C++17 STL Cookbook

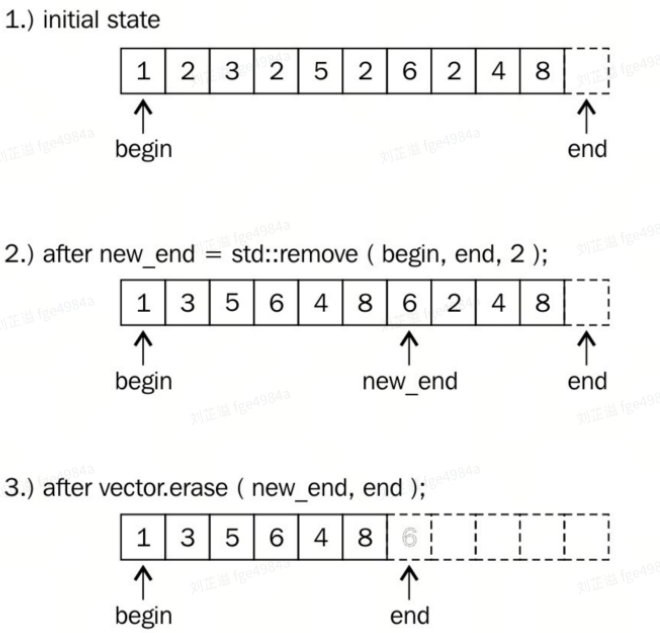
STL容器

- 连续存储：array, vector, deque(存储在多段固定长度的连续内存中，这些内存段是相互独立的)
- 列表存储：std::list 是一个典型的双向链表。如果是单向列表，那就需要进行遍历，所 以std::forward_list 的优势在维护的复杂性上，因为其指针方向只有一个方向。列 表遍历的时间复杂度是线性的O(n)。其在特定位置上插入和删除元素的时间复杂度 为O(1)。
- 搜索树：set，map
- 哈希表：std::unordered_set 和 std::unordered_map
- 容器适配器：数组、列表、树和哈希表并不是存储和访问数据的唯一方式，这里还有栈、队列等 其他的方式也可以存储和访问数据。类似的，越复杂的结构可以使用越原始的方式 实现，并且STL使用以下形式的容器适配器进行操作： **std::stack，std::queue 和 std::priotity_queue** 。

```
1   std::stack<std::vector<int>> stack;
```

擦除/移除std::vector元素

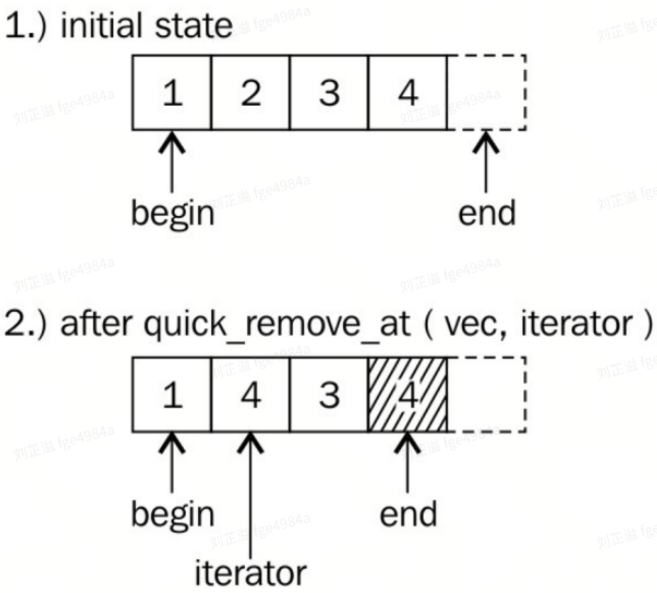
```
1   const auto
    new_end(remove(begin(v),
    end(v), 2));
2   v.erase(new_end, end(v));
3   // 等价于
4   v.erase(remove_if(begin(v),
    end(v), func), end(v));
5
6   v.shrink_to_fit();
7   // 会让vector重新分配一段内存，以匹
    配相应元素长度，vector中已存的元素
    会移动到新的内存块中。
```



