



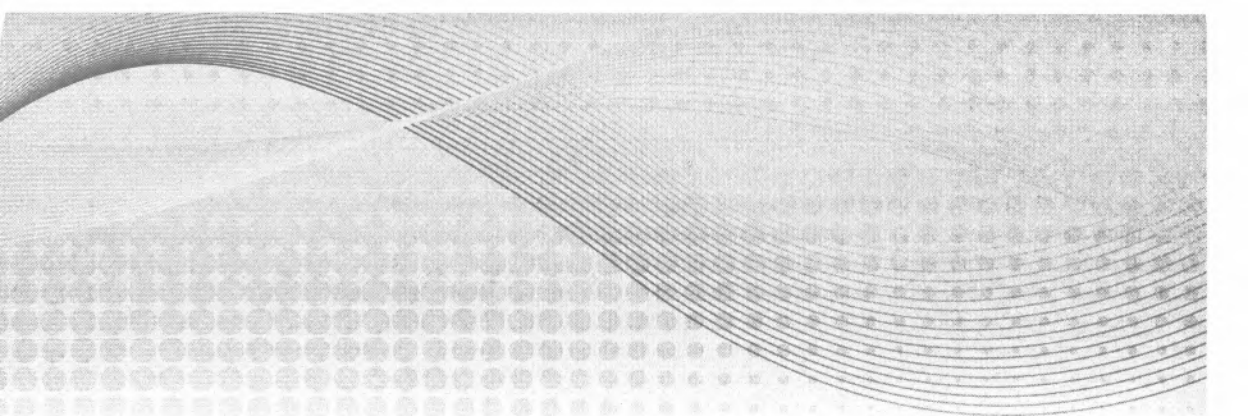
Professional C++, Fifth Edition

C++20高级编程

(第5版) (下册)

[比] 马克·格雷戈勒(Marc Gregoire) 著
程序喵大人 惠惠 墨梵 译

清华大学出版社



C++20 高级编程

(第 5 版)

(下册)

[比] 马克·格雷戈勒(Marc Gregoire) 著
程序喵大人 惠惠 墨梵 译

清华大学出版社

北 京

北京市版权局著作权合同登记号 图字：01-2021-3641

All Rights Reserved. This translation published under license. Authorized translation from the English language edition, entitled Professional C++, Fifth Edition, ISBN 9781119695400, by Marc Gregoire, Published by John Wiley & Sons. Copyright © 2021 by Marc Gregoire. No part of this book may be reproduced in any form without the written permission of the original copyrights holder.

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书中文简体字版由 Wiley Publishing, Inc. 授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或传播本书内容。

本书封面贴有 Wiley 公司防伪标签，无标签者不得销售。

版权所有，侵权必究。举报：010-62782989, beiqinquan@tup.tsinghua.edu.cn。

图书在版编目(CIP)数据

C++20高级编程：第5版 / (比)马克·格雷戈勒(Marc Gregoire) 著；程序喵大人，惠惠，墨梵译. —北京：清华大学出版社，2022.3 (2022.11重印)

书名原文：Professional C++, Fifth Edition

ISBN 978-7-302-60213-2

I. ①C… II. ①马… ②程… ③惠… ④墨… III. ①C++语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2022)第 033304 号

责任编辑：王 军

装帧设计：孔祥峰

责任校对：成凤进

责任印制：宋 林

出版发行：清华大学出版社

网 址：http://www.tup.com.cn, http://www.wqbook.com

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-83470000 邮 购：010-62786544

投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈：010-62772015, zhiliang@tup.tsinghua.edu.cn

印 装 者：北京同文印刷有限责任公司

经 销：全国新华书店

开 本：170mm×240mm 印 张：63 字 数：1805 千字

版 次：2022 年 4 月第 1 版 印 次：2022 年 11 月第 3 次印刷

定 价：228.00 元(全二册)

产品编号：091909-01

第 17 章

理解迭代器与范围库

本章内容

- 迭代器的概念
- 如何使用流迭代器
- 迭代器适配器的定义，以及如何使用标准迭代器适配器
- 范围库的强大功能包括：范围、基于范围的算法、投影、视图和工厂

第 16 章“C++标准库概述”介绍了标准库，描述了标准库的基本原理，并对各种不同的容器和算法做了概述。本章通过介绍标准库中广泛使用的迭代器背后的思想，开始深入讲解标准库容器。本章还讨论可用的流迭代器和迭代器适配器。本章的第二部分讨论了 C++20 范围库，这是标准库的一个功能强大的新添加的库，它允许更多的函数式编程(functional-style programming)：可以编写代码来指定想要完成的任务。

17.1 迭代器

标准库通过迭代器模式提供了访问容器元素使用的通用抽象。每个容器都提供了容器特定的迭代器，迭代器实际上是增强版的智能指针，这种指针知道如何遍历特定容器的元素，也就是说，迭代器支持遍历容器的元素。所有不同容器的迭代器都遵循 C++标准中定义的特定接口。因此，即使容器提供不同的功能，访问容器元素的代码也可以使用迭代器的统一接口。

可将迭代器想象为指向容器中特定元素的指针。与指向数组元素的指针一样，迭代器可以通过 `operator++` 移到下一个元素。与此类似，通常还可在迭代器上使用 `operator*` 和 `operator->` 来访问实际元素或元素中的字段。一些迭代器支持通过 `operator==` 和 `operator!=` 进行比较，还支持通过 `operator--` 转移到前一个元素。

所有迭代器都必须是可以拷贝构造和拷贝赋值的，并且是可以析构的。迭代器的左值必须是可以交换的。不同容器提供的迭代器具有略微不同的功能。C++标准定义了 6 大类迭代器，如表 17-1 所示。

表 17-1 迭代器

迭代器类别	要求的操作	注释
输入迭代器(也称为“读”迭代器)	<code>operator++</code> <code>operator*</code> <code>operator-></code> 拷贝构造函数 <code>operator=</code> <code>operator==</code> <code>operator!=</code>	提供只读访问，只能前向访问(没有 <code>operator--</code> 提供的反向访问功能)。这个迭代器可以赋值和复制，可以比较判等
输出迭代器(也称为“写”迭代器)	<code>operator++</code> <code>operator*</code> 拷贝构造函数 <code>operator=</code>	提供只写访问，只能前向访问 这个迭代器只赋值，不能比较判等 输出迭代器的特有操作是 <code>*iter = value</code> 注意此类迭代器缺少 <code>operator-></code> 提供前缀和后缀 <code>operator++</code>
前向迭代器	输入迭代器的功能加上默认构造函数	提供读访问，只能前向访问 这个迭代器可以赋值、复制和比较判等
双向迭代器	前向迭代器的功能加上 <code>operator--</code>	提供前向迭代器的一切功能 迭代器还可以回退到前一个元素 提供前缀和后缀 <code>operator--</code>
随机访问迭代器	双向迭代器的功能加上: <code>operator+</code> <code>operator-</code> <code>operator+=</code> <code>operator-=</code> <code>operator<</code> <code>operator></code> <code>operator<=</code> <code>operator>=</code> <code>operator[]</code>	等同于普通指针：此类迭代器支持指针运算、数组索引语法以及所有形式的比较
连续迭代器	随机访问功能和逻辑上相邻的容器元素必须在内存中物理上相邻	例如 <code>std::array</code> 、 <code>vector</code> (非 <code>vector<bool></code>)、 <code>string</code> 和 <code>string_view</code> 的迭代器

根据表 17-1，有 6 种类型的迭代器：输入、输出、前向、双向、随机访问和连续迭代器。这些迭代器没有正式类层次结构。但是，可以根据需要提供的功能推断出层次结构。具体来说，每个连续迭代器也是随机访问的，每个随机访问迭代器也是双向的，每个双向迭代器也是前向的，每个前向迭代器也是输入的。另外满足输出迭代器要求的迭代器称为可迭代器；否则它们被称为常量迭代器。图 17-1 显示了这样的层次结构。使用虚线是因为该图所示不是一个真正的类层次结构。

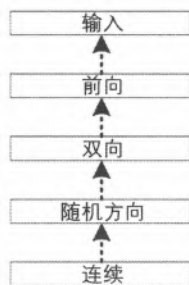


图 17-1 层次结构

第 20 章中讨论的算法指定需要哪种类型的迭代器的标准方法是对迭代器模板类型参数使用以下的名称：InputIterator、OutputIterator、ForwardIterator、BidirectionalIterator、RandomAccessIterator 和 ContiguousIterator。这些名称只是名称，它们不提供绑定类型检查。因此，例如，可以尝试通过传递一个双向迭代器来调用一个需要随机访问迭代器的算法。模板无法进行类型检查，因此它将允许这个实例化。但是，函数中使用随机访问迭代器功能的代码将无法在双向迭代器的参数上编译通过。因此，该要求是强制执行的，并不是原先以为的那样。因此，错误消息可能会有些混乱。例如，试图在只提供双向迭代器的列表上使用泛型 sort() 算法，它需要一个随机访问迭代器，但这可能会导致一个晦涩的错误。以下是 Visual C++ 2019 生成的错误信息：

```

...\\VC\\Tools\\MSVC\\14.27.29109\\include\\algorithm(4138,45): error C2676:
binary '-': 'const std::_List_unchecked_iterator<std::_List_val<std::_List_simple_
types<_Ty>>>' does not define this operator or a conversion to a type acceptable
to the predefined operator
    with
    [
        _Ty=int
    ]

```

在本章的后面将介绍 C++20 的范围库，它与大多数标准库算法的基于范围的版本一起提供。这些基于范围的算法对它们的模板类型参数具有适当的类型约束(参见第 12 章“利用模板编写泛型代码”)。因此，如果试图在提供错误迭代器类型的容器上执行这样的算法，编译器可以提供更清晰的错误消息。

注意：

迭代器是算法和容器之间的中介。它提供了一个标准接口来依次遍历容器中的元素，这样任何算法都可以在任何容器上工作，只要容器提供了算法所需的迭代器类别。

迭代器的实现类似于智能指针类，因为它重载了特定的运算符。运算符重载详见第 15 章“C++ 运算符重载”。

基本的迭代器操作类似于原始指针(raw pointer)支持的操作，因此原始指针可以用作特定容器的迭代器。事实上，从技术上讲，vector 迭代器可以实现为一个简单的原始指针。然而，作为容器的客户，不用担心实现的细节；只需要简单地使用迭代器的抽象就可以了。

注意：

迭代器在内部可能不是实现为指针，因此本书在讨论通过迭代器访问元素时，使用的是“引用”而不是“指向”。

17.1.1 获取容器的迭代器

标准库中每个支持迭代器的容器类都为其迭代器类型提供了公共类型别名，名为 `iterator` 和 `const_iterator`。例如，`int` 类型的 `vector` 的 `const` 迭代器类型是 `std::vector<int>::const_iterator`。允许反向迭代元素的容器还提供了名为 `reverse_iterator` 和 `const_reverse_iterator` 的公共类型别名。通过这种方式，客户使用容器迭代器时不需要关心实际类型。

注意：
`const_iterator` 和 `const_reverse_iterator` 提供对容器元素的只读访问。

容器还提供了 `begin()` 和 `end()` 方法。`begin()` 方法返回引用容器中第一个元素的迭代器，`end()` 方法返回的迭代器等于在引用序列中最后一个元素的迭代器上执行 `operator++` 后的结果。`begin()` 和 `end()` 一起提供了一个半开区间(half-open range)，包含第一个元素但不包含最后一个元素。采用这种看似复杂方式的原因是为了支持空区间(不包含任何元素的容器)，此时 `begin()` 等于 `end()`。由 `begin()` 和 `end()` 限定的半开区间常写成数学形式：`[begin, end)`。

此外，还可以使用以下方法：

- 返回 `const` 迭代器的 `cbegin()` 和 `cend()` 方法
- 返回反向迭代器的 `rbegin()` 和 `rend()` 方法
- 返回 `const` 反向迭代器的 `crbegin()` 和 `crend()` 方法

`<iterator>` 头文件中还提供了如表 17-2 所示的全局非成员函数来查找容器中的特定迭代器。

表 17-2 全局非成员函数

函数名称	函数概要
<code>begin()</code> <code>end()</code>	返回一个非常量迭代器，指向序列中的第一个元素和最后一个元素的下一个元素
<code>cbegin()</code> <code>cend()</code>	返回一个常量迭代器，指向序列中的第一个元素和最后一个元素的下一个元素
<code>rbegin()</code> <code>rend()</code>	返回一个非常量反向迭代器，指向序列中最后一个元素和第一个元素前一个的元素
<code>crbegin()</code> <code>crend()</code>	返回一个常量反向迭代器，指向序列中最后一个元素和第一个元素前一个的元素

注意：
建议使用这些非成员函数而不是其成员版本。

这些非成员函数在 `std` 名称空间中定义；但是，特别是在为类和函数模板编写泛型代码时，建议使用下面的这些非成员函数：

```
using std::begin;
begin(...);
```

请注意，调用 `begin()` 时没有使用 `std::` 限定，因为这会启用所谓的参数相关查找(ADL)。当为自定义的类型特化这些非成员函数之一时，可以将这些特化放在 `std` 名称空间中，或者将它们放在与特化它们的类型相同的名称空间中。建议使用后者，因为这样就可以启用 ADL。这意味着不需要使用任何

名称空间就可以调用特化，因为编译器可以根据传递给特化函数模板的类型在你的名称空间中找到正确的特化。该标准要求编译器首先使用 ADL 在其参数类型的名称空间中查找正确的重载。如果编译器无法使用 ADL 找到重载，由于 `using std::begin` 声明，因此它应该尝试在 `std` 名称空间中查找合适的重载。在没有 `using` 声明的情况下，调用 `begin()` 只会通过 ADL 调用用户定义的重载；而只调用 `std::begin()` 的情况下，则只会在 `std` 名称空间中查找。

注意：

通常不允许向 `std` 名称空间添加任何内容；但是，将标准库模板的特化放在 `std` 名称空间中是合法的。

还可使用 `std::distance()` 计算容器的两个迭代器之间的距离。

17.1.2 迭代器萃取

一些算法实现需要关于迭代器的附加信息。例如，可能需要知道迭代器为存储临时值所引用的元素的类型，或者可能想知道迭代器是双向访问，还是随机访问。

C++ 提供了一个名为 `iterator_traits` 的类模板，定义在 `<iterator.h>` 头文件中，它允许用户查找这些信息。可以使用感兴趣的迭代器类型来实例化 `iterator_traits` 类模板，并访问 5 个类型别名中的一个。

- `value_type`: 引用的元素类型
- `difference_type`: 一种能够表示距离的类型，例如，两个迭代器之间的元素的数量
- `iterator_category`: 迭代器的类型是 `input_iterator_tag`、`output_iterator_tag`、`forward_iterator_tag`、`bidirectional_iterator_tag`、`random_access_iterator_tag` 或 `contiguous_iterator_tag` (C++20)
- `pointer`: 指向元素的指针类型
- `reference`: 元素引用的类型

例如，下面的函数模板声明了一个临时变量，该变量的类型与 `IteratorType` 类型的迭代器所指向的类型相同。注意，在 `iterator_traits` 行前面使用了 `typename` 关键字。每当访问基于一个或多个模板类型参数的类型时，必须显式指定 `typename`。在这种情况下，模板类型参数 `IteratorType` 用于访问 `value_type` 类型。

```
template <typename IteratorType>
void iteratorTraitsTest(IteratorType it)
{
    typename iterator_traits<IteratorType>::value_type temp;
    temp = *it;
    cout << temp << endl;
}
```

这个函数可以使用下面的代码测试：

```
vector v { 5 };
iteratorTraitsTest(cbegin(v));
```

在这段代码中，`iteratorTraitsTest()` 中的变量 `temp` 的类型是 `int`。输出结果为 5。

当然，在本例中，可以使用 `auto` 关键字简化代码，但这样就没办法展示如何使用 `iterator_traits`。

17.1.3 示例

第一个示例简单地使用 `for` 循环和迭代器迭代 `vector` 中的每个元素，并将它们打印到标准输出。

```
vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for (auto iter { cbegin(values) }; iter != cend(values); ++iter) {
    cout << *iter << " ";
}
```

可能会尝试使用运算符<测试范围的结束，如 `iter<cend(values)`。然而，不建议这样做。测试范围结束的规范方法是使用`!=`，比如 `iter!=cend(values)`。原因是运算符`!=`适用于所有类型的迭代器，但是双向和向前迭代器并不支持`<`操作符。

可以实现一个辅助方法，它接受一个给定的元素范围作为 `begin` 和 `end` 迭代器，并将该范围内的所有元素打印到标准输出。

```
template <typename Iter>
void myPrint(Iter begin, Iter end)
{
    for (auto iter { begin }; iter != end; ++iter) { cout << *iter << " "; }
}
```

这个辅助函数可以按下面的方式使用：

```
myPrint(cbegin(values), cend(values));
```

第 2 个示例是 `myFind()` 函数模板，它在给定范围内查找给定值。如果没有找到该值，那么返回给定范围的 `end` 迭代器。注意 `value` 参数的特殊类型。它使用 `iterator_traits` 获取给定迭代器所指向的值的类型。

```
template <typename Iter>
auto myFind(Iter begin, Iter end,
const typename iterator_traits<Iter>::value_type& value)
{
    for (auto iter { begin }; iter != end; ++iter) {
        if (*iter == value) { return iter; }
    }
    return end;
}
```

这个函数模板可以按照如下方式使用：

```
vector values { 11, 22, 33, 44 };
auto result { myFind(cbegin(values), cend(values), 22) };
if (result != cend(values)) {
    cout << format("Found value at position {} ", distance(cbegin(values), result));
}
```

本章和后续章节提供了更多使用迭代器的示例。

17.2 流迭代器

标准库提供了 4 个流迭代器。这些都是类似迭代器的类模板，允许将输入和输出流视为输入和输出迭代器。使用这些流迭代器，可以调整输入流和输出流，以便它们可以分别作为各种标准库算法的源和目标。下面是可用的流迭代器：

- `ostream_iterator`：输出流迭代器
- `istream_iterator`：输入流迭代器

还有一个 `ostreambuf_iterator` 和 `istreambuf_iterator`，但它们很少使用，在本书中也没有进一步展开

讨论。

17.2.1 输出流迭代器

`ostream_iterator` 是一个输出流迭代器。它是一个接受元素类型作为模板类型参数的类模板。构造函数接受一个输出流和一个分隔符字符串，用于写入每个元素后面的流。`ostream_iterator` 类使用操作符`<<`写入元素。

示例如下。假设有以下 `myCopy()` 函数模板，它将 `begin` 和 `end` 迭代器给出的范围复制到 `begin` 迭代器给出的目标范围：

```
template <typename InputIter, typename OutputIter>
void myCopy(InputIter begin, InputIter end, OutputIter target)
{
    for (auto iter { begin }; iter != end; ++iter, ++target) { *target = *iter; }
}
```

使用 `myCopy()` 函数模板将一个 `vector` 的元素复制到另一个 `vector` 是很简单的。`myCopy()` 的前两个参数是要复制范围的 `begin` 和 `end` 迭代器，第三个参数指向目标范围的迭代器。必须确保目标范围足够大才行。

```
vector myVector { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
// Use myCopy() to copy myVector to vectorCopy.
vector<int> vectorCopy(myVector.size());
myCopy(cbegin(myVector), cend(myVector), begin(vectorCopy));
```

现在，通过使用 `ostream_iterator`，`myCopy()` 函数模板也可以只用一行代码就可以打印容器的元素。下面的代码片段打印 `myVector` 和 `vectorCopy` 的内容：

```
// Use the same myCopy() to print the contents of both vectors.
myCopy(cbegin(myVector), cend(myVector), ostream_iterator<int> { cout, " " });
cout << endl;
myCopy(cbegin(vectorCopy), cend(vectorCopy), ostream_iterator<int> { cout, " " });
cout << endl;
```

输出结果如下：

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

17.2.2 输入流迭代器

可以使用输入流迭代器 `istream_iterator`，通过迭代器抽象从输入流中读取值。它是一个接受元素类型作为模板类型参数的类模板。使用运算符`>>`读取元素。可以使用 `istream_iterator` 作为算法和容器方法的来源。

假设有以下 `sum()` 函数模板，它计算给定范围内所有元素的和：

```
template <typename InputIter>
auto sum(InputIter begin, InputIter end)
{
    auto sum { *begin };
    for (auto iter { ++begin }; iter != end; ++iter) { sum += *iter; }
    return sum;
}
```


现在, 可以使用 `istream_iterator` 从控制台读取整数, 直到到达流的末尾。在 Windows 上, 在按 Ctrl+Z 组合键后再按 Enter 键时, 就会发生这种情况, 而在 Linux 上按 Enter 键后再按 Ctrl+D 组合键就会发生这种情况。函数的作用是: 计算所有整数的和。注意, `istream_iterator` 的默认构造函数创建了一个 `end` 迭代器。

```
cout << "Enter numbers separated by whitespace." << endl;
cout << "Press Ctrl+Z followed by Enter to stop." << endl;
istream_iterator<int> numbersIter { cin };
istream_iterator<int> endIter;
int result { sum(numbersIter, endIter) };
cout << "Sum: " << result << endl;
```

花点时间思考这段代码。如果删除了所有输出语句和变量声明, 那么就只剩下对 `sum()` 的调用。多亏了迭代器和输入流迭代器, 这一行代码从控制台读取任意数量的整数, 并将它们相加, 而不使用任何显式的循环。

17.3 迭代器适配器

标准库提供了 5 个迭代器适配器(iterator adapter), 它们是特殊的迭代器, 都在 `<iterator>` 头文件中定义。它们被分成两组。第一组适配器从容器中创建, 通常作为输出迭代器。

- `back_insert_iterator`: 使用 `push_back()` 将元素插入容器中。
- `front_insert_iterator`: 使用 `push_front()` 将元素插入容器中。
- `insert_iterator`: 使用 `insert()` 将元素插入容器中。

最后两个适配器是通过另一个迭代器创建的, 不是容器, 通常作为输入迭代器。

- `reverse_iterator`: 反转另一个迭代器的迭代顺序。
- `move_iterator`: `move_iterator` 的解引用运算符自动将左值转换为右值引用, 因此可以将其移动到新目标。

也可以编写自定义的迭代器适配器, 但这不在本书讨论范围。请参阅附录 B 中列出的标准库中的参考资料。

17.3.1 插入迭代器

本章前面实现的 `myCopy()` 函数模板不将元素插入容器中: 它只是用新元素替换范围内的旧元素。为了使这些算法更有用, 标准库提供了 3 种插入迭代器适配器(insert iterator adapter), 它们实际上将元素插入容器中: `insert_iterator`、`back_insert_iterator` 和 `front_insert_iterator`。它们都是在容器类型上进行模板化, 并在其构造函数中接受实际的容器引用。因为它们提供了必要的迭代器接口, 所以可以将这些适配器用作 `myCopy()` 等算法的目标迭代器。但是, 它们并不是替换容器中的元素, 而是调用它们的容器来实际插入新元素。

基本的 `insert_iterator` 调用容器上的 `insert(position,element)`, `back_insert_iterator` 调用 `push_back(element)`, `front_insert_iterator` 调用 `push_front(element)`。

下面的示例使用带有 `myCopy()` 的 `back_insert_iterator`, 将 `vectorOne` 中所有元素的副本填充到 `vectorTwo`。注意, `vectorTwo` 不会首先调整大小以获得足够的元素, 这是因为插入迭代器负责正确插入新元素。

```
vector vectorOne { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
vector<int> vectorTwo;
```

```
back_insert_iterator<vector<int>> inserter { vectorTwo };
myCopy(cbegin(vectorOne), cend(vectorOne), inserter);

myCopy(cbegin(vectorTwo), cend(vectorTwo), ostream_iterator<int> { cout, " " });
```

正如所见，当使用插入迭代器时，不需要提前调整目标容器的大小。

还可以使用 `std::back_inserter()` 实用函数创建 `back_insert_iterator`。在前面的示例中，可以删除定义 `inserter` 变量的行，并按照下面的方式重写 `myCopy()` 调用。其结果与前面的实现相同：

```
myCopy(cbegin(vectorOne), cend(vectorOne), back_inserter(vectorTwo));
```

使用类模板参数推断(CTAD)，也可以这样写：

```
myCopy(cbegin(vectorOne), cend(vectorOne), back_insert_iterator { vectorTwo });
```

`front_insert_iterator` 和 `insert_iterator` 的工作方式类似，不同之处在于 `insert_iterator` 在其构造函数中也有一个初始迭代器的位置，它将该位置传递给对 `insert(position, element)` 的第一次调用。后续迭代器的位置提示是根据每次 `insert()` 调用的返回值生成的。

使用 `insert_iterator` 的一个好处是：它允许使用关联容器作为修改算法的目标。第 20 章解释了关联容器的问题在于，不允许修改要迭代的元素。通过使用 `insert_iterator`，可以改为插入元素。关联容器有一个 `insert()` 方法，该方法接受一个迭代器位置，并使用该位置作为“提示”，可以忽略这个“提示”。在关联容器上使用 `insert_iterator` 时，可以传递容器的 `begin()` 或 `end()` 迭代器作为提示。`insert_iterator` 在每次调用 `insert()` 之后修改传递给 `insert()` 的迭代器提示，以便位置是位于刚刚插入的元素的下一位置。

下面是前面修改后的示例，目标容器是 `set` 容器而不是 `vector` 容器：

```
vector vectorOne { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
set<int> setOne;

insert_iterator<set<int>> inserter { setOne, begin(setOne) };
myCopy(cbegin(vectorOne), cend(vectorOne), inserter);

myCopy(cbegin(setOne), cend(setOne), ostream_iterator<int> { cout, " " });
```

类似于 `back_insert_iterator` 的示例，可以使用 `std::inserter()` 实用函数创建 `insert_iterator`：

```
myCopy(cbegin(vectorOne), cend(vectorOne), inserter(setOne, begin(setOne)));
```

或者，使用类模板的实参推导：

```
myCopy(cbegin(vectorOne), cend(vectorOne),
insert_iterator { setOne, begin(setOne) });
```

17.3.2 逆向迭代器

标准库提供了一个 `std::reverse_iterator` 类模板，它以相反的方向遍历双向或随机访问迭代器。标准库中的每个可逆容器(恰好是标准库中除了 `forward_list` 和无序关联容器外的所有容器)都提供了一个 `reverse_iterator` 的类型别名、名为 `rbegin()` 和 `rend()` 的方法。这些 `reverse_iterator` 类型别名的类型是 `std::reverse_iterator<T>`，其中 `T` 等于容器的迭代器类型别名。方法 `rbegin()` 返回指向容器最后一个元素的 `reverse_iterator`，而 `rend()` 返回指向容器第一个元素之前的元素的 `reverse_iterator`。将 `operator++` 应用到 `reverse_iterator` 将调用底层容器迭代器上的 `operator--`，反之亦然。例如，可以按照如下方式从头到尾迭代一个集合：

```
for (auto iter { begin(collection) }; iter != end(collection); ++iter) {}
```

`std::reverse_iterator` 主要用于标准库中的算法或自定义函数中没有以相反顺序工作的等价物。本章前面介绍的 `myFind()` 函数将搜索序列中的第一个元素。如果想找到序列中的最后一个元素，可以使用 `reverse_iterator`。注意，当使用 `reverse_iterator` 调用诸如 `myFind()` 的算法时，它也会返回一个 `reverse_iterator`。总是可以通过调用其 `base()` 方法从 `reverse_iterator` 获取底层迭代器。然而，由于 `reverse_iterator` 的实现方式，从 `base()` 返回的迭代器总是指向一个元素，这个元素是 `reverse_iterator` 所指向的元素的前一个元素。为了得到相同的元素，必须减去 1。

下面是一个带有 `reverse_iterator` 的 `myFind()` 示例：

```
vector myVector { 11, 22, 33, 22, 11 };
auto it1 { myFind(begin(myVector), end(myVector), 22) };
auto it2 { myFind(rbegin(myVector), rend(myVector), 22) };
if (it1 != end(myVector) && it2 != rend(myVector)) {
    cout << format("Found at position {} going forward.",
                  distance(begin(myVector), it1)) << endl;
    cout << format("Found at position {} going backward.",
                  distance(begin(myVector), --it2.base())) << endl;
} else {
    cout << "Failed to find." << endl;
}
```

程序的输出结果如下：

```
Found at position 1 going forward.
Found at position 3 going backward.
```

17.3.3 移动迭代器

第 9 章讨论了移动语义，在源对象将在赋值操作或拷贝构造后被销毁的情况下，或者在使用 `std::move()` 时，可以使用移动语义防止不必要的复制。标准库提供了一个名为 `std::move_iterator` 的迭代器适配器。`move_iterator` 的解引用运算符自动将左值转换为右值引用，这意味着可以将左值移动到新的目标，而不必复制开销。在使用 `move` 语义之前，需要确保对象支持它。下面的 `MoveableClass` 支持移动语义。要了解更多细节，请参阅第 9 章。

```
class MoveableClass
{
public:
    MoveableClass() {
        cout << "Default constructor" << endl;
    }
    MoveableClass(const MoveableClass& src) {
        cout << "Copy constructor" << endl;
    }
    MoveableClass(MoveableClass&& src) noexcept {
        cout << "Move constructor" << endl;
    }
    MoveableClass& operator=(const MoveableClass& rhs) {
        cout << "Copy assignment operator" << endl;
        return *this;
    }
    MoveableClass& operator=(MoveableClass&& rhs) noexcept {
```


警告:

记住，一旦一个对象被移动到另一个对象，就不应该再使用它了。

17.4 范围

C++标准库的迭代器支持允许算法独立于实际容器工作，因为它们抽象了在容器元素中导航的机制。正如迄今为止在所有迭代器示例中所看到的，大多数算法都需要一个迭代器对，该迭代器对由指向序列中第一个元素的 `begin` 迭代器和指向序列中最后一个元素后一个元素的 `end` 迭代器组成。这使得算法可以在所有类型的容器上工作，但总是必须提供两个迭代器来指定元素序列并确保只提供匹配的迭代器，这有点麻烦。范围库提供的范围是迭代器之上的抽象层，消除了不匹配的迭代器错误，并添加了额外的功能，例如允许范围适配器延迟过滤和转换底层元素序列。在 `<ranges>` 头文件中定义的范围库由以下主要组件组成。

- **范围:** 范围是一个概念(请参阅第 12 章)，它定义了允许元素迭代的类型的要求。任何支持 `begin()` 和 `end()` 的数据结构都是有效的范围。例如，`std::array`、`vector`、`string_view`、`span`、固定大小的 C 风格数组等，这些都是有效的范围。
- **基于范围的算法:** 第 16 章和第 20 章讨论了接受迭代器对来执行工作的标准库算法。这些算法中的大多数都有接受范围而不是迭代器的等价物。
- **投影:** 很多基于范围的算法都接受所谓的投影回调。这个回调函数会为范围中的每个元素所调用，并且可以在元素传递给算法之前将其转换为其其他值。
- **视图:** 视图可以用来转换或过滤底层范围的元素。视图可以组合在一起，形成所谓的操作管道，以应用于一个范围。
- **工厂:** 范围工厂被用来构建一个按需生成值的视图。

可以使用迭代器对范围内的元素进行迭代，这些迭代器可以通过诸如 `ranges::begin()`、`end()`、`rbegin()` 等访问器进行检索。范围库还支持 `ranges::empty()`、`data()`、`cdata()` 和 `size()`。后者返回范围内的元素数量，但只有在可以在常量时间内检索大小时才有效。否则，可以使用 `std::distance()` 计算范围的 `begin` 迭代器和 `end` 迭代器之间的元素数目。所有这些访问器都不是成员函数，而是独立的自由函数，都需要一个范围作为参数。

17.4.1 基于范围的算法

`std::sort()` 算法就是一个示例，它需要指定一个元素序列作为 `begin` 迭代器和 `end` 迭代器。算法在第 16 章介绍，相关细节会在第 20 章进行详细讨论。`sort()` 算法使用起来很简单。例如，下面的代码对向量中的所有元素进行排序：

```
vector data { 33, 11, 22 };
sort(begin(data), end(data));
```

这段代码对数据容器中的所有元素进行排序，但必须将序列指定为 `begin/end` 迭代器对。在代码中更准确地描述真正想要做的事情不是更好吗？这就是范围库中基于范围的算法(范围库的一部分)的用武之地。这些算法位于 `std::ranges` 名称空间中，并与相应的非基于范围的版本在相同的头文件中定义。有了这些，可以简单地编写下面的示例：

```
ranges::sort(data);
```

这段代码清楚地描述了意图，即对数据容器中的所有元素进行排序。由于没有指定迭代器，这些

基于范围的算法消除了意外提供不匹配的 `begin` 和 `end` 迭代器的可能性。基于范围的算法对其模板类型参数有适当的类型约束(请参阅第 12 章)。这允许编译器提供更清晰的错误消息,以避免为基于范围的算法提供容器,而该算法不提供所需的迭代器类型。例如,在 `std::list` 上调用 `ranges::sort()` 算法将更清楚地说明 `sort()` 需要一个随机访问范围,但 `list` 不具备。与迭代器类似,有输入、输出、前向、双向、随机访问和连续范围,以及相应的概念,例如 `ranges::continuous_range`、`ranges::random_access_range` 等等。

注意:

大多数标准库算法已在第 16 章介绍,相关细节将会在第 20 章进行详细讨论,在 `std::ranges` 名称空间中都有基于范围的等价物。

投影

许多基于范围的算法都有一个所谓的投影参数,即一个回调函数,用于在将每个元素移交给算法之前对其进行转换。先来看一个示例。假设有一个简单的类表示一个人:

```
class Person
{
public:
    Person(string first, string last)
        : m_firstName { move(first) }, m_lastName { move(last) } { }
    const string& getFirstName() const { return m_firstName; }
    const string& getLastName() const { return m_lastName; }
private:
    string m_firstName;
    string m_lastName;
};
```

下面的代码在 `vector` 中存储了几个 `Person` 对象:

```
vector<Person> persons { Person {"John", "White"}, Person {"Chris", "Blue"} };
```

因为 `Person` 类没有实现 `operator<`, 所以不能使用普通的 `std::sort()` 算法对这个 `vector` 进行排序,因为它使用了 `operator<` 来比较元素。因此,下面的代码将无法编译:

```
sort(begin(persons), end(persons)); // Error: does not compile.
```

乍一看,切换到基于范围的 `ranges::sort()` 算法并没有多大帮助。下面的代码仍然无法编译,因为算法仍然不知道如何比较范围内的元素:

```
ranges::sort(persons); // Error: does not compile.
```

但是,如果希望根据人名进行排序,则可以为排序算法指定一个投影函数,将每个人投影到他们的名字上。投影参数是第 3 个参数,所以还必须指定第 2 个参数,即需要使用的比较器,默认情况下是 `std::ranges::less`。在下面的调用中,投影函数被指定为 `lambda` 表达式,请参阅后续的介绍。

```
ranges::sort(persons, {},
    [](const Person& person) { return person.getFirstName(); });
```

或者甚至更短的示例如下:

```
ranges::sort(persons, {}, &Person::getFirstName);
```


注意:

本章中关于范围的讨论使用了一些简单的 `lambda` 表达式。`lambda` 表达式将在第 19 章“函数指针、函数对象和 `lambda` 表达式”中详细讨论,但所有这些细节对于当前的讨论都不重要。现在,只需要知道基本的用法就足够了。本章中使用的 `lambda` 表达式具有以下语法:

```
[] (const Person& person) { return person.getFirstName(); }
```

`[]` 表示 `lambda` 表达式的开始。接下来是一个逗号分隔的参数列表,就像函数一样。最后, `lambda` 表达式的主体位于一组大括号之间。

基本上, `lambda` 表达式允许在需要的地方编写小的、未命名的内联函数。前面的 `lambda` 表达式可以用下面的独立函数替换:

```
auto getFirstName(const Person& person) {  
    return person.getFirstName();  
}
```

`lambda` 表达式参数的类型也可以是 `auto`。示例如下:

```
[] (const auto& person) { return person.getFirstName(); }
```

17.4.2 视图

视图允许对基础范围的元素执行操作。视图可以被链接/组合在一起,形成一个管道,对一个范围的元素执行多个操作。组合视图很容易,只需要使用按位 `OR` 运算符, `operator|` 组合不同的操作。例如,可以轻松地首先过滤范围中的元素,然后再转换其余的元素。相反,如果想做类似的事情,先过滤再转换,使用非基于范围的算法,代码可读性和性能可能会差很多,因为将不得不创建临时容器来存储中间结果。

视图具有以下重要属性:

- 惰性求值——仅仅构建一个视图还不能对底层范围执行任何操作。视图的操作仅在迭代视图的元素和解引用这样的迭代器时应用。
- 非占有——视图不拥有任何元素。顾名思义,它是一个可以存储在某个容器中的范围元素的视图,并且该容器是数据的所有者。视图只允许以不同的方式查看数据。因此,视图中元素的数量不会影响复制、移动或销毁视图的成本。这类似于第 2 章讨论的 `std::string_view` 和第 18 章讨论的 `std::span`。
- 非变异——视图永远不会修改底层范围中的数据。

视图本身也是一个范围,但并非每个范围都是一个视图。容器是一个范围而不是一个视图,因为它拥有元素。

可以使用范围适配器创建视图。范围适配器接收一个范围(可以是另一个视图)和一些可选参数,并返回一个新视图。表 17-3 列出了标准库提供的最重要的范围适配器。

表 17-3 范围适配器

范围适配器	描述
<code>views::all</code>	创建一个包含范围内所有元素的视图
<code>filter_view</code> <code>views::filter</code>	根据给定谓词过滤范围中的元素。如果谓词返回 <code>true</code> , 则保留该元素, 否则跳过该元素
<code>transform_view</code> <code>views::transform</code>	对范围中的每个元素应用回调, 以便将元素转换为其他值(可能是不同类型的值)

(续表)

范围适配器	描述
take_view views::take	创建范围的前 <i>N</i> 个元素的视图
take_while_view views::take_while	创建范围的初始元素的视图，直到到达给定谓词返回 <code>false</code> 的元素为止
drop_view views::drop	通过删除范围的前 <i>N</i> 个元素创建视图
drop_while_view views::drop_while	通过删除范围的所有初始元素创建视图，直到到达给定谓词返回 <code>false</code> 的元素
reverse_view views::reverse	创建一个视图，该视图以相反的顺序迭代范围中的元素。这个范围必须是双向范围
elements_view views::elements	需要一组类似 <code>tuple</code> 的元素，创建一个类似 <code>tuple</code> 元素的第 <i>N</i> 个元素的视图
keys_view views::keys	需要一组类似 <code>pair</code> 的元素，创建每个 <code>pair</code> 的第 1 个元素的视图
values_view views::values	需要一组类似 <code>pair</code> 的元素，创建每个 <code>pair</code> 的第 2 个元素的视图
common_view views::common	根据范围的类型， <code>begin()</code> 和 <code>end()</code> 可能会返回不同的类型，例如 <code>begin</code> 迭代器和所谓的哨兵。这意味着，不能将这样的迭代器对传递给期望它们具有相同类型的函数。 <code>common_view</code> 可用于将这样一个范围转换为一个普通范围，即 <code>begin()</code> 和 <code>end()</code> 返回相同类型的范围。在其中一个练习中，将会使用这个范围适配器

表 17-3 中第 1 列中的范围适配器既显示了 `std::ranges` 名称空间中的类名，也显示了来自 `std::ranges::views` 名称空间中的所谓范围适配器对象(range adapter object)。标准库还提供了—个名为 `std::views` 的名称空间别名，它等于 `std::ranges::views`。

每个范围适配器都可以通过调用其构造函数并传递任何必需的参数来构造。第 1 个参数总是要操作的范围，后面跟着 0 个或多个附加参数，如下所示：

```
std::ranges::operation_view { range, arguments... }
```

通常，不会使用它们的构造函数创建这些范围适配器，而是使用 `std::ranges::views` 名称空间中的范围适配器对象，并结合按位 OR 运算符，如下所示：

```
range | std::ranges::views::operation(arguments...)
```

一起来看看这些范围适配器的运行情况。下面的示例首先定义了一个名为 `printRange()`的简化函数模板，用于打印一条消息，后面跟着给定范围内的所有元素。接下来，`main()`函数首先创建一个整数向量，1……10，然后对它应用多个范围适配器，每次对结果调用 `printRange()`，以便可以跟踪正在发生的事情。

```
void printRange(string_view message, auto& range)
{
    cout << message;
    for (const auto& value : range) { cout << value << " "; }
    cout << endl;
```

```

}

int main()
{
    vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    printRange("Original sequence: ", values);

    // Filter out all odd values, leaving only the even values.
    auto result1 { values
        | views::filter([](const auto& value) { return value % 2 == 0; }) };
    printRange("Only even values: ", result1);

    // Transform all values to their double value.
    auto result2 { result1
        | views::transform([](const auto& value) { return value * 2.0; }) };
    printRange("Values doubled: ", result2);

    // Drop the first 2 elements.
    auto result3 { result2 | views::drop(2) };
    printRange("First two dropped: ", result3);

    // Reverse the view.
    auto result4 { result3 | views::reverse };
    printRange("Sequence reversed: ", result4);
}

```

程序的输出结果如下：

```

Original sequence: 1 2 3 4 5 6 7 8 9 10
Only even values: 2 4 6 8 10
Values doubled: 4 8 12 16 20
First two dropped: 12 16 20
Sequence reversed: 20 16 12

```

值得重申的是，视图是惰性求值的。在本例中，**result1** 视图的构建没有执行任何实际的过滤。过滤发生在 **printRange()** 函数迭代 **result1** 的元素时。

代码片段使用 **std::ranges::views** 中的范围适配器对象。还可以使用范围适配器的构造函数来构造范围适配器。例如，**result1** 视图可以构造如下：

```

auto result1 { ranges::filter_view { values,
    [](const auto& value) { return value % 2 == 0; } } };

```

这个示例创建了几个中间视图，以便能够输出它们的元素，这样可以更容易地了解每个步骤中发生的事情。如果不需要这些中间视图，可以将它们全部链接在一个管道中，如下所示：

```

vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
printRange("Original sequence: ", values);

auto result { values
    | views::filter([](const auto& value) { return value % 2 == 0; })
    | views::transform([](const auto& value) { return value * 2.0; })
    | views::drop(2)
    | views::reverse };
printRange("Final sequence: ", result);

```

输出结果如下所示。最后一行显示了最终序列与前面的 `result4` 视图相同。

```
Original sequence: 1 2 3 4 5 6 7 8 9 10
Final sequence: 20 16 12
```

通过视图修改元素

只要该范围不是只读的，就可以修改范围的元素。例如，`views::transform` 的结果是一个只读视图，因为它返回一个包含已转换元素的视图，但不转换底层范围中的实际值。

下面的示例再次构造了一个包含 10 个元素的 `vector`。然后，它在偶数上创建了一个视图，删除了前 2 个偶数，最后再反转元素。然后，基于范围的 `for` 循环将结果视图中的元素乘以 10。最后一行输出 `values vector` 中的元素，以确认某些元素已通过视图更改。

```
vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
printRange("Original sequence: ", values);

// Filter out all odd values, leaving only the even values.
auto result1 { values
    | views::filter([](const auto& value) { return value % 2 == 0; }) };
printRange("Only even values: ", result1);

// Drop the first 2 elements.
auto result2 { result1 | views::drop(2) };
printRange("First two dropped: ", result2);

// Reverse the view.
auto result3 { result2 | views::reverse };
printRange("Sequence reversed: ", result3);

// Modify the elements using a range-based for loop.
for (auto& value : result3) { value *= 10; }
printRange("After modifying elements through a view, vector contains: ", values);
```

输出结果如下：

```
Original sequence: 1 2 3 4 5 6 7 8 9 10
Only even values: 2 4 6 8 10
First two dropped: 6 8 10
Sequence reversed: 10 8 6
After modifying elements through a view, vector contains: 1 2 3 4 5 60 7 80 9 100
```

映射元素

转换范围的元素并不需要产生具有相同类型元素的范围。相反，可以将元素映射到另一种类型。下面的例子从一个整数范围开始，过滤掉所有的奇数，只保留前 3 个偶数，并使用 `std::format()` 和一个原始字符串字面量将它们转换为带引号的字符串(请参阅第 2 章)：

```
vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
printRange("Original sequence: ", values);

auto result { values
    | views::filter([](const auto& value) { return value % 2 == 0; })
    | views::take(3)
    | views::transform([](const auto& v) { return format(R"({})", v); }) };
printRange("Result: ", result);
```

输出结果如下：

```
Original sequence: 1 2 3 4 5 6 7 8 9 10
Result: "2" "4" "6"
```

17.4.3 范围工厂

范围库提供了表 17-4 的范围工厂(range factory)来构建视图，这些视图可以根据需要惰性地生成元素。

表 17-4 范围工厂

范围工厂	描述
empty_view	创建一个空视图
single_view	创建具有单个给定元素的视图
iota_view	创建一个无限或有界的视图，其中包含以初始值开始的元素，每个后续元素的值等于前一个元素的值加 1
basic_istream_view istream_view	创建一个视图，其中包含通过调用底层输入流上的调用提取运算符(运算符>>)检索到的元素

与上一节的范围适配器一样，范围工厂表中的名称是位于 std::ranges 名称空间中的类名称，可以使用它们的构造函数直接创建它们。或者，也可以使用 std::ranges::views 名称空间中提供的工厂函数。例如，下面的两个语句是等价的，它们用元素 10、11、12……创建了一个无限视图。

```
std::ranges::iota_view { 10 }
std::ranges::views::iota(10)
```

来看一个实际中的范围工厂。下面的示例基于前面的示例，但这段代码并没有构造一个包含 10 个元素的 vector，而是使用 iota 范围工厂创建了一个从 10 开始的惰性无限数字序列。然后它删除了所有奇数，将其余元素加倍，最后只保留前 10 个元素，随后使用 printRange()输出到控制台。

```
// Create an infinite sequence of the numbers 10, 11, 12, ...
auto values { views::iota(10) };
// Filter out all odd values, leaving only the even values.
auto result1 { values
    | views::filter([](const auto& value) { return value % 2 == 0; }) };
// Transform all values to their double value.
auto result2 { result1
    | views::transform([](const auto& value) { return value * 2.0; }) };
// Take only the first ten elements.
auto result3 { result2 | views::take(10) };
printRange("Result: ", result3);
```

输出结果如下：

```
Result: 20 24 28 32 36 40 44 48 52 56
```

值范围表示无限的范围，随后对其进行过滤和转换。使用无限范围是可能的，因为所有这些操作只有在 printRange()迭代视图中的元素时才会被延迟计算。这也意味着在这个示例中，不能调用 printRange()来输出 values、result1 或 result2 的内容，因为这将导致 printRange()中的 for 循环永远循环，因为它们是无范围。

当然，也可以去掉这些中间视图，简单地构建一个大的管道。

```
auto result { views::iota(10)
    | views::filter([](const auto& value) { return value % 2 == 0; })
    | views::transform([](const auto& value) { return value * 2.0; })
    | views::take(10) };
printRange("Result: ", result);
```

输入流作为视图

`basic_istream_view/istream_view` 范围工厂可用于构建从输入流(例如标准输入)读取的元素的视图。使用运算符`>>`读取元素。

例如，下面的代码片段继续从标准输入读取整数。对于每个小于 5 的读取数字，将该数字加倍，然后打印在标准输出中。一旦输入 5 或者更大的数字，循环停止。

```
for (auto value : ranges::istream_view<int>(cin)
    | views::take_while([](const auto& v) { return v < 5; })
    | views::transform([](const auto& v) { return v * 2; }))) {
    cout << format("> {}\\n", value);
}
cout << "Terminating..." << endl;
```

以下是可能的输出序列：

```
1
> 2
3
> 6
4
> 8
5
Terminating...
```

17.5 本章小结

本章解释了迭代器背后的思想，迭代器是一种抽象概念，允许在不需要了解容器结构的情况下导航容器的元素。输出流迭代器可以使用标准输出作为基于迭代器的算法的目标，类似地，输入流迭代器可以使用标准输入作为算法的数据来源。本章还讨论了可用于适配其他迭代器的插入、反向和移动迭代器适配器。

本章的最后一部分讨论了范围库，它是 C++20 标准库的一部分。通过指定想要完成什么，而不是如何完成，它允许编写更多的函数式代码。它可以构造由应用于范围元素的操作组合组成的管道。这样的管道是惰性求值的；也就是说，在迭代结果视图之前，它们不会执行任何操作。

17.6 练习

通过完成以下习题，可以巩固本章讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而，如果你困在一个练习中，在网站上查看答案之前，请先重新阅读本章的内容，试着自己找到答案。

练习 17-1 编写一个程序，惰性构造 10~100 的元素序列，然后对每个数进行平方计算，再删除所有可被 5 整除的数，最后使用 `std::to_string()` 将剩余的数转换为字符串。

练习 17-2 编写一个程序，创建一个 `pair` 的 `vector`，其中每一对包含本章前面介绍的 `Person` 类的一个实例，以及他们的年龄。接下来，使用范围库构建一个单一的管道，从向量中提取所有人的年龄，并删除所有年龄在 12 岁以下和 65 岁以上的人。最后，使用本章前面的 `sum()` 算法计算剩余人的年龄平均值。当将一个范围传递给 `sum()` 算法时，必须使用一个普通范围。

练习 17-3 进一步构建练习 17-2 的解决方案，为 `Person` 类添加 `operator<<` 的实现。

接下来，创建一个管道，从 `pair` 的 `vector` 中提取每对的 `Person`，并且只保留前 4 个 `Person`。使用本章前面介绍的 `myCopy()` 算法将这 4 个人的名字打印到标准输出；每行显示一个名称。

最后，创建一个类似的管道，并将所有过滤的 `Person` 投影到他们的姓氏。再次使用 `myCopy()` 将这些姓氏打印到标准输出。

练习 17-4 编写一个程序，使用基于范围的 `for` 循环和 `range::istream_view()` 从标准输入读取整数，直到输入 -1。将读取的整数存储到一个 `vector` 中，并使用本章前面介绍的 `myCopy()` 算法来验证向量是否包含正确的值。

附加练习 能否找到一些方法来改变练习 17-4 的解决方案，避免使用任何循环？提示：也许一种选择是使用 `std::ranges::copy()` 算法将范围从源复制到目标。它可以用一个范围作为第 1 个参数，输出迭代器作为第 2 个参数来调用。

第 18 章

标准库容器

本章内容

- 容器概述：元素的需求、一般错误处理
- 顺序容器：vector、deque、list、forward_list 和 array
- 如何使用 span 类
- 容器适配器：queue、priority_queue 和 stack
- 关联容器：pair 实用工具、map、multimap、set 和 multiset
- 无序关联容器/哈希表：unordered_map、unordered_multimap、unordered_set 和 unordered_multiset
- 其他容器：标准 C 风格数组、string、流和 bitset

本章深入讲解标准库可用的容器。介绍了不同的容器、它们的类别以及它们之间的权衡。相比其他一些容器，对某些容器的讨论要详细得多。一旦知道如何使用每个类别的容器，那么使用同一类别的任何其他容器就都不会有问题。有关所有容器的所有方法的完整参考资源，请参阅最喜欢的标准库参考。

18.1 容器概述

标准库中的容器是泛型数据结构，特别适合保存数据集合。使用标准库时，几乎不需要使用标准 C 风格数组、编写链表或者设计栈。容器被实现为类模板，因此可利用任何满足以下基本条件的类型进行实例化。除 array 和 bitset 外，大部分标准库容器的大小灵活多变，都能自动增长或收缩，以容纳更多或更少的元素。和固定大小的旧的标准 C 风格数组相比，这有着巨大优势。由于本质上标准 C 风格数组的大小是固定的，因此容易受到溢出的破坏。如果数据溢出，轻则导致程序崩溃(因为数据被破坏了)，重则导致某些类型的安全攻击。使用标准库容器，程序遇到这种问题的可能性就会小得多。

