于减少代码的重复编写，因为通用代码可以在多个地方重复使用，而不需要为每个特定的数据类型编写新的代码。
4. 性能优化：泛型编程在保持通用性的同时，通常还注重性能优化。通过使用模板元编程技术，可以在编译时进行优化，以提高代码的性能。
5. 容器和算法：STL（Standard Template Library）是C++标准库中的一个泛型编程范例，它提供了一组通用的容器和算法，可用于处理各种数据类型。STL的设计允许用户根据需要选择合适的容器和算法，而无需关心数据类型。

代码示例

```
// 泛型函数模板，适用于任意类型 T
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // 使用模板函数处理不同类型的数据
    std::cout << "整数相加: " << add(3, 5) << std::endl;          // 输出 8
    std::cout << "浮点数相加: " << add(2.5, 4.5) << std::endl;    // 输出 7
    std::cout << "字符相加: " << add('A', 2) << std::endl;        // 输出 'C' (ASCII 运算)
    return 0;
}
```

这段代码是对 函数模板（Function Template）的一种实现，是 C++ 中泛型编程的一个例子。它让你能够用一种通用的方式处理不同类型的数据。以下是对这段代码语法的详细解释：

1. 模板定义：`template <typename T>`

```
1 template <typename T>
```

- `template` 关键字：这是模板定义的开始，它告诉编译器接下来将定义一个模板，可以用于处理任意类型的数据。
- `<typename T>`：`typename` 表示一个类型占位符，而 `T` 是这个占位符的名称。你也可以用 `class` 替代 `typename`，它们在模板参数声明中是等价的：

  ```
  1 template <class T>
  ```

  它们都表示 `T` 可以是任何类型，比如 `int`、`double`、`char` 等。

2. 模板函数定义：`T add(T a, T b)`

```
1 T add(T a, T b) {
2     return a + b;
3 }
```

- `T add(T a, T b)`：这是函数模板定义。`T` 是一个通用类型，在你调用模板函数时，编译器会根据你传递的参数来推导出 `T` 的实际类型。
  - `T a` 和 `T b`：这两个参数都是类型 `T`，可以是任何类型。
  - `return a + b;`：函数返回两个参数 `a` 和 `b` 之和，这个 `+` 运算符在 `T` 类型上要有意义（即，`T` 支持 `+` 操作）。

3. 使用模板函数：`main()` 函数

```cpp
int main() {
    // 使用模板函数处理不同类型的数据
    std::cout << "整数相加: " << add(3, 5) << std::endl;          // 输出 8
    std::cout << "浮点数相加: " << add(2.5, 4.5) << std::endl;    // 输出 7
    std::cout << "字符相加: " << add('A', 2) << std::endl;        // 输出 'C' (ASCII 运算)
    return 0;
}
```

- **`add(3, 5)`**：调用模板函数 `add`，参数为两个 `int` 类型。编译器推导出 `T` 是 `int` 类型，因此 `add<int>(3, 5)` 会被调用，返回 `8`。
- **`add(2.5, 4.5)`**：调用模板函数 `add`，参数为两个 `double` 类型。编译器推导出 `T` 是 `double` 类型，因此 `add<double>(2.5, 4.5)` 会被调用，返回 `7.0`。
- **`add('A', 2)`**：调用模板函数 `add`，参数为一个 `char` 和一个 `int`。编译器推导出 `T` 是 `char` 类型，因此 `add<char>('A', 2)` 会被调用。`'A'` 的 ASCII 值是 `65`，所以返回结果是 `'C'`（`'A' + 2`），输出 `C`。

4. 编译器推导和模板实例化

当你调用 `add()` 时，编译器会根据传入的参数类型推导出模板的类型 `T`，然后实例化出对应类型的 `add` 函数：

- **`add<int>(int, int)`**：用于 `add(3, 5)`，实际执行 `3 + 5`。
- **`add<double>(double, double)`**：用于 `add(2.5, 4.5)`，实际执行 `2.5 + 4.5`。
- **`add<char>(char, char)`**：用于 `add('A', 2)`，实际执行 `'A' + 2`，输出 `'C'`。

- **模板是 C++ 中的泛型编程工具**，允许编写与类型无关的函数。
- **编译器在编译时推导类型**：编译器会根据你传递的参数类型来推导出模板参数 `T`。
- **模板实例化**：模板定义只是一个框架，而模板实例化会根据具体类型生成不同的函数版本。