O(1)删除未排序的std::vector的元素

- 代码直接使用最后一个值覆盖了选定元素的值。为什么要这样？当我们交 换元素时，就需要将选定的元素存储在一个临时变量中，并在最后将这个临时变量 中的值放在 vector 的最后。这个临时变量是多余的，而且要删除的值对于我们来说 是没有意义的，所以这里选择了直接覆盖的方式，更加高效的实现了删除。

```
template  void
quick_remove_at(std::vector &v,
 typename std::vector::iterator
it) {
    if(it != std::end(v)) {
        *it =
std::move(v.back());
        v.pop_back();
    }
}
```



保持对std::vector实例的排序

- 这里我们假设的情况是，在插入之前， vector 已经排序。否则，这种方法无 法工作。

```
void insert_sorted(vector &v, const string &word)
{
    const auto insert_pos (lower_bound(begin(v), end(v), word));
    v.insert(insert_pos, word);
}
```

向std::map实例中高效并有条件的插入元素

- std::map 中 insert 和 emplace 方法完全相同。 try_emplace 与它们不同的地 方在于，**在遇到已经存在的键时，不会去构造组对**。当相应对象的类型需要很大开销进行构造时，这对于程序性能是帮助的。

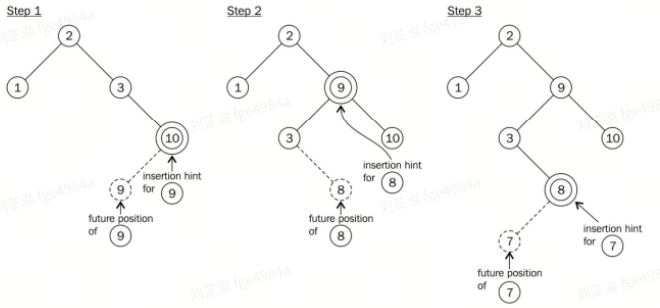
```
std::pair try_emplace(const key_type& k, Args&&... args);
```

std::map::insert新的插入提示语义

- 我们将插入多个元素，对于每次插入，我们都会传入一个hint迭代器。第一次 插入我们不指定其开始位置，只将插入位置指向 map 的 end 迭代器之前

- 如果hint提示正确，那么插入时间复杂度为**O(1)**
- 当我们将元素插入到树中时，这些键值就会成为邻居(就如整数1和2互邻一样)。如 果有 hint 传入，那么很容易检查键是否正确。**如果这种情况出现，则可以省去搜索的时间**。而后，平衡算法会可能还要运行。虽然优化并不是总能成功，不过平均 下来，性能上还是会有提升。**可以使用多次插入的方式，来统计运行的耗时，这被称之为摊销复杂度**

```
auto insert_it (std::end(m));
for (const auto &s : {"z", "y", "x", "w"}) {
  insert_it = m.insert(insert_it, {s, 1});
}
```



高效的修改std::map元素的键值

- 当使用第二种方式去提取一个不存在的节点时，会返回一个空 node_type 实 例。

```
node_type extract(const_iterator position);
node_type extract(const key_type& x)

{
    auto a(race_placement.extract(3));
    auto b(race_placement.extract(8));
    swap(a.key(), b.key());
    race_placement.insert(move(a));
    race_placement.insert(move(b));
}
```