第 16 章“C++标准库概述”对标准库提供的不同容器进行了高级概述。标准库提供了 16 个容器，分为 4 大类。

- 顺序容器
 - vector(动态数组)

- deque
- list
- forward_list
- array
- 关联容器
 - map
 - multimap
 - set
 - multiset
- 无序关联容器或哈希表
 - unordered_map
 - unordered_multimap
 - unordered_set
 - unordered_multiset
- 容器适配器
 - queue
 - priority_queue
 - stack

此外, C++的 `string` 和流也可在某种程度上用作标准库容器, `bitset` 可以用于存储固定数目的位。

标准库中的所有内容都在 `std` 名称空间中。本书中的例子通常都在源文件中使用 `using namespace std`; 语句覆盖所有范围(千万不要在头文件中使用!), 也可以在自己的程序中更有选择性地选择使用 `std` 中的哪些符号。

18.1.1 对元素的要求

标准库容器对元素使用值语义(value semantic)。也就是说, 在输入元素时保存元素的一份副本, 通过赋值运算符给元素赋值, 通过析构函数销毁元素。因此, 编写要用于标准库的类时, 一定要保证它们是可以复制的。请求容器中的元素时, 会返回所存副本的引用。

如果喜欢引用语义, 可存储元素的指针而非元素本身。当容器复制指针时, 结果仍然指向同一元素。另一种方式是在容器中存储 `std::reference_wrapper`。可使用 `std::ref()` 或 `std::cref()` 创建 `reference_wrapper`, 使引用变得可以复制。`reference_wrapper` 类模板以及 `ref()` 和 `cref()` 函数模板在 `<functional>` 头文件中定义。在本章后面的“在 `vector` 中存储引用”一节中给出了一个示例。

在容器中, 可存储“仅移动”类型, 这是非可复制类型, 但这么做时, 容器上的一些操作可能无法通过编译。“仅移动”类型的一个例子是 `std::unique_ptr`。

警告:

如果要在容器中保存指针, 应该使用 `unique_ptr`, 使容器成为指针所指对象的拥有者, 或者使用 `shared_ptr`, 使容器与其他拥有者共享拥有权。不要在容器中使用旧的和已删除的 `auto_ptr` 类, 因为这个类没有正确实现复制操作(就标准库而言)。

标准库容器的一个模板类型参数是所谓的分配器(allocator)。标准库容器可使用分配器为元素分配或释放内存。分配器类型参数具有默认值, 因此几乎总是可以忽略它。例如, `vector` 类模板如下所示:

```
template <class T, class Allocator = std::allocator<T>> class vector;
```

某些容器(例如 map)也允许将比较器(comparator)作为模板类型参数。比较器用于排序元素,具有默认值,通常不需要指定。例如, map 类模板如下所示:

```
template <class Key, class T, class Compare = std::less<Key>,  
        class Allocator = std::allocator<std::pair<const Key, T>>> class map;
```

有关使用默认内存分配器和比较器的容器中元素的特别需求在表 18-1 中列出。

表 18-1 特别需求

方法	说明	注意
拷贝构造 函数	创建一个“等于”旧元素的新元素,但这个新元素可以安全地析构,而不会影响旧元素	每次插入元素时使用,但使用稍后介绍的 emplace 方法时除外
移动构造 函数	创建一个新元素,将源元素中的所有内容转移到新元素中	当源元素是右值,并且在构建新元素后要销毁源元素时使用;或者在 vector 增加其容量时使用。移动构造函数应当标记 noexcept,否则无法使用
赋值运算符	用源元素的副本替换一个元素的内容	每次修改元素时使用
移动赋值 运算符	通过移动源元素的所有内容替换一个元素的内容	当源元素是右值,并且在执行赋值操作后要销毁源元素时使用。移动赋值运算符应当标记 noexcept,否则无法使用
析构函数	清理一个元素	每次删除元素时使用;或者 vector 增加其容量,并且元素不能 noexcept 移动时使用
默认构造 函数	构建一个元素时不接收任何参数	只有特定的操作才需要,例如带一个参数的 vector::resize()方法和 map::operator[]访问
operator==	比较两个元素是否相等	无序容器中的键需要;还有在执行特定操作时需要,例如对两个容器应用 operator==时
operator<	判断一个元素是否比另一个元素小	关联容器中的键需要,还有在执行某些操作时需要,例如对两个容器应用 operator<时

第 9 章“精通类与对象”讲解了如何编写这些方法。

警告:

标准库容器经常移动或复制元素。因此,为了获得最佳性能,请确保容器中存储的对象类型支持移动语义,详情请参阅第 9 章。如果无法使用移动语义,请确保拷贝构造函数和拷贝赋值运算符尽可能高效。

18.1.2 异常和错误检查

标准库容器提供非常有限的错误检查功能。客户应确保使用正确。然而,一些容器方法和函数会在特定条件下抛出异常,例如越界索引。当然,不可能全面包罗这些方法抛出的异常,因为这些方法操作的用户自定义类型没有已知的异常特征。本章在恰当的地方提到了异常。可参阅标准库资料,以便了解每个方法可能抛出的异常列表。

18.2 顺序容器

`vector`、`deque`、`list`、`forward_list` 和 `array` 都称为顺序容器(sequential container)，因为它们存储了一个元素序列。学习顺序容器的最好方法是学习一个 `vector` 示例，`vector` 是默认容器。本节首先详细描述 `vector`，然后简要描述 `deque`、`list`、`forward_list` 和 `array`。熟悉了顺序容器后，就可以很方便地在其中进行切换。

18.2.1 vector

标准库容器 `vector` 类似于标准 C 风格数组：元素保存在连续的内存空间中，每个元素都有自己的“槽”。可以在 `vector` 中建立索引，还可以在尾部或任何位置插入新的元素。向 `vector` 插入元素或从 `vector` 删除元素通常需要线性时间，当这些操作在 `vector` 尾部执行时，实际运行时间为摊还常量时间。随机访问单个元素的复杂度为常量时间。

1. vector 概述

`vector` 在 `<vector>` 头文件中被定义为一个带有 2 个类型参数的类模板：一个参数为要保存的元素类型，另一个参数为分配器(allocator)类型。

```
template <class T, class Allocator = allocator<T>> class vector;
```

`Allocator` 参数指定了内存分配器对象的类型，客户可设置内存分配器，以便使用自定义的内存分配器。这个模板参数具有默认值。

注意：

`Allocator` 类型参数的默认值足够大部分应用程序使用。本章假定总是使用默认分配器。第 25 章“自定义和扩展标准库”将提供更多感兴趣的细节。

从 C++20 开始，`std::vector` 是 `constexpr`，就像 `std::string` 一样。这意味着 `vector` 可用于在编译期执行操作，并可用于 `constexpr` 函数和类的实现。目前，`vector` 和 `string` 是唯一的 `constexpr` 的标准库容器。

固定长度的 vector

使用 `vector` 的最简单方式是将其用作固定长度的数组。`vector` 提供了一个可以指定元素数量的构造函数，还提供了一个重载的 `operator[]` 以便访问和修改这些元素。通过 `operator[]` 访问 `vector` 边界之外的元素时，得到的结果是未定义的。也就是说，编译器可以自行决定如何处理这种情况。例如，Microsoft Visual C++ 的默认行为是：在调试模式下编译程序时，会给出运行时错误消息；在发布模式下，出于性能原因这些边界检查都被禁用了。可以修改这些默认行为。

警告：

与真正的数组索引一样，`vector` 上的 `operator[]` 没有提供边界检查功能。

除使用 `operator[]` 运算符外，还可通过 `at()`、`front()` 和 `back()` 访问 `vector` 中的元素。`at()` 方法等同于 `operator[]` 运算符，区别在于 `at()` 会执行边界检查，如果索引超出边界，`at()` 会抛出 `out_of_range` 异常。`front()` 和 `back()` 分别返回 `vector` 的第 1 个元素和最后 1 个元素的引用。在空的容器上调用 `front()` 和 `back()` 会引发未定义的行为。

警告:

所有 `vector` 元素访问操作的复杂度都是常量时间。

下面是一个用于“标准化”考试分数的小程序，经过标准化后，最高分设置为 100，其他所有分数都依此进行调整。这个程序创建了一个带有 10 个 `double` 值的 `vector`，然后从用户那里读入 10 个值，将每个值除以最高分(再乘以 100)，最后打印出新值。为简单起见，这个程序略去了错误检查部分。

```
vector<double> doubleVector(10); // Create a vector of 10 doubles

// Initialize max to smallest number
double max { -numeric_limits<double>::infinity() };

for (size_t i { 0 }; i < doubleVector.size(); i++) {
    cout << format("Enter score {}: ", i + 1);
    cin >> doubleVector[i];
    if (doubleVector[i] > max) {
        max = doubleVector[i];
    }
}

max /= 100.0;
for (auto& element : doubleVector) {
    element /= max;
    cout << element << " ";
}
```

从这个例子可以看出，可以像使用标准 C 风格数组一样使用 `vector`。注意第一个 `for` 循环使用 `size()` 方法来确定容器中的元素个数。本例还演示了如何给 `vector` 使用基于范围的 `for` 循环。在本例中，基于范围的 `for` 循环使用的是 `auto&` 而不是 `auto`，因为这里需要一个引用，才能在每次迭代时修改元素。

注意:

对 `vector` 应用 `operator[]` 运算符通常会返回一个对元素的引用，可将这个引用放在赋值语句的左侧。如果对 `const vector` 对象应用 `operator[]` 运算符，就会返回一个对 `const` 元素的引用，这个引用不能用作赋值的目标。第 15 章“C++运算符重载”将详细讲解这个技巧的实现细节。

动态长度的 vector

`vector` 的真正强大之处在于动态增长的能力。例如，考虑前面的测试分数标准化程序，对这个程序再添加一项要求：处理任意数量的测试分数。下面是这个程序的新版本：

```
vector<double> doubleVector; // Create a vector with zero elements.

// Initialize max to smallest number
double max { -numeric_limits<double>::infinity() };

for (size_t i { 1 }; true; i++) {
    double temp;
    cout << format("Enter score {} (-1 to stop): ", i);
    cin >> temp;
    if (temp == -1) {
        break;
    }
    doubleVector.push_back(temp);
}
```

```

        if (temp > max) {
            max = temp;
        }
    }

    max /= 100.0;
    for (auto& element : doubleVector) {
        element /= max;
        cout << element << " ";
    }
}

```

这个新版本的程序使用默认的构造函数创建了一个不包含元素的 `vector`。每读取一个分数，便通过 `push_back()` 方法将分数添加到 `vector` 中，`push_back()` 方法能为新元素分配范围。基于范围的 `for` 循环不需要任何修改。

2. vector 详解

前面初步介绍了 `vector`，下面将深入讲解 `vector` 的细节。

构造函数和析构函数

默认的构造函数创建一个不包含元素的 `vector`。

```
vector<int> intVector; // Creates a vector of ints with zero elements
```

可指定元素个数，还可指定这些元素的值，如下所示：

```
vector<int> intVector(10, 100); // Creates vector of 10 ints with value 100
```

如果没有提供默认值，那么对新对象进行 0 初始化。0 初始化通过默认构造函数构建对象，将基本的整数类型(例如 `char` 和 `int` 等)初始化为 0，将基本的浮点类型初始化为 0.0，将指针类型初始化为 `nullptr`。

还可创建内建类的 `vector`，如下所示：

```
vector<string> stringVector(10, "hello");
```

用户自定义的类也可以用作 `vector` 元素：

```

class Element
{
public:
    Element() {}
    virtual ~Element() = default;
};
...
vector<Element> elementVector;

```

可以使用包含初始元素的 `initializer_list` 构建 `vector`：

```
vector<int> intVector({ 1, 2, 3, 4, 5, 6 });
```

第 1 章提到的统一初始化(uniform initialization)可用于包括 `vector` 在内的大部分标准库容器。例如：

```

vector<int> intVector1 = { 1, 2, 3, 4, 5, 6 };
vector<int> intVector2{ 1, 2, 3, 4, 5, 6 };

```

多亏了类模板实参推导(CTAD)，可以省略模板类型参数。示例如下：

```
vector<int> intVector { 1, 2, 3, 4, 5, 6 };
```


不过,请谨慎使用统一初始化:通常,在调用对象的构造函数时,可以使用统一的初始化语法。示例如下:

```
string text { "Hello World." };
```

对于 `vector` 的使用也需要小心。例如,下面的代码调用 `vector` 构造函数来创建一个包含 10 个整数,且值为 100 的 `vector`:

```
vector<int> intVector(10, 100); // Creates vector of 10 ints with value 100
```

如下使用统一初始化不是创建一个包含 10 个整数的 `vector`,而是一个只有 2 个元素的 `vector`,初始化为 10 和 100:

```
vector<int> intVector { 10, 100 }; // Creates vector of two elements: 10 and 100
```

还可以在自由存储区上分配 `vector`:

```
auto elementVector { make_unique<vector<Element>>(10) };
```

vector 的复制和赋值

`vector` 存储对象的副本,其析构函数调用每个对象的析构函数。`vector` 类的拷贝构造函数和赋值运算符对 `vector` 中的所有元素执行深度复制。因此,出于效率方面的考虑,应该通过非 `const` 引用或 `const` 引用而不是通过传值向函数传递 `vector`。

除普通的复制和赋值外, `vector` 还提供了 `assign()` 方法,这个方法删除了所有现有的元素,并添加任意数目的新元素。这个方法特别适合于 `vector` 的重用。下面是一个简单的例子。`intVector` 包含 10 个默认值为 0 的元素。然后通过 `assign()` 删除所有 10 个元素,并填充以 5 个值为 100 的元素代之。

```
vector<int> intVector(10);
...
intVector.assign(5, 100);
```

如下所示, `assign()` 还可接收 `initializer_list`。`intVector` 现在有 4 个具有给定值的元素:

```
intVector.assign({ 1, 2, 3, 4 });
```

`vector` 还提供了 `swap()` 方法,这个方法可交换两个 `vector` 的内容,并且具有常量时间复杂度。下面是一个简单的示例:

```
vector<int> vectorOne(10);
vector<int> vectorTwo(5, 100);
vectorOne.swap(vectorTwo);
// vectorOne now has 5 elements with the value 100.
// vectorTwo now has 10 elements with the value 0.
```

vector 的比较

标准库在 `vector` 中提供了 6 个重载的比较运算符: `==`、`!=`、`<`、`>`、`<=` 和 `>=`。如果两个 `vector` 的元素数量相等,而且对应元素都相等,那么这两个 `vector` 相等。两个 `vector` 的比较采用字典顺序:如果第一个 `vector` 中从 0 到 $i-1$ 的所有元素都等于第二个 `vector` 中从 0 到 $i-1$ 的所有元素,但第一个 `vector` 中的元素 i 小于第二个 `vector` 中的元素 i ,其中 i 在 0 到 n 之间,且 n 必须小于 `size()`,那么第一个 `vector` “小于”第二个 `vector`;其中的 `size()` 是指两个 `vector` 中较小者的 size。

注意:

通过 `operator==` 和 `operator!=` 比较两个 `vector` 时, 要求每个元素都能通过 `operator==` 运算符进行比较。通过 `operator<`、`operator>`、`operator<=` 或 `operator>=` 比较两个 `vector` 时, 要求每个元素都能通过 `operator<` 运算符进行比较。如果要在 `vector` 中保存自定义类的对象, 务必编写这些运算符。

下面是一个比较元素类型为 `int` 的两个 `vector` 的简单程序:

```
vector<int> vectorOne(10);
vector<int> vectorTwo(10);

if (vectorOne == vectorTwo) {
    cout << "equal!" << endl;
} else {
    cout << "not equal!" << endl;
}

vectorOne[3] = 50;

if (vectorOne < vectorTwo) {
    cout << "vectorOne is less than vectorTwo" << endl;
} else {
    cout << "vectorOne is not less than vectorTwo" << endl;
}
```

这个程序的输出为:

```
equal!
vectorOne is not less than vectorTwo
```

vector 迭代器

第 17 章“理解迭代器与范围库”讲解了容器迭代器的基础知识。该讨论比较抽象, 因此看一下代码示例会有所帮助。下面是之前使用迭代器替换基于范围的 `for` 循环的测试分数标准化程序的最后一个 `for` 循环:

```
for (vector<double>::iterator iter { begin(doubleVector) };
     iter != end(doubleVector); ++iter) {
    *iter /= max;
    cout << *iter << " ";
}
```

首先, 看一下 `for` 循环的初始化语句:

```
vector<double>::iterator iter { begin(doubleVector) };
```

前面提到, 每个容器都定义了一种名为 `iterator` 的类型, 以表示那种容器类型的迭代器。`begin()` 返回引用容器中第一个元素的相应类型的迭代器。因此, 这条初始化语句在 `iter` 变量中获取了引用 `doubleVector` 中第一个元素的迭代器。下面看一下 `for` 循环的比较语句:

```
iter != end(doubleVector);
```

这条语句检查迭代器是否超越了 `vector` 中元素序列的尾部。当到达这一点时, 循环终止。递增语句 `++iter` 递增迭代器, 以便引用 `vector` 中的下一个元素。

注意：

只要可能，尽量使用前递增而不要使用后递增，因为前递增至少效率不会差，一般更高效。`iter++` 必须返回一个新的迭代器对象，而 `++iter` 只是返回对 `iter` 的引用。关于 `operator++` 运算符的实现详情，请参阅第 15 章“C++运算符重载”。

for 循环体包含以下两行：

```
*iter /= max;
cout << *iter << " ";
```

从中可以看出，这段代码可以访问和修改所迭代的元素。第一行通过*解引用 `iter`，从而获得 `iter` 所引用的元素，然后给这个元素赋值。第二行再次解引用 `iter`，但这次只是将元素流式输出到 `cout`。

上述使用迭代器的 for 循环可通过 `auto` 关键字进行简化：

```
for (auto iter { begin(doubleVector) };
     iter != end(doubleVector); ++iter) {
    *iter /= max;
    cout << *iter << " ";
}
```

有了 `auto`，编译器会根据初始化语句右侧的内容自动推导变量 `iter` 的类型，在本例中，初始化语句右侧的内容是调用 `begin()` 得到的结果。

`vector` 支持以下功能：

- `(c)begin()`和`(c)end()`返回指向第一个元素和最后一个元素的后一个元素的(const)迭代器。
- `(c)rbegin()`和`(c)rend()`返回指向第一个元素的前一个元素和最后一个元素的反向迭代器。

访问对象元素中的字段

如果容器中的元素是对象，那么可对迭代器使用->运算符，调用对象的方法或访问对象的成员。例如，下面的程序创建了包含 10 个字符串的 `vector`，然后迭代所有字符串，并给每个字符串追加一个新的字符串：

```
vector<string> stringVector(10, "hello");
for (auto it { begin(stringVector) }; it != end(stringVector); ++it) {
    it->append(" there");
}
```

使用基于范围的 for 循环，这段代码可以重写为：

```
for (auto& str : stringVector) {
    str.append(" there");
}
```

const_iterator

普通的迭代器支持读和写。然而，如果对 `const` 对象调用 `begin()`和`end()`，或调用 `cbegin()`和`cend()`，将得到 `const_iterator`。`const_iterator` 是只读的，不能通过 `const_iterator` 修改它引用的元素。`iterator` 始终可以转换为 `const_iterator`，因此下面这种写法是安全的：

```
vector<type>::const_iterator it { begin(myVector) };
```

然而，`const_iterator` 不能转换为 `iterator`。如果 `myVector` 是 `const` 修饰的，那么下面这行代码无法通过编译：

```
vector<type>::iterator it { begin(myVector) };
```

注意:

如果不需要修改 `vector` 中的元素, 那么应该使用 `const_iterator`。遵循这条原则, 将更容易保证代码的正确性, 还允许编译器执行特定的优化。

在使用 `auto` 关键字时, `const_iterator` 的使用看上去有一点区别。假设有以下代码:

```
vector<string> stringVector(10, "hello");
for (auto iter { begin(stringVector) }; iter != end(stringVector); ++iter) {
    cout << *iter << endl;
}
```

由于使用了 `auto` 关键字, 编译器会自动推导 `iter` 变量的类型, 然后将其设置为普通的 `iterator`, 因为 `stringVector` 不是 `const` 修饰的。如果需要结合 `auto` 使用只读的 `const_iterator`, 那么需要使用 `cbegin()` 和 `cend()`, 而不是 `begin()` 和 `end()`, 如下所示:

```
for (auto iter { cbegin(stringVector) }; iter != cend(stringVector); ++iter) {
    cout << *iter << endl;
}
```

现在编译器会将 `iter` 变量的类型设置为 `const_iterator`, 因为 `cbegin()` 返回的就是 `const_iterator`。基于范围的 `for` 循环也可用于强制使用 `const iterator`, 如下所示:

```
for (const auto& element : stringVector) {
    cout << element << endl;
}
```

迭代器的安全性

通常情况下, 迭代器的安全性和指针接近: 非常不安全。例如, 可以编写如下代码:

```
vector<int> intVector;
auto iter { end(intVector) };
*iter = 10; // BUG! iter doesn't refer to a valid element.
```

此前提到过, `end()` 返回的迭代器越过了 `vector` 尾部。不是引用最后一个元素的迭代器! 试图解引用这个迭代器会产生不确定的行为。然而, 并没有要求迭代器本身执行任何验证操作。

如果使用了不匹配的迭代器, 则会引发另一个问题。例如, 下面的 `for` 循环初始化 `vectorTwo` 的一个迭代器, 然后试图和 `vectorOne` 的 `end` 迭代器进行比较。毫无疑问, 这个循环不会按照预想的行为执行, 可能永远都不会终止。在循环中解除引用迭代器可能产生不确定的后果。

```
vector<int> vectorOne(10);
vector<int> vectorTwo(10);

// Fill in the vectors.

// BUG! Possible infinite loop
for (auto iter { begin(vectorTwo) }; iter != end(vectorOne); ++iter) {
    // Loop body
}
```

注意:

Microsoft Visual C++ 在运行程序的调试版本时, 如果遇到前面的两个问题, 会给出断言错误。默认情况下, 不对发布版本执行任何迭代器验证操作。也可以给发布版本启用迭代器验证功能, 但这会降低性能。

其他迭代器操作

`vector` 迭代器是随机访问的，因此可以向前或向后移动，还可以随意跳跃。例如，下面的代码最终将 `vector` 中第 5 个元素(索引为 4)的值改为 4：

```
vector<int> intVector(10);
auto it { begin(intVector) };
it += 5;
--it;
*it = 4;
```

迭代器还是索引

既然可以编写 `for` 循环，使用简单索引变量和 `size()` 方法迭代 `vector` 中的元素，为什么还要使用迭代器？这个问题提得好，主要有 3 个原因：

- 使用迭代器可在容器的任意位置插入、删除元素或元素序列。详见后面的“添加和删除元素”部分。
- 使用迭代器可使用标准库算法，详情请参阅第 20 章“掌握标准库算法”的讨论。
- 通过迭代器顺序访问元素，通常比编制容器索引以单独检索每个元素的效率要高。这种特性不适用于 `vector`，但适用于 `list`、`map` 和 `set`。

在 `vector` 中存储引用

如本章开头所述，可在诸如 `vector` 的容器中存储引用。为此，可以在容器中存储 `std::reference_wrapper::std::ref()` 和 `cref()` 函数模板用于创建非 `const` 和 `const reference_wrapper` 实例。`get()` 方法用于访问由 `reference_wrapper` 包装的对象。但是需要包含 `<functional>` 头文件。示例如下：

```
string str1 { "Hello" };
string str2 { "World" };

// Create a vector of references to strings.
vector<reference_wrapper<string>> vec{ ref(str1) };
vec.push_back(ref(str2)); // push_back() works as well.

// Modify the string referred to by the second reference in the vector.
vec[1].get() += "!";

// The end result is that str2 is actually modified.
cout << str1 << " " << str2 << endl;
```

添加和删除元素

根据前面的描述，通过 `push_back()` 方法可向 `vector` 追加元素。`vector` 还提供了删除元素的对应方法：`pop_back()`。

警告：

`pop_back()` 不会返回已删除的元素。如果要访问这个元素，必须首先通过 `back()` 获得这个元素。

通过 `insert()` 方法可在 `vector` 中的任意位置插入元素，这个方法在迭代器指定的位置添加 1 个或多个元素，并将所有后续元素向后移动，给新元素腾出空间。`insert()` 有 5 种不同的重载形式：

- 插入单个元素。
- 插入单个元素的 n 份副本。

- 从某个迭代器范围插入元素。回顾一下，迭代器范围是半开区间，因此包含起始迭代器所指的元素，但不包含末尾迭代器所指的元素。
- 使用移动语义，将给定元素转移到 `vector` 中，插入一个元素。
- 向 `vector` 中插入一系列元素，这列元素是通过 `initializer_list` 指定的。

注意：

`push_back()` 和 `insert()` 还有把左值或右值作为参数的版本。两个版本都根据需要分配内存，以存储新元素。左值版本保存新元素的副本。右值版本使用移动语义，将给定元素的所有权转移到 `vector`，而不是复制它们。

通过 `erase()` 可在 `vector` 中的任意位置删除元素，通过 `clear()` 可删除所有元素。`erase()` 有两种形式：一种接收单个迭代器，删除单个元素；另一种接收两个迭代器，删除迭代器指定的元素范围。

下面的程序演示了添加和删除元素的方法。它使用一个辅助函数模板 `printVector()`，将 `vector` 的内容打印到 `cout`。第 12 章“利用模板编写泛型代码”详细讲解了如何编写函数模板。

```
template<typename T>
void printVector(const vector<T>& v)
{
    for (auto& element : v) { cout << element << " "; }
    cout << endl;
}
```

这个示例还演示了 `erase()` 的双参数版本和 `insert()` 的以下版本：

- `insert(const_iterator pos, const T& x)`：将值 `x` 插入位置 `pos`。
- `insert(const_iterator pos, size_type n, const T& x)`：将值 `x` 在位置 `pos` 插入 `n` 次。
- `insert(const_iterator pos, InputIterator first, InputIterator last)`：将范围 `[first, last)` 内的元素插入位置 `pos`。

该例的代码如下：

```
vector<int> vectorOne { 1, 2, 3, 5 };
vector<int> vectorTwo;

// Oops, we forgot to add 4. Insert it in the correct place
vectorOne.insert(cbegin(vectorOne) + 3, 4);

// Add elements 6 through 10 to vectorTwo
for (int i { 6 }; i <= 10; i++) {
    vectorTwo.push_back(i);
}
printVector(vectorOne);
printVector(vectorTwo);

// Add all the elements from vectorTwo to the end of vectorOne
vectorOne.insert(cend(vectorOne), cbegin(vectorTwo), cend(vectorTwo));
printVector(vectorOne);

// Now erase the numbers 2 through 5 in vectorOne
vectorOne.erase(cbegin(vectorOne) + 1, cbegin(vectorOne) + 5);
printVector(vectorOne);

// Clear vectorTwo entirely
vectorTwo.clear();
```

```
// And add 10 copies of the value 100
vectorTwo.insert(cbegin(vectorTwo), 10, 100);

// Decide we only want 9 elements
vectorTwo.pop_back();
printVector(vectorTwo);
```

这个程序的输出如下：

```
1 2 3 4 5
6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 6 7 8 9 10
100 100 100 100 100 100 100 100 100
```

回顾一下，迭代器对表示的是半开区间，而 `insert()` 将元素添加在迭代器位置引用的元素之前。因此，可按以下方法将 `vectorTwo` 的完整内容插入 `vectorOne` 尾部，如下所示：

```
vectorOne.insert(cend(vectorOne), cbegin(vectorTwo), cend(vectorTwo));
```

警告：

把 `vector` 范围作为参数的 `insert()` 和 `erase()` 等方法做了如下假定：头尾迭代器引用的是同一个容器中的元素，尾迭代器引用头迭代器所在的元素或其后面的元素。如果这些前提条件不满足，这些方法就不能正常工作。

如果想要删除所有满足条件的元素，一种解决方案是编写一个循环，遍历所有元素，并删除所有符合条件的元素。但是，该解决方案具有 $O(n^2)$ 复杂度，这对性能不利。这种复杂度可以通过使用 `remove-erase-idiom` 来避免，它具有线性复杂度。`remove-erase-idiom` 将在第 20 章讨论。

然而，从 C++20 开始，有一个更优雅的解决方案，它的形式是非成员函数 `std::erase()` 和 `std::erase_if()`，这是为所有标准库容器定义的。下面的代码片段演示了前者：

```
vector values { 1, 2, 3, 2, 1, 2, 4, 5 };
printVector(values);

erase(values, 2); // Removes all values equal to 2.
printVector(values);
```

输出结果如下：

```
1 2 3 2 1 2 4 5
1 3 1 4 5
```

`erase_if()` 的工作原理类似，但不是将值作为第 2 个参数传递，而是传递一个谓词，该谓词对于应该删除的元素返回 `true`，对于应该保留的元素返回 `false`。谓词可以采用函数指针、函数对象或 `lambda` 表达式的形式，所有这些都将在第 19 章“函数指针、函数对象和 `lambda` 表达式”中详细讨论。

移动语义

在某些情况下，向 `vector` 添加元素时可以利用移动语义提高性能。例如，假设有一个类型为字符串的 `vector`，如下所示：

```
vector<string> vec;
```


向这个 `vector` 添加元素，如下所示：

```
string myElement(5, 'a'); // Constructs the string "aaaaa"
vec.push_back(myElement);
```

但是，因为 `myElement` 不是临时对象，所以 `push_back()` 会生成 `myElement` 的副本，并存入 `vector`。

`vector` 类还定义了 `push_back(T&& val)`，这是 `push_back(const T& val)` 的移动版本。如果按照下列方式调用 `push_back()` 方法，那么就可以避免这种复制：

```
vec.push_back(move(myElement));
```

现在可以明确地说，`myElement` 应移入 `vector`。注意，在执行这个调用后，`myElement` 处于有效但不确定的状态。不应再使用 `myElement`，除非通过调用 `clear()` 等使其重返确定状态。还可以这样调用 `push_back()`：

```
vec.push_back(string(5, 'a'));
```

上述 `vec.push_back()` 调用会触发移动版本的调用，因为调用 `string` 构造函数后生成的是一个临时 `string` 对象。`push_back()` 方法将这个临时 `string` 对象移到 `vector` 中，从而避免了复制。

emplace 操作

C++ 在大部分标准库容器(包括 `vector`)中添加了对 `emplace` 操作的支持。`emplace` 的意思是“放置到位”。`emplace` 操作的一个示例是 `vector` 对象上的 `emplace_back()`，这个方法不会复制或移动任何数据，而是在容器中分配空间，然后就地构建对象。例如：

```
vec.emplace_back(5, 'a');
```

`emplace` 操作以变参模板的形式接收可变数目的参数。第 26 章“高级模板”将讨论可变参数模板 (variadic template)，但理解如何使用 `emplace_back()` 不需要这些细节。基本上，传递给 `emplace_back()` 的参数被转发给存储在 `vector` 中的类型的构造函数。`emplace_back()` 和使用移动语义的 `push_back()` 之间的性能差异取决于特定编译器实现这些操作的方式。大部分情况下，可根据自己喜好的语法来选择。

```
vec.push_back(string(5, 'a'));
// Or
vec.emplace_back(5, 'a');
```

从 C++17 开始，`emplace_back()` 方法返回已插入元素的引用。在 C++17 之前，`emplace_back()` 的返回类型是 `void`。

还有一个 `emplace()` 方法，可在 `vector` 的指定位置就地构建对象，并返回所插入元素的迭代器。

算法复杂度和迭代器失效

在 `vector` 中插入或删除元素，会导致后面的所有元素向后移动(给插入的元素腾出空间)或向前移动(将删除元素后空出来的空间填满)。因此，这些操作都服从线性复杂度。此外，引用插入点、删除点或随后位置的所有迭代器在操作之后都失效了。迭代器并不会自动移动，以便与 `vector` 中向前或向后移动的元素保持一致；这项工作需要你完成。

还要记住，`vector` 内部的重分配可能导致引用 `vector` 中元素的所有迭代器失效，而不只是那些引用插入点或删除点之后的元素的迭代器。相关详情，请参阅下一节。

vector 内存分配方案

`vector` 会自动分配内存来保存插入的元素。回顾一下，`vector` 要求元素必须放在连续的内存中，这与标准 C 风格数组类似。由于不可能请求在当前内存块的尾部添加内存，因此每次 `vector` 申请更

多内存时，都一定要在另一个位置分配一块新的更大的内存块，然后将所有元素复制/移动到新的内存块。这个过程非常耗时，因此 `vector` 的实现在执行重分配时，会分配比所需内存更多的内存，以尽量避免这个复制转移过程。通过这种方式，`vector` 可避免在每次插入元素时都重新分配内存。

现在，一个明显的问题是，作为 `vector` 的客户，为什么要关心 `vector` 内部是如何管理内存的。也许有人认为，抽象的原则应该允许不用考虑 `vector` 内部的内存分配方案。遗憾的是，必须理解 `vector` 内部的内存工作原理有 2 个原因：

(1) **效率。**`vector` 分配方案能保证元素插入采用摊还常量时间复杂度：也就是说，大部分操作都采用常量时间，但是也会有线性时间(需要重新分配内存时)。如果关注运行效率，那么可控制 `vector` 执行内存重分配的时机。

(2) **迭代器失效。**重分配会使引用 `vector` 内元素的所有迭代器失效。

因此，`vector` 接口允许查询和控制 `vector` 的重分配，这 2 项将在后面的小节中解释。如果不显式地控制重分配，那么应该假定每次插入都会导致重分配以及所有迭代器失效。

警告：

如果不显式地控制重分配，那么应该假定每次插入都会导致重分配，从而使所有迭代器失效。

大小和容量

`vector` 提供了两个可获得大小信息的方法：`size()`和 `capacity()`。`size()`方法返回 `vector` 中元素的个数，而 `capacity()`返回的是 `vector` 在重分配之前可以保存的元素个数。因此，在重分配之前还能插入的元素个数为 `capacity()-size()`。

注意：

通过 `empty()`方法可以查询 `vector` 是否为空。`vector` 可以为空，但容量不能为 0。

还有非成员 `std::ssize()`和 `std::empty()`全局函数，它们可以用于所有容器。它们还可以用于静态分配的 C 风格数组(不通过指针访问)以及 `initializer_list`。下面是一个将它们用于 `vector` 的例子：

```
vector<int> vec { 1,2,3 };
cout << size(vec) << endl;
cout << empty(vec) << endl;
```

此外，C++20 引入 `std::ssize()`作为全局非成员辅助函数，以有符号整型的形式返回 `size`。示例如下：

```
auto s1 { size(vec) }; // Type is size_t (unsigned)
auto s2 { ssize(vec) }; // Type is long long (signed)
```

预留容量

如果不关心效率和迭代器失效，那么也不需要显式地控制 `vector` 的内存分配。然而，如果希望程序尽可能高效，或要确保迭代器不会失效，就可以强迫 `vector` 预先分配足够的空间，以保存所有元素。当然，需要知道 `vector` 将保存多少元素，但有时这是无法预测的。

一种预分配空间的方式是调用 `reserve()`。这个方法负责分配保存指定数目元素的足够空间。接下来的时间片轮转类示例展示了 `reserve()`方法的实际使用。

警告：

为元素预留空间改变的是容量而非大小。也就是说，这个过程不会创建真正的元素。不要越过 `vector` 大小访问元素。

另一种预分配空间的方法是在构造函数中，或者通过 `resize()` 或 `assign()` 方法指定 `vector` 要保存的元素数目。这种方法会创建指定大小的 `vector` (容量也可能就是这么大)。

内存回收

如果需要，`vector` 会自动分配更多的内存；然而，它永远不会释放任何内存，除非 `vector` 被销毁。从 `vector` 中删除元素会减少 `vector` 的大小，但不会减少其容量。那如何回收它占用的内存呢？回收 `vector` 占用的所有内存的一个技巧是将其与空的 `vector` 交换。下面的代码片段展示了如何用一条语句回收名为 `values` 的 `vector` 的内存。第 3 行代码构造一个与 `values` 类型相同的临时空默认构造 `vector`，并将其与 `values` 进行交换。为 `values` 分配的所有内存现在都属于这个临时向量，在语句结束时将自动销毁，释放其所有内存。最终的结果是：为 `values` 分配的所有内存都被回收，`values` 的容量变为 0。

```
vector<int> values;
// Populate values ...
vector<int>().swap(values);
```

直接访问数据

`vector` 在内存中连续存储数据，可使用 `data()` 方法获取指向这块内存的指针。

非成员的 `std::data()` 全局函数可以用来获取数据的指针。它可用于 `array`、`vector` 容器、字符串、静态分配的 C 风格数组(不通过指针访问)和 `initializer_list`。下面是一个用于 `vector` 的示例：

```
vector<int> vec { 1,2,3 };
int* data1 { vec.data() };
int* data2 { data(vec) };
```

另一种访问 `vector` 内存块的方法是取第 1 个元素的地址，例如：`&vec[0]`。在遗留代码库中可能会发现这类代码，但这对空的 `vector` 来说并不安全；因此，建议不要使用它，而是使用 `data()`。

3. 移动语义

所有标准库容器都通过包含移动构造函数和移动赋值运算符来支持移动语义。有关移动语义的详细信息，请参阅第 9 章。这样做的一个好处是，可以很容易地从函数中按值返回标准库容器，而不会降低性能。示例如下：

```
vector<int> createVectorOfSize(size_t size)
{
    vector<int> vec(size);
    for (int contents { 0 }; auto& i : vec) { i = contents++; }
    return vec;
}
...
vector<int> myVector;
myVector = createVectorOfSize(123);
```

如果没有移动语义，将 `createVectorOfSize()` 的结果赋值给 `myVector`，这可能会调用拷贝赋值运算符。通过标准库容器中的移动语义支持，可以避免对 `vector` 的复制。相反，对 `myVector` 赋值会触发对移动赋值运算符的调用。

但是请记住，为了让移动语义在标准库容器中正常工作，容器中存储的类型的移动构造函数和移动赋值运算符必须标记为 `noexcept`！为什么这些移动方法不允许抛出任何异常？假设允许它们抛出异常。现在，当向 `vector` 添加新元素时，可能是 `vector` 的容量不够，需要分配更大的内存块。随后，`vector` 必须将所有数据从原始内存块复制或移动到新内存块。如果使用可能引发异常的移动方法来完成这一

操作，那么当部分数据已经被移动到新的内存块时，可能会引发异常。但实际上能做的并不多。为了避免这些问题，标准库容器仅在保证不抛出任何异常的情况下才使用移动方法。如果它们没有被标记为 `noexcept`，则将使用复制方法保证异常安全。

在实现自定义的类似标准库的容器时，有一个有用的辅助函数 `std::move_if_noexcept()`，在 `<utility>` 头文件中定义。这可用于调用移动构造函数或拷贝构造函数，具体取决于移动构造函数是否为 `noexcept`。就本身而言，`move_if_noexcept()` 并没有做太多的事情。它接收一个引用作为参数，如果移动构造函数为 `noexcept`，那么将其转换为右值引用，否则转换为对 `const` 的引用，但通过这个简单的技巧可以一次就调用到正确的构造函数。

标准库没有提供类似的辅助函数，根据前者是否为 `noexcept` 来调用移动赋值运算符或拷贝赋值运算符。虽然自己实现一个并不会太复杂，但是需要一些模板元编程技术和所谓的类型萃取来检查类型的属性。这两个主题将在第 26 章讨论，该章还提供了实现自定义的 `move_assign_if_noexcept()` 的示例。

4. vector 示例：一个时间片轮转类

计算机科学中的一个常见问题是在有限的资源列表中分配请求。例如，一个简单的操作系统可能保存了一个进程列表，然后给每个进程分配一个时间片（例如 100ms），进程在自己的时间片内完成一些工作。当时间片用完时，操作系统挂起当前进程，然后把时间片给予列表中的下一个进程，让那个进程执行一些操作。这个问题的一种最简单解决方法是时间片轮转调度（round-robin）。当最后一个进程的时间片用完时，调度器返回并开始执行第一个进程。例如，在一个 3 进程的例子中，将第 1 个时间片分配给第 1 个进程，将第 2 个时间片分配给第 2 个进程，将第 3 个时间片分配给第 3 个进程，第 4 个时间片则又回到第 1 个进程。这个循环按照这种方式无限继续下去。

假设编写一个通用的时间片轮转调度类，可以用于任何类型的资源。这个类应该支持添加和删除资源，还要支持循环遍历资源，以便获得下一资源。尽管可以直接使用 `vector`，但是通常最好编写一个包装类，以更直接地提供特定应用所需的功能。下例展示了一个 `RoundRobin` 类模板，其中带有解释代码的注释。首先给出类定义（从模块 `round_robin` 中导出）：

```
module;
#include <cstddef>
export module round_robin;
import <stdexcept>;
import <vector>;

// Class template RoundRobin
// Provides simple round-robin semantics for a list of elements.
export template <typename T>
class RoundRobin
{
public:
    // Client can give a hint as to the number of expected elements for
    // increased efficiency.
    explicit RoundRobin(size_t numExpected = 0);
    virtual ~RoundRobin() = default;
    // Prevent assignment and pass-by-value
    RoundRobin(const RoundRobin& src) = delete;
    RoundRobin& operator=(const RoundRobin& rhs) = delete;
    // Explicitly default a move constructor and move assignment operator
    RoundRobin(RoundRobin&& src) noexcept = default;
    RoundRobin& operator=(RoundRobin&& rhs) noexcept = default;
```

```

// Appends element to the end of the list. May be called
// between calls to getNext().
void add(const T& element);
// Removes the first (and only the first) element
// in the list that is equal (with operator==) to element.
// May be called between calls to getNext().
void remove(const T& element);
// Returns the next element in the list, starting with the first,
// and cycling back to the first when the end of the list is
// reached, taking into account elements that are added or removed.
T& getNext();
private:
    std::vector<T> m_elements;
    typename std::vector<T>::iterator m_nextElement;
};

```

从中可以看出, 这个公共接口非常简单明了: 只有 3 个方法, 再加上构造函数和析构函数。资源都保存在名为 `m_elements` 的 `vector` 中。迭代器 `m_nextElement` 总是指向下次调用 `getNext()` 返回的元素。如果还没有调用 `getNext()`, `m_nextElement` 就等于 `begin(m_elements)`。注意, 声明 `m_nextElement` 那一行前面的 `typename` 关键字。目前, 该关键字只用于指定模板类型参数, 但它还有另一个用途。当访问基于一个或多个模板参数的类型时, 必须显式地指定 `typename`。在这个示例中, 模板参数 `T` 用于访问迭代器类型。因此, 必须指定 `typename`。这是神秘 C++ 语法的另一个示例。

因为 `m_nextElement` 数据成员的存在, 这个类还避免了赋值和按值传递操作。为了让赋值和按值传递操作能正常工作, 必须要实现赋值运算符和拷贝构造函数, 并确保 `m_nextElement` 在目标对象中是可用的。

下面是 `RoundRobin` 类的实现代码, 其中带有代码的注释。注意构造函数中 `reserve()` 的使用, 以及 `add()`、`remove()` 和 `getNext()` 中迭代器的大量使用。最有技巧的部分是在 `add()` 和 `remove()` 方法中处理 `m_nextElement`。

```

template <typename T> RoundRobin<T>::RoundRobin(size_t numExpected)
{
    // If the client gave a guideline, reserve that much space.
    m_elements.reserve(numExpected);

    // Initialize m_nextElement even though it isn't used until
    // there's at least one element.
    m_nextElement = begin(m_elements);
}

// Always add the new element at the end
template <typename T> void RoundRobin<T>::add(const T& element)
{
    // Even though we add the element at the end, the vector could
    // reallocate and invalidate the m_nextElement iterator with
    // the push_back() call. Take advantage of the random-access
    // iterator features to save our spot.
    // Note: ptrdiff_t is a type capable of storing the difference
    // between two random-access iterators.
    ptrdiff_t pos { m_nextElement - begin(m_elements) };

    // Add the element.
    m_elements.push_back(element);
}

```

```

    // Reset our iterator to make sure it is valid.
    m_nextElement = begin(m_elements) + pos;
}

template <typename T> void RoundRobin<T>::remove(const T& element)
{
    for (auto it { begin(m_elements) }; it != end(m_elements); ++it) {
        if (*it == element) {
            // Removing an element invalidates the m_nextElement iterator
            // if it refers to an element past the point of the removal.
            // Take advantage of the random-access features of the iterator
            // to track the position of the current element after removal.
            ptrdiff_t newPos;

            if (m_nextElement == end(m_elements) - 1 &&
                m_nextElement == it) {
                // m_nextElement refers to the last element in the list,
                // and we are removing that last element, so wrap back to
                // the beginning.
                newPos = 0;
            } else if (m_nextElement <= it) {
                // Otherwise, if m_nextElement is before or at the one
                // we're removing, the new position is the same as before.
                newPos = m_nextElement - begin(m_elements);
            } else {
                // Otherwise, it's one less than before.
                newPos = m_nextElement - begin(m_elements) - 1;
            }

            // Erase the element (and ignore the return value).
            m_elements.erase(it);

            // Now reset our iterator to make sure it is valid.
            m_nextElement = begin(m_elements) + newPos;

            return;
        }
    }
}

template <typename T> T& RoundRobin<T>::getNext()
{
    // First, make sure there are elements.
    if (m_elements.empty()) {
        throw std::out_of_range("No elements in the list");
    }

    // Store the current element which we need to return.
    auto& toReturn { *m_nextElement };

    // Increment the iterator modulo the number of elements.
    ++m_nextElement;
    if (m_nextElement == end(m_elements)) { m_nextElement = begin(m_elements); }

    // Return a reference to the element.
    return toReturn;
}

```

```
}
```

下面是使用这个 **RoundRobin** 类模板的调度器的简单实现，其中包含代码的注释。

```
// Basic Process class.
class Process final
{
public:
    // Constructor accepting the name of the process.
    explicit Process(string_view name) : m_name { move(name) } {}

    // Lets a process perform its work for the duration of a time slice.
    void doWorkDuringTimeSlice() {
        cout << "Process " << m_name
              << " performing work during time slice." << endl;
        // Actual implementation omitted.
    }

    // Needed for the RoundRobin::remove() method to work.
    bool operator==(const Process&) const = default; // = default since C++20.
private:
    string m_name;
};

// Basic round-robin based process scheduler.
class Scheduler final
{
public:
    // Constructor takes a vector of processes.
    explicit Scheduler(const vector<Process>& processes)
    {
        // Add the processes
        for (auto& process : processes) { m_processes.add(process); }

        // Selects the next process using a round-robin scheduling algorithm
        // and allows it to perform some work during this time slice.
        void scheduleTimeSlice()
        {
            try {
                m_processes.getNext().doWorkDuringTimeSlice();
            } catch (const out_of_range&) {
                cerr << "No more processes to schedule." << endl;
            }
        }

        // Removes the given process from the list of processes.
        void removeProcess(const Process& process);
    private:
        RoundRobin<Process> m_processes;
};

int main()
{
```



```
vector processes { Process { "1" }, Process { "2" }, Process { "3" } };

Scheduler scheduler { processes };
for (int i { 0 }; i < 4; ++i) { scheduler.scheduleTimeSlice(); }

scheduler.removeProcess(processes[1]);
cout << "Removed second process" << endl;

for (int i { 0 }; i < 4; ++i) { scheduler.scheduleTimeSlice(); }
}
```

输出如下所示:

```
Process 1 performing work during time slice.
Process 2 performing work during time slice.
Process 3 performing work during time slice.
Process 1 performing work during time slice.
Removed second process
Process 3 performing work during time slice.
Process 1 performing work during time slice.
Process 3 performing work during time slice.
Process 1 performing work during time slice.
```

18.2.2 vector<bool>特化

C++标准要求对布尔值的 `vector` 进行部分特化, 目的是通过“打包”布尔值的方式来优化空间分配。布尔值要么是 `true`, 要么是 `false`, 因此可以通过一个位来表示, 一个位正好可以表示两个值。C++ 没有正好保存一个位的原始类型。一些编译器使用和 `char` 大小相同的类型来表示布尔值。其他一些编译器使用 `int` 类型。`vector<bool>` 特化应该是使用 1 位存储“布尔数组”, 从而节省空间。

注意:

可将 `vector<bool>` 表示为位字段(bit-field)而不是 `vector`。本章后面介绍的 `bitset` 容器是比 `vector<bool>` 功能更全面的位字段实现。然而, `vector<bool>` 的优势在于可以动态改变大小。

作为向 `vector<bool>` 提供一些位字段例程的非专门性尝试, 有一个额外的方法 `flip()` 可以对位进行补充。这个方法可在容器上调用, 此时对容器中的所有元素取反; 还可在 `operator[]` 或类似方法返回的单个引用上调用, 此时对单个元素取反。

那么, 可以对布尔值的引用调用方法吗? 答案是不可以。`vector<bool>` 特化实际上定义了一个名为 `reference` 的类, 用作底层布尔(或位)值的代理。当调用 `operator[]`、`at()` 或类似方法时, `vector<bool>` 返回 `reference` 对象, 这个对象是实际布尔值的代理。

警告:

由于 `vector<bool>` 返回的引用实际上是代理, 因此不能取地址以获得指向容器中实际元素的指针。

在实际应用中, 通过包装布尔值而节省一点空间似乎得不偿失。更糟糕的是, 访问和修改 `vector<bool>` 中的元素比访问 `vector<int>` 中的元素慢得多。很多 C++ 专家建议, 应该避免使用 `vector<bool>`, 而是使用 `bitset`。如果确实需要动态大小的位字段, 建议使用 `vector<std::int_fast8_t>` 或 `vector<unsigned char>`。`std::int_fast8_t` 类型在 `<cstdint>` 头文件中定义。这是一种有符号的整数类型, 编译器必须为其使用最快的整数类型(至少 8 位)。

18.2.3 deque

deque(double-ended queue 的简称)几乎和 vector 是等同的, 但用得更少。deque 定义在<deque>头文件中。主要区别如下所示。

- 不要求元素保存在连续内存中。
- deque 支持首尾两端常量时间的元素插入和删除操作(vector 只支持尾端的摊还常量时间)。
- deque 提供了 vector 中忽略的以下方法:
 - push_front(): 在开头插入一个元素。
 - pop_front(): 删除第一个元素。
 - emplace_front(): 在开始时, 就地创建一个新元素。从 C++17 开始, emplace_front() 返回已插入元素的引用而非 void。
- 在开头和末尾插入元素时, deque 未使迭代器失效。
- deque 没有通过 reserve() 和 capacity() 公开内存管理方案。

与 vector 相比, deque 用得非常少。因此, 这里不作详细讨论。要了解更多支持的方法, 请参阅标准库参考资源。

18.2.4 list

标准库 list 类模板定义在<list>头文件中, 是一种标准的双链表。list 支持链表中任意位置常量时间的元素插入和删除操作, 但访问单独元素的速度较慢(线性时间)。事实上, list 根本没有提供诸如 operator[] 的随机访问操作。只有通过迭代器才能访问单个元素。

list 的大部分操作都和 vector 的操作一致, 包括构造函数、析构函数、复制操作、赋值操作和比较操作。本节重点介绍那些和 vector 不同的方法。

1. 访问元素

list 提供的访问元素的方法仅有 front() 和 back(), 这两个方法的复杂度都是常量时间。这两个方法返回链表中第一个元素和最后一个元素的引用。对所有其他元素的访问都必须通过迭代器进行。

与 vector 类似, list 还支持 cbegin()、cend()、rbegin()、rend()、crbegin() 和 crend()。

警告:

list 不支持元素的随机访问。

2. 迭代器

list 迭代器是双向的, 不像 vector 迭代器那样提供随机访问。这意味着 list 迭代器之间不能进行加减操作和其他指针运算。例如, 如果 p 是一个 list 迭代器, 那么可以通过 ++p 或 --p 遍历链表元素, 但是不能使用加减运算符, p+n 和 p-n 都是不可以的。

