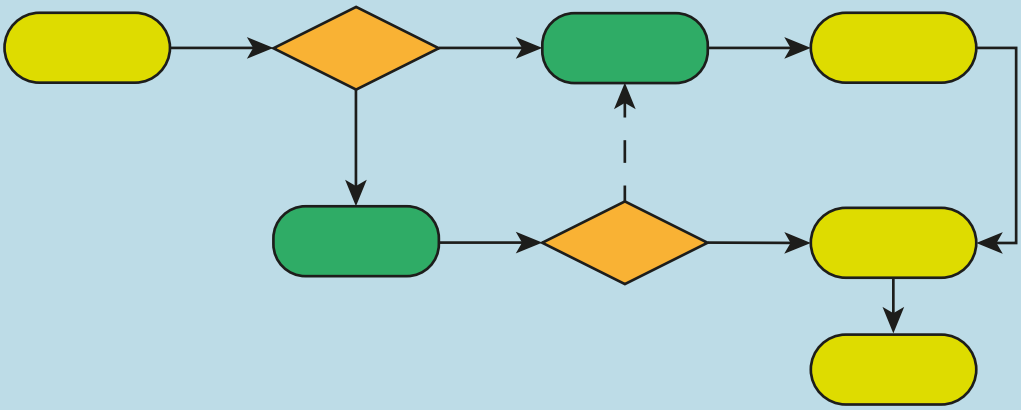
A Complete Guide to

Standard C++ Algorithms



RNDr. Šimon Tóth

完整指南

标准 C++ 算法

RNDr. Šimon Tóth

version 1.0.1
https://github.com/HappyCerberus/book-cpp-algorithms

February 3, 2023

Original copy of this book can be obtained at https://github.com/HappyCerberus/book-cpp-algorithms.

The book is also available through LeanPub where the proceeds go to Electronic Frontier Foundation (after LeanPub takes their cut) https://leanpub.com/cpp-algorithms-guide.

This copy of the book is version 1.0.1.

前言

本书不会以个人故事或其他花哨的回忆开始。相反，为了节省你的时间，我将明确地说明本书的内容以及我为何有资格写这本书。希望这能帮助你决定阅读本书是否值得花费时间。

关于本书

本书是C++标准算法的完整指南。然而，这对你来说可能意义不大，让我来解释一下这句话。

这本书是一本指南，而非参考书，这意味着它不是描述每个细节，而是集中于示例，并指出不同算法中值得注意、令人惊讶、危险或有趣的方面。此外，与参考书不同，它应该像一本书一样，按顺序大部分时间阅读。

C++ 已经有一个标准参考，C++ 标准，且对于快速查阅，cppreference.com 维基是一个很好的资源。

陈述中“complete”的部分指的是覆盖范围的广度。本书涵盖截至 C++20 标准的所有算法及相关理论（在撰写时，C++23 stan- dard 尚未最终确定）。所有信息仅以满足语境所需的充分深度呈现。这一深度限制使本书的整体篇幅保持合理，并符合“guide”风格。

关于作者

我是 imon Tóth，本书的唯一作者。我的主要资历是拥有20年的C++经验，其中大约15年里，C++是我在专业环境中的主要语言。

我的背景是 HPC，涵盖学术界、大型科技公司以及初创公司环境。我曾架构、构建并运维过各种规模的系统，从由单机硬件支撑的高可用系统到行星级规模的服务。[1]

在我的整个职业生涯中，我的热情始终在于教学和指导初级工程师，这也是你现在正在阅读这本书的原因。

[1]You can check my LinkedIn profile for a detailed view of my past career.

反馈

创作免费的教育内容很像是在对着虚空呐喊。既没有可追踪的销售统计，也没有需要达成的里程碑。因此，如果你读了这本书，觉得它有帮助，或者讨厌它，请告诉我。这将为我今后的努力提供参考。

为什么是cc-by-sa-nc

本书采用CC-BY-SA-NC许可协议，这是一个开放但同时又最小化的许可。我旨在允许衍生作品（例如翻译），但不允许商业用途，例如将本书作为商业培训的基础或销售印刷版。

明确地，允许任何个人用途。例如，你可以阅读、打印，或与朋友分享这本书。

如果你想使用这些内容，但不确定自己是否符合 Creative Commons 商业定义 [2]，欢迎通过 Mastodon、LinkedIn 或电子邮件联系我（我的私信始终开放）。

书籍状态

本书目前内容已完成（截至并包括 C++20）。如需跟进变更，请访问托管仓库：`https://github.com/ HappyCerberus/book-cpp-algorithms`。

更新日志

1.0.1 小幅（主要是）格式调整。

1.0.0 首个完整版本。

[2] primarily intended for or directed toward commercial advantage or monetary compensation

内容

第1章

引言

可以说，C++ 标准库在功能上相当有限。然而，在数据处理和数值计算方面，C++ 标准库提供了一套多功能的算法工具包。

如果你是一名 C++ 开发者，良好地熟悉 C++ 标准算法可以为你节省大量精力并避免意外的缺陷。尤其是，当你在代码中看到原始循环时，你应该思考是否调用标准算法会是更好的解决方案（通常确实如此）。

1.1 标准 C++ 算法的历史

尽管每一版 C++ 标准都引入了新的算法或变体，但在 C++ 标准算法的发展历史中，显著的里程碑并不多。

C++98 标准引入了大多数算法。然而，正是 C++11 标准通过引入 lambda，使得算法变得值得使用。在引入 lambda 之前，编写自定义函数对象所需的时间投资让算法的实用性变得值得怀疑。

Example of `std::for_each` algorithm with a custom function object, calculating the number of elements and their sum.

```
struct StatsFn {
    int cnt = 0;
    int sum = 0;
    void operator()(int v) {
        cnt++;
        sum += v;
    }
};

std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto result = std::for_each(data.begin(), data.end(), StatsFn{});
// result == {9, 45}
```

Open in Compiler Explorer

Example of `std::for_each` algorithm with a capturing lambda, calculating the number of elements and their sum.

```
int cnt = 0, sum = 0;
std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};
std::for_each(data.begin(), data.end(), [&](int el) {
    cnt++;
        sum += el;
    });
// cnt == 9, sum == 45
```

Open in Compiler Explorer

C++17 标准引入了并行算法，以最少的努力提供了一种轻松加速处理的方式。你所需要做的只是指定所需的执行模型，库就会负责将执行过程并行化。

Example of `std::for_each` algorithm using unsequenced parallel execution model. Note that counters are now shared state and need to be `std::atomic` or protected by a `std::mutex`.

```
std::atomic<int> cnt = 0, sum = 0;
std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};
std::for_each(std::execution::par_unseq,
    data.begin(), data.end(),
    [&](int el) {
        cnt++;
        sum += el;
    });
// cnt == 9, sum == 45
```

Open in Compiler Explorer

最后，C++20 标准以范围和视图的形式引入了一项重大的重新设计。算法的范围版本现在可以在范围上运行，而不是 `begin` 和 `end` 迭代器，而视图则提供了算法和实用工具的惰性求值版本。

Example of the range version of the `std::for_each` algorithm.

```
int cnt = 0, sum = 0;
std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};
std::ranges::for_each(data, [&](int el) {
    cnt++;
    sum += el;
});
// cnt == 9, sum == 45
```

Open in Compiler Explorer

截至撰写本文时，C++23 标准尚未最终定稿。然而，我们已经知道它将引入更多的 ranges 算法、更多的视图，以及实现自定义视图的能力。

1.2 迭代器与范围

算法作用于数据结构，这带来了一个问题。如何抽象出特定数据结构的实现细节，并使算法能够与任何满足算法要求的数据结构一起工作?

C++ 标准库对此问题的解决方案是迭代器和范围。迭代器封装了数据结构遍历的实现细节，同时以常数时间和空间复杂度提供在给定数据结构上可执行的一组操作。

一个范围由一对迭代器表示，或者更一般地，从 C++20 起，表示为一个迭代器和一个哨兵。在数学术语中，一对迭代器 `it1`、`it2` 表示一个范围 `[it1, it2)`，即该范围包括 `it1` 所引用的元素，并在 `it2` 所引用的元素之前结束。

要引用数据结构的整个内容，我们可以使用 `begin()` 和 `end()` 方法，分别返回指向第一个元素的迭代器和指向最后一个元素之后位置的迭代器。因此，范围 `[begin, end)` 包含了所有数据结构元素。

Example of specifying a range using two iterators.

```
std::vector<int> data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

auto it1 = data.begin();
auto it2 = it1 + 2;
std::for_each(it1, it2, [](int el) {
    std::cout << el << ", ";
});
// Prints: 1, 2,

auto it3 = data.begin() + 5;
auto it4 = data.end();
std::for_each(it3, it4, [](int el) {
    std::cout << el << ", ";
});
// Prints: 6, 7, 8, 9,
```

Open in Compiler Explorer

哨兵遵循相同的思想。然而，它们不需要是迭代器类型。相反，它们只需要能够与迭代器进行比较。这样，区间的不包含末端就是第一个与该哨兵比较相等的迭代器。

Example of specifying a range using an iterator and custom sentinel. The sentinel will compare `true` with iterators at least the given distance from the start iterator, therefore defining a range with the specified number of elements.

```
std::vector<int> data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

struct Sentinel {
    using iter_t = std::vector<int>::iterator;
    iter_t begin;
    std::iter_difference_t<iter_t> cnt;
    bool operator==(const iter_t& l) const {
        return std::distance(begin, l) >= cnt;
    }
};

auto it1 = data.begin();
std::ranges::for_each(it1, Sentinel{it1, 5}, [](int el) {
    std::cout << el << ", ";
});
// Prints: 1, 2, 3, 4, 5,
```

Open in Compiler Explorer

1.2.1 迭代器类别

在常数时间和空间内可执行的操作集合定义了以下几类迭代器（以及相应的范围）：

输入/输出迭代器每次读取/写入一个元素，前进
data streams, e.g. writing/reading data to/from a network socket

前向迭代器可反复读/写每个元素，前进
singly-linked list, e.g. std::forward_list

双向迭代器 前向迭代器 + 向后移动
doubly-linked list, e.g. std::list, std::map, std::set

随机访问迭代器 双向迭代器 + 可以按任意整数前进和后退，并计算两个迭代器之间的距离 *multi-array data structures, e.g. std::deque*

连续迭代器 随机访问迭代器 + 元素的存储是连续的 *arrays, e.g. std::vector*

Example demonstrating the difference between a random access iterator provided by `std::vector` and a bidirectional iterator provided by `std::list`.

```
std::vector<int> arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto it1 = arr.begin();
it1 += 5; // OK, std::vector provides random access iterator
++it1; // OK, all iterators provide advance operation

ptrdiff_t dst1 = it1 - arr.begin(); // OK, random access iterator
// dst1 == 6

std::list<int> lst = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto it2 = lst.begin();
// it2 += 5; Would not compile.
std::advance(it2, 5); // OK, linear advance by 5 steps
++it2; // OK, all iterators provide advance operation

// it2 - lst.begin(); Would not compile
ptrdiff_t dst2 = std::distance(lst.begin(), it2); // OK, linear calc.
// dst2 == 6
```

Open in Compiler Explorer

1.2.2 范围类别

范围可以使用与迭代器相同的类别进行分类。在本书中，我们将使用范围的命名法而不是迭代器（例如，输入范围、前向范围、双向范围等）。

1.3 命名和常见行为

尽管许多算法的命名并不理想，但仍有一些常见的命名模式。

1.3.1 计数变体 ”`_n`”

计数变体的算法接受使用起始迭代器和元素数量指定的范围（而不是开始和结束）。当处理输入和输出范围时，这种行为可以成为一个方便的替代方法，因为我们通常没有显式的结束迭代器。

示例：`std::for_each_n`, `std::copy_n`

注意：虽然 `std::search_n` 遵循命名，但并不遵循相同的语义。这里的 `_n` 指的是被搜索元素的实例数量。

1.3.2 复制变体 ”`_copy`”

就地算法的复制变体不会将其输出写回源范围。相反，它们将结果输出到一个或多个输出范围，通常由一个表示要写入的第一个元素的迭代器定义（元素数量由源范围隐含）。复制行为允许这些变体在不可变范围上操作。

示例：`std::remove_copy`，`std::partial_sort_copy`

1.3.3 谓词变体 ”`_if`”

算法的谓词变体使用谓词来确定“匹配”，而不是与一个值进行比较。该标准还具有一个`_if_not`变体实例，该实例反转谓词逻辑（`false`被视为匹配）。

示例：`std::find_if`，`std::replace_if`

1.3.4 对 invocable 的限制

许多算法可以使用可调用对象进行定制。然而，除少数例外情况外，不允许可调用对象修改范围中的元素或使迭代器失效。此外，除非明确说明，这些算法并不保证任何特定的调用顺序。

这些限制在实践中意味着所传入的可调用对象必须是 regular 的。该可调用对象在使用相同参数再次调用时必须返回相同的结果。此定义允许访问诸如缓存之类的全局状态，但不允许那些根据其内部状态而改变结果的可调用对象（例如生成器）。

1.4 用于迭代器的一个更简单的心智模型

在使用标准算法时，掌握与迭代器相关的所有规则可能会有些棘手。一个可以帮助的简化方法是从范围的角度思考，而不是从迭代器的角度。

作为参数传递的范围通常是显而易见的，通常由一对迭代器指定。

Example with two ranges passed in as an argument. The input range is fully specified, and the end iterator for the output range is implied from the number of elements in the input range.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7};
std::vector<int> out(7,0);

std::copy(data.begin(), data.end(), // input range
    out.begin() // output range, end iterator is implied:
    // std::next(out.begin(),
    //        std::distance(data.begin(), data.end()));
);
```

Open in Compiler Explorer

返回的范围也可以从算法的语义中体现出来。

Example of `std::is_sorted_until` that returns an iterator to the first out-of-order element, which can also be thought as the end iterator for a maximal sorted sub-range.

```
std::vector<int> data{1, 4, 5, 7, 9, 2, 3};

// is_sorted_until returns the first out of order element.
auto result = std::is_sorted_until(data.begin(), data.end());

// [begin, result) is the maximal sorted sub-range
for (auto it = data.begin(); it != result; it++) {
    // Iterate over all elements in the sorted sub-range.
    // {1, 4, 5, 7, 9}
}
for (auto v : std::ranges::subrange(data.begin(), result)) {
    // Same, but using a range-based for loop.
}
```

Open in Compiler Explorer

考虑将返回值视为范围的末尾迭代器的好处在于，它消除了潜在的边界情况。例如，如果算法没有找到任何不按顺序排列的元素怎么办？返回值将是源范围的末尾迭代器，这意味着返回的范围就是整个源范围。

在某些情况下，单个返回的迭代器表示多个范围。

Example of `std::lower_bound` that splits the range into two sub-ranges.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};

// lower_bound returns the first element !(el < 4)
auto lb = std::lower_bound(data.begin(), data.end(), 4);

for (auto v : std::ranges::subrange(data.begin(), lb)) {
    // lower range [begin, lb): elements < 4
}
for (auto v : std::ranges::subrange(lb, data.end())) {
    // upper range [lb, end): elements >= 4
}
```

Open in Compiler Explorer

即使算法返回的是指向某个特定元素的迭代器，也可能值得考虑其所隐含的范围。

Example of `std::find` establishing a prefix range that doesn't contain the searched-for element.

```
std::string str("Hello World!");

// Returns the iterator to the first occurence of ' '
auto it = std::find(str.begin(), str.end(), ' ');

// Range [begin, it) is the maximal prefix range
// that doesn't contain ' '
for (auto v : std::string_view(str.begin(), it)) {
    // iterate over "Hello"
}
```

Open in Compiler Explorer

第二章

算法

2.1 介绍算法

在本章中，我们介绍了每个标准算法。算法的分组是任意的，主要是为了呈现的清晰性。因此，你可能会正确地认为某个特定的算法更适合归属于另一个分组。

在开始之前，我们将使用 `std::for_each` 和 `std::for_each_n` 算法来展示本章针对每种算法的结构。

①每个算法的介绍将从简短的描述开始。

②边际将包含有关算法历史的信息：C++ 标准何时引入该算法，以及它是否具有 constexpr、并行和范围变体，包括引入它们的标准版本。

③随后将给出约束条件的描述。将数据写入不同输出范围的算法用箭头表示：`input_range -> output_range`。

④最后，每个描述将以一个或多个带有解释的示例结束。

2.1.1 `std::for_each`

①`std::for_each` 算法按顺序将所提供的可调用对象应用于范围中的每个元素。如果底层范围是可变的，则该可调用对象可以改变元素的状态，但不得使迭代器失效。

② `std::for_each`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

③ constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_invocable`

④C++11 标准引入了基于范围的 for 循环，它在很大程度上取代了 `std::for_each` 的用法。

Example of a range loop over all elements of a `std::vector`.

```
std::vector<int> data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = 0;
for(auto el : data) {
    sum += el;
}
// sum == 45
```

Open in Compiler Explorer

Example of a `std::for_each` loop over all elements of a `std::vector`.

```
std::vector<int> data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = 0;
std::for_each(data.begin(), data.end(), [&sum](int el) {
    sum += el;
    });
// sum == 45
```

Open in Compiler Explorer

然而，在某些特殊情况下，使用 `std::for_each` 更为优选。

第一种情况是简单的并行化。使用 `std::for_each` 对每个元素并行调用开销高昂的操作是轻而易举的。只要这些操作彼此独立，就不需要同步原语。

Example of a parallel `std::for_each` invoking a method on each element independently in parallel.

```
struct Custom {
    void expensive_operation() {
        // ...
    }
};

std::vector<Custom> data(10);

std::for_each(std::execution::par_unseq,
    data.begin(), data.end(),
    [](Custom& el) {
        el.expensive_operation();
    });
```

Open in Compiler Explorer

其次，由于 C++20 中引入的投影支持，范围版本在某些情况下可以提供更简洁且更明确的语法。

Example of the range version of `std::ranges::for_each` utilizing a projection to invoke the method `getValue()` (line 13) on each element and summing the resulting values using a lambda (line 12).

```
1  struct Custom {
2      explicit Custom(double value) : value_(value) {}
3      double getValue() { return value_; }
4  private:
5      double value_;
6  };
7
8  std::vector<Custom> data(10, Custom{1.0});
9
10 double sum = 0;
11 std::ranges::for_each(data,
12     [&sum](auto v) { sum += v; },
13     &Custom::getValue);
14 // sum == 10.0
```

Open in Compiler Explorer

2.1.2 `std::for_each_n`

①`std::for_each_n` 算法将所提供的可调用对象应用于由一个迭代器和元素数量指定的范围中的每个元素。如果底层范围是可变的，则该可调用对象可以更改元素的状态，但不得使迭代器失效。

② `for_each_n`	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	C++20

③ constraints		
domain	`input_iterator`	
parallel domain	`forward_iterator`	
invocable	default	custom
	N/A	`unary_invocable`

④当 `std::for_each` 在整个范围上操作时，在区间 $[begin, end)$ 中，`std::for_each_n` 在范围 $[first, first + n)$ 上操作。重要的是，由于该算法无法访问源范围的结束迭代器，它不会进行越界检查，因此由调用方负责确保范围 $[first, first + n)$ 是有效的。

