大多数算法都在算法头中定义。该库还定义了一组在数字头中定义的通用数字算法。

一般来说，算法不直接在容器上工作。相反，它们通过遍历由两个迭代器限定的元素范围来进行操作（第9.2.1节）。通常，当算法遍历范围时，它会对每个元素执行一些操作。

find将给定范围内的每个元素与给定值进行比较。它返回一个迭代器到第一个等于该值的元素。

它访问序列中的第一个元素。2.它将该元素与我们想要的值进行比较。3.如果这个元素与我们想要的元素匹配，find返回一个标识这个元素的值。4.否则，find前进到下一个元素并重复步骤2和3。5.find必须在到达序列末尾时停止。6.如果find到达序列的末尾，它需要返回一个值，指示未找到该元素。该值和步骤3返回的值必须具有兼容的类型。

这些操作都不依赖于保存元素的容器的类型。只要有一个迭代器可以用来访问元素，find就不以任何方式依赖于容器类型

然而，正如我们将看到的，大多数算法都提供了一种方法，让我们提供自己的操作来代替默认操作符。

它们的核心思想是：

* **基于迭代器**:
  + 泛型算法通过迭代器访问和操作容器中的元素。
  + 迭代器是算法和容器之间的桥梁，算法不需要知道容器的具体类型。
* **不直接操作容器**:
  + 泛型算法不会直接调用容器的成员函数（如 push\_back、erase 等）。
  + 它们只通过迭代器访问和修改元素。

理解算法的最基本方法是知道它们是读元素、写元素，还是重新排列元素的顺序。

只读算法

int sum = accumulate(vec.cbegin(), vec.cend(), 0);

accumulate的第三个参数的类型确定使用哪个加法运算符，并且是accumulate返回的类型。

通常，最好在读而不写元素的算法中使用ccbegin（）和ccbegin（）（第9.2.3节）。但是，如果计划使用算法返回的迭代器来更改元素的值，则需要传递开始（）和结束（）。

equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());

使用单个迭代器表示第二个序列的算法假定第二个序列至少与第一个序列一样大。我们需要确保算法不会试图访问第二个序列中不存在的元素。例如，equal算法可能会将第一个序列中的每个元素与第二个序列中的元素进行比较。如果第二个序列是第一个序列的一个子集，那么我们的程序就有一个严重的错误--equal将试图访问第二个序列末尾以外的元素。

fill\_n 是C++标准库中的一个泛型算法，用于将指定数量的元素设置为给定值。

* **参数**:
  + dest: 目标位置的迭代器，表示写入的起始位置。
  + n: 要写入的元素数量。
  + val: 要设置的值。

lambda表达式表示可调用的代码单元。它可以被认为是一个未命名的内联函数。像任何函数一样，lambda有返回类型，参数列表和函数体。与函数不同的是，可以在函数内部定义。lamba表达式的形式为

[capture list] (parameter list) -> return type

{ function body }

图形用户界面, 文本

描述已自动生成

Lambda函数体包含除单个return语句之外的任何内容，并且没有指定返回类型return void。

**Lambda 用于简化代码**

* **临时函数**:
  + Lambda 表达式非常适合用于需要临时函数的场景，例如作为算法参数、回调函数等。
  + 它避免了定义额外的命名函数，使代码更简洁

**Lambda 表达式** 可以看作是一个**没有函数名**（匿名）的**简化函数**，它的主要目的是为了**简化代码**，尤其是在需要临时函数的场景中。

* **解引用发生在 std::for\_each 内部**:
  + std::for\_each 在遍历时，会通过迭代器访问每个元素。
  + 例如，\*it 会解引用迭代器 it，获取当前元素的值。
* **传递给 Lambda 表达式**:
  + 解引用后的值会作为参数传递给 Lambda 表达式的参数 n。