3. 添加和删除元素

和 vector 一样, list 也支持添加和删除元素的方法, 包括 push_back()、pop_back()、emplace()、emplace_back()、5 种形式的 insert() 以及 2 种形式的 erase() 和 clear()。和 deque 一样, list 还提供了 push_front()、emplace_front() 和 pop_front()。只要找到正确的操作位置, 所有这些方法(clear() 除外)的复杂度都是常量时间。因此, list 适用于要在数据结构上执行很多插入和删除操作, 但不需要基于索引快速访问元素的应用程序。尽管如此, vector 可能还是要更快一些。具体可以使用性能分析器进行

确定。

4. list 大小

与 `deque` 一样，但和 `vector` 不同，`list` 不暴露底层的内存模型。因此，`list` 支持 `size()`、`empty()` 和 `resize()`，但不支持 `reserve()` 和 `capacity()`。注意，`list` 的 `size()` 方法具有常量时间复杂度。

5. list 特殊操作

`list` 提供了一些特殊操作，以利用其元素插入和删除很快这一特性。下面对这些操作进行概述并举一些例子。标准库参考资源提供了所有方法的全面参考。

串联

由于 `list` 类的本质是链表，因此可在另一个 `list` 的任意位置串联(`splice`)或插入整个 `list`，其复杂度是常量时间。使用的 `splice()` 方法的最简单版本如下：

```
// Store the a words in the main dictionary.
list<string> dictionary { "aardvark", "ambulance" };
// Store the b words.
list<string> bWords { "bathos", "balderdash" };
// Add the c words to the main dictionary.
dictionary.push_back("canticle");
dictionary.push_back("consumerism");
// Splice the b words into the main dictionary.
if (!bWords.empty()) {
    // Get an iterator to the last b word.
    auto iterLastB { --(cend(bWords)) };
    // Iterate up to the spot where we want to insert b words.
    auto it { cbegin(dictionary) };
    for (; it != cend(dictionary); ++it) {
        if (*it > *iterLastB)
            break;
    }
    // Add in the b words. This action removes the elements from bWords.
    dictionary.splice(it, bWords);
}
// Print out the dictionary.
for (const auto& word : dictionary) {
    cout << word << endl;
}
```

运行这个程序的结果如下所示：

```
aardvark
ambulance
bathos
balderdash
canticle
consumerism
```

`splice()` 还有其他两种形式：一种是插入其他 `list` 中的某个元素，另一种是插入其他 `list` 中的某个范围。另外，`splice()` 方法的所有形式都可以使用指向源 `list` 的普通引用或右值引用。

警告：

串联操作对作为参数传入的 `list` 来说是破坏性的：从一个 `list` 中删除要插入另一个 `list` 的元素。

更高效的算法版本

除 splice() 外, list 类还提供了一些泛型标准库算法的特殊实现。第 20 章将描述泛型算法。这里只讨论 list 提供的特殊版本。

注意:

如果可以选择, 请使用 list 方法而不是泛型标准库算法, 因为前者更高效。有时不必选择, 必须使用 list 特定方法。例如, 泛型算法 std::sort() 需要使用 list 没有提供的 RandomAccessIterator。

表 18-2 总结了 list 以方法形式提供特殊实现的算法。相关算法详情, 请参阅第 20 章。

表 18-2 list 以方法形式提供特殊实现的算法

方法	说明
remove()	从 list 中删除特定元素。从 C++20 开始, 这些方法以 size_t 的形式返回已移除元素的数量, 而在此之前, 它们不返回任何内容
remove_if()	
unique()	根据 operator== 运算符或用户提供的二元谓词, 从 list 中删除连续重复元素。从 C++20 开始, unique() 以 size_t 的形式返回已删除元素的数量, 而在此之前, 它不返回任何内容
merge()	合并两个 list。在开始前, 两个 list 都必须根据 operator< 运算符或用户定义的比较器排序。与 splice() 类似, merge() 对作为参数传入的 list 也具有破坏性
sort()	对 list 中的元素执行稳定排序操作
reverse()	翻转 list 中元素的顺序

6. list 示例: 确定注册情况

假设要为一所大学编写一个计算机注册系统。要提供的一项功能是: 从每个班的学生列表中生成大学录取学生的完整列表。在这个示例中, 假定只编写一个方法, 这个方法接收以学生姓名(用字符串表示)的 list 为元素的 vector, 以及因为没有支付学费而退学的学生 list。这个方法应该生成所有课程中所有学生的完整 list, 其中没有重复的学生, 也没有退学的学生。注意, 学生可能选择一门以上的课程。

下面是这个方法的代码, 带有代码注释。由于标准库 list 的巨大威力, 这个方法本身比描述信息还要短! 注意, 在标准库中允许容器嵌套: 在本例中, 使用了元素为 list 的 vector。

```
// courseStudents is a vector of lists, one for each course. The lists
// contain the students enrolled in those courses. They are not sorted.
//
// droppedStudents is a list of students who failed to pay their
// tuition and so were dropped from their courses.
//
// The function returns a list of every enrolled (non-dropped) student in
// all the courses.
list<string> getTotalEnrollment(const vector<list<string>>& courseStudents,
                                const list<string>& droppedStudents)
{
    list<string> allStudents;

    // Concatenate all the course lists onto the master list
    for (auto& lst : courseStudents) {
        allStudents.insert(cend(allStudents), cbegin(lst), cend(lst));
    }
}
```

```

// Sort the master list
allStudents.sort();

// Remove duplicate student names (those who are in multiple courses).
allStudents.unique();

// Remove students who are on the dropped list.
// Iterate through the dropped list, calling remove on the
// master list for each student in the dropped list.
for (auto& str : droppedStudents) {
    allStudents.remove(str);
}

// done!
return allStudents;
}

```

注意：

这个示例演示了 `list` 特定算法的使用。如前所述，`vector` 通常比 `list` 更快。因此，对于学生注册问题，建议只使用 `vector`，并将这些与泛型标准库算法结合在一起，但这些将在第 20 章讨论。

18.2.5 forward_list

`forward_list` 在 `<forward_list>` 头文件中定义，与 `list` 类似，区别在于 `forward_list` 是单链表，而 `list` 是双链表。这意味着 `forward_list` 只支持前向迭代，因此，范围的定义和 `list` 有所不同。如果需要修改任何链表，首先需要访问第一个元素之前的那个元素。由于 `forward_list` 没有提供反向遍历的迭代器，因此没有简单的方法可以访问前一个元素。所以，要修改的范围(例如提供给 `erase()` 和 `splice()` 的范围)必须是前开的。前面展示的 `begin()` 函数返回第一个元素的迭代器，因此只能用于构建前闭的范围。`forward_list` 类定义了一个 `before_begin()` 方法，它返回一个指向链表开头元素之前的假想元素的迭代器。当这个迭代器指向非法数据时，不能对这个迭代器解引用。然而，将这个迭代器递增 1 可得到与 `begin()` 返回的迭代器同样的效果；因此，这个方法可以用于构建前开的范围。

`list` 和 `forward_list` 之间的构造函数和赋值运算符类似。C++ 标准要求 `forward_list` 最小化其内存使用。这就是没有 `size()` 方法的原因，因为不提供它，就不需要存储列表的大小。表 18-3 总结了 `list` 和 `forward_list` 之间的区别。

表 18-3 list 和 forward_list 之间的区别

操作	list	forward_list
<code>assign()</code>	支持	支持
<code>back()</code>	支持	不支持
<code>before_begin()</code>	不支持	支持
<code>begin()</code>	支持	支持
<code>cbegin()</code>	不支持	支持
<code>cbegin()</code>	支持	支持
<code>cend()</code>	支持	支持
<code>clear()</code>	支持	支持
<code>crbegin()</code>	支持	不支持

(续表)

操作	list	forward_list
crend()	支持	不支持
emplace()	支持	不支持
emplace_after()	不支持	支持
emplace_back()	支持	不支持
emplace_front()	支持	支持
empty()	支持	支持
end()	支持	支持
erase()	支持	不支持
erase_after()	不支持	支持
front()	支持	支持
insert()	支持	不支持
insert_after()	不支持	支持
iterator/const_iterator	支持	支持
max_size	支持	支持
merge()	支持	支持
pop_back()	支持	不支持
pop_front()	支持	支持
push_back()	支持	不支持
push_front()	支持	支持
rbegin()	支持	不支持
remove()	支持	支持
remove_if()	支持	支持
rend()	支持	不支持
resize()	支持	支持
reverse()	支持	支持
reverse_iterator/const_reverse_iterator	支持	不支持
size()	支持	不支持
sort()	支持	支持
splice()	支持	不支持
splice_after()	不支持	支持
swap()	支持	支持
unique()	支持	支持

下例展示了 forward_list 的用法:

```
// Create 3 forward lists using an initializer_list
// to initialize their elements (uniform initialization).
forward_list<int> list1 { 5, 6 };
forward_list list2 { 1, 2, 3, 4 }; // CTAD is supported.
forward_list list3 { 7, 8, 9 };
```

```
// Insert list2 at the front of list1 using splice.
list1.splice_after(list1.before_begin(), list2);

// Add number 0 at the beginning of the list1.
list1.push_front(0);

// Insert list3 at the end of list1.
// For this, we first need an iterator to the last element.
auto iter { list1.before_begin() };
auto iterTemp { iter };
while (++iterTemp != end(list1)) { ++iter; }
list1.insert_after(iter, cbegin(list3), cend(list3));

// Output the contents of list1.
for (auto& i : list1) { cout << i << ' '; }
```

要插入 list3，需要一个指向链表中最后一个元素的迭代器。然而，由于这是一个 forward_list，因此不能使用--end(list1)，需要从头开始遍历这个链表，直到最后一个元素为止。这个示例的输出结果如下所示：

```
0 1 2 3 4 5 6 7 8 9
```

18.2.6 array

array 类定义在<array>头文件中，和 vector 类似，区别在于 array 的大小是固定的，不能增加或收缩。这个类的目的是让 array 能分配在栈上，而不是像 vector 那样总是需要访问自由存储区。

对于包含基本类型(整数、浮点数、字符、布尔值等)的 array，初始化元素的方式与 vector、list 等容器的初始化方式不同。如果在创建 array 时没有给初始化值，那么 array 的元素将是未初始化的，即包含垃圾。对于其他容器，例如 vector 和 list，元素总是初始化的，要么使用给定值，要么使用零初始化。因此，array 的行为实际上与 C 风格数组相同。

和 vector 一样，array 支持随机访问迭代器，元素都保存在连续内存中。array 支持 front()、back()、at() 和 operator[]，还支持使用 fill() 方法通过特定元素将 array 填满。由于 array 大小固定，因此不支持 push_back()、pop_back()、insert()、erase()、clear()、resize()、reserve() 和 capacity()。与 vector 相比，array 的缺点是，array 的 swap() 方法具有线性时间复杂度，而 vector 的 swap() 方法具有常量时间复杂度。array 的移动不是常量时间，而 vector 是。array 有 size() 方法，这显然是优于 C 风格数组的。下面的示例展示了如何使用 array 类。注意 array 声明需要两个模板参数；第 1 个参数指定元素类型，第 2 个参数指定 array 中元素的固定数量。

```
// Create an array of 3 integers and initialize them
// with the given initializer_list using uniform initialization.
array<int, 3> arr { 9, 8, 7 };
// Output the size of the array.
cout << "Array size = " << arr.size() << endl; // or std::size(arr);
// Output the contents using a range-based for loop.
for (const auto& i : arr) {
    cout << i << endl;
}

cout << "Performing arr.fill(3)..." << endl;
// Use the fill method to change the contents of the array.
```



```
arr.fill(3);
// Output the contents of the array using iterators.
for (auto iter { cbegin(arr) }; iter != cend(arr); ++iter) {
    cout << *iter << endl;
}
```

运行这段代码的输出结果如下：

```
Array size = 3
9
8
7
Performing arr.fill(3)...
3
3
3
```

可使用 `std::get<n>()` 函数模板，从 `std::array` 检索位于索引位置 n 的元素。索引必须是常量表达式，不能是循环变量等。使用 `std::get<n>()` 的优势在于编译器在编译期会检查给定索引是有效的，否则将导致编译错误，如下所示。

```
array<int, 3> myArray{ 11, 22, 33 };
cout << std::get<1>(myArray) << endl;
cout << std::get<10>(myArray) << endl; // BUG! Compilation error!
```

C++20 引入了一个新的非成员函数 `std::to_array()`，它在 `<array>` 头文件中定义，它使用元素的复制初始化将给定的 C 风格数组转换为 `std::array`。该函数仅适用于一维数组。下面是一个简单的示例：

```
auto arr1 { to_array({ 11, 22, 33 }) }; // Type is array<int, 3>

double carray[] { 9, 8, 7, 6 };
auto arr2 { to_array(carray) }; // Type is array<double, 4>
```

C++20 18.2.7 span

假设使用这个函数打印一个 `vector` 中的内容：

```
void print(const vector<int>& values)
{
    for (const auto& value : values) { cout << value << " "; }
    cout << endl;
}
```

假设还想打印 C 风格数组的内容，一种选择是重载 `print()` 函数，以接受指向数组第一个元素的指针，以及要打印的元素数量。

```
void print(const int values[], size_t count)
{
    for (size_t i { 0 }; i < count; ++i) { cout << values[i] << " "; }
    cout << endl;
}
```

如果还想打印 `std::arrays`，那么可以提供第 3 个重载，但是函数的参数类型是什么？对于 `std::array`，必须指定 `array` 中的元素类型和数量作为模板参数。事情似乎变得越来越复杂了。

`std::span` 在 C++20 中引入并在 `<span>` 头文件中定义，它在这里的作用至关重要，因为它允许编写

单个函数来处理 `vector`、C 风格数组和任意大小的 `std::array`。下面是使用 `span` 的 `print()` 函数的实现：

```
void print(span<int> values)
{
    for (const auto& value : values) { cout << value << " "; }
    cout << endl;
}
```

注意，就像第2章中的 `string_view` 一样，`span` 的复制成本很低；它基本上只包含一个指向序列第1个元素的指针和一些元素。`span` 永远不会复制数据！因此，它通常是通过按值传递的。

有几个用于创建 `span` 的构造函数。例如，可以创建一个包含给定 `vector`、`std::array` 或 C 风格数组的所有元素的数组。还可以通过传递第1个元素的地址和想要在 `span` 中包含的元素的数量，来创建一个只包含部分容器元素的 `span`。

可以使用 `subspan()` 方法从现有的 `span` 创建子视图。它的第1个参数是 `span` 中的偏移量，第2个参数是包含在子视图中的元素数量。还有两个名为 `first()` 和 `last()` 的附加方法，分别返回包含前 n 个元素或后 n 个元素的 `span` 的子视图。

`span` 有两个类似于 `vector` 和 `array` 的方法：`begin()`、`end()`、`rbegin()`、`rend()`、`front()`、`back()`、`operator[]`、`data()`、`size()` 和 `empty()`。

下面的代码片段演示了调用 `print(span)` 函数的几种方法：

```
vector v { 11, 22, 33, 44, 55, 66 };
// Pass the whole vector, implicitly converted to a span.
print(v);
// Pass an explicitly created span.
span mySpan { v };
print(mySpan);
// Create a subview and pass that.
span subspan { mySpan.subspan(2, 3) };
print(subspan);
// Pass a subview created in-line.
print({ v.data() + 2, 3 });

// Pass an std::array.
array<int, 5> arr { 5, 4, 3, 2, 1 };
print(arr);
print({ arr.data() + 2, 3 });

// Pass a C-style array.
int carr[] { 9, 8, 7, 6, 5 };
print(carr); // The entire C-style array.
print({ carr + 2, 3 }); // A subview of the C-style array.
```

输出结果如下：

```
11 22 33 44 55 66
11 22 33 44 55 66
33 44 55
33 44 55
5 4 3 2 1
3 2 1
9 8 7 6 5
7 6 5
```

与提供字符串只读视图的 `string_view` 不同，`span` 可以提供对底层元素的读/写访问。记住，`span`

只包含一个指向序列中第 1 个元素的指针和元素的数量；也就是说，`span` 永远不会复制数据！因此，修改 `span` 中的元素实际上是修改底层序列中的元素。如果不需要，可以创建一个包含 `const` 元素的 `span`。例如，`print()` 函数没有理由修改指定 `span` 中的任何元素。可以通过以下方式防止修改：

```
void print(span<const int> values)
{
    for (const auto& value : values) { cout << value << " "; }
    cout << endl;
}
```

注意：

在编写接收 `const vector<T>&` 的函数时，请考虑使用 `span<const T>` 作为替换。这样函数就可以处理来自 `vector`、`array`、`C` 风格数组等的数据序列的视图和子视图。

18.3 容器适配器

除标准的顺序容器外，标准库还提供了 3 种容器适配器：`queue`、`priority_queue` 和 `stack`。每种容器适配器都对一种顺序容器的包装。它们允许交换底层容器，不必修改其他代码。容器适配器的作用是简化接口，只提供那些 `stack` 和 `queue` 抽象所需的功能。例如，容器适配器没有提供迭代器，也没有提供同时插入或删除多个元素的功能。

18.3.1 queue

`queue` 容器适配器定义在 `<queue>` 头文件中，`queue` 提供了标准的“先入先出”语义。与通常情况一样，`queue` 也写为类模板形式，如下所示。

```
template <class T, class Container = deque<T>> class queue;
```

`T` 模板参数指定要保存在 `queue` 中的类型。另一个模板参数指定 `queue` 适配的底层容器。不过，由于 `queue` 要求顺序容器同时支持 `push_back()` 和 `pop_front()` 两个操作，因此只有两个内建的选项：`deque` 和 `list`。大部分情况下，只使用默认的选项 `deque` 即可。

1. queue 操作

`queue` 接口非常简单：只有 8 个方法，再加上构造函数和普通的比较运算符。`push()` 和 `emplace()` 方法在 `queue` 的尾部添加一个新元素，`pop()` 从 `queue` 的头部移除元素。通过 `front()` 和 `back()` 可以分别获得第 1 个元素和最后 1 个元素的引用，而不会删除元素。与其他容器一样，在调用 `const` 对象时，`front()` 和 `back()` 返回的是 `const` 引用；调用非 `const` 对象时，这些方法返回的是非 `const` 引用(可读写)。

警告：

`pop()` 不会返回弹出的元素。如果需要获得一份元素的副本，必须首先通过 `front()` 获得这个元素。

`queue` 还支持 `size()`、`empty()` 和 `swap()`。

2. queue 示例：网络数据包缓冲

当两台计算机通过网络通信时，互相发送的信息被分割为离散的块，称为数据包(packet)。计算机操作系统的网络层必须捕获数据包，并在数据包到达时将数据包保存起来。然而，计算机可能没有

足够的带宽同时处理所有数据包。因此，网络层通常会将数据包缓存或保存起来，直到更高的层有机会处理它们。数据包应该按照到达的顺序处理，因此这个问题特别适用于 `queue` 结构。下面是一个简单的 `PacketBuffer` 类，其中附带代码的注释，这个类将收到的数据包保存在 `queue` 中，直到数据包被处理。这是一个类模板，因此网络层中的不同层可以使用它处理不同类型的数据包，例如 IP 包或 TCP 包。这个类允许客户指定最大大小，因为操作系统为避免使用过多内存，通常会限制可保存的数据包的数目。当缓冲区变满时，后续到达的数据包都被丢弃了。

```
export template <typename T>
class PacketBuffer
{
public:
    // If maxSize is 0, the size is unlimited, because creating
    // a buffer of size 0 makes little sense. Otherwise only
    // maxSize packets are allowed in the buffer at any one time.
    explicit PacketBuffer(size_t maxSize = 0);

    virtual ~PacketBuffer() = default;

    // Stores a packet in the buffer.
    // Returns false if the packet has been discarded because
    // there is no more space in the buffer, true otherwise.
    bool bufferPacket(const T& packet);

    // Returns the next packet. Throws out_of_range
    // if the buffer is empty.
    [[nodiscard]] T getNextPacket();
private:
    std::queue<T> m_packets;
    size_t m_maxSize;
};

template <typename T> PacketBuffer<T>::PacketBuffer(size_t maxSize/*= 0*/)
    : m_maxSize { maxSize }
{
}

template <typename T> bool PacketBuffer<T>::bufferPacket(const T& packet)
{
    if (m_maxSize > 0 && m_packets.size() == m_maxSize) {
        // No more space. Drop the packet.
        return false;
    }
    m_packets.push(packet);
    return true;
}

template <typename T> T PacketBuffer<T>::getNextPacket()
{
    if (m_packets.empty()) {
        throw std::out_of_range { "Buffer is empty" };
    }
    // Retrieve the head element
    T temp { m_packets.front() };
    // Pop the head element
    m_packets.pop();
}
```

```

        // Return the head element
        return temp;
    }

```

这个类的实际应用需要使用多线程。C++提供了一些同步类，允许对共享对象的线程进行安全访问。如果没有提供显式的同步，那么当至少一个线程修改标准库对象时，任何标准库对象都无法安全地用于多线程环境。第 27 章“C++多线程编程”将讨论同步。本例的关注点是 `queue` 类，所以这是一个使用了 `PacketBuffer` 的单线程示例：

```

class IPPacket final
{
public:
    explicit IPPacket(int id) : m_id { id } {}
    int getID() const { return m_id; }
private:
    int m_id;
};

int main()
{
    PacketBuffer<IPPacket> ipPackets { 3 };

    // Add 4 packets
    for (int i { 1 }; i <= 4; ++i) {
        if (!ipPackets.bufferPacket(IPPacket { i })) {
            cout << "Packet " << i << " dropped (queue is full)." << endl;
        }
    }

    while (true) {
        try {
            IPPacket packet { ipPackets.getNextPacket() };
            cout << "Processing packet " << packet.getID() << endl;
        } catch (const out_of_range&) {
            cout << "Queue is empty." << endl;
            break;
        }
    }
}

```

程序的输出结果如下所示：

```

Packet 4 dropped (queue is full).
Processing packet 1
Processing packet 2
Processing packet 3
Queue is empty.

```

18.3.2 priority_queue

优先队列(priority queue)是一种按顺序保存元素的队列。优先队列不保证严格的 FIFO 顺序，而是保证在队列头部的元素任何时刻都具有最高优先级。这个元素可能是队列中最老的那个元素，也可能是最新的那个元素。如果两个元素的优先级相等，那么它们在队列中的相对顺序是未确定的。

`priority_queue` 容器适配器也定义在 `<queue>` 头文件中。其模板定义如下(稍有简化):

```
template <class T, class Container = vector<T>,
         class Compare = less<T>>;
```

这个类没有看上去这么复杂。之前看到了前两个参数: `T` 是 `priority_queue` 中保存的元素类型; `Container` 是 `priority_queue` 适配的底层容器。 `priority_queue` 默认使用 `vector`, 但是也可以使用 `deque`。这里不能使用 `list`, 因为 `priority_queue` 要求随机访问元素。第 3 个参数 `Compare` 复杂一些。正如第 19 章将要介绍的, `less` 是一个类模板, 支持两个类型为 `T` 的元素通过 `operator<` 运算符进行比较。也就是说, 要根据 `operator<` 确定队列中元素的优先级, 可以自定义这里使用的比较操作, 但这是第 19 章的内容。目前, 只要保证为保存在 `priority_queue` 中的类型正确定义了 `operator<` 即可。

注意:

`priority_queue` 的头元素是优先级最高的元素, 默认情况下优先级是通过 `operator<` 运算符来判断的, 比其他元素“小”的元素的优先级要比其他元素低。

1. `priority_queue` 提供的操作

`priority_queue` 提供的操作比 `queue` 还要少。`push()` 和 `emplace()` 可以插入元素, `pop()` 可以删除元素, `top()` 可以返回头元素的 `const` 引用。

警告:

在非 `const` 对象上调用 `top()`, `top()` 返回的也是 `const` 引用, 因为修改元素可能会改变元素的顺序, 所以不允许修改元素。 `priority_queue` 没有提供获得尾元素的机制。

警告:

`pop()` 不返回弹出的元素。如果需要获得一份副本, 必须首先通过 `top()` 获得这个元素。

与 `queue` 一样, `priority_queue` 支持 `size()`、`empty()` 和 `swap()`。然而, `priority_queue` 没有提供任何比较运算符。

2. `priority_queue` 示例: 错误相关器

系统上的单个故障通常会导致不同组件生成多个错误。优秀的错误处理系统通过错误相关性 (error correlation) 首先处理最重要的错误。通过 `priority_queue` 可以编写一个非常简单的错误相关器 (error correlator)。假设所有的错误事件都编码了自己的优先级。下面这个类根据优先级对错误事件进行排序, 因此优先级最高的错误总是最先处理。这个类的定义如下:

```
// Sample Error class with just a priority and a string error description.
export class Error final
{
public:
    Error(int priority, std::string errorString)
        : m_priority { priority }, m_errorString { std::move(errorString) } {}
    int getPriority() const { return m_priority; }
    const std::string& getErrorString() const { return m_errorString; }
    // Compare Errors according to their priority. (C++20 operator<=)
    auto operator<=(const Error& rhs) const {
        return getPriority() <= rhs.getPriority(); }
private:
    int m_priority;
```

```

        std::string m_errorString;
    };

    // Stream insertion overload for Errors.
    export std::ostream& operator<<(std::ostream& os, const Error& err)
    {
        os << std::format("{} (priority {})", err.getErrorString(), err.getPriority());
        return os;
    }

    // Simple ErrorCorrelator class that returns highest priority errors first.
    export class ErrorCorrelator final
    {
    public:
        // Add an error to be correlated.
        void addError(const Error& error) { m_errors.push(error); }
        // Retrieve the next error to be processed.
        [[nodiscard]] Error getError()
        {
            // If there are no more errors, throw an exception.
            if (m_errors.empty()) {
                throw out_of_range { "No more errors." };
            }
            // Save the top element.
            Error top { m_errors.top() };
            // Remove the top element.
            m_errors.pop();
            // Return the saved element.
            return top;
        }
    private:
        std::priority_queue<Error> m_errors;
    };

```

下面这个简单的单元测试展示了 `ErrorCorrelator` 的用法。实际需要使用多线程，以便一个线程添加错误，另一个线程处理错误。正如之前的 `queue` 示例中提到的，这需要显式地同步，相关讨论，请参阅第 27 章。

```

ErrorCorrelator ec;
ec.addError(Error { 3, "Unable to read file" });
ec.addError(Error { 1, "Incorrect entry from user" });
ec.addError(Error { 10, "Unable to allocate memory!" });

while (true) {
    try {
        Error e { ec.getError() };
        cout << e << endl;
    } catch (const out_of_range&) {
        cout << "Finished processing errors" << endl;
        break;
    }
}

```

这个程序的输出如下所示：

```

Unable to allocate memory! (priority 10)
Unable to read file (priority 3)

```



```
Incorrect entry from user (priority 1)
Finished processing errors
```

18.3.3 stack

`stack` 和 `queue` 几乎相同，区别在于 `stack` 提供先入后出(FILO)的语义，这种语义也称为后入先出，以区别于 FIFO。`stack` 定义在 `<stack>` 头文件中。模板定义如下所示：

```
template <class T, class Container = deque<T>> class stack;
```

可将 `vector`、`list` 或 `deque` 用作 `stack` 的底层容器。

1. stack 操作

与 `queue` 类似，`stack` 提供了 `push()`、`emplace()` 和 `pop()` 方法。区别在于：`push()` 在 `stack` 顶部添加一个新元素，将之前插入的所有元素都“向下推”；而 `pop()` 从 `stack` 顶部删除一个元素，这个元素是最近插入的元素。如果在 `const` 对象上调用，`top()` 方法返回顶部元素的 `const` 引用；如果在非 `const` 对象上调用，`top()` 方法返回非 `const` 引用。

警告：

`pop()` 不返回弹出的元素。如果需要获得一份副本，必须首先通过 `top()` 获得这个元素。

`stack` 支持 `empty()`、`size()`、`swap()` 和标准的比较运算符。

2. stack 示例：修改后的错误相关器

可以重写之前的 `ErrorCorrelator` 类，使其给出最新错误而不是最高优先级的错误。唯一要修改的是将 `priority_queue` 的 `m_errors` 替换为 `stack`，根据现在这个变更，错误以 LIFO 顺序而不是优先级顺序分发。方法定义不需要做任何修改，因为 `priority_queue` 和 `stack` 中都有 `push()`、`pop()`、`top()` 和 `empty()` 方法。

18.4 有序关联容器

与顺序容器不同，有序关联容器不采用线性方式保存元素。相反，有序关联容器将键映射到值。通常情况下，有序关联容器的插入、删除和查找时间是相等的。

标准库提供的 4 个有序关联容器分别为：`map`、`multimap`、`set` 和 `multiset`。每种有序关联容器都将元素保存在类似于树的有序数据结构中。还有 4 个无序关联容器：`unordered_map`、`unordered_multimap`、`unordered_set` 和 `unordered_multiset`。它们将在本章后面进行讨论。

18.4.1 pair 工具类

在学习有序关联容器之前，首先要熟悉第 1 章介绍过的 `pair` 类模板，它在 `<utility>` 头文件中定义，并将两个可能属于不同类型的值组合起来。可以通过 `first` 和 `second` 公共数据成员访问这两个值。所有比较运算符都支持对 `first` 值和 `second` 值进行比较。下面给出了一些示例：

```
// Two-argument constructor and default constructor
pair<string, int> myPair { "hello", 5 };
pair<string, int> myOtherPair;
```

```
// Can assign directly to first and second
myOtherPair.first = "hello";
myOtherPair.second = 6;

// Copy constructor
pair<string, int> myThirdPair { myOtherPair };

// operator<
if (myPair < myOtherPair) {
    cout << "myPair is less than myOtherPair" << endl;
} else {
    cout << "myPair is greater than or equal to myOtherPair" << endl;
}

// operator==
if (myOtherPair == myThirdPair) {
    cout << "myOtherPair is equal to myThirdPair" << endl;
} else {
    cout << "myOtherPair is not equal to myThirdPair" << endl;
}
```

输出结果如下所示:

```
myPair is less than myOtherPair
myOtherPair is equal to myThirdPair
```

通过使用类模板参数推导, 可以省略模板类型参数, 并简单地编写如下代码。注意, 使用了标准的用户自定义的字符串字面量 `s`。

```
pair myPair { "hello"s, 5 }; // Type is pair<string, int>.
```

在 C++17 引入对 CTAD 的支持之前, 可以使用 `std::make_pair()` 工具函数模板, 从两个值构造一个 `pair`。下面是构造一个 `pair`(两个值类型分别为 `int` 型和 `double` 型)的 3 种方法:

```
pair<int, double> aPair = make_pair(5, 10.10);
auto pair2 { make_pair(5, 10.10) };
pair pair3 { 5, 10.10 }; // CTAD
```

18.4.2 map

`map` 定义在 `<map>` 头文件中, 它保存的是键/值对, 而不是只保存值。插入、查找和删除操作都是基于键的, 值只不过是附属品。从概念上讲, `map` 这个术语源于容器将键“映射”到值。

`map` 根据键对元素排序存储, 因此插入、删除和查找的复杂度都是对数时间。由于排好了序, 因此枚举元素时, 元素按类型的 `operator<` 或用户自定义的比较器确定的顺序出现。通常情况下, `map` 实现为某种形式的平衡树, 例如红黑树。不过, 树的结构并没有向用户公开。

当需要根据键保存和获取元素时, 以及需要按特定顺序保存元素时, 应该使用 `map`。

1. 构建 map

`map` 类模板接收 4 种类型: 键类型、值类型、比较类型以及分配器类型。和以往一样, 本章不考虑分配器。比较类型和之前描述的 `priority_queue` 中的比较类型类似, 允许提供与默认不同的比较类。本章只使用默认的 `less` 比较。使用默认的比较类型时, 要确保键都支持 `operator<` 运算符。如果对更多细节感兴趣, 第 19 章将介绍如何编写自己的比较类。

如果忽略比较参数和分配器参数，那么 `map` 的构建和 `vector` 或 `list` 的构建是一样的，区别在于，在模板实例化中需要分别指定键和值的类型。例如，下面的代码构建了一个 `map`，它使用 `int` 值作为键，`Data` 类的对象作为值：

```
class Data final
{
public:
    explicit Data(int value = 0) : mValue { value } { }
    int getValue() const { return mValue; }
    void setValue(int value) { mValue = value; }

private:
    int mValue;
};
...
map<int, Data> dataMap;
```

在内部，`dataMap` 为 `map` 中的每个元素存储一个 `pair<int, Data>`。

`map` 还支持统一初始化机制。下面的 `map` 在内部存储了一个 `pair<string, int>` 的实例：

```
map<string, int> m {
    { "Marc G.", 123 },
    { "Warren B.", 456 },
    { "Peter V.W.", 789 }
};
```

类模板参数推导不能像往常一样工作。例如，下面的代码将不能通过编译：

```
map m {
    { "Marc G.", 123 },
    { "Warren B.", 456 },
    { "Peter V.W.", 789 }
};
```

这不起作用，因为编译器不能从 `{"Marc G", 123}` 推导出 `pair<string, int>`。如果需要，那么可以这样编写代码(注意字符串字面量 `s` 后缀！)：

```
map m {
    pair { "Marc G."s, 123 },
    pair { "Warren B."s, 456 },
    pair { "Peter V.W."s, 789 }
};
```

2. 插入元素

向顺序容器(例如 `vector` 和 `list`)插入元素时，总是需要指定要插入元素的位置，而 `map` 不一样，它和其他关联容器都不需要指定插入位置。`map` 的内部实现会决定需要保存新元素的位置，只需要提供键和值即可。

注意：

`map` 和其他有序关联容器提供了接收迭代器位置作为参数的 `insert()` 方法。然而，这个位置只是容器找到正确位置的一种“提示”。不强制容器在那个位置插入元素。

在插入元素时，一定要记住 `map` 需要“唯一键”：`map` 中的每个元素都要有不同的键。如果需要支持多个带有同一键的元素，有两个选择：可使用 `map`，把另一个容器(如 `vector` 或 `array`)用作键的值，

也可以使用后面描述的 `multimap`。

`insert()`方法

可使用 `insert()` 方法向 `map` 添加元素，它有一个好处：允许判断键是否已经存在。`insert()` 方法的一个问题是必须将键/值对指定为 `pair` 对象或 `initializer_list`。`insert()` 的基本形式的返回类型是迭代器和布尔值组成的 `pair`。返回类型这么复杂的原因是，如果指定的键已经存在，那么 `insert()` 不会改写元素值。返回的 `pair` 中的 `bool` 元素指出，`insert()` 是否真的插入了新的键/值对。迭代器引用的是 `map` 中带有指定键的元素(根据插入成功与否，这个键对应的值可能是新值或旧值)。`map` 迭代器将在下一节中更详细地讨论。继续上一节的 `map` 示例，可以采用下面的方式使用 `insert()`：

```
map<int, Data> dataMap;

auto ret { dataMap.insert({ 1, Data { 4 } }) }; // Using an initializer_list
if (ret.second) {
    cout << "Insert succeeded!" << endl;
} else {
    cout << "Insert failed!" << endl;
}

ret = dataMap.insert(make_pair(1, Data { 6 })); // Using a pair object
if (ret.second) {
    cout << "Insert succeeded!" << endl;
} else {
    cout << "Insert failed!" << endl;
}
```

`ret` 变量的类型是 `pair`，如下所示：

```
pair<map<int, Data>::iterator, bool> ret;
```

`pair` 的第 1 个元素是 `map` 迭代器，用于键为 `int` 类型、值为 `Data` 类型的 `map`。`pair` 的第 2 个元素是布尔值。

程序的输出结果如下：

```
Insert succeeded!
Insert failed!
```

使用 `if` 语句的初始化器(从 C++17 开始)，只使用一条语句，即可将数据插入 `map` 并检查结果，示例如下：

```
if (auto result { dataMap.insert({ 1, Data { 4 } }) }; result.second) {
    cout << "Insert succeeded!" << endl;
} else {
    cout << "Insert failed!" << endl;
}
```

这甚至可以与 C++17 的结构化绑定结合使用：

```
if (auto [iter, success] { dataMap.insert({ 1, Data { 4 } }) }; success) {
    cout << "Insert succeeded!" << endl;
} else {
    cout << "Insert failed!" << endl;
}
```

`insert_or_assign()`方法

`insert_or_assign()` 与 `insert()` 的返回类型类似。但是，如果已经存在具有给定键的元素，`insert_or_assign()` 将用新值改写旧值，而 `insert()` 在这种情况下不会改写旧值。与 `insert()` 的另一个区别

在于, `insert_or_assign()` 有两个独立的参数: 键和值。下面是一个示例:

```
ret = dataMap.insert_or_assign(1, Data { 7 });
if (ret.second) {
    cout << "Inserted." << endl;
} else {
    cout << "Overwritten." << endl;
}
```

operator[]

向 `map` 插入元素的另一种方法是通过重载的 `operator[]`。这种方法的区别主要在于语法: 键和值是分别指定的。此外, `operator[]` 总是成功的。如果给定键没有对应的元素值, 那么就会创建带有对应键值的新元素。如果具有给定键的元素已经存在, `operator[]` 会将元素值替换为新指定的值。下面是前面使用操作符 `[]` 代替 `insert()` 的部分示例:

```
map<int, Data> dataMap;
dataMap[1] = Data { 4 };
dataMap[1] = Data { 6 }; // Replaces the element with key 1
```

不过, `operator[]` 有一点要注意: 它总会构建一个新的值对象, 即使并不需要使用这个值对象, 也同样如此。因此, 需要为元素值提供一个默认的构造函数, 这样可能会比 `insert()` 的效率低。

如果请求的元素不存在, 那么 `operator[]` 会在 `map` 中创建一个新元素, 所以这个运算符没有被标记为 `const`。尽管这很明显, 但有时可能会看上去违背常理。例如, 假设有下面这个函数:

```
void func(const map<int, int>& m)
{
    cout << m[1] << endl; // Error
}
```

这段代码无法成功编译, 尽管看上去只是想读取 `m[1]` 的值。这段代码编译失败的原因是: 变量 `m` 是对 `map` 的 `const` 引用, 而 `operator[]` 没有被标记为 `const`。相反, 应该使用“查找元素”一节中描述的 `find()` 方法。

emplace 方法

`map` 支持 `emplace()` 和 `emplace_hint()`, 从而在原位置构建元素, 这与 `vector` 的 `emplace` 方法类似。还有一个 `try_emplace()` 方法, 如果给定的键还不存在, 那么它将在原位置插入元素; 如果 `map` 中已经存在相应的键, 则什么也不做。

3. map 迭代器

`map` 迭代器的工作方式类似于顺序容器的迭代器。主要区别在于迭代器引用的是键值对, 而不只是值。如果要访问值, 那么必须通过 `pair` 对象的 `second` 字段来访问。下面展示了如何遍历前一个示例中的 `map`:

```
for (auto iter { cbegin(dataMap) }; iter != cend(dataMap); ++iter) {
    cout << iter->second.getValue() << endl;
}
```

再来看一下用于访问值的表达式:

```
iter->second.getValue()
```

`iter` 引用了一个键值对, 因此可通过 `->` 运算符访问这个 `pair` 的 `second` 字段, 这个字段是一个 `Data`

对象。然后可以调用 `Data` 对象的 `getValue()` 方法。

注意，下面的代码功能等效：

```
(*iter).second.getValue()
```

使用基于范围的 `for` 循环，可按如下可读性更高的方法编写代码：

```
for (const auto& p : dataMap) {
    cout << p.second.getValue() << endl;
}
```

结合使用基于范围的 `for` 循环和结构化绑定，实现的方式会更优雅：

```
for (const auto& [key, data] : dataMap) {
    cout << data.getValue() << endl;
}
```

警告：

可以通过非 `const` 迭代器修改元素值，但如果试图修改元素的键（即使通过非 `const` 迭代器来修改），编译器会生成错误，因为修改键会破坏 `map` 中元素的排序。

4. 查找元素

`map` 可根据指定的键查找元素，时间复杂度为指数时间。如果已经知道指定键的元素存在于 `map` 中，那么查找它的最简单方法就是通过 `operator[]`，只需要在非 `const` `map` 或 `map` 的非 `const` 引用上调用它。`operator[]` 的好处在于返回可直接使用和修改的元素引用，而不必考虑从 `pair` 对象中获得值。下面是对之前示例的扩展，这里对键为 1 的 `Data` 对象值调用了 `setValue()` 方法。

```
map<int, Data> dataMap;
dataMap[1] = Data { 4 };
dataMap[1] = Data { 6 };
dataMap[1].setValue(100);
```

然而，如果不知道元素是否存在，就不能使用 `operator[]`。因为如果元素不存在，这个运算符会插入一个包含相应键的新元素。作为替换方案，`map` 提供了 `find()` 方法。如果元素在 `map` 中存在，那么这个方法会返回指向具有指定键的元素的迭代器；如果元素在 `map` 中不存在，则返回 `end()` 迭代器。下面的示例通过 `find()` 方法对键为 1 的 `Data` 对象执行同样的修改操作：

```
auto it { dataMap.find(1) };
if (it != end(dataMap)) {
    it->second.setValue(100);
}
```

从以上代码可以看出，使用 `find()` 有点笨拙，但有时这是必要的。

如果只想知道在 `map` 中是否存在具有给定键的元素，那么可以使用 `count()` 成员函数。这个函数返回 `map` 中给定键的元素个数。对于 `map` 来说，这个函数返回的结果不是 0 就是 1，因为 `map` 中不允许有具有重复键的元素。

从 C++20 开始，所有关联容器（有序和无序）都有一个名为 `contains()` 的方法。如果容器中存在给定的键，那么返回 `true`，否则返回 `false`。这样，就不再需要使用 `count()` 来确定某个键是否在关联容器中。示例如下：

```
auto isKeyInMap { dataMap.contains(1) };
```

5. 删除元素

`map` 允许在指定的迭代器位置删除一个元素或删除指定迭代器范围内的所有元素，这两种操作的复杂度分别为摊还常量和对数时间。从用户的角度看，用于执行上述操作的两个 `erase()` 方法等同于顺序容器中的 `erase()` 方法。然而，`map` 的一个重要特性是，它还提供了另一个版本的 `erase()`，用来删除匹配键的元素。示例如下：

```
map<int, Data> dataMap;
dataMap[1] = Data { 4 };
cout << format("There are {} elements with key 1.", dataMap.count(1)) << endl;
dataMap.erase(1);
cout << format("There are {} elements with key 1.", dataMap.count(1)) << endl;
```

输出结果如下：

```
There are 1 elements with key 1
There are 0 elements with key 1
```

6. 结点

所有有序和无序的关联容器都被称为基于结点的数据结构。从 C++17 开始，标准库以结点句柄 (node handle) 的形式提供对结点的直接访问。确切类型并未指定，但每个容器都有一个名为 `node_type` 的类型别名，它指定容器结点句柄的类型。结点句柄只能移动，是结点中存储的元素的所有者。它提供对键和值的读写访问。

可以基于给定的迭代器位置或键，从关联容器(作为结点句柄)中使用 `extract()` 方法提取结点。从容器提取结点时，将会把它从容器中删除，因为返回的结点句柄是所提取元素的唯一拥有者。

C++ 提供了新的 `insert()` 重载，以允许在容器中插入结点句柄。

使用 `extract()` 提取结点句柄和使用 `insert()` 插入结点句柄，可以有效地将数据从一个关联容器传递给另一个关联容器，而不需要执行任何复制或移动操作。甚至可以将结点从 `map` 移到 `multimap`，从 `set` 移到 `multiset`。继续上一节的示例，下面的代码片段将键为 1 的结点转到第 2 个 `map`：

```
map<int, Data> dataMap2;
auto extractedNode { dataMap.extract(1) };
dataMap2.insert(std::move(extractedNode));
```

可将最后两行合并为一行：

```
dataMap2.insert(dataMap.extract(1));
```

还有一个附加的操作 `merge()`，可以将所有结点从一个关联容器移到另一个关联容器。无法移动的结点将留在源容器中，因为它们可能会导致目标容器中的重复，而目标容器不允许重复。示例如下：

```
map<int, int> src { {1, 11}, {2, 22} };
map<int, int> dst { {2, 22}, {3, 33}, {4, 44}, {5, 55} };
dst.merge(src);
```

完成合并操作后，`src` 仍然包含一个元素 {2, 22}，因为目标已经包含这个元素，所以无法移动。操作后，`dst` 包含 {1, 11}、{2, 22}、{3, 33}、{4, 44} 和 {5, 55}。

7. map 示例：银行账号

通过 `map` 可实现一个简单的银行账号数据库。一种常用模式是使用类或结构体的一个字段作为保存在 `map` 中的键。在本例中，这个键就是账号。下面是简单的 `BankAccount` 类和 `BankDB` 类：


```

export class BankAccount final
{
public:
    BankAccount(int accountNumber, std::string name)
        : m_accountNumber { accountNumber }, m_clientName { std::move(name) } {}

    void setAccountNumber(int accountNumber) {
        m_accountNumber = accountNumber; }
    int getAccountNumber() const { return m_accountNumber; }

    void setClientName(std::string name) { m_clientName = std::move(name); }
    const std::string& getClientName() const { return m_clientName; }
private:
    int m_accountNumber;
    std::string m_clientName;
};

export class BankDB final
{
public:
    // Adds account to the bank database. If an account exists already
    // with that number, the new account is not added. Returns true
    // if the account is added, false if it's not.
    bool addAccount(const BankAccount& account);

    // Removes the account accountNumber from the database.
    void deleteAccount(int accountNumber);

    // Returns a reference to the account represented
    // by its number or the client name.
    // Throws out_of_range if the account is not found.
    BankAccount& findAccount(int accountNumber);
    BankAccount& findAccount(std::string name);

    // Adds all the accounts from db to this database.
    // Deletes all the accounts from db.
    void mergeDatabase(BankDB& db);
private:
    std::map<int, BankAccount> m_accounts;
};

```

下面是 BankDB 方法的实现，其中带有代码注释：

```

bool BankDB::addAccount(const BankAccount& account)
{
    // Do the actual insert, using the account number as the key
    auto res { m_accounts.emplace(account.getAccountNumber(), account) };
    // or: auto res { m_accounts.insert(
    //     pair { account.getAccountNumber(), account } ) };

    // Return the bool field of the pair specifying success or failure
    return res.second;
}

void BankDB::deleteAccount(int accountNumber)
{

```

```

    m_accounts.erase(accountNumber);
}

BankAccount& BankDB::findAccount(int accountNumber)
{
    // Finding an element via its key can be done with find()
    auto it { m_accounts.find(accountNumber) };
    if (it == end(m_accounts)) {
        throw out_of_range("No account with that number.");
    }

    // Remember that iterators into maps refer to pairs of key/value
    return it->second;
}

BankAccount& BankDB::findAccount(std::string name)
{
    // Finding an element by a non-key attribute requires a linear
    // search through the elements. With C++17 structured bindings:
    for (auto& [accountNumber, account] : m_accounts) {
        if (account.getClientName() == name) {
            return account; // found it!
        }
    }
    throw out_of_range("No account with that name.");
}

void BankDB::mergeDatabase(BankDB& db)
{
    // Use C++17 merge().
    m_accounts.merge(db.m_accounts);
    // Or: m_accounts.insert(begin(db.m_accounts), end(db.m_accounts));

    // Now clear the source database.
    db.m_accounts.clear();
}

```

可通过以下代码测试 BankDB 类:

```

BankDB db;
db.addAccount(BankAccount { 100, "Nicholas Solter" });
db.addAccount(BankAccount { 200, "Scott Kleper" });

try {
    auto& acct { db.findAccount(100) };
    cout << "Found account 100" << endl;
    acct.setClientName("Nicholas A Solter");

    auto& acct2 { db.findAccount("Scott Kleper") };
    cout << "Found account of Scott Kleper" << endl;

    auto& acct3 { db.findAccount(1000) };
} catch (const out_of_range& caughtException) {
    cout << "Unable to find account: " << caughtException.what() << endl;
}

```

输出结果如下所示:

```
Found account 100
Found account of Scott Kleper
Unable to find account: No account with that number.
```

18.4.3 multimap

multimap 是一种允许多个元素使用同一个键的 map。和 map 一样, multimap 支持统一初始化。multimap 的接口和 map 的接口几乎相同, 区别在于:

- multimap 不提供 operator[] 和 at()。它们的语义在多个元素可以使用同一个键的情况下没有意义。
- 在 multimap 上执行插入操作总是会成功。因此, 添加单个元素的 multimap::insert() 方法只返回迭代器而不返回 pair。
- map 支持 insert_or_assign() 和 try_emplace() 方法, 而 multimap 不支持。

注意:

multimap 允许插入相同的键值对。如果要避免这种冗余, 必须在插入新元素之前执行显式检查。

multimap 的最棘手之处是查找元素。不能使用 operator[], 因为并没有提供 operator[]。find() 也不是非常有用, 因为 find() 返回的是指向具有给定键的任意一个元素的迭代器(未必是具有这个键的第一个元素)。

然而, multimap 将所有带同一个键的元素保存在一起, 并提供方法以获得这个子范围的迭代器, 这个子范围内的元素在容器中具有相同的键。lower_bound() 和 upper_bound() 方法分别返回匹配给定键的第一个元素和最后一个元素之后那个元素(one-past-the-last)的对应迭代器。如果没有元素匹配这个键, 那么 lower_bound() 和 upper_bound() 返回的迭代器相等。

如果需要获得具有给定键的元素对应的迭代器, 使用 equal_range() 方法比依次调用 lower_bound() 和 upper_bound() 更高效。equal_range() 方法返回一个包含两个迭代器的 pair, 这两个 iterator 分别是 lower_bound() 和 upper_bound() 返回的迭代器。

注意:

map 中也有 lower_bound()、upper_bound() 和 equal_range() 方法, 但由于 map 中不允许多个元素带有同一个键, 因此在 map 中, 这些方法的用处不大。

multimap 示例: 好友列表

大部分在线聊天软件都允许用户有一个“好友列表”。聊天软件给好友列表中的用户赋予特殊权限, 例如允许他们向用户发送未经请求的消息。

在线聊天软件实现好友列表的一种方式是将信息保存在 multimap 中。一个 multimap 可保存每个用户的好友列表。容器中的每一项保存用户的一个好友。键是用户, 值是好友。例如, 如果 Harry Potter 和 Ron Weasley 都出现在对方的好友列表中, 那么应该有两项, 一项将 Harry Potter 映射到 Ron Weasley, 另一项将 Ron Weasley 映射到 Harry Potter。multimap 允许同一个键有多个值, 因此同一个用户允许有多个好友。下面是 BuddyList 类的定义:

```
export class BuddyList final
{
    public:
```

```

    // Adds buddy as a friend of name.
    void addBuddy(const std::string& name, const std::string& buddy);
    // Removes buddy as a friend of name
    void removeBuddy(const std::string& name, const std::string& buddy);
    // Returns true if buddy is a friend of name, false otherwise.
    bool isBuddy(const std::string& name, const std::string& buddy) const;
    // Retrieves a list of all the friends of name.
    std::vector<std::string> getBuddies(const std::string& name) const;
private:
    std::multimap<std::string, std::string> m_buddies;
};

```

下面是这个类的实现，其中包含代码注释。这个实现演示了 `lower_bound()`、`upper_bound()` 和 `equal_range()` 的用法：

```

void BuddyList::addBuddy(const string& name, const string& buddy)
{
    // Make sure this buddy isn't already there. We don't want
    // to insert an identical copy of the key/value pair.
    if (!isBuddy(name, buddy)) {
        m_buddies.insert({ name, buddy }); // Using initializer_list
    }
}

void BuddyList::removeBuddy(const string& name, const string& buddy)
{
    // Obtain the beginning and end of the range of elements with
    // key 'name'. Use both lower_bound() and upper_bound() to demonstrate
    // their use. Otherwise, it's more efficient to call equal_range().
    auto begin { m_buddies.lower_bound(name) }; // Start of the range
    auto end { m_buddies.upper_bound(name) }; // End of the range

    // Iterate through the elements with key 'name' looking
    // for a value 'buddy'. If there are no elements with key 'name',
    // begin equals end, so the loop body doesn't execute.
    for (auto iter { begin }; iter != end; ++iter) {
        if (iter->second == buddy) {
            // We found a match! Remove it from the map.
            m_buddies.erase(iter);
            break;
        }
    }
}

bool BuddyList::isBuddy(const string& name, const string& buddy) const
{
    // Obtain the beginning and end of the range of elements with
    // key 'name' using equal_range(), and C++17 structured bindings.
    auto [begin, end] { m_buddies.equal_range(name) };

    // Iterate through the elements with key 'name' looking
    // for a value 'buddy'.
    for (auto iter { begin }; iter != end; ++iter) {
        if (iter->second == buddy) {
            // We found a match!
            return true;
        }
    }
}