Example demonstrating multiple uses of `std::for_each_n`.

```
1  std::vector<Player> final_ranking = get_rankings();
2  std::ranges::sort(final_ranking, std::greater<>(), &Player::score);
3
4  std::for_each_n(std::execution::par_unseq,
5      final_ranking.begin(),
6      std::min(MAIN_SEATS, final_ranking.size()),
7      send_invitation_to_main_tournament);
8
9  auto it = final_ranking.begin();
10 uint32_t page = 0;
11 while (it != final_ranking.end()) {
12     size_t cnt = std::min(PAGE_SIZE, size_t(final_ranking.end() -
       ↪  it));
13     std::for_each_n(it, cnt, [page](const Player& p) {
14         store_final_score(page, p.display_name, p.score);
15     });
16     page++;
17     it += cnt;
18 }
```

Open in Compiler Explorer

向排名前 `MAIN_SEATS` 的玩家发送邀请是并行完成的（第 4–7 行）。随后，所有用户的分数以每 `PAGE_SIZE` 条记录为一块进行存储（第 13–15 行）。请注意，计算剩余元素的数量（第 12 行）以及按已存储元素的数量向前跳转（第 17 行）需要一个随机访问迭代器（在本例中由 `std::vector` 提供）。

2.2 掉期

在 C++11 以及移动操作引入之前，交换是具有值语义的对象在不涉及深拷贝的情况下交换内容的唯一方式。

2.2.1 `std::swap`

swap	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

`std::swap` 的非范围版本将使用三步移动交换来交换两个参数的值。用户可以在其类型上以友元函数的形式提供更优化的实现。

正确地调用 swap 需要将默认的 std::swap 版本引入到局部作用域。要进一步了解为何需要这样做，请查阅本书的理论章节，尤其是关于 ADL 的部分（5.1）。

Example of correctly calling `std::swap`.

```
1  void some_algorithm(auto& x, auto& y) {
2      using std::swap;
3      swap(x, y);
4  }
```

Open in Compiler Explorer

swap 的 C++20 范围化版本消除了这种复杂性，并且它将：

- 调用由用户提供的（通过 ADL 查找到的）swap 重载
- 如果不存在该项，且参数是具有相同跨度的数组，`std::ranges::swap` 的行为将等同于 `std::ranges::swap_ranges`
- 如果参数也不是数组，它将默认进行 move-swap

Example of specializing and calling `std::ranges::swap`.

```
1  namespace Library {
2  struct Storage {
3      int value;
4  };
5
6  void swap(Storage& left, Storage& right) {
7      std::ranges::swap(left.value, right.value);
8  }
9  }
10
11 int main() {
12     int a = 1, b = 2;
13     std::ranges::swap(a, b); // 3-step-swap
14
15     Library::Storage j{2}, k{3};
16     std::ranges::swap(j, k); // calls custom Library::swap()
17 }
```

Open in Compiler Explorer

2.2.2 `std::iter_swap`

`std::iter_swap` 是一种间接交换，交换两个前向迭代器背后的值。

iter_swap	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

constraints	
domain	(`forward_iterator`, `forward_iterator`) (non-range)

Example demonstrating the use `std::iter_swap` in a generic two-way partition algorithm. The algorithm uses concepts to constrain the acceptable types of its arguments.

```
template <typename It, typename Cond>
    requires std::forward_iterator<It>
        && std::indirectly_swappable<It,It>
        && std::predicate<Cond, It>
auto partition(It first, It last, Cond cond) {
    while (first != last && cond(first)) ++first;
    if (first == last) return last;

    for (auto it = std::next(first); it != last; it++) {
        if (!cond(it)) continue;

        std::iter_swap(it, first);
        ++first;
    }
    return first;
}
```

Open in Compiler Explorer

范围版本将功能扩展到其他可解引用对象。

Example demonstrating the use of range version of `std::ranges::iter_swap` to swap the values pointed to by two instances of `std::unique_ptr`.

```
auto p1 = std::make_unique<int>(1);
auto p2 = std::make_unique<int>(2);
int* p1_pre = p1.get();
int* p2_pre = p2.get();

std::ranges::iter_swap(p1, p2);
// p1.get() == p1_pre, *p1 == 2
// p2.get() == p2_pre, *p2 == 1
```

Open in Compiler Explorer

2.2.3 `std::swap_ranges`

swap_ranges	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::swap_ranges` 算法在两个不重叠的范围之间交换元素（可能来自同一个容器）。

Example of swapping the first three elements of an array with the last three elements using `std::swap_ranges`. Note the reversed order of elements due to the use of `rbegin`.

```
std::vector<int> data{ 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::swap_ranges(data.begin(), data.begin()+3, data.rbegin());
// data = { 9, 8, 7, 4, 5, 6, 3, 2, 1 }
```

Open in Compiler Explorer

2.3 排序

在讨论排序之前，我们需要先讨论标准对类型可比较性的要求——具体来说，是排序算法所要求的 `strict_weak_ordering`。

为自定义类型实现 `strict_weak_ordering` 至少需要提供一个 `operator<` 的重载，并具有以下行为：

- 非自反的 $\neg f(a, a)$
- 反对称 $f(a, b) \Rightarrow \neg f(b, a)$
- 及物 $(f(a, b) \wedge f(b, c)) \Rightarrow f(a, c)$

`strict_weak_ordering` 实现的一个良好默认选择是字典序排序。字典序排序也是标准容器所提供的排序方式。

自从 C++20 引入了太空船运算符，用户自定义类型可以轻松地访问字典序排序的默认版本。

为自定义类型实现字典序比较的三种方法示例。

```
struct Point {
    int x;
    int y;

    friend bool operator<(const Point& left, const Point& right) {
        if (left.x != right.x)
            return left.x < right.x;
        return left.y < right.y;
    }

    friend auto operator<=>(const Point&, const Point&) = default;
};

// pre-C++20 lexicographical less-than
// default operator<=> lexicographical comparison
```

```
18      if (left.x != right.x)
19          return left.x <=> right.x;
20      return left.y <=> right.y;
21  }
22
23  };
```

Open in Compiler Explorer

默认的字典顺序（第14行）是递归工作的。它首先从对象的基类开始，按从左到右、深度优先的顺序，然后按声明顺序处理非静态成员（按元素逐个处理数组，从左到右）。

返回的类型是基类和成员的通用比较类别类型，可能是以下之一:

- `std::strong_ordering`
- `std::weak_ordering`
- `std::partial_ordering`

2.3.1 `std::lexicographical_compare`

lex…compare	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

范围的词典序 `strict_weak_ordering` 通过 `std::lexicographical_compare` 算法提供。

constraints		
domain	(`input_range`, `input_range`)	
parallel domain	(`forward_range`, `forward_range`)	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

使用`lexicographical_compare`和内置的“小于”运算符比较整数向量的示例。

```
1  std::vector<int> range1{1, 2, 3};
2  std::vector<int> range2{1, 3};
3  std::vector<int> range3{1, 3, 1};
4
5  bool cmp1 = std::lexicographical_compare(range1.begin(), range1.end(),
6      range2.begin(), range2.end());
7  // same as
8  bool cmp2 = range1 < range2;
9  // cmp1 == cmp2 == true
10
11 bool cmp3 = std::lexicographical_compare(range2.begin(), range2.end(),
12     range3.begin(), range3.end());
```

```
// same as
bool cmp4 = range2 < range3;
// cmp3 == cmp4 == true
```

Open in Compiler Explorer

因为标准容器已经提供了内置的字典序比较，算法主要用于比较原始 C 数组，以及在需要指定自定义比较器的情况下。

Example of using `lexicographical_compare` for C-style arrays and customizing the comparator.

```
// for demonstration only, prefer std::array
int x[] = {1, 2, 3};
int y[] = {1, 4};

bool cmp1 = std::lexicographical_compare(&x[0], &x[3], &y[0], &y[2]);
// cmp1 == true

std::vector<std::string> names1{"Zod", "Celeste"};
std::vector<std::string> names2{"Adam", "Maria"};

bool cmp2 = std::ranges::lexicographical_compare(names1, names2,
    [](const std::string& left, const std::string& right) {
        return left.length() < right.length();
    });
// different than
bool cmp3 = names1 < names2; // Zod > Adam
// cmp2 == true, cmp3 == false
```

Open in Compiler Explorer

2.3.2 `std::lexicographical_compare_three_way`

`std::lexicographical_compare_three_way` 是与 `std::lexicographical_compare` 等价的太空船运算符。它返回以下之一：

- `std::strong_ordering`
- `std::weak_ordering`
- `std::partial_ordering`

类型取决于元素的太空船操作符返回的类型。

lex…three_way	
introduced	C++20
constexpr	C++20
parallel	N/A
rangified	N/A

constraints		
domain	(input_range, input_range)	
invocable	default	custom
	operator<=>	strong_ordering, weak_ordering, partial_ordering

Example of using `std::lexicographical_compare_three_way`.

```
1  std::vector<int> data1 = { 1, 1, 1 };
2  std::vector<int> data2 = { 1, 2, 3 };
3
4  auto cmp = std::lexicographical_compare_three_way(
5      data1.begin(), data1.end(),
6      data2.begin(), data2.end());
7  // cmp == std::strong_ordering::less
```

Open in Compiler Explorer

2.3.3 `std::sort`

sort	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::sort` 算法是经典的 $O(n \log n)$ 排序（通常实现为 intro-sort）。

constraints		
domain	`random_access_range`	
parallel domain	`random_access_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

由于$O(n \log n)$复杂度保证，`std::sort`仅在`random_access`范围上操作。值得注意的是，`std::list`提供了一种具有近似$n \log n$复杂- 度的方法。

Basic example of using `std::sort` and `std::list::sort`.

```
1  std::vector<int> data1 = {9, 1, 8, 2, 7, 3, 6, 4, 5};
2  std::sort(data1.begin(), data1.end());
3  // data1 == {1, 2, 3, 4, 5, 6, 7, 8, 9}
4
5  std::list<int> data2 = {9, 1, 8, 2, 7, 3, 6, 4, 5};
6  // std::sort(data.begin(), data.end()); // doesn't compile
7  data2.sort();
8  // data2 == {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Open in Compiler Explorer

借助 C++20，我们可以利用投影按方法或成员进行排序：

Example of using a projection in conjunction with a range algorithm. The algorithm will sort the elements based on the values obtained by invoking the method `value` on each element.

```
1  struct Account {
2      double value() { return value_; }
3      double value_;
```

```
};

std::vector<Account> accounts{{0.1}, {0.3}, {0.01}, {0.05}};
std::ranges::sort(accounts, std::greater<>{}, &Account::value);
// accounts = { {0.3}, {0.1}, {0.05}, {0.01} }
```

Open in Compiler Explorer

在 C++14 之前，您必须完全指定比较器的类型，即 `std::greater<double>{}`。类型擦除变体 `std::greater<>{}` 依赖于类型推导来确定参数类型。投影接受一元可调用对象，包括指向成员和成员函数的指针。

2.3.4 `std::stable_sort`

`std::sort` 可以自由重新排列等价元素，这在重新排序已排序的范围时可能是不可取的。`std::stable_sort` 提供了额外的保证，即保留相等元素的相对顺序。

stable_sort	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

constraints		
domain	random_access_range	
invocable	default	custom
	operator<	strict_weak_ordering

如果有额外的内存可用，stable_sort 仍然是 $O(n \log n)$。然而，如果分配失败，它将退化为一种 $O(n \log n \log n)$ 算法。

Example of re-sorting a range using `std::stable_sort`, resulting in a guaranteed order of elements.

```
struct Record {
    std::string label;
    int rank;
};

std::vector<Record> data {{"q", 1}, {"f", 1}, {"c", 2},
                          {"a", 1}, {"d", 3}};

std::ranges::stable_sort(data, {}, &Record::label);
std::ranges::stable_sort(data, {}, &Record::rank);
// Guarantted order: a-1, f-1, q-1, c-2, d-3
```

Open in Compiler Explorer

2.3.5 `std::is_sorted`

`std::is_sorted` 算法是一种线性检查，返回一个布尔值，用于表示范围中的元素是否按非递减顺序排列。

is_sorted	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`std::less`	`strict_weak_ordering`

Example of testing a range using `std::is_sorted`.

```
1  std::vector<int> data1 = {1, 2, 3, 4, 5};
2  bool test1 = std::is_sorted(data1.begin(), data1.end());
3  // test1 == true
4
5  std::vector<int> data2 = {5, 4, 3, 2, 1};
6  bool test2 = std::ranges::is_sorted(data2);
7  // test2 == false
8  bool test3 = std::ranges::is_sorted(data2, std::greater<>{});
9  // test3 == true
```

Open in Compiler Explorer

2.3.6 `std::is_sorted_until`

is_sorted_until	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

`std::is_sorted_until` 算法返回给定范围内第一个乱序的元素，从而界定一个已排序的子范围。

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`std::less`	`strict_weak_ordering`

Example of testing a range using `std::is_sorted_until`.

```
1  std::vector<int> data {1, 5, 9, 2, 4, 6};
2  auto it = std::is_sorted_until(data.begin(), data.end());
3  // *it == 2
```

Open in Compiler Explorer

请注意，由于 `std::is_sorted_until` 的行为，以下内容始终为真：

```
std::is_sorted(r.begin(), std::is_sorted_until(r.begin(), r.end()))
```

2.3.7 `std::partial_sort`

partial_sort	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::partial_sort` 算法会重新排列该范围的元素，使得前导子范围的顺序与完全排序时相同。然而，该算法会将其余范围保持为未指定的顺序。

constraints		
domain	(random_access_range, random_access_iterator)	
parallel domain	(random_access_range, random_access_iterator)	
invocable	default	custom
	operator<	strict_weak_ordering

使用部分排序的好处是更快的运行时间——大约是 $O(n * logk)$，其中 k 是排序元素的数量。

Example of using `std::partial_sort` to sort the first three elements of a range.

```
1  std::vector<int> data{9, 8, 7, 6, 5, 4, 3, 2, 1};
2  std::partial_sort(data.begin(), data.begin()+3, data.end());
3  // data == {1, 2, 3, -unspecified order-}
4
5  std::ranges::partial_sort(data, data.begin()+3, std::greater<>());
6  // data == {9, 8, 7, -unspecified order-}
```

Open in Compiler Explorer

2.3.8 `std::partial_sort_copy`

`std::partial_sort_copy` 算法与 `std::partial_sort` 的行为相同；但它不是就地运行的。相反，该算法会将结果写入第二个范围。

partial_sort_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	input_range -> random_access_range	
parallel domain	forward_range -> random_access_range	
invocable	default	custom
	operator<	strict_weak_ordering

将输出写入第二个范围的结果是，源范围不必是可变的，也不必提供随机访问。

Example of using `std::partial_sort_copy` to iterate over ten integers read from standard input and storing the top three values in sorted order.

```
1  // input == "0 1 2 3 4 5 6 7 8 9"
2  std::vector<int> top(3);
3
4  auto input = std::istream_iterator<int>(std::cin);
5  auto cnt = std::counted_iterator(input, 10);
6
7  std::ranges::partial_sort_copy(cnt, std::default_sentinel,
8      top.begin(), top.end(),
9      std::greater<>{});
10 // top == { 9, 8, 7 }
```

Open in Compiler Explorer

2.3.9 `qsort` - C 标准库

因为 C 标准库是 C++ 标准库的一部分，我们也可以使用 `qsort`。

Example of using `qsort` to sort an array of integers.

```
int data[] = {2, 1, 9, -1, 7, -8};
int size = sizeof data / sizeof(int);

qsort(data, size, sizeof(int),
    [](const void* left, const void* right){
        int vl = *(const int*)left;
        int vr = *(const int*)right;

        if (vl < vr) return -1;
        if (vl > vr) return 1;
        return 0;
    });
// data == {-8, -1, 1, 2, 7, 9}
```

Open in Compiler Explorer

我强烈建议避免使用 `qsort`，因为在任何情况下，`std::sort` 和 `std::ranges::sort` 都应该是更好的选择。此外，`qsort` 仅对平凡可拷贝类型有效，而且即使使用 `std::sort`，这些类型也会被正确地优化为 `memcpy` / `memmove` 操作。

Example of using `std::sort` to achieve the same result as in the previous example.

```
int data[] = {2, 1, 9, -1, 7, -8};
int size = sizeof data / sizeof(int);

std::sort(&data[0], &data[size], std::less<>());
// data == {-8, -1, 1, 2, 7, 9}
```

Open in Compiler Explorer

2.4 划分

分区算法根据谓词对范围内的元素重新排列，使得谓词返回 `true` 的元素位于谓词返回 `false` 的元素之前。

当我们需要根据某个特定属性对元素进行分组时，经常会用到划分（partitioning）。如果我们基于一个布尔属性的取值进行排序，也可以把划分看作等同于排序。

2.4.1 `std::partition`

`std::partition` 算法提供基本的分区功能，基于一元谓词对元素进行重新排序。该算法返回分区点，即一个迭代器，指向谓词返回 `false` 的第一个元素。

partition	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::partition` to process exam results.

```
std::vector<ExamResult> results = get_results();

auto pp = std::partition(results.begin(), results.end(),
    [threshold = 49](const auto& r) {
        return r.score >= threshold;
    });

// process passing students
for (auto it = results.begin(); it != pp; ++it) {
    std::cout << "[PASS] " << it->student_name << "\n";
}
// process failed students
for (auto it = pp; it != results.end(); ++it) {
    std::cout << "[FAIL] " << it->student_name << "\n";
}
```

Open in Compiler Explorer

2.4.2 `std::stable_partition`

`std::partition` 算法被允许重新排列元素，唯一的保证是：谓词求值为 `true` 的元素将先于谓词求值为 `false` 的元素。这种行为在某些情况下可能并不理想，例如对于 UI 元素。

`std::stable_partition` 算法增加了在两个分区中保持元素相对顺序的保证。

stable_partition	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

constraints		
domain	`bidirectional_range`	
parallel domain	`bidirectional_range`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::stable_partition` to move selected items to the beginning of a list.

```
auto& widget = get_widget();
std::ranges::stable_partition(widget.items, &Item::is_selected);
```

Open in Compiler Explorer

2.4.3 `std::is_partitioned`

is_partitioned	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

`std::is_partitioned` 算法是一种线性检查，返回一个布尔值，用于表示是否根据谓词对范围内的元素进行了分区。

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

请注意，对于任何可能的值，已排序的范围总是已经被分区（只是分区点不同）。

Example of using `std::is_partitioned`.

```
std::vector<int> data{2, 4, 6, 7, 9, 11};

auto is_even = [](int v) { return v % 2 == 0; };
bool test1 = std::ranges::is_partitioned(data, is_even);
// test1 == true

bool test2 = true;
for (int i = 0; i < 16; ++i) {
    test2 = test2 && std::is_partitioned(data.begin(), data.end(),
        [&i](int v) { return v < i; });
}
// test2 == true
```

Open in Compiler Explorer

2.4.4 `std::partition_copy`

partition_copy	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

`std::partition_copy` 是 `std::partition` 的一种变体，它不是对元素进行重新排序，而是将分区后的元素输出到由两个迭代器表示的两个输出范围中。

constraints		
domain	`input_range -> (output_iterator, output_iterator)`	
parallel domain	`forward_range -> (forward_iterator, forward_iterator)`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::partition_copy` to copy even elements into one range and odd elements into another range.