一个lambda可以使用一个局部变量到它周围的函数，只有当lambda捕获它的捕获列表中的变量。

捕获列表仅用于局部非静态变量;捕获列表可以直接使用局部静态变量和在函数外部声明的变量。

* **Lambda 表达式获取值的两种方式**:
  1. **捕获列表 []**:
     + 用于访问外部作用域中的变量。
     + 可以是值捕获或引用捕获。
  2. **参数列表 ()**:
     + 用于接受调用时传入的值。
* **适用场景**:
  1. 如果需要访问外部变量，使用捕获列表。
  2. 如果需要动态传入值，使用参数列表。

**2. 引用捕获的限制**

* **必须确保引用的对象存在**:
  + 使用引用捕获时，必须确保被引用的对象在 Lambda 表达式执行时仍然存在。
  + 如果 Lambda 表达式可能在函数返回后执行，应避免捕获局部变量的引用。
* **全局变量和静态变量**:
  + 如果捕获的是全局变量或静态变量，则不存在生命周期问题，因为它们在整个程序运行期间都存在。

当我们通过引用捕获变量时，我们必须确保该变量在lambda执行时存在。

lambda捕获存储在创建lambda的时间（即，当定义lambda的代码被执行时）和lambda本身被执行的时间（或次数）。程序员的责任是确保每次执行lambda时捕获的任何信息都具有预期的含义

捕获一个普通的变量int、字符串或其他非指针类型的值通常很简单。在这种情况下，我们只需要关心变量是否具有我们在捕获它时需要的值。

如果我们捕获一个指针或迭代器，或者通过引用捕获一个变量，我们必须确保无论何时lambda执行，绑定到该迭代器、指针或引用的对象仍然存在。此外，我们需要确保对象具有预期的值。在创建lambda和执行lambda之间执行的代码可能会更改lambda捕获指向（或引用）的对象的值。在指针（或引用）被捕获时对象的值可能就是我们想要的。当lambda执行时，该对象的值可能完全不同。

通常，我们可以通过最小化捕获的数据来避免捕获的潜在问题。此外，如果可能的话，避免捕获指针或引用。

* **尽量减少捕获的数据:**
  + **只捕获 Lambda 表达式真正需要的数据，减少依赖性和复杂性。**
* **避免捕获指针或引用:**
  + **如果必须捕获指针或引用，确保被指向的对象在 Lambda 表达式执行时仍然存在。**
* **最佳实践:**
  + **尽量使用值捕获。**
  + **使用智能指针管理动态分配的对象。**

可以混合使用隐式捕获和显式捕获，以便更灵活地控制捕获方式。

**（1）混合捕获的规则**

* **第一个捕获项必须是 & 或 =**:
  + 设置默认的捕获方式（引用捕获或值捕获）。
* **显式捕获的变量必须使用相反的方式**:
  + 如果默认是引用捕获（&），则显式捕获的变量必须使用值捕获（不能加 &）。
  + 如果默认是值捕获（=），则显式捕获的变量必须使用引用捕获（加 &）。

文本

描述已自动生成

Lambda表达式对于我们不需要在多个地方使用的简单操作最有用。如果我们需要在许多地方执行相同的操作，我们通常应该定义一个函数，而不是多次编写相同的lambda表达式。同样，如果一个操作需要许多语句，通常最好使用函数。

**2. 为什么需要 string::size\_type**

* **平台无关性**:
  + std::size\_t 的大小可能因平台而异（通常是 32 位或 64 位）。
  + 使用 string::size\_type 可以确保代码在不同平台上都能正确工作。
* **语义清晰**:
  + 使用 string::size\_type 可以明确表示该变量用于存储字符串的长度或索引，提高代码的可读性。

**4. std::bind 的常见用途**

* **参数绑定**:
  + 将多参数函数绑定为单参数函数。
* **参数重排**:
  + 改变函数参数的顺序。
* **部分应用**:
  + 固定函数的部分参数，生成一个新的可调用对象。

sort(words.begin(), words.end(), bind(isShorter, \_2, \_1));