```

```

    }
    // No matches
    return false;
}

vector<string> BuddyList::getBuddies(const string& name) const
{
    // Obtain the beginning and end of the range of elements with
    // key 'name' using equal_range(), and C++17 structured bindings.
    auto [begin, end] { m_buddies.equal_range(name) };

    // Create a vector with all names in the range (all buddies of name).
    vector<string> buddies;
    for (auto iter { begin }; iter != end; ++iter) {
        buddies.push_back(iter->second);
    }
    return buddies;
}

```

注意，`removeBuddy()`不能使用删除具有给定键的所有元素的那个 `erase()`版本，它只应删除具有指定键的一个元素，而不是删除具有指定键的所有元素。还要注意，`getBuddies()`不能在 `vector` 上通过 `insert()`向 `equal_range()`返回的范围插入元素，因为 `multimap` 迭代器引用的元素是键值对，而不是字符串。`getBuddies()`方法必须显式地遍历范围，将字符串从每一个键值 `pair` 中抽取出来，然后插入要返回的新 `vector`。

下面是对 `BuddyList` 的测试：

```

BuddyList buddies;
buddies.addBuddy("Harry Potter", "Ron Weasley");
buddies.addBuddy("Harry Potter", "Hermione Granger");
buddies.addBuddy("Harry Potter", "Hagrid");
buddies.addBuddy("Harry Potter", "Draco Malfoy");
// That's not right! Remove Draco.
buddies.removeBuddy("Harry Potter", "Draco Malfoy");

buddies.addBuddy("Hagrid", "Harry Potter");
buddies.addBuddy("Hagrid", "Ron Weasley");
buddies.addBuddy("Hagrid", "Hermione Granger");

auto harrysFriends { buddies.getBuddies("Harry Potter") };

cout << "Harry's friends: " << endl;
for (const auto& name : harrysFriends) {
    cout << "\t" << name << endl;
}

```

输出结果如下所示：

```

Harry's friends:
    Ron Weasley
    Hermione Granger
    Hagrid

```

18.4.4 set

`set` 容器定义在 `<set>` 头文件中，和 `map` 非常类似。区别在于 `set` 保存的不是键值对，在 `set` 中，值

本身就是键。如果信息没有显式的键，且希望进行排序(不包含重复)以便快速地执行插入、查找和删除，就可以考虑使用 `set` 容器存储此类信息。

`set` 提供的接口几乎和 `map` 提供的接口完全相同，主要区别在于 `set` 没有提供 `operator[]`、`insert_or_assign()` 和 `try_emplace()`。

不能修改 `set` 中元素的键/值，因为修改容器中的 `set` 元素会破坏顺序。

set 示例：访问控制列表

在计算机系统上实现基本安全控制的一种方法是使用访问控制列表。系统上的每个实体(如文件和设备)都有一个用户列表，列出了有权访问相应实体的用户。通常只有拥有特殊权限的用户才能在实体的访问权限列表中添加和删除用户。在系统内部，`set` 容器可以很好地表示访问控制列表。每个实体可以使用一个 `set`，其中包含所有允许访问这个实体的用户名。下面是这个简单访问控制列表的类定义：

```
export class AccessList final
{
public:
    // Default constructor
    AccessList() = default;
    // Constructor to support uniform initialization.
    AccessList(std::initializer_list<std::string> users)
    {
        m_allowed.insert(begin(users), end(users));
    }
    // Adds the user to the permissions list.
    void addUser(std::string user)
    {
        m_allowed.emplace(user);
    }
    // Removes the user from the permissions list.
    void removeUser(std::string& user)
    {
        m_allowed.erase(string(user));
    }
    // Returns true if the user is in the permissions list.
    bool isAllowed(std::string& user) const
    {
        return (m_allowed.count(user) != 0);
    }
    // Returns a vector of all the users who have permissions.
    std::vector<std::string> getAllUsers() const
    {
        return { begin(m_allowed), end(m_allowed) };
    }
private:
    std::set<std::string> m_allowed;
};
```

`getAllUsers()` 这行的实现十分有趣，有必要分析一下。将这一行构建的 `vector<string>` 返回给 `vector` 构造函数，`return` 的参数是 `m_allowed` 的 `begin` 和 `end` 迭代器。如有必要，可将其拆分为两行：

```
std::vector<std::string> users(begin(m_allowed), end(m_allowed));
return users;
```

下面是一个简单的测试程序：

```
AccessList fileX { "mgregoire", "baduser" };
fileX.addUser("pvw");
fileX.removeUser("baduser");

if (fileX.isAllowed("mgregoire")) {
    cout << "mgregoire has permissions" << endl;
}

if (fileX.isAllowed("baduser")) {
    cout << "baduser has permissions" << endl;
}

auto users { fileX.getAllUsers() };
for (const auto& user : users) {
    cout << user << " ";
}
```

`AccessList` 类有一个构造函数使用 `initializer_list` 作为参数，这样就可以使用统一初始化语法，测试程序中 `fileX` 变量的初始化演示了这种用法。

程序的输出结果如下所示：

```
mgregoire has permissions
mgregoire pvw
```

注意，`m_allowed` 数据成员需要的是 `std::strings` 的 `set`，而不是 `string_views` 的 `set`。将其更改为 `string_views` 的 `set` 将会引入悬空指针的问题。例如，假设有下面的代码：

```
AccessList fileX;
{
    string user { "someuser" };
    fileX.addUser(user);
}
```

这个代码片段创建了一个名为 `user` 的字符串，然后将其添加到 `fileX` 访问控制列表中。但是，字符串和对 `addUser()` 的调用都在一组花括号中；也就是说，字符串的生存期比 `fileX` 短。在结束大括号处，字符串超出作用域将会被销毁。这将使 `fileX` 访问控制列表的 `string_view` 指向一个被破坏的字符串，即一个悬空指针！

18.4.5 multiset

`multiset` 和 `set` 的关系等同于 `multimap` 和 `map` 的关系。`multiset` 支持 `set` 的所有操作，但允许容器中同时保存多个互等的元素。这里没有提供 `multiset` 的示例，这是因为 `multiset` 与 `set` 和 `multimap` 实在太相似了。

18.5 无序关联容器/哈希表

标准库支持无序关联容器或哈希表。这种容器有 4 个：`unordered_map`、`unordered_multimap`、`unordered_set` 和 `unordered_multiset`。此前讨论的 `map`、`multimap`、`set` 和 `multiset` 容器会对元素进行排序，而这些新的无序版本不会对元素进行排序。

18.5.1 哈希函数

无序关联容器也称为哈希表，这是因为它们使用了哈希函数(hash function)。哈希表的实现通常会使用某种形式的数组，数组中的每个元素都称为桶(bucket)。每个桶都有一个特定的数值索引，例如 0、1、2 直到最后一个桶。哈希函数将键转换为哈希值，再转换为桶索引。与这个键关联的值在桶中存储。

哈希函数的结果未必是唯一的。两个或多个键哈希到同一个桶索引，就称为冲突(collision)。当使用不同的键得到相同的哈希值，或把不同的哈希值转换为同一桶索引时，就会发生冲突。可采用多种方法处理冲突，例如二次重哈希(quadratic re-hashing)和线性链(linear chaining)等方法。感兴趣的读者可参阅附录 B 中“算法和数据结构”部分列出的任意参考文献。标准库没有指定要求使用哪种冲突处理算法，但目前大部分实现都选择通过线性链解决冲突。使用线性链时，桶不直接包含与键关联的数据值，而包含一个指向链表的指针。这个链表包含特定桶中的所有数据值。图 18-1 展示了原理。

图 18-1 中有两个冲突。之所以出现第 1 个冲突，是因为对键“Marc G.”和“John D.”应用哈希函数后得到同一个哈希值，该哈希值被映射到桶索引 128。这个桶指向一个包含键“Marc G.”和“John D.”及其对应数据值的链表。第 2 个冲突由“Scott K.”和“Johan G.”的哈希值引起，它们被映射到相同的桶索引 129。

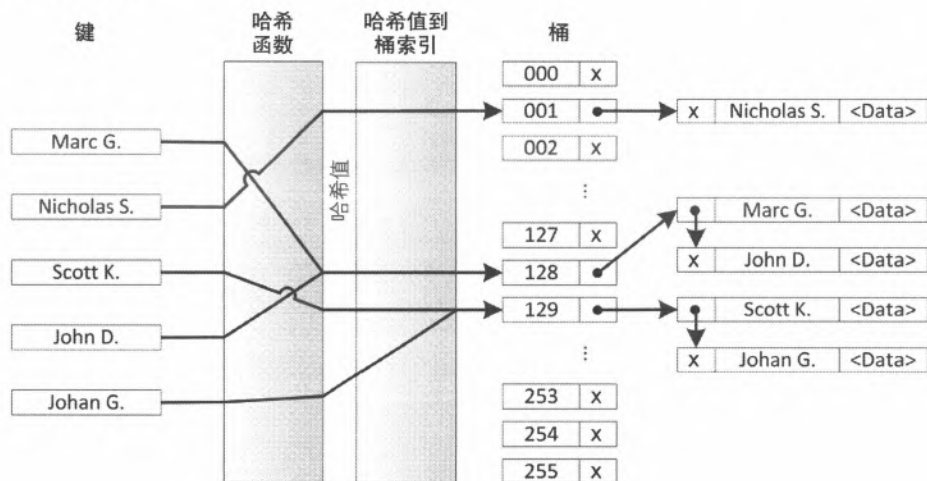


图 18-1 两个冲突

从图 18-1 中还可看出基于键的查找的工作原理以及查找的复杂度。查找过程包括调用一次哈希函数来计算哈希值，哈希值此后被转换为桶索引。一旦知道了桶索引，将在链表中通过一次或多次相等操作找到正确的键。从中还能看出，相比普通 map 的查找方式，这种查找方式要快得多，但查找速度完全取决于冲突次数。

哈希函数的选择非常重要。不产生冲突的哈希函数称为“完美哈希”。完美哈希的查找时间是常量；常规的哈希查找时间平均接近于 1，与元素数量无关。随着冲突数的增加，查找时间会增加，性能会降低。增加基本哈希表的大小，可以减少冲突，但需要考虑高速缓存的大小。

C++ 标准为指针和所有基本数据类型(例如 bool、char、int、float、double 等)提供了哈希函数，还为 error_code、error_condition、optional、variant、bitset、unique_ptr、shared_ptr、type_index、string、string_view、vector<bool> 和 thread::id 提供了哈希函数。如果要使用的键类型没有可用的标准哈希函

数，就必须实现自己的哈希函数。即使键集是固定的、已知的，创建完美哈希也并不简单；需要进行深入的数学分析。纵然创建得不算完美，但性能较高，仍然充满挑战。由于篇幅所限，本书不会详细解释哈希函数的数学原理，只会列举一个十分简单的哈希函数示例。

下面的示例演示了如何编写自定义的哈希函数。代码定义了一个类 `IntWrapper`，它仅封装了一个整数。还提供了 `operator==`，因为这是在无效关联容器中使用键所必需的。

```
class IntWrapper
{
public:
    IntWrapper(int i) : m_wrappedInt { i } {}
    int getValue() const { return m_wrappedInt; }
    bool operator==(const IntWrapper&) const = default; // = default since C++20
private:
    int m_wrappedInt;
};
```

为给 `IntWrapper` 编写哈希函数，应该先给 `IntWrapper` 编写 `std::hash` 模板的特例。`std::hash` 模板在 `<functional>` 头文件中定义。这个特例需要实现函数调用运算符，计算并返回给定 `IntWrapper` 实例的哈希。对于本例，请求被简单地转发给整数的标准哈希函数：

```
namespace std
{
    template<> struct hash<IntWrapper>
    {
        size_t operator()(const IntWrapper& x) const {
            return std::hash<int>{}(x.getValue());
        }
    };
}
```

注意，一般不允许把任何内容放在 `std` 名称空间中，但 `std` 类模板的特例是这条规则的例外。函数调用运算符的实现只有一行代码，它为整数的标准哈希函数创建了一个实例 `std::hash<int>{}`，然后对该实例通过参数 `f.getValue()` 执行函数调用运算符。注意，这个转发在本例中是有效的，因为 `IntWrapper` 只包含一个数据成员：一个整数。如果该类包含多个数据成员，就需要在计算哈希时考虑所有数据成员，但这些细节超出了本书的讨论范围。

18.5.2 unordered_map

`unordered_map` 容器在 `<unordered_map>` 头文件中定义，也是一个类模板，如下所示：

```
template <class Key,
          class T,
          class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Alloc = std::allocator<std::pair<const Key, T>>>
class unordered_map;
```

总共有 5 个模板参数：键类型、值类型、哈希类型、判等比较类型和分配器类型。通过后面 3 个参数可以分别自定义哈希函数、判等比较函数和分配器函数。通常可忽略这些参数，因为它们有默认值。最重要的参数是前两个参数。与 `map` 一样，可使用统一初始化机制来初始化 `unordered_map`，如下所示：

```
unordered_map<int, string> m {
```

```
{1, "Item 1"},
{2, "Item 2"},
{3, "Item 3"},
{4, "Item 4"}
};
// Using C++17 structured bindings.
for (const auto& [key, value] : m) {
    cout << format("{} = {}", key, value) << endl;
}

// Without structured bindings.
for (const auto& p : m) {
    cout << format("{} = {}", p.first, p.second) << endl;
}
```

表 18-4 总结了 map 和 unordered_map 之间的区别。

表 18-4 map 和 unordered_map 之间的区别

操作	map	unordered_map
at()	支持	支持
begin()	支持	支持
begin(n)	不支持	支持
bucket()	不支持	支持
bucket_count()	不支持	支持
bucket_size()	不支持	支持
cbegin()	支持	支持
cbegin(n)	不支持	支持
cend()	支持	支持
cend(n)	不支持	支持
clear()	支持	支持
contains()	支持	支持
count()	支持	支持
crbegin()	支持	不支持
crend()	支持	不支持
emplace()	支持	支持
emplace_hint()	支持	支持
empty()	支持	支持
end()	支持	支持
end(n)	不支持	支持
equal_range()	支持	支持
erase()	支持	支持
extract()	支持	支持
find()	支持	支持
insert()	支持	支持
insert_or_assign()	支持	支持

C++20

(续表)

操作	map	unordered_map
iterator/const_iterator	支持	支持
load_factor()	不支持	支持
local_iterator/const_local_iterator	不支持	支持
lower_bound()	支持	不支持
max_bucket_count()	不支持	支持
max_load_factor()	不支持	支持
max_size()	支持	支持
merge()	支持	支持
operator[]	支持	支持
rbegin()	支持	不支持
rehash()	不支持	支持
rend()	支持	不支持
reserve()	不支持	支持
reverse_iterator/const_reverse_iterator	支持	不支持
size()	支持	支持
swap()	支持	支持
try_emplace()	支持	支持
upper_bound()	支持	不支持

与普通的 map 一样, unordered_map 中的所有键都应该是唯一的。表 18-4 包含一些哈希专用方法。例如, load_factor() 返回每一个桶的平均元素数, 以反映冲突的次数。bucket_count() 方法返回容器中桶的数目。还提供了 local_iterator 和 const_local_iterator, 用于遍历单个桶中的元素, 但不能用于遍历多个桶。bucket(key) 方法返回包含指定键的桶索引, begin(n) 返回引用索引为 n 的桶中第一个元素的 local_iterator, end(n) 返回引用索引为 n 的桶中最后一个元素之后的那个元素(one-past-the-last)的 local_iterator。下面的示例将演示这些方法的用法。

unordered_map 示例: 电话簿

下面的示例将使用 unordered_map 表示电话簿。使用人名表示键, 电话号码则是与键关联的值。

```

void printMap(const auto& m) // C++20 abbreviated function template
{
    for (auto& [key, value] : m) {
        cout << format("{} (Phone: {})", key, value) << endl;
    }
    cout << "-----" << endl;
}

int main()
{
    // Create a hash table.
    unordered_map<string, string> phoneBook {
        { "Marc G.", "123-456789" },
        { "Scott K.", "654-987321" } };
}

```

```

printMap(phoneBook);

// Add/remove some phone numbers.
phoneBook.insert(make_pair("John D.", "321-987654"));
phoneBook["Johan G."] = "963-258147";
phoneBook["Freddy K."] = "999-256256";
phoneBook.erase("Freddy K.");
printMap(phoneBook);

// Find the bucket index for a specific key.
const size_t bucket { phoneBook.bucket("Marc G.") };
cout << format("Marc G. is in bucket {} containing the following {} names:",
    bucket, phoneBook.bucket_size(bucket)) << endl;
// Get begin and end iterators for the elements in this bucket.
// 'auto' is used here. The compiler deduces the type of
// both as unordered_map<string, string>::const_local_iterator
auto localBegin { phoneBook.cbegin(bucket) };
auto localEnd { phoneBook.cend(bucket) };
for (auto iter { localBegin }; iter != localEnd; ++iter) {
    cout << format("{} (Phone: {})", iter->first, iter->second) << endl;
}
cout << "-----" << endl;

// Print some statistics about the hash table
cout << format("There are {} buckets.", phoneBook.bucket_count()) << endl;
cout << format("Average number of elements in a bucket is {}.",
    phoneBook.load_factor()) << endl;
}

```

这段代码的可能输出结果如下所示。注意，在不同的系统上，输出可能不同，因为它取决于所用哈希函数和 `unordered_map` 自身的实现。

```

Scott K. (Phone: 654-987321)
Marc G. (Phone: 123-456789)
-----
Scott K. (Phone: 654-987321)
Marc G. (Phone: 123-456789)
Johan G. (Phone: 963-258147)
John D. (Phone: 321-987654)
-----
Marc G. is in bucket 1 which contains the following 2 elements:
    Scott K. (Phone: 654-987321)
    Marc G. (Phone: 123-456789)
-----
There are 8 buckets.
Average number of elements in a bucket is 0.5

```

18.5.3 unordered_multimap

`unordered_multimap` 是允许多个元素带有同一个键的 `unordered_map`。两者的接口几乎相同，区别在于：

- `unordered_multimap` 没有提供 `operator[]` 运算符和 `at()`，它们的语义在多个元素使用同一个键的情况下没有意义。

- 在 `unordered_multimap` 上执行插入操作总是会成功。因此，添加单个元素的 `unordered_multimap::insert()` 方法只返回迭代器而非 `pair`。
- `unordered_map` 支持 `insert_or_assign()` 和 `try_emplace()` 方法，但是，`unordered_multimap` 不支持这两个方法。

注意：

`unordered_multimap` 允许插入相同的键值对。如果想要避免这种冗余，必须在插入新元素之前执行显式的检查。

根据之前对 `multimap` 的描述，不能使用 `operator[]` 运算符在 `unordered_multimap` 中查找元素，因为没有提供这个运算符。`find()` 虽然可供使用，但它返回的是引用具有给定键的任意一个元素的迭代器（未必是具有这个键的第 1 个元素）。相反，最好使用 `equal_range()` 方法，它返回一个包含两个迭代器的 `pair`：一个引用匹配给定键的第 1 个元素，另一个引用匹配给定键的最后 1 个元素之后的那个元素（one-past-the-last）。`equal_range()` 的用法和之前讨论 `multimap` 的 `equal_range()` 完全一样，因此可以参考 `multimap` 的示例来了解 `equal_range()` 的工作方式。

18.5.4 unordered_set/unordered_multiset

`<unordered_set>` 头文件定义了 `unordered_set` 和 `unordered_multiset`，这两者分别类似于 `set` 和 `multiset`；区别在于它们不会对键进行排序，而且使用了哈希函数。`unordered_set` 和 `unordered_map` 的区别和之前讨论的 `set` 和 `map` 之间的区别类似，因此这里不再赘述。标准库参考资源完整总结了 `unordered_set` 和 `unordered_multiset` 操作。

18.6 其他容器

C++ 语言中还有其他一些在不同程度上与标准库合作的部分，包括标准 C 风格数组、`string`、流和 `bitset`。

18.6.1 标准 C 风格数组

回顾一下，普通指针也算是迭代器，因为它们支持所需的运算符。这一点并不是琐碎的小知识。它意味着可以把标准 C 风格数组看成标准库容器，只要把指向数组元素的指针当成迭代器即可。当然，标准 C 风格数组并没有提供 `size()`、`empty()`、`insert()` 和 `erase()` 这类方法，因此它们并非真正的标准库容器。不管怎么样，它们通过指针的方式支持迭代器，因此可以在第 20 章描述的算法和本章描述的一些方法中使用它们。

例如，可通过 `vector` 中接收任何容器迭代器范围的 `insert()` 方法，将标准 C 风格数组中的所有元素复制到 `vector` 中。这个 `insert()` 方法的原型如下所示：

```
template <class InputIterator> iterator insert(const_iterator position,
                                             InputIterator first, InputIterator last);
```

如果想用标准的 C 风格 `int` 数组作为数据来源，那么可以将 `InputIterator` 的模板化类型替换为 `int*`。下面是一个完整的示例：

```
const size_t count { 10 };
int values[count]; // standard C-style array
// Initialize each element of the array to the value of its index.
```

```

for (int i { 0 }; i < count; i++) { values[i] = i; }

// Insert the contents of the array at the end of a vector.
vector<int> vec;
vec.insert(end(vec), values, values + count);

// Print the contents of the vector.
for (const auto& i : vec) { cout << i << " "; }

```

注意，引用数组中第 1 个元素的迭代器是第 1 个元素的地址，也就是本例中的 `values`。数组名字本身可解释为第 1 个元素的地址。尾部的迭代器所引用的元素必须是最后一个元素之后的那个元素 (one-past-the-last element)，因此地址是第 1 个元素加 `count` 的结果，即 `values+count`。

很容易使用 `std::begin()` 或 `std::cbegin()` 获得指向静态分配的 C 风格数组 (不通过指针访问) 中第 1 个元素的迭代器，使用 `std::end()` 或 `std::cend()` 获得此类数组中最后一个元素之后那个元素的迭代器。例如，前面示例中对 `insert()` 的调用可以写为：

```
vec.insert(end(vec), cbegin(values), cend(values));
```

警告：

`std::begin()` 和 `std::end()` 等函数仅用于静态分配的 C 风格数组 (不通过指针访问)。如果涉及指针或使用动态分配的 C 风格数组，则不可行。

18.6.2 string

可以将 `string` 看成字符的顺序容器。因此，C++ 中的 `string` 实际上是一个功能完备的顺序容器。`string` 中包含的 `begin()` 和 `end()` 方法返回 `string` 中的迭代器，还包含 `insert()`、`push_back()`、`erase()`、`size()` 和 `empty()` 方法，以及基本顺序容器包含的其他所有内容。`string` 非常接近于 `vector`，甚至还提供了 `reserve()` 和 `capacity()` 方法。

可以像使用 `vector` 那样将 `string` 作为标准库容器使用。示例如下：

```

string myString;
myString.insert(cend(myString), 'h');
myString.insert(cend(myString), 'e');
myString.push_back('l');
myString.push_back('l');
myString.push_back('o');

for (const auto& letter : myString) {
    cout << letter;
}
cout << endl;

for (auto it { cbegin(myString) }; it != cend(myString); ++it) {
    cout << *it;
}
cout << endl;

```

除了标准库顺序容器方法外，`string` 还提供了很多有用的方法和友元函数。`string` 接口实际上是一个很好的示例，它是第 6 章“设计可重用代码”中讨论的设计缺陷之一。`string` 类将在第 2 章详细讨论。

18.6.3 流

传统意义上，输入流和输出流并不是容器，因为它们并不保存元素。然而，可以把它们看成元素的序列，因而具有标准库容器的一些特性。C++流没有直接提供与标准库相关的任何方法，但是标准库提供了名为 `istream_iterator` 和 `ostream_iterator` 的特殊迭代器，用于“遍历”输入流和输出流。第 17 章将会介绍这些迭代器的用法。

18.6.4 bitset

`bitset` 是固定长度的位序列的抽象。一个位只能表示两个值——1 和 0，这两个值可以表示开/关和真/假等。`bitset` 还使用了设置(`set`)和清零(`unset`)两个术语。可将一个位从一个值切换(`toggle`)或翻转(`flip`)为另一个值。

`bitset` 并不是真正的标准库容器：`bitset` 的大小固定，没有对元素类型进行模板化，也不支持迭代。然而，这是一个有用的工具类，而且常和容器在一起，因此这里做一下简要介绍。标准库参考资源对 `bitset` 操作做了全面总结。

1. bitset 基础

`bitset` 定义在 `<bitset>` 头文件中，根据保存的位数进行模板化。默认构造函数将 `bitset` 的所有字段初始化为 0。另一个构造函数根据由 0 和 1 字符组成的字符串创建 `bitset`。

可通过 `set()`、`reset()` 和 `flip()` 方法改变单个位的值，通过重载的 `operator[]` 运算符可以访问和设置单个字段的值。注意，对非 `const` 对象应用 `operator[]` 会返回一个代理对象，可为这个代理对象赋予一个布尔值，调用 `flip()` 或 `~` 取反。还可通过 `test()` 方法访问单个字段。`bitset` 以包含 0 和 1 字符的字符串形式进行流式处理。此外，通过普通的插入和抽取运算符可以流式处理 `bitset`。

最后，通过普通的插入和抽取运算符可以流式处理 `bitset`。`bitset` 作为包含 0 和 1 字符的字符串进行流处理。

下面是一个简单的示例：

```
bitset<10> myBitset;

myBitset.set(3);
myBitset.set(6);
myBitset[8] = true;
myBitset[9] = myBitset[3];

if (myBitset.test(3)) {
    cout << "Bit 3 is set!" << endl;
}
cout << myBitset << endl;
```

输出结果为：

```
Bit 3 is set!
1101001000
```

注意，所输出字符串的最左边字符表示最高位。这符合我们对二进制数表示方式的直观看法，表示 $2^0=1$ 的最低位出现在印刷表示方式的最右边。

2. 按位运算符

除基本的位操作外，`bitset` 还实现了所有按位运算符：`&`、`|`、`^`、`~`、`<<`、`>>`、`&=`、`|=`、`^=`、`<<=` 和 `>>=`。这些运算符的行为和操作真正的位序列相同。示例如下：

```
auto str1 { "0011001100" };
auto str2 { "0000111100" };
bitset<10> bitsOne { str1 };
bitset<10> bitsTwo { str2 };

auto bitsThree { bitsOne & bitsTwo };
cout << bitsThree << endl;
bitsThree <<= 4;
cout << bitsThree << endl;
```

这个程序的输出结果如下所示：

```
0000001100
0011000000
```

3. `bitset` 示例：表示有线电视频道

`bitset` 的一种可能应用是跟踪有线电视用户的频道。每个用户都有一组用 `bitset` 表示的频道，这个 `bitset` 与用户的订阅情况相关，设置的位表示用户实际订阅的频道。这个系统还可以支持频道“套餐”，套餐也表示为 `bitset`，通过 `bitset` 表示常用的频道组合。

下面的 `CableCompany` 类是这个模型的简单示例。这个类使用了两个 `map`，它们都是 `string/bitset` 的 `map`，保存了有线频道套餐和用户信息。

```
export class CableCompany final
{
public:
    // Number of supported channels.
    static const size_t NumChannels { 10 };

    // Adds package with the channels specified as a bitset to the database.
    void addPackage(std::string_view packageName,
        const std::bitset<NumChannels>& channels);
    // Adds package with the channels specified as a string to the database.
    void addPackage(std::string_view packageName, std::string_view channels);
    // Removes the specified package from the database.
    void removePackage(std::string_view packageName);
    // Retrieves the channels of a given package.
    // Throws out_of_range if the package name is invalid.
    const std::bitset<NumChannels>& getPackage(
        std::string_view packageName) const;
    // Adds customer to database with initial channels found in package.
    // Throws out_of_range if the package name is invalid.
    // Throws invalid_argument if the customer is already known.
    void newCustomer(std::string_view name, std::string_view package);
    // Adds customer to database with given initial channels.
    // Throws invalid_argument if the customer is already known.
    void newCustomer(std::string_view name,
        const std::bitset<NumChannels>& channels);
    // Adds the channel to the customer's profile.
    // Throws invalid_argument if the customer is unknown.
    void addChannel(std::string_view name, int channel);
```

```

        // Removes the channel from the customer's profile.
        // Throws invalid_argument if the customer is unknown.
        void removeChannel(std::string_view name, int channel);
        // Adds the specified package to the customer's profile.
        // Throws out_of_range if the package name is invalid.
        // Throws invalid_argument if the customer is unknown.
        void addPackageToCustomer(std::string_view name,
                                   std::string_view package);
        // Removes the specified customer from the database.
        void deleteCustomer(std::string_view name);
        // Retrieves the channels to which a customer subscribes.
        // Throws invalid_argument if the customer is unknown.
        const std::bitset<NumChannels>& getCustomerChannels(
            std::string_view name) const;
    private:
        // Retrieves the channels for a customer. (non-const)
        // Throws invalid_argument if the customer is unknown.
        std::bitset<NumChannels>& getCustomerChannelsHelper(
            std::string_view name);

        using MapType = std::map<std::string, std::bitset<NumChannels>>;
        MapType m_packages, m_customers;
};

```

下面是上述方法的实现，其中包含代码注释：

```

void CableCompany::addPackage(string_view packageName,
                              const bitset<NumChannels>& channels)
{
    m_packages.emplace(packageName, channels);
}

void CableCompany::addPackage(string_view packageName, string_view channels)
{
    addPackage(packageName, bitset<NumChannels> { channels.data() });
}

void CableCompany::removePackage(string_view packageName)
{
    m_packages.erase(packageName.data());
}

const bitset<CableCompany::NumChannels>& CableCompany::getPackage(
    string_view packageName) const
{
    // Get a reference to the specified package.
    if (auto it { m_packages.find(packageName.data()) }; it != end(m_packages)) {
        // Found package. Note that 'it' is a reference to a name/bitset pair.
        // The bitset is the second field.
        return it->second;
    }
    throw out_of_range { "Invalid package" };
}

void CableCompany::newCustomer(string_view name, string_view package)
{
    // Get the channels for the given package.
    auto& packageChannels { getPackage(package) };
    // Create the account with the bitset representing that package.
    newCustomer(name, packageChannels);
}

void CableCompany::newCustomer(string_view name,

```

```

    const bitset<NumChannels>& channels)
{
    // Add customer to the customers map.
    if (auto [iter, success] { m_customers.emplace(name, channels) }; !success) {
        // Customer was already in the database. Nothing changed.
        throw invalid_argument { "Duplicate customer" };
    }
}

void CableCompany::addChannel(string_view name, int channel)
{
    // Get the current channels for the customer.
    auto& customerChannels { getCustomerChannelsHelper(name) };
    // We found the customer; set the channel.
    customerChannels.set(channel);
}

void CableCompany::removeChannel(string_view name, int channel)
{
    // Get the current channels for the customer.
    auto& customerChannels { getCustomerChannelsHelper(name) };
    // We found this customer; remove the channel.
    customerChannels.reset(channel);
}

void CableCompany::addPackageToCustomer(string_view name, string_view package)
{
    // Get the channels for the given package.
    auto& packageChannels { getPackage(package) };
    // Get the current channels for the customer.
    auto& customerChannels { getCustomerChannelsHelper(name) };
    // Or-in the package to the customer's existing channels.
    customerChannels |= packageChannels;
}

void CableCompany::deleteCustomer(string_view name)
{
    m_customers.erase(name.data());
}

const bitset<CableCompany::NumChannels>& CableCompany::getCustomerChannels(
    string_view name) const
{
    // Find a reference to the customer.
    if (auto it { m_customers.find(name.data()) }; it != end(m_customers)) {
        // Found customer. Note that 'it' is a reference to a name/bitset pair.
        // The bitset is the second field.
        return it->second;
    }
    throw invalid_argument { "Unknown customer" };
}

bitset<CableCompany::NumChannels>& CableCompany::getCustomerChannelsHelper(
    string_view name)
{
    // Forward to const getCustomerChannels() to avoid code duplication.
    return const_cast<bitset<NumChannels>&>(&{
        as_const(*this).getCustomerChannels(name));
}

```

最后，下面这个简单程序演示了如何使用 `CableCompany` 类：

```
CableCompany myCC;
```

```
myCC.addPackage("basic", "1111000000");
myCC.addPackage("premium", "1111111111");
myCC.addPackage("sports", "0000100111");

myCC.newCustomer("Marc G.", "basic");
myCC.addPackageToCustomer("Marc G.", "sports");
cout << myCC.getCustomerChannels("Marc G.") << endl;
```

输出如下所示:

```
1111100111
```

18.7 本章小结

本章介绍了标准库容器,还列举了示例代码来演示这些容器的不同使用方式。希望读者能体会到 vector、deque、list、forward_list、array、span、stack、queue、priority_queue、map、multimap、set、multiset、unordered_map、unordered_multimap、unordered_set、unordered_multiset、string 和 bitset 的强大之处。即使不能立即将这些容器用于自己的程序,也至少要知道有这些容器,以便在未来的项目中使用。

在下一步讨论标准库的泛型算法,并深入领略其真正的强大功能之前,首先必须要讨论函数指针、函数对象和 lambda 表达式。这些都是下一章的主题。

18.8 练习

通过完成以下习题,可以巩固本章中讨论的知识点。所有练习的答案都可扫描封底二维码下载。然而,如果你困在一个练习中,在网站上查看答案之前,请先重新阅读本章的内容,试着自己找到答案。

练习 18-1 这道练习是学习使用 vector。创建一个程序,该程序包含被称为 values 的整型 vector,使用数字 2 和 5 进行初始化。接下来,实现以下操作:

- (1) 使用 insert() 在 values 的正确位置插入数字 3 和 4。
- (2) 创建使用 0 和 1 初始化的第 2 个整型 vector,然后将这个新 vector 的内容插入 values 的开头。
- (3) 创建第 3 个整型 vector。反向遍历 values 的元素,并将它们插入第 3 个 vector 中。
- (4) 使用基于范围的 for 循环打印第 3 个 vector 中的内容。

练习 18-2 使用练习 15-4 中 Person 类的实现。添加一个名为 phone_book 的新模块,定义 PhoneBook 类,该类将一个或多个电话号码存储为个人的字符串。提供向电话簿中添加和删除个人电话号码的方法。还提供了一种方法,该方法返回包含给定人员的所有电话号码的 vector。在 main() 函数中测试具体的实现。在测试中,使用在练习 15-4 中开发的自用户定义的 person 字面量。

练习 18-3 在练习 15-1 中,编写了 AssociativeArray。修改该练习 main() 中的测试代码,以便使用任意一个标准库容器。

练习 18-4 编写一个 average() 函数(不是函数模板),用于计算一系列 double 的平均值。确保它可以适用于来自 vector 或 array 的序列或子序列。在 main() 函数中使用 vector 或 array 测试编写的代码。

附加练习 可以把 average() 函数转换成函数模板吗? 函数模板只能使用整数类型或浮点类型进行实例化。请问,这对 main() 中的测试代码会有什么影响?

第 19 章

函数指针，函数对象， lambda 表达式

本章内容

- 如何使用函数指针
- 如何使用指向类方法的指针
- 如何使用多态函数封装
- 函数对象定义以及如何编写自己的函数对象
- lambda 表达式细节

C++中的函数被称为一级函数，因为函数可以像普通变量一样使用，例如将它们作为参数传递给其他函数，从其他函数返回它们，将它们赋值给变量。在这个上下文中经常出现的术语叫回调，它表示可以调用的东西，它可以是函数指针或任何行为类似于函数指针的东西，比如重载了 `operator()` 的对象，或内联 lambda 表达式。重载 `operator()` 的类称为函数对象，简称 functor。标准库提供了一组可用于创建回调对象和调整现有回调对象的类，这很方便。是时候仔细看看回调的概念了，因为下一章中解释的很多算法都接收这样的回调来定制行为。

19.1 函数指针

通常不考虑函数在内存中的位置，但每个函数实际上都位于一个特定的地址。在 C++中，可以将函数作为数据，就是说 C++具有一级函数，换句话说，可以拿到函数的地址并且像使用变量一样使用它们。

函数指针根据参数类型和兼容函数的返回类型进行类型划分，使用函数指针的一种方法是使用类型别名，类型别名允许将类型名称赋值给具有给定特征的函数族。举例，下面一行定义了一种名为 `Matcher` 的类型，它表示指向任何具有两个 `int` 形参并返回 `bool` 的函数的指针：

```
using Matcher = bool (*)(int, int);
```

下面的类型别名定义了一种名为 `MatchHandler` 的类型，用于接收一个 `size_t` 和两个 `int` 型作为参

数和没有返回值的函数。

```
using MatchHandler = void(*) (size_t, int, int);
```

既然定义了这些类型，可以编写一个接收两个回调(Matcher 和 MatchHandler)作为参数的函数。接收其他函数作为参数的函数，或返回函数的函数称为高阶函数。例如，下面的函数接收两个 `int` 型的 `span`，以及一个 `Matcher` 和一个 `MatchHandler`，它并行地遍历这个 `span`，并在两个 `span` 的对应元素上调用 `Matcher`。如果 `Matcher` 返回 `true`，`MatchHandler` 会被调用，第一个参数是匹配的位置，第二个和第三个参数是使得 `Matcher` 返回 `true` 的值。注意，即使 `Matcher` 和 `MatchHandler` 作为变量传入，它们也可以像普通函数一样被调用：

```
void findMatches(span<const int> values1, span<const int> values2,
    MatchHandler matcher, MatchHandler handler)
{
    if (values1.size() != values2.size()) { return; } // Must be same size.

    for (size_t i { 0 }; i < values1.size(); ++i) {
        if (matcher(values1[i], values2[i]) {
            handler(i, values1[i], values2[i]);
        }
    }
}
```

注意，这个实现要求两个 `span` 具有相同数量的元素。要调用 `findMatches()` 函数，需要的是任何遵循已定义的 `Matcher` 类型的函数，即接收两个 `int` 型参数并返回 `bool` 类型的任何类型，以及一个遵循 `MatchHandler` 类型的函数。下面是一个可能为 `Matcher` 的示例，如果两个参数相等则返回 `true`。

```
bool intEqual(int item1, int item2) { return item1 == item2; }
```

下面是一个简单打印匹配的 `MatchHandler` 的示例：

```
void printMatch(size_t position, int value1, int value2)
{
    cout << format("Match found at position {}({}, {})",
        position, value1, value2) << endl;
}
```

`intEqual()` 和 `printMatch()` 函数可以作为参数传递给 `findMatches()`，如下：

```
vector values1 { 2, 5, 6, 9, 10, 1, 1 };
vector values2 { 4, 4, 2, 9, 0, 3, 1 };
cout << "Calling findMatches() using intEqual():" << endl;
findMatches(values1, values2, &intEqual, &printMatch);
```

回调函数通过获取它们的地址传递给 `findMatches()`。从技术上讲，`&` 字符是可选的，如果省略它们而只传入函数名，编译器就会知道你想要的是获取它们的地址，输出如下：

```
Calling findMatches() using intEqual():
Match found at position 3 (9, 9)
Match found at position 6 (1, 1)
```

函数指针的好处在于，`findMatches` 是个泛型函数，它比较两个 `vector` 中的并行的值。正如前面示例中使用到的，它基于相等来比较值。但因为它接收一个函数指针，所以它可以根据其他条件比较值。例如，下面的函数也遵循了 `Matcher` 的定义：

```
bool bothOdd(int item1, int item2) { return item1 % 2 == 1 && item2 % 2 == 1; }
```

下面的代码片段展示了 `bothOdd()` 也可以在 `findMatches()` 的调用中使用：

```
cout << "Calling findMatches() using bothOdd():" << endl;
findMatches(values1, values2, bothOdd, printMatch);
```

输出如下：

```
Calling findMatches() using bothOdd():
Match found at position 3 (9, 9)
Match found at position 5 (1, 3)
Match found at position 6 (1, 1)
```

通过使用函数指针，可以基于作为参数传入的函数对单个函数 `findMatches()` 进行定制，来适应不同的用途。

注意：

在本章的后面将提到 `std::function`，这是比旧式函数指针更被推荐的解决方案。

使用函数指针的一个常见用例是获取指向动态链接库中的函数的指针。下面的示例获取指向 Microsoft Windows 动态链接库(DLL)中的函数的指针，DLL 大体上是个由代码和数据组成的库，任何程序都可以使用它。一个特殊的 Windows DLL 示例是 `Comdlg32 DLL`，它提供了常见的对话框函数。关于 Windows DLL 的细节超出了这本平台不相关的 C++ 书籍的讨论范畴，但是它对 Windows 程序员来说非常重要，值得简要讨论，而且它是函数指针的一个很好的例子。

考虑一个 `DLL(hardware.dll)`，它有一个名为 `Connect()` 的函数，只有在需要调用 `Connect()` 时才会加载这个库，在运行时加载库是通过 Windows 的 `LoadLibrary()` 函数实现的：

```
HMODULE lib { ::LoadLibrary("hardware.dll") };
```

调用的结果称为库句柄，如果出现错误，句柄为 `NULL`。在加载库中的函数之前，需要知道函数的原型。假设下面是 `Connect()` 的原型，它返回一个整型并接收 3 个参数：布尔型、整型和 C 风格字符串：

```
int __stdcall Connect(bool b, int n, const char* p);
```

`__stdcall` 是一个 Microsoft 特定的指令，用于指定函数调用约定，规定如何将参数传递给函数以及如何清理函数。

可以使用类型别名定义函数指针的名称(`ConnectFunction`)，该函数具有要求的原型：

```
using ConnectFunction = int(__stdcall*)(bool, int, const char*);
```

成功加载库并且定义了函数指针的名称后，就可以得到一个指向库中函数的指针，如下所示：

```
ConnectFunction connect { (ConnectFunction)::GetProcAddress(lib, "Connect") };
```

如果失败，`connect` 会为 `nullptr`；如果成功，可以调用加载的函数：

```
connect(true, 3, "Hello world");
```

19.2 指向方法(和数据成员)的指针

如前一节所述，可以创建和使用指向独立函数的指针。也可以使用指向独立变量的指针，现在考虑指向类数据成员和方法的指针。在 C++ 中，拿到类数据成员和方法的地址来获得指向它们的指针是

完全合法的，但是，没有对象就不能访问非静态数据成员或调用非静态方法。类数据成员和方法的全部意义在于它们以每个对象为基础存在，因此，当想要通过指针调用方法或者访问数据成员时，必须在对象的上下文中解引用该指针。下面是第 1 章介绍的 `Employee` 类的一个示例：

```
int (Employee::*methodPtr) () const { &Employee::getSalary };
Employee employee { "John", "Doe" };
cout << (employee.*methodPtr)() << endl;
```

不要为语法恐慌，第一行声明了一个名为 `methodPtr` 的变量，它的类型是指向 `Employee` 的一个非静态 `const` 方法的指针，该方法不接收任何参数并返回 `int` 型。同时，它初始化这个变量，使其指向 `Employee` 类的 `getSalary()` 方法。这种语法非常类似于声明一个简单的函数指针，除了在 `*methodPtr` 之前添加了 `Employee::`。注意在本例中必须使用 `&`。

第三行通过 `methodPtr` 指针调用了 `employee` 对象上的 `getSalary()` 方法，注意在 `employee.*methodPtr` 周围括号的使用，之所以需要括号，是因为 `operator()` 的优先级高于 `*`。

如果有一个指向对象的指针，可以使用 `->*` 而不是 `*`，如下面的代码片段说明：

```
int (Employee::*methodPtr) () const { &Employee::getSalary };
Employee* employee { new Employee { "John", "Doe" } };
cout << (employee->*methodPtr)() << endl;
```

使用类型别名可以让 `methodPtr` 的定义更易阅读：

```
using PtrToGet = int (Employee::*) () const;
PtrToGet methodPtr { &Employee::getSalary };
Employee employee { "John", "Doe" };
cout << (employee.*methodPtr)() << endl;
```

最后，可以使用 `auto` 进一步简化：

```
auto methodPtr { &Employee::getSalary };
Employee employee { "John", "Doe" };
cout << (employee.*methodPtr)() << endl;
```

注意：

可以使用 `std::mem_fn()` 避免 `*或->*` 语法，本章后面的函数对象上下文中将会解释这点。

指向方法和数据成员的指针通常不会出现在程序中，但重要的是要记住，如果没有对象，就不能解引用指向非静态方法和数据成员的指针。每隔一段时间，你可能想要尝试一些事情，比如将一个指向非静态方法的指针传递给需要函数指针的函数(比如 `qsort`)，这是行不通的。

注意：

C++ 确实允许在不使用对象的情况下解引用指向静态数据成员和静态方法的指针。

std::function

在 `<functional>` 中定义的 `std::function` 函数模板是一个多态函数包装器，可以用来创建指向任何可调用对象的类型，如函数、函数对象或者 `lambda` 表达式，本章后面将会讨论后两项。`std::function` 的实例可以用作函数指针，也可以用作函数实现回调的参数，可以存储、赋值、移动，当然也可以执行。函数模板的模板参数看起来和大多数模板参数有些不同，语法如下：

```
std::function<R(ArgTypes...)>
```


R 是函数的返回类型, ArgTypes 是函数参数类型的逗号分隔列表。

下面的示例演示了如何使用 `std::function` 实现函数指针, 它创建一个指向函数 `func()` 的函数指针 `f1`。一旦定义了 `f1`, 就可以用它调用 `func()`:

```
void func(int num, string_view str)
{
    cout << format("func({}, {})", num, str) << endl;
}

int main()
{
    function<void(int, string_view)> f1 { func };
    f1(1, "test");
}
```

由于类模板参数推导, 可以简化 `f1` 的创建, 如下所示:

```
function f1 { func };
```

当然, 在前面的实例中可以只使用 `auto` 关键字, 这样就不需要指定 `f1` 的类型, 下面对 `f1` 的定义同样有效, 而且更短, 但是编译器推导出来的 `f1` 类型是个函数指针, 即 `void(*f1)(int, string_view)`, 而不是 `std::function`:

```
auto f1 { func };
```

因为 `std::function` 行为像函数指针, 它可以传递给接收回调的函数。前面的 `findMatches()` 示例可以使用 `std::function` 重写, 唯一需要更改的是下面两个类型别名:

```
// A type alias for a function accepting two integer values,
// returning true if both values are matching, false otherwise.
using Matcher = function<bool(int, int)>;

// A type alias for a function to handle a match. The first
// parameter is the position of the match,
// the second and third are the values that matched.
using MatchHandler = function<void(size_t, int, int)>;
```

这些类型的回调使许多标准库算法变得如此强大, 标准库算法将在第 20 章“掌握标准库算法”中讨论。

然而从技术上讲, `findMatches()` 不需要 `std::function` 参数来接收回调参数, 相反, 可以将 `findMatches()` 转换为函数模板, 唯一需要更改的是删除 `Matcher` 和 `MatchHandler` 类型别名, 使其成为函数模板。高亮部分为变化:

```
template<typename Matcher, typename MatchHandler>
void findMatcher(span<const int> values1, span<const int> values2,
    Matcher matcher, MatchHandler handler)
{ /* ... */ }
```

`findMatches()` 的实现需要两个模板类型参数, 即 `Matcher` 回调类型和 `MatchHandler` 回调类型, 由于函数模板参数的推导, 调用这个新版本与调用以前的版本效果是相同的。

使用 C++20 的简化函数模板语法, `findMatches()` 函数模板可以这样写:

```
void findMatches(span<const int> values1, span<const int> values2,
    auto matcher, auto handler)
{ /* ... */ }
```

`findMatches()`函数模板或简化函数模板实际上是实现它的推荐方法，因此在所有这些例子中，看起来 `std::function` 并不是真正有用，然而当需要将回调存储为类的数据成员时，`std::function` 真的很有用。这是本章最后练习的其中一个主题。

19.3 函数对象

可以重载类中的函数调用运算符，以使用类的对象代替函数指针。这些对象称为函数对象，或简称仿函数。使用函数对象而不是使用简单函数的好处是，函数对象可以在调用之间保持状态。

19.3.1 编写第一个函数对象

要使任何类成为函数对象，只需要重载函数调用运算符，下面是一个简单的函数对象示例：

```
class IsLargerThan
{
public:
    explicit IsLargerThan(int value) : m_value { value } {}
    bool operator()(int value1, int value2) const {
        return value1 > m_value && value2 > m_value;
    }
private:
    int m_value;
};

int main()
{
    vector values1 { 2, 500, 6, 9, 10, 101, 1 };
    vector values2 { 4, 5, 2, 9, 0, 300, 1 };
    findMatches(values1, values2, IsLargerThan { 100 }, printMatch);
}
```

注意，`IsLargerThan` 类中的重载函数调用运算符标记为 `const`，在本例中这并不是严格要求的，但是下一章会解释，对于大多数标准库算法，谓词的函数调用运算符必须是 `const`。

19.3.2 标准库中的函数对象

下一章讨论的多数标准库算法，例如 `find_if()`、`accumulate()`等，接收回调(函数指针和仿函数)作为参数来自定义算法的行为。C++在`<functional>`中提供了几个预定义的函数类，它们执行最常用的回调操作，本节会概述这些预定义仿函数。

`<functional>`中可能还包含像 `bind1st()`、`bind2nd()`、`mem_fun()`、`mem_fun_ref()`和 `ptr_fun()`这样的函数，这些函数在 C++17 标准中已经被正式移除，因此本书不再进一步讨论，应该尽量避免使用它们。

1. 算术函数对象

C++为 5 种二元算术运算符提供了仿函数类模板：`plus`、`minus`、`multiplies`、`divides`、`modulus`，此外还提供了一元运算符 `negate`。这些类根据操作数的类型模板化，是实际运算符的包装器。它们接收模板类型的一或两个参数，执行运算并返回结果。下面是一个使用 `plus` 类模板的示例：

```
plus<int> myPlus;
int res { myPlus(4, 5) };
```

```
cout << res << endl;
```

这个示例当然很蠢, 因为可以直接使用 `operator+`, 没有理由使用 `plus` 类模板。算术函数对象的好处是可以将它们直接作为回调传递给其他函数, 直接使用算术运算符不能这样。例如下面的代码片段定义了接收 `Operation` 作为参数的函数模板 `accumulateData()`。`geometricMean()` 函数的实现调用了一个预定义的 `multiplies` 函数对象实例的 `accumulateData()` 方法:

```
template <typename Iter, typename StartValue, typename Operation>
auto accumulateData(Iter begin, Iter end, StartValue startValue, Operation op)
{
    auto accumulated { startValue };
    for (Iter iter { begin }; iter != end; ++iter) {
        accumulated = op(accumulated, *iter);
    }
    return accumulated;
}

double geometricMean(span<const int> values)
{
    auto mult {accumulateData(cbegin(values), cend(values), 1, multiplies<int>{})};
    return pow(mult, 1.0 / values.size()); // pow() requires <cmath>
}
```

表达式 `multiplies<int>()` 创建了 `multiplies` 仿函数类模板的新对象, 用 `int` 类型来实例化。其他算术函数对象的行为类似。

警告:

算术函数对象只是算术运算符的包装器, 要在特定类型的对象上使用它们, 必须确保这些类型实现了恰当的操作, 例如 `operator*` 或 `operator+`。