```
1  std::vector<int> data{2, 4, 6, 1, 3, 5};
2  auto is_even = [](int v) { return v % 2 == 0; };
3
4  std::vector<int> even, odd;
5  std::partition_copy(data.begin(), data.end(),
6      std::back_inserter(even),
7      std::back_inserter(odd),
8      is_even);
9
10 // even == {2, 4, 6}
11 // odd == {1, 3, 5}
```

Open in Compiler Explorer

2.4.5 `std::nth_element`

`nth_element` 算法是一种分区算法，它确保第 n 个位置上的元素就是如果对该范围进行排序后会位于该位置的元素。

第 n 个元素还将该范围划分为 `[begin, nth)`、`[nth, end)`，使得位于第 n 个元素之前的所有元素都小于或等于该范围中其余的元素。或者，对于任意元素 $i \in$ `[begin, nth)` 和 $j \in$ `[nth, end)`，都有 $\neg(j < i)$ 成立。

nth_element	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`(random_access_range, random_access_iterator)`	
parallel domain	`(random_access_range, random_access_iterator)`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

由于其选择/分区的特性，`std::nth_element` 在理论复杂度上优于 `std::partial_sort` - $O(n)$ 对比 $O(n * logk)$。

不过需要注意的是，该标准仅要求平均 $O(n)$ 复杂度，而 `std::nth_element` 的实现可能具有较高的开销，因此务必进行测试，以确定哪一种在你的使用场景中能提供更好的性能。

Example of using `std::nth_element`.

```
1  std::vector<int> data{9, 1, 8, 2, 7, 3, 6, 4, 5};
2  std::nth_element(data.begin(), data.begin() + 4, data.end());
3  // data[4] == 5, data[0..3] < data[4]
4
5  std::nth_element(data.begin(), data.begin() + 7, data.end(),
6      std::greater<>());
7  // data[7] == 2, data[0..6] > data[7]
```

Open in Compiler Explorer

2.5 分而治之

分治算法提供了性能和功能的极佳结合。

虽然我们可以利用基于哈希的容器在$O(1)$摊销时间内查找任何特定元素，但这种方法有两个缺点。首先，我们只能查找特定的元素；如果该元素不在容器中，我们会得到一个简单的查找失败。其次，我们的类型必须是可哈希的，并且哈希函数必须足够快速。

分治算法允许基于严格弱排序查找边界，并且即使容器中没有特定的值时也能工作。此外，由于我们使用的是已排序的容器，一旦确定了边界，我们可以轻松访问相邻的值。

2.5.1 `std::lower_bound`，`std::upper_bound`

lower_bound	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

upper_bound	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

`std::lower_bound` 和 `std::upper_bound` 算法为随机访问范围提供具有对数复杂度的边界搜索。

constraints		
domain	`forward_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

这两种算法在返回的界上有所不同：

- `std::lower_bound` 返回第一个 `elem < value` 返回 false 的元素（即第一个 `elem >= value` 的元素）
- 该 `std::upper_bound` 返回满足 `value < elem` 的第一个元素
- 如果不存在这样的元素，这两个算法都会返回 end 迭代器

Example of using `std::lower_bound` and `std::upper_bound` to divide a sorted range into three parts: lower than the bottom threshold, between the bottom and upper threshold and higher than the upper threshold.

```
1  const std::vector<ExamResult>& results = get_results();
2
3  auto lb = std::ranges::lower_bound(results, 49, {},
4                                     &ExamResult::score);
5  // First element for which: it->score >= 49
6  auto ub = std::ranges::upper_bound(results, 99, {},
7                                     &ExamResult::score);
8  // First element for which: 99 < it->score
9
10 for (auto it = results.begin(); it != lb; it++) {
11     // Process exam fails, upto 48
```

```
}

for (auto it = lb; it != ub; it++) {
    // Process exam passes, 49-99
}

for (auto it = ub; it != results.end(); it++) {
    // Process exams with honors, 100+
}
```

Open in Compiler Explorer

虽然这些算法可以在任何 `forward_range` 上运行，但对数级的分治行为仅适用于 `random_access_range`。像 `std::set`、`std::multiset`、`std::map` 和 `std::multimap` 这样的数据结构以方法的形式提供其下界和上界的 $O(\log n)$ 实现。

Example of using `lower_bound` and `upper_bound` methods on a `std::multiset`.

```
std::multiset<int> data{1, 2, 3, 4, 5, 6, 6, 6, 7, 8, 9};

auto lb = data.lower_bound(6);
// std::distance(data.begin(), lb) == 5, *lb == 6

auto ub = data.upper_bound(6);
// std::distance(data.begin(), ub) == 8, *ub == 7
```

Open in Compiler Explorer

2.5.2 `std::equal_range`

equal_range	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

该 `std::equal_range` 算法返回给定值的下界和上界。

constraints		
domain	`forward_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

因为下界返回满足 `elem >= value` 的第一个元素，而上界返回满足 `value < elem` 的第一个元素，所以结果是一个由等于该值的元素组成的区间 `[lb, ub)`。

Example of using `std::equal_range`.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 6, 6, 7, 8, 9};

auto [lb, ub] = std::equal_range(data.begin(), data.end(), 6);
```

```
4  // std::distance(data.begin(), lb) == 5, *lb == 6
5  // std::distance(data.begin(), ub) == 8, *ub == 7
```

在 Compiler Explorer 中打开

2.5.3 `std::partition_point`

partition_point	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++20

尽管名称如此，`std:partition_point` 的工作方式与 `std::upper- _bound` 非常相似，不过它不是搜索某个特定的值，而是使用谓词进行搜索。

constraints		
domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

`std::partition_point` 将返回第一个不满足提供的谓词的元素。此算法仅要求范围根据谓词进行分区。

Example of using `std::partition_point`.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
2  auto pp = std::partition_point(data.begin(), data.end(),
3      [](int v) { return v < 5; });
4  // *pp == 5
```

Open in Compiler Explorer

2.5.4 `std::binary_search`

binary_search	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

`std::binary_search` 提供存在性检查，返回一个布尔值，用于指示请求的值是否存在于排序范围中。

constraints		
domain	`forward_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

使用 `std::binary_search` 等同于调用 `std::equal_range` 并检查返回结果是否非空；然而，`std::binary_search` 提供一次查找的性能，而 `std::equal_range` 需要进行两次查找来确定下界和上界。

Example of using `std::binary_search` with an equivalent check using `std::equal_range`.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};

bool exists = std::ranges::binary_search(data, 5);
// exists == true
auto [lb, ub] = std::ranges::equal_range(data, 5);
// lb != ub, i.e. the value is in the range
```

Open in Compiler Explorer

2.5.5 **bsearch** - C 标准库

在 C 标准库中，C++ 继承自 `bsearch`。该算法返回一个与所提供键相等的元素之一；如果未找到这样的元素，则返回 `nullptr`。

Example of using `bsearch`.

```
int data[] = {-2, -1, 0, 1, 2};
int size = sizeof data / sizeof(int);

auto cmp = [](const void* left, const void* right){
    int vl = *(const int*)left;
    int vr = *(const int*)right;

    if (vl < vr) return -1;
    if (vl > vr) return 1;
    return 0;
};

int value = 1;
void* el1 = bsearch(&value, data, size, sizeof(int), cmp);
// *static_cast<int*>(el1) == 1

value = 3;
void *el2 = bsearch(&value, data, size, sizeof(int), cmp);
// el2 == nullptr
```

Open in Compiler Explorer

与 `qsort` 一样，在 C++ 代码中实际上没有理由使用 `bsearch`。根据具体的使用场景，前文提到的算法之一应当是合适的替代方案。

Example demonstrating alternatives to bsearch.

```
int data[] = {-2, -1, 0, 1, 2};
int size = sizeof data / sizeof(int);

int value = 1;
bool exist = std::binary_search(&data[0], &data[size], value);
// exist == true

auto candidate = std::lower_bound(&data[0], &data[size], value);
if (candidate != &data[size] && *candidate == value) {
    // process element
}

auto [lb, ub] = std::equal_range(&data[0], &data[size], value);
if (lb != ub) {
    // process equal elements
}
```

Open in Compiler Explorer

2.6 排序范围上的线性运算

在本节中，我们将讨论在排序区间上以线性时间运行的算法。在未排序区间上执行相同功能则需要在二次时间复杂度下运行的算法。

2.6.1 `std::includes`

includes	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::includes` 算法将确定一个范围（所有元素）是否包含在另一个范围内。

constraints		
domain	(input_range, input_range)	
parallel domain	(forward_range, forward_range)	
invocable	default	custom
	operator<	strict_weak_ordering

```
```

在编译器探索器中打开 使

用std::includes检查字符串是否包含所有英文字母（小写字母）的示例。

```
std::vector<char> letters('z'-'a'+1,'\0');
std::iota(letters.begin(), letters.end(), 'a');
std::string input = "the quick brown fox jumps over the lazy dog";
std::ranges::sort(input);                          7  // test == true

bool test = std::ranges::includes(input, letters);
```

该示例使用 `std::iota` 生成小写字母（第 2 行），这要求目标向量事先进行预分配（第 1 行）。

2.6.2 `std::merge`

merge	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::merge` 算法合并两个已排序的区间，输出写入第三个区间，该区间不能与任一输入区间重叠。

constraints		
domain	`(input_range, input_range) -> output_iterator`	
parallel domain	`(forward_range, forward_range) -> forward_iterator`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

合并操作是稳定的。来自第一个范围的相等元素将在来自第二个范围的相等元素之前排序。

Example of using `std::merge`.

```
1  struct LabeledValue {
2      int value;
3      std::string label;
4  };
5
6  std::vector<LabeledValue> data1{
7      {1, "first"}, {2, "first"}, {3, "first"}};
8  std::vector<LabeledValue> data2{
9      {0, "second"}, {2, "second"}, {4, "second"}};
10
11 std::vector<LabeledValue> result;
12 auto cmp = [](const auto& left, const auto& right)
13     { return left.value < right.value; };
14
15 std::ranges::merge(data1, data2, std::back_inserter(result), cmp);
16 // result == {0, second}, {1, first}, {2, first},
17 //           {2, second}, {3, first}, {4, second}
```

Open in Compiler Explorer

并行版本要求输出为一个正向范围（由`forward_iterator`表示）。因此，我们不能使用像`std::back_inserter`这样的包装器，并且必须预分配足够容量的输出范围。

Example of parallel `std::merge`.

```
1  std::vector<int> data1{1, 2, 3, 4, 5, 6};
2  std::vector<int> data2{3, 4, 5, 6, 7, 8};
3
4  std::vector<int> out(data1.size()+data2.size(), 0);
5  std::merge(std::execution::par_unseq,
6      data1.begin(), data1.end(),
7      data2.begin(), data2.end(),
8      out.begin());
9  // out == {1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 8}
```

Open in Compiler Explorer

inplace_merge	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

2.6.3 `std::inplace_merge`

`std::inplace_merge` 算法合并两个连续的子范围。

constraints		
domain	`(bidirectional_range, bidirectional_iterator)`	
parallel domain	`(bidirectional_range, bidirectional_iterator)`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

在使用基于迭代器的接口时，中间迭代器（即指向第二个子范围中第一个元素的迭代器）是第二个参数。

Example of using `std::inplace_merge`.

```
1  std::vector<int> range{1, 3, 5, 2, 4, 6};
2  std::inplace_merge(range.begin(), range.begin()+3, range.end());
3  // range == { 1, 2, 3, 4, 5, 6 }
```

Open in Compiler Explorer

2.6.4 `std::unique`, `std::unique_copy`

`std::unique` 算法会移除连续的重复值。典型的使用场景是与已排序的范围配合使用。然而，这并不是 `std::unique` 的必需条件。

unique	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==`	`binary_predicate`

因为 unique 在原地工作并且不能调整底层范围的大小，它会留下范围的末尾未指定的值，并返回一个指向该子范围起始位置的迭代器。

使用 std::unique 的示例。

```
1  std::vector<int> data{1, 1, 2, 2, 3, 4, 5, 6, 6, 6};
2
3  auto it = std::unique(data.begin(), data.end());
4  // data == {1, 2, 3, 4, 5, 6, unspec, unspec, unspec, unspec}
5
6  data.resize(std::distance(data.begin(), it));
7  // data == {1, 2, 3, 4, 5, 6}
```

在 Compiler Explorer 中打开

std::unique_copy 是 std::unique 的一种变体，它将唯一值输出到第二个范围。

unique_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	input_range -> output_iterator	
parallel domain	forward_range -> forward_iterator	
invocable	default	custom
	operator==	binary_predicate

Example of using std::unique_copy.

```
1  std::vector<int> data{1, 1, 2, 2, 3, 4, 5, 6, 6, 6};
2  std::vector<int> out;
3
4  std::ranges::unique_copy(data, std::back_inserter(out));
5  // out == {1, 2, 3, 4, 5, 6}
```

Open in Compiler Explorer

2.7 集合运算

这组集合算法在两个已排序的范围上模拟不同的集合操作。

2.7.1 std::set_difference

std::set_difference 算法生成一个范围，其中包含存在于第一个范围但不在第二个范围中的元素。

set_difference	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	(input_range, input_range) -> output_iterator	
parallel domain	(forward_range, forward_range) -> forward_iterator	
invocable	default	custom
	operator<	strict_weak_ordering

Example of using `std::set_difference`.

```
std::vector<int> data1{1, 2, 3, 4, 5, 6};
std::vector<int> data2{3, 4, 5};

std::vector<int> difference;
std::ranges::set_difference(data1, data2,
    std::back_inserter(difference));
// difference == {1, 2, 6}
```

Open in Compiler Explorer

对于等价元素，当第一个范围包含 M 个此类元素且第二个范围包含 N 个此类元素时，结果将包含来自第一个范围的最后 `std::max(M-N,0)` 个此类元素。

Example demonstrating `std::set_difference` behaviour when equivalent elements are present.

```
struct Labeled {
    std::string label;
    int value;
};

auto cmp = [](const auto& l, const auto& r) {
    return l.value < r.value;
};

std::vector<Labeled> equal1{{"first_a", 1}, {"first_b", 1},
                            {"first_c", 1}, {"first_d", 1}};
std::vector<Labeled> equal2{{"second_a", 1}, {"second_b", 1}};

std::vector<Labeled> equal_difference;
std::ranges::set_difference(equal1, equal2,
    std::back_inserter(equal_difference), cmp);
// equal_difference == { {"first_c", 1}, {"first_d", 1} }
```

Open in Compiler Explorer

2.7.2 `std::set_symmetric_difference`

set_sym…diff…	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::set_symetric_difference` 算法生成一个范围，其中包含仅存在于其中一个范围而不同时存在于两个范围中的元素。

constraints		
domain	`(input_range, input_range) -> output_iterator`	
parallel domain	`(forward_range, forward_range) -> forward_iterator`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `std::set_symmetric_difference`.

```
1  std::vector<int> data1{1, 3, 5, 7, 9};
2  std::vector<int> data2{3, 4, 5, 6, 7};
3
4  std::vector<int> symmetric_difference;
5  std::ranges::set_symmetric_difference(data1, data2,
6      std::back_inserter(symmetric_difference));
7  // symmetric_difference == {1, 4, 6, 9}
```

Open in Compiler Explorer

对于等价元素，当第一个范围包含 M 个此类元素且第二个范围包含 N 个此类元素时，结果将包含来自相应范围的最后 `std::abs(M-N)` 个此类元素。也就是说，如果 $M > N$，将从第一个范围复制 $M - N$ 个元素；否则，将从第二个范围复制 $N - M$ 个元素。

Example demonstrating `std::set_symmetric_difference` behaviour when equivalent elements are present.

```
1  struct Labeled {
2      std::string label;
3      int value;
4  };
5
6  auto cmp = [](const auto& l, const auto& r) {
7      return l.value < r.value;
8  };
9
10 std::vector<Labeled> equal1{{"first_a", 1}, {"first_b", 2},
11                               {"first_c", 2}};
12 std::vector<Labeled> equal2{{"second_a", 1}, {"second_b", 1},
13                               {"second_c", 2}};
14
15 std::vector<Labeled> equal_symmetric_difference;
16 std::ranges::set_symmetric_difference(equal1, equal2,
17     std::back_inserter(equal_symmetric_difference), cmp);
18 // equal_symmetric_difference == { {"second_b", 1}, {"first_c", 2} }
```

Open in Compiler Explorer

2.7.3 `std::set_union`

`std::set_union` 算法生成一个包含出现在任一范围中的元素的范围。

set_union	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`(input_range, input_range) -> output_iterator`	
parallel domain	`(forward_range, forward_range) -> forward_iterator`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `std::set_union`.

```
1  std::vector<int> data1{1, 3, 5};
2  std::vector<int> data2{2, 4, 6};
3
4  std::vector<int> set_union;
5  std::ranges::set_union(data1, data2,
6      std::back_inserter(set_union));
7  // set_union == { 1, 2, 3, 4, 5, 6 }
```

Open in Compiler Explorer

对于等价元素，当第一个范围包含 M 个此类元素而第二个范围包含 N 个此类元素时，结果将包含来自第一个范围的 M 个元素，随后是来自第二个范围的最后 `std::max(N-M,0)` 个元素。

Example demonstrating `std::set_union` behaviour when equivalent elements are present.

```
1  struct Labeled {
2      std::string label;
3      int value;
4  };
5
6  auto cmp = [](const auto& l, const auto& r) {
7      return l.value < r.value;
8  };
9
10 std::vector<Labeled> equal1{{"first_a", 1}, {"first_b", 1},
11                               {"first_c", 2}};
12 std::vector<Labeled> equal2{{"second_a", 1}, {"second_b", 2},
13                               {"second_c", 2}};
14
15 std::vector<Labeled> equal_union;
16 std::ranges::set_union(equal1, equal2,
17     std::back_inserter(equal_union), cmp);
18 // equal_union == { {"first_a", 1}, {"first_b", 1},
19 //                  {"first_c", 2}, {"second_c", 2} }
```

Open in Compiler Explorer

2.7.4 `std::set_intersection`

set_intersection	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::set_intersection` 算法生成一个包含同时存在于两个范围中的元素的范围。

constraints		
domain	`(input_range, input_range) -> output_iterator`	
parallel domain	`(forward_range, forward_range) -> forward_iterator`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `std::set_intersection`.

```
1  std::vector<int> data1{1, 2, 3, 4, 5};
2  std::vector<int> data2{2, 4, 6};
3
4  std::vector<int> intersection;
5  std::ranges::set_intersection(data1, data2,
6      std::back_inserter(intersection));
7  // intersection == {2, 4}
```

Open in Compiler Explorer

对于等价元素，当第一个范围包含 M 个此类元素且第二个范围包含 N 个此类元素时，结果将包含来自第一个范围的前 `std::min(M,N)` 个元素。

Example demonstrating `std::set_intersection` behaviour when equivalent elements are present.

```
1  struct Labeled {
2      std::string label;
3      int value;
4  };
5
6  auto cmp = [](const auto& l, const auto& r) {
7      return l.value < r.value;
8  };
9
10 std::vector<Labeled> equal1{{"first_a", 1}, {"first_b", 2}};
11 std::vector<Labeled> equal2{{"second_a", 1}, {"second_b", 2},
12                              {"second_c", 2}};
13
14 std::vector<Labeled> intersection;
15 std::ranges::set_intersection(equal1, equal2,
16     std::back_inserter(intersection), cmp);
17 // intersection == { {"first_a", 1}, {"first_b", 2} }
```

Open in Compiler Explorer

2.8 变换算法

在本节中，我们将讨论通过改变元素的值以及删除和重新排序元素来变换范围的算法。

2.8.1 `std::transform`

transform	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

最直接的转换方式是对每个元素应用一个转换函数。`std::transform` 算法提供了这种功能，具有一元和二元两种变体（输入来自一个或两个范围）。

constraints		
domain	`input_range -> output_iterator` `(input_range, input_iterator) -> output_iterator`	
parallel domain	`forward_range -> forward_iterator` `(forward_range, forward_iterator) -> forward_iterator`	
invocable	default	custom
	N/A	`unary_functor` `binary_functor`

Example of unary and binary version of `std::transform`. Note that the output iterator can be one of the input ranges' begin iterator (line 4 and 12).