std::unordered_map使用自定义类型

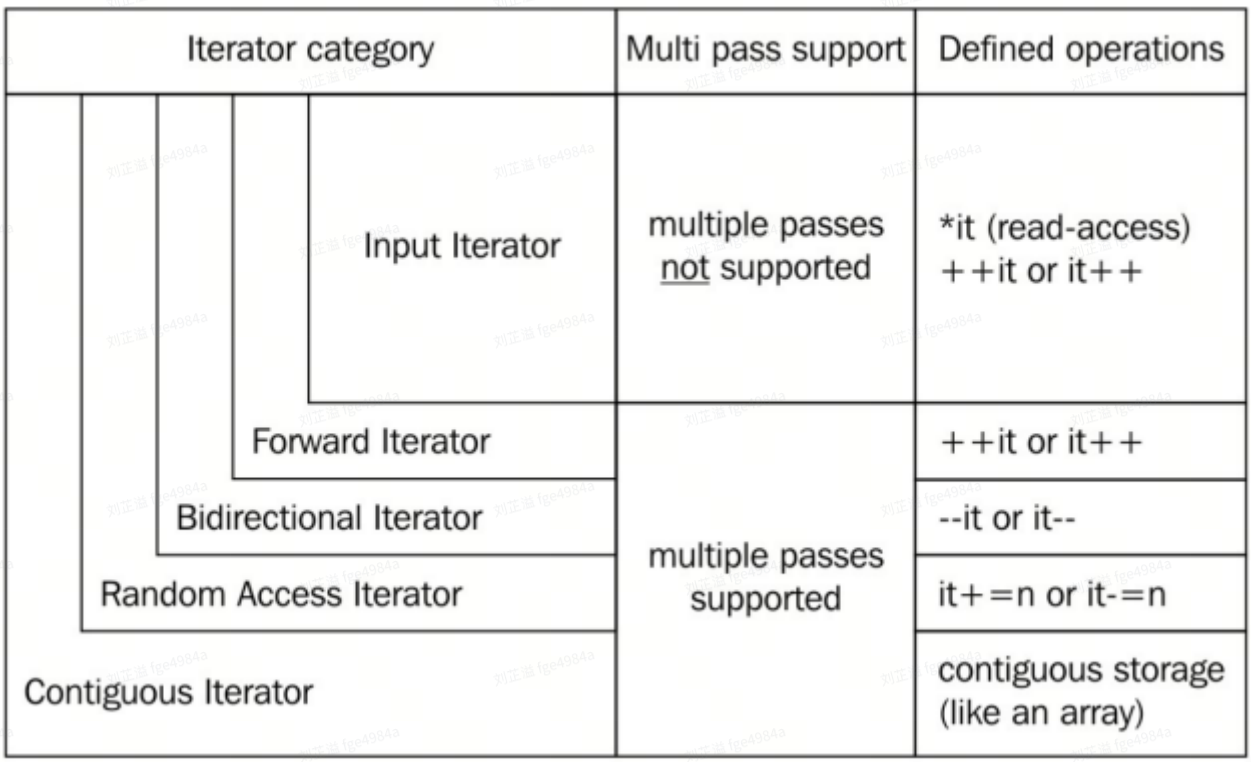
```
struct coord{
    int x;
    int y;
};

bool operator==(const coord &l, const coord &r)
{  return l.x == r.x && l.y == r.y; }

namespace std {
    template <> struct hash {
        using argument_type = coord;
```

```
12          using result_type = size_t;
13          result_type operator()(const argument_type &c) const
14          {
15              return static_cast(c.x)  + static_cast(c.y);
16          }
17      };
18  }
19
20  std::unordered_map<coord, value_type, my_hash_type> my_unordered_map;
```

迭代器



- **输入迭代器**：只能用来读取指向的值；当该迭代器自加时，之前指向的值就不可访问。
  - std::istream_iterator 就是这样的迭代器。
- **前向迭代器**：类似于输入迭代器，可以在指示范围迭代多次
  - std::forward_list 就是这样的迭代器。就像一个单向链表一样，只能向前遍历，不能向后遍历，但可以反复迭代。
- **双向迭代器**：这个迭代器可以自增，也可以自减，迭代器可以向前或向后迭代。
  - std::list，std::set 和 std::map 都支持双向迭代器。
- **随机访问迭代器**：与其他迭代器不同，随机访问迭代器一次可以跳转到任何容器中的元素上，而非之 前的迭代器，一次只能移动一格。 std::vector 和 std::deque 的迭代器就是这种类 型。

- **连续迭代器**：这种迭代器具有前述几种迭代器的所有特性，不过需要容器内容在内存上是连续 的，类似一个数组或 std::vector 。
- **输出迭代器**：该迭代器与其他迭代器不同。因为这是一个单纯用于写出的迭代器，其只能增加， 并且将对应内容写入文件当中。如果要读取这个迭代中的数据，那么读取到的值就是未定义的
- **可变迭代器**：如果一个迭代器既有输出迭代器的特性，又有其他迭代器的特性，那么这个迭代器 就是可变迭代器。该迭代器可读可写。如果我们从一个非常量容器的实例中获取一个迭代器，那么这个迭代器通常都是可变迭代器。