2. 透明运算符仿函数

C++ 支持透明运算符仿函数, 允许你忽略模板类型参数。例如可以指定 `multiplies<>()` 为 `multiplies<void>()` 而不是 `multiplies<int>()` 的缩写:

```
double geometricMeanTransparent(span<const int> values)
{
    auto mult { accumulateData(cbegin(values), cend(values), 1, multiplies<>()) };
    return pow(mult, 1.0 / values.size());
}
```

这些透明运算符的一个重要特征是异构。也就是说, 它们不仅比非透明运算符更简洁, 而且具有真正的功能性优势。举例来说, 下面的代码使用了透明运算符仿函数 `multiplies<>()`, 使用了 `double` 类型的 1.1 作为初始值, 而 `vector` 中包含整数。`accumulateData()` 计算的结果也是 `double` 型, 结果会是 6.6。

```
vector<int> values { 1, 2, 3 };
double result {accumulateData(cbegin(values), cend(values), 1.1, multiplies<>())};
```

如果这段代码使用了非透明运算符仿函数 `multiplies<int>()`, 那么 `accumulateData()` 计算的结果就会是 `int` 型, 即结果是 6。当编译这段代码时, 编译器会给出可能丢失数据的警告。

```
vector<int> values { 1, 2, 3 };
double result {
    accumulateData(cbegin(values), cend(values), 1.1, multiplies<int>()) };
}
```

最后，使用透明运算符而不是非透明运算符可以提高性能，在下一节的示例中会讲到。

注意：

推荐经常使用透明运算符仿函数。

3. 比较函数对象

除了算术函数对象类之外，所有标准比较运算符也可作为仿函数：equal_to、not_equal_to、less、greater、less_equal 和 greater_equal。第 18 章“标准库容器”中 less 作为 priority_queue 和关联容器中元素的默认比较运算，现在可以学习改变这个标准。下面是 priority_queue 使用默认比较运算符 std::less 的示例：

```
priority_queue<int> myQueue;
myQueue.push(3);
myQueue.push(4);
myQueue.push(2);
myQueue.push(1);
while (!myQueue.empty()) {
    cout << myQueue.top() << " ";
    myQueue.pop();
}
```

程序的输出如下：

```
4 3 2 1
```

如上所见，队列中的元素根据 less 比较运算按降序被移除，通过指定比较模板参数，可以将其更改为 greater。priority_queue 模板的定义如下：

```
template <class T, class Container = vector<T>, class Compare = less<T>>;
```

但是，Compare 类型参数是最后一个，意味着必须指定它，也必须指定容器类型。如果想使用一个通过 greater 进行升序排列的 priority_queue，需要将前面例子中的 priority_queue 的定义修改如下：

```
priority_queue<int, vector<int>, greater<>> myQueue;
```

输出如下：

```
1 2 3 4
```

注意 myQueue 使用透明运算符 greater 定义。事实上，对于接收比较器类型的标准库容器来说，建议经常使用透明运算符，使用透明比较器相对于不透明运算符可以获得更好的性能。例如，如果 set<string> 使用了不透明比较器，对于给定的字符串字面量执行查询会导致产生一次没有必要的拷贝，因为字符串实例必须从字符串字面量构造：

```
set<string> mySet;
auto i1 { mySet.find("Key") }; // string constructed, allocates memory!
//auto i2 { mySet.find("Key"sv) }; // Compilation error!
```

当使用透明比较器时，可以避免这种复制，这称为异构查找，这里有个示例：

```
set<string, less<>> mySet;
auto i1 { mySet.find("Key") }; // No string constructed, no memory allocated.
auto i2 { mySet.find("Key"sv) }; // No string constructed, no memory allocated.
```

C++20

C++20 增加了对无序关联容器的透明运算符的支持, 比如 `unordered_map` 和 `unordered_set`。与有序关联容器使用透明运算符相比, 对无序关联容器使用透明运算符更复杂一些。基本上, 需要实现一个自定义哈希函数, 包含一个定义为 `void` 的 `is_transparent` 类型别名。

```
class Hasher
{
public:
    using is_transparent = void;
    size_t operator()(string_view sv) const { return hash<string_view>{}(sv); }
};
```

当使用自定义哈希时, 需要指定透明的 `equal_to<>` 仿函数作为键相等模板类型参数的类型, 如下例:

```
unordered_set<string, Hasher, equal_to<>> mySet;
auto i1 { mySet.find("Key") }; // No string constructed, no memory allocated.
auto i2 { mySet.find("Key"sv) }; // No string constructed, no memory allocated.
```

4. 逻辑函数对象

对于 3 种逻辑运算, 运算符 `!`、`&&`、`||` 提供了下面的函数对象类: `logical_not`、`logical_and` 和 `logical_or`。这些逻辑操作只处理值 `true` 和 `false`, 位函数对象会在下一节介绍。

例如, 逻辑仿函数可以用于实现 `allTrue()` 函数, 该函数检查容器中的所有布尔值标志是否为 `true`:

```
bool allTrue(const vector<bool>& flags)
{
    return accumulateData(begin(flags), end(flags), true, logical_and<>{});
}
```

类似地, `logical_or` 仿函数可以用来实现 `anyTrue()` 函数, 如果容器中至少有一个布尔值标志为 `true`, 则返回 `true`:

```
bool allTrue(const vector<bool>& flags)
{
    return accumulateData(begin(flags), end(flags), true, logical_or<>{});
}
```

注意:

`allTrue()` 和 `anyTrue()` 函数只是作为示例给出, 实际上, 标准库提供了 `std::all_of()` 和 `std::any_of()` 算法(见第 20 章), 它们执行相同的操作, 但是具有短路的优点(见第 1 章), 因此性能更好。

5. 位函数对象

C++ 具有与位操作运算符 `&`、`|`、`^` 和 `~` 相对应的函数对象 `bit_and`、`bit_or`、`bit_xor` 和 `bit_not`。例如, 这些位仿函数可以和 `transform()` 算法(在第 20 章讨论)一起使用, 对容器中的所有元素执行位操作。

6. 适配器函数对象

当尝试使用标准提供的基本函数对象时, 经常感觉就像是把一个方形的钉子放在一个圆形的洞里。如果想使用标准函数对象中的一个, 但是签名却不完全符合需求, 可以尝试使用适配器函数对象来纠正签名不匹配的问题。它们允许修改函数对象、函数指针和任何本质上可调用的对象。适配器提供了对组合功能少量的支持, 也就是说, 将功能组合在一起创建需要的确切行为。

绑定器

绑定器可用于将可调用对象的参数绑定到某些值。为此可以使用定义在<functional>中的 `std::bind()`，它允许以灵活的方式绑定可调用对象的参数，可以将参数绑定到固定值，甚至可以不同的顺序重新排列参数。最后用一个例子来解释，假设一个函数 `func()`接收两个参数：

```
void func(int num, string_view str)
{
    cout << format("func({}, {})", num, str) << endl;
}
```

下面的代码演示了如何使用 `bind()`将 `func()`的第二个参数绑定到一个固定值 `myString`，结果存储在 `f1()`中。这里之所以使用 `auto` 关键字是因为 C++标准没有指定 `bind()`的返回类型，因此是特定于实现的。没有绑定到特定值的参数应指定为 `_1`、`_2`、`_3` 等，这些在 `std::placeholders` 名称空间中定义。在 `f1()`的定义中，`_1` 指定了在调用 `func()`时 `f1()`的第一个参数需要放在哪里，结果 `f1()`可以只用一个整数参数来调用：

```
string myString { "abc" };
auto f1 { bind(func, placeholders::_1, myString) };
f1(16);
```

输出如下：

```
func(16, abc)
```

如下面的代码所示，`bind()`还可以用于重新排列参数。`_2` 指定在调用 `func()`时 `f2()`的第二个参数需要放在哪里，换句话说，`f2()`的绑定意味着 `f2()`的第一个参数将称为 `func()`的第二个参数，而 `f2()`的第二个参数将称为 `func()` 的第一个参数：

```
auto f2 { bind(func, placeholders::_2, placeholders::_1) };
f2("Test", 32);
```

输出如下所示：

```
func(32, Test)
```

正如第 18 章讨论，<functional>定义了 `std::ref()`和 `cref()`辅助函数模板，它们可用于绑定非 `const` 引用和 `const` 引用。举个例子，假设有下面的函数：

```
void increment(int& value) { ++value; }
```

如果这样调用这个函数，`index` 的值会变成 1：

```
int index { 0 };
increment(index);
```

如果使用 `bind()`像下面这样调用它，`index` 的值不会增加，因为生成的是 `index` 的拷贝，这个拷贝的引用被绑定到 `increment()`函数的第一个参数中：

```
auto incr { bind(increment, index) };
incr();
```

使用 `std::ref()`传递正确的引用就会正确地增加 `index`：

```
auto incr { bind(increment, ref(index)) };
incr();
```

绑定参数与重载函数结合使用有一个小问题，假定有下面两个重载函数，一个接收整型，另一个接收浮点型：

```
void overloaded(int num) {}
void overloaded(float f) {}
```

如果想在这些重载函数上使用 `bind()`，需要显式指定要绑定两个重载中的哪一个，下面代码将无法通过编译：

```
auto f3 { bind(overload, placeholders::_1) }; // ERROR
```

如果想绑定接收浮点参数的重载函数的形参，需要使用以下语法：

```
auto f4 { bind((void (*)(float))overloaded, placeholders::_1) }; // OK
```

`bind()` 的另一个示例是使用前面定义过的 `findMatches()`，作为 `MatchHandler` 类的方法。例如假设下面的 `Handler` 类：

```
class Handler
{
public:
    void handleMatch(size_t position, int value1, int value2)
    {
        cout << format("Match found at position {} ({}), ({})",
            position, value1, value2) << endl;
    }
};
```

如何传递 `handleMatch()` 方法作为 `findMatches()` 的最后一个参数？这里的问题在于方法总是在对象的上下文中被调用。从技术上讲，类的每个方法都有隐式的第一个参数，包含一个指向对象实例的指针，并且可以在方法体中以 `this` 名称访问。因此我们的 `MatchHandler` 类型只接收 3 个参数：一个 `size_t` 和两个 `int`，会导致签名不匹配问题，解决办法是绑定这个隐式的第一个参数，如下所示：

```
Handler handler;
findMatches(values1, values2, intEqual, bind(&Handler::handleMatch, &handler,
    placeholders::_1, placeholders::_2, placeholders::_3));
```

可以使用 `bind()` 绑定标准函数对象的参数，例如可以将 `greater_equal` 的第二个参数绑定为经常与固定值进行比较：

```
auto greaterEqualTo100 { bind(greater_equal<>{}, placeholders::_1, 100) };
```

注意：

C++11 之前有 `bind2nd()` 和 `bind1st()`，这两个在 C++17 标准中被移除。可以使用本章后面讨论的 `lambda` 表达式或者 `bind()` 来代替。

否定器

`not_fn()` 是所谓的 `negator` (否定器)，类似于 `binder` (绑定器)，但它补充了可调用对象的结果。例如，如果想使用 `findMatches()` 查找不相等的值对时，可以像下面这样对 `intEqual()` 的结果应用 `not_fn()` 的 `negator` 适配器。

```
findMatches(values1, values2, not_fn(intEqual), printMatch);
```

`not_fn()` 仿函数对作为参数的可调用对象的每次调用结果进行补充。使用仿函数和适配器可能变

得复杂, 建议使用 `lambda` 表达式(本章后面讨论), 而不是仿函数。相对于仿函数, `lambda` 表达式方便写出更易于阅读和理解的代码。

注意:

`std::not_fn` 适配器在 C++17 中被引入, 在 C++17 前可以使用 `std::not1()` 和 `not2()` 适配器, 然而 `not1()` 和 `not2()` 在 C++17 中被弃用并且在 C++20 中被移除, 因此不再进一步讨论, 应该避免使用它们。

调用成员函数

可能希望将指向类方法的指针当作算法的回调, 例如, 假设有以下算法, 它从匹配特定条件的容器中输出字符串 `s`:

```
template <typename Matcher>
void printMatchingStrings(const vector<string>& strings, Matcher matcher)
{
    for (const auto& string : strings) {
        if (matcher(string)) { cout << string << " "; }
    }
}
```

通过 `string` 的 `empty()` 方法, 可以使用此算法打印所有的非空字符串 `s`, 然而如果只是把指向 `string::empty()` 的指针作为 `printMatchingString()` 的第二个参数传递, 算法就无法知道它接收到的是指向方法的指针, 而不是普通的函数指针或仿函数。调用方法指针的代码和调用普通函数指针的代码不同, 因为前者必须在对象的上下文中调用。

C++ 提供了名为 `mem_fn()` 的转换函数, 在将其传递给算法前, 可以使用方法指针调用该函数, 下面的示例演示了这一点, 并将其与 `not_fn()` 结合起来反转 `mem_fn()` 的结果, 注意必须指定方法指针为 `&string::empty`。&string:: 部分不是可选的。

```
vector<string> values { "Hello", "", "", "World", "!" };
printMatchingStrings(values, not_fn(mem_fn(&string::empty)));
```

`not_fn(mem_fn())` 生成一个函数对象, 作为 `printMatchingStrings()` 的回调函数。每次调用它时, 会对其参数调用 `empty()` 方法, 并反转结果。

注意:

`mem_fn()` 并不是实现所需行为的最直观方法, 建议使用 `lambda` 表达式(下一节讨论), 以一种更具可读性的方式实现它。

19.4 lambda 表达式

编写函数或者仿函数类, 给它一个和其他名称不冲突的命名, 然后使用这个名称, 这对于本质上简单的概念来说有相当大的开销。在这些情况下, 使用 `lambda` 表达式表示所谓的匿名(未命名)函数非常方便, `lambda` 表达式允许编写匿名内联函数。它的语法更简单, 可以使代码更紧凑更易于阅读, `lambda` 表达式更适合定义小点的回调内联传递给其他函数, 而不是在其他地方定义一个完整的函数对象, 并在其重载函数调用运算符中实现回调逻辑。这样所有的逻辑都保持在单独的位置上, 容易理解和维护, `lambda` 表达式可以接收参数, 可返回值, 可模板化, 可通过值或引用从其闭包范围内访问变量等, 很灵活, 但是首先一步步熟悉 `lambda` 表达式的语法。

19.4.1 语法

从一个简单的 lambda 表达式开始, 下面的示例定义了一个 lambda 表达式, 只将字符串写入控制台。lambda 表达式以称为 lambda 导入器的方括号[]开始, 后面跟着包含 lambda 表达式主体的花括号 {}, lambda 表达式被赋值给 auto 类型变量 basicLambda, 第二行使用普通函数调用语法执行 lambda 表达式。

```
auto basicLambda { []{ cout << "Hello from Lambda" << endl; } };
basicLambda();
```

输出如下所示:

```
Hello from Lambda
```

编译器自动将任何 lambda 表达式转换为函数对象, 也称为 lambda 闭包, 它具有唯一的编译器生成的命名。在本例中, lambda 表达式被转换成行为类似于下面函数对象的函数对象, 注意函数调用运算符是一个 const 方法, 返回类型是 auto, 方便编译器根据方法体自动推导出返回类型。

```
class CompilerGeneratedName
{
public:
    auto operator()() const { cout << "Hello from Lambda" << endl; }
};
```

编译器生成的 lambda 闭包名字可以是一些奇特的名字, 例如 __Lambda_17Za, 无法知道这个名字, 但幸运的是不需要知道它的名字。

lambda 表达式可以接收参数, 参数在圆括号之间指定, 多个参数用逗号分隔, 就像普通函数一样, 下面是一个名为 value 的参数的示例:

```
auto parametersLambda {
    [](int value){ cout << "The value is " << value << endl; } };
parametersLambda(42);
```

如果 lambda 表达式不接收任何参数, 可以指定空括号或者直接省略括号。

在编译器为此 lambda 表达式生成的函数对象中, 参数被简单地转换为重载函数调用运算符的参数:

```
class CompilerGeneratedName
{
public:
    auto operator()(int value) const {
        cout << "The value is " << value << endl; }
};
```

lambda 表达式可以返回值, 返回类型在箭头后面指定, 称为尾返回类型。下面的例子定义了 lambda 表达式, 它接收两个参数并返回它们的和:

```
auto returningLambda { [](int a, int b) -> { return a + b; } };
int sum { returningLambda(11, 22) };
```

返回类型可以忽略, 在这种情况下, 编译器根据与函数返回类型推导相同的规则来推导 lambda 表达式的返回类型(见第 1 章), 在前面的示例中, 返回类型可以省略, 如下所示。

```
auto returningLambda { [](int a, int b){ return a + b; } };
int sum { returningLambda(11, 22) };
```

lambda 表达式的闭包行为如下:

```
class CompilerGeneratedName
{
public:
    auto operator()(int a, int b) const { return a + b; }
};
```

返回类型推导会移除任何引用和 `const` 限定, 例如, 假设有下面的 `Person` 类:

```
class Person
{
public:
    Person(std::string name) : m_name { std::move(name) } { }
    const std::string& getName() const { return m_name; }
private:
    std::string m_name;
};
```

下面 lambda 表达式的返回类型会被推导为 `string`, 因此会生成 `person` 的名字的拷贝, 即使 `getName()` 返回的是 `const string&`:

```
[] (const Person& person) { return person.getName(); }
```

可以将尾返回类型与 `decltype(auto)` 结合使用, 使推导出来的类型与 `getName()` 的返回类型匹配, 即 `const string&`:

```
[] (const Person& person) -> decltype(auto) { return person.getName(); }
```

到目前为止, 本节中的 lambda 表达式被称为无状态表达式, 因为它们没有从闭包作用域捕获任何内容。lambda 表达式可以通过从闭包作用域捕获变量而具有状态, 例如, 下面的 lambda 表达式捕获变量数据, 以便在其主体中使用。

```
double data { 1.23 };
auto capturingLambda { [data]{ cout << "Data = " << data << endl; } }
```

方括号部分称为 lambda 捕获块, 捕获变量意味着该变量在 lambda 表达式主体中可用, 指定一个空的捕获块 `[]`, 意味着不从闭包作用域中捕获任何变量, 当向前面的示例一样在捕获块中写入变量的名称时, 那就是通过值捕获该变量。

捕获的变量变为 lambda 闭包的数据成员, 值捕获的变量被复制到仿函数的数据成员中, 这些数据成员与捕获的变量具有相同的 `const` 属性。在前面的 `capturingLambda` 示例中, 仿函数得到一个名为 `data` 的非 `const` 的数据成员, 因为捕获的变量 `data` 是非 `const`, 编译器生成的仿函数行为如下:

```
class CompilerGeneratedName
{
public:
    CompilerGeneratedName(const double& d) : data { d } {}
    auto operator()() const { cout << "Data = " << data << endl; }
private:
    double data;
};
```

在下例中, 仿函数获得一个名为 `data` 的 `const` 数据成员, 因为捕获的变量是 `const`。

```
const double data { 1.23 };
auto capturingLambda { [data]{ cout << "Data = " << data << endl; } };
```

如前所述, lambda 闭包有个重载的函数调用运算符, 它被默认标记为 `const`。这意味着, 即使在 lambda 表达式中按值捕获非 `const` 变量, lambda 表达式中也不能修改该拷贝。如果将 lambda 表达式指定为 `mutable`, 可以将函数调用运算符标记为非 `const`, 如下所示:

```
double data { 1.23 };
auto capturingLambda {
    [data] () mutable { data *= 2; cout << "Data = " << data << endl; } };
```

在这个例子中, 非 `const` 变量 `data` 通过值捕获, 因此仿函数获得一个非 `const` 数据成员, 它是 `data` 的拷贝。由于 `mutable` 关键字, 函数调用运算符被标记为非 `const`, 因此 lambda 表达式主体可以修改 `data` 的拷贝。注意, 如果指定了 `mutable`, 那就必须为参数指定圆括号, 即使它们是空的。

可以在变量的名称前面加一个 `&`, 表示通过引用捕获它, 下面的示例通过引用捕获变量 `data`, 这样 lambda 表达式可以在闭包作用域内直接修改 `data`:

```
double data { 1.23 };
auto capturingLambda { [&data]{ data *= 2; } }
```

当通过引用捕获变量时, 必须确保引用在 lambda 表达式执行期间是合法的。

有两种方法从闭包作用域捕获所有变量, 称为默认捕获:

- `[=]` 值捕获所有变量
- `[&]` 引用捕获所有变量

注意:

当使用默认捕获时, 通过值(`=`)或引用(`&`)。只有那些在 lambda 表达式中真正使用的变量才会被捕获, 未使用的变量不会被捕获。

通过可选的默认捕获指定一个捕获列表, 可以选择性地决定捕获哪些变量以及如何捕获。以 `&` 为前缀的变量会通过引用捕获, 没有前缀的变量会通过值捕获。如果存在, 默认捕获应该是捕获列表中的第一个元素, 并且是 `&` 或者 `=`, 下面是一些捕获块的示例:

- `[&x]` 只通过引用捕获 `x`, 没有其他内容。
- `[x]` 只通过值捕获 `x`, 没有其他内容。
- `[=, &x, &y]` 默认通过值捕获, 除了变量 `x` 和 `y` 通过引用捕获。
- `[&, x]` 默认通过引用捕获, 除了变量 `x` 通过值捕获。
- `[&x, &x]` 非法, 因为标识符不能重复。
- `[this]` 捕获当前对象, 在 lambda 表达式主体中, 即使不使用 `this->`, 也可以访问该对象, 需要确保所指向的对象在 lambda 表达式最后一次执行时始终存活。
- `[*this]` 捕获当前对象的拷贝, 这在执行 lambda 表达式且原始对象不再存活时非常有用。
- `[=, this]` 按值捕获所有内容, 并显式捕获 `this` 指针。在 C++20 之前, `[=]` 会隐式捕获 `this` 指针, 这个在 C++20 中已经被弃用, 如果需要, 需要显式捕获它。

下面是一些关于捕获块的注意事项:

- 如果默认指定了值捕获(`=`)或者引用捕获(`&`), 则不允许额外通过值或引用捕获特定的变量, 例如 `[=, x]` 和 `[&, &x]` 都无效。
- 对象的数据成员不能被捕获, 除非使用后面一节讨论的 lambda 捕获表达式。
- 当通过拷贝 `this` 指针 `[this]`, 或拷贝当前对象 `[*this]` 来捕获 `this` 时, lambda 表达式可以访问被捕获对象的所有公有、保护和私有数据成员和方法。

警告:

不建议使用默认捕获, 即使默认捕获只捕获那些在 `lambda` 表达式主体中真正使用的变量。通过使用 `=` 默认捕获, 可能会意外地导致高代价的拷贝。通过使用 `&` 默认捕获, 可能意外地在闭包作用域中修改变量, 建议明确指定想要捕获哪些变量以及捕获方式。

警告:

全局变量总是通过引用捕获, 即便要求值捕获! 例如, 在下面的代码片段中, 默认捕获用于按值捕获所有内容, 然而, 全局变量 `global` 是通过引用捕获的, 在执行 `lambda` 后它的值被更改。

```
int global { 42 };
int main()
{
    auto lambda { [=] { global = 2; } };
    lambda();
    // global now has the value 2!
}
```

此外, 不允许像下面这样显式捕获全局变量, 这会导致编译错误:

```
auto labmda { [global] { global = 2; } };
```

即使没有这些问题, 也不推荐使用全局变量。

`lambda` 表达式的完整语法如下:

```
[capture_block] <template_params> (parameters) mutable constexpr
noexcept_specifier attributes
-> return_type requires {body}
```

除了捕获块和主体, 其他都是可选的。

- **捕获块:** 也称为 `lambda` 导入器, 可以指定如何从闭包作用域中捕获变量, 并使其在 `lambda` 主体中可用。
- **模板参数(C++20):** 允许编写模板化 `lambda` 表达式, 本章后面会讨论。
- **参数:** 这是 `lambda` 表达式的参数列表, 只有当不需要任何参数且不指定 `mutable`、`constexpr`、`noexcept` 说明符、属性、返回类型或可选条款时, 才可以省略此列表, 这个参数列表和普通函数的参数列表相似。
- **mutable:** 将 `lambda` 表达式标记为 `mutable`, 看前面的例子。
- **constexpr:** 将 `lambda` 表达式标记为 `constexpr`, 它就可以在编译期计算。即使省略, 如果满足 `constexpr` 函数的约束, `lambda` 表达式也会隐式成为 `constexpr`。
- **noexcept 说明符:** 可用于指定 `noexcept` 条款, 类似于普通函数的 `noexcept` 条款。
- **属性:** 可用于指定 `lambda` 表达式的属性, 属性在第 1 章解释过。
- **返回类型:** 返回值的类型。如果省略, 编译器会根据与函数返回类型推导相同的规则推导 `lambda` 表达式的返回类型, 见第 1 章。
- **可选条款(C++20):** 为 `lambda` 闭包的函数调用运算符添加模板类型约束, 第 12 章“利用模板编写泛型代码”解释了如何指定这些约束。

注意:

对于不捕获任何内容的 `lambda` 表达式, 编译器自动提供转换运算符, 将 `lambda` 表达式转换为函数指针。例如, 这样的 `lambda` 表达式可用于传递给接收函数指针作为参数之一的函数。

19.4.2 lambda 表达式作为参数

lambda 表达式可以通过两种方式作为参数传递给函数, 一种是使用与 lambda 表达式签名匹配的 `std::function` 类型的函数参数, 另一种是使用模板类型参数。

例如, lambda 表达式可以传递给本章签名的 `findMatches()` 函数:

```
vector values1 { 2, 5, 6, 9, 10, 1, 1 };
vector values2 { 4, 4, 2, 9, 0, 3, 1 };
findMatches(values1, values2,
    [](int value1, int value2) { return value1 == value2; },
    printMatch);
```

19.4.3 泛型 lambda 表达式

可以为 lambda 表达式的参数使用自动类型推导, 而不是显式地为它们指定具体类型。若要为参数指定自动类型推导, 只需要简单地将类型指定为 `auto`, 类型推导规则与模板参数推导规则相同。

下面的示例定义了名为 `areEqual` 的泛型 lambda 表达式, 这个 lambda 表达式用作本章前面提到过的 `findMatches()` 函数的回调:

```
// Define a generic lambda expression to find equal values.
auto areEqual { [](const auto& value1, const auto& value2) {
    return value1 == value2; } };
// Use the generic lambda expression in a call to findMatches().
vector values1 { 2, 5, 6, 9, 10, 1, 1 };
vector values2 { 4, 4, 2, 9, 0, 3, 1 };
findMatches(values1, values2, areEqual, printMatch);
```

编译器为泛型 lambda 表达式生成的仿函数行为如下:

```
class CompilerGeneratedName
{
public:
    template <typename T1, typename T2>
    auto operator()(const T1& value1, const T2& value2) const
    { return value1 == value2; }
};
```

如果 `findMatches()` 函数被更改为不仅支持整型区间, 还支持其他类型, 那么 `areEqual` 泛型表达式不需要做任何修改也可继续使用。

19.4.4 lambda 捕获表达式

lambda 捕获表达式允许使用任何类型的表达式初始化捕获变量, 它可用于在 lambda 表达式中引入未从闭包作用域捕获的变量。例如, 下面的代码创建一个 lambda 表达式, 在这个 lambda 表达式中有两个可用变量: 一个是 `myCapture`, 使用 lambda 捕获表达式初始化为字符串 "Pi: ", 另一个是 `pi`, 从闭包作用域中通过值捕获到。注意使用捕获初始化来初始化的非引用捕获变量(如 `myCapture`)是拷贝构造的, 这意味着会移除 `const` 限定符。

```
double pi { 3.1415 };
auto myLambda { [myCapture = "Pi: ", pi]{ cout << myCapture << pi; } };
```

lambda 捕获变量可以用任何类型的表达式初始化, 因此也可以用 `std::move()` 来初始化。这对于不

能复制、只能移动的对象(如 `unique_ptr`)来说非常重要。默认情况下,值捕获使用复制语义,因此不可能在 `lambda` 表达式中按值捕获 `unique_ptr`。使用 `lambda` 捕获表达式,可以通过移动来捕获它,如下例所示:

```
auto myPtr { make_unique<double>(3.1415) };
auto myLambda { [p = move(myPtr)]{ cout << *p; } };
```

允许,但不推荐捕获变量的名称与闭包作用域中的名称相同,上面的示例可以写成下面这样:

```
auto myPtr { make_unique<double>(3.1415) };
auto myLambda { [myPtr = move(myPtr)]{ cout << *myPtr; } };
```

19.4.5 模板化 `lambda` 表达式

C++20 支持模板化 `lambda` 表达式,这样更容易访问泛型 `lambda` 表达式参数的类型信息。例如,假设有一个 `lambda` 表达式,它要求 `vector` 作为参数,但是 `vector` 中的元素可以是任何类型,因此,它是使用 `auto` 作为参数的泛型 `lambda` 表达式。`lambda` 表达式的主体想要推导出 `vector` 中元素的类型。在 C++20 之前,这只能通过 `decltype`()和 `std::decay_t` 来实现,称为类型萃取,类型萃取会在第 26 章“高级模板”中解释,但是这些细节对于掌握模板化 `lambda` 表达式的好处并不重要。只知道 `decay_t` 移除了类型中的任何 `const` 和引用限定就足够了,下面是泛型 `lambda` 表达式:

```
auto lambda { [] (const auto& values) {
    using V = decay_t<decltype(values)>; // The real type of the vector.
    using T = typename V::value_type;   // The type of the elements of the vector.
    T someValue { };
    T::some_static_function();
} };
```

可以像下面这样调用这个 `lambda` 表达式:

```
vector values { 1, 2, 100, 5, 6 };
lambda(values);
```

使用 `decltype`()和 `decay_t` 非常复杂,模板化 `lambda` 表达式更容易些。下面的 `lambda` 表达式强制其参数是 `vector` 类型,但仍为 `vector` 元素类型使用模板类型参数。

```
[] <typename T> (const vector<T>& values) {
    T someValue { };
    T::some_static_function();
}
```

模板化 `lambda` 表达式的另一个用法:如果想对泛型 `lambda` 表达式添加某种限制,例如,假设下面的泛型 `lambda` 表达式:

```
[] (const auto& value1, const auto& value2) { /* ... */ }
```

这个 `lambda` 表达式接收两个参数,编译器自动推导出每个参数的类型。因为两个参数的类型是分开推导的,所以 `value1` 和 `value2` 的类型可以不同。如果想要加以限制,希望两个参数具有相同的类型,可以将其转换为一个模板化 `lambda` 表达式:

```
[] <typename T> (const T& value1, const T& value2) { /* ... */ }
```

可以通过添加 `requires` 条款对模板类型设置约束,第 12 章讨论过这一点,这里有个示例:

```
[] <typename T> (const T& value1, const T& value2) requires integral<T> { /* ... */ }
```

19.4.6 lambda 表达式作为返回类型

通过使用本章前面讨论的 `std::function`，lambda 表达式可以从函数中返回，看下面的定义：

```
function<int(void)> multiplyBy2Lambda(int x)
{
    return [x]{ return 2 * x; };
}
```

该函数体创建一个 lambda 表达式，该表达式通过值捕获从闭包范围内捕获变量 `x`，并返回一个整数，该整数是传递给 `multiplyBy2Lambda()` 的参数值的两倍。`multiplyBy2Lambda()` 函数的返回类型是 `function<int(void)>`，这是不接收参数并返回整数的函数，函数体中定义的 lambda 表达式恰好与此原型匹配。变量 `x` 通过值捕获，因此在从函数返回 lambda 之前，`x` 值的一份拷贝会绑定到 lambda 表达式中的 `x` 上，函数的调用方法如下：

```
function<int(void)> fn { multiplyBy2Lambda(5) };
cout << fn() << endl;
```

auto 关键字更简单：

```
auto fn { multiplyBy2Lambda(5) };
cout << fn() << endl;
```

输出是 10。

函数返回类型推导(见第 1 章)允许更优雅地编写 `multiplyBy2Lambda()` 函数，如下所示：

```
auto multiplyBy2Lambda(int x)
{
    return [x]{ return 2 * x; };
}
```

`multiplyBy2Lambda()` 函数通过值 `[x]` 捕获变量 `x`，如下所示。假设函数重写为通过引用 `[&x]` 捕获变量，这没有效果。因为 lambda 表达式会在程序后面执行，不会在 `multiplyBy2Lambda()` 函数的范围内，此时对 `x` 的引用不再有效。

```
auto multiplyBy2Lambda(int x)
{
    return [&x]{ return 2 * x; }; // BUG!
}
```

19.4.7 未计算上下文中的 lambda 表达式

C++20 允许在所谓的未计算上下文中使用 lambda 表达式。例如，传递给 `decltype()` 的参数只在编译期使用且从不计算，因此，下面的语句在 C++17 及更早的版本中无效，但在 C++20 之后可以使用。

```
using LambdaType = decltype([](int a, int b) { return a + b; });
```

19.4.8 默认构造、拷贝和赋值

C++20 开始，可以默认构造、拷贝和赋值无状态 lambda 表达式，这里有个简单的示例：

```
auto lambda { [](int a, int b) { return a + b; } }; // A stateless lambda.
decltype(lambda) lambda2;                          // Default construction.
```



```
auto copy { lambda }; // Copy construction.
copy = lambda2; // Copy assignment.
```

结合在未计算上下文中使用 `lambda` 表达式，下面的代码有效：

```
Using LambdaType = decltype([](int a, int b) { return a + b; }); // Unevaluated.

LambdaType getLambda()
{
    return LambdaType{}; // Default construction.
}
```

19.5 调用

定义在 `<functional>` 中的 `std::invoke()`，可用于调用任何带有一组参数的可调用对象。下面的示例使用了 3 次 `invoke()`，一次调用普通函数，一次调用 `lambda` 表达式，一次调用 `string` 实例上的成员函数。

```
void printMessage(string_view message) { cout << message << endl; }

int main()
{
    invoke(printMessage, "Hello invoke.");
    invoke([](const auto& msg) { cout << msg << endl; }, "Hello invoke.");
    string msg { "Hello invoke." };
    cout << invoke(&string::size, msg) << endl;
}
```

就其本身而言，`invoke()` 没那么有用，因为可以直接调用函数或 `lambda` 表达式，但是在编写需要调用某个任意可调用对象的泛型模板代码时，它很有用。

19.6 本章小结

本章解释了回调的概念，回调是传递给其他函数来自定义它们行为的函数。回调可以是函数指针、函数对象或 `lambda` 表达式，相对于函数对象和适配器函数对象组合的操作，`lambda` 表达式可编译更可读的代码。记住，编写可读的代码与编写可工作的代码同等重要，即使 `lambda` 表达式比适配器函数对象长一些，`lambda` 表达式还是更具可读性，因此更易于维护。

现在已经熟练使用了回调，是时候深入了解标准库的真正强大之处，并讨论它的泛型算法了。

19.7 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

练习 19-1 使用 `lambda` 表达式重写本章中的 `IsLargerThan` 函数对象示例，可以在源码归档文件的 `c19_code\04_FunctionObjects\01_IsLargerThan.cpp` 中找到该代码。

练习 19-2 使用 `lambda` 表达式重写 `bind()` 示例，可以在源码归档文件的 `c19_code\04_FunctionObjects\07_bind.cpp` 中找到该代码。

练习 19-3 使用 lambda 表达式重写绑定类方法 `Handler::handleMatch()` 中的示例，可以在源码归档文件的 `c19_code\04_FunctionObjects\10_FindMatchesWithMethodPointer.cpp` 中找到该代码。

练习 19-4 第 18 章介绍的 `std::erase_if()` 函数，用于从容器中删除某个谓词返回 `true` 的元素，现在已经了解了关于回调的所有内容，接下来编写一个小程序，创建整型 `vector`，然后使用 `erase_if()` 从 `vector` 中删除所有奇数值，传递给 `erase_if()` 的谓词应该接收单个值并返回一个布尔值。

练习 19-5 实现一个名为 `Processor` 的类，构造函数应该接收一个回调函数，这个回调函数接收一个整数并返回一个整数。将这个回调保存在类的数据成员中，接下来，为接收整数并返回整数的函数调用运算符添加重载，该实现只是简单地将工作转发给存储的回调中。用不同的回调测试你的类。

第20章

掌握标准库算法

本章内容

- 算法的概念
- 标准库算法详解

由第 18 章“标准库容器”可知，标准库提供了大量泛型数据结构。大部分库都只提供数据结构。标准库却包含了大量泛型算法，这些算法大部分(只有少部分例外)都可以应用于任何容器的元素。通过这些算法，可在容器中查找、排序和处理元素，并执行其他大量操作。算法之美在于算法不仅独立于底层元素的类型，而且独立于操作的容器的类型。算法仅使用迭代器接口执行操作，第 17 章“理解迭代器和范围库”会对此讨论。

第 16 章“C++标准库概述”给出了所有标准库算法的高级概述，但没有任何编码细节，现在介绍这些算法如何在实践中使用。

20.1 算法概述

算法的魔力在于，算法把迭代器作为中介操作容器，而不直接操作容器本身。这样，算法没有绑定至特定的容器实现。所有标准库算法都实现为函数模板的形式，其中模板类型参数一般都是迭代器类型。将迭代器本身指定为函数的参数。模板化的函数通常可通过函数参数推导出模板类型，因此通常情况下可以像调用普通函数(而非模板)那样调用算法。

迭代器参数通常都是迭代器范围。根据第 17 章的描述，对于大部分容器来说，迭代器范围都是半开区间，因此包含范围内的第一个元素，但不包括最后一个元素。尾迭代器实际上是最后一个元素之后(past-the-end)的标记。

算法对传递给它们的迭代器有一些要求。例如，`copy_backward()`需要双向迭代器，用于从最后一个元素开始，将元素从一个序列拷贝到另一个序列。类似地，`stable_sort()`需要随机访问迭代器，用于在原地对元素进行排序的同时保持重复元素的顺序。这意味着这种算法不能操作没有提供所需迭代器的容器。`forward_list`容器只支持前向迭代器，不支持双向访问迭代器或随机访问迭代器，因此`copy_backward()`和`stable_sort()`不能用于`forward_list`。

大部分算法都定义在`<algorithm>`中，一些数值算法定义在`<numeric>`中。它们都在 `std` 名称空间中。

从 C++20 开始, 大多数算法都是 `constexpr`, 这意味着它们可以用于 `constexpr` 函数的实现, 请参考标准库指南以准确发现哪些算法是 `constexpr`。

理解算法的最好方法是首先分析一些示例。了解了其中几个算法的工作方式后, 就很容易了解其他算法。本节详细描述 `find()`、`find_if()` 和 `accumulate()` 算法。接下来的章节会讨论具有代表性样本的每一类算法。

20.1.1 `find()` 和 `find_if()` 算法

`find()` 在某个迭代器范围内查找特定元素。可将其用于任意容器类型的元素。这个算法返回所找到元素的迭代器; 如果没有找到元素, 则返回迭代器范围的尾迭代器。注意调用 `find()` 时指定的范围不要求是容器中元素的完整范围, 还可以是元素的子集。

警告:

如果 `find()` 没有找到元素, 那么返回的迭代器等于函数调用中指定的尾迭代器, 而不是底层容器的尾迭代器。

下面是一个 `std::find()` 示例。注意这个示例假定用户正常操作, 输入的是合法数值; 这个程序不会对用户输入执行任何错误检查。第 13 章“揭秘 C++ I/O”讨论了如何对流式输入执行错误检查。

```
import <algorithm>;
import <vector>;
import <iostream>;
using namespace std;

int main()
{
    vector<int> myVector;
    while (true) {
        cout << "Enter a number to add (0 to stop): ";
        int number;
        cin >> number;
        if (number == 0) { break; }
        myVector.push_back(number);
    }

    while (true) {
        cout << "Enter a number to lookup (0 to stop): ";
        int number;
        cin >> number;
        if (number == 0) { break; }
        auto endIt = cend(myVector);
        auto it = find(cbegin(myVector), endIt, number);
        if (it == endIt) {
            cout << "Could not find " << number << endl;
        } else {
            cout << "Found " << *it << endl;
        }
    }
}
```

调用 `find()` 时将 `cbegin(myVector)` 和 `endIt` 作为参数, 其中, `endIt` 定义为 `cend(myVector)`, 因此搜索的是 `vector` 的所有元素。如果需要搜索一个子范围, 可修改这两个迭代器。

下面是运行这个程序的示例输出：

```
Enter a number to add (0 to stop): 3
Enter a number to add (0 to stop): 4
Enter a number to add (0 to stop): 5
Enter a number to add (0 to stop): 6
Enter a number to add (0 to stop): 0
Enter a number to lookup (0 to stop): 5
Found 5
Enter a number to lookup (0 to stop): 8
Could not find 8
Enter a number to lookup (0 to stop): 0
```

使用 if 语句的初始化器(C++17)，可使用如下加粗语句来调用 find()并查找结果：

```
if (auto it { find(cbegin(myVector), endIt, number) }; it == endIt) {
    cout << "Could not find " << number << endl;
} else {
    cout << "Found " << *it << endl;
}
```

一些容器(例如 map 和 set)以类方法的方式提供自己的 find()版本，正如第 18 章讨论这些容器时的示例所示。

警告：

如果容器提供的方法具有与泛型算法同样的功能，那么应该使用相应的方法，那样速度更快。比如，泛型算法 find()的复杂度为线性时间，用于 map 迭代器时也是如此；而 map 中 find()方法的复杂度是对数时间。

find_if()和 find()类似，区别在于 find_if()接收谓词函数回调作为参数，谓词返回 true 或 false，而不是简单地匹配元素。find_if()算法对范围内的每个元素调用谓词，直到谓词返回 true；如果返回了 true，find_if()返回引用这个元素的迭代器。下面的程序从用户读入测试分数，检查是否存在“完美”分数。完美分数指的是大于或等于 100 的分数。这个程序与前一个例子中的程序相似。两个程序的区别已加粗显示。

```
bool perfectScore(int num) { return (num >= 100); }

int main()
{
    vector<int> myVector;
    while (true) {
        cout << "Enter a test score to add (0 to stop): ";
        int score;
        cin >> score;
        if (score == 0) { break; }
        myVector.push_back(score);
    }

    auto endIt { cend(myVector) };
    auto it { find_if(cbegin(myVector), endIt, perfectScore) };
    if (it == endIt) {
        cout << "No perfect scores" << endl;
    } else {
        cout << "Found a \"perfect\" score of " << *it << endl;
    }
}
```

```

    }
    return 0;
}

```

这个程序传递指向 `perfectScore()` 函数的指针，然后 `find_if()` 算法对每个元素调用这个函数，直到其返回 `true` 为止。

下面这个示例与对 `find_if()` 的调用相同，但是使用了 `lambda` 表达式，第 19 章“函数指针、函数对象和 `lambda` 表达式”讨论过。注意这个例子中没有 `perfectScore()` 函数。

```

auto it { find_if(cbegin(myVector), endIt, [](int i){ return i >= 100; }) };

```

20.1.2 accumulate()算法

我们经常需要计算容器中所有元素的总和或其他算术值。`accumulate()` 函数就提供了这种功能，该函数在 `<numeric>` (而非 `<algorithm>`) 中定义。通过这个函数的最基本形式可计算指定范围内元素的总和。例如，下面的函数计算 `span` 中整数序列的算术平均值。将所有元素的总和除以元素数目，就得到算术平均值。

```

double arithmeticMean(const vector<int>& nums)
{
    double sum { accumulate(cbegin(nums), cend(nums), 0.0) };
    return sum / nums.size();
}

```

`accumulate()` 算法接收的第三个参数是总和的初始值，在这个例子中为 `0.0` (加法计算的恒等值)，表示从 `0.0` 开始累加总和。

`accumulate()` 的第二种重载允许调用者指定要执行的操作，而不是执行默认加法操作。这个操作的形式是二元回调。假设需要计算几何平均数。如果一个序列中有 `m` 个数字，那么几何平均数就是 `m` 个数字连乘的 `m` 次方根。在这个例子中，调用 `accumulate()` 计算乘积而不是总和。因此这个程序可以这样写：

```

int product(int value1, int value2) { return value1 * value2; }

double geometricMean(span<const int> values)
{
    int mult { accumulate(cbegin(values), cend(values), 1, product) };
    return pow(mult, 1.0 / values.size()); // pow() needs <cmath>
}

```

注意，将 `product()` 函数作为回调传递给 `accumulate()`，而把累计的初始值设置为 `1` (乘法计算的恒等值)。

可以使用 `lambda` 表达式替代单独的 `product()` 函数：

```

double geometricMean(span<const int> values)
{
    double mult { accumulate(cbegin(values), cend(values), 1,
        [](int value1, int value2) { return value1 * value2; }) };
    return pow(mult, 1.0 / values.size()); // pow() needs <cmath>
}

```

也可以使用第 19 章讨论的透明 `multiply<>` 函数对象来实现 `geometricMean()` 函数：

```
double geometricMean(span<const int> values)
{
    int mult { accumulate(cbegin(values), cend(values), 1, multiplies<>()) };
    return pow(mult, 1.0 / values.size());
}
```

20.1.3 在算法中使用移动语义

与标准库容器一样，标准库算法也做了优化，以便在合适时使用移动语义。也就是说它们可以移动对象，而不是执行潜在昂贵的拷贝操作。这可极大地加速特定的算法，例如 `remove()`。因此，强烈建议在需要保存到容器中的自定义元素类中实现移动语义。通过实现移动构造函数和移动赋值运算符，任何类都可添加移动语义。正如第 18 章所讨论的，它们都被标记为 `noexcept`，否则它们不会被标准库容器和算法使用。有关如何向自定义类添加移动语义的详细信息，请参阅第 9 章“精通类和对象”中的“实现移动语义”一节。

20.1.4 算法回调

警告：

允许算法对给定回调(例如仿函数和 `lambda` 表达式)进行多次拷贝，并对不同的元素调用不同的拷贝。

回调可以有多个拷贝，这一事实对此类回调的副作用有很强的限制。基本上回调必须是无状态的，对于仿函数，意味着函数调用运算符必须是 `const`，因此不能编写这样的仿函数，其依赖于对象的任何内部状态在调用之间保持一致。类似于 `lambda` 表达式不能标记为 `mutable`。

也有些例外，`generate()` 和 `generate_n()` 算法可以接收有状态的回调，但即使这样也会生成回调的一份拷贝。最重要的是，它们不返回那个拷贝，所以一旦算法完成，就不能访问它对状态所做的改动。唯一的例外就是 `for_each()`，它将给定的谓词复制一次到 `for_each()` 算法里，并在完成时返回该拷贝，可以通过返回值访问改动的状态。

为了避免回调被算法复制，可以使用 `std::ref()` 辅助函数向算法传递回调引用。这确保了算法总是使用相同的回调。下面的代码片段基于本章前面的示例，但是使用了一个存储在名为 `isPerfectScore` 变量里的 `lambda` 表达式，`lambda` 表达式计算它被调用的频率并将其写到标准输出。`isPerfectScore` 被传递到 `find_if()` 算法中，代码片段的最后一条语句额外显式调用了一次 `isPerfectScore`。

```
auto isPerfectScore { [tally = 0] (int i) mutable {
    cout << ++tally << endl; return i >= 100; } };

auto endIt { cend(myVector) };
auto it { find_if(cbegin(myVector), endIt, isPerfectScore) };
if (it != endIt) { cout << "Found a \"perfect\" score of " << *it << endl; }
isPerfectScore(1);
```

输出如下：

```
Enter a test score to add (0 to stop): 1
Enter a test score to add (0 to stop): 2
Enter a test score to add (0 to stop): 3
Enter a test score to add (0 to stop): 0
```



```
1
2
3
1
```

输出显示 `find_if()` 算法调用 `isPerfectScore` 3 次，输出 1、2、3。最后一行对 `isPerfectScore` 的显式调用发生在 `isPerfectScore` 的另一个实例上，因为它再次从 1 开始。

现在将 `find_if()` 的调用改为下面这样：

```
auto it { find_if(cbegin(myVector), endIt, ref(isPerfectScore)) };
```

现在的输出会是 1、2、3、4，表示没有拷贝 `isPerfectScore`。

20.2 算法详解

第 16 章列出了所有可用的标准库算法，并分为不同的类别。大多数算法在 `<algorithm>` 中定义，但有几个算法位于 `<numeric>` 中。它们都在 `std` 名称空间中。

本章的目的不是提供所有可用算法的参考概述，相反这里只选择几个类别，并举例说明。知道如何使用它们后，就能顺利地使用其他算法了。标准库参考资源包含所有算法的总结，可参阅附录 B。

20.2.1 非修改序列算法

非修改序列算法是不修改它们所操作的元素序列的算法，它们包括在某个范围内搜索元素的函数、比较两个范围的函数以及许多工具算法。

1. 搜索算法

本章开头介绍了使用两个搜索算法(`find()`和 `find_if()`)的示例。标准库提供了基本 `find()` 算法的一些其他变种，这些算法对元素序列执行操作。第 16 章中的“搜索算法”部分描述了可供使用的不同搜索算法，还包含了算法的复杂度。

所有算法都使用默认的比较运算符 `operator==` 和 `operator<`，还提供了重载版本，以允许指定比较回调。

下面是一些搜索算法的示例：

```
// The list of elements to be searched.
vector myVector { 5, 6, 9, 8, 8, 3 };
auto beginIter { cbegin(myVector) };
auto endIter { cend(myVector) };

// Find the first element that does not satisfy the given lambda expression.
auto it { find_if_not(beginIter, endIter, [](int i){ return i < 8; }) };
if (it != endIter) {
    cout << "First element not < 8 is " << *it << endl;
}

// Find the first pair of matching consecutive elements.
it = adjacent_find(beginIter, endIter);
if (it != endIter) {
    cout << "Found two consecutive equal elements with value " << *it << endl;
}

// Find the first of two values.
```

```

vector targets { 8, 9 };
it = find_first_of(beginIter, endIter, cbegin(targets), cend(targets));
if (it != endIter) {
    cout << "Found one of 8 or 9: " << *it << endl;
}

// Find the first subsequence.
vector sub { 8, 3 };
it = search(beginIter, endIter, cbegin(sub), cend(sub));
if (it != endIter) {
    cout << "Found subsequence {8,3}" << endl;
} else {
    cout << "Unable to find subsequence {8,3}" << endl;
}

// Find the last subsequence (which is the same as the first in this example).
auto it2 { find_end(beginIter, endIter, cbegin(sub), cend(sub)) };
if (it != it2) {
    cout << "Error: search and find_end found different subsequences "
        << "even though there is only one match." << endl;
}

// Find the first subsequence of two consecutive 8s.
it = search_n(beginIter, endIter, 2, 8);
if (it != endIter) {
    cout << "Found two consecutive 8s" << endl;
} else {
    cout << "Unable to find two consecutive 8s" << endl;
}

```

输出结果如下：

```

First element not < 8 is 9
Found two consecutive equal elements with value 8
Found one of 8 or 9: 9
Found subsequence {8,3}
Found two consecutive 8s

```

注意：

记住，一些容器中具有与泛型算法对等的方法。这种情况下，建议使用容器方法而非泛型算法，因为容器方法更高效。

2. 专用的搜索算法

从 C++17 开始，`search()` 算法的一个可选参数允许指定要使用的搜索算法。有 3 个选项：`default_searcher`、`boyer_moore_searcher` 或 `boyer_moore_horspool_searcher`，它们都在 `<functional>` 中定义。后两个选项实现了知名的 Boyer-Moore 和 Boyer-Moore-Horspool 搜索算法。它们比默认搜索器更高效，可用于在一大块文本中查找子字符串。Boyer-Moore 搜索算法的复杂度如下， N 是在其中搜索的序列(haystack)的大小， M 是要查找的模式(needle)的大小。

- 如果未找到模式，最坏情况下的复杂度为 $O(N+M)$ 。
- 如果找到模式，最坏情况下的复杂度为 $O(N*M)$ 。

这些是理论上最坏情况下的复杂度。在实际中，这些专用搜索算法比 $O(N)$ 更好，比默认算法更快！因为它们可以跳过字符，而非查找 haystack 中的每个字符。它们还有一个有趣的特性，即 `needle`