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8};
2
3  std::transform(data.begin(), data.end(),
4      data.begin(),
5      [](int v) { return v*2; });
6  // data == {2, 4, 6, 8, 10, 12, 14, 16}
7
8  std::vector<int> add{8, 7, 6, 5, 4, 3, 2, 1};
9
10 std::transform(data.begin(), data.end(),
11     add.begin(),
12     data.begin(),
13     [](int left, int right) { return left+right; });
14 // data == {10, 11, 12, 13, 14, 15, 16, 17}
```

Open in Compiler Explorer

请注意，`std::transform` 并不保证严格的从左到右求值。如果需要这样做，请改用 `std::for_each`。

2.8.2 `std::remove`, `std::remove_if`

remove, remove_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::remove` 和 `std::remove_if` 算法会“移除”与给定值匹配的元素，或使给定谓词求值为 true 的元素。

由于这些算法无法调整底层区间的大小，删除操作是通过移动区间中的其他元素来实现的。随后，这些算法会返回一个指向最后一个未被移除元素之后的迭代器，即新的末尾迭代器。

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::remove` and `std::remove_if`.

```
1  std::vector<int> data{1, 2, 3, 4, 5};
2
3  auto it = std::remove(data.begin(), data.end(), 3);
4  // data == { 1, 2, 4, 5, ?}
5
6  data.erase(it, data.end()); // Erase sub-range
7  // data == {1, 2, 4, 5}
8
9  auto is_even = [](int v) { return v % 2 == 0; };
10 it = std::remove_if(data.begin(), data.end(), is_even);
11 // data == {1, 5, ?, ?}
12
13 data.resize(it - data.begin()); // Random Access Ranges only
14 // data = {1, 5}
```

Open in Compiler Explorer

2.8.3 `std::replace`, `std::replace_if`

`std::replace` 和 `std::replace_if` 算法会替换与给定值匹配的元素，或给定谓词求值为 true 的元素。

replace, replace_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::replace` and `std::replace_if`.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7};
2
3  std::ranges::replace(data, 4, 0);
4  // data == {1, 2, 3, 0, 5, 6, 7}
5
6  auto is_odd = [](int v) { return v % 2 != 0; };
7  std::ranges::replace_if(data, is_odd, -1);
8  // data == {-1, 2, -1, 0, -1, 6, -1}
```

Open in Compiler Explorer

2.8.4 `std::reverse`

`std::reverse` 算法通过将 `std::swap` 应用于成对的元素来反转该范围内元素的顺序。

reverse	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints	
domain	`bidirectional_range`
parallel domain	`bidirectional_range`

请注意，只有在必须对范围进行变异时，使用 `std::reverse` 才是一个合理的解决方案，因为双向范围已经支持反向迭代。

Example of using `std::reverse` and reverse iteration, provided by bidirectional ranges.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7};
2
3  std::reverse(data.begin(), data.end());
4  // data == {7, 6, 5, 4, 3, 2, 1}
5
6  for (auto it = data.rbegin(); it != data.rend(); ++it) {
7      // iterate over: 1, 2, 3, 4, 5, 6, 7
8  }
```

Open in Compiler Explorer

C 风格数组和 C 风格字符串可以使用 `std::span` 和 `std::string- _view` 进行适配，以允许反向迭代。

Example of using `std::span` and `std::string_view` to addapt C-style constructs for reverse iteration.

```
1  int c_array[] = {1, 2, 3, 4, 5, 6, 7};
2  auto arr_view = std::span(c_array, sizeof(c_array)/sizeof(int));
3
4  for (auto it = arr_view.rbegin(); it != arr_view.rend(); ++it) {
5      // iterate over: {7, 6, 5, 4, 3, 2, 1}
6  }
7
8  const char* c_string = "No lemon, no melon";
9  auto str_view = std::string_view(c_string);
10
11 for (auto it = str_view.rbegin(); it != str_view.rend(); ++it) {
12     // iterate over: "nolem on ,nomel oN"
13 }
```

Open in Compiler Explorer

2.8.5 `std::rotate`

rotate	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

`std::rotate` 算法会重新排列从 `[first, middle)`, `[middle, last)` 到 `[middle, last)`, `[first, middle)` 范围内的元素。

constraints	
domain	(forward_range, forward_iterator)
parallel domain	(forward_range, forward_iterator)

Example of using `std::rotate`.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7};
2  std::rotate(data.begin(), data.begin()+3, data.end());
3  // data == {4, 5, 6, 7, 1, 2, 3}
```

Open in Compiler Explorer

2.8.6 `std::shift_left`, `std::shift_right`

`std::shift_left` 和 `std::shift_right` 算法将元素在提供的范围内移动指定数量的位置。然而，与 std::rotate 不同，移位不会绕回。

shift_left	
introduced	C++20
constexpr	C++20
parallel	C++20
rangified	C++20

shift_right	
introduced	C++20
constexpr	C++20
parallel	C++20
rangified	C++20

constraints	
domain	`forward_range`
parallel domain	`forward_range`

使用 `std::shift_left` 和 `std::shift_right` 的示例，适用于一个可以轻松复制的类型。

```
1  std::vector<int> data{1,2,3,4,5,6,7,8,9};
2  std::shift_left(data.begin(), data.end(), 3);
3  // data == {4, 5, 6, 7, 8, 9, 7, 8, 9}
4
5  data = {1,2,3,4,5,6,7,8,9};
6  std::shift_right(data.begin(), data.end(), 3);
7  // data == {1, 2, 3, 1, 2, 3, 4, 5, 6}
```

在 Compiler Explorer 中打开

一种思考这些算法的方式是，它们为请求的新元素数量“腾出空间”。

Example of using `std::shift_right` to make space for four new elements. Note that `'d'` wasn't moved as there is no place to move it to, and in the context of "making space" will be overwritten anyway.

```
1  struct EmptyOnMove {
2      char value;
3      EmptyOnMove(char value) : value(value) {}
4      EmptyOnMove(EmptyOnMove&& src) :
5          value(std::exchange(src.value,'-')) {}
6      EmptyOnMove& operator=(EmptyOnMove&& src) {
7          value = std::exchange(src.value, '-');
8          return *this;
9      }
10     EmptyOnMove(const EmptyOnMove&) = default;
11     EmptyOnMove& operator=(const EmptyOnMove&) = default;
```

```
};

int main() {
    std::vector<EmptyOnMove> nontrivial{
        {'a'},{'b'},{'c'},{'d'},{'e'},{'f'},{'g'}};

    std::shift_right(nontrivial.begin(), nontrivial.end(), 4);
    // nontrivial == { '-', '-', '-', 'd', 'a', 'b', 'c' }
}
```

Open in Compiler Explorer

2.8.7 `std::shuffle`

shuffle	
introduced	C++11
constexpr	N/A
parallel	N/A
rangified	C++20

`std::shuffle` 算法是现已废弃的（在 C++ 14 中被弃用，在 C++17 中被移除）`std::random_shuffle` 算法的继任者，并依赖于在 C++11 中新增的随机数设施。

约束条件	
domain	random_access_range

Example of using `std::shuffle` to shuffle a deck of cards.

```
struct Card {
    unsigned index;

friend std::ostream& operator << (std::ostream& s, const Card& card) {
    static constexpr std::array<const char*, 13> ranks =
        {"Ace", "Two", "Three", "Four", "Five", "Six", "Seven",
         "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
    static constexpr std::array<const char*, 4> suits =
        {"Hearts", "Diamonds", "Clubs", "Spades"};

    if (card.index >= 52)
        throw std::domain_error(
            "Card index has to be in the range 0..51");

    s << ranks[card.index%13] << " of " << suits[card.index/13];
    return s;
}

};

int main() {

std::vector<Card> deck(52, Card{});
std::ranges::generate(deck, [i = 0u]() mutable { return Card{i++}; });
// deck == {Ace of Hearts, Two of Hearts, Three of Hearts, Four...}

std::random_device rd;
```

```
28  std::mt19937 gen{rd()};
29
30  std::ranges::shuffle(deck, gen);
31  // deck == { random order }
32
33  }
```

Open in Compiler Explorer

2.8.8 `std::next_permutation`, `std::prev_permutation`

`std::next_permutation` 和 `std::prev_permutation` 算法将重新排列元素，使其成为下一个或上一个（按字典序比较）排列。当不存在这样的顺序时，这两个算法都会回绕，但也会返回 `false` 以通知调用者。

next_permutation	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

prev_permutation	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

constraints		
domain	`bidirectional_range`	
invocable	default	custom
	`operator <`	`strict_weak_ordering`

Example of using `std::next_permutation` to iterate over all permutations of three unique elements.

```
1  std::vector<int> data{1, 2, 3};
2  do {
3      // iterate over:
4      // 1, 2, 3
5      // 1, 3, 2
6      // 2, 1, 3
7      // 2, 3, 1
8      // 3, 1, 2
9      // 3, 2, 1
10 } while (std::next_permutation(data.begin(), data.end()));
11 // data == {1, 2, 3}
```

Open in Compiler Explorer

2.8.9 `std::is_permutation`

`std::is_permutation` 算法是一个出色的工具，用于检查两个范围是否具有相同的内容，但不一定具有相同的元素顺序。

is_permutation	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`(forward_range, forward_range)`	
parallel domain	`(forward_range, forward_range)`	
invocable	default	custom
	`operator==`	`binary_predicate`

Example of using `std::is_permutation` to validate a simple sort implementation.

```
std::vector<int> data = { 8, 1, 7, 3, 4, 6, 2, 5};
for (size_t i = 0; i < data.size()-1; ++i)
    for (size_t j = i+1; j < data.size(); ++j)
        if (data[i] > data[j])
            std::swap(data[i], data[j]);

bool is_sorted = std::ranges::is_sorted(data);
// is_sorted == true

bool is_permutation = std::ranges::is_permutation(data,
    std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8});
// is_permutation == true
```

Open in Compiler Explorer

2.9 左折叠

左折叠是数值算法的两大类别之一。折叠以严格的顺序运行，通过对每个元素求值`acc=fold_op(acc,el)`，一次“折叠”一个元素。对于左折叠，操作方向是从左到右。

由于严格的线性操作，任何左折叠算法都不支持并行版本。

在使用数值算法时，了解C++中数值类型的行为是很有价值的，特别是静默隐式转换和模板类型推导规则之间的相互作用。你可以在理论章节的5.2节中阅读更多相关内容。

accumulate	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	N/A

2.9.1 `std::accumulate`

`std::accumulate` 算法是单范围左折叠。

constraints		
domain	`input_range`	
invocable	default	custom
	`operator +`	`binary_functor`

Example of using `std::accumulate`.

```
std::vector<int> data{1, 2, 3, 4, 5};
auto sum = std::accumulate(data.begin(), data.end(), 0);
// sum == 15

auto product = std::accumulate(data.begin(), data.end(), 1,
                               std::multiplies<>{});
// product == 120
```

Open in Compiler Explorer

虽然左折叠严格按从左到右运行，但我们可以通过使用反向迭代器来缓解这一限制——当然，这至少要求该范围是双向的。

Example of using `std::accumulate` as a right fold.

```
std::vector<int> data{1, 2, 3, 4, 5};

auto left_fold = std::accumulate(data.begin(), data.end(), 0,
    [](int acc, int el) {
        return acc / 2 + el;
    });
// left_fold == 8

auto right_fold = std::accumulate(data.rbegin(), data.rend(), 0,
    [](int acc, int el) {
        return acc / 2 + el;
    });
// right_fold == 3
```

Open in Compiler Explorer

2.9.2 `std::inner_product`

inner_product	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	N/A

`std::inner_product` 算法是对两个范围进行的左折叠。元素对先进行归约，然后再累积。

constraints		
domain	`(input_range, input_iterator)`	
invocable	default	custom
	`operator *, operator +`	`(binary_functor, binary_functor)`

默认版本使用 `operator*` 进行约简操作，使用 `operator+` 进行折叠操作。

Example of using `std::inner_product`.

```
std::vector<int> heights{1, 2, 3, 4, 5};
std::vector<int> widths{2, 3, 4, 5, 6};

auto total_area = std::inner_product(heights.begin(), heights.end(),
    widths.begin(), 0);
// total_area == 70
```

Open in Compiler Explorer

由于该算法仅使用区间作为输入，因此不存在区间重叠的问题。

Example of using `std::inner_product` on a single range to calculate sum of absolute differences between elements.

```
1  std::vector<int> data{6, 4, 3, 7, 2, 1};
2  auto sum_of_diffs = std::inner_product(
3      data.begin(), std::prev(data.end()),
4      std::next(data.begin()), 0,
5      std::plus<>{},
6      [](int left, int right) { return std::abs(left-right); }
7  );
8  // sum_of_diffs == 13
```

Open in Compiler Explorer

2.9.3 `std::partial_sum`

partial_sum	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	N/A

`std::partial_sum` 算法以左折叠的方式运行。然而，它不会将范围规约为单一值。相反，该算法会将每个部分结果写入输出范围。

constraints		
domain	`input_range -> output_iterator`	
invocable	default	custom
	`operator+`	`binary_functor`

输出迭代器可以是输入范围的起始迭代器。

Example of using `std::partial_sum`.

```
1  std::vector<int> data(6, 1);
2  // data == {1, 1, 1, 1, 1, 1}
3
4  std::partial_sum(data.begin(), data.end(), data.begin());
5  // data == {1, 2, 3, 4, 5, 6}
6
7  std::vector<int> out;
8  std::partial_sum(data.begin(), data.end(),
9      std::back_inserter(out), std::multiplies<>{});
10 // out == {1, 2, 6, 24, 120, 720}
```

Open in Compiler Explorer

2.9.4 `std::adjacent_difference`

`std::adjacent_difference` 是一种数值算法，它与众不同，因为它提供了严格的从左到右的变体，同时也支持并行执行，在这种情况下它表现得像一种广义的归约（我们将在下一节讨论这些）。

算法的运行方式类似于`std::transform`算法，对范围内每一对连续元素进行操作。然而，与`std::transform`不同，它保证了从左到右的执行顺序。

算法还支持输入范围，这通过内部存储最后一个右操作数的副本来实现，该副本将作为后续化简步骤的左操作数重复使用。

adjacent_difference	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	N/A

constraints		
domain	`input_range -> output_iterator`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	`operator -`	`binary_functor`

Example of the default version of `std::adjacent_difference`, which will calculate the difference of adjacent elements, with the first element copied.

```
1  std::vector<int> data{2, 3, 5, 7, 11, 13};
2  std::adjacent_difference(data.begin(), data.end(), data.begin());
3  // data == {2, 1, 2, 2, 4, 2}
```

Open in Compiler Explorer

从左到右的操作可以被用于生成式应用。

Example of more inventive use of `std::adjacent_difference` to generate the Fibonacci sequence.

```
1  std::vector<int> data(10, 1);
2  // data == {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
3
4  std::adjacent_difference(data.begin(), std::prev(data.end()),
5      std::next(data.begin()), std::plus<int>());
6  // Fibonacci sequence:
7  // data == {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

Open in Compiler Explorer

对于并行过载，输入和输出范围无法 o verlap.

Example of the parallel `std::adjacent_difference` overload.

```
1  std::vector<int> data{2, 3, 5, 7, 11, 13};
2  std::vector<int> out(data.size());
3
4  std::adjacent_difference(std::execution::par_unseq,
5      data.begin(), data.end(), out.begin());
6  // out == {2, 1, 2, 2, 4, 2}
```

Open in Compiler Explorer

2.10 一般约简

左折叠为严格的从左到右求值提供了强有力的保证，从而使折叠操作具有高度的灵活性。

然而，当我们处理既是结合的 $op(a, op(b, c)) == op(op(a, b), c)$ 又是交换的 $op(a, b) == op(b, a)$ 的运算时，实际上无论我们以何种元素排列和运算顺序进行计算，最终都会得到相同的结果。

这就是为什么在 C++17 中提供并行支持的同时，我们也获得了一批广义化的归约算法，它们以未指定的顺序和排列来归约元素。

2.10.1 `std::reduce`

reduce	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

`std::reduce` 算法是 `std::accumulate` 的广义版本。也就是说，它通过以未指定的顺序和排列将所提供的累积操作应用于各个元素，从而对一个范围进行归约。

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`std::plus<>()`	`binary_functor`

请注意，虽然我们可以使用顺序执行策略（即 `std::execution::seq`），但这并不意味着 `std::reduce` 在左折叠意义上是顺序的。

Example of using `std::reduce` with and without an execution policy.

```
1  std::vector<int> data{1, 2, 3, 4, 5};
2
3  auto sum = std::reduce(data.begin(), data.end(), 0);
4  // sum == 15
5
6  sum = std::reduce(std::execution::par_unseq,
7      data.begin(), data.end(), 0);
8  // sum == 15
9
10 auto product = std::reduce(data.begin(), data.end(), 1,
11     std::multiplies<>{});
12 // product == 120
13
14 product = std::reduce(std::execution::par_unseq,
15     data.begin(), data.end(), 1, std::multiplies<>{});
16 // product == 120
```

Open in Compiler Explorer

在 `std::accumulate` 等价版本的基础上，我们还新增了一个重载，它取消了初始累加器值，从而消除了使用错误字面量的可能性。相应地，累加器将采用范围元素的类型，并进行值初始化。

Example of using `std::reduce` without specifying the initial value of the accumulator.