`T` 在模板中代表相同的类型。所以在函数模板 `add(T a, T b)` 中，参数 `a` 和 `b` 必须是相同类型。因此像 `int + double` 这样的操作是不被允许的，因为它们是不同类型。

如果想要允许不同类型的参数，就需要引入不同的模板参数，比如 `T` 和 `U`，来表示不同类型的参数。下面是一个修改后的例子：

```cpp
// 泛型函数模板，允许不同类型的参数 T 和 U
template <typename T, typename U>
auto add(T a, U b) -> decltype(a + b) {
    return a + b;
}

int main() {
    // 使用模板函数处理不同类型的数据
    std::cout << "整数相加: " << add(3, 5) << std::endl;          // 输出 8
    std::cout << "浮点数相加: " << add(2.5, 4.5) << std::endl;    // 输出 7
    std::cout << "字符相加: " << add('A', 2) << std::endl;        // 输出 'C' (ASCII 运算)
    std::cout << "整数和浮点数相加: " << add(3, 4.5) << std::endl; // 输出 7.5

    return 0;
}
```

解释

1. **`template <typename T, typename U>`** 引入了两个不同的模板参数 `T` 和 `U`，使得参数 `a` 和 `b` 可以是不同类型。

2. **`auto add(T a, U b) -> decltype(a + b)`**
   - 使用 `auto` 关键字来推导返回类型。
   - **`decltype(a + b)`**：`decltype` 用于推导表达式 `a + b` 的结果类型。这样模板函数的返回类型将根据 `a + b` 的实际类型来确定。

因此，`template <typename T>` 可以让你编写更通用的函数，能够处理任意类型，而不需要重复编写类似功能的代码。

STL容器

STL容器分为序列容器和关联容器两大类，以及容器适配器。

a. 序列容器

- **vector（向量）**：动态数组，支持快速随机访问。
- **list（链表）**：双向链表，支持快速插入和删除。
- **deque（双端队列）**：支持两端的快速插入和删除。

b. 关联容器

- **set（集合）**：存储唯一元素，按特定顺序排序。
- **map（映射）**：键值对集合，按键排序，每个键对应一个值。
- **multiset和multimap**：允许存储重复的元素。

c. 容器适配器

- **stack（栈）**：后进先出的数据结构。
- **queue（队列）**：先进先出的数据结构。
- **priority_queue（优先队列）**：元素按优先级出列的队列。

3. Vector（向量）

Vector是一种动态数组，支持随机访问。当其存储的元素超过当前分配的容量时，它会自动重新分配空间以适应新元素。

基本用法

1. 创建 `vector`

可以通过不同的方式创建 `vector`：

```
#include <vector>  // 必须包含 vector 头文件

int main() {
    std::vector<int> vec1;  // 创建一个空的 vector
```

```
    std::vector<int> vec2(5, 10);  // 创建一个包含 5 个值为 10 的元素的 vector
}
```

- `vec1` 是一个空的 `vector`，还没有存储任何元素。
- `vec2` 初始化时包含 5 个元素，每个元素的值都是 10。

2. 向 `vector` 中添加元素

使用 `push_back()` 向 `vector` 的末尾添加元素：

```
std::vector<int> vec;
vec.push_back(1);  // 在 vector 的末尾添加 1
vec.push_back(2);  // 在 vector 的末尾添加 2
vec.push_back(3);  // 在 vector 的末尾添加 3
```

3. 访问元素

- 使用 **下标运算符** `[]` 访问元素：

```
std::cout << vec[0];  // 输出第一个元素，结果是 1
std::cout << vec[1];  // 输出第二个元素，结果是 2
```

- 使用 `at()` 方法进行边界检查：

```
std::cout << vec.at(2);  // 输出第三个元素，结果是 3
```

4. 获取 `vector` 的大小

使用 `size()` 函数获取 `vector` 中的元素个数：

```
std::cout << "Size of vector: " << vec.size() << std::endl;  // 输出: Size of vector: 3
```

5. 遍历 `vector`

可以使用 `for` 循环或者迭代器遍历 `vector` 中的元素：

```
// 使用范围 for 循环遍历
for (int n : vec) {
    std::cout << n << " ";  // 输出: 1 2 3
}

// 使用迭代器遍历
for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
    std::cout << *it << " ";  // 输出: 1 2 3
}
```