越长，速度越快，因为那时可跳过 haystack 中的更多字符。Boyer-Moore 和 Boyer-Moore-Horspool 算法的区别在于，在初始化以及算法的每个循环迭代中，后者的固定开销较少；但是，后者在最坏情况下的复杂度明显高于前者的算法。因此，需要根据具体情况进行选择。

下面是一个使用 Boyer-Moore 搜索算法的例子：

```
string text { "This is the haystack to search a needle in." };
string toSearchFor { "needle" };
boyer_moore_searche searcher { cbegin(toSearchFor), cend(toSearchFor) };
auto result { search(cbegin(text), cend(text), searcher) };
if (result != cend(text)) {
    cout << "Found the needle." << endl;
} else {
    cout << "Needle not found." << endl;
}
```

3. 比较算法

可通过 4 种不同的方法比较整个范围内的元素：equal()、mismatch()、lexicographical_compare() 和 lexicographical_compare_three_way() (C++20)。这些算法的好处是可比较不同容器内的范围。例如，可比较 vector 和 list 的内容。一般情况下，这些算法最适用于顺序容器。这些算法的工作方法是比较两个集合中对应位置的值。下面列出每个算法的工作方式。

- equal(): 如果所有对应元素都相等，则返回 true。最初，equal()接收 3 个迭代器，分别是第一个范围的首尾迭代器，以及第二个范围的首迭代器。该版本要求两个范围的元素数目相同。从 C++14 开始，有了接收 4 个迭代器的重载版本，分别是第一个范围的首尾迭代器，以及第二个范围的首尾迭代器。该版本可处理不同大小的范围；为保险起见，建议始终使用四迭代器版本。
- mismatch(): 返回多个迭代器，每个范围对应一个迭代器，表示范围内不匹配的对对应元素。与 equal()一样，存在三迭代器版本和四迭代器版本。同样，为保险起见，建议使用四迭代器版本。
- lexicographical_compare(): 在两个提供的范围内对具有相同位置的元素相互比较(按顺序)，如果第一个范围内的第一个不相等元素小于第二个范围内的对应元素，或如果第一个范围内的元素个数少于第二个范围，且第一个范围内的所有元素都等于第二个范围内对应的初始子序列，那么返回 true。名称 lexicographical_compare 来自这个算法对字符串比较规则的模仿，但对规则集进行了扩展，能够处理任意类型的对象。
- lexicographical_compare_three_way(): 与 lexicographical_compare()类似，除了一点，它执行 C++20 的三路比较操作，并返回一个比较类别类型(strong_ordering、weak_ordering 或 partial_ordering，这在第 1 章讨论过)，而不是布尔类型。

注意：

如果要比较两个同类型容器的元素，可使用运算符 operator==和 operator<，而不是这些算法。这些算法可用于比较不同容器类型的元素序列、子范围和 C 风格数组等。

下面是使用这些算法的示例：

```
// Function template to populate a container of ints.
// The container must support push_back().
template<typename Container>
void populateContainer(Container& cont)
```

```

{
    while (true) {
        cout << "Enter a number (0 to quit): ";
        int value;
        cin >> value;
        if (value == 0) { break; }
        cont.push_back(value);
    }
}

int main()
{
    vector<int> myVector;
    list<int> myList;

    cout << "Populate the vector:" << endl;
    populateContainer(myVector);
    cout << "Populate the list:" << endl;
    populateContainer(myList);

    // Compare the two containers
    if (equal(cbegin(myVector), cend(myVector),
             cbegin(myList), cend(myList))) {
        cout << "The two containers have equal elements" << endl;
    } else {
        // If the containers were not equal, find out why not
        auto miss { mismatch(cbegin(myVector), cend(myVector),
                             cbegin(myList), cend(myList)) };
        cout << "The following initial elements are the same in "
              << "the vector and the list:" << endl;
        for (auto i { cbegin(myVector) }; i != miss.first; ++i) {
            cout << *i << '\t';
        }
        cout << endl;
    }

    // Now order them.
    if (lexicographical_compare(cbegin(myVector), cend(myVector),
                               cbegin(myList), cend(myList))) {
        cout << "The vector is lexicographically first." << endl;
    } else {
        cout << "The list is lexicographically first." << endl;
    }
}

```

下面是这个程序的示例运行结果：

```

Populate the vector:
Enter a number (0 to quit): 5
Enter a number (0 to quit): 6
Enter a number (0 to quit): 7
Enter a number (0 to quit): 0
Populate the list:
Enter a number (0 to quit): 5
Enter a number (0 to quit): 6
Enter a number (0 to quit): 9
Enter a number (0 to quit): 8

```

```
Enter a number (0 to quit): 0
The following initial elements are the same in the vector and the list:
5      6
The vector is lexicographically first.
```

4. 计数算法

非修改计数算法有 `all_of()`、`any_of()`、`none_of()`、`count()`和 `count_if()`。下面是前 3 个算法的示例:

```
// all_of()
vector vec2 { 1, 1, 1, 1 };
if (all_of(cbegin(vec2), cend(vec2), [](int i){ return i == 1; })) {
    cout << "All elements are == 1" << endl;
} else {
    cout << "Not all elements are == 1" << endl;
}

// any_of()
vector vec3 { 0, 0, 1, 0 };
if (any_of(cbegin(vec3), cend(vec3), [](int i){ return i == 1; })) {
    cout << "At least one element == 1" << endl;
} else {
    cout << "No elements are == 1" << endl;
}

// none_of()
vector vec4 { 0, 0, 0, 0 };
if (none_of(cbegin(vec4), cend(vec4), [](int i){ return i == 1; })) {
    cout << "All elements are != 1" << endl;
} else {
    cout << "Some elements are == 1" << endl;
}
```

输出如下:

```
All elements are == 1
At least one element == 1
All elements are != 1
```

下面的示例使用 `count_if()`算法计算 `vector` 中满足特定条件的元素个数。条件以 `lambda` 表达式的形式给出, 该表达式从闭包作用域内按值捕获 `value` 变量:

```
vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int value { 3 };
auto tally { count_if(cbegin(values), cend(values),
    [value](int i){ return i > value; }) };
cout << format("Found {} values > {}.", tally, value) << endl;
```

输出如下:

```
Found 6 values > 3
```

该示例可以扩展为通过引用捕获变量, 下面的 `lambda` 表达式计算在闭包区域内通过递增变量来调用的个数, `lambda` 表达式通过引用捕获变量:

```
vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int value { 3 };
int callCounter { 0 };
```

```
auto tally { count_if(cbegin(values), cend(values),
    [value, &callCounter](int i){ ++callCounter; return i > value; }) };
cout << "The lambda expression was called " << callCounter
    << " times." << endl;
cout << format("Found {} values > {}.", tally, value) << endl;
```

输出如下:

```
The lambda expression was called 9 times.
Found 6 values > 3
```

20.2.2 修改序列算法

标准库提供了多种修改序列算法, 这些算法执行的任务包括: 从一个范围向另一个范围复制元素、删除元素以及反转某个范围内元素的顺序。

一些修改算法涉及源范围和目标范围的概念。从源范围读取元素, 然后在目标范围中进行修改, 这种算法的一个示例是 `copy()`。

其他算法就地(in place)执行操作; 也就是说, 只需要一个范围, 例如 `generate()` 算法。

警告:

修改算法不能将元素插入目标范围中, 仅可重写/修改目标范围中已经存在的元素。第 17 章描述了如何使用迭代器适配器, 在目标范围中真正插入元素。

注意:

`map` 和 `multimap` 的范围不能用作修改算法的目标范围。这些算法改写全部元素, 而在 `map` 中, 元素是键值对。`map` 和 `multimap` 将键标记为 `const`, 因此不能为其赋值。`set` 和 `multiset` 也是如此。替换方案是使用插入迭代器, 详见第 17 章。

16.2.18 节的“修改序列算法”部分列出了所有可用的修改算法, 并给出了描述信息。本节列举其中不少算法的代码示例。如果理解了本节讲述的算法, 就应能使用没有给出示例的其他算法。

1. 生成

`generate()` 算法需要一个迭代器范围, 它把该迭代器范围内的值替换为从函数返回的值, 并作为第三个参数。下例结合 `generate()` 算法和一个 `lambda` 表达式将 2、4、8、16 等值填充到 `vector`。

```
vector<int> vec(10);
int value { 1 };
generate(begin(vec), end(vec), [&value]{ value *= 2; return value; });
for (const auto& i : vec) { cout << i << " "; }
```

输出如下所示:

```
2 4 8 16 32 64 128 256 512 1024
```

2. 转换

`transform()` 算法有多种重载, 其中之一是对范围内的每个元素应用回调, 期望回调生成一个新元素, 并保存在指定的目标范围中。如果希望 `transform()` 就地工作, 那么源范围和目标范围可以是同一范围。其参数是源序列的首尾迭代器、目标序列的首迭代器以及回调。例如, 下面的代码片段将 `vector` 中的每个元素增加 100, `populateContainer()` 函数与本章前面定义的相同。

```

vector<int> myVector;
populateContainer(myVector);

cout << "The vector contains:" << endl;
for (const auto& i : myVector) { cout << i << " "; }
cout << endl;

transform(begin(myVector), end(myVector), begin(myVector),
    [](int i){ return i + 100;});

cout << "The vector contains:" << endl;
for (const auto& i : myVector) { cout << i << " "; }

```

transform()的另一种形式对范围内的元素对调用二元函数，需要将第一个范围的首尾迭代器、第二个范围的首迭代器以及目标范围的首迭代器作为参数。下例创建两个 **vector**，然后通过 **transform()** 计算元素对的和，并将结果保存回第一个 **vector**：

```

vector<int> vec1, vec2;
cout << "Vector1:" << endl; populateContainer(vec1);
cout << "Vector2:" << endl; populateContainer(vec2);

if (vec2.size() < vec1.size())
{
    cout << "Vector2 should be at least the same size as vector1." << endl;
    return 1;
}

// Create a lambda to print the contents of a container
auto printContainer { [](const auto& container) {
    for (auto& i : container) { cout << i << " "; }
    cout << endl;
} };

cout << "Vector1: "; printContainer(vec1);
cout << "Vector2: "; printContainer(vec2);

transform(begin(vec1), end(vec1), begin(vec2), begin(vec1),
    [](int a, int b){return a + b;});

cout << "Vector1: "; printContainer(vec1);
cout << "Vector2: "; printContainer(vec2);

```

输出如下所示：

```

Vector1:
Enter a number (0 to quit): 1
Enter a number (0 to quit): 2
Enter a number (0 to quit): 0
Vector2:
Enter a number (0 to quit): 11
Enter a number (0 to quit): 22
Enter a number (0 to quit): 33
Enter a number (0 to quit): 0
Vector1: 1 2
Vector2: 11 22 33
Vector1: 12 24
Vector2: 11 22 33

```

注意:

`transform()`和其他修改算法通常返回一个引用目标范围内最后一个值后面那个位置(past-the-end)的迭代器。本书中的例子通常都忽略了返回值。

3. 拷贝

`copy()`算法可将一个范围内的元素复制到另一个范围,从这个范围内的第一个元素开始直到最后一个元素。源范围和目标范围必须不同,但在一定限制条件下可以重叠。限制条件如下:对于 `copy(b,e,d)`,如果 `d` 在 `b` 之前,则可以重叠;但如果 `d` 处于 `[b,e)` 范围,则行为不确定。与所有修改算法类似,`copy()`不会向目标范围插入元素,只改写已有的元素。第 17 章将描述如何使用迭代器适配器,使用 `copy()`向容器或流插入元素。

下面举一个使用 `copy()`的简单例子,这个例子对 `vector` 应用 `resize()`方法,以确保目标容器中有足够空间。这个例子将 `vec1` 中的所有元素复制到 `vec2`:

```
vector<int> vec1, vec2;
populateContainer(vec1);
vec2.resize(size(vec1));
copy(cbegin(vec1), cend(vec1), begin(vec2));
for (const auto& i : vec2) { cout << i << " "; }
```

还有一个 `copy_backward()`算法,这个算法将源范围内的元素反向复制到目标范围。换句话说,这个算法从源范围的最后一个元素开始,将这个元素放在目标范围的最后一个位置,然后在每一次复制之后反向移动。分析 `copy_backward()`,源范围和目标范围必须是不同的,但在一定限制条件下可以重叠。限制条件如下:对于 `copy_backward(b,e,d)`,如果 `d` 在 `e` 之后,则能正确重叠;但如果 `d` 处于 `(b,e]` 范围,则行为不确定。前面的例子可按如下代码修改为使用 `copy_backward()`而不是 `copy()`。注意第三个参数应该指定 `end(vec2)`而不是 `begin(vec2)`,输出和 `copy()`的一致。

```
copy_backward(cbegin(vec1), cend(vec1), end(vec2));
```

在使用 `copy_if()`算法时,需要提供由两个迭代器指定的输入范围、由一个迭代器指定的输出范围以及一个谓词(函数或 `lambda` 表达式)。该算法将满足给定谓词的所有元素复制到目标范围。记住,复制不会创建或扩大容器,只是替换现有元素。因此,目标范围应当足够大,从而保存要复制的所有元素。当然,复制元素后,最好删除超出最后一个元素复制位置的空间。为便于达到这个目的,`copy_if()`返回了目标范围内最后一个复制的元素后面那个位置(one-past-the-last-copied element)的迭代器,以便确定需要从目标容器中删除的元素个数。下例演示了这个操作,这个例子只把偶数复制到 `vec2`:

```
vector<int> vec1, vec2;
populateContainer(vec1);
vec2.resize(size(vec1));
auto endIterator { copy_if(cbegin(vec1), cend(vec1),
    begin(vec2), [](int i){ return i % 2 == 0; }) };
vec2.erase(endIterator, end(vec2));
for (const auto& i : vec2) { cout << i << " "; }
```

`copy_n()`从源范围复制 `n` 个元素到目标范围。`copy_n()`的第一个参数是起始迭代器,第二个参数是指定要复制的元素个数,第三个参数是目标迭代器。`copy_n()`算法不执行任何边界检查,因此一定要确保起始迭代器递增 `n` 个要复制的元素后,不会超过集合的 `end()`,否则程序会产生未定义的行为。下面是一个例子:

```
vector<int> vec1, vec2;
```



```

populateContainer(vec1);
size_t tally { 0 };
cout << "Enter number of elements you want to copy: ";
cin >> tally;
tally = min(tally, size(vec1));
vec2.resize(tally);
copy_n(cbegin(vec1), tally, begin(vec2));
for (const auto& i : vec2) { cout << i << " "; }

```

4. 移动

有两个和移动相关的算法：`move()`和`move_backward()`。它们都使用了第9章“掌握类和对象”讨论的移动语义。如果要在自定义类型元素的容器中使用这两个算法，那么需要在元素类中提供移动赋值运算符，请参阅下例。`main()`函数创建了一个带有3个`MyClass`对象的`vector`，然后将这些元素从`vecSrc`移到`vecDst`。注意这段代码包含两种不同的`move()`用法。一种是，`move()`函数接收一个参数，将`lvalue`转换为`rvalue`，在`<utility>`中定义；而另一种是，接收3个参数的`move()`是标准库的`move()`算法，这个算法在容器之间移动元素。有关移动赋值运算符的实现和`std::move()`单参数版本的使用，请参阅第9章。

```

class MyClass
{
public:
    MyClass() = default;
    MyClass(const MyClass& src) = default;
    MyClass(string str) : m_str { move(str) } {}
    virtual ~MyClass() = default;

    // Move assignment operator
    MyClass& operator=(MyClass&& rhs) noexcept {
        if (this == &rhs) { return *this; }
        m_str = move(rhs.m_str);
        cout << format("Move operator= (m_str={})", m_str) << endl;
        return *this;
    }

    void setString(string str) { m_str = move(str); }
    const string& getString() const { return m_str; }
private:
    string m_str;
};

int main()
{
    vector<MyClass> vecSrc { MyClass { "a" }, MyClass { "b" }, MyClass { "c" } };
    vector<MyClass> vecDst (vecSrc.size());
    move(begin(vecSrc), end(vecSrc), begin(vecDst));
    for (const auto& c : vecDst) { cout << c.getString() << " "; }
}

```

输出如下所示：

```

Move operator= (m_str=a)
Move operator= (m_str=b)
Move operator= (m_str=c)
a b c

```

注意:

第 9 章解释过, 在移动操作中, 源对象将处于有效但不确定的状态。在前面的例子中, 这意味着在执行移动操作后, 不应该再使用 `vecSrc` 中的元素了, 除非使它们重回确定状态; 例如, 在它们之上调用方法(不包含任何预置条件), 如 `setString()`。

`move_backward()` 使用和 `move()` 同样的移动机制, 但按从最后一个元素向第一个元素的顺序移动。对于 `move()` 和 `move_backward()`, 在符合某些限制条件的情况下允许源范围和目标范围重叠。限制条件与 `copy()` 和 `copy_backward()` 的相同。

5. 替换

`replace()` 和 `replace_if()` 算法将一个范围内匹配某个值或满足某个谓词的元素替换为新的值。比如 `replace_if()` 算法的第一个和第二个参数指定了容器中元素的范围。第三个参数是一个返回 `true` 或 `false` 的 lambda 表达式, 如果它返回 `true`, 那么容器中的对应值被替换为第四个参数指定的值; 如果它返回 `false`, 则保留原始值。

例如, 假定要将容器中的所有奇数值替换为 0:

```
vector<int> vec;
populateContainer(vec);
replace_if(begin(vec), end(vec), [](int i){ return i % 2 != 0; }, 0);
for (const auto& i : vec) { cout << i << " "; }
```

`replace()` 和 `replace_if()` 也有名为 `replace_copy()` 和 `replace_copy_if()` 的变体, 这些变体将结果复制到不同的目标范围。它们类似于 `copy()`, 因为目标范围必须足够大, 以容纳新元素。

6. 擦除

正如第 18 章介绍, C++20 对几乎所有标准库容器都支持 `std::erase()` 和 `std::erase_if()`。官方将这些操作称为统一容器擦除, `erase()` 函数删除容器中与给定值匹配的所有元素, 而 `erase_if()` 函数则删除与给定谓词匹配的所有元素。注意这些算法需要容器的引用, 而不是迭代器范围。从 C++20 开始, 这些函数是删除容器中元素的首选方法。

例如, 下面的 `removeEmptyStrings()` 函数删除字符串 `vector` 中的所有空元素, 使用 `erase_if()` 完成了所有操作, 函数只使用简单的一行:

```
void removeEmptyStrings(vector<string>& strings)
{
    erase_if(strings, [](const string& str){ return str.empty(); });
}

int main()
{
    vector<string> myVector {"", "one", "two", "three", "four"};
    for (auto& str : myVector) { cout << "\"" << str << "\" "; }
    cout << endl;
    removeEmptyStrings(myVector);
    for (auto& str : myVector) { cout << "\"" << str << "\" "; }
    cout << endl;
}
```

输出如下:

```
"" "one" "" "two" "three" "four"
"one" "two" "three" "four"
```

注意:

`std::erase()`不适用于有序和无序的关联容器,因为这些容器有 `erase(key)`方法,该方法性能更好,应该使用它。另一方面, `erase_if()`函数适用于所有容器。

7. 删除

如果编译器还不支持前一节讨论的 C++20 的 `erase()`和 `erase_if()`算法,那么不得不求助于其他方案。你可能想到的第一个解决方案是查看文档,确定容器是否有 `erase()`方法,然后迭代所有元素,并对每个满足条件的元素调用 `erase()`。`vector` 是包含 `erase()`方法的容器之一。然而,如果对 `vector` 容器应用 `erase()`,这个解决方案的效率非常低下,因为要保持 `vector` 在内存中的连续性,会涉及很多内存操作,因而得到二次(平方)复杂度;所谓二次复杂度,是指运行时间是输入大小的平方的函数,即 $O(n^2)$ 。这个解决方案还容易产生错误,因为必须非常小心地确保每次调用 `erase()`之后迭代器依然有效。例如,如下函数从 `string vector` 中删除空字符串,而未使用算法。注意,需要在 `for` 循环中精心操纵 `iter`:

```
void removeEmptyStringsWithoutAlgorithms(vector<string>& strings)
{
    for (auto iter { begin(strings) }; iter != end(strings); )
        if (iter->empty())
            iter = strings.erase(iter);
        else
            ++iter;
}
```

注意:

二次(平方)复杂度意味着运行时间是输入大小的平方的函数,即 $O(n^2)$ 。

上面的解决方案效率低下,不建议使用。这个问题更好的解决方案是“删除-擦除法”(remove-erase-idiom),下面讲解这种线性时间方法。

这种算法只能访问迭代器抽象,不能访问容器。因此删除算法不能真正地从底层容器中删除元素,而是将匹配给定值或谓词的元素替换为下一个不匹配给定值或谓词的元素。为此使用移动赋值。结果是将所有要保留的元素移动到范围的开头,因此范围分为两个集合:一个用于保存要保留的元素,另一个用于保存要删除的元素。返回的迭代器指向要删除的元素范围内的第一个元素。如果真的需要从容器中删除这些元素,必须先使用 `remove()`或 `remove_if()`算法,然后调用容器的 `erase()`方法,将从返回的迭代器开始到范围尾部的所有元素删除。这就是删除-擦除法。下面是使用这一风格的 `removeEmptyStrings()`函数的实现:

```
void removeEmptyStrings(vector<string>& strings)
{
    auto it { remove_if(begin(strings), end(strings),
        [](const string& str){ return str.empty(); }) };
    // Erase the removed elements.
    strings.erase(it, end(strings));
}
```

警告:

使用“删除-擦除法”时,切勿忘记 `erase()`的第二个参数!如果忘掉第二个参数, `erase()`将仅从容器中删除一个元素,即作为第一个参数传递的迭代器引用的元素。

`remove()`和 `remove_if()`的 `remove_copy()`和 `remove_copy_if()`变体不会改变源范围，而将所有未删除的元素复制到另一个目标范围。这些算法和 `copy()`类似，要求目标范围必须足够大，以便保存新元素。

注意：

`remove()`函数系列是稳定的，这些函数保持了容器中剩余元素的顺序，尽管这些算法将保留的元素向前移动了。

注意：

建议使用 C++20 中关于删除-擦除法的 `std::erase_if()`和 `std::erase()`算法，或者关联容器的 `erase(key)`方法，并明确手写循环。

8. 唯一化

`unique()`算法是特殊的 `remove()`，`remove()`能将所有重复的连续元素删除。`list` 容器提供了自己的具有同样语义的 `unique()`方法。通常情况下应该对有序序列使用 `unique()`，但 `unique()`也能用于无序序列。

`unique()`的基本形式是就地操作数据，还有一个名为 `unique_copy()`的版本，这个版本将结果复制到一个新的目标范围。

第 18 章的“list 示例：确定注册情况”部分展示了 `list::unique()`算法的一个示例，因此这里略去这个算法一般形式的例子。

9. 乱序

`shuffle()`以随机顺序重新安排某个范围内的元素，其复杂度为线性时间。它可用于实现洗牌等任务。`shuffle()`的参数是要乱序的范围的首尾迭代器，以及一个统一的随机数生成器对象，它指定如何生成随机数。下面是个示例(关于如何使用随机数生成器以及如何“设置种子”，详细信息会在第 23 章解释)：

```
vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

random_device seeder;
const auto seed { seeder.entropy() ? seeder() : time(nullptr) };
default_random_engine engine {
    static_cast<default_random_engine::result_type>(seed) };

for (int i { 0 }; i < 6; ++i) {
    shuffle(begin(values), end(values), engine);
    for (const auto& value : values) { cout << value << " "; }
    cout << endl;
}
```

可能的输出如下：

```
2 5 6 9 7 8 4 3 10 1
8 5 6 4 3 1 2 9 7 10
10 8 3 9 7 2 1 6 4 5
7 3 2 10 4 5 9 8 6 1
1 5 9 6 8 10 7 4 2 3
3 6 8 9 4 7 1 2 5 10
```

10. 抽样

`sample()`算法从给定的源范围返回 n 个随机选择的元素，并存储在目标范围。它需要 5 个参数：

- 要从中抽样的范围的首尾迭代器
- 目标范围的首迭代器，将随机选择的元素存储在目标范围
- 要选择的元素数量
- 随机数生成引擎

有关如何使用随机数生成引擎的详情，以及如何设置“种子”，请参阅第 23 章。下面是一个示例：

```
vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
const size_t numberOfSamples { 5 };
vector<int> samples(numberOfSamples);

random_device seeder;
const auto seed { seeder.entropy() ? seeder() : time(nullptr) };
default_random_engine engine(
    static_cast<default_random_engine::result_type>(seed));

for (int i { 0 }; i < 6; ++i) {
    sample(cbegin(values), cend(values), begin(samples), numberOfSamples, engine);
    for (const auto& sample : samples) { cout << sample << " "; }
    cout << endl;
}
```

可能的输出如下：

```
1 2 5 8 10
1 2 4 5 7
5 6 8 9 10
2 3 4 6 7
2 5 7 8 10
1 2 5 6 7
```

11. 反转

`reverse()`算法反转某个范围内元素的顺序。将范围内的第一个元素和最后一个元素交换，将第二个元素和倒数第二个元素交换，以此类推。

`reverse()`最基本的形式是就地运行，要求两个参数：范围的首尾迭代器。还有一个名为 `reverse_copy()`的版本，这个版本将结果复制到新的目标范围，它需要 3 个参数：源范围的首尾迭代器以及目标范围的起始迭代器。目标范围必须足够大，以便保存新元素。

下面是使用 `reverse()`的示例：

```
vector<int> values;
populateContainer(values);
reverse(begin(values), end(values));
for (const auto& i : values) { cout << i << " "; }
```

12. 移动元素

C++20 引入了 `shift_left()`和 `shift_right()`算法来移动给定范围内的元素，将它们移动到新位置。位于范围两端的元素会被删除，`shift_left()`返回指向新范围末尾的迭代器，而 `shift_right()`返回指向新范围开头的迭代器。下面是示例：

```
vector values { 11, 22, 33, 44, 55 };
for (const auto& value : values) { cout << value << " "; }
cout << endl;
// Shift elements to the left by 2 positions.
auto newEnd { shift_left(begin(values), end(values), 2) };
// Resize the vector to its proper size.
values.erase(newEnd, end(values));
for (const auto& value : values) { cout << value << " "; }
cout << endl;

// Shift elements to the right by 2 positions.
auto newBegin { shift_right(begin(values), end(values), 2) };
// Resize the vector to its proper size.
values.erase(begin(values), newBegin);
for (const auto& value : values) { cout << value << " "; }
cout << endl;
```

输出如下:

```
11 22 33 44 55
33 44 55
33
```

20.2.3 操作算法

此类算法只有两个: `for_each()` 和 `for_each_n()`。 `for_each()` 对范围内的每个元素执行回调, `for_each_n()` 对范围内的前 n 个元素执行回调。如果给定的迭代器类型是非 `const`, 则回调可以修改范围内的元素。这里提到这两个算法, 是因为可能会在已有的代码中遇到它们, 但使用基于区间的 `for` 循环通常比使用这两个算法更简单、更容易理解。

1. `for_each()`

下面这个示例使用 `lambda` 表达式打印 `map` 中的元素:

```
map<int, int> myMap { { 4, 40 }, { 5, 50 }, { 6, 60 } };
for_each(cbegin(myMap), cend(myMap), [](const auto& p)
    { cout << p.first << "->" << p.second << endl; });
```

`p` 的类型是 `const pair<int, int>&`。输出如下所示:

```
4->40
5->50
6->60
```

下例说明如何使用 `for_each()` 算法和 `lambda` 表达式, 计算范围内元素的和与积。注意, `lambda` 表达式只显式捕获需要的变量, 它按引用捕获变量, 否则 `lambda` 表达式内对 `sum` 和 `product` 的修改无法在 `lambda` 表达式外可见:

```
vector<int> myVector;
populateContainer(myVector);

int sum { 0 };
int product { 1 };
for_each(cbegin(myVector), cend(myVector),
    [&sum, &product](int i){
        sum += i;
```

```

        product *= i;
    });
    cout << "The sum is " << sum << endl;
    cout << "The product is " << product << endl;

```

也可用一个仿函数编写该例，其中，可累加信息，在 `for_each()` 处理完所有元素后检索信息。例如，可编写仿函数 `SumAndProduct` 同时跟踪，一次性计算元素的和以及运算的积。

```

class SumAndProduct
{
public:
    void operator()(int value)
    {
        m_sum += value;
        m_product *= value;
    }
    int getSum() const { return m_sum; }
    int getProduct() const { return m_product; }
private:
    int m_sum { 0 };
    int m_product { 1 };
};

int main()
{
    vector<int> myVector;
    populateContainer(myVector);

    SumAndProduct calculator;
    calculator = for_each(cbegin(myVector), cend(myVector), calculator);
    cout << "The sum is " << calculator.getSum() << endl;
    cout << "The product is " << calculator.getProduct() << endl;
}

```

你可能想要忽略 `for_each()` 的返回值，但在调用完成后仍试图读取 `calculator` 的信息。然而，这样做不可行，因为 `for_each()` 拷贝了仿函数，最终从调用返回这个副本。为获得正确的行为，必须捕获返回值。

关于 `for_each()` 和下面讨论的 `for_each_n()`，最后需要指出的一点是，使用 `lambda` 或回调时，允许回调函数使用非 `const` 引用作为参数并对其进行修改。这样可以修改实际迭代器范围内的值。下面是示例：

```

vector values { 11, 22, 33, 44 };
// Double each element in the values vector.
for_each(begin(values), end(values), [](auto& value) { value *= 2; });
// Print all the elements of the values vector.
for_each(cbegin(values), cend(values),
    [](const auto& value) { cout << value << endl; });

```

2. `for_each_n()`

`for_each_n()` 算法需要范围的起始迭代器、要迭代的元素数量以及函数回调。它返回的迭代器等于 `begin + n`。它通常不执行任何边界检查。下例只迭代 `map` 的前两个元素：

```

map<int, int> myMap { { 4, 40 }, { 5, 50 }, { 6, 60 } };

```

```
for_each_n(cbegin(myMap), 2, [](const auto& p)
    { cout << p.first << "->" << p.second << endl; });
```

20.2.4 分区算法

`partition_copy()` 算法将来自某个来源的元素复制到两个不同的目标。为每个元素选择目标的依据是谓词的结果: `true` 或 `false`。`partition_copy()` 的返回值是一对迭代器: 指向第一个和第二个目标范围内最后一次拷贝的元素的迭代器。将这些返回的迭代器与 `erase()` 结合使用, 可删除两个目标范围内多余的元素, 就像之前的 `copy_if()` 示例那样。下例要求用户输入一些整数, 然后将这些整数分区到两个目标 `vector` 中, 一个保存偶数, 另一个保存奇数。

```
vector<int> vec1, vecOdd, vecEven;
populateContainer(vec1);
vecOdd.resize(size(vec1));
vecEven.resize(size(vec1));

auto pairIters { partition_copy(cbegin(vec1), cend(vec1),
    begin(vecEven), begin(vecOdd),
    [](int i){ return i % 2 == 0; }) };

vecEven.erase(pairIters.first, end(vecEven));
vecOdd.erase(pairIters.second, end(vecOdd));

cout << "Even numbers: ";
for (const auto& i : vecEven) { cout << i << " "; }
cout << endl << "Odd numbers: ";
for (const auto& i : vecOdd) { cout << i << " "; }
```

输出如下所示:

```
Enter a number (0 to quit): 11
Enter a number (0 to quit): 22
Enter a number (0 to quit): 33
Enter a number (0 to quit): 44
Enter a number (0 to quit): 0
Even numbers: 22 44
Odd numbers: 11 33
```

`partition()` 算法对序列排序, 使谓词返回 `true` 的所有元素放在前面, 使谓词返回 `false` 的所有元素放在后面, 在每个分区中不保留元素最初的顺序。下例演示了如何把 `vector` 分为偶数在前、奇数在后的分区:

```
vector<int> values;
populateContainer(values);
partition(begin(values), end(values), [](int i){ return i % 2 == 0; });
cout << "Partitioned result: ";
for (const auto& i : values) { cout << i << " "; }
```

输出如下:

```
Enter a number (0 to quit): 55
Enter a number (0 to quit): 44
Enter a number (0 to quit): 33
Enter a number (0 to quit): 22
Enter a number (0 to quit): 11
```



```
Enter a number (0 to quit): 0
Partitioned result: 22 44 33 55 11
```

还有其他几个分区算法，参见第 16 章。

20.2.5 排序算法

标准库提供了一些不同的排序算法。排序算法重新排列容器中元素的顺序，使集合中的元素保持连续顺序。因此，排序算法只能应用于顺序集合。排序和有序关联容器无关，因为有序关联容器已经维护了元素的顺序。排序也和无序关联容器无关，因为无序关联容器就没有排序的概念。一些容器(例如 `list` 和 `forward_list`)提供了自己的排序方法，因为这些方法内部实现的效率比通用排序机制的效率要高。因此，通用的排序算法最适用于 `vector`、`deque`、`array` 和 C 风格数组。

`sort()` 函数一般情况下在 $O(N \log N)$ 时间内对某个范围内的元素排序。将 `sort()` 应用于一个范围之后，根据运算符 `operator<`，这个范围内的元素以非递减顺序排列(最低到最高)。如果不希望使用这个顺序，可以指定一个不同的比较回调，例如 `greater`。

`sort()` 函数的一个名为 `stable_sort()` 的变体能保持范围内相等元素的相对顺序。然而，由于这个算法需要维护范围内相等元素的相对顺序，因此这个算法比 `sort()` 算法低效。

下面是使用了 `greater` 比较器的 `sort()` 算法的一个示例：

```
vector<int> values;
populateContainer(values);
sort(begin(values), end(values), greater<>());
```

还有 `is_sorted()`，如果给定的范围是有序的，`is_sorted()` 就返回 `true`，而 `is_sorted_until()` 返回给定范围内的一个迭代器，该迭代器之前的所有元素都是有序的。

`nth_element()` 是个强大的选择算法，给定一个元素范围和一个指向该范围内的第 n 个元素的迭代器，该算法会重新排列该范围内的元素，这样如果整个范围都已排序，则第 n 个元素指向位置的元素就是那个元素。此外，它会重新排列所有元素，使第 n 个元素前面的所有元素都小于第 n 个新元素，而后面的所有元素都大于第 n 个新元素。这个算法的有趣之处在于它在线性时间 $O(n)$ 内完成这些工作，除了使用 `nth_element()`，还可以对整个范围进行排序然后检索感兴趣的元素，但这会导致线性对数复杂度 $O(n \log n)$ 。

这些听起来很复杂，来看看这个算法的实际应用，第一个示例是找到给定范围内第三大的元素，假设用户至少输出 3 个值。

```
vector<int> values;
populateContainer(values);
// Find the third largest value.
nth_element(begin(values), begin(values) + 2, end(values), greater<>());
cout << "3rd largest value: " << value[2] << endl;
```

另一个示例是按顺序从一个范围中获取 5 个最大的元素，它假设用户至少输入 5 个值。

```
vector<int> values;
populateContainer(values);
// Get the 5 largest elements in sorted order.
nth_element(begin(values), begin(values) + 4, end(values), greater<>());
// nth_element() has partitioned the elements, now sort the first subrange.
sort(begin(values), begin(values) + 5);
// And finally, output the sorted subrange.
for_each_n(begin(values), 5, [](const auto& element) { cout << element << " "; });
```

20.2.6 二分查找算法

有几个查找算法只用于有序序列或至少已分区的元素序列。这些算法有 `binary_search()`、`lower_bound()`、`upper_bound()`和 `equal_range()`。注意，关联容器(比如 `map` 和 `set`)具有应该使用的等效方法。用法示例参见第 18 章。

`lower_bound()`算法在有序范围内查找不小于(即大于或等于)给定值的第一个元素，经常用于发现在有序的 `vector` 中应将新值插入哪个位置，使 `vector` 依然有序。下面是一个示例：

```
vector<int> values;
populateContainer(values);

// Sort the container
sort(begin(values), end(values));

cout << "Sorted vector: ";
for (const auto& i : values) { cout << i << " "; }
cout << endl;

while (true) {
    int number;
    cout << "Enter a number to insert (0 to quit): ";
    cin >> number;
    if (number == 0) { break; }

    auto iter { lower_bound(begin(values), end(values), number) };
    values.insert(iter, number);

    cout << "New vector: ";
    for (const auto& i : values) { cout << i << " "; }
    cout << endl;
}
```

`binary_search()`算法以对数时间而不是线性时间搜索元素，需要指定范围的首尾迭代器、要搜索的值以及可选的比较回调。如果在指定范围内找到这个值，这个算法返回 `true`，否则返回 `false`。二分查找要求范围是有序的，首先比较范围的中间元素，根据中间元素是大于或者小于要查找的值，然后继续比较范围的左半部分或右半部分的中间元素，继续下去直到找到中间元素为止。基本数每次迭代范围都会减半，因此有对数复杂度。下面的例子演示了这个算法：

```
vector<int> values;
populateContainer(values);

// Sort the container
sort(begin(values), end(values));

while (true) {
    int num;
    cout << "Enter a number to find (0 to quit): ";
    cin >> number;
    if (number == 0) { break; }
    if (binary_search(cbegin(values), cend(values), number)) {
        cout << "That number is in the vector." << endl;
    } else {
        cout << "That number is not in the vector." << endl;
    }
}
```

20.2.7 集合算法

集合算法可用于任意有序范围。`includes()`算法实现了标准的子集判断功能，检查某个有序范围内的所有元素是否包含在另一个有序范围内，顺序任意。

`set_union()`、`set_intersection()`、`set_difference()`和`set_symmetric_difference()`算法实现了这些操作的标准语义。在集合论中，并集得到的结果是两个集合中的所有元素。交集得到的结果是所有同时存在于两个集合中的元素。差集得到的结果是所有存在于第一个集合中，但是不存在于第二个集合中的元素。对称差集得到的结果是两个集合的“异或”：所有存在于其中一个集合中，但不同时存在于两个集合中的元素。

警告：

务必确保结果范围足够大，以保存操作的结果。对于`set_union()`和`set_symmetric_difference()`，结果大小的上限是两个输入范围的总和。对于`set_intersection()`，结果大小的上限是两个输入范围的最小大小。对于`set_difference()`，结果大小的上限是第一个输入范围的大小。

警告：

不能使用关联容器(包括`set`)中的迭代器范围来保存结果，因为这些容器不允许修改键。

看一下这些集合算法的实际应用，首先`DumpRange()`是一个辅助函数模板，将给定范围内的元素写入标准输出流。实现方式如下：

```
template<typename Iterator>
void DumpRange(string_view message, Iterator begin, Iterator end)
{
    cout << message;
    for_each(begin, end, [](const auto& element) { cout << element << " "; });
    cout << endl;
}
```

定义了辅助函数后，下面是`set`算法的使用示例：

```
vector<int> vec1, vec2, result;
cout << "Enter elements for set 1:" << endl;
populateContainer(vec1);
cout << "Enter elements for set 2:" << endl;
populateContainer(vec2);

// set algorithms work on sorted ranges
sort(begin(vec1), end(vec1));
sort(begin(vec2), end(vec2));

DumpRange("Set 1: ", cbegin(vec1), cend(vec1));
DumpRange("Set 2: ", cbegin(vec2), cend(vec2));

if (includes(cbegin(vec1), cend(vec1), cbegin(vec2), cend(vec2))) {
    cout << "The second set is a subset of the first." << endl;
}
if (includes(cbegin(vec2), cend(vec2), cbegin(vec1), cend(vec1))) {
    cout << "The first set is a subset of the second" << endl;
}
```

```

result.resize(size(vec1) + size(vec2));
auto newEnd { set_union(cbegin(vec1), cend(vec1), cbegin(vec2),
    cend(vec2), begin(result)) };
DumpRange("The union is: ", begin(result), newEnd);

newEnd = set_intersection(cbegin(vec1), cend(vec1), cbegin(vec2),
    cend(vec2), begin(result));
DumpRange("The intersection is: ", begin(result), newEnd);

newEnd = set_difference(cbegin(vec1), cend(vec1), cbegin(vec2),
    cend(vec2), begin(result));
DumpRange("The difference between set 1 and 2 is: ", begin(result), newEnd);

newEnd = set_symmetric_difference(cbegin(vec1), cend(vec1),
    cbegin(vec2), cend(vec2), begin(result));
DumpRange("The symmetric difference is: ", begin(result), newEnd);

```

下面是这个程序的运行示例:

```

Enter elements for set 1:
Enter a number (0 to quit): 5
Enter a number (0 to quit): 6
Enter a number (0 to quit): 7
Enter a number (0 to quit): 8
Enter a number (0 to quit): 0
Enter elements for set 2:
Enter a number (0 to quit): 8
Enter a number (0 to quit): 9
Enter a number (0 to quit): 10
Enter a number (0 to quit): 0
Set 1: 5 6 7 8
Set 2: 8 9 10
The union is: 5 6 7 8 9 10
The intersection is: 8
The difference between set 1 and set 2 is: 5 6 7
The symmetric difference is: 5 6 7 9 10

```

`merge()` 算法可将两个排好序的范围归并在一起, 并保持排好的顺序。结果是一个包含两个源范围内所有元素的有序范围。这个算法的复杂度为线性时间。这个算法需要以下参数:

- 第一个源范围的首尾迭代器
- 第二个源范围的首尾迭代器
- 目标范围的起始迭代器
- (可选) 比较回调

如果没有 `merge()`, 还可通过串联两个范围, 然后对串联的结果应用 `sort()`, 以达到同样的目的, 但这样做的效率更低, 复杂度为 $O(N \log N)$ 而不是 `merge()` 的线性复杂度。

警告:

一定要确保提供足够大的目标范围, 以保存归并的结果。

下例演示了 `merge()` 算法:

```

vector<int> vectorOne, vectorTwo, vectorMerged;
cout << "Enter values for first vector:" << endl;

```

```

populateContainer(vectorOne);
cout << "Enter values for second vector:" << endl;
populateContainer(vectorTwo);

// Sort both containers
sort(begin(vectorOne), end(vectorOne));
sort(begin(vectorTwo), end(vectorTwo));

// Make sure the destination vector is large enough to hold the values
// from both source vectors.
vectorMerged.resize(size(vectorOne) + size(vectorTwo));

merge(cbegin(vectorOne), cend(vectorOne),
      cbegin(vectorTwo), cend(vectorTwo), begin(vectorMerged));

DumpRange("Merged vector: ", cbegin(vectorMerged), cend(vectorMerged));

```

20.2.8 最小最大算法

`min()`和`max()`算法通过运算符`operator<`或用户提供的二元谓词比较两个或多个任意类型的元素，分别返回一个引用较小或较大元素的`const`引用。`minmax()`算法返回一个包含两个或多个元素中最小值和最大值的`pair`。这些算法不接收迭代器参数。

`min_element()`和`max_element()`算法处理迭代器范围，并分别返回一个指向范围内最小或最大的迭代器。`minmax_element()`算法也用于迭代器范围，返回一对包含迭代器范围的对象，分别指向范围内最小和最大的元素。

下面的程序给出了一些示例：

```

int x { 4 }, y { 5 };
cout << format("x is {} and y is{}", x, y) << endl;
cout << "Max is " << max(x, y) << endl;
cout << "Min is " << min(x, y) << endl;

// Using max() and min() on more than two values
int x1 { 2 }, x2 { 9 }, x3 { 3 }, x4 { 12 };
cout << "Max of 4 elements is " << max({ x1, x2, x3, x4 }) << endl;
cout << "Min of 4 elements is " << min({ x1, x2, x3, x4 }) << endl;

// Using minmax()
auto p2 { minmax({ x1, x2, x3, x4 }) }; // p2 is of type pair<int, int>.
cout << format("Minmax of 4 elements is <{},{}>", p2.first, p2.second) << endl;

// Using minmax() + C++17 structured bindings
auto[min1, max1] { minmax({ x1, x2, x3, x4 }) };
cout << format("Minmax of 4 elements is <{},{}>", min1, max1) << endl;

// Using minmax_element() + C++17 structured bindings
vector values { 11, 33, 22 };
auto[min2, max2] { minmax_element(cbegin(values), cend(values)) };
cout << format("minmax_element() result: <{},{}>", *min2, *max2) << endl;

```

下面是这个程序的输出：

```

x is 4 and y is 5
Max is 5

```

```
Min is 4
Max of 4 elements is 12
Min of 4 elements is 2
Minmax of 4 elements is <2,12>
Minmax of 4 elements is <2,12>
minmax_element() result: <11,33>
```

注意:

你有时可能会遇到查找最大最小值的非标准宏。例如 GNU C Library(glibc)有宏 MIN()和 MAX(), Windows.h 头文件定义了宏 min()和 max(), 因为它们是宏, 所以可能对其参数进行二次求值; 而 std::min()和 std::max()对每个参数正好进行一次求值。确保总是使用 C++版本的 std::min()和 std::max()。当需要使用 std::min()和 std::max()时, 这些 min()和 max()宏可能会干扰你。此时, 可再加一对括号来禁用宏, 如下所示:

```
auto maxValue = {(std::max)(1, 2)};
```

在 Windows 上, 也可在添加 Windows.h 之前添加#define NOMINMAX, 以禁用 Windows min()和 max()宏。

std::clamp()是一个小型辅助函数, 在<algorithm>中定义, 可用于确保值(v)在给定的最小值(lo)和最大值(hi)之间。如果 v < lo, 它返回对 lo 的引用; 如果 v > hi, 它返回对 hi 的引用; 否则返回对 v 的引用。下面是一个例子:

```
cout << clamp(-3, 1, 10) << endl;
cout << clamp(3, 1, 10) << endl;
cout << clamp(22, 1, 10) << endl;
```

输出如下所示:

```
1
3
10
```

20.2.9 并行算法

对于 60 多种标准库算法, C++支持并行执行它们以提高性能, 示例包括 for_each()、all_of()、copy()、count_if()、find()、replace()、search()、sort()和 transform()等。支持并行执行的算法包含选项, 接收所谓的执行策略作为第一个参数。执行策略允许指定是否允许算法以并行方式或矢量方式执行。当编译器矢量化代码时, 它会用一个所谓的矢量 CPU 指令替换多个 CPU 指令。矢量指令用一条硬件指令对多个数据执行一些操作, 这些也称为单指令多数据(SIMD)指令。有四类标准执行策略, 以及这些类型相对应的全局实例, 它们全部定义在 std::execution 名称空间的<execution>中。如表 20-1 所示。

表 20-1 执行策略和全局实例

执行策略类型	全局实例	描述
sequenced_policy	seq	不允许算法并行执行
parallel_policy	par	允许算法并行执行
parallel_unsequenced_policy	par_unseq	允许算法并行执行和矢量化执行, 还允许在线程之间迁移执行
unsequenced_policy	unseq	允许矢量执行, 但不允许并行执行

也可以给标准库实现添加其他执行策略。

看看如何为算法指定执行策略，下面是一个使用并行策略对 `vector` 内容排序的示例：

```
sort(execution::par, begin(myVector), end(myVector));
```

警告：

传递给并行算法的回调函数不允许抛出任何未捕获的异常，这样做会触发对 `std::terminate()` 的调用来终止程序。

对算法使用 `parallel_unsequenced_policy` 或 `unsequenced_policy` 执行策略，以允许对回调进行交错函数调用，即不按顺序执行。这帮助编译器矢量化代码。然而，这也意味着会对函数回调施加诸多限制。例如，不能分配/释放内存、获取互斥以及使用非锁 `std::atomic` 等，见第 27 章“C++多线程编程”。对于其他标准策略，函数调用按顺序执行，但顺序无法确定。此类策略不会对函数调用操作施加限制。

并行算法未采取措施来避免数据争用和死锁，在并行执行算法时，由你来设法避免此类情况。第 27 章将讨论在多线程编程上下文中如何避免数据争用和死锁。

即便非并行版本的算法是 `constexpr`，但并行版本的算法不是。

与非并行版本相比，某些并行版本算法的返回类型可能略有不同。例如，`for_each()` 的非并行版本返回所提供的回调，而并行版本不返回任何东西。有关并行和非并行重载的所有算法的完整概述，包括它们的参数和返回类型，请参考标准库指南。

注意：

当处理大型数据集或必须对数据集中的每个单独元素执行大量工作时，可使用并行算法版本来提高性能。



20.2.10 约束算法

大多数算法在 `std::ranges` 名称空间中都有所谓的受约束版本。这些算法也在 `<algorithm>` 中定义，但与 `std` 名称空间中的等价算法不同，受约束的版本使用概念约束它们的模板类型参数，`concepts` 参见第 12 章“用模板编写泛型代码”。这意味着，如果传递无效参数，通常会从编译器得到更好的错误消息。例如 `sort()` 算法需要随机访问迭代器，将一对 `std::list` 迭代器作为参数传递给 `sort()` 可能会导致编译器产生一堆神秘的错误。而使用受约束的 `sort()` 算法，编译器会告诉传递的迭代器不是随机访问的。

这些约束算法的另一个好处是，它们可以处理作为起始和结束迭代器或范围的元素序列，此外它们还可以支持投影。范围和投影在第 17 章讨论。

例如，`std::ranges::find()` 算法与 `std::find()` 调用方式相同，即使用一对迭代器：

```
vector values {1, 2, 3};
auto result { ranges::find(cbegin(values), cend(values), 2) };
if (result != cend(values)) { cout << *result << endl; }
```

然而，如果想对容器中的所有元素都应用一种算法(这是常事)，总是必须指定 `begin/end` 迭代器来定义序列是相当乏味的。而有了范围的支持，就可以只使用单个参数指定一个范围，前面的 `find()` 调用可以写的代码更具可读性且更不容易出错，如下所示：

```
auto result { ranges::find(values, 2) };
```

这些受约束的算法不支持并行执行，因此它们不接受并行执行策略作为参数。

20.2.11 数值处理算法

前面介绍了数值处理算法的一个示例：`accumulate()`。本节介绍其他几个数值算法的示例。

1. `iota()`

`<numeric>`定义的 `iota()`算法会生成指定范围内的序列值，从给定的值开始，并应用 `operator++`生成每个后续值。下面的例子展示了如何将这个新算法用于整数的 `vector`，不过要注意这个算法可用于任意实现了 `operator++`的元素类型：

```
vector<int> values(10);
iota(begin(values), end(values), 5);
for (auto& i : values) { cout << i << " "; }
```

输出如下所示：

```
5 6 7 8 9 10 11 12 13 14
```

2. `reduce` 算法

标准库有 4 种 `reduce` 算法：`accumulate()`、`reduce()`、`inner_product()`和 `transform_reduce()`，它们都定义在`<numeric>`中。`accumulate()`算法在本章前面已经讨论过。所有的 `reduce` 算法重复地应用一个运算符来组合给定范围的两个元素或两个给定的范围，直到只剩下一个值。这些算法也被称为累加、聚合、压缩、注入或折叠算法。

`reduce`

不支持并行执行的算法很少，`std::accumulate()`就是其中之一。相反，需要使用新引入的 `std::reduce()` 算法，通过并行执行选项，计算广义和。

例如，以下两行同样是求和，但是 `reduce()`以并行和矢量化方式运行，因此速度更快，对于大型输入范围尤其如此：

```
double result1 = std::accumulate(cbegin(vec), cend(vec), 0.0);
double result2 = std::reduce(std::execution::par_unseq, cbegin(vec), cend(vec));
```

一般而言，`accumulate()`和 `reduce()`计算 $[x_0, x_n]$ 范围内元素的和，初始值为 `Init`，并且给定了二元运算符 Θ ：

$$\text{Init} \Theta x_0 \Theta x_1 \Theta \dots \Theta x_{n-1}$$

默认情况下，`accumulate()`的二元运算符是 `operator+`，`reduce()`的二元运算符是 `std::plus`。

`inner_product`

`<numeric>`中定义的 `inner_product()`算法计算两个序列的内积，例如，下面程序中的内积计算为 $(1*9)+(2*8)+(3*7)+(4*6)$ ，输出是 70：

```
vector<int> v1 { 1, 2, 3, 4 };
vector<int> v2 { 9, 8, 7, 6 };
cout << inner_product(cbegin(v1), cend(v1), cbegin(v2), 0) << endl;
```