```
1  struct Duck {
2      std::string sound = "Quack";
3      Duck operator+(const Duck& right) const {
4          return {sound+right.sound};
5      }
6  };
7
8  std::vector<Duck> data(2, Duck{});
9  Duck final_duck = std::reduce(data.begin(), data.end());
10 // final_duck.sound == "QuackQuackQuack"
```

Open in Compiler Explorer

累加器的初始值将是“Quack”（第2行）。加上另外两只鸭子（第8行），最终我们得到“QuackQuackQuack”。

2.10.2 `std::transform_reduce`

`std::transform_reduce` 算法是 `std::inner_product` 的泛化对应版本。在双区间变体的基础上，该算法还提供了一元重载。

transform_reduce	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

constraints		
domain	input_range	
parallel domain	forward_range	
invocable	default	custom
	N/A	(binary_functor, unary_functor)

constraints		
domain	(input_range, input_iterator)	
parallel domain	(forward_range, forward_iterator)	
invocable	default	custom
	(std::plus<>(), std::multiplies<>())	(binary_functor, binary_functor)

使用 `std::transform_reduce` 的一元版本计算平方和，以及使用二元版本计算元素与系数相乘之和的示例。

```
1  std::vector<int> data{1, 2, 3, 4, 5};
2  auto sum_of_squares = std::transform_reduce(data.begin(), data.end(),
3      0, std::plus<>{}, [](int v) { return v*v; });
```

```
// sum_of_squares == 55

std::vector<int> coef{1, -1, 1, -1, 1};
auto result = std::transform_reduce(data.begin(), data.end(),
    coef.begin(), 0);
// result == 1*1 + 2*(-1) + 3*1 + 4*(-1) + 5*1 == 3
```

Open in Compiler Explorer

2.10.3 `std::inclusive_scan`, `std::exclusive_scan`

`std::inclusive_scan` 是 `std::partial_sum` 的泛化版本。此外，我们还可以访问 `std::exclusive_scan`。

对于 `std::inclusive_scan`，第 n 个生成的元素是前 n 个源元素之和。对于 `std::exclusive_scan`，第 n 个元素是前 n-1 个源元素之和。

inclusive_scan	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

exclusive_scan	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

constraints		
domain	`input_range -> output_iterator`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	`std::plus<>()`	`binary_functor`

Example of using `std::inclusive_scan`.

```
std::vector<int> src{1, 2, 3, 4, 5, 6};
std::vector<int> out;

std::inclusive_scan(src.begin(), src.end(),
    std::back_inserter(out));
// out == {1, 3, 6, 10, 15, 21}

std::inclusive_scan(src.begin(), src.end(),
    out.begin(), std::multiplies<>{}, 1);
// out == {1, 2, 6, 24, 120, 720}
```

Open in Compiler Explorer

因此，由于 `std::exclusive_scan` 生成的第一个元素是零个元素之和，我们必须指定累加器的初始值，该初始值将成为第一个生成的元素的值。

Example of using std::exclusive_scan.

```
std::vector<int> src{1, 2, 3, 4, 5, 6};
std::vector<int> out;

std::exclusive_scan(src.begin(), src.end(),
    std::back_inserter(out), 0);
// out == {0, 1, 3, 6, 10, 15}

std::exclusive_scan(src.begin(), src.end(),
    out.begin(), 1, std::multiplies<>{});
// out == {1, 1, 2, 6, 24, 120}
```

Open in Compiler Explorer

2.10.4 std::transform_inclusive_scan, std::transform_exclusive_scan

std::transform_inclusive_scan 和 std::transform_exclusive_scan 算法是 std::inclusive_scan 和 std::exclusive_scan 的变体，在归约操作之前对每个元素应用一元变换函数。

transform_inclu…	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

transform_exclu…	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

constraints		
domain	input_range -> output_iterator	
parallel domain	forward_range -> forward_iterator	
invocable	default	custom
	N/A	(binary_functor, unary_functor)

使用std::transform_inclusive_scan来累积元素的绝对值的示例。

```
std::vector<int> data{-10, 3, -2, 5, 6};

std::vector<int> out1;
std::inclusive_scan(data.begin(), data.end(),
    std::back_inserter(out1), std::plus<>{});
// out1 == {-10, -7, -9, -4, 2}

std::vector<int> out2;
std::transform_inclusive_scan(data.begin(), data.end(),
    std::back_inserter(out2), std::plus<>{},
    [](int v) { return std::abs(v); });
// out2 == {10, 13, 15, 20, 26}
```

在 Compiler Explorer 中打开

2.11 布尔简化

在简化布尔表达式时，我们可以利用布尔逻辑提供的提前终止。该标准提供了一组三种布尔简化算法。

2.11.1 `std::all_of`，`std::any_of`，`std::none_of`

all_of	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

any_of	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

none_of	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

算法要么要求元素可转换为布尔值，要么要求指定一个谓词。

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

这些算法的行为与其命名一致。`std::all_of` 算法只有在不存在评估为 false 的元素时才返回 true。`std::any_of` 算法在至少有一个元素评估为 true 时返回 true。最后，`std::none_of` 只有在不存在评估为 true 的元素时才返回 true。

algorithm \ elements	all true	all false	mixed	empty
all_of	true	false	false	true
any_of	true	false	true	false
none_of	false	true	false	true

Example demonstrating all three boolean reduction algorithms.

```
 1  std::vector<int> data{-2, 0, 2, 4, 6, 8};
 2
 3  bool all_even = std::ranges::all_of(data,
 4      [](int v) { return v % 2 == 0; });
 5  // all_even == true
 6
 7  bool one_negative = std::ranges::any_of(data,
 8      [](int v) { return std::signbit(v); });
 9  // one_negative == true
10
11  bool none_odd = std::ranges::none_of(data,
12      [](int v) { return v % 2 != 0; });
13  // none_odd == true
```

Open in Compiler Explorer

2.12 生成器

C++ 标准提供三种生成器类型：用某个值的副本填充、用调用生成器函子得到的结果填充，以及用按顺序递增的值填充。

2.12.1 `std::fill`, `std::generate`

`std::fill` 算法通过将给定的值连续地赋给每个元素来填充一个范围。
`std::generate` 算法通过连续地将所提供生成器的结果赋给每个元素来填充一个范围。

fill, generate	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	forward_range	
parallel domain	forward_range	
invocable	default	custom
	N/A	generator

提供给`std::generate`的生成器可以是一个非常规函数，因为`std::generate`保证严格的从左到右求值。

Example of using `std::fill` and `std::generate`.

```
1  std::vector<int> data(5, 0);
2  // data == {0, 0, 0, 0, 0}
3
4  std::fill(data.begin(), data.end(), 11);
5  // data == {11, 11, 11, 11, 11}
6
7  std::ranges::generate(data, []() { return 5; });
8  // data == {5, 5, 5, 5, 5}
9
10 // iota-like
11 std::ranges::generate(data, [i = 0]() mutable { return i++; });
12 // data == {0, 1, 2, 3, 4}
```

Open in Compiler Explorer

2.12.2 `std::fill_n`, `std::generate_n`

`std::fill_n` 和 `std::generate_n` 是 `std::fill` 和 `std::generate` 的变体，它们在使用起始迭代器和元素数量指定的范围上运行。此行为使算法可以与迭代器适配器（例如 `std::back_inserter`）一起使用。

fill_n, generate_n	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	output_iterator	
parallel domain	forward_iterator	
invocable	default	custom
	N/A	generator

Example of using `std::fill_n` and `std::generate_n`.

```
1  std::vector<int> data;
2  // data == {}
3
4  std::fill_n(std::back_inserter(data), 5, 11);
5  // data == {11, 11, 11, 11, 11}
6
7  std::ranges::generate_n(std::back_inserter(data), 5,
8      []() { return 7; });
9  // data == {11, 11, 11, 11, 11, 7, 7, 7, 7, 7}
```

Open in Compiler Explorer

2.12.3 `std::iota`

iota	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++23

该`std::iota`通过连续赋值应用前缀`operator++`的结果来生成元素，从初始值开始。

constraints	
domain	`forward_range`

Example of using `std::iota`.

```
1  std::vector<int> data(9, 0);
2  // data == {0, 0, 0, 0, 0, 0, 0, 0, 0}
3
4  std::iota(data.begin(), data.end(), -4);
5  // data == {-4, -3, -2, -1, 0, 1, 2, 3, 4}
```

Open in Compiler Explorer

值得注意的是，`std::iota` 算法也是在 C++20 标准新增支持中的一个例外。`std::iota` 算法并未获得范围版本。然而，我们确实可以使用一个 iota 视图。

Example of using both finite and infinite `std::views::iota`.

```
1  std::vector<int> data;
2
3  std::ranges::transform(std::views::iota(1, 10), std::views::iota(5),
4      std::back_inserter(data), std::plus<>{});
5  // data == { 6, 8, 10, 12, 14, 16, 18, 20, 22 }
```

Open in Compiler Explorer

这里我们利用有限视图构造器 `std::views::iota(1,10)` 来确定输出大小（第 3 行），这使得我们可以将无限视图 `std::views- ::iota(5)` 用作第二个参数。从功能上讲，我们甚至可以交换第二个

把它当作一个有限的视图。然而，这将引入一次额外的（且不必要的）边界检查。

2.13 复制和移动

该标准提供了范围广泛的复制算法，大致分为三类：简单复制与移动、选择性复制，以及带有重排的复制。

2.13.1 `std::copy`，`std::move`

`std::copy` 和 `std::move` 算法提供前向复制和移动。对于重叠的区间，方向很重要，因此我们不会覆盖尚未复制的元素。

对于前向方向，输出迭代器不允许位于输入范围的`[first, last)`()之内。因此，只有输出范围的尾部可以与输入范围重叠。

constraints	
domain	`input_range -> output_iterator`
parallel domain	`forward_range -> forward_iterator`

`copy`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`move`	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

一个关于 `std::` 的非重叠情况和允许重叠情况的示例 `copy`.

```
1  std::vector<std::string> data{ "a", "b", "c", "d", "e", "f"};
2
3  std::copy(data.begin(), data.begin()+3, data.begin()+3);
4  // data = { "a", "b", "c", "a", "b", "c" }
5
6  // Overlapping case:
7  std::copy(std::next(data.begin()), data.end(), data.begin());
8  // data = { "b", "c", "a", "b", "c", "c" }
```

在 Compiler Explorer 中打开 Ex

Move 的行为完全相同，唯一的区别是在赋值之前将每个元素转换为右值，从而把拷贝变为移动。

Example of using `std::move`.

```
1  std::vector<std::string> data{ "a", "b", "c", "d", "e", "f"};
2
3  std::move(data.begin(), data.begin()+3, data.begin()+3);
4  // data = { ?, ?, ?, "a", "b", "c" }
5  // Note: most implementations will set
6  //       a moved-out-of std::string to empty.
```

Open in Compiler Explorer

值得注意的是，std::move 是否真正执行移动取决于底层元素类型。如果底层类型是仅可拷贝的，std::move 的行为将与 std::copy 完全相同。

Example of using `std::move` with a copy-only type.

```
1  struct CopyOnly {
2      CopyOnly() = default;
3      CopyOnly(const CopyOnly&) = default;
4      CopyOnly& operator=(const CopyOnly&) {
5          std::cout << "Copy assignment.\n";
6          return *this;
7      };
8  };
9
10 std::vector<CopyOnly> test(6);
11
12 std::move(test.begin(), test.begin()+3, test.begin()+3);
13 // 3x Copy assignment
```

Open in Compiler Explorer

2.13.2 `std::copy_backward`, `std::move_backward`

copy_backward	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

move_backward	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++20

`std::copy_backward` 和 `std::move_backward` 是以相反方向进行复制的变体，从范围的末尾开始。因此，输出范围的起始部分现在可以与输入范围发生重叠。

constraints	
domain	`bidirectional_range -> bidirectional_iterator`

输出迭代器不能位于 `(first, last]` 之内，并将被视为目标范围的结束迭代器，这意味着算法会将第一个值写入 `std::prev(end)`。

Example of a non-overlapping and permitted overlapping case of `std::copy_backward`.

```
1  std::vector<std::string> data{ "a", "b", "c", "d", "e", "f"};
2  std::vector<std::string> out(9, "");
3  // out == {"", "", "", "", "", "", "", "", ""}
4
5  std::copy_backward(data.begin(), data.end(), out.end());
6  // out == {"", "", "", "a", "b", "c", "d", "e", "f"}
7
8  std::copy_backward(data.begin(), std::prev(data.end()), data.end());
9  // data == { "a", "a", "b", "c", "d", "e" }
```

Open in Compiler Explorer

2.13.3 `std::copy_n`

copy_n	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

`std::copy_n` 算法是 `std::copy` 的计数变体，它接受使用迭代器指定的输入范围和元素数量。

constraints	
domain	`input_iterator -> output_iterator`
parallel domain	`forward_iterator -> forward_iterator`

算法无法检查请求的计数是否有效并且是否超出范围，因此这个负担由调用者承担。

Example of using `std::copy_n`.

```
1 std::list<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
2 std::vector<int> out;
3
4 std::copy_n(data.begin(), 5, std::back_inserter(out));
5 // out == { 1, 2, 3, 4, 5 }
```

Open in Compiler Explorer

2.13.4 `std::copy_if`，`std::remove_copy`，`std::remove_copy_if`

copy_if	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

remove_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

remove_copy_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::copy_if`、`std::remove_copy` 和 `std::remove_copy_if` 是选择性复制算法。

constraints		
domain	`input_range -> output_range`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	N/A	`unary_predicate`

`std::remove_copy` 算法将复制与提供的值不匹配的元素。`std::copy_if` 和 `std::remove_copy_if` 算法将根据谓词复制元素，其中 `std::copy_if` 复制谓词返回 true 的元素，`std::remove_copy_if` 复制谓词返回 false 的元素。

Example demonstrating differences between `std::copy_if`, `std::remove_copy` and `std::remove_copy_if`.

```
1 std::vector<int> data{ 1, 2, 3, 4, 5, 6, 7, 8, 9};
2 std::vector<int> even, odd, no_five;
3
4 auto is_even = [](int v) { return v % 2 == 0; };
5
6 std::ranges::copy_if(data, std::back_inserter(even), is_even);
7 // even == { 2, 4, 6, 8 }
```

```
 8
 9  std::ranges::remove_copy_if(data, std::back_inserter(odd), is_even);
10  // odd == { 1, 3, 5, 7, 9 }
11
12  std::ranges::remove_copy(data, std::back_inserter(no_five), 5);
13  // no_five == { 1, 2, 3, 4, 6, 7, 8, 9 }
```

Open in Compiler Explorer

2.13.5 `std::sample`

sample	
introduced	C++17
constexpr	N/A
parallel	N/A
rangified	C++20

`std::sample` 算法是一种随机选择复制算法。该算法将从源范围随机选择 N 个元素，并利用提供的随机数生成器将其复制到目标范围。

constraints	
domain	`forward_range -> output_iterator` `input_range -> random_access_iterator`

该算法的两个域是由于采样的稳定性，保持了源范围中元素的顺序。这一特性要求输入范围至少是一个前向范围，或者目标范围需要是一个随机访问范围。

Example of using `std::sample`.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
2  std::vector<int> out;
3
4  std::sample(data.begin(), data.end(), std::back_inserter(out),
5      5, std::mt19937{std::random_device{}()});
6  // e.g. out == {2, 3, 4, 5, 9}
7  // guaranteed ascending, because source range is ascending
```

Open in Compiler Explorer

2.13.6 `std::replace_copy`, `std::replace_copy_if`

`std::replace_copy` 和 `std::replace_copy_if` 算法的运作方式类似于 `std::copy`；然而，它们会复制提供的值，而不是特定的元素。

对于 `std::replace_copy`，算法将替换匹配 `std::replace_copy_if` 值的元素，算法将替换使谓词评估为真的元素。

replace_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

replace_copy_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`input_range -> output_iterator`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::replace_copy` and `std::replace_copy_if`.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> out, odd;

std::ranges::replace_copy(data, std::back_inserter(out), 5, 10);
// out == { 1, 2, 3, 4, 10, 6, 7, 8, 9 }

auto is_even = [](int v) { return v % 2 == 0; };
std::ranges::replace_copy_if(data, std::back_inserter(odd),
                             is_even, -1);
// odd == { 1, -1, 3, -1, 5, -1, 7, -1, 9 }
```

Open in Compiler Explorer

2.13.7 `std::reverse_copy`

reverse_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::reverse_copy` 算法以逆序复制元素。

constraints	
domain	`bidirectional_range -> output_iterator`
parallel domain	`bidirectional_range -> forward_iterator`

不要与 `std::copy_backwards` 混淆，`std::copy_backwards` 会按原始顺序复制元素。`std::reverse_copy` 不允许源范围与目标范围重叠。

Example of using `std::reverse_copy`.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> out;

std::ranges::reverse_copy(data, std::back_inserter(out));
// out == { 9, 8, 7, 6, 5, 4, 3, 2, 1 }
```

Open in Compiler Explorer

2.13.8 `std::rotate_copy`

rotate_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::rotate_copy` 算法将复制元素 `[middle, last)`，随后是 `[first, middle)`，这与 `std::rotate` 算法的行为一致。

constraints	
domain	`(forward_range, forward_iterator) -> output_iterator`
parallel domain	`(forward_range, forward_iterator) -> forward_iterator`

输入和输出范围不能重叠。

Example of using `std::rotate_copy`.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
2  std::vector<int> out;
3
4  std::ranges::rotate_copy(data, data.begin() + 4,
5                           std::back_inserter(out));
6  // out == { 5, 6, 7, 8, 9, 1, 2, 3, 4 }
```

Open in Compiler Explorer

2.14 未初始化内存算法

未初始化内存算法是一组相对低层的算法，旨在在实现手动内存管理时提供帮助。这些算法提供了在原始内存之上以事务方式构造、复制、移动并销毁元素序列的功能。

本节未列出这些算法的计数型变体[1]。不过请注意，本节中所有对区间进行操作的算法都具有计数型变体，其中区间通过迭代器和元素数量来指定。这些变体在一些示例中被用来进行演示。

2.14.1 `std::construct_at`, `std::destroy_at`

construct_at	
introduced	C++20
constexpr	C++20
parallel	N/A
rangified	C++20

destroy_at	
introduced	C++17
constexpr	C++20
parallel	N/A
rangified	C++20

`std::construct_at` 和 `std::destroy_at` 算法将在给定地址构造/销毁单个元素。如果指定了额外的参数，`std::construct_at` 会将这些参数转发给对象的构造函数。

Example of using `std::create_at` to create a `std::string` object using the arguments eight and 'X', which results in a string filled with eight copies of the X character.

```
1  alignas(alignof(std::string)) char mem[sizeof(std::string)];
2  auto *ptr = reinterpret_cast<std::string*>(mem);
3
4  std::construct_at(ptr, 8, 'X');
5  // *ptr == "XXXXXXXX", ptr->length() == 8
6  std::destroy_at(ptr);
```

Open in Compiler Explorer

[1]The names of these algorithms are particularly long and obnoxious.

2.14.2 `std::uninitialized_default_construct`, `std::uninitialized_value_construct`, `std::uninitialized_fill`, `std::destroy`

三个未初始化算法涵盖了元素的默认初始化、值初始化和拷贝初始化。`std::destroy` 算法提供了在不释放底层内存的情况下对元素进行销毁。

constraints	
domain	`forward_range` `forward_iterator (counted)`

un…default_construct	
introduced	C++17
constexpr	N/A
parallel	C++17
rangified	C++20

un…value_construct	
introduced	C++17
constexpr	N/A
parallel	C++17
rangified	C++20

uninitialized_fill	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

destroy	
introduced	C++17
constexpr	N/A
parallel	C++17
rangified	C++20

Example demonstrating the use of the counted variants of the uninitialized construction algorithm and the `std::destroy_n` algorithms. Note that for `std::string`, there is no difference between default and value construction.