6. 删除元素

- 使用 `pop_back()` 从 `vector` 的末尾删除元素：

  ```
  vec.pop_back();  // 删除最后一个元素
  ```

- 使用 `erase()` 删除特定位置的元素：

```
vec.erase(vec.begin());  // 删除第一个元素
```

7. 清空 `vector`

使用 `clear()` 来删除 `vector` 中的所有元素：

```
vec.clear();  // 清空 vector
```

示例代码

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

int main() {
    // 创建并初始化 vector
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // 向 vector 中添加元素
    numbers.push_back(6);

    // 访问和遍历 vector
    for (int num : numbers) {
        std::cout << num << " ";  // 输出: 1 2 3 4 5 6
    }
    std::cout << std::endl;

    // 获取 vector 的大小
    std::cout << "Size of vector: " << numbers.size() << std::endl;

    // 删除最后一个元素
```

```
    numbers.pop_back();

    // 输出删除后的 vector
    for (int num : numbers) {
        std::cout << num << " ";  // 输出: 1 2 3 4 5
    }
    std::cout << std::endl;

    return 0;
}
```

原理与特点

- **动态扩容：vector在需要时会自动扩大其存储容量。**
- 内存连续：vector中的元素在内存中是连续存储的。
- 随机访问：可以通过下标直接访问vector中的任何元素。

优势

- 动态扩容：在不知道具体需要多少元素的情况下非常有用。
- 便利性：提供了大量的函数，简化了很多操作。
- 迭代器支持：支持随机访问迭代器，可以用在标准算法中。

和Array的对比（一般算法题是不用array的）

`array` 在 C++11 中引入，是一个容器，它封装了固定大小的数组。

特点

1. 固定大小：一旦定义，其大小就不能改变。
2. 栈分配：默认在栈上分配内存（除非是动态分配的数组）。
3. 性能：访问速度快，因为没有动态内存分配的开销。
4. 类型安全：相比于原始数组，提供更好的类型安全性。

Array的优势

- 性能优越：在性能敏感的场景中有优势，因为没有动态内存分配和释放的开销。
- 确定性：固定大小意味着在编译时就确定了大小，有利于确定性的内存使用。
- 堆栈分配：通常分配在栈上，可以提供更快的访问速度，且避免了堆分配的复杂性。

实现原理

1. 动态数组结构：`vector` 在底层使用动态分配的数组来存储元素。这意味着它能在运行时根据需要调整容量。
2. 包含三个迭代器，start和finish之间是已经被使用用的空间范围，end_of_storage是整块连续空间包括备用空间的尾部。
3. 自动管理内存：当一个 `vector` 的元素数量超过其当前分配的空间时，它会自动重新分配更多的空间来容纳新元素。这个过程通常涉及分配一个更大的数组、复制旧数组的内容到新数组，然后释放旧数组的内存。

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

int main() {
    // 创建一个空的vector
    std::vector<int> vec;

    // 添加元素
    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);

    // 输出vector的内容
    for (int i : vec) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

4. 连续内存：`vector` 中的元素是连续存储的，这意味着可以通过指针算术直接访问它们。这也使得 `vector` 对于随机访问非常高效，但在中间插入或删除元素时可能效率较低，因为这可能需要移动大量的元素。

```cpp
// 随机访问vector中的元素
int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 访问第三个元素
    std::cout << "The third element is: " << vec[2] << std::endl;

    return 0;
}
```

```
10
```

5. 大小和容量：`vector` 维护两个关键的属性：大小（size）和容量（capacity）。大小是指 `vector` 当前包含的元素数，而容量是指在不重新分配内存的情况下 `vector` 可以容纳的元素数。当添加新元素导致大小超过容量时，`vector` 将增加其容量。

```cpp
// 检查vector的大小和容量
int main() {
    std::vector<int> vec;

    // 添加一些元素
    for (int i = 0; i < 10; ++i) {
        vec.push_back(i);
    }

    std::cout << "Size: " << vec.size() << std::endl;
    std::cout << "Capacity: " << vec.capacity() << std::endl;

    return 0;
}

```

6. 模板类：作为模板类，`vector` 可以存储任何类型的元素，从基本数据类型到用户定义的类。

```cpp
// 创建一个存储string的vector
int main() {
    std::vector<std::string> vec;