`inner_product()`需要两个二元运算符，默认情况下分别是 `operator+`和 `operator*`，`inner_product()`是另一个不支持并行执行的算法，如果需要并行执行，请使用 `transform_reduce()`，对此下面会讨论。

transform_reduce

transform_reduce()支持并行执行，可以在单个元素范围或两个范围内执行。在第一个版本中，它计算 $[x_0, x_n]$ 范围内元素的和，初始值为 Init，并且给定了一元函数 f 以及二元运算符 Θ ：

$$\text{Init} \Theta f(x_0) \Theta f(x_1) \Theta \dots \Theta f(x_{n-1})$$

当在两个范围上执行时，它的行为与 inner_product()相同，但默认情况下它分别使用二元运算符 std::plus 和 std::multiplies，而不是 operator+和 operator*。

3. 扫描算法

扫描算法也被称为前缀和、累积和或部分和算法。将这种算法应用于一个范围的结果是另一个包含源范围各元素和的范围。

有 5 个扫描算法：exclusive_scan()、inclusive_scan()/partial_sum()、transform_exclusive_scan()和 transform_inclusive_scan()，它们全定义在<numeric>中。

表 20-2 中，针对 $[x_0, x_n]$ 元素范围，由 exclusive_scan()和 inclusive_scan()/partial_sum()计算和 $[y_0, y_n]$ ，初始值为 Init(partial_sum()为 0)，给定运算符 Θ 。

表 20-2 计算和 $[y_0, y_n]$

exclusive_scan()	inclusive_scan()/partial_sum()
$y_0 = \text{Init}$	$y_0 = \text{Init} \Theta x_0$
$y_1 = \text{Init} \Theta x_0$	$y_1 = \text{Init} \Theta x_0 \Theta x_1$
$y_2 = \text{Init} \Theta x_0 \Theta x_1$	...
...	...
$y_{n-1} = \text{Init} \Theta x_0 \Theta x_1 \Theta \dots \Theta x_{n-2}$	$y_{n-1} = \text{Init} \Theta x_0 \Theta x_1 \Theta \dots \Theta x_{n-1}$

transform_exclusive_scan()和 transform_inclusive_scan()在计算广义和之前，都首先给元素应用一元函数，这类似于 transform_reduce()在执行前给元素应用一元函数。

注意，这些扫描算法除了 partial_sum()，其他可接收可选的执行策略，以并行地执行。这些扫描算法的计算顺序不确定，partial_sum()和 accumulate()的顺序是从左到右；正因为如此，partial_sum()和 accumulate()无法并行化！

20.3 本章小结

本章提供了一些标准库算法的代码示例。将这些算法与 lambda 表达式相结合可以编写优雅且易于理解的代码。和前面的章节一样，希望读者可以欣赏到标准库容器和算法的有用性和强大功能。

后续章节将讨论 C++标准库的其他功能。第 21 章讨论正则表达式，第 22 章解释了日期和时间的支持，第 23 章展示了如何生成随机数，第 24 章讲解一些可供使用的其他库工具。第 25 章讲解一些高级特性，比如分配器以及如何编写自己的与标准库兼容的算法和容器。

20.4 练习

通过完成下面的习题，可以巩固本章讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己

找到答案。

练习 20-1 使用最喜欢的标准库指南查找 `fill()` 算法的参数。向用户请求一个数字，然后使用 `fill()` 用给定的数字填充一个包含 10 个整数的 `vector`，再将 `vector` 的内容写到标准输出以进行验证。

练习 20-2 回顾第 16 章“排列算法”一节，并使用标准库指南找到它们的参数。编写一个程序，要求用户输入一些数字，然后使用其中一种排列算法打印出这些数字所有可能的排列。

练习 20-3 编写一个名为 `trim()` 的函数，删除给定字符串开头和结尾的所有空格并返回结果。提示：检查字符 `c` 是否为空白字符，可以使用 `<cctype>` 中定义的 `std::isspace(c)`。如果 `c` 是一个空白字符，则返回非零值，否则返回 0。在 `main()` 函数中使用几个字符串测试你的实现。

练习 20-4 使用一种算法创建一个包含数字 1 到 100 的 `vector`。然后使用一个算法，将所有偶数和奇数拷贝到偶数和奇数容器中，而不对这些容器进行任何空间预留，确保偶数按升序排列，奇数按降序排列，谨慎地为偶数和奇数容器选择类型。提示：或许第 17 章有些东西可以用到。

第21章

字符串的本地化与正则表达式

本章内容

- 如何本地化应用程序以适合全球用户使用
- 如何通过正则表达式实现强大的模式匹配

本章的前半部分讨论本地化，本地化现在越来越重要，它允许编写为全世界不同地区进行本地化的软件。

本章的后半部分介绍正则表达式库，通过这个库很容易对字符串执行模式匹配。通过这个库，不仅可搜索匹配特定模式的子字符串，还可验证、解析和转换字符串。正则表达式非常强大，建议使用正则表达式，而不要自己编写字符串处理代码。

21.1 本地化

在学习 C 或 C++ 编程时，为方便学习，将字符等同于字节，把所有字符当成美国标准信息交换代码(ASCII)字符集中的成员。ASCII 是一个 7 位的集合，通常保存在 8 位的 `char` 类型中。在现实中，富有经验的 C++ 程序员意识到，成功的程序应该世界通用。即使程序一开始没有考虑到国际用户，也不应该在日后不考虑本地化或软件对本地语言的支持。

注意：

本章简要介绍本地化、不同字符编码以及字符串代码的可移植性。本书不详细讨论所有这些主题，无论要讲清楚其中的哪个主题，都需要一整本书。

21.1.1 宽字符

用字节表示字符的问题在于，并不是所有的语言(或字符集)都可以用 8 位(即 1 字节)来表示。C++ 有一种内建类型 `wchar_t`，可以保存宽字符(wide character)。带有非 ASCII(U.S.)字符的语言，例如日语和阿拉伯语，在 C++ 中可以用 `wchar_t` 表示。然而，C++ 标准并没有定义 `wchar_t` 的大小。一些编译器使用 16 位，而另一些编译器使用 32 位。为编写跨平台代码，将 `wchar_t` 假定为任何特定的数值都是不安全的。

如果软件可能会用在非西方字符集的环境中(注意: 一定会有!), 那么应该从一开始就使用宽字符。在使用 `wchar_t` 时, 需要在字符串和字符字面量的前面加上字母 `L`, 以表示应该使用宽字符编码。例如, 要将 `wchar_t` 字符初始化为字母 `m`, 应该编写以下代码:

```
wchar_t myWideCharacter { L'm' };
```

大部分常用类型和类都有宽字符版本。宽字符版本的 `string` 类为 `wstring`。“前缀字母 `w`”模式也可以应用于流。`wofstream` 处理宽字符文件输出流, `wifstream` 处理宽字符文件输入流。读这些类名(`woof-stream?``whiff-stream?`)有足够的理由让程序识别语言环境, 流已在第 13 章中详细讨论。

`cout`、`cin`、`cerr` 和 `clog` 也有宽字节版本: `wcout`、`wcin`、`wcerr` 和 `wclog`。这些版本的使用和非宽字节版本的使用没有区别:

```
wcout << L"I am a wide-character string literal." << endl;
```

21.1.2 本地化字符串字面量

本地化的一个关键点在于不能在源代码中放置任何本机语言的字符串字面量, 除非是面向开发人员的调试字符串。在 Microsoft Windows 应用程序中, 通过将程序的所有字符串放在 `STRINGTABLE` 资源中达到了这个目的。其他大部分平台都提供了类似的功能。如果需要将应用程序翻译为其他语言, 只要翻译那些资源即可, 而不需要修改任何源代码。有一些工具可以帮助完成翻译过程。

为使源代码能本地化, 不应该利用字符串字面量组成句子, 即使单独的字面量也可以被本地化。例如:

```
size_t n { 5 };
wstring filename { L"file1.txt" };
wcout << L"Read " << n << L" bytes from " << filename << endl;
```

这条语句不能本地化为中文, 因为中文的语序有所变化。中文的翻译如下(能否在标准输出控制台中正确地看到这些中文字符完全取决于系统配置):

```
wcout << L"从" << filename << L"中读取" << n << L"个字节" << endl;
```

为能正确地本地化这个字符串, 可采用下面的方式来实现:

```
cout << format(loadResource(IDS_TRANSFERRED), n, filename) << endl;
```

`IDS_TRANSFERRED` 是字符串资源表中一个条目的名称。对于英文版, `IDS_TRANSFERRED` 可定义为 `"Read {0} bytes from {1}"`; 对于中文版, 这条资源可以定义为 `"从{1}中读取{0}个字节"`。`loadResource()` 函数用给定的名称加载字符串资源, `format()` 用 `n` 的值替换 `{0}`, 用 `filename` 的值替换 `{1}`。

21.1.3 非西方字符集

宽字符是很大的进步, 因为宽字符增加了定义一个字符可用的空间。下一步是要解决如何利用这个空间的问题。在宽字符集中, 和 ASCII 一样, 字符用编号表示, 现在称为码点。唯一的区别在于编号不能放在 8 个位中。字符到码点的映射要大得多, 因为这个映射除了能够处理英语为母语的程序员所熟悉的字符外, 还处理很多不同的字符集。

国际标准 ISO 10646 定义的 Universal Character Set (UCS) 和 Unicode 都是标准化的字符集。这些字符集包含大约 10 万个抽象字符, 每个字符都由一个无歧义的名字和一个码点标识。两个标准中都有带有同样编号的相同字符, 并且都有可以使用的特定编码。例如, UTF-8 是 Unicode 编码的一个实

例, 其中 Unicode 字符编码为 1 到 4 个 8 位字节, UTF-16 将 Unicode 字符编码为一个或两个 16 位的值, UTF-32 将 Unicode 字符编码为正好 32 位。

不同应用程序可使用不同编码。遗憾的是, 本章前面提到过, C++ 标准并没有定义宽字符 `wchar_t` 的大小。在 Windows 平台上为 16 位, 在其他平台上可能为 32 位。在使用宽字符编写跨平台代码时, 应该注意到这一点。为解决这个问题, 可使用另外两个字符类型: `char16_t` 和 `char32_t`。下面的列表总结了支持的所有字符类型。

- `char`: 存储 8 位。可用于保存 ASCII 字符, 还可用作保存 UTF-8 编码的 Unicode 字符的基本构建块。使用 UTF-8 时, 一个 Unicode 字符编码为 1 到 4 个 `char`。
- `charx_t`: 存储至少 `x` 位, `x` 可以是 8(C++20)、16 或 32。这种类型可用作 UTF-`x` 编码的 Unicode 字符的基本构建块, 最多用 4 个 `char8_t`、2 个 `char16_t` 或 1 个 `char32_t` 编码一个 Unicode 字符。
- `wchar_t`: 保存一个宽字符, 宽字符的大小和编码取决于编译器。

使用 `charx_t` 而不是 `wchar_t` 的好处在于: 标准保证 `charx_t` 的最小大小, 它们的大小和编译器无关, 而 `wchar_t` 不能保证最小的大小。

使用字符串前缀可将字符串字面量转换为特定类型。下面列出所有支持的字符串前缀。

- `u8`: 采用 UTF-8 编码的 `char`(C++20 中是 `char8_t`) 字符串字面量。
- `u`: 采用 UTF-16 编码的 `char16_t` 字符串字面量(C++20 保证)。
- `U`: 采用 UTF-32 编码的 `char32_t` 字符串字面量(C++20 保证)。
- `L`: 采用编译器相关编码的 `wchar_t` 字符串字面量。

所有这些字符串字面量都可与第 2 章介绍的原始字符串字面量前缀 `R` 结合使用。例如:

```
const char* s1 { u8R"(Raw UTF-8 encoded string literal)" };
const wchar_t* s2 { LR"(Raw wide string literal)" };
const char16_t* s3 { uR"(Raw char16_t string literal)" };
const char32_t* s4 { UR"(Raw char32_t string literal)" };
```

如果通过 `u8` UTF-8 字符串字面量使用了 Unicode 编码, 那么在非原始字符串字面量中可通过 `\uABCD` 符号插入指定的 Unicode 码点。例如, `\u03C0` 表示 `pi` 字符, `\u00B2` 表示字符², 因此以下代码会打印出 πr^2 :

```
const char* formula { u8"\u03C0 r\u00B2" };
```

与此类似, 字符字面量也可具有前缀, 以转换为特定类型。支持前缀 `u8`、`u`、`U` 和 `L`, 一些例子有: `u'a`、`U'a`、`L'a` 和 `u8'a`。

除 `std::string` 类外, 目前还支持 `wstring`、`u8string`(C++20)、`u16string` 和 `u32string`。它们的定义如下:

- `using string = basic_string<char>;`
- `using wstring = basic_string<wchar_t>;`
- `using u8string = basic_string<char8_t>;`
- `using u16string = basic_string<char16_t>;`
- `using u32string = basic_string<char32_t>;`

类似地, 标准库提供了 `std::string_view`、`wstring_view`、`u8string_view`(C++20)、`u16string_view` 和 `u32string_view`, 这些都基于 `basic_string_view`。

多字节字符串使用与区域设置相关的编码, 是由一个或多个字节组成的字符串, 本章稍后将讨论区域设置。多字节字符串可以使用 Unicode 编码或任何其他类型的编码。下面的转换函数可在 `char8_t/char16_t/char32_t` 和多字节字符之间来回转换: (C++20)`mbrtoc8()` 和 `c8rtomb()`、`mbrtoc16()`、

c16rtomb()、mbrtoc32()和 c32rtomb()。

遗憾的是,对 char8_t、char16_t 和 char32_t 的支持就这么多了。有一些转换类可供使用(参见 21.1.4 节),但数量不多。例如,并没有支持这些字符类型的 cout 或 cin 版本,因此很难向控制台打印这种字符串,或从用户输入读入这种字符串。如果想要更多地使用这样的字符串,那么需要求助于第三方库。ICU(International Components for Unicode)是一个十分知名的库,可为应用程序提供 Unicode 和全球化支持。

21.1.4 locale 和 facet

不同国家间的数据表示中,字符集并不是唯一的不同之处。即使使用相似字符集的国家之间,例如英国和美国,也会存在数据表示的不同,例如日期和货币。

标准的 C++中,将一组特定的文化参数相关的数据组合起来的机制称为 locale。locale 中的独立组件,例如日期格式、时间格式和数字格式等称为 facet。U.S. English 是一个 locale 实例。显示日期时采用的格式是一个 facet 实例。有一些内建的 facet 是所有 locale 共用的。C++语言还提供了一种自定义和添加 facet 的方式。

有些第三方库可以更容易地处理 locale。boost 就是例子,它能使用 ICU 作为后端,支持排序和转换,将字符串转换为大写(不是一个字符一个字符地转换为大写)等等。

1. 使用 locale

使用 I/O 流时,根据特定的 locale 对数据进行格式化。locale 是可以关联到流的对象。locale 在 <locale>中定义。locale 的名字和具体的实现相关。POSIX 标准将语言和区域分隔在两字母的段中,再加上可选的编码。例如,美国使用的英语语言 locale 为 en_US,而英国使用的英语语言 locale 为 en_GB。日本使用的日语再加上 Japanese Industrial Standard 编码的 locale 为 ja_JPjis。

Windows 上的 locale 名称使用不同的格式。首选格式与 POSIX 格式十分类似,但用虚线替代下画线。次选格式是旧格式,方括号中内容可选,如下所示:

```
lang[_country_region[.code_page]]
```

表 21-1 列出了一些示例,首选 Windows 和旧 Windows 的 locale 格式。

表 21-1 Windows 和旧 Windows 的 locale 格式

语言	POSIX	Windows	Windows(旧式)
U.S. English	en_US	en-US	English_United States
Great Britain English	en_GB	en-GB	English_Great Britain

大部分操作系统都提供了一种根据用户的定义判断 locale 的机制。在 C++中,向 std::locale 对象的构造函数传入一个空的字符串,可以根据用户的环境创建 locale。一旦创建这个对象,就可以用它查询 locale,根据它做出一些程序的判断。下面的代码演示了如何在流上调用 imbue()方法,使用用户的 locale。结果就是所有发送到 wcout 的内容都会根据环境的格式化规则进行格式化。

```
wcout.imbue(locale { "" });  
wcout << 32767 << endl;
```

这意味着如果系统 locale 为美式英语,那么输出数字 32767 时会显示为 32,767;如果系统 locale 为比利时荷兰语,那么同样的数字会显示为 32.767。

默认 locale 通常是经典 locale, 不是用户的 locale。经典 locale 使用 ANSI C 风格的约定。经典 C locale 类似于 U.S. English, 但有一些细微区别。例如, 在输出数字时不会带任何标点符号:

```
wcout.imbue(locale { "C" });
wcout << 32767 << endl;
```

这段代码的输出如下所示:

```
32767
```

以下代码手工设置了美式英语 locale, 因此数字 32767 会通过美式英语标点格式化, 与系统 locale 无关:

```
wcout.imbue(locale { "en-US" }); // "en_US" for POSIX
wcout << 32767 << endl;
```

这段代码的输出如下所示:

```
32,767
```

可通过 locale 对象查询 locale 的信息。例如, 下面的程序创建了一个匹配用户环境的 locale。通过 name() 方法可得到描述这个 locale 的 C++ 字符串。然后, 通过 find() 方法在这个字符串中查找指定的子串。如果没有找到指定的子串, 则返回 string::npos。这段代码检查 Windows 标准和 POSIX 标准的名称。根据 locale 是否为美式英语, 这段程序输出两条信息中的一条:

```
locale loc { "" };
if (loc.name().find("en_US") == string::npos &&
    loc.name().find("en-US") == string::npos) {
    wcout << L"Welcome non-US. English speaker!" << endl;
} else {
    wcout << L"Welcome US English speaker!" << endl;
}
```

注意:

如果要数据写入一个文件, 而程序将从这个文件读回数据, 建议使用中性的 C locale; 否则, 将很难解析。另外, 在用户界面中显示数据时, 建议根据用户 locale 设置数据格式。

2. 全局 Locale

std::locale::global() 函数用于在应用程序中用给定的语言环境替换全局 C++ 语言环境。std::locale 的默认构造函数返回这个全局 locale 的拷贝。请记住, 使用 locale 的 C++ 标准库对象, 例如 cout 等流, 在构造时存储了全局 locale 的拷贝。之后更改全局 locale 不会影响之前创建的对象, 如果需要, 可以在构造后使用 imbue() 方法更改 locale。

下面是使用默认 locale 输出数字的示例, 将全局 locale 改为美式英语, 然后再次输出相同的数字:

```
void print()
{
    stringstream stream;
    stream << 32767;
    cout << stream.str() << endl;
}

int main()
{
    print();
}
```



```
    locale::global(locale { "en-US" }); // "en_US" for POSIX
    print();
}
```

输出如下：

```
32767
32,767
```

3. 字符分类

<locale>包含以下字符分类函数：std::isspace()、isblank()、iscntrl()、isupper()、islower()、isalpha()、isdigit()、ispunct()、isxdigit()、isalnum()、isprint()和 isgraph()。它们都接收两个参数：要分类的字符，以及用于分类的 locale。不同字符类的确切含义会在本章后面的正则表达式上下文中讨论，下面的 isupper()示例使用法语环境 locale：

```
bool result { isupper('E', locale { "fr-FR" }) }; // result = true
```

4. 字符转换

<locale>也定义了两个字符转换函数：std::toupper()和 tolower()。它们接收两个参数：要转换的字符，以及用于转换的 locale，示例如下：

```
auto upper { toupper('e', locale { "fr-FR" }) }; // é
```

5. 使用 facet

通过 std::use_facet()函数模板可获得特定 locale 中的某个特定 facet。模板类型参数指定要检索的 facet，而函数参数指定检索 facet 的 locale。例如，以下表达式通过 POSIX 标准的 locale 名称获得英语 locale 中的标准货币符号 facet：

```
use_facet<moneypunct<wchar_t>>(locale { "en_GB" });
```

注意，最内层的模板类型决定了要使用的字符类型。嵌套模板类的使用不合时宜，但不要管这些语法，其结果包含英国货币符号相关的所有信息。标准 facet 中的数据定义在<locale>中。表 21-2 列出了标准中定义的标准 facet 类别。可参阅标准库参考资源，以了解各个 facet 的详情。

表 21-2 标准 facet 类别

facet	描述
ctype	字符类别 facet
codecvt	转换 facet，见下节
collate	按字典顺序比较字符串
time_get	解析日期和时间
time_put	设置日期和时间格式
num_get	解析数字值
num_put	设置数字值的格式
numpunct	定义数字值的格式化参数
money_get	解析货币值
money_put	设置货币值的格式
moneypunct	定义货币值的格式化参数

下面的代码片段综合使用 `locale` 和 `facet`，输出了美式英语和英式英语中的货币符号。注意，根据环境配置，英国货币符号可能显示为问号或方框，或什么都不显示。如果环境能够处理这些符号，那么可得到英镑符号：

```
locale locUSEng { "en-US" };      // "en-US" For POSIX
locale locBritEng { "en-GB" };    // "en-GB For POSIX

wstring dollars { use_facet<moneypunct<wchar_t>>(locUSEng).curr_symbol() };
wstring pounds { use_facet<moneypunct<wchar_t>>(locBritEng).curr_symbol() };

wcout << L"In the US, the currency symbol is " << dollars << endl;
wcout << L"In Great Britain, the currency symbol is " << pounds << endl;
```

21.1.5 转换

C++标准提供 `codecvt` 类模板，以帮助在不同编码之间转换。`<locale>`定义了如表 21-3 所示的 4 个编码转换类。

表 21-3 编码转换类

类	描述
<code>codecvt<char, char, mbstate_t></code>	恒等转换，也就是无转换
<code>codecvt<char16_t, char, mbstate_t></code> <code>codecvt<char16_t, char8_t, mbstate_t></code>	UTF-16 和 UTF-8 之间的转换
<code>codecvt<char32_t, char, mbstate_t></code> <code>codecvt<char32_t, char8_t, mbstate_t></code>	UTF-32 和 UTF-8 之间的转换
<code>codecvt<wchar_t, char, mbstate_t></code>	宽字符编码(取决于实现)与窄字符编码之间的转换

但是，这些 `facet` 使用起来相当复杂。例如，下面的代码片段将窄字符串转换为宽字符串：

```
auto& facet { use_facet<codecvt<wchar_t, char, mbstate_t>>(locale { "" }) };
string narrowString { "Hello" };
mbstate_t mb { };
wstring wideString(narrowString.size(), '\0');
const char* fromNext { nullptr };
wchar_t* toNext { nullptr };
facet.in(mb,
    narrowString.data(), narrowString.data() + narrowString.size(), fromNext,
    wideString.data(), wideString.data() + wideString.size(), toNext);
wideString.resize(toNext - wideString.data());
wcout << wideString << endl;
```

在 C++17 之前，`<codecvt>`中定义了以下 3 种代码转换：`codecvt_utf8`、`codecvt_utf16` 和 `codecvt_utf8_utf16`。可通过两种简便的转换接口使用它们：`wstring_convert` 和 `wbuffer_convert`。C++17 不赞成使用这 3 个转换(整个`<codecvt>`头文件)和这两个简便接口，因此本书不再讨论。C++标准委员会决定不再使用该功能，因为它不能正确地处理错误。结构有误的 Unicode 字符串会带来安全风险，实际上，它们已被用作危害系统安全的攻击矢量。另外，API 过于晦涩，难以理解。在 C++标准委员会提出恰当的、安全的、易用的功能来替换不赞成使用的功能前，建议使用第三方库(如 ICU)正确处理 Unicode 字符串。

21.2 正则表达式

正则表达式在`<regex>`中定义，是标准库中字符串相关的一个强大特性。正则表达式是一种用于字符串处理的微型语言。尽管一开始看上去比较复杂，但一旦了解这种语言，字符串的处理就会简单得多。正则表达式适用于一些与字符串相关的操作。

- **验证**：检查输入字符串是否格式正确。例如：输入字符串是不是格式正确的电话号码？
- **决策**：判断输入表示哪种字符串。例如：输入字符串表示 JPEG 文件名还是 PNG 文件名？
- **解析**：从输入字符串中提取信息。例如：从日期中提取出年月日。
- **转换**：搜索子字符串，并将子字符串替换为新的格式化的子字符串。例如：搜索所有的“C++20”，并替换为“C++”。
- **遍历**：搜索所有子字符串。例如：从输入字符串中提取所有电话号码。
- **符号化**：根据一组分隔符将字符串分解为多个子字符串。例如：根据空白字符、逗号和句号等将字符串分割为独立的单词。

当然，还可自己编写代码，对字符串执行上述任何操作，但是强烈建议使用正则表达式特性，因为编写正确且安全的代码来处理字符串并不容易。

在深入介绍正则表达式的细节之前，需要介绍一些重要的术语。下面的术语贯穿于后面的讨论。

- **模式(pattern)**：正则表达式实际上是通过字符串表示的模式。
- **匹配(match)**：判断给定的正则表达式和给定序列`[first, last)`中的所有字符是否匹配。
- **搜索(search)**：判断在给定序列`[first, last)`中是否存在匹配给定正则表达式的子字符串。
- **替换(replace)**：在给定序列中识别子字符串，然后将子字符串替换为从其他模式计算得到的新子字符串，其他模式称为替换模式(substitution pattern)。

有几种不同的正则表达式语法。C++包含对以下几种语法的支持。

- **ECMAScript**：基于 ECMAScript 标准的语法。ECMAScript 是符合 ECMA-262 标准的脚本语言。JavaScript、ActionScript 和 Jscript 等语言都使用 ECMAScript 语言标准。
- **basic**：基本的 POSIX 语法。
- **extended**：扩展的 POSIX 语法。
- **awk**：POSIX awk 实用工具使用的语法。
- **grep**：POSIX grep 实用工具使用的语法。
- **egrep**：POSIX grep 实用工具使用的语法，包含 -E 参数。

如果已经了解了其中任何一种正则表达式语法，就可在 C++中立即使用这种语法，只需要告诉正则表达式库使用那种语法(syntax_option_type)。C++中的默认语法是 ECMAScript，21.2.1 节将详细讲解这种语法。这也是最强大的正则表达式语法，因此强烈建议使用 ECMAScript，而不要使用其他功能受限的语法。本书由于受篇幅限制，不再讲解其他正则表达式语法。

注意：

如果是第一次听说正则表达式，那么只要学习默认的 ECMAScript 语法即可。

21.2.1 ECMAScript 语法

正则表达式模式是一个字符序列，这种模式表达了要匹配的内容。正则表达式中的任何字符都表示匹配自己，但以下特殊字符除外：

```
^ $ \ . * + ? ( ) [ ] { } |
```

下面将逐一讲解这些特殊字符。如果需要匹配这些特殊字符，那么需要通过字符将其转义。例如：

`\[` 或 `\.` 或 `\*` 或 `\\`

1. 锚点

特殊字符`^`和`$`称为锚点(anchor)。`^`字符匹配行终止符前面的位置，`$`字符匹配行终止符所在的位置。`^`和`$`默认还分别匹配字符串的开头和结尾位置，但可以禁用这种行为。

例如，`^test$`只匹配字符串 `test`，不匹配包含 `test` 和其他任何字符的字符串，例如 `1test`、`test2` 和 `test abc` 等。

2. 通配符

通配符(wildcard)可用于匹配除换行符外的任意字符。例如，正则表达式 `a.c` 可以匹配 `abc` 和 `a5c`，但不匹配 `ab5c` 和 `ac`。

3. 替代

| 字符表示“或”的关系。例如，`a|b` 表示匹配 `a` 或 `b`。

4. 分组

圆括号`()`用于标记子表达式，子表达式也称为捕获组(capture group)。捕获组有以下用途：

- 捕获组可用于识别源字符串中单独的子序列，在结果中会返回每一个标记的子表达式(捕获组)。以如下正则表达式为例：`(.)(ab|cd)(.)`。其中有 3 个标记的子表达式。对字符串 `1cd4` 运行 `regex_search()`，执行这个正则表达式会得到含有 4 个条目的匹配结果。第一个条目是完整匹配 `1cd4`，接下来的 3 个条目是 3 个标记的子表达式。这 3 个条目为：`1`、`cd` 和 `4`。
- 捕获组可在匹配的过程中用于后向引用(back reference)的目的(后面解释)。
- 捕获组可在替换操作的过程中用于识别组件(后面解释)。

5. 重复

使用以下 4 个重复字符可重复匹配正则表达式中的部分模式：

- `*`匹配零次或多次之前的部分。例如：`a*b` 可匹配 `b`、`ab`、`aab` 和 `aaaab` 等字符串。
- `+`匹配一次或多次之前的部分。例如：`a+b` 可匹配 `ab`、`aab` 和 `aaaab` 等字符串，但不能匹配 `b`。
- `?`匹配零次或一次之前的部分。例如：`a?b` 匹配 `b` 和 `ab`，不能匹配其他任何字符串。
- `{...}`表示区间重复。`a{n}`重复匹配 `a` 正好 `n` 次；`a{n,}`重复将 `a` 匹配 `n` 次或更多次；`a{n,m}` 重复将 `a` 匹配 `n` 到 `m` 次，包含 `n` 次和 `m` 次。例如，`^a{3,4}$`可以匹配 `aaa` 和 `aaaa`，但不能匹配 `a`、`aa` 和 `aaaaa` 等字符串。

重复匹配字符称为贪婪匹配，因为这些字符可以找出最长匹配，但仍匹配正则表达式的其余部分。为进行非贪婪匹配，可在重复字符的后面加上一个`?`，例如`*?`、`+`、`??`和`{...}?`。非贪婪匹配将其模式重复尽可能少的次数，但仍匹配正则表达式的其余部分。

例如，表 21-4 列出了贪婪匹配和非贪婪匹配的正则表达式，以及在输入序列 `aaabbb` 上运行它们后得到的子字符串。

表 21-4 正则表达式

正则表达式	匹配的子字符串
贪婪匹配: <code>(a+)(ab)*(b+)</code>	"aaa" " " "bbb"
非贪婪匹配: <code>(a+?)(ab)*(b+)</code>	"aa" "ab" "bb"

6. 优先级

与数学公式一样，正则表达式中元素的优先级也很重要。正则表达式的优先级如下。

- 元素：例如 `a`，是正则表达式最基本的构建块。
- 量词：例如 `+`、`*`、`?` 和 `{...}`，紧密绑定至左侧的元素，例如 `b+`。
- 串联：例如 `ab+c`，在量词之后绑定。
- 替代符：例如 `|`，最后绑定。

例如正则表达式 `ab+c|d`，它匹配 `abc`、`abbc` 和 `abbbc` 等字符串，还能匹配 `d`。圆括号可以改变优先级顺序。例如，`ab+(c|d)` 可以匹配 `abc`、`abbc`、`abbbc`、`....`、`abd`、`abbd` 和 `abbbd` 等字符串。不过，如果使用了圆括号，也意味着将圆括号内的内容标记为子表达式或捕获组。使用 `(?:...)`，可以在避免创建新捕获组的情况下修改优先级。例如，`ab+(?:c|d)` 和之前的 `ab+(c|d)` 匹配的内容是一样的，但没有创建多余的捕获组。

7. 字符集合匹配

`(a|b|c|...|z)` 这种表达式既冗长，又会引入捕获组，为了避免这种正则表达式，可以使用一种特殊的语法，指定一组字符或字符的范围。此外，还可以使用“否定”形式的匹配。在方括号之间指定字符集合，`[c1c2...cn]` 可以匹配字符 `c1`，`c2`，...，`cn` 中的任意字符。例如，`[abc]` 可以匹配 `a`、`b` 和 `c` 中的任意字符。如果第一个字符是 `^`，那么表示“除了这些字符之外的任意字符”：

- `ab[cde]` 匹配 `abc`、`abd` 和 `abe`。
- `ab[^cde]` 匹配 `abf` 和 `abp` 等字符串，但不匹配 `abc`、`abd` 和 `abe`。

如果需要匹配 `^`、`[` 或 `]` 字符本身，需要转义这些字符，例如：`[\\[\\^]]` 匹配 `[`，`^` 或 `]`。

如果想要指定所有字母，可编写下面这样的字符集合：`[abcdefghijklmnopqrstuvwxyzABCEFGHIJKLMNOPQRSTUVWXYZ]`，但这种写法非常冗长，这样的模式出现多次的话，看上去会很优雅，甚至有可能出现拼写错误或不小心漏掉一个字母。这个问题有两种解决方案。

一种方案是使用方括号内的范围描述，这允许使用 `[a-zA-Z]` 这样的表达方式，这种表达方式能识别 `a` 到 `z` 和 `A` 到 `Z` 范围内的所有字母。如果需要匹配连字符，则需要转义这个字符，例如 `[a-zA-Z\\-]` 匹配任意单词，包括带连字符的单词。

另一种方案是使用某种字符类(character class)。字符类表示特定类型的字符，表示方法为 `[.name:]`，可使用什么字符类取决于 `locale`，但表 21-5 中的名称总是可以识别的。这些字符类的含义也取决于 `locale`。这个表假定使用标准的 C `locale`。

表 21-5 字符类别名称

字符类别名称	说明
digit	数字
d	同 digit
xdigit	数字和下列十六进制数字使用的字母：'a'、'b'、'c'、'd'、'e'、'f'、'A'、'B'、'C'、'D'、'E'、'F'
alpha	字母数字字符。对于 C locale，这些是所有的小写和大写字母
alnum	alpha 类和 digit 类的组合
w	同 alnum
lower	小写字母(假定适用于 locale)
upper	大写字母(假定适用于 locale)
blank	空白字符是在一行文本中用于分割单词的空格符，对于 C locale，就是 ' ' 或 '\t'

(续表)

字符类别名称	说明
space	空白字符，对于 C locale，就是' '、'\t'、'\n'、'\r'、'\f'和'\f'
s	同 space
print	可打印字符。它们占用打印位置，例如在显示器上。与控制符(ctrl)相反，示例有小写字母、大写字母、数字、标点符号字符和空白字符
cntrl	控制符，与可打印字符(print)相反，不占用打印位置，例如在显示器上。对于 C locale，示例有换页符'\f'、换行符'\n'和回车符'\r'等
graph	带有图形表示的字符，包括除空格' '外的所有可打印字符(print)
punct	标点符号字符。对于 C locale，包括不是字母数字(alnum)的所有图形字符(graph)，例如'!'、'#'、'@'、'}'等

字符类用在字符集中，例如，英语中的`[[alpha:]]*`等同于`[a-zA-Z]*`。

由于有些概念使用非常频繁，例如匹配数字，因此这些字符类有缩写模式。例如，`[:digit:]`和`[d:]`等同于`[0-9]`。有些类甚至有更短的使用转义符号的模式。例如，`\d`表示`[:digit:]`。因此，通过以下任意模式可以识别一个或多个数字序列：

- `[0-9]+`
- `[[digit:]]+`
- `[[d:]]+`
- `\d+`

表 21-6 列出了字符类可用的转义符号。

表 21-6 转义符号

转义符号	等价于
<code>\d</code>	<code>[[d:]]</code>
<code>\D</code>	<code>[^[:d:]]</code>
<code>\s</code>	<code>[[s:]]</code>
<code>\S</code>	<code>[^[:s:]]</code>
<code>\w</code>	<code>[[:w:]]</code>
<code>\W</code>	<code>[^[:w:]]</code>

下面举一些示例：

- `Test[5-8]`匹配 `Test5`、`Test6`、`Test7` 和 `Test8`。
- `[[lower:]]`匹配 `a` 和 `b` 等，但不匹配 `A` 和 `B` 等。
- `[[:lower:]]`匹配除了小写字母(例如 `a` 和 `b` 等)之外的任意字符。
- `[[lower:5-7]]`匹配任意小写字母，例如 `a` 和 `b` 等，还匹配数字 `5`、`6` 和 `7`。

8. 词边界

词边界(word boundary)的意思可能是：

- 单词的第一个字符，这个字符是单词字符之一，而且之前的字符不是单词字符。单词字符是字母、数字或者下划线，对于标准的 C locale，这等于`[A-Za-z0-9_]`。

- 单词的结尾字符，这是单词字符之后的非单词字符，之前的字符是单词字符。
- 如果源字符串的第一个字符在单词字符(即字母、数字或下画线)之后，则表示源字符串的开头位置。匹配源字符串的开头位置默认为启用，但也可以禁用(regex_constants::match_not_bow, bow 表示字母开头)。
- 如果源字符串的最后一个字符是单词字符之一，则表示源字符串的结束位置。匹配源字符串的结束位置默认为启用，但也可以禁用(regex_constants::match_not_eow, eow 表示字母结尾)。

通过\b 可匹配单词边界，通过\B 匹配除单词边界外的任何内容。

9. 后向引用

通过后向引用(back reference)可引用正则表达式本身的捕获组：\n 表示第 n 个捕获组，且 n>0。

例如，正则表达式(d+)-.*\1 匹配以下格式的字符串：

- 在一个捕获组中(d+)捕获的一个或多个数字
- 接下来是一个连字符-
- 接下来是 0 个或多个字符.*
- 接下来是另一个连字符-
- 接下来是第一个捕获组捕获到的相同数字\1

这个正则表达式能匹配 123-abc-123 和 1234-a-1234 等字符串，但不能匹配 123-abc-1234 和 123-abc-321 等字符串。

10. lookahead

正则表达式支持正向 lookahead(?=模式)和负向 lookahead(?!模式)。lookahead 后面的字符必须匹配(正向)或不匹配(负向)lookahead 模式，但这些字符还没有使用。

例如，a(?!b)模式包含一个负向 lookahead，以匹配后面不跟 b 的字母。a(?=b)模式包含一个正向 lookahead，以匹配后跟 b 的字母，但不使用 b，b 不是匹配的一部分。

下面是一个更复杂的示例。正则表达式匹配一个输入序列，该输入序列至少包含一个小写字母、至少一个大写字母、至少一个标点符号，并且至少 8 个字符长。例如，可使用下面这样的正则表达式来强制密码满足特定条件。

```
(?=.*[[:lower:]]) (?=.*[[:upper:]]) (?=.*[[:punct:]]) .{8,}
```

在本章最后的练习中，可尝试使用这个密码验证正则表达式。

11. 正则表达式和原始字符串字面量

从前面的讨论可以看出，正则表达式经常使用很多应该在普通 C++ 字符串字面量中转义的特殊字符。例如，如果在正则表达式中写一个\d，这个\d 能匹配任何数字。然而，由于\是 C++ 中的一个特殊字符，因此需要在正则表达式的字符串字面量中将其转义为\\d，否则 C++ 编译器会试图将其解释为d。如果需要正则表达式匹配单个反斜杠\，那么会更加麻烦。因为\是正则表达式语法本身的一个特殊字符，所以应该将其转义为\\；而\在 C++ 字符串字面量中也是一个特殊字符，因而还需要在 C++ 字符串字面量中进行转义，最终得到\\。

使用原始字符串字面量可使 C++ 源代码中的复杂正则表达式更容易阅读。第 2 章讲解了原始字符串字面量。例如以下正则表达式：

```
"( \\n|\\r|\\\\\\)"
```


这个正则表达式搜索空格、换行符、回车符和反斜杠。从中可看出，这个正则表达式需要使用很多转义字符。使用原始字符串字面量，这个正则表达式可替换为以下更便于阅读的版本：

```
R"(( |\n|\r|\\))"
```

原始字符串字面量以 `R`(开头，以`"`结束。开头和结尾之间的所有内容都是正则表达式。当然，在最后还需要双反斜杠，因为反斜杠在正则表达式本身中需要转义。

12. 常用正则表达式

编写正确的正则表达式并不总是很简单。对于常见的模式，例如验证密码、电话号码、社会安全号码、IP 地址、邮件地址、信用卡号码和日期等，不需要这么做。如果使用最喜欢的搜索引擎在线搜索正则表达式，会找到几个具有预定义模式集合的网站，例如 regexr.com、regex101.com 和 regextester.com 等。这些网站中有一部分允许在线测试这些预定义的模式，甚至是自己的模式，因此可以在代码中使用它们之前轻松地验证它们是否正确。

以上就是对 ECMAScript 语法的简单介绍。下面开始讲解如何在 C++ 代码中真正使用正则表达式。

21.2.2 regex 库

正则表达式库的所有内容都在 `<regex>` 中和 `std` 名称空间中。正则表达式库中定义的基本模板类型包括如下几种。

- **basic_regex**: 表示某个特定正则表达式的对象。
- **match_results**: 匹配正则表达式的子字符串，包括所有的捕获组。它是 `sub_match` 的集合。
- **sub_match**: 包含输入序列中一个迭代器对的对象，这些迭代器表示匹配的特定捕获组。迭代器对中的一个迭代器指向匹配的捕获组中的第一个字符，另一个迭代器指向匹配的捕获组中最后一个字符后面的那个字符。它的 `str()` 方法把匹配的捕获组返回为字符串。

`regex` 库提供了 3 个关键算法：`regex_match()`、`regex_search()` 和 `regex_replace()`。所有这些算法都有不同的版本，不同的版本允许将源字符串指定为 STL 字符串、C 风格字符串或表示开始和结束的迭代器对。迭代器可以具有以下类型：

- `const char*`
- `const wchar_t*`
- `string::const_iterator`
- `wstring::const_iterator`

事实上，可使用任何具有双向迭代器行为的迭代器。第 17 章(理解迭代器和范围库)更深入地讨论了迭代器。

`regex` 库还定义了以下两类正则表达式迭代器，这两类正则表达式迭代器适合于查找源字符串中的所有模式。

- **regex_iterator**: 遍历一个模式在源字符串中出现的所有位置。
- **regex_token_iterator**: 遍历一个模式在源字符串中出现的所有捕获组。

为方便 `regex` 库的使用，C++ 标准定义了很多属于以上模板的类型别名，如下所示：

```
using regex = basic_regex<char>;
using wregex = basic_regex<wchar_t>;

using csub_match = sub_match<const char*>;
using wsub_match = sub_match<const wchar_t*>;
using ssub_match = sub_match<string::const_iterator>;
```



```

using wssub_match = sub_match<wstring::const_iterator>;

using cmatch = match_results<const char*>;
using wcmatch = match_results<const wchar_t*>;
using smatch = match_results<string::const_iterator>;
using wsmatch = match_results<wstring::const_iterator>;

using cregex_iterator = regex_iterator<const char*>;
using wcregex_iterator = regex_iterator<const wchar_t*>;
using sregex_iterator = regex_iterator<string::const_iterator>;
using wsregex_iterator = regex_iterator<wstring::const_iterator>;

using cregex_token_iterator = regex_token_iterator<const char*>;
using wcregex_token_iterator = regex_token_iterator<const wchar_t*>;
using sregex_token_iterator = regex_token_iterator<string::const_iterator>;
using wsregex_token_iterator = regex_token_iterator<wstring::const_iterator>;

```

下面将讲解 `regex_match()`、`regex_search()`和 `regex_replace()`算法以及 `regex_iterator` 和 `regex_token_iterator` 类。

21.2.3 `regex_match()`

`regex_match()`算法可用于比较给定的源字符串和正则表达式模式。如果正则表达式模式匹配整个源字符串，则返回 `true`，否则返回 `false`。这个算法很容易使用。`regex_match()`算法有 6 个版本，这些版本接收不同类型的参数。它们都使用如下形式：

```

template<...>
bool regex_match(InputSequence[, MatchResults], RegEx[, Flags]);

```

`InputSequence` 可以表示为：

- 源字符串的首尾迭代器
- `std::string`
- C 风格的字符串

可选的 `MatchResults` 参数是对 `match_results` 的引用，它接收匹配。如果 `regex_match()`返回 `false`，就只能调用 `match_results::empty()`或 `match_results::size()`，其余内容都未定义。如果 `regex_match()`返回 `true`，表示找到匹配，可以通过 `match_results` 对象查看匹配的具体内容。具体方法稍后用示例说明。

`RegEx` 参数是需要匹配的正则表达式。可选的 `Flags` 参数指定匹配算法的选项。大多数情况下，可使用默认选项。更多细节可参阅标准库参考资料，见附录 B。

`regex_match()`示例

假设要编写一个程序，要求用户输入采用以下格式的日期：年/月/日，其中年是 4 位数，月是 1 到 12 之间的数字(包括 1 和 12)，日是 1 到 31 之间的数字(包括 1 和 31)。通过正则表达式和 `regex_match()` 算法可以验证用户的输入，如下所示：

```

regex r { "\\d{4}/(?:0?[1-9]|1[0-2])/(?:0?[1-9]|[1-2][0-9]|3[0-1])" };
while (true) {
    cout << "Enter a date (year/month/day) (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q") { break; }

    if (regex_match(str, r)) { cout << " Valid date." << endl; }
}

```

```
    else { cout << " Invalid date!" << endl; }
}
```

第一行创建了一个正则表达式，它由三部分组成，这三部分通过斜杠字符/隔开，分别表示年、月、日。下面解释这三部分。

- `\d{4}`: 这部分匹配任意 4 位数的组合，例如 1234 和 2010 等。
- `(?:0?[1-9]|1[0-2])`: 正则表达式的这一部分包括在括号中，从而确保正确的优先级。这里不需要使用任何捕获组，因此使用了`(?:...)`。内部的表达式由|字符分隔的两部分组成。
 - `0?[1-9]`: 匹配 1 到 9 之间的任何数字(包括 1 和 9)，前面有一个可选的 0。例如，可以匹配 1、2、9、03 和 04 等。不匹配 0、10 和 11 等。
 - `1[0-2]`: 只能匹配 10、11 和 12，不能匹配除此之外的其他任何字符串。
- `(?:0?[1-9]|1[2][0-9]|3[0-1])`: 这一部分也包括在非捕获组中，由 3 个可选的部分组成。
 - `0?[1-9]`: 匹配 1 到 9 之间的任何数字(包括 1 和 9)，前面有一个可选的 0。例如，可以匹配 1、2、9、03 和 04 等。不匹配 0、10 和 11 等。
 - `1[2][0-9]`: 匹配 10 和 29 之间的任何数字(包括 10 和 29)，不能匹配除此之外的其他任何字符串。
 - `3[0-1]`: 只能匹配 30 和 31，不能匹配除此之外的其他任何字符串。

这个例子然后进入一个无限循环，要求用户输入一个日期。将接下来输入的每一个日期都传入 `regex_match()` 算法。当 `regex_match()` 返回 `true` 时，表示用户输入的日期匹配正则表达式的日期模式。

这个例子可稍做扩充，要求 `regex_match()` 算法在结果对象中返回捕获到的子表达式。为理解这段代码，首先要理解捕获组的作用。通过指定 `match_results` 对象，例如调用 `regex_match()` 时指定的 `smatch`，正则表达式匹配字符串时会将 `match_results` 对象中的元素填入。为提取这些子字符串，必须使用括号创建捕获组。

`match_results` 对象中的第一个元素`[0]`包含匹配整个模式的字符串。在使用 `regex_match()` 且找到匹配时，这就是整个源序列。在使用 19.2.4 节讲解的 `regex_search()` 时，这表示源序列中匹配正则表达式的一个子字符串。元素`[1]`是第一个捕获组匹配的子字符串，`[2]`是第二个捕获组匹配的子字符串，以此类推。为获得捕获组的字符串表示，可像下面的代码这样编写 `m[i]` 或 `m[i].str()`。

如下代码将年、月、日提取到 3 个独立的整型变量中。修改后的例子中的正则表达式有一些微小变化。匹配年的第一部分被放在捕获组中，匹配月和日的部分现在也在捕获组中，而不在非捕获组中。调用 `regex_match()` 时提供了 `smatch` 参数，现在这个参数会包含匹配的捕获组。下面是修改后的示例：

```
regex r { "(\\d{4})/(0?[1-9]|1[0-2])/(0?[1-9]|1[2][0-9]|3[0-1])" };
while (true) {
    cout << "Enter a date (year/month/day) (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q") { break; }

    if (smatch m; regex_match(str, m, r)) {
        int year { stoi(m[1]) };
        int month { stoi(m[2]) };
        int day { stoi(m[3]) };
        cout << format(" Valid date: Year={}, month={}, day={}",
            year, month, day) << endl;
    } else {
        cout << " Invalid date!" << endl;
    }
}
```

在这个例子中，`smatch` 结果对象中有 4 个元素。

- [0]: 匹配整个正则表达式的字符串，在这个例子中就是完整的日期
- [1]: 年
- [2]: 月
- [3]: 日

执行这个例子，可得到以下输出：

```
Enter a date (year/month/day) (q=quit): 2011/12/01
Valid date: Year=2011, month=12, day=1
Enter a date (year/month/day) (q=quit): 11/12/01
Invalid date!
```

注意：

这个日期匹配示例只检查日期是否由年(4 位数)、月(1~12)、日(1~31)组成，但没有对是否为闰年、月份中的天数是否正确等进行验证。如果需要执行这些验证，还必须编写代码，对 `regex_match()` 提取出来的年、月、日进行验证。如果在代码中验证年、月、日，那么可以简化正则表达式：

```
regex r { "(\\d{4})/(\\d{1,2})/(\\d{1,2})" };
```

21.2.4 regex_search()

如果整个源字符串匹配正则表达式，那么前面介绍的 `regex_match()` 算法返回 `true`，否则返回 `false`。如果要搜索匹配的子字符串，需要使用 `regex_search()`。`regex_search()` 算法有 6 个不同版本。它们都具有如下形式：

```
template<...>
bool regex_search(InputSequence[, MatchResults], RegEx[, Flags]);
```

在输入字符串中找到匹配时，所有变体返回 `true`，否则返回 `false`；参数类似于 `regex_match()` 的参数。

有两个版本的 `regex_search()` 算法接收要处理的字符串的首尾迭代器。你可能想在循环中使用 `regex_search()` 的这个版本，通过操作每个 `regex_search()` 调用的首尾迭代器，找到源字符串中某个模式的所有实例。千万不要这样做！如果正则表达式中使用了锚点(^或\$)和单词边界等，这样的程序会出问题。由于空匹配，这样会产生无限循环。根据本章后面讲解的内容，使用 `regex_iterator` 或 `regex_token_iterator` 在源字符串中提取出某个模式的所有实例。

警告：

绝对不要在循环中通过 `regex_search()` 在源字符串中搜索一个模式的所有实例。要改用 `regex_iterator` 或 `regex_token_iterator`。

regex_search() 示例

`regex_search()` 算法可在输入序列中提取匹配的子字符串。下例从输入的代码行中提取代码注释。正则表达式搜索的子字符串以 `//` 开头，然后跟上一些可选的空白字符 `\s*`，之后是一个或多个在捕获组中捕获的字符 `(+)`。这个捕获组只能捕获注释子字符串。`smatch` 对象 `m` 将收到搜索结果。如果成功，`m[1]` 包含找到的注释。可检查 `m[1].first` 和 `m[1].second` 迭代器，以确定注释在源字符串中的准确位置。

```
regex r("//\\s*(.+)$");
while (true) {
    cout << "Enter a string with optional code comments (q=quit): ";
    string str;
```

```

if (!getline(cin, str) || str == "q") { break; }

if (smatch m; regex_search(str, m, r))
    cout << format("{} Found comment '{}'", m[1].str()) << endl;
else
    cout << " No comment found!" << endl;
}

```

该程序的输出如下所示:

```

Enter a string (q=quit): std::string str; // Our source string
Found comment 'Our source string'
Enter a string (q=quit): int a;           // A comment with // in the middle
Found comment 'A comment with           // in the middle'
Enter a string (q=quit): float f;         // A comment with a      (tab) character
Found comment 'A comment with a      (tab) character'

```

`match_results` 对象还有 `prefix()` 和 `suffix()` 方法, 这两个方法分别返回这个匹配之前和之后的字符串。

21.2.5 regex_iterator

根据前面的解释, 绝对不要在循环中通过 `regex_search()` 获得模式在源字符串中的所有实例。应改用 `regex_iterator` 或 `regex_token_iterator`。这两个迭代器和标准库容器的迭代器类似。

regex_iterator 示例

下面的例子要求用户输入源字符串, 然后从源字符串中提取出所有的单词, 最后将单词打印在引号之间。这个例子中的正则表达式为 `[w]+`, 以搜索一个或多个单词字母。这个例子使用 `std::string` 作为来源, 所以使用 `sregex_iterator` 作为迭代器。这里使用了标准的迭代器循环, 但是在这个例子中, 尾迭代器的处理和普通标准库容器的尾迭代器稍有不同。一般情况下, 需要为某个特定的容器指定尾迭代器, 但对于 `regex_iterator`, 只有一个 `end` 迭代器。通过默认构造一个 `regex_iterator`, 就可获得这个尾迭代器。

for 循环创建了一个首迭代器 `iter`, 它接收源字符串的首尾迭代器以及正则表达式作为参数。每次找到匹配时调用循环体, 在这个例子中是每个单词。`sregex_iterator` 遍历所有的匹配。通过解引用 `sregex_iterator`, 可得到一个 `smatch` 对象。访问这个 `smatch` 对象的第一个元素 `[0]` 可得到匹配的子字符串:

```

regex reg { "[\\w]+" };
while (true) {
    cout << "Enter a string to split (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q") { break; }

    const sregex_iterator end;
    for (sregex_iterator iter { cbegin(str), cend(str), reg };
         iter != end; ++iter) {
        cout << format("{}{}{}", (*iter)[0].str()) << endl;
    }
}

```

这个程序的输出如下所示:

```

Enter a string to split (q=quit): This, is a test.