```
1  alignas(alignof(std::string)) char buffer[sizeof(std::string)*10];
2  auto *begin = reinterpret_cast<std::string*>(buffer);
3  auto *it = begin;
4
5  it = std::uninitialized_default_construct_n(it, 3);
6  it = std::uninitialized_fill_n(it, 2, "Hello World!");
7  it = std::uninitialized_value_construct_n(it, 3);
8  it = std::uninitialized_fill_n(it, 2, "Bye World!");
9
10 // {"", "", "", "Hello World!", "Hello World!", "", "", "",
11 //  "Bye World!", "Bye World!"}
12
13 std::destroy_n(begin, 10);
```

Open in Compiler Explorer

2.14.3 `std::uninitialized_copy`, `std::uninitalized_move`

`std::unitialized_copy` 和 `std::unitialized_move` 算法遵循 `std::copy` 和 `std::move` 算法的行为，不同之处在于目标范围是未初始化的内存。

constraints	
domain	`input_range -> forward_iterator` `input_iterator -> forward_iterator (counted)`
parallel domain	`forward_range -> forward_iterator` `forward_iterator -> forward_iterator (counted)`

uninitialized_copy	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

uninitialized_move	
introduced	C++17
constexpr	N/A
parallel	C++17
rangified	C++20

Example of using `std::uninitialized_copy` and `std::uninitialized_move`.

```
alignas(alignof(std::string)) char buff1[sizeof(std::string)*5];
alignas(alignof(std::string)) char buff2[sizeof(std::string)*5];
std::vector<std::string> data = {
    "hello", "world", "and", "everyone", "else"};

auto *bg1 = reinterpret_cast<std::string*>(buff1);
std::uninitialized_copy(data.begin(), data.end(), bg1);
// buff1 == { "hello", "world", "and", "everyone", "else"}
// data == { "hello", "world", "and", "everyone", "else"}
std::destroy_n(bg1, 5);

auto *bg2 = reinterpret_cast<std::string*>(buff2);
std::uninitialized_move(data.begin(), data.end(), bg2);
// buff2 == { "hello", "world", "and", "everyone", "else"}
// data == { ?, ?, ?, ?, ?}
// In most implementations a moved-out-of string will be empty.
std::destroy_n(bg2, 5);
```

Open in Compiler Explorer

交易型行为

使用未初始化内存算法的主要好处在于它们能够正确处理事务性行为。当对象的构造函数可能抛出异常时，事务性尤为重要。如果其中一个对象构造失败，这些算法会通过析构已构造的对象来正确回滚。

Example demonstrating the roll-back behaviour of uninitialized algorithms when the third invocation of the constructor throws. Note that the exception is re-thrown after the partial work is rolled back.

```
struct Custom {
    static int cnt;
    Custom() {
        if (++cnt >= 3)
            throw std::runtime_error("Deliberate failure.");
        std::cout << "Custom()\n";
    }
    ~Custom() {
        std::cout << "~Custom()\n";
    }
};

int Custom::cnt = 0;

int main() {
    alignas(alignof(Custom)) char buffer[sizeof(Custom)*10];
```

```
17      auto *begin = reinterpret_cast<Custom*>(buffer);
18
19      try {
20          std::uninitialized_default_construct_n(begin, 10);
21          std::destroy_n(begin, 10); // not reached
22      } catch (std::exception& e) {
23          std::cout << e.what() << "\n";
24      }
25  }
26  /* OUTPUT:
27      Custom()
28      Custom()
29      ~Custom()
30      ~Custom()
31      Deliberate failure.
32  */
```

Open in Compiler Explorer

2.15 堆数据结构

该标准通过 `std::priority_queue` 为最大堆数据结构提供了一个便捷的封装。然而，在使用 `std::priority_queue` 时，我们会失去对底层数据的访问，这可能不太方便。

2.15.1 `std::make_heap`，`std::push_heap`，`std::pop_heap`

堆数据结构是一个二叉树，其中每个元素满足堆属性：父节点的值大于或等于其子节点的值。

当讨论堆算法时，我们将指的是堆的数组表示，其中索引为 i 的元素的子元素位于索引 $2i+1$ 和 $2i+2$。

堆的一个宝贵特性是，它可以在线性时间内构建，然后在提取最大值和插入新值时提供对数复杂度。

make_heap	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

push_heap, pop_heap	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

constraints		
domain	random_access_range	
invocable	default	custom
	operator<	strict_weak_ordering

`std::make_heap` 算法重新排序给定范围内的元素，使得元素保持最大堆属性，即索引 i 处的元素大于或等于索引 $2i+1$ 和 $2i+2$ 处的元素。

Example of using `std::make_heap` to construct a max-heap and a min-heap (using a custom comparator).

```
std::vector<int> data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

std::make_heap(data.begin(), data.end());
// data == {9, 8, 7, 4, 5, 6, 3, 2, 1} - different ordering possible
// 9 >= 8, 9 >= 7
// 8 >= 4, 8 >= 5
// 7 >= 6, 7 >= 3
// ...

std::make_heap(data.begin(), data.end(), std::greater<>{});
// data == {1, 2, 3, 4, 5, 6, 7, 8, 9} - different ordering possible
// 1 <= 2, 1 <= 3
// 2 <= 4, 2 <= 5
// 3 <= 6, 3 <= 7
// ...
```

Open in Compiler Explorer

`std::push_heap` 和 `std::pop_heap` 算法模拟最大堆数据结构的推入和弹出操作。然而，由于它们在范围之上操作，无法直接操作底层数据结构。因此，它们使用范围的最后一个元素作为输入/输出。

`std::push_heap` 算法将范围的最后一个元素插入堆中，而 `std::pop_heap` 将最大元素提取到范围的最后位置。如前所述，两个操作的复杂度均为对数级。

Example of using `std::push_heap` and `std::pop_heap`.

```
std::vector<int> data = { 1, 1, 2, 2 };
std::make_heap(data.begin(), data.end());
// data == { 2, 2, 1, 1} - different ordering possible

// Push 9 to the heap
data.push_back(9);
// data == { [heap_part], 9 }
std::push_heap(data.begin(), data.end());
// data == { 9, 2, 1, 1, 2 } - different ordering possible

// Push 7 to the heap
data.push_back(7);
// data == { [heap_part], 7 }
std::push_heap(data.begin(), data.end());
// data == { 9, 2, 7, 1, 2, 1 } - different ordering possible

std::pop_heap(data.begin(), data.end());
```

```
// data == { [heap_part], 9 }
std::pop_heap(data.begin(), std::prev(data.end()));
// data == { [heap_part], 7, 9 }
```

在 Compiler Explorer 中打开

2.15.2 `std::sort_heap`

`std::sort_heap` 将堆中的元素重新排序为有序顺序。注意，这与反复调用 `std::pop_heap` 相同；因此该算法具有 $O(n \log n)$ 复杂度。

sort_heap	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

constraints		
domain	random_access_range	
invocable	default	custom
	operator<	strict_weak_ordering

Example of using `std::sort_heap`.

```
std::vector<int> data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

std::make_heap(data.begin(), data.end());
// data == {9, 8, 7, 4, 5, 6, 3, 2, 1} - different ordering possible

std::sort_heap(data.begin(), data.end());
// data == {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Open in Compiler Explorer

2.15.3 `std::is_heap`, `std::is_heap_until`

`std::is_he ap` 和 `std::is_heap_until` 算法检查堆 翻译种。

is_heap	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++20

is_heap_until	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++20

constraints		
domain	random_access_range	
invocable	default	custom
	operator<	strict_weak_ordering

这两个算法遵循与 `std::is_sorted` 和 `std::is- _sorted_until` 相同的逻辑，分别返回一个布尔值和一个指向第一个乱序元素的迭代器。

Example of using `std::is_heap` and `std::is_heap_until`.

```
std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};

bool test1 = std::is_heap(data.begin(), data.end());
```

```
// test1 == false
auto it1 = std::is_heap_until(data.begin(), data.end());
// *it1 == 2

std::make_heap(data.begin(), data.end());

bool test2 = std::is_heap(data.begin(), data.end());
// test2 == true
auto it2 = std::is_heap_until(data.begin(), data.end());
// it2 == data.end()
```

Open in Compiler Explorer

2.15.4 与 `std::priority_queue` 的比较

如本节开头所述，堆算法在功能上与 `std::priority_queue` 基本相同。为了展示差异，我们来看 topk 算法的两个版本：一种算法接受一个范围，并以 `std::vector` 的形式按已排序的顺序返回前 k 个元素。

Example of implementing a topk algorithm using a `std::priority_queue`.

```
auto topk_queue(
    std::input_iterator auto begin,
    std::sentinel_for<decltype(begin)> auto end,
    size_t k) {

    using vtype = std::iter_value_t<decltype(begin)>;
    using arrtype = std::vector<vtype>;

    std::priority_queue<vtype, arrtype, std::greater<vtype>> pq;

    while (begin != end) {
        pq.push(*begin);
        if (pq.size() > k)
            pq.pop();
        ++begin;
    }

    arrtype result(k);
    for (auto &el: result | std::views::reverse) {
        el = std::move(pq.top());
        pq.pop();
    }

    return result;
}

```

Open in Compiler Explorer

该示例依赖于 C++20 概念来提供通用接口，因此应适用于任何输入范围，并返回相同元素类型的 `std::vector`。

使用优先队列时，我们可以利用提供的简单 `push()` 和 `pop()` 接口（第12行和第14行）。然而，只有通过反复应用 `pop()`，直到队列为空（第20行），才能提取队列中的所有数据。

Example of implementing a topk algorithm using heap algorithms.

```
 1  auto topk_heap(
 2      std::input_iterator auto begin,
 3      std::sentinel_for<decltype(begin)> auto end,
 4      size_t k) {
 5
 6      std::vector<std::iter_value_t<decltype(begin)>> result;
 7
 8      while (begin != end) {
 9          result.push_back(*begin);
10          std::ranges::push_heap(result, std::greater<>{});
11
12          if (result.size() > k) {
13              std::ranges::pop_heap(result, std::greater<>{});
14              result.pop_back();
15          }
16
17          ++begin;
18      }
19
20      std::ranges::sort_heap(result, std::greater<>{});
21      return result;
22  }
```

Open in Compiler Explorer

当使用堆算法时，我们需要手动管理底层数据结构（第9-10行和第13-14行）。然而，我们不需要提取数据，并且如果不需要按排序顺序获取前k个元素，我们可以省略最后的`std::sort_heap` (第20行)。

2.16 搜索与比较算法

搜索和比较类别提供了直观的线性算法（与单一值比较时）和二次算法（与范围比较时）。

2.16.1 `std::find`, `std::find_if`, `std::find_if_not`

find, find_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

find_if_not	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

std::find 算法提供了一种基本的线性搜索。标准提供了三种变体，其中一种按值搜索，另外两种使用谓词。

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==` (`find`)	`unary_predicate`

Example of utilizing `std::find` to find delimiters in a string.

```
1  std::string data = "John;Doe;April;1;1900;";
2  auto it = data.begin(), token = data.begin();
3  std::vector<std::string> out;
4
5  while ((token = find(it, data.end(), ';')) != data.end()) {
6      out.push_back("");
7      std::copy(it, token, std::back_inserter(out.back()));
8      it = std::next(token);
9  }
10 // out == { "John", "Doe", "April", "1", "1900" }
```

Open in Compiler Explorer

如果我们想搜索元素的类别，可以使用 `std::find_if` 和 `std::find_if_not`，因为这两种变体是使用谓词进行搜索的。

Example of utilizing `std::find_if_not` to find leading and trailing whitespace.

```
1  std::string data = "   hello world!  ";
2
3  auto begin = std::find_if_not(data.begin(), data.end(),
4      [](char c) { return isspace(c); });
5  if (begin == data.end()) // only spaces
6      return 0;
7
8  std::string out;
9  std::copy(begin,
10     std::find_if_not(data.rbegin(), data.rend(),
11         [](char c) { return isspace(c); }
12     ).base(),
13     std::back_inserter(out));
14 // out == "hello world!"
```

Open in Compiler Explorer

2.16.2 `std::adjacent_find`

adjacent_find	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::adjacent_find` 是一种二分查找算法，用于在单个范围内搜索成对的相邻元素。

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==`	`binary_predicate`

如果算法找到一对元素，它将返回指向这两个元素中第一个的迭代器（否则返回 end 迭代器）。

Example of using `std::adjacent_find` to find the first pair of equal elements and the first pair of elements that sum up to more than ten.

```
1  std::vector<int> data = { 1, 2, 3, 4, 4, 5, 6, 7, 7, 8, 9 };
2  auto it1 = std::adjacent_find(data.begin(), data.end());
3  // *it1 == 4, i.e. {4, 4}
4
5  auto it2 = std::adjacent_find(data.begin(), data.end(),
6      [](int l, int r) { return l + r > 10; });
7  // *it2 == 5, i.e. {5, 6}
```

Open in Compiler Explorer

2.16.3 `std::search_n`

search_n	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

该 `std::search_n` 算法用于查找给定值的 n 个实例。

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==`	`binary_predicate`

与 `std::search_n` 的接口可能有点令人困惑。该算法将实例数量和要搜索的值作为两个连续的参数，然后是一个可选的自定义比较器函数。

Example of using `std::search_n` to find two consecutive elements equal to 3, three elements equal to 3 (in modulo 5 arithmetic) and finally, two elements equal to 0.

```
1  std::vector<int> data = { 1, 0, 5, 8, 3, 3, 2 };
2
3  auto it1 = std::search_n(data.begin(), data.end(), 2, 3);
4  // *it1 == 3, i.e. {3, 3}
5
```

```
auto it2 = std::search_n(data.begin(), data.end(), 3, 3,
    [](int l, int r) { return l % 5 == r % 5; });
// *it2 == 8, i.e. {8, 3, 3}

auto it3 = std::search_n(data.begin(), data.end(), 2, 0);
// it3 == data.end(), i.e. not found
```

Open in Compiler Explorer

请注意，`std::search_n` 是 `_n` 命名方案的一个例外。

2.16.4 `std::find_first_of`

find_first_of	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

使用 `std::find_if`，我们可以轻松地搜索某一类元素。然而，有时将我们要查找的元素逐一完整地列出来会更加方便。

constraints		
domain	`(input_range, forward_range)`	
parallel domain	`(forward_range, forward_range)`	
invocable	default	custom
	`operator==`	`binary_predicate`

请注意，我们正在从线性搜索转变为 $O(m * n)$ 的时间复杂度，因为对于第一个范围中的每个元素，都需要将其与第二个范围中的所有元素进行比较（最坏情况）。

Example of using `std::find_first_of`.

```
std::vector<int> haystack = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::vector<int> needles = { 7, 5, 3 };

auto it = std::find_first_of(haystack.begin(), haystack.end(),
    needles.begin(), needles.end());
// *it == 3, i.e. haystack[2]
```

Open in Compiler Explorer

2.16.5 `std::search`，`std::find_end`

search, find_end	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::search` 和 `std::find_end` 算法都在一个序列中搜索子序列。`std::search` 算法将返回第一个实例，而 `std::find- _end` 将返回最后一个。

constraints		
domain	`(forward_range, forward_range)`	
parallel domain	`(forward_range, forward_range)`	
invocable	default	custom
	`operator==`	`binary_predicate`

Example of using `std::search` and `std::find_end`.

```
std::string haystack = "abbabba";
std::string needle = "bba";

auto it1 = std::search(haystack.begin(), haystack.end(),
    needle.begin(), needle.end());
// it1..end == "bbabba"

auto it2 = std::find_end(haystack.begin(), haystack.end(),
    needle.begin(), needle.end());
// it2..end == "bba"
```

Open in Compiler Explorer

搜索者

自 C++17 起，我们还可以为搜索算法指定自定义搜索器。除了基本的搜索器之外，标准还实现了 Boyer-Moore 和 Boyer-Moore-Horspool 字符串搜索器，它们在最佳、最坏以及平均情况下提供不同的复杂度。

Example of using `std::search` with custom searchers.

```
std::string haystack = "abbabba";
std::string needle = "bba";

auto it1 = std::search(haystack.begin(), haystack.end(),
    std::default_searcher(needle.begin(), needle.end()));

auto it2 = std::search(haystack.begin(), haystack.end(),
    std::boyer_moore_searcher(needle.begin(), needle.end()));

auto it3 = std::search(haystack.begin(), haystack.end(),
    std::boyer_moore_horspool_searcher(needle.begin(), needle.end()));
// it1 == it2 == it3
```

Open in Compiler Explorer

2.16.6 `std::count`, `std::count_if`

`std::count` 和 `std::count_if` 算法用于统计匹配元素的数量。

count, count_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==`	`unary_predicate`

要查找的元素可以使用一个值（`std::count`）或一个谓词（`std::count_if`）来指定。

Example of using `std::count` and `std::count_if`.

```
1  std::vector<int> data = { 1, 2, 3, 2, 1, 2, 3, 2, 1 };
2
3  auto one_cnt = std::count(data.begin(), data.end(), 1);
4  // one_cnt == 3
5
6  auto even_cnt = std::count_if(data.begin(), data.end(),
7      [](int v) { return v % 2 == 0; });
8  // even_cnt == 4
```

Open in Compiler Explorer

equal	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

2.16.7 `std::equal`, `std::mismatch`

该 `std::equal` 算法为范围提供相等比较。

constraints		
domain	(input_range, input_iterator)	
parallel domain	(forward_range, forward_iterator)	
invocable	default	custom
	operator==	binary_predicate

Example of using `std::equal`.

```
1  std::vector<int> first = { 1, 2, 3, 4, 5 };
2  std::vector<int> second = { -1, -2, -3, -4, -5 };
3
4  bool test1 = std::equal(first.begin(), first.end(), second.begin());
5  // test1 == false
6
7  bool test2 = std::equal(first.begin(), first.end(), second.begin(),
8      [](int l, int r) { return std::abs(l) == std::abs(r); });
9  // test2 == true
```

Open in Compiler Explorer

mismatch	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`std::mismatch` 算法的行为与 `std::equal` 完全一致；然而，它并非返回一个简单的布尔值，而是返回一对迭代器，用以表示不匹配的元素。

constraints		
domain	(input_range, input_iterator)	
parallel domain	(forward_range, forward_iterator)	
invocable	default	custom
	operator==	binary_predicate

Example of using std::mismatch.

```
1  std::vector<int> first = { 1, 2, 3, 4, 5 };
2  std::vector<int> second = { 1, 2, 2, 4, 5 };
3
4  auto it_pair = std::mismatch(first.begin(), first.end(),
5                               second.begin());
6  // *it_pair.first == 3, *it_pair.second == 2
```

Open in Compiler Explorer

在基本变体之上，`std::equal` 和 `std::mismatch` 都提供了一种两个范围都被完全指定的版本（自 C++14 起）。这些版本可以检测超出第一个范围作用域的不匹配。

constraints		
domain	(input_range, input_range) since C++14	
parallel domain	(forward_range, forward_range)	
invocable	default	custom
	operator==	binary_predicate