    // 添加字符串元素
    vec.push_back("Hello");
    vec.push_back("World");

    // 输出vector内容
    for (const std::string& str : vec) {
        std::cout << str << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

7. 迭代器支持：`vector` 提供了迭代器，使得我们可以用类似于指针的方式来遍历 `vector` 中的元素。（迭代器类似于指针，允许你在容器中移动和访问元素，但它们提供了更高级的抽象和安全性。）

```cpp
// 使用迭代器遍历vector
int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 使用迭代器遍历vector
    for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}

```

8. 内存分配策略：大多数 `vector` 实现会在需要增加容量时将当前容量增加一倍，这种策略在均摊分析下能保持添加新元素的均摊时间复杂度为常数级别。

```cpp
// 观察vector扩容过程
int main() {
    std::vector<int> vec;

    // 观察扩容过程
    for (int i = 0; i < 33; ++i) {
        vec.push_back(i);
        std::cout << "Size: " << vec.size() << " Capacity: " << vec.capacity() << std::endl;
    }

    return 0;
}
//运行输出结果
Size: 1 Capacity: 1
Size: 2 Capacity: 2
Size: 3 Capacity: 4
Size: 4 Capacity: 4
Size: 5 Capacity: 8
Size: 6 Capacity: 8
Size: 7 Capacity: 8
Size: 8 Capacity: 8
Size: 9 Capacity: 16
```

```
Size: 10 Capacity: 16
Size: 11 Capacity: 16
Size: 12 Capacity: 16
Size: 13 Capacity: 16
Size: 14 Capacity: 16
Size: 15 Capacity: 16
Size: 16 Capacity: 16
Size: 17 Capacity: 32
Size: 18 Capacity: 32
Size: 19 Capacity: 32
Size: 20 Capacity: 32
Size: 21 Capacity: 32
Size: 22 Capacity: 32
Size: 23 Capacity: 32
Size: 24 Capacity: 32
Size: 25 Capacity: 32
Size: 26 Capacity: 32
Size: 27 Capacity: 32
Size: 28 Capacity: 32
Size: 29 Capacity: 32
Size: 30 Capacity: 32
Size: 31 Capacity: 32
Size: 32 Capacity: 32
Size: 33 Capacity: 64

```

Vector的常用函数

- `push_back(value)`：在末尾添加一个元素。
- `pop_back()`：删除最后一个元素。
- `size()`：返回元素数量。
- `capacity()`：返回分配的存储大小。
- `clear()`：移除所有元素。
- `erase(iterator)`：移除迭代器所指的元素。
- `insert(iterator, value)`：在指定位置插入一个元素。

`emplace_back` 和 `push_back` 的区别

`emplace_back` 和 `push_back` 都是向 C++ 向量容器中添加元素的方法，但它们有一些重要的区别：

1. 构造元素的方式：

- `push_back`：将一个已经构造的元素副本添加到向量末尾。这意味着你需要首先创建一个对象，然后将其添加到向量中，可能涉及到对象的复制构造函数。
- `emplace_back`：直接在向量的末尾构造一个新的元素。这意味着你可以直接传递构造元素所需的参数给 `emplace_back`，它会在容器中构造元素，避免了复制构造的开销。

2. 性能差异：
   - `emplace_back` 通常比 `push_back` 更高效，因为它避免了复制构造的步骤，直接在容器中构造元素。
   - `push_back` 需要先创建一个元素的副本，然后将该副本添加到容器中，可能涉及到额外的内存分配和复制操作，因此在某些情况下可能会导致性能下降。

C++ 无序哈希表详细解析

概念

无序哈希表是一种基于哈希表的数据结构，用于存储键值对。C++ STL 中的 `std::unordered_map` 和 `std::unordered_set` 分别用于存储唯一键和键值对。

基本用法

1. 创建 `unordered_map`

```cpp
#include <unordered_map>  // 必须包含 unordered_map 头文件

int main() {
    // 创建一个 unordered_map，键是字符串，值是整数
    std::unordered_map<std::string, int> umap;

    return 0;
}
```

2. 插入元素

- 使用 `operator[]` 或 `insert()` 方法插入键值对：

```
umap["apple"] = 5;  // 通过[]插入键值对，键是 "apple"，值是 5
umap["banana"] = 3;  // 插入另一个键值对

// 使用 insert 插入
umap.insert({"orange", 7});  // 使用 insert 方法插入键值对
```