迭代器与标准库兼容

- num_iterator 结构体，我们会对 std::iterator_traits 进行特化。这个特化就 是告诉STL我们的**num_iterator 是一种前向迭代器，并且指向的对象是 int 类型的值。**
- 迭代器需要指定的特性
  - difference_type：it1 - it2结果的类型
  - value_type：迭代器解引用的类型
  - pointer：指向元素指针的类型
  - reference：引用元素的类型
  - iterator_category：迭代器属于哪种类型

```
namespace std {
    template <> struct iterator_traits {
        using iterator_category = std::forward_iterator_tag;
        using value_type = int;
    };
}
```

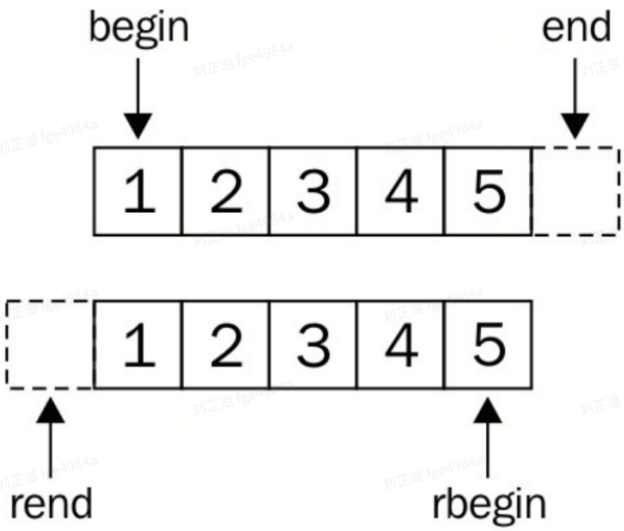
使用迭代适配器填充通用数据结构

- **std::back_insert_iterator：**
  - back_insert_iterator 可以包装 std::vector 、 std::deque 、 std::list 等容器。其会**调用容器的 push_back 方法**在容器最后插入相应的元素。如果容器实例不够长，那 么容器的容量会自动增长。
- **std::front_insert_iterator**
  - front_insert_iterator 和 back_insert_iterator 一样，不过 front_insert_iterator **调用的是容器的 push_front 函数**，也就是在所有元素前插入元素。这里需要注意的是， **当对类似于 std::vector 的容器进行插入时，其已经存在的所有元素都要后移，从而空出位置来放插入元素，这会对性能造成一定程度的影响。**

- **std::insert_iterator**
  - 这个适配器与其他插入适配器类似，不过能在容器的中间位置插入新元素。使 用 std::inserter 包装辅助函数需要两个参数。**第一个参数是容器的实例，第二个参数是迭代器指向的位置，就是新元素插入的位置**。
- **std::istream_iterator**
  - istream_iterator 是另一种十分方便的适配器。其能对任何 std::istream 使用(文件流或标准输入流)，并且可以根据实例的具体特化类型，对流进行分析。本节中， **我们使用了 std::istram_iterator(std::cin) ，其会将整数从标准输入中拉出来**。 通常，对于流来说，其长度我们是不知道的。这就存在一个问题，也就是 end 迭代器指向的位置在哪里？**对于流迭代器来说，它就知道相应的 end 迭代器的位置。**这样就使得迭代器的比较更加高效，不需要通过遍历来完成。这样就是为什么 end 流迭代器不需要传入任何参数的原因。
- **std::ostream_iterator**
  - ostream_iterator 和 istream_iterator 类似，不过是用来进行输出的流迭代器。 **与 istream_iterator 不同在于，构造时需要传入两个参数，且第二个参数必须要是一个字符串，这个字符串将会在各个元素之后，推入输出流中**。这样我们就能很容易的在元素中间插入逗号或者换行的符号，以便用户进行观察。