Example of detecting a mismatch beyond the scope of the first range.

```
1  std::vector<int> first = { 1, 2, 3, 4, 5 };
2  std::vector<int> second = { 1, 2, 3, 4, 5, 6 };
3
4  bool test1 = std::equal(first.begin(), first.end(), second.begin());
5  // test1 == true, cannot detect mismatch in number of elements
6
7  bool test2 = std::equal(first.begin(), first.end(),
8      second.begin(), second.end());
9  // test2 == false, different number of elements -> not equal.
10
11 auto pair_it = std::mismatch(first.begin(), first.end(),
12     second.begin(), second.end());
13 // pair_it.first == first.end()
14 // *pair_it.second == 6
```

Open in Compiler Explorer

2.17 极小极大算法

该组包含处理最小值和最大值的算法。然而，这里有两个相关的理论主题，我们需要先讨论它们：`std::initializer_list` 和 `const_cast`。

对于接受 `std::initializer_list` 的函数，值得记住的是 `std::initializer_list` 是通过复制构造的；它的内部数组

元素是从列出的元素复制构造的。因此，我们在编译时上下文之外使用 `std::initializer_list` 时需要小心。

Example demonstrating case when utilizing `std::initializer_list` leads to excessive copies.

```
struct X {
    static int copy_cnt;
    X(int v) : value(v) {}
    X(const X& other) : value(other.value) {
        ++copy_cnt;
    }
    int value;
    friend auto operator <=>(const X&, const X&) = default;
};

int X::copy_cnt = 0;

void example() {
    X a{1}, b{2}, c{3}, d{4}, e{5};
    auto max = std::max({a, b, c, d, e});
    // max.value == 5
    // X::copy_cnt == 6
}
```

Open in Compiler Explorer

在这个例子中，使用 `std::initializer_list` 导致六个副本（我们通过静态数据成员 `X::copy_cnt` 来计算）。五个副本是通过将变量 `a` 和 `e` 传入 `std::initializer_list` 生成的，一个副本是由 `std::max` 返回的结果。

在极少数情况下，我们可以强制一个可变实体保持常量性。如果常量性不需要，可以使用 `const_cast` 来去除常量性。

Example demonstrating the valid and invalid uses for `const_cast`.

```
int x = 10, y = 20;

auto& v = const_cast<int&>(std::min(x, y));
v = 5;
// x == 5, y == 20

// !IMPORTANT! the following compiles, but is undefined behaviour
// i.e. the program is ill-formed
const int z = 3;
auto& w = const_cast<int&>(std::min(x, z));
w = 10;
```

Open in Compiler Explorer

请记住，当使用类似 `const_cast` 的强制转换时，您实际上是在覆盖编译器的判断。因此，确保给定的强制转换有效完全取决于您。

2.17.1 `std::min`，`std::max`，`std::minmax`

`std::min`、`std::max` 和 `std::minmax` 的基本版本操作于两个元素，通过常量引用接收参数并通过常量引用返回结果。不幸的是，如前所述，这会产生常量性问题，我们还必须小心在传入临时对象时通过值捕获结果。

min, max	
introduced	C++98
constexpr	C++14
parallel	N/A
rangified	C++20

minmax	
introduced	C++11
constexpr	C++14
parallel	N/A
rangified	C++20

Example demonstrating use of `std::min` and `std::max`.

```
int x = 10, y = 20;
int min = std::min(x, y);
// min == 10

std::string hello = "hello", world = "world";
std::string& universe =
    const_cast<std::string&>(std::max(hello, world));
universe = "universe";

std::string greeting = hello + " " + world;
// greeting == "hello universe"

int j = 20;
auto& k = std::max(5, j);
// IMPORTANT! only works because 5 < j
// would produce dangling reference otherwise
// k == 20
```

Open in Compiler Explorer

通过值捕获结果在使用`std::minmax`时变得有些复杂，因为它返回一个`std::pair`的常量引用。为了避免对过期的右值引用悬挂，我们必须显式命名结果类型。不幸的是，在使用`auto`或结构化绑定时，无法绕过这个问题。

Example demonstrating use of `std::minmax`.

```
struct X {
    int rank;
    auto operator <=> (const X&) const = default;
};

X a{1}, b{2};
auto [first, second] = std::minmax(b, a);
// first.rank == 1, second.rank == 2
```

```
 9
10  // Operating on prvalues requires capture by value
11  std::pair<int,int> result = std::minmax(5, 10);
12  // result.first = 5, result.second = 10
```

在 Compiler Explorer 中打开

C++14 和 C++20 标准引入了按值返回的 min-max 算法的额外变体。改为按值返回在解决悬空引用问题的同时，也引入了产生过多拷贝的潜在风险。

constraints		
domain	C++14: `initializer_list` C++20 range version: `input_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of `std::initializer_list` and range variants of `std::min`, `std::max` and `std::minmax`.

```
1  auto min = std::min({5, 3, -2, 0});
2  // min == -2
3
4  auto minmax = std::minmax({5, 3, -2, 0});
5  // minmax.first == -2, minmax.second == 5
6
7  std::list<int> data{5, 3, -2, 0};
8  auto max = std::ranges::max(data);
9  // max == 5
```

Open in Compiler Explorer

2.17.2 `std::clamp`

clamp	
introduced	C++17
constexpr	C++17
parallel	N/A
rangified	C++20

`std::clamp` 算法接受三个参数：数值、最小值和最大边界，并会将该数值限制在所提供的最小值和最大值之间：

- 如果 $value < minimum$、`std::clamp` 返回最小值
- 如果 $maximum < value$，`std::clamp` 返回最大值
- 否则，`std::clamp` 返回该值

因为该算法通过 const 引用接受其参数并返回一个 const 引用，它在 const 正确性和悬空引用方面与 min-max 算法存在相同的问题。

Examples of using `std::clamp` with prvalues and with lvalues and `const_cast` to get mutable access to the original variable.

```
int a = std::ranges::clamp(10, 0, 20);
// a == 10 (0 < 10 && 10 < 20)

int b = std::clamp(-20, 0, 20);
// b == 0 (-20 < 0)

int c = std::clamp(30, 0, 20);
// c == 20 ( 30 > 20 )

int x = 10, y = 20, z = 30;
int &w = const_cast<int&>(std::clamp(z, x, y));
w = 99;
// x == 10, y == 99, z == 30
```

Open in Compiler Explorer

2.17.3 `std::min_element`, `std::max_element`, `std::minmax_element`

元素版本的最小-最大算法作用于范围，并且不通过常量引用或值返回，而是返回指向最小或最大元素的迭代器。

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

`min_element`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`max_element`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`minmax_element`	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

使用极小极大算法元素版本的示例。

```
std::vector<int> data = { 5, 3, -2 , 0};
auto i = std::min_element(data.begin(), data.end());
// *i == -2 (i.e. data[2])
auto j = std::max_element(data.begin(), data.end());
// *j == 5 (i.e. data[0])

auto k = std::minmax_element(data.begin(), data.end());
// *k.first == -2, *k.second == 5
```

在 Compiler Explorer 中打开

你可能会想知道我们是否可以在最小-最大算法的元素版本中引发相同的悬挂引用（迭代器）问题。幸运的是，C++20 范围的一个非常好的特点就是能够防止这个问题。

Example demonstrating protection from dangling iterators.

```
1  auto i = std::ranges::min_element(std::vector<int>{5, 3, -2, 0});
2  // decltype(i) == std::ranges::dangling
3
4  std::vector<int> data = { 5, 3, -2 , 0};
5  auto j = std::ranges::min_element(std::span(data.begin(), 2));
6  // *j == 3, std::span is a borrowed_range
```

Open in Compiler Explorer

所有返回迭代器的算法的所有范围版本，在对临时范围调用时，将返回 `std::ranges- ::dangling` 类型。这将排除使用 `std::span` 来对子引用范围的用例，这也是范围算法具有额外的 `borrowed_range` 概念的原因。由于这些范围不拥有其元素，因此可以作为临时对象传递。

第3章

区间简介

C++20 标准引入了 Ranges 库（即 STLv2），它有效地替代了现有的算法和功能。在本章中，我们将回顾主要的变化。

请注意，Ranges 是在 C++20 中以部分实现状态引入的功能之一。尽管 C++23 引入了大量额外特性，我们已经知道仍有许多特性不会进入 C++23。

3.1 对概念的依赖

该标准一直描述了传递给算法的每个参数所需满足的要求。然而，这种描述纯属说明性的，语言本身并未提供一等工具来强制执行这些要求。因此，错误消息往往令人困惑。

Concepts 是 C++20 中引入的一项新的语言特性，它允许库的实现者对泛型代码的参数加以约束。Concepts 超出了本书的讨论范围；不过，有两个值得注意的影响。

首先，所有概念的定义现在都已成为标准库的一部分，例如，你可以查阅某个类型成为 `random_access_range` 究竟意味着什么，并且还可以使用这些概念来约束你的代码。

Example of using standard concepts in user code. The function accepts any random access range as the first argument and an output iterator with the same underlying type as the second argument.

```
template <std::ranges::random_access_range T>
auto my_function(T&& rng,
    std::output_iterator<std::ranges::range_value_t<T>> auto it) {
    if (rng.size() >= 5)
        *it++ = rng[4];
    if (rng.size() >= 7)
        *it++ = rng[6];
}
```

Open in Compiler Explorer

其次，错误消息现在会引用未满足的约束，而不是报告发生在库实现深处的错误。

```
note: candidate: 'template<class _Iter, class _Sent, class _Comp, class _Proj>
    requires (random_access_iterator<_Iter>)
```

3.2 区间的概念

非范围算法一直以来都严格基于迭代器进行操作。从概念上讲，元素的范围由两个迭代器`[first, last)`来定义，或者在处理标准数据结构中的所有元素时`[begin, end)`。

范围算法在形式上将一个范围定义为由迭代器和哨兵组成的一对。将哨兵改为不同的类型，可以在结束迭代器无法自然映射到元素的情况下释放简化代码的潜力。标准提供了两种默认的哨兵类型，`std::default_sentinel` 和`std::unreachable_sentinel`。

除了简化某些使用场景外，哨兵值还支持无限范围，并可能带来性能提升。

Example of an infinite range when the data guarantees termination. Using `std::unreachable_sentinel` causes the boundary check to be optimized-out, removing one comparison from the loop.

```
std::vector<int> dt = { 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::ranges::shuffle(dt, std::mt19937(std::random_device()()));

auto pos = std::ranges::find(
    dt.begin(),
    std::unreachable_sentinel,
    7);
```

Open in Compiler Explorer

只有当我们拥有一个上下文上的保证，即该

算法将在不越界的情况下终止。然而，这消除了算法性能不如手写代码的少数情况之一。

最后，随着 ranges 的引入，算法现在提供了范围重载，使代码更加简洁且更易于阅读。

Example of using the range overload of `std::sort`.

```
1  std::vector<int> dt = {1, 4, 2, 3};
2  std::ranges::sort(dt);
```

Open in Compiler Explorer

3.3 投影

每个范围算法都带有一个额外的参数，即投影。投影本质上是在元素被传递给主函子之前应用的一种内置变换操作。

该投影可以是任何可调用对象，包括成员指针。

Example of using a projection to sort elements of a range based on a computed value from an element method.

```
1  struct Account{
2      double value();
3  };
4
5  std::vector<Account> data = get_data();
6
7  std::ranges::sort(data, std::greater<>{}, &Account::value);
```

Open in Compiler Explorer

因为投影结果仅用于主函子，它不会修改算法的输出，除了 `std::ranges::transform`。

`std::ranges::copy_if` 使用谓词的投影结果，而 `std::ranges::transform` 使用投影结果作为变换函数的输入（因此使其影响输出类型）。

```
1  struct A{};
2  struct B{};
3
4  std::vector<A> data(5);
5
6  std::vector<A> out1;
7  // std::vector<B> out1; would not compile
8  std::ranges::copy_if(data, std::back_inserter(out1),
```

```
 9      [](B) { return true; }, // predicate accepts B
10      [](A) { return B{}; }); // projection projects A->B
11
12  std::vector<B> out2;
13  std::ranges::transform(data, std::back_inserter(out2),
14      [](auto x) { return x; },  // no-op transformation functor
15      [](A) { return B{}; }); // projection projects A->B
```

Open in Compiler Explorer

3.4 悬空迭代器保护

由于算法的范围版本提供了接受整个范围的重载，当用户传入临时范围时，它们就会带来悬空迭代器的潜在风险。

然而，对于某些范围，传入临时对象并不成问题。为区分这两种情况并防止悬空迭代器，范围库引入了借用范围的概念以及一种特殊类型 `std::ranges::dangling`。

借用范围是从另一个范围“借用”其元素的范围。主要示例是 `std::string_view` 和 `std::span`。对于借用范围，元素的生命周期并不与范围自身的生命周期绑定。

Example demonstrating the different handling for a borrowed range `std::string_view` and a `std::string`.

```
 1  const char* c_str = "1234567890";
 2
 3  // find on a temporary string_view
 4  auto sep1 = std::ranges::find(std::string_view(c_str), '0');
 5  // OK, string_view is a borrowed range, *sep1 == '0',
 6
 7  int bad = 1234567890;
 8
 9  // find on a temporary string
10  auto sep2 = std::ranges::find(std::to_string(bad), '0');
11  // decltype(sep2) == std::ranges::dangling, *sep2 would not compile
```

Open in Compiler Explorer

用户类型可以通过特化 `enable- _borrowed_range` 常量将自身声明为借用范围。

Example demonstrating declaring MyView as a borrowed range.

```
1  template<typename T> inline constexpr bool
2      std::ranges::enable_borrowed_range<MyView<T>> = true;
```

Open in Compiler Explorer

3.5 视图

算法的一个核心问题是它们不易组合。因此，使用算法的代码往往相当冗长，并且在处理不可变数据时需要额外的拷贝。

视图旨在通过提供在编译期组合的、复制和移动开销低的设施来解决这一问题。

Example of composing several views.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
for (auto v : data | std::views::reverse |
         std::views::take(3) | std::views::reverse) {
    // iterate over 7, 8, 9 (in order)
}
```

Open in Compiler Explorer

值得注意的是，视图是有状态的，应将其视为有状态的 lambda 表达式（例如，在迭代器有效性和线程安全方面）。

Example of `std::views::drop` caching behaviour.

```
std::list<int> data{1, 2, 3, 4, 5, 6, 7, 8};
auto view = data | std::views::drop(3);

for (auto v : view) {
    // iterate over: 4, 5, 6, 7, 8
}

// Note, if we used std::vector, push could invalidate
// the cached iterator inside of std::views::drop.
data.push_front(99);
for (auto v : view) {
    // iterate over: 4, 5, 6, 7, 8
}

// Fresh view
for (auto v : data | std::views::drop(3)) {
    // iterate over: 3, 4, 5, 6, 7, 8
}

const auto view2 = data | std::views::drop(3);
// for (auto v : view2) {}
// Wouldn't compile, std::views::drop requires mutability.
```

在 Compiler Explorer 中打开

第四章

这些观点

在本章中，我们将介绍标准库提供的所有视图。

4.1 `std::views::keys`, `std::views::values`

`std::views::keys` 和 `std::views::values` 对成对样式的元素范围进行操作。
`std::views::keys` 将生成由每个配对中的第一个元素组成的一个范围。
`std::views::values` 将生成由每个配对中的第二个元素组成的一个范围。

Example of decomposing a `std::unordered_map` into a view over the keys and a view over the values.

```
std::unordered_map<int,double> map{
    {0, 1.0}, {1, 1.5}, {2, 2.0}, {3, 2.5}
};

std::vector<int> keys;
std::ranges::copy(std::views::keys(map), std::back_inserter(keys));
// keys == {0, 1, 2, 3} in unspecified order (std::unordered_map)

std::vector<double> values;
std::ranges::copy(std::views::values(map),
        std::back_inserter(values));
// values == {1.0, 1.5, 2.0, 2.5}
// in unspecified order matching order of keys
```

Open in Compiler Explorer

4.2 `std::views::elements`

`std::views::elements` 将从一组类似元组的元素范围中生成第 N 个元素的一个范围。

Example of creating element views for each tuple's second and third elements in the source range.

```
std::vector<std::tuple<int,int,std::string>> data{
    {1, 100, "Cat"}, {2, 99, "Dog"}, {3, 17, "Car"},
};

std::vector<int> second;
std::ranges::copy(data | std::views::elements<1>,
    std::back_inserter(second));
// second == {100, 99, 17}

std::vector<std::string> third;
std::ranges::copy(data | std::views::elements<2>,
    std::back_inserter(third));
// third == {"Cat", "Dog", "Car"}
```

Open in Compiler Explorer

4.3 `std::views::transform`

变换 view 将一个变换函子应用到每个元素上 范围的。

Example of using `std::views::transform` that also changes the base type of the range.

```
std::vector<double> data{1.2, 2.3, 3.1, 4.5, 7.1, 8.2};

std::vector<int> out;
std::ranges::copy(data |
    std::views::transform([](auto v) -> int {
        return v*v;
    }), std::back_inserter(out));
// out == {1, 5, 9, 20, 50, 67}
```

Open in Compiler Explorer

4.4 `std::views::take`，`std::views::take_while`

这两种视图都由源范围的前导元素构成。对于 `std::views::take`，该视图由前 N 个元素组成。对于 `std::views::take_while`，该视图由使谓词求值为 true 的元素序列组成。

Example of a view of the first three elements and a view of the leading sequence of odd elements.

```
std::vector<int> data{1, 3, 5, 7, 2, 4, 6, 8};

std::vector<int> out1;
std::ranges::copy(data | std::views::take(3),
    std::back_inserter(out1));
// out1 == {1, 3, 5}

std::vector<int> out2;
std::ranges::copy(data |
    std::views::take_while([](int v) { return v % 2 != 0; }),
    std::back_inserter(out2));
// out2 == {1, 3, 5, 7}
```

Open in Compiler Explorer

4.5 `std::views::drop`, `std::views::drop_while`

drop 视图是 take 视图的逆操作。`std::views::drop` 包含除前 N 个元素之外的所有元素。`std::views::drop_while` 包含除谓词求值为 true 的起始元素序列之外的所有元素。

Example of a view of all but the first three elements and a view skipping over the leading sequence of odd elements.