除了为每个容器定义的迭代器之外，库还在迭代器头中定义了几种其他类型的迭代器。这些迭代器包括·

插入迭代器：这些迭代器绑定到容器，可以用来将元素插入容器。

·流迭代器：这些迭代器绑定到输入或输出流，可用于遍历关联的IO流

。·反向迭代器：这些迭代器向后移动，而不是向前移动。

除了forward\_list之外，库容器都有反向迭代器。·

移动迭代器：这些特殊用途的迭代器移动而不是复制它们的元素。我们将在第13.6.2节中介绍移动迭代器。

插入迭代器是一种特殊的迭代器适配器，用于在容器中插入元素，而不是覆盖现有元素。常见的插入迭代器包括：

当我们调用front\_inserter（c）时，我们得到一个插入迭代器，它连续调用push\_front。当每个元素被插入时，它成为c中新的第一个元素

任何算法最基本的属性是它需要迭代器的操作列表。有些算法，比如find，只需要通过迭代器访问元素、递增迭代器和比较两个迭代器是否相等的能力。其他的，比如sort，需要读、写和随机访问元素的能力。算法所需的迭代器操作被分为表10 - 5中列出的五种迭代器类别。每个算法指定必须为其每个迭代器参数提供哪种迭代器。

1. **输入迭代器（Input Iterator）**：
   * **功能**：可以读取元素，但不能写入。
   * **遍历方式**：单次遍历，只能向前移动（仅支持递增）。
   * **使用场景**：通常用于从序列中读取数据（例如从文件或容器中读取）。
2. **输出迭代器（Output Iterator）**：
   * **功能**：可以写入元素，但不能读取。
   * **遍历方式**：单次遍历，只能向前移动（仅支持递增）。
   * **使用场景**：通常用于向序列中写入数据（例如向文件或容器中写入）。
3. **前向迭代器（Forward Iterator）**：
   * **功能**：可以读取和写入元素。
   * **遍历方式**：多次遍历，只能向前移动（仅支持递增）。
   * **使用场景**：适用于需要多次遍历的场景（例如链表）。
4. **双向迭代器（Bidirectional Iterator）**：
   * **功能**：可以读取和写入元素。
   * **遍历方式**：多次遍历，支持向前和向后移动（支持递增和递减）。
   * **使用场景**：适用于需要双向遍历的场景（例如双向链表）。
5. **随机访问迭代器（Random-access Iterator）**：
   * **功能**：可以读取和写入元素。
   * **遍历方式**：多次遍历，支持完整的迭代器算术（如加减、比较、随机访问）。
   * **使用场景**：适用于需要随机访问的场景（例如数组或向量）。

当我们传递错误的迭代器类别给算法时，许多编译器不会抱怨。

std::list 和 std::forward\_list 是链表容器，它们的内部结构与顺序容器（如 std::vector 或 std::deque）不同。由于链表的特殊性质，**成员函数版本**的算法通常比**泛型算法**更适合

标准库定义了大约100种对序列进行操作的独立于类型的算法。序列可以是库容器类型、内置数组中的元素，或者（例如）通过阅读或写入流而生成。

算法通过在迭代器中操作来实现其类型独立性。大多数算法将一对表示元素范围的迭代器作为它们的前两个参数。其他迭代器参数可能包括一个表示目的地的输出迭代器，或者另一个表示第二个输入序列的迭代器或迭代器对。

迭代器根据其支持的操作分为五类。迭代器类别包括输入、输出、前向、双向和随机访问。一个迭代器属于一个特定的类别，如果它支持该迭代器类别所需的操作。就像迭代器是按其操作分类的一样，算法的迭代器参数也是按其所需的迭代器操作分类的。只读取其序列的算法只需要输入迭代器操作。那些写入目标迭代器的算法只需要输出迭代器的操作，等等。

算法从不直接改变它们所操作的序列的大小。它们可以将元素从一个位置复制到另一个位置，但不能直接添加或删除元素。虽然算法不能向序列中添加元素，但插入迭代器可以这样做。插入迭代器绑定到容器。

当我们将容器的元素类型的值赋给插入迭代器时，迭代器将给定的元素添加到容器中。forward\_list和list容器定义了它们自己的一些泛型算法的版本。与泛型算法不同，这些特定于列表的版本修改给定的列表。