反向迭代器

- std::make_reverse_iterator 工厂函数，将普通的迭代器包装成反向迭代 器即可，背后的操作STL会帮我们完成。



使用哨兵终止迭代

- C++17添加了一项新的特性，其不需要 begin 迭代器和 end 迭代器是同一类型的迭 代器。

```
1  // 确定c风格字符串的end()迭代器
2  class cstring_end_iterator {};
3
4  class cstring_iterator {
5  private:
```

```
    const char *s{nullptr};

public:
    explicit cstring_iterator(const char *str) : s(str) {}
    char operator*() const { return *s; }
    cstring_iterator &operator++() {
      ++s;
      return *this;
    }

    bool operator!=(const cstring_end_iterator) const {
      return s != nullptr && *s != '\0';
    }
};

class cstring_range {
    const char *s{nullptr};

public:
    cstring_range(const char *str) : s(str) {}

    cstring_iterator begin() const { return cstring_iterator{s}; }
    cstring_end_iterator end() const { return {}; }
};

int main(int argc, char *argv[]) {
    if (argc < 1) {
      std::cout << "Usage: param\n";
      return 1;
    }
    for (char c : cstring_range(argv[1])) {
      std::cout << c;
    }
    std::cout << '\n';
}
```

构建zip迭代适配器

```
class zip_iterator {
    using item_type = std::vector<double>::iterator;
    item_type it1;
    item_type it2;

public:
    zip_iterator(item_type iterator1, item_type iterator2)
```

```
        : it1(iterator1), it2(iterator2) {}

    zip_iterator &operator++() {
      ++it1;
      ++it2;
      return *this;
    }

    bool operator!=(const zip_iterator &other) const {
      return it1 != other.it1 && it2 != other.it2;
    }

    std::pair<double, double> operator*() const { return {*it1, *it2}; }
  };

  namespace std {
  template <> struct iterator_traits<zip_iterator> {
    using iterator_category = std::forward_iterator_tag;
    using value_type = std::pair<double, double>;
    using difference_type = long int;
  };
  } // namespace std

  class zipper {
    using vec_type = std::vector<double>;
    vec_type &v1;
    vec_type &v2;

  public:
    zipper(vec_type &va, vec_type &vb) : v1(va), v2(vb) {}
    zip_iterator begin() const { return {std::begin(v1), std::begin(v2)}; }
    zip_iterator end() const { return {std::end(v1), std::end(v2)}; }
  };
```

Lambda表达式

Lambda表达式定义函数

```
[capture list] (parameters)
    mutable (optional)
    constexpr (optional)
    exception attr (optional)
    -> return type (optional)
    {  body }
```

捕获列表capture list

- 将Lambda表达式写成 [=] () {...} 时，会捕获到外部所有变量的副本。
- 将Lambda表达式写成 [&] () {...} 时，会捕获到外部所有变量的引用。

mutable (optional)

- 当函数对象需要去修改通过副本传入的变量时，表达式必须用 mutable 修饰。这就相当于对捕获的对象使用非常量函数。

constexpr (optional)

- 如果我们显式的将Lambda表达式修饰为 constexpr ，编译器将不会通过编译，因为 其不满足 constexpr 函数的标准。 constexpr 函数有很多条件，编译器会在编译时对 Lambda表达式进行评估，看其在编译时是否为一个常量参数，这样就会让程序的二进制文件体积减少很多。 **当我们不显式的将Lambda表达式声明为 constexpr 时，编译器就会自己进行判断， 如果满足条件那么会将Lambda表达式隐式的声明为 constexpr 。当我们需要一个 Lambda表达式为 constexpr 时，我们最好显式的对Lambda的表达式进行声明**，当编译不通过时，编译器会告诉我们哪里做错了。

exception attr (optional)

- 这里指定在运行错误时，是否抛出异常。

return type (optional)

- 想完全控制返回类型时，我们不会让编译器来做类型推导。我们可以写成这样 [] () -> Foo {} ，这样就告诉编译器，这个Lambda表达式总是返回 Foo 类型的结果。

使用同一输入调用多个函数

- 初始化表达式中，其将 f(xs)... 包装进 (f(xs),0)... 表达式中。这会让程序将返回值完全抛弃，不过0将会放置在初始化列表中。

```
template  static auto multicall (Ts ...functions)
{
    return [=](auto x) {
    (void)std::initializer_list{
            ((void)functions(x), 0)...
        };
    };
}

// 我能做的只能是构造出一个 std::initializer_list 列表，其具有一个可变的 构造函数。
// 表达式可以直接通过 return std::initializer_list{f(xs)...}; 方式构 建
```

```
template <typename F, typename... Ts> static auto for_each(F f, Ts... xs) {
  (void)std::initializer_list<int>{((void)f(xs), 0)...};
}
```