- `operator[]` 插入时，如果键不存在会创建一个新元素；如果键已存在，会更新其值。

3. 访问元素

- 使用 **`operator[]`** 或 **`at()`** 方法访问值：

```
std::cout << "apple: " << umap["apple"] << std::endl;  // 访问键 "apple" 的值，输出：5
std::cout << "banana: " << umap.at("banana") << std::endl;  // 使用 at 方法访问
```

- `operator[]` 会在键不存在时创建一个新元素，而 `at()` 不会，如果键不存在，`at()` 会抛出异常。

4. 遍历 `unordered_map`

可以使用范围 `for` 循环或迭代器遍历 `unordered_map`：

```
// 使用范围 for 循环遍历 unordered_map
for (const auto& pair : umap) {
    std::cout << pair.first << ": " << pair.second << std::endl;
}

// 使用迭代器遍历 unordered_map
for (std::unordered_map<std::string, int>::iterator it = umap.begin(); it != umap.end(); ++it) {
```

```
    std::cout << it->first << ": " << it->second << std::endl;
}
```

5. 查找元素

- 使用 `find()` 方法查找元素：

```
auto search = umap.find("apple");  // 查找键 "apple"
if (search != umap.end()) {
    std::cout << "Found: " << search->first << " -> " << search->second << std::endl;
} else {
    std::cout << "Not found" << std::endl;
}
```

`find()` 返回一个迭代器，指向找到的键值对。如果没有找到，返回 `end()` 迭代器。

6. 删除元素

- 使用 `erase()` 方法删除指定键的元素：

```
umap.erase("banana");  // 删除键为 "banana" 的键值对
```

- `erase()` 方法也可以使用迭代器范围来删除多个元素。

7. 清空 `unordered_map`

使用 `clear()` 方法清空所有元素：

```
umap.clear();  // 清空 unordered_map 中的所有元素
```

完整示例代码

```
#include <iostream>
#include <unordered_map>
#include <string>

int main() {
    // 创建一个 unordered_map，存储水果及其数量
    std::unordered_map<std::string, int> umap;

    // 插入键值对
    umap["apple"] = 5;
    umap["banana"] = 3;
    umap.insert({"orange", 7});

    // 访问元素
    std::cout << "apple: " << umap["apple"] << std::endl;
    std::cout << "banana: " << umap.at("banana") << std::endl;

    // 遍历 unordered_map
    std::cout << "Fruits in the map:" << std::endl;
    for (const auto& pair : umap) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    // 查找元素
    auto search = umap.find("apple");
    if (search != umap.end()) {
        std::cout << "Found: " << search->first << " -> " << search->second <<
std::endl;
    } else {
        std::cout << "Not found" << std::endl;
```

```cpp
    }

    // 删除元素
    umap.erase("banana");
    std::cout << "After erasing banana:" << std::endl;
    for (const auto& pair : umap) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

输出结果：

```
apple: 5
banana: 3
Fruits in the map:
apple: 5
banana: 3
orange: 7
Found: apple -> 5
After erasing banana:
apple: 5
orange: 7
```

特性

- 快速操作：平均情况下，插入、查找、删除操作的时间复杂度为 O(1)。
- 无序存储：元素无序存储，不保证迭代顺序。
- 键的唯一性：`unordered_set` 的每个元素是唯一的，`unordered_map` 中每个键对应唯一值。

实现原理

无序哈希表的实现包括以下几个关键部分：

- 哈希函数：将键转换成数组索引。哈希函数的质量直接影响哈希表的效率。理想情况下，哈希函数应该均匀分布键以减少碰撞。
- 碰撞解决：`unordered_map` 无需考虑，但 `unordered_multimap` 需要，当多个键哈希到同一位置时，会发生碰撞。解决方法包括链地址法（每个桶是一个链表或者其他形式的动态数据结构，存储所有哈希到该位置的元素）（`unordered_multimap` 通过哈希函数将键映射到桶（buckets），每个桶中可以包含多个键值对，解决了键碰撞的问题。）
- 动态扩容：随着元素的增加，哈希表的加载因子（元素个数与桶个数的比值）增加。当加载因子超过某个阈值（通常是 0.75），哈希表需要扩容，这包括申请一个更大的数组和重新哈希所有元素。
- 内存管理：为了有效管理内存，无序哈希表可能会在空间不足时重新分配内存，这可能导致大量元素的复制和重新哈希。

与 Set 的区别

- 数据结构：`std::set` 基于红黑树实现，保持元素有序；而 `std::unordered_set` 基于哈希表，不保持元素顺序。
- 性能差异：`set` 在维护元素顺序方面更加高效，适用于需要有序遍历的场景。`unordered_set` 在插入和查找方面通常更快，但不保证元素顺序。

常用函数及代码示例

- 插入元素：`insert`
- 删除元素：`erase`
- 查找元素：`find`
- 获取大小：`size`
- 检查是否为空：`empty`

示例：std::unordered_set