```
std::vector<int> data{1, 3, 5, 7, 2, 4, 6, 8};

std::vector<int> out1;
std::ranges::copy(data | std::views::drop(3),
    std::back_inserter(out1));
// out1 == {7, 2, 4, 6, 8}

std::vector<int> out2;
std::ranges::copy(data |
    std::views::drop_while([](int v) { return v % 2 != 0; }),
    std::back_inserter(out2));
// out2 == {2, 4, 6, 8}
```

Open in Compiler Explorer

4.6 `std::views::filter`

过滤视图由所有满足所提供谓词的元素组成。

Example of a view of even elements.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8};
2
3  std::vector<int> even;
4  std::ranges::copy(data |
5      std::views::filter([](int v) { return v % 2 == 0; }),
6      std::back_inserter(even));
7  // even == {2, 4, 6, 8}
```

Open in Compiler Explorer

4.7 `std::views::reverse`

reverse 视图是双向范围的反向迭代视图。

Example of a reverse view.

```
1  std::vector<int> data{1, 2, 3, 4};
2
3  std::vector<int> out;
4  std::ranges::copy(data | std::views::reverse,
5      std::back_inserter(out));
6  // out == {4, 3, 2, 1}
```

Open in Compiler Explorer

4.8 `std::views::counted`

counted view 将一个迭代器和元素数量适配为一个视图。

Example of using a counted view to iterate over a subrange.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8};
2
3  std::vector<int> out;
4  std::ranges::copy(std::views::counted(std::next(data.begin()), 3),
5      std::back_inserter(out));
6  // out == {2, 3, 4}
```

Open in Compiler Explorer

4.9 `std::views::common`

常见视图将视图适配为一个常见范围，该范围具有匹配类型的起始和结束迭代器。非范围版本的算法需要一个常见范围。

Example of using adapting a view for a non-range algorithm.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8};

std::vector<int> out;
auto view = data |
    std::views::filter([](int v) { return v % 2 == 0; }) |
    std::views::common;

std::copy(view.begin(), view.end(), std::back_inserter(out));
// out == {2, 4, 6, 8}
```

Open in Compiler Explorer

4.10 `std::views::all`

all 视图是一个展示某个范围内所有元素的视图。

Example of creating a view over all elements. Note that view over all elements is the default.

```
std::vector<int> data{1, 2, 3, 4};

std::vector<int> out;
std::ranges::copy(std::views::all(data), std::back_inserter(out));
// out == {1, 2, 3, 4}
```

Open in Compiler Explorer

4.11 `std::views::split`，`std::views::lazy_split`，`std::views::join_view`

这两个 split 视图将单个范围拆分为基于子范围的视图。然而，它们在实现上有所不同。`std::view::split` 保留了底层范围的双向、随机访问或连续性特性，而 `std::view::- lazy_split` 则不具备这些特性，但它确实支持输入范围。

Example of using split view to parse a version number.

```
std::string version = "1.23.13.42";
std::vector<int> parsed;

std::ranges::copy(version |
    std::views::split('.') |
    std::views::transform([](auto v) {
        int result = 0;
        // from_chars requires contiguous range
```

```
 9          std::from_chars(v.data(), v.data()+v.size(), result);
10          return result;
11      }),
12      std::back_inserter(parsed));
13  // parsed == {1, 23, 13, 42}
```

Open in Compiler Explorer

连接视图扁平化了一个范围视图。

Example of using `std::views::lazy_split` to split a string into tokens and then join them using the `std::views::join`.

```
 1  std::string_view data = "Car Dog Window";
 2  std::vector<std::string> words;
 3  std::ranges::for_each(data | std::views::lazy_split(' '),
 4      [&words](auto const& view) {
 5          // string constructor needs common range.
 6          auto common = view | std::views::common;
 7          words.emplace_back(common.begin(), common.end());
 8      });
 9  // words == {"Car", "Dog", "Window"}
10
11  auto joined = data | std::views::lazy_split(' ') | std::views::join |
    ↪  std::views::common;
12  std::string out(joined.begin(), joined.end());
13  // out == "CarDogWindow"
```

Open in Compiler Explorer

4.12 `std::views::empty`，`std::views::single`

空视图是一个空视图（不包含任何元素），单视图是一个包含单个元素的视图。请注意，视图拥有该单个元素，该元素将在视图一起复制/移动。

Example of using `std::views::empty` and `std::views::single`.

```
1  std::vector<int> out;
2  std::ranges::copy(std::views::empty<int>, std::back_inserter(out));
3  // out == {}
4
5  std::ranges::copy(std::views::single(42), std::back_inserter(out));
6  // out == {42}
```

Open in Compiler Explorer

4.13 `std::views::iota`

iota 视图表示一个通过反复递增初始值而生成的序列。

Example of using the finite and infinite iota view.

```
std::vector<int> out1;
std::ranges::copy(std::views::iota(2,5), std::back_inserter(out1));
// finite view [2, 5), out == {2, 3, 4}

std::vector<int> out2;
std::ranges::copy(std::views::iota(42) | std::views::take(5),
std::back_inserter(out2));
// infinite view starting with 42, take(5) takes the first five
   ↪  elements from this view
// out2 == {42, 43, 44, 45, 46}
```

Open in Compiler Explorer

4.14 `std::views::istream`

istream 视图提供了与 istream 迭代器类似的功能，只是以视图的形式存在。它表示通过连续应用 istream 输入运算符而获得的一个视图。

Example of using istream view.

```
std::ranges::for_each(std::views::istream<int>(std::cin), [](int v) {
    // iterate over integers on standard input
});
```

Open in Compiler Explorer

第五章

C++ 理论点滴

本章将深入探讨全书中提及的各个主题。虽然本章可作为参考，但这些主题仍以高度简化、以示例为主的形式呈现。如需权威参考，请参阅 C++ 标准。

5.1 参数依赖查找（ADL）

当调用一个未加限定的方法（即未指定命名空间）时，编译器需要确定候选函数的集合。作为第一步，编译器会进行非限定名称查找，该查找从局部作用域开始，并逐级检查父作用域，直到找到该名称的第一个实例（此时即停止）。

Example of unqualified lookup. Both calls to `some_call` will resolve to `::A::B::some_call` since this is the first instance discovered by the compiler.

```
namespace A {
void some_call(int) {}
void some_call(const char*) {}
namespace B {
void some_call(double) {}

void calling_site() {
    some_call(1);   // A::B::some_call
    some_call(2.0); // A::B::some_call
    // some_call("hello world"); will not compile
    // no conversion from const char* -> double
}
}
}
```

Open in Compiler Explorer

由于未限定查找的简单性，我们需要一个额外的机制来发现重载。特别地，这对于运算符重载是一个必要条件，因为运算符调用是未限定的。这就是依赖于参数的查找（argument-dependent lookup）发挥作用的地方。

Without ADL, any call to a custom operator overload would have to be fully qualified, requiring the function call syntax.

```
namespace Custom {
struct X {};

std::ostream& operator << (std::ostream& s, const X&) {
    s << "Output of X\n";
    return s;
}
}

void calling_site() {
    Custom::X x;
    Custom::operator << (std::cout, x); // OK, explicit call
    std::cout << x; // Requires ADL
}
```

Open in Compiler Explorer

尽管 ADL 的完整规则相当复杂，但一个高度简化的说法是：编译器在确定可行的函数重载时，还会考虑所有参数的最内层命名空间。

```
namespace A {
    struct X {};
    void some_func(const X&) {}
}

namespace B {
    struct Y {};
    void some_func(const Y&) {}
}

void some_func(const auto&) {}

void calling_site() {
    A::X x; B::Y y;
    some_func(x); // Calls A::some_func
    some_func(y); // Calls B::some_func
}
```

Open in Compiler Explorer

可以说，ADL 的真正力量在于与其他语言的交互。

特性，所以让我们看看 ADL 如何与友元函数和函数对象交互。

5.1.1 友元函数与 ADL

友元函数（当以内联方式定义时）不参与普通的名称查找（它们属于外围命名空间的一部分，但不可见）。然而，它们仍然对 ADL 可见，这使得通过 ADL 提供定制点的默认实现这一常见实现模式成为可能。

Example demonstrating a default implementation with a customization point through ADL. The default implementation needs to be discovered during the unqualified lookup; therefore, if any custom implementation is visible in the surrounding namespaces, it will block this discovery.

```
namespace Base {
    template <typename T>
    std::string serialize(const T&) {
        return std::string{"{Unknown type}"};
    }
}

namespace Custom {
struct X {
    friend std::string serialize(const X&) {
        return std::string{"{X}"};
    }
};

struct Y {};
}

void calling_site() {
    Custom::X x;
    Custom::Y y;

    auto serialized_x = serialize(x);
    // serialized_x == "{X}"

    // auto serialized_y = serialize(y); // Would not compile.

    using Base::serialize;  // Pull in default version.
    auto serialized_y = serialize(y);
    // serialized_y == "{Unknown type}"
}

```

Open in Compiler Explorer

5.1.2 函数对象 vs ADL

第二个显著的交互发生在非函数符号上。在本节的上下文中，重要的符号是函数对象（和lambda）。

参数依赖查找不会考虑非函数符号。这意味着一个函数对象或一个 lambda 必须对非限定查找可见，或者需要完全限定。

Example demonstrating a lambda stored in an inline variable that can only be invoked through a qualified call.

```
namespace Custom {
    struct X {};

    constexpr inline auto some_func = [](const X&) {};
}

void calling_site() {
    Custom::X x;
    // some_func(x); // Will not compile, not visible to ADL.
    Custom::some_func(x); // OK
}
```

Open in Compiler Explorer

最终，在不合格查找阶段发现一个非函数符号将完全阻止ADL。

Example demonstrating a non-function object preventing ADL, making a friend function impossible to invoke.

```
namespace Custom {
    struct X {
        friend void some_func(const X&) {}
    };
}

constexpr inline auto some_func = [](const auto&) {};

void calling_site() {
    Custom::X x;
    some_func(x); // calls ::some_func
    // Because ADL is skipped, Custom::some_func cannot be called.
}
```

Open in Compiler Explorer

5.1.3 C++ 20 ADL 自定义点

随着C++20中概念的引入，我们现在有了一种更简洁的方式，通过ADL引入定制点。

The concept on line 4 will be satisfied if an ADL call is valid, i.e. there is a custom implementation. This is then used on lines 8 and 13 to differentiate between the two overloads. One calls the custom implementation, and the other contains the default implementation. The inline variable on line 18 is in an inline namespace to prevent collision with friend functions in the same namespace. However, because friend functions are only visible to ADL, a fully qualified call will always invoke this function object.

```
 1 namespace dflt {
 2 namespace impl {
 3     template <typename T>
 4     concept HasCustomImpl = requires(T a) { do_something(a); };
 5
 6     struct DoSomethingFn {
 7         template <typename T> void operator()(T&& arg) const
 8         requires HasCustomImpl<T> {
 9             do_something(std::forward<T>(arg));
10         }
11
12         template <typename T> void operator()(T&&) const
13         requires (!HasCustomImpl<T>) { /* default implementation */ }
14     };
15 }
16
17 inline namespace var {
18     constexpr inline auto do_something = impl::DoSomethingFn{};
19 }
20 }
21
22 namespace custom {
23     struct X { friend void do_something(const X&){}; };
24     struct Y {};
25 }
26
27 void calling_site() {
28     custom::X x;
29     custom::Y y;
30     dflt::do_something(x); // calls custom::do_something(const X&)
31     dflt::do_something(y); // calls default implementation
32 }
```

Open in Compiler Explorer

这种方法有几个优点，特别是在调用位置。我们不再需要记住引入默认实现的命名空间。此外，由于调用现在是完全限定的，我们避免了可能完全阻止 ADL 的符号冲突问题。

5.2 整型和浮点类型

可以说，C++ 中最容易出错的部分之一是整数和浮点表达式。由于这部分语言继承自 C，它在很大程度上依赖相当复杂的隐式转换规则，并且有时会与 C++ 语言中更为静态的部分产生不直观的交互。

5.2.1 整型类型

在处理整型类型时，潜在的类型变化分为两个阶段。首先，对等级低于 int 的类型进行提升；如果所得表达式仍然包含不同的整型类型，则会应用一次转换以得到一个共同的类型。

整数类型的等级在标准中定义：

1. `bool`
2. `char`, `signed char`, `unsigned char`
3. `short int`, `unsigned short int`
4. `int`, `unsigned int`
5. `long int`，`unsigned long int`
6. `long long int`，`unsigned long long int`

促销活动

如前所述，整型提升适用于等级低于 `int` (的类型，例如 `bool`、`char`、`short`)。此类操作数将被提升为 int（如果 `int` 能表示该类型的所有值，则提升为 int；否则为 `unsigned int`）。

促销通常是无害且不可见的，但当我们将它们与静态 C++ 特性混合时，它们可能会突然出现（稍后会详细介绍）。

Example of `uint16_t` (both a and b) being promoted to `int`.

```
1  uint16_t a = 1;
2  uint16_t b = 2;
3
4  auto v = a - b;
5  // v == -1, decltype(v) == int
```

Open in Compiler Explorer

转换

当两个操作数在提升之后仍然是不同的整数类型时，才会应用转换。

如果类型具有相同的符号性，则将等级较低的操作数转换为等级较高的操作数的类型。

Example of integral conversion. Both operands have the same signedness.

```
1  int a = -100;
2  long int b = 500;
3
4  auto v = a + b;
5  // v == 400, decltype(v) == long int
```

Open in Compiler Explorer

当我们混合使用具有不同符号性的整数类型时，可能会出现三种结果。

当无符号操作数与有符号操作数的等级相同或更高时，有符号操作数会转换为无符号操作数的类型。

Example of integral conversion. The operands have different signedness but the same rank.

```
1  int a = -100;
2  unsigned b = 0;
3
4  auto v = a + b;
5  // v ~ -100 + (UINT_MAX + 1), decltype(v) == unsigned
```

Open in Compiler Explorer

当有符号操作数的类型能够表示无符号操作数的所有值时，无符号操作数将被转换为有符号操作数的类型。

Example of integral conversion. The operands have different signedness, and the type of the signed operand can represent all values of the unsigned operand.

```
1  unsigned a = 100;
2  long int b = -200;
3
4  auto v = a + b;
5  // v = -100, decltype(v) == long int
```

Open in Compiler Explorer

否则，两个操作数都会被转换为该有符号操作数类型的无符号版本。

Example of integral conversion. The operands have different signedness, and the type of the signed operand cannot represent all values of the unsigned operand.

```
long long a = -100;
unsigned long b = 0; // assuming sizeof(long) == sizeof(long long)

auto v = a + b;
// v ~ -100 + (ULLONG_MAX + 1), decltype(v) == unsigned long long
```

Open in Compiler Explorer

由于这些规则，混合整数类型有时会导致非直观的行为。

Demonstration of a condition that changes value based on the type of one of the operands.

```
int x = -1;
unsigned y = 1;
long z = -1;

auto t1 = x > y;
// x -> unsigned, t1 == true

auto t2 = z < y;
// y -> long, t2 == true
```

Open in Compiler Explorer

C++20 安全的整数运算

C++20 标准引入了若干工具，可用于缓解在处理不同整数类型时出现的问题。

首先，标准引入了 `std::ssize()`，允许依赖有符号整数的代码在处理容器时避免混合有符号和无符号整数。

Demonstration of using `std::ssize` to avoid a signed-unsigned mismatch between the index variable and the container size.

```
std::vector<int> data{1,2,3,4,5,6,7,8,9};
// std::ssize returns ptrdiff_t, avoiding mixing
// a signed and unsigned integer in the comparison
for (ptrdiff_t i = 0; i < std::ssize(data); i++) {
    std::cout << data[i] << " ";
}
std::cout << "\n";
// prints: "1 2 3 4 5 6 7 8 9"
```

Open in Compiler Explorer

其次，引入了一组安全的整数比较，用于正确比较不同整数类型的值（避免因类型转换而导致的任何数值变化）。

Demonstration of safe comparison functions.

```
int x = -1;
unsigned y = 1;
long z = -1;

auto t1 = x > y;
auto t2 = std::cmp_greater(x,y);
// t1 == true, t2 == false

auto t3 = z < y;
auto t4 = std::cmp_less(z,y);
// t3 == true, t4 == true
```

Open in Compiler Explorer

最后，一个小型工具 `std::in_range` 将返回被测试的类型是否能够表示所提供的值。

Demonstration of `std::in_range`.

```
auto t1 = std::in_range<int>(UINT_MAX);
// t1 == false
auto t2 = std::in_range<int>(0);
// t2 == true
auto t3 = std::in_range<unsigned>(-1);
// t3 == false
```

Open in Compiler Explorer

5.2.2 浮点类型

浮点类型的规则要简单得多。表达式的结果类型是两个参数中级别最高的浮点类型，即使其中一个参数是整型也是如此（最高级别顺序为：`float`, `double`, `long double`）。重要的是，这一逻辑是按运算符逐个应用的，因此顺序很重要。

In this example, both expressions end up with the resulting type `long double`; however, in the first expression, we lose precision by first converting to `float`.

```
auto src = UINT64_MAX - UINT32_MAX;
auto m = (1.0f * src) * 1.0L;
auto n = 1.0f * (src * 1.0L);
// decltype(m) == decltype(n) == long double

```

```
6  std::cout << std::fixed << m << "\n" << n << "\n" << src << "\n";
7  // 18446744073709551616.000000
8  // 18446744069414584320.000000
9  // 18446744069414584320
```

Open in Compiler Explorer

排序是处理浮动小数时需要记住的主要事项之一（这是一个通用规则，非特定于 C++）。浮动小数的运算是非结合性的。

Demonstration of non-associativity of floating point expressions.

```
1  float v = 1.0f;
2  float next = std::nextafter(v, 2.0f);
3  // next is the next higher floating pointer number
4  float diff = (next-v)/2;
5  // diff is below the resolution of float
6  // importantly: v + diff == v
7
8  std::vector<float> data1(100, diff);
9  data1.front() = v; // data1 == { v, ... }
10 float r1 = std::accumulate(data1.begin(), data1.end(), 0.f);
11 // r1 == v
12 // we added diff 99 times, but each time, the value did not change
13
14 std::vector<float> data2(100, diff);
15 data2.back() = v; // data2 == { ..., v }
16 float r2 = std::accumulate(data2.begin(), data2.end(), 0.f);
17 // r2 != v
18 // we added diff 99 times, but we did that before adding to v
19 // the sum of 99 diffs is above the resolution threshold
```

Open in Compiler Explorer

对不同大小的浮动点数进行任何操作时应小心。

5.2.3 与其他 C++ 功能的互动

尽管整数类型是隐式可互转的，不同整数类型的引用不是相关类型，因此不能相互绑定。这有两个后果。

首先，尝试将左值引用绑定到不匹配的整数类型将不会成功。其次，如果目标引用可以绑定到临时值（右值、常量左值），值将经过隐式转换，引用将绑定到生成的临时值。

Demonstration of integral type behaviour with reference types.

```
void function(const int& v) {}

long a = 0;
long long b = 0;
// Even when long and long long have the same size
static_assert(sizeof(a) == sizeof(b));
// The two types are unrelated in the context of references
// The following two statements wouldn't compile:
// long long& c = a;
// long& d = b;

// OK, but dangerous, implict conversion to int
// int temporary can bind to const int&
function(a);
function(b);
```

Open in Compiler Explorer

最后，我们需要讨论类型推导。由于类型推导是一个静态过程，它确实消除了进行隐式转换的机会。然而，这也带来了潜在的问题。

Demonstration of potential problem of type deduction when using standard algorithms.

```
std::vector<unsigned> data{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v = std::accumulate(data.begin(), data.end(), 0);
// 0 is a literal of type int. Internally this means that
// the accumulator (and result) type of the algorithm will be
// int, despite iterating over a container of type unsigned.

// v == 45, decltype(v) == int
```

Open in Compiler Explorer

但与此同时，当与概念混合时，我们可以减轻隐式转换，同时仅接受特定的整数类型。

Demonstration of using concepts to prevent implicit conversions.

```
template <typename T> concept IsInt = std::same_as<int, T>;
void function(const IsInt auto&) {}

function(0); // OK
// function(0u); // will fail to compile, deduced type unsigned
```

Open in Compiler Explorer

索引