- std::accumulate 函数会将所对应范围内的数值进行累加。

编译时生成笛卡尔乘积

- 第一次看起来有些奇怪。调用 call_cart 时，我们第一次对 xs 进行了 扩展。第二次扩展将会使得 call_cart 调用多次，并且每次的第二个参数都会不同。
- 其内部Lambda表达式将会生成如下调用：call_cart(f, 1, 1, 2, 3); call_cart(f, 2, 1, 2, 3); call_cart(f, 3, 1, 2, 3);。

```
constexpr auto cartesian ([=](auto ...xs)
constexpr {
    return [=] (auto f) constexpr {
        (void)std::initializer_list{
            ((void)call_cart(f, xs, xs...), 0)...
        };
    };
});
```

STL基础算法

STL算法好处

- 维护性：算法的名字已经说明它要做什么了。显式使用循环的方式与使用STL 算法的方式没法对比。
- 正确性：STL是由专家编写和审阅过的，并且经过了良好的测试，重新实现的 复杂程度可能是你无法想象的。
- 高效性：STL算法真的很高效，至少要比手写的循环要强许多。

容器间相互复制元素

- std::copy 是STL中最简单的算法之一，其实现也非常短。我们可以看一下等价实 现：

```
template  OutputIterator copy(InputIterator it, InputIterator end_it,
OutputIterator out_it) {
    for (; it != end_it; ++it, ++out_it)
        {  *out_it = *it;  }
```

```
4       return out_it;
5   }
```

容器元素排序

算法函数	作用
std::sort	接受一定范围的元素，并对元素进行排序。
std::is_sorted	接受一定范围的元素，并判断该范围的元素是否经过排序。
std::shuffle	类似于反排序函数；其接受一定范围的元素，并打乱这些元素。
std::partial_sort	接受一定范围的元素和另一个迭代器，前两个参数决定排序的范围，后两个参数决定不排序的范围。
std::partition	能够接受谓词函数。所有元素都会在谓词函数返回true时，被移动到范围的前端。剩下的将放在范围的后方。

- 当然排序还有其他类似 std::stable_sort 的函数，其能保证排序后元素的原始顺 序，std::stable_partition 也有类似的功能。

从容器中删除指定元素

算法函数	作用
std::remove	接受一个容器范围和一个具体的值作为参数，并且移除对应的值。返回一个新的end迭代器，用于修改容器的范围。
std::replace	接受一个容器范围和两个值作为参数，将使用第二个数值替换所有与第一个数值相同的值。
std::remove_copy	接受一个容器范围，一个输出迭代器和一个值作为参数。并且将所有不满足条件的元素拷贝到输出迭代器的容器中。
std::replace_copy	与 `std::replace` 功能类似，但与 `std::remove_copy` 更类似些。源容器的范围并没有变化。
std::copy_if	与 `std::copy` 功能相同，可以多接受一个谓词函数作为是否进行拷贝的依据。

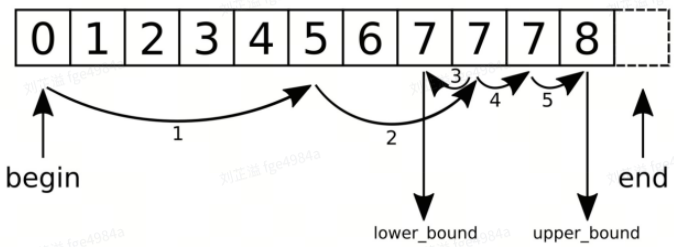
改变容器内容

- std::transform 函数工作原理和 std::copy 差不多，不过在拷贝的过程中会对源容器 中的元素进行变换，这个变换函数由用户提供。

在有序和无序的vector中查找元素

算法函数	作用
std::find	可将一个搜索范围和一个值作为参数。函数将返回找到的第一个值的迭代器。线性查找。
std::find_if	与 std::find 原理类似，不过其使用谓词函数替换比较值。
std::binary_search	可将一个搜索范围和一个值作为参数。执行二分查找，当找到对应元素时，返回true；否则，返回false。
std::lower_bound	可将一个查找返回和一个值作为参数，并且执行二分查找，返回第一个不小于给定值元素的迭代器。
std::upper_bound	与 std::lower_bound 类似，不过会返回第一个大于给定值元素的迭代器。
std::equal_range	可将一个搜索范围和一个值作为参数，并且返回一对迭代器。其第一个迭代器和 std::lower_bound 返回结果一样，第二个迭代器和 std::upper_bound 返回结果一样。