```
include <iostream>
include <unordered_set>
int main() {
    std::unordered_set<intset = {1, 2, 3, 4, 5};
    set.insert(6);  // 插入元素if (set.find(4) != set.end()) {
        std::cout << "4 is in the set." << std::endl;
    }
    set.erase(3);  // 删除元素for (int element : set) {
        std::cout << element << " ";
    }
    std::cout << std::endl;return 0;
}
```

示例：std::unordered_map

```cpp
#include <iostream>
#include <unordered_map>
int main() {
    std::unordered_map<int, std::string> map;
    map[1] = "one";
    map[2] = "two";
    map[3] = "three";
    std::cout << "The value of key 2 is " << map[2] << std::endl;
    map.erase(1);  // 删除键为 1 的元素for (auto& pair : map) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }return 0;
}
```

动态规划的基本应用

斐波那契数列是一个经典的动态规划问题。在这个数列中，第一个数和第二个数分别是 0 和 1，之后的每个数都是前两个数的和。求第 `n` 个斐波那契数。

```cpp
//非动态规划 常规的递归解法会有很多重复的计算，因为它会多次计算相同的斐波那契数。
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

//动态规划 使用 unordered_map 来存储已经计算过的斐波那契数，可以避免重复计算，大大提高效率
//本质是空间换时间
#include <iostream>
#include <unordered_map>

int fibonacci(int n, std::unordered_map<int, int>& memo) {
    if (n <= 1) {
        return n;
    }
    // 检查是否已经计算过这个值
    if (memo.find(n) != memo.end()) {
        return memo[n];
    }

    // 计算并存储结果
```

```cpp
    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo);
    return memo[n];
}

int main() {
    std::unordered_map<int, int> memo;
    int n = 10; // 比如计算第10个斐波那契数
    std::cout << "Fibonacci number at position " << n << " is " << fibonacci(n, memo) << std::endl;
    return 0;
}
```

I/O操作（一般不会问，感兴趣看看就好）

C++ 标准库提供了一套丰富的输入/输出（I/O）机制，主要通过流（stream）对象进行操作。流是一个抽象概念，可以理解为数据的源（输入）或目的地（输出）。

标准 I/O 流

1. **cin**：`std::cin` 是一个与标准输入设备（通常是键盘）连接的输入流对象。
2. **cout**：`std::cout` 是一个与标准输出设备（通常是控制台屏幕）连接的输出流对象。
3. **cerr**：`std::cerr` 是标准错误流，用于输出错误消息和其他诊断信息。与 `cout` 不同，`cerr` 不缓冲，直接输出。
4. **clog**：`std::clog` 类似于 `cerr`，但是它是缓冲的。这意味着输出可能不会立即出现在屏幕上。

这些标准流对象提供了进行基本输入和输出操作的接口。

文件流

1. **ifstream**：用于从文件读取数据（Input File Stream）。
2. **ofstream**：用于向文件写入数据（Output File Stream）。
3. **fstream**：可以同时用于读写操作（File Stream）。

文件流对象可以与文件关联，使得文件的读写操作变得简单直观。

I/O 流库特性

- **格式化 I/O**：C++ I/O 流支持格式化输入和输出。例如，可以控制输出的宽度、精度、对齐方式等。
- **流控制操作**：流对象提供了多种控制和状态检查的操作，如检查流的状态（eof、fail等），设置条件状态等。
- **与 STL 容器兼容**：I/O 流可以与 STL 容器如 `std::string`、`std::vector` 等直接交互，方便容器内容的输入和输出。

示例代码

基本 I/O

```cpp
#include <iostream>

int main() {
    int number;
    std::cout << "Enter a number: ";
    std::cin >> number;
    std::cout << "You entered: " << number << std::endl;
    return 0;
}
```

文件 I/O

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

int main() {
    std::ofstream outfile("example.txt");
    outfile << "Writing to a file." << std::endl;
    outfile.close();

    std::ifstream infile("example.txt");
    std::string line;
    while (getline(infile, line)) {
```

```
        std::cout << line << std::endl;
    }
    infile.close();
    return 0;
}
```

C++ 的 I/O 流库提供了一个强大、灵活且类型安全的方式来处理输入和输出。无论是标准的控制台 I/O 还是文件 I/O，C++ 的流机制都能有效地满足不同的编程需求。

常见的 C++ STL 面试问题及答案：

---

1. 什么是 C++ STL，它由哪些组件组成？

回答：

C++ 标准模板库（STL）是一个通用的、可重用的模板类和函数集合，用于处理数据结构和算法。它主要由以下组件组成：

- **容器（Containers）**：用于存储和组织数据的对象，例如 `vector`、`list`、`deque`、`set`、`map` 等。
- **迭代器（Iterators）**：用于遍历容器元素的对象，类似于指针。
- **算法（Algorithms）**：一组用于操作容器的函数，如排序、搜索、复制等。
- **函数对象（Function Objects）**：也称为仿函数，用于定制算法行为。
- **适配器（Adapters）**：用于修改容器、迭代器或函数接口的工具。
- **分配器（Allocators）**：用于抽象化内存分配过程的类。

---

2. `std::vector` 和 `std::list` 有什么区别？应如何选择？

回答：

- `std::vector`：

- 基于动态数组实现，支持高效的随机访问（时间复杂度 O(1)）。
- 在末尾插入或删除元素速度快（摊销时间复杂度 O(1)）。
- 在中间位置插入或删除元素效率低（时间复杂度 O(n)）。

- `std::list`：

- 基于双向链表实现，不支持随机访问（时间复杂度 O(n)）。
- 在任何位置插入或删除元素都很高效（时间复杂度 O(1)）。
- 内存开销较大，每个元素需要额外的指针空间。

选择建议：

- 需要频繁随机访问元素时，选择 `std::vector`。
- 需要频繁在中间位置插入或删除元素时，选择 `std::list`。

---

3. `std::map` 和 `std::unordered_map` 的区别是什么？

回答：

- `std::map`：

- 基于红黑树实现，内部元素按键值有序存储。
- 查找、插入、删除操作的时间复杂度为 O(log n)。

- 适用于需要有序遍历键值对的场景。

- `std::unordered_map`：

- 基于哈希表实现，内部元素无序存储。
- 查找、插入、删除操作的平均时间复杂度为 O(1)，最坏情况下为 O(n)。
- 适用于不需要有序性、注重查找效率的场景。

---

4. 如何避免 `std::vector` 频繁的内存重新分配?

回答:

使用 `reserve()` 方法预先分配足够的容量，减少内存重新分配的次数。

```
std::vector<int> vec;
vec.reserve(1000); // 预先分配 1000 个元素的空间
```

5. **什么是 `std::sort` 算法的时间复杂度？它是如何工作的?**

回答:

`std::sort` 是STL中的一个高效排序算法，其时间复杂度为平均情况下的O(n log n)。`std::sort` 通常使用快速排序（Quick Sort）、堆排序（Heap Sort）或混合排序算法（如Introsort）来实现，以确保在不同数据分布下都能保持良好的性能。

工作原理:

- **快速排序**：通过选择一个基准元素，将数组分为两部分，左边小于基准，右边大于基准，然后递归排序两部分。
- **堆排序**：将数组视为一个堆结构，构建最大堆，然后依次取出堆顶元素，重新调整堆。
- **Introsort**：结合快速排序和堆排序，当递归深度超过一定阈值时切换到堆排序，避免快速排序在最坏情况下的性能下降。

课程总结

通过本课程，我们详细了解了STL的核心组件，包括容器、迭代器、算法和函数对象。特别是对vector和multimap进行了深入的讨论，包括它们的原理、特点和常用函数。了解这些知识，可以帮助我们在C++编程中更有效地使用STL，构建更复杂和高效的程序。