- 首先， equal_range 会使用典型的二分查找，直到其找到那个不小于查找值的那个 元素。而后，另一个迭代器也是用同样的方式找到。如同分开调 用 lower_bound 和 upper_bound 一样。




```
template  Iterator standard_binary_search(Iterator it, Iterator end_it, T value) {
const auto potential_match (lower_bound(it, end_it, value));
    if (potential_match != end_it && value == *potential_match) {
        return potential_match;
    }
    return end_it;
}
```

- 需要留意 std::map 和 std::set 等数据结构，它们有自己的 find 函数。它们携带的 find 函数要比通用的算法快很多，因为他们的实现与数据结构强耦合。

vector值控制在特定范围

- 将 vector 中的值使用两种不同的方式进行归一化，一种使用std::minmax_element ，另一种使用 std::clamp

```
static auto norm (int min, int max, int new_max) {
    const double diff (max - min);
```

```
    return [=] (int val) {
        return int((val - min) / diff * new_max);
    };
}

static auto clampval (int min, int max) {
    return [=] (int val) -> int {
        return clamp(val, min, max);
    };
}
```

- std::minmax_element 能接受一对 begin 和 end 迭代器作为输入。其会对这个范围进 行遍历，然后找到这个范围内的最大值和最小值。其返回值是一个组对，我们会在 我们的缩放函数中使用这个组对。
- std::clamp 函数无法对一个范围进行可迭代操作。其接受三个值作为参数：一个给 定值，一个最小值，一个最大值。这个函数的返回值则会将对应的值截断在最大值 和最小值的范围内。我

在字符串中定位模式并选择最佳实现 ——std::search

- C++17版本的 std::search 将会使用一组 begin/end 迭代器和一个所要查找的对 象。std::default_searcher 能接受一组子字符串的 begin 和 end 迭代器，再在 一个更大的字符串中，查找这个字符串：

```
{
auto match (search(
    begin(long_string),
    end(long_string),
    default_searcher(begin(needle), end(needle))));
    print(match, 5);
}
```

- 其他搜索方式将会以更复杂的方式实现：
  - std::default_searcher： 其会重定向到 std::search 的实现。
  - std::boyer_moore_searcher： 使用Boyer-Moore查找算法。
  - std::boyer_moore_horspool_searcher： 使用Boyer-Moore-Horspool查找算法。

工具类

std::variant

v = variant<int, double, std::string> ，则 v 是一个可存放 int, double, std::string 这三种类型数据的变体类型的对象。

- v.index() 返回变体类型 v 实际所存放数据的类型的下标。变体中第1种类型下标为0，第2种类型下标为1，以此类推。
- std::holds_alternative<T>(v) 可查询变体类型 v 是否存放了 T 类型的数据。
- std::get<I>(v) 如果变体类型 v 存放的数据类型下标为 I，那么返回所存放的数据，否则报错。
- std::get_if<I>(&v) 如果变体类型 v 存放的数据类型下标为 I，那么返回所存放数据的指针，否则返回空指针。
- std::get<T>(v) 如果变体类型 v 存放的数据类型为 T，那么返回所存放的数据，否则报错。
- std::get_if<T>(&v) 如果变体类型 v 存放的数据类型为 T，那么返回所存放数据的指针，否则返回空指针。
- visit() 基本上是一個用來處理 variant 型別的函式

```
struct SOutput
{
        void operator()(const int& i)
        {
                std::cout << i << std::endl;
        }

        void operator()(const double& d)
        {
                std::cout << d << std::endl;
        }

        void operator()(const std::string& s)
        {
                std::cout << s << std::endl;
        }
};

std::visit(SOutput(), y);
```