```

```
"This"
"is"
"a"
"test"
```

从这个例子中可以看出，即使是简单的正则表达式，也能执行强大的字符串操作。

注意，`regex_iterator` 和 `regex_token_iterator` (下一节讨论) 在内部都包含一个指向给定正则表达式的指针。它们都显式删除接收右值正则表达式的构造函数，防止使用临时 `regex` 对象构建它们。例如，下面的代码无法编译：

```
for (sregex_iterator iter { cbegin(str), cend(str), regex { "[\\w]+" } };
    iter != end; ++iter) { ... }
```

21.2.6 regex_token_iterator

21.2.5 节讲解了 `regex_iterator`，这个迭代器遍历每个匹配的模式。在循环的每次迭代中都得到一个 `match_results` 对象，通过这个对象可提取出捕获组捕获的那个匹配的子表达式。

`regex_token_iterator` 可用于在所有匹配的模式中自动遍历所有的或选中的捕获组。`regex_token_iterator` 有 4 个构造函数，格式如下：

```
regex_token_iterator(BidirectionalIterator a,
                    BidirectionalIterator b,
                    const regex_type& re
                    [, SubMatches
                    [, Flags]]);
```

所有构造函数都需要把首尾迭代器作为输入序列，还需要一个正则表达式。可选的 `SubMatches` 参数用于指定应迭代哪个捕获组。可以用 4 种方式指定 `SubMatches`：

- 一个整数，表示要迭代的捕获组的索引。
- 一个 `vector`，其中的整数表示要迭代的捕获组的索引。
- 带有捕获组索引的 `initializer_list`。
- 带有捕获组索引的 C 风格数组。

忽略 `SubMatches` 或把它指定为 0 时，获得的迭代器将遍历索引为 0 的所有捕获组，这些捕获组是匹配整个正则表达式的子字符串。可选的 `Flags` 参数指定匹配算法的选项。大多数情况下，可以使用默认选项。更多细节可参阅标准库参考资料。

regex_token_iterator 示例

可用 `regex_token_iterator` 重写前面的 `regex_iterator` 示例，如下所示。注意，与 `regex_iterator` 示例一样，在循环体中使用 `*iter->str()` 而非 `(*iter)[0].str()`，因为使用 `submatch` 的默认值 0 时，记号迭代器会自动遍历索引为 0 的所有捕获组。这段代码的输出和 `regex_iterator` 示例完全一致：

```
regex reg { "[\\w]+" };
while (true) {
    cout << "Enter a string to split (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q") { break; }

    const sregex_token_iterator end;
    for (sregex_token_iterator iter { cbegin(str), cend(str), reg };
        iter != end; ++iter) {
```

```

        cout << format("{}{}{}", iter->str()) << endl;
    }
}

```

下面的示例要求用户输入一个日期, 然后通过 `regex_token_iterator` 遍历第二个和第三个捕获组(月和日), 这是通过整数 `vector` 指定的。本章已经解释了用于日期的正则表达式。唯一的区别是添加了 `^` 和 `$` 锚点, 以匹配整个源序列。前面的示例不需要它们, 因为使用了 `regex_match()`, 这会自动匹配整个输入字符串。

```

regex reg { "^(\\d{4})/(0?[1-9]|1[0-2])/(0?[1-9]|[1-2][0-9]|3[0-1])$" };
while (true) {
    cout << "Enter a date (year/month/day) (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q") { break; }

    vector<int> indices{ 2, 3 };
    const sregex_token_iterator end;
    for (sregex_token_iterator iter { cbegin(str), cend(str), reg, indices };
        iter != end; ++iter) {
        cout << format("{}{}{}", iter->str()) << endl;
    }
}

```

这段代码只打印合法日期的月和日。这个例子的输出如下所示:

```

Enter a date (year/month/day) (q=quit): 2011/1/13
"1"
"13"
Enter a date (year/month/day) (q=quit): 2011/1/32
Enter a date (year/month/day) (q=quit): 2011/12/5
"12"
"5"

```

`regex_token_iterator` 还可用于执行字段分解(field splitting)或标记化(tokenization)这样的任务。使用这种方法比使用 C 语言中的旧式 `strtok()` 函数(之前没有进一步讨论)更加灵活和安全。标记化是在 `regex_token_iterator` 构造函数中通过将要遍历的捕获组索引指定为 -1 触发的。在标记化模式中, 迭代器会遍历源字符串中不匹配正则表达式的所有子字符串。下面的代码演示了这个过程, 这段代码根据前后带有任意数量的空白字符的分隔符(,)和(;)对一个字符串进行标记化操作。

```

regex reg { R"(\s*[;,]\s*)" };
while (true) {
    cout << "Enter a string to split on ',' and ';' (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q") { break; }

    const sregex_token_iterator end;
    for (sregex_token_iterator iter { cbegin(str), cend(str), reg, -1 };
        iter != end; ++iter) {
        cout << format("{}{}{}", iter->str()) << endl;
    }
}

```

这个例子中的正则表达式被指定为源字符串字面量, 搜索匹配以下内容的模式:

- 0 个或多个空白字符
- 后面跟着,或;字符

- 后面跟着 0 个或多个空白字符

输出如下所示：

```
Enter a string to split on ', ' and ';' (q=quit): This is,  a; test string.
"This is"
"a"
"test string."
```

从输出可以看出，对字符串根据,和;进行了分割，而且,和;周围所有的空白字符都被删除了，因为标记化迭代器遍历所有不匹配正则表达式的子字符串，正则表达式匹配的是前后带有空白字符的,和;。

21.2.7 regex_replace()

regex_replace()算法要求输入一个正则表达式，以及一个用于替换匹配子字符串的格式化字符串。这个格式化字符串可通过表 21-7 中的转义序列，引用匹配子字符串中的部分内容。

表 21-7 转义序列

转义序列	替换为
\$n	匹配第 <i>n</i> 个捕获组的字符串，例如\$1 表示第一个捕获组，\$2 表示第二个捕获组，以此类推； <i>n</i> 必须大于 0
\$&	匹配整个正则表达式的字符串
\$`	在输入序列中，在匹配正则表达式的子字符串左侧的部分
\$'	在输入序列中，在匹配正则表达式的子字符串右侧的部分
\$\$	单个美元符号

regex_replace()算法有 6 个不同版本。这些版本之间的区别在于参数的类型。其中的 4 个版本使用如下格式：

```
string regex_replace(InputSequence, RegEx, FormatString[, Flags]);
```

这 4 个版本都在执行替换操作后返回得到的字符串。InputSequence 和 FormatString 可以是 std::string 或 C 风格的字符串。RegEx 参数是需要匹配的正则表达式。可选的 Flags 参数指定替换算法的选项。

regex_replace()算法的另外两个版本采用如下形式：

```
OutputIterator regex_replace(OutputIterator,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             RegEx, FormatString[, Flags]);
```

这两个版本把得到的字符串写入给定的输出迭代器，并返回这个输出迭代器。输入序列给定为首尾迭代器。其他参数与 regex_replace()的另外 4 个版本相同。

regex_replace()示例

第一个例子的源 HTML 字符串是<body><h1>Header</h1><p>Some text</p></body>，正则表达式为<h1>(.*)</h1><p>(.*)</p>。表 21-8 展示了不同的转义序列以及替换后的文字。

表 21-8 转义序列和替换后的文字

转义序列	替换为
\$1	Header
\$2	Some text

(续表)

转义序列	替换为
<code>\$&</code>	<code><h1>Header</h1><p>Some text</p></code>
<code>\$'</code>	<code><body></code>
<code>\$'</code>	<code></body></code>

下面的代码演示了 `regex_replace()` 的使用：

```
const string str { "<body><h1>Header</h1><p>Some text</p></body>" };
regex r { "<h1>(.*?)</h1><p>(.*?)</p>" };

const string replacement { "H1=$1 and P=$2" }; // See above table
string result { regex_replace(str, r, replacement) };

cout << format("Original string: '{}'", str) << endl;
cout << format("New string      : '{}'", result) << endl;
```

这个程序的输出如下所示：

```
Original string: '<body><h1>Header</h1><p>Some text</p></body>'
New string      : '<body>H1=Header and P=Some text</body>'
```

`regex_replace()` 算法接收一系列改变行为的标志。表 21-9 列出了最重要的标志。

表 21-9 最重要的标志

标志	说明
<code>format_default</code>	默认操作是替换模式的所有实例，并将所有不匹配模式的内容复制到结果字符串
<code>format_no_copy</code>	默认操作是替换模式的所有实例，但是不将所有不匹配模式的内容复制到结果字符串
<code>format_first_only</code>	只替换模式的第一个实例

下例将此前的代码片段中的 `regex_replace` 调用改为使用 `format_no_copy` 标志：

```
string result { regex_replace(str, r, replacement,
                             regex_constants::format_no_copy) };
```

输出如下所示。可将下列输出和前一个版本的输出进行比较。

```
Original string: '<body><h1>Header</h1><p>Some text</p></body>'
New string      : 'H1=Header and P=Some text'
```

另一个例子是接收一个输入字符串，然后将每个单词边界替换为一个换行符，使目标字符串在每一行只包含一个单词。下面的例子演示了这一点，但没有使用任何循环来处理给定的输入字符串。这段代码首先创建一个匹配单个单词的正则表达式。当发现匹配时，匹配字符串被替换为 `$1\n`，其中 `$1` 将被匹配的单词替代。还要注意，这里使用了 `format_no_copy` 标志以避免将空白字符和其他非单词字符从源字符串复制到输出。

```
regex reg { "([\\w]+)" };
const string replacement { "$1\n" };
while (true) {
    cout << "Enter a string to split over multiple lines (q=quit): ";
    string str;
    if (!getline(cin, str) || str == "q") { break; }

    cout << regex_replace(str, reg, replacement,
                          regex_constants::format_no_copy) << endl;
}
```


这个程序的输出如下所示：

```
Enter a string to split over multiple lines (q=quit): This is a test.
This
is
a
test
```

21.3 本章小结

本章指明在编写代码时要考虑到本地化。任何经历过本地化的人都知道，如果提前规划好了(例如使用 Unicode 字符并注意使用 locale)，那么加入新的语言或 locale 支持会简单得多。

本章接下来讲解正则表达式库。理解了正则表达式的语法后，字符串操作会简单得多。通过正则表达式可轻松验证字符串、在输入字符串中搜索子字符串、对字符串执行查找替换操作等。强烈建议了解正则表达式，并开始使用正则表达式，而不是编写自己的字符串操作例程。正则表达式会让编程工作更简单。

21.4 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

练习 21-1 使用适当的 facet 计算根据用户环境格式化数字的十进制分隔符，可能需要查看标准库指南来了解所选 facet 的确切方法。

练习 21-2 编写一个程序，要求用户输入美国格式的电话号码。例如：202-555-0108。使用正则表达式验证电话号码的格式，即 3 个数字后面跟着破折号，另外 3 个数字，另外一个破折号和最后 4 个数字。如果电话号码有效，打印这三部分到单独的行。例如前面的电话号码，结果一定是：

```
202
555
0108
```

练习 21-3 编写一个程序，向用户请求一段可以跨越多行并且包含//风格注释的源代码。可以使用一个哨兵字符，例如@，来表示输入的结束。可以使用带有'@'分隔符的 std::getline() 从标准控制台读取多行文本。最后，使用正则表达式删除代码片段的所有行中的注释。确保代码在下面这行代码中正确工作：

```
string str; // A comment // Some more comments.
```

练习 21-4 本章前面的 Lookahead 小节提高过密码验证正则表达式。写一个程序来测试这个正则表达式。要求用户输入密码并进行验证。一旦验证通过，添加一个新的验证规则：密码也必须包含至少两个数字。

第22章

日期和时间工具

本章内容

- 编译期有理数的使用
- 时间的使用
- 日期和日历的使用
- 如何转换不同时区的时间点

本章讨论 C++ 标准库中的一些附加库功能，因为这些内容不适合放在其他章节。本章讨论 C++ 标准库提供的与时间相关的功能，统称为 `chrono` 库。它是用于处理时间和日期的类和函数的集合。这些库包含下列组件：

- 持续时间(Duration)
- 时钟(Clock)
- 时间点(Time point)
- 日期(Date)(C++20)
- 时区(Time zone)(C++20)

所有内容都在 `std::chrono` 名称空间中的 `<chrono>` 中定义。然而开始讨论这些 `chrono` 库组件之前，先稍微跑题，看一下 C++ 中可用的编译期有理数(rational number)的支持，因为这在 `chrono` 库中被大量使用。

22.1 编译期有理数

可通过 `ratio` 库精确地表示任何可在编译期使用的有限有理数。所有内容都在 `<ratio>` 中定义，并且都在 `std` 名称空间中。有理数的分子和分母通过类型为 `std::intmax_t` 的编译期常量表示，这是一种有符号的整数类型，其最大宽度由编译器指定。由于这些有理数编译期的特性，它们在使用时看上去比较复杂，不同寻常。`ratio` 对象的定义方式和普通对象的定义方式不同，而且不能调用 `ratio` 对象的方法。需要使用类型别名。例如，下面这行代码定义了一个表示 $1/60$ 的有理数编译期常量：

```
using r1 = ratio<1, 60>;
```

`r1` 有理数的分子(num)和分母(den)是编译期常量, 可通过以下方式访问:

```
intmax_t num { r1:num };
intmax_t den { r1:den };
```

记住 `ratio` 是一个编译期常量, 也就是说, 分子和分母需要在编译时确定。下面的代码会产生编译错误:

```
intmax_t n { 1 };           // Numerator
intmax_t d { 60 };          // Denominator
using r1 = ratio<n, d>;     // Error
```

将 `n` 和 `d` 定义为常量就不会有编译错误了:

```
const intmax_t n { 1 };     // Numerator
const intmax_t d { 60 };    // Denominator
using r1 = ratio<n, d>;     // Ok
```

有理数总是标准化的。对于有理数 `ratio<n, d>`, 计算最大公约数 `gcd`、分子 `num` 和分母 `den` 的定义如下:

- `num = sign(n)*sign(d)*abs(n)/gcd`
- `den = abs(d)/gcd`

`ratio` 库支持有理数的加法、减法、乘法和除法运算。由于所有这些操作都是在编译期进行的, 因此不能使用标准的算术运算符, 而应使用特定的模板和类型别名组合。可用的算术 `ratio` 模板包括 `ratio_add`、`ratio_subtract`、`ratio_multiply` 和 `ratio_divide`, 分别执行加法、减法、乘法和除法。这些模板将结果计算为新的 `ratio` 类型。这种类型可通过名为 `type` 的内嵌类型别名访问。例如, 下面的代码首先定义了两个 `ratio` 对象, 一个表示 `1/60`, 另一个表示 `1/30`。`ratio_add` 模板将两个有理数相加, 得到的 `result` 有理数应该是化简之后的 `1/20`。

```
using r1 = ratio<1, 60>;
using r2 = ratio<1, 30>;
using result = ratio_add<r1, r2::type>;
```

C++标准还定义了一些 `ratio` 比较模板: `ratio_equal`、`ratio_not_equal`、`ratio_less`、`ratio_less_equal`、`ratio_greater` 和 `ratio_greater_equal`。与算术 `ratio` 模板一样, `ratio` 比较模板也是在编译期求值的。这些比较模板创建了一种新类型 `std::bool_constant` 来表示结果。`bool_constant` 也是 `std::integral_constant`, 即 `struct` 模板, 里面保存了一种类型和一个编译期常量值。例如, `integral_constant<int, 15>` 保存了一个值为 `15` 的整型值。`bool_constant` 还是布尔类型的 `integral_constant`。例如, `bool_constant<true>` 是 `integral_constant<bool, true>`, 存储值为 `true` 的布尔值。`ratio` 比较模板的结果要么是 `bool_constant<bool, true>`, 要么是 `bool_constant<bool, false>`。与 `bool_constant` 或 `integral_constant` 关联的值可通过 `value` 数据成员访问。下面的代码演示了 `ratio_less` 的使用:

```
using r1 = ratio<1, 60>;
using r2 = ratio<1, 30>;
using res = ratio_less<r2, r1>;
cout << res::value << endl;
```

下面的例子整合了所有内容。注意, 由于 `ratio` 是编译期常量, 因此不能编写像 `cout << r1` 这样的代码, 而是需要获得分子和分母并分别进行打印。

```
// Define a compile-time rational number.
using r1 = ratio<1, 60>;
```

```
// Get numerator and denominator.
intmax_t num { r1::num };
intmax_t den { r1::den };
cout << format("1) r1 = {}/{}", num, den) << endl;

// Add two rational numbers.
using r2 = ratio<1, 30>;
cout << format("2) r2 = {}/{}", r2::num, r2::den) << endl;
using result = ratio_add<r1, r2>::type;
cout << format("3) sum = {}/{}" result::num, result::den) << endl;

// Compare two rational numbers.
using res = ratio_less<r2, r1>;
cout << format("4) r2 < r1: {}", res::value) << endl;
```

输出如下所示:

```
1) r1 = 1/60
2) r2 = 1/30
3) sum = 1/ 20
4) r2 < r1: false
```

为方便起见, `ratio` 库还提供了一些 SI(国际单位制)的类型别名, 如下所示:

```
using yocto = ratio<1, 1'000'000'000'000'000'000'000'000>; // *
using zepto = ratio<1, 1'000'000'000'000'000'000'000'000>; // *
using atto = ratio<1, 1'000'000'000'000'000'000'000>;
using femto = ratio<1, 1'000'000'000'000'000'000>;
using pico = ratio<1, 1'000'000'000'000'000>;
using nano = ratio<1, 1'000'000'000'000>;
using micro = ratio<1, 1'000'000>;
using milli = ratio<1, 1'000>;
using centi = ratio<1, 100>;
using deci = ratio<1, 10>;
using deca = ratio<10, 1>;
using hecto = ratio<100, 1>;
using kilo = ratio<1'000, 1>;
using mega = ratio<1'000'000, 1>;
using giga = ratio<1'000'000'000, 1>;
using tera = ratio<1'000'000'000'000, 1>;
using peta = ratio<1'000'000'000'000'000, 1>;
using exa = ratio<1'000'000'000'000'000'000, 1>;
using zetta = ratio<1'000'000'000'000'000'000'000, 1>; // *
using yotta = ratio<1'000'000'000'000'000'000'000'000, 1>; // *
```

只有在编译器能表示类型别名为 `intmax_t` 的常量分子和分母值时, 才会定义结尾处标了星号的 SI 单位。本章后面讨论 `duration` 时会给出这些预定义 SI 单位的使用示例。

22.2 持续时间

持续时间(`duration`)表示的是两个时间点之间的间隔时间, 通过模板化的 `duration` 类来表示。`duration` 类保存了滴答数和滴答周期(tick period)。滴答周期指的是两个滴答之间的秒数, 是一个编译期 `ratio` 常量, 也就是说可以是 1 秒的分数。`duration` 模板接收两个模板参数, 定义如下所示:

```
template <class Rep, class Period = ratio<1>> class duration {...}
```

第一个模板参数 Rep 表示保存滴答数的变量类型，应该是一种算术类型，例如 long 和 double 等。第二个模板参数 Period 是表示滴答周期的有理数常量。如果不指定滴答周期，那么会使用默认值 ratio<1>，也就是说默认滴答周期为 1 秒。

duration 类提供了 3 个构造函数：一个是默认构造函数；另一个构造函数接收一个表示滴答数的值作为参数；第三个构造函数接收另一个 duration 作为参数。后者可用于将一个 duration 转换为另一个 duration，例如将分钟转换为秒。本节后面会列举一个示例。

duration 支持算术运算，例如+、-、*、/、%、++、--、+=、-=、*=、/=和%=，还支持比较运算符==和<=。duration 类包含多个方法，如表 22-1 所示。

表 22-1 duration 类的方法

方法	说明
Rep count() const	以滴答数返回 duration 值，返回类型是 duration 模板中指定的类型参数
static duration zero()	返回持续时间值等于 0 的 duration
static duration min()	返回 duration 模板指定的类型参数表示的最小值/最大值持续时间的 duration 值
static duration max()	

库添加了用于持续时间的 floor()、round()和 abs()操作，行为与用于数值数据时类似。

下面看一下如何在实际代码中使用 duration。每一个滴答周期为 1 秒的 duration 定义如下所示：

```
duration<long> d1;
```

由于 ratio<1>是默认的滴答周期，因此这行代码等同于：

```
duration<long, ratio<1>> d1;
```

下面的代码指定了滴答周期为 1 分钟的 duration(滴答周期为 60 秒)：

```
duration<long, ratio<60>> d2;
```

下面的代码定义了每个滴答周期为 1/60 秒的 duration：

```
duration<double, ratio<1, 60>> d3;
```

根据本章前面的描述，<ratio>中定义了一些 SI 有理数常量。这些预定义的常量在定义滴答周期时非常方便。例如，下面这行代码定义了每个滴答周期为 1 毫秒的 duration：

```
duration<long long, milli> d4;
```

看下 duration 的应用，下面的例子展示了 duration 的几个方面。它展示了如何定义 duration，如何对 duration 执行算术操作，以及如何将一个 duration 转换为另一个滴答周期不同的 duration：

```
// Specify a duration where each tick is 60 seconds
duration<long, ratio<60>> d1 { 123 };
cout << d1.count() << endl;

// Specify a duration represented by a double with each tick
// equal to 1 second and assign the largest possible duration to it.
auto d2 { duration<double>::max() };
cout << d2.count() << endl;

// Define 2 durations
```

```

// For the first duration, each tick is 1 minute
// For the second duration, each tick is 1 second
duration<long, ratio<60>> d3 { 10 }; // = 10 minutes
duration<long, ratio<1>> d4 { 14 }; // = 14 seconds

// Compare both durations
if (d3 > d4){ cout << "d3 > d4" << endl; }
else { cout << "d3 <= d4" << endl; }

// Increment d4 with 1 resulting in 15 seconds
++d4;

// Multiply d4 by 2 resulting in 30 seconds
d4 *= 2;

// Add both durations and store as minutes
duration<double, ratio<60>> d5 { d3 + d4 };

// Add both durations and store as seconds
duration<long, ratio<1>> d6 { d3 + d4 };
cout << format("{} minutes + {} seconds = {} minutes or {} seconds",
    d3.count(), d4.count(), d5.count(), d6.count()) << endl;

// Create a duration of 30 seconds
duration<long> d7 { 30 };

// Convert the seconds of d7 to minutes
duration<double, ratio<60>> d8 { d7 };
cout << format("{} seconds = {} minutes", d7.count(), d8.count()) << endl;

```

输出如下所示:

```

123
1.79769e+308
d3 > d4
10 minutes + 30 seconds = 10.5 minutes or 630 seconds
30 seconds = 0.5 minutes

```

注意:

上面输出中的第二行表示类型 `double` 能表示的最大 `duration`。具体的值因编译器而异。

特别注意下面两行:

```

duration<double, ratio<60>> d5 { d3 + d4 };
duration<long, ratio<1>> d6 { d3 + d4 };

```

这两行都计算了 `d3+d4`, `d3` 以分钟为单位, `d4` 以秒为单位, 但第一行将结果保存在表示分钟的浮点值中, 第二行将结果保存在表示秒的整数中。分钟到秒的转换(或秒到分钟的转换)自动进行。

上例中的下面两行展示了如何在不同时间单位间进行显式转换:

```

duration<long> d7 { 30 }; // seconds
duration<double, ratio<60>> d8 { d7 }; // minutes

```

第一行定义了一个滴答周期为 30 秒的 `duration`。第二行将这 30 秒转换为 0.5 分钟。使用这个方向的转换可能会得到非整数, 因此要求使用浮点数类型表示的 `duration`; 否则会得到一些很诡异的

编译错误。例如，下面的代码不能成功编译，因为 d8 使用了 long 类型而不是浮点类型：

```
duration<long> d7 { 30 };           // seconds
duration<long, ratio<60>> d8 { d7 }; // minutes // Error!
```

但可以使用 `duration_cast()` 进行强制转换：

```
duration<long> d7 { 30 };           // seconds
auto d8 { duration_cast<duration<long, ratio<60>>>(d7) }; // minutes
```

此处，d8 的滴答周期为 0 分钟，因为整数除法用于将 30 秒转换为分钟数。

在另一个方向的转换中，如果源数据是整数类型，那么不要求转换至浮点类型。因为如果从整数值开始转换，那么得到的总是整数值。例如，下面的代码将 10 分钟转换为秒数，两者都用整数类型 long 表示：

```
duration<long, ratio<60>> d9 { 10 }; // minutes
duration<long> d10 { d9 };           // seconds
```

chrono 库还提供了以下标准的 duration 类型，它们位于 `std::chrono` 名称空间中：

```
using nanoseconds = duration<X 64 bits, nano>;
using microseconds = duration<X 55 bits, micro>;
using milliseconds = duration<X 45 bits, milli>;
using seconds      = duration<X 35 bits>;
using minutes      = duration<X 29 bits, ratio<60>>;
using hours        = duration<X 23 bits, ratio<3600>>;
```

C++20 增加了下面的类型：

```
using days = duration<X 25 bits, ratio_multiply<ratio<24>, hours::period>>;
using weeks = duration<X 22 bits, ratio_multiply<ratio<7>, days::period>>;
using years = duration<X 17 bits,
    ratio_multiply<ratio<146'097, 400>, days::period>>;
using months = duration<X 20 bits, ratio_divide<years::period, ratio<12>>>;
```

X 的具体类型取决于编译器，但 C++ 标准要求 X 的类型为至少指定大小的整数类型。上面列出的类型别名使用了本章前面描述的预定义的 SI ratio 类型别名。使用这些预定义的类型，不是编写：

```
duration<long, ratio<60>> d9 { 10 }; // minutes
```

而是编写：

```
minutes d9 { 10 }; // minutes
```

下面是一个讲解如何使用这些预定义 duration 的例子。这段代码首先定义了一个变量 t，这个变量保存的是 1 小时+23 分钟+45 秒的结果。这里使用了 auto 关键字，让编译器自动推导出 t 的准确类型。第二行使用了预定义的 seconds duration 构造函数，将 t 的值转换为秒数，并将结果输出到控制台：

```
auto t { hours { 1 } + minutes { 23 } + seconds { 45 } };
cout << format("{} seconds", seconds { t }.count()) << endl;
```

由于 C++ 标准要求预定义的 duration 使用整数类型，因此如果转换后得到非整数的值，那么会出现编译错误。尽管整数除法通常都会截断，但在使用通过 ratio 类型实现的 duration 时，编译器会将所有可能导致非零余数的计算声明为编译期错误。例如，下面的代码无法编译，因为转换 90 秒后得到的是 1.5 分钟：

```
seconds s { 90 };
minutes m { s };
```


下面的代码也无法成功编译，即使 60 秒刚好为 1 分钟也是如此。这段代码也会产生编译错误，因为从秒到分钟的转换可能会产生非整数：

```
seconds s { 60 };
minutes m { s };
```

另一个方向的转换可正常完成，因为 `minutes duration` 是整数值，将它转换为 `seconds` 总能得到整数值：

```
minutes m { 2 };
seconds s { m };
```

可使用标准的用户自定义字面量 `h`、`min`、`s`、`ms`、`us` 和 `ns` 来创建 `duration`。从技术角度看，这些定义在 `std::literals::chrono_literals` 名称空间中，就像第 2 章中讨论的标准用户自定义字面量一样，`chrono_literals` 名称空间是内联名称空间。因此，可以使用下面任何一个 `using` 指令来使用 `chrono` 字面量：

```
using namespace std;
using namespace std::literals;
using namespace std::chrono_literals;
using namespace std::literals::chrono_literals;
```

此外，字面量在 `std::chrono` 名称空间中也可用，下面是示例：

```
using namespace std::chrono;
// ...
auto myDuration { 42min }; // 42 minutes
```

C++20 将 `hh_mm_ss` 类模板添加到 `chrono` 库中。它的构造函数接收 `Duration` 并将其分为小时、分钟、秒和子秒(subseconds)。它有 `hours()`、`minutes()`、`seconds()` 和 `subseconds()` 来检索数据，总是返回非负值。如果持续时间为负数，那么 `is_negative()` 方法返回 `true`，否则返回 `false`。本章末尾的练习中会使用 `hh_mm_ss` 类模板。

22.3 时钟

`clock` 类由 `time_point` 和 `duration` 组成。`time_point` 类在 22.4 节详细讨论，不过理解 `clock` 的工作方式并不需要这些细节。但由于 `time_point` 本身依赖于 `clock`，因此应该首先详细了解 `clock` 的工作方式。

C++ 标准定义了一些 `clock`，其中 3 个最重要的是 `system_clock`、`steady_clock` 和 `high_resolution_clock`。第一个称为 `system_clock`，表示来自系统实时时钟的真实时间。第二个称为 `steady_clock`，是一个能保证其 `time_point` 绝不递减的时钟。`system_clock` 无法做出这种保证，因为系统时钟可以随时调整。第三个称为 `high_resolution_clock`，这个时钟的滴答周期达到了最小值。`high_resolution_clock` 可能就是 `stead_clock` 或 `system_clock` 的别名，具体取决于编译器。

注意：

C++20 增加了几个新的时钟：`utc_clock`、`tai_clock`、`gps_clock` 和 `file_clock`。这些是更高级的内容，本文不再进一步讨论。

每个 clock 都有一个静态的 now()方法, 用于把当前时间用作 time_point。system_clock 定义了两个静态辅助函数, 用于 time_point 和 C 风格的时间表示方法 time_t 之间的相互转换。第一个辅助函数为 to_time_t(), 它将给定的 time_point 转换为 time_t; 第二个为 from_time_t(), 它执行相反的转换。time_t 类型在<ctime>头文件中定义。

下例展示了一个完整程序, 它从系统获得当前时间, 然后将这个时间以可供用户读取的格式输出到控制台。localtime()函数将 time_t 转换为用 tm 表示的本地时间, 定义在<ctime>头文件中。C++的 put_time()流操作算子定义在<iomanip>中, 详见第 13 章“揭秘 C++ I/O”的讨论。

```
// Get current time as a time_point
system_clock::time_point tpoint { system_clock::now() };
// Convert to a time_t
time_t tt { system_clock::to_time_t(tpoint) };
// Convert to local time
tm* t { localtime(&tt) };
// Write the time to the console
cout << put_time(t, "%H:%M:%S") << endl;
```

如果想要将时间转换为字符串, put_time 可与 std::stringstream 或 C 风格的 strftime()函数一起使用, 这些定义在<ctime>中。使用 strftime()函数时, 要求提供一个足够大的缓冲区, 以容纳用户可读格式的给定时间。

```
// Convert to readable format
stringstream ss;
Ss << put_time(t, "%H:%M:%S"); // t is a tm*
String stringTime { ss.str() };
cout << stringTime << endl;
// Or:
char buffer[80] {0};
strftime(buffer, sizeof(buffer), "%H:%M:%S", t); // t is a tm*
cout << buffer << endl;
```

注意:

以上例子可能对 localtime()的调用给出与安全相关的错误或警告。在 Microsoft Visual C++中可使用安全版本的 localtime_s()。在 Linux 上, 可使用 localtime_r()函数。

通过 high_resolution_clock 还可计算一段代码执行所消耗的时间。下面的代码片段给出了示例。变量 start 和 end 的实际类型为 system_clock::time_point, diff 的实际类型为 duration:

```
// Get the start time
auto start { high_resolution_clock::now() };
// Execute code that you want to time
double d { 0 };
for (int i { 0 }; i < 1'000'000; ++i) {
    d += sqrt(sin(i) * cos(i));
}
// Get the end time and calculate the difference
auto end { high_resolution_clock::now() };
auto diff { end - start };
// Convert the difference into milliseconds and output to the console
cout << duration<double, milli> { diff }.count() << "ms" << endl;
```

这个例子中的循环执行一些算术操作, 例如 sqrt()、sin()和 cos(), 确保循环不会太快结束。如果

在系统上获得非常小的毫秒差值，那么这些值不会很准确，应该增加循环的迭代次数，使循环执行时间更长。小的计时值不会很准确，因为尽管定时器的精度为毫秒级，但在大多数操作系统上，这个定时器的更新频率不高，例如每 10 毫秒或 15 毫秒更新一次。这会导致一种称为门限错误(gating error)的现象，也就是说，任何持续时间小于 1 个定时器滴答的事件只花了 0 个时间单位，任何持续时间在 1 个和 2 个定时器滴答之间的事件花了 1 个时间单位。例如，在一个定时器更新频率为 15 毫秒的系统上，运行了 44 毫秒的循环看上去只花了 30 毫秒。当使用这类定时器确定计算所用的时间时，一定要确保整个计算消耗了大量的基本定时器滴答单位，这样误差才能最小化。

22.4 时间点

`time_point` 类表示的是时间中的某个时点，存储为相对于纪元(epoch)的 `duration`，表示开始时间。`time_point` 总是和特定的 `clock` 关联，纪元就是所关联 `clock` 的原点。例如，经典 UNIX/Linux 的时间纪元是 1970 年 1 月 1 日，`duration` 用秒来度量。Windows 的纪元是 1601 年 1 月 1 日，`duration` 用 100 纳秒作为单位来度量。其他操作系统还有不同的纪元日期和 `duration` 单位。

`time_point` 类包含 `time_since_epoch()` 函数，它返回的 `duration` 表示所关联 `clock` 的纪元和保存的时间点之间的时间。

C++ 支持合理的 `time_point` 和 `duration` 算术运算。下面列出这些运算。`tp` 代表 `time_point`，`d` 代表 `duration`。

<code>tp + d = tp</code>	<code>tp - d = tp</code>
<code>d + tp = tp</code>	<code>tp - tp = d</code>
<code>tp += d</code>	<code>tp -= d</code>

C++ 不支持的操作示例是 `tp+tp`。

C++ 支持使用比较运算符 `==` 和 `<=>` 来比较两个时间点，提供了两个静态方法：`min()` 和 `max()`，分别返回最小的时间点和最大的时间点。

`time_point` 类有 3 个构造函数。

- **`time_point()`**: 构造一个 `time_point`，通过 `duration::zero()` 进行初始化。得到的 `time_point` 表示所关联 `clock` 的纪元。
- **`time_point(const duration& d)`**: 构造一个 `time_point`，通过给定的 `duration` 进行初始化。得到的 `time_point` 表示纪元+d。
- **`template <class Duration2> time_point(const time_point<clock, Duration2>& t)`**: 构造一个 `time_point`，通过 `t.time_since_epoch()` 进行初始化。

每个 `time_point` 都关联一个 `clock`。创建 `time_point` 时，指定 `clock` 作为模板参数：

```
time_point<steady_clock> tp1;
```

每个 `clock` 都知道各自的 `time_point` 类型，因此可编写以下代码：

```
steady_clock::time_point tp1;
```

下面的示例演示了 `time_point` 类：

```
// Create a time_point representing the epoch
// of the associated steady clock
time_point<steady_clock> tp1;
```

```
// Add 10 minutes to the time_point
tp1 += minutes { 10 };
// Store the duration between epoch and time_point
auto d1 { tp1.time_since_epoch() };
// Convert the duration to seconds and output to the console
duration<double> d2 { d1 };
cout << d2.count() << " seconds" << endl;
```

输出如下所示:

```
600 seconds
```

可通过隐式方法或显式方法转换 `time_point`, 这与 `duration` 转换类似。下面是一个隐式转换示例, 输出是 42000 ms。

```
time_point<steady_clock, seconds> tpSeconds { 42s };
// Convert seconds to milliseconds implicitly.
time_point<steady_clock, milliseconds> tpMilliseconds { tpSeconds };
cout << tpMilliseconds.time_since_epoch().count() << " ms" << endl;
```

如果隐式转换导致丢失数据, 则需要使用 `time_point_cast()` 进行显式转换, 就像需要使用 `duration_cast()` 进行显式 `duration` 转换。虽然开始时是 42424 ms, 但本例输出 42000 ms。

```
time_point<steady_clock, milliseconds> tpMilliseconds { 42'424ms };
// Convert milliseconds to seconds explicitly.
time_point<steady_clock, seconds> tpSeconds {
    time_point_cast<seconds>(tpMilliseconds) };
// Or:
// auto tpSeconds { time_point_cast<seconds>(tpMilliseconds) };

// Convert seconds back to milliseconds and output the result.
milliseconds ms { tpSeconds.time_since_epoch() };
cout << ms.count() << " ms" << endl;
```

库为 `time_point` 增加了 `floor()`、`ceil()` 和 `round()` 操作, 类似于处理数值数据。

C++20

22.5 日期

C++20 在标准库中增加了对日历日期的支持。目前只支持 `Gregorian` 日历, 但如果需要, 可以经常实现自己的日历, 与 `<chrono>` 其余的功能交互, 例如 `Coptic` 和 `Julian` 日历。

标准库提供了相当多的类和函数来处理日期(以及下一节要讨论的时区)。本文讨论了最重要的类和函数, 可参考标准库以获得所有可用内容的完整概述。

引入表 22-2 所示的类来处理年、月、日和工作日, 所有这些都在 `std::chrono` 中定义。

表 22-2 类

类	说明
year	表示年份, 范围为[-32767, 32767]。year 有个名为 <code>is_leap()</code> 的方法, 如果给定的年份是闰年, 则返回 <code>true</code> , 否则返回 <code>false</code> 。 <code>min()</code> 和 <code>max()</code> 静态方法分别返回最小和最大年份
month	表示月份, 范围为[1, 12]。此外还提供了 12 个用于 12 个月的命名常量, 例如 <code>std::chrono::January</code>
day	表示天, 范围为[1, 31]

(续表)

类	说明
weekday	表示一周中的一天，范围为[0, 6]，其中 0 表示周日。此外还有 6 个用于这 6 个工作日的命名常量，例如 <code>std::chrono::Sunday</code>
weekday_indexed	表示一个月的第一个、第二个、第三个、第四个和第五个工作日。可以很容易地从工作日构造，例如： <code>Monday[2]</code> 表示一个月的第二个星期一
weekday_last	表示某个月的最后一个工作日
month_day	表示特定的月和日
month_day_last	表示特定月份的最后一天
month_weekday	表示特定月份的第 <i>n</i> 个工作日
month_weekday_last	表示特定月份的最后个工作日
year_month	表示特定的年和月
year_month_day	表示特定的年月日
year_month_day_last	表示特定年份和月份的最后一天
year_month_weekday	表示特定年份和月份的第 <i>n</i> 个工作日
year_month_weekday_last	表示特定年份和月份的最后个工作日

所有这些类都有一个名为 `ok()` 的方法，如果给定的对象在有效范围内，该方法返回 `true`，否则返回 `false`。另外提供了两个新的标准用户定义字面量，`std::literals::chrono_literals::y` 用于创建年，`d` 用于创建天。可以使用运算符/以 3 个顺序 `Y/M/D`、`M/D/Y`、`D/M/Y` 指定年月日的完整日期。下面是一些创建日期的示例：

```
year y1 { 2020 };
auto y2 { 2020y };

month m1 { 6 };
auto m2 { June };

day d1 { 22 };
auto d2 { 22d };

// Create a date for 2020-06-22.
year_month_day fulldate1 { 2020y, June, 22d };
auto fulldate2 { 2020y / June / 22d };
auto fulldate3 { 22d / June / 2020y };

// Create a date for the 3rd Monday of June 2020.
year_month_day fulldate4 { Monday[3] / June / 2020 };

// Create a month_day for June 22 of an unspecified year.
auto june22 { June / 22d };
// Create a year_month_day for June 22, 2020.
auto june22_2020 { 2020y / june22 };

// Create a month_day_last for the last day of a June of an unspecified year.
auto lastDayOfAJune { June / last };
// Create a year_month_day_last for the last day of June for the year 2020.
auto lastDayOfJune2020 { 2020y / lastDayOfAJune };
```

```
// Create a year_month_weekday_last for the last Monday of June 2020.
auto lastMondayOfJune2020 { 2020y / June / Monday[last] };
```

`sys_time` 是 `system_clock` 中一个具有一定持续时间的 `time_point` 的新类型别名，它的定义如下：

```
template <typename Duration>
using sys_time = std::chrono::time_point<std::chrono::system_clock, Duration>;
```

基于 `sys_time` 类型别名，定义了两个额外的类型别名，来表示精度为秒的 `sys_time` 和精度为天的 `sys_time`：

```
using sys_seconds = sys_time<std::chrono::seconds>;
using sys_days = sys_time<std::chrono::days>;
```

例如，`sys_days` 表示自 `system_clock` 纪元以来的天数，因此，它是一个基于串行的类型，也就是说，它只包含一个数字(从纪元开始的天数)。另一方面，`year_month_day` 是基于字段的类型，它将年月日存储在单独的字段中。当对日期进行大量算术运算时，基于串行的类型比基于字段的类型具有更好的性能。

类似的类型别名用于处理本地时间：`local_time`、`local_seconds` 和 `local_days`。下一节将讨论时区。如下所示，已创建表示今天的 `sys_days`。注意 `floor()` 用于将 `time_point` 截短为精确的天数：

```
auto today = { floor<days>(system_clock::now()) };
```

`sys_days` 可用于将 `year_month_day` 转换为 `time_point`，例如：

```
system_clock::time_point t1 { sys_days { 2020y / June / 22d } };
```

反向转换，即将 `time_point` 转换为 `year_month_day`，可以使用 `year_month_day` 构造函数来完成。下面的代码片段给出了两个示例：

```
year_month_day yearmonthday { floor<days>(t1) };
year_month_day today2 { floor<days>(system_clock::now()) };
```

也可以构建一个完整的日期。这里有个示例：

```
// Full date with time: 2020-06-22 09:35:10 UTC.
auto t2 { sys_days { 2020y / June / 22d } + 9h + 35min + 10s };
```

可以使用日期进行算术运算。这里有个示例：

```
// Add 5 days to t2.
auto t3 { t2 + days { 5 } };
// Add 1 year to t3.
auto t4 { t3 + years { 1 } };
```

可以使用熟悉的插入运算符将日期写入流中：

```
cout << yearmonthday << endl;
```

如果日期无效，将其写入流也会发生错误。例如字符串 "is not a valid date" 被追加到无效 `year_month_day` 后面写入流中。

22.6 时区

为了方便使用时区，C++标准库包含了一份 IANA(Internet Assigned Numbers Authority)时区数据库

(www.iana.org/time-zones)的副本。可以通过调用 `std::chrono::get_tzdb()` 来访问数据库，该方法返回对单个已有 `std::chrono::tzdb` 类型实例的 `const` 引用。该数据库通过名为 `zones` 的公有 `vector` 提供对所有已知时区的访问。这个 `vector` 中的每个条目都是一个 `time_zone`，它有名字，可以通过 `name()` 访问，以及 `to_sys()` 和 `to_local()` 方法将本地时间转换为 `sys_time`，反之亦然。

举例，下面的片段列出了所有可用时区：

```
const auto& database { get_tzdb() };
for (const auto& timezone : database.zones) {
    cout << timezone.name() << endl;
}
```

`std::chrono::locate_zone()` 函数可以用于根据名称检索 `time_zone`，如果在数据库中找不到请求的时区，则抛出 `runtime_error` 异常。`current_zone()` 函数可用于获取当前时区。这里有个示例：

```
auto* brussels { locate_zone("Europe/Brussels") };
auto* gmt { locate_zone("GMT") };
auto* current { current_zone() };
```

`time_zone` 实例可用于在不同时区间转换时间，这里有一些示例：

```
// Convert the current system time to GMT.
gmt->to_local(system_clock::now());

// Construct a UTC time. (2020-06-22 09:35:10 UTC)
auto t { sys_days { 2020y / June / 22d } + 9h + 35min + 10s };
// Convert UTC time to Brussels' local time.
brussels->to_local(t);
```

`zoned_time` 类用于表示 `time_zone` 中的 `time_point`。例如，下面的代码片段在 `Brussels` 时区中构造了一个特定的时间，并将其转换为纽约时间：

```
// Construct a local time in the Brussels' time zone.
zoned_time<hours> brusselsTime{ brussels, local_days { 2020y / June / 22d } + 9h };
// Convert to New York time.
zoned_time<hours> newYorkTime { "America/New_York", brusselsTime };
```

22.7 本章小结

本章讨论了如何使用 `ratio` 模板来定义和处理编译期有理数。还学习了如何使用 C++ 标准库通过 `chrono` 提供的持续时间、时钟、时间点、日期和时区。

22.8 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题中，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

练习 22-1 创建一个持续时间 `d1`，精度为秒，初始化为 42 秒。创建第二个持续时间 `d2`，初始化为 1.5 分钟。计算 `d1` 和 `d2` 的和。结果写到标准输出，一次以秒表示，一次以分钟表示。

练习 22-2 让用户输入日期 `yyyy-mm-dd`，例如 2020-06-22。使用正则表达式(见第 21 章“字符串本地化和正则表达式”)提取年月日组件，最后使用 `year_month_day` 验证日期。

练习 22-3 编写一个 `getNumberOfDaysBetweenDates()` 函数，计算两个给定日期之间的天数。在 `main()` 函数中测试实现。

练习 22-4 编写一个打印出 2020 年 6 月 22 日星期几的程序。

练习 22-5 构造 UTC 时间。将此时间转换为日本东京的当地时间。进一步将得到的时间转换为纽约时间。最后将计算出的时间换算成 GMT 时间。验证原始 UTC 时间和最终 GMT 时间是否相等。提示：东京的时区标识符是 `Asia/Tokyo`，纽约的时区标识符是 `America/New_york`，GMT 的时区标识符是 `GMT`。

练习 22-6 编写一个 `getDurationSinceMidnight()` 函数，返回从半夜到当前时间之间的持续时间。在 `<ctime>` 中定义的 `mktime()` 函数在实现中可能很有用。它接收 `tm*` 作为参数并返回 `time_t`。它基本上与 `localtime()` 所做的相反。使用函数将半夜以来的秒数打印到标准输出控制台中。最后，使用 `hh_mm_ss` 类将函数返回的持续时间转换为小时、分钟和秒，并将结果打印到标准输出中。

第23章

随机数工具

本章内容

- 随机数引擎和引擎适配器的概念
- 如何生成随机数
- 如何改变随机数的分布

本章讨论如何在 C++ 中生成随机数。在软件中生成好的随机数是个复杂的话题。本章不讨论生成实际随机数所涉及的复杂数学公式；然而它解释了如何使用标准库提供的功能生成随机数。

C++ 随机数生成库可以使用不同的算法和分布生成随机数。这个库由 `std` 名称空间中的 `<random>` 定义。它有三大组件：引擎、引擎适配器和分布。随机数引擎负责生成实际的随机数，并存储生成后随机数的状态。分布决定了生成的随机数范围以及它们在该范围内的数学分布方式。随机数引擎适配器修改与之关联的随机数引擎的结果。

在深入研究这个 C++ 随机数生成库之前，先简要解释旧的 C 风格随机数生成机制及其问题。

23.1 C 风格随机数生成器

在 C++11 之前，生成随机数的唯一方法是使用 C 风格的 `srand()` 和 `rand()` 函数。`srand()` 函数需要在应用程序中调用一次，这个函数初始化随机数生成器，也称为设置种子 (seeding)。通常应该使用当前系统时间作为种子。

警告：

在基于软件的随机数生成器中，一定要使用高质量的种子。如果每次都使用同一个种子初始化随机数生成器，那么每次生成的随机数序列都是一样的。这也是通常要采用当前系统时间作为种子的原因。

初始化随机数生成器后，通过 `rand()` 生成随机数。下例展示了如何使用 `srand()` 和 `rand()`。定义在 `<ctime>` 中的 `time()` 函数，返回系统时间，通常编码为系统纪元后的秒数。纪元表示起始时间。

```
srand(static_cast<unsigned int>(time(nullptr)));  
cout << rand() << endl;
```


可通过以下函数生成特定范围内的随机数:

```
int getRandom(int min, int max)
{
    return static_cast<int>(rand() % (max + 1UL - min)) + min;
}
```

旧式的 C 风格 `rand()` 函数生成的随机数在 0 到 `RAND_MAX` 之间, 根据标准, `RAND_MAX` 至少应该为 32 767。不能生成大于 `RAND_MAX` 的随机数。在某些系统上, 例如 GCC, `RAND_MAX` 是 2 147 483 647, 它等于一个有符号整数的最大值。这就是 `getRandom()` 中的公式将 1UL(无符号长整型值)添加到 `max` 以防止算术溢出的原因。

此外, `rand()` 的低位通常不是非常随机, 也就是说, 通过上面的 `getRandom()` 生成的随机数范围较小(例如 1 和 6 之间), 并且得到的随机性并不是非常好。

注意:

基于软件的随机数生成器永远都不可能生成真正的随机数, 而是根据数学公式生成随机的效果, 因此称为伪随机数生成器。

除了生成的随机数质量差之外, 旧的 `srand()` 和 `rand()` 函数并没有提供太多的灵活性。例如不能改变生成的随机数的分布。总之, 强烈建议不再使用 `srand()` 和 `rand()`, 而使用 `<random>` 中的类, 接下来的章节会解释。

23.1.1 随机数引擎

C++ 随机数生成库的第一个组件是随机数引擎, 它负责生成实际的随机数。如上所述, 所有内容都在 `<random>` 中定义。

以下随机数引擎可供使用:

- `random_device`
- `linear_congruential_engine`
- `mersenne_twister_engine`
- `subtract_with_carry_engine`

`random_device` 引擎不是基于软件的随机数生成器; 这是一种特殊引擎, 要求计算机连接能真正生成不确定随机数的硬件, 例如通过物理原理生成。一种经典的机制是通过计算每个时间间隔内的 α 粒子数或类似的方法来测量放射性同位素的衰变, 但还有很多其他类型的基于物理原理的生成器, 包括测量反向偏压二极管“噪声”的方法(不必担心计算机内有放射源)。由于篇幅受限, 本书不讲解这些机制的细节。

根据 `random_device` 的规范, 如果计算机没有连接此类硬件设备, 这个库可选用一种软件算法。算法的选择取决于库的设计者。

随机数生成器的质量由随机数的熵(entropy)决定。如果 `random_device` 类使用的是基于软件的伪随机数生成器, 那么这个类的 `entropy()` 方法返回的值为 0.0; 如果使用硬件设备, 则返回非零值。这个非零值是对硬件设备的熵的估计。

`random_device` 引擎的使用非常简单:

```
random_device rnd;
cout << "Entropy: " << rnd.entropy() << endl;
cout << "Min value: " << rnd.min()
    << ", Max value: " << rnd.max() << endl;
```

```
cout << "Random number: " << rnd() << endl;
```

这个程序的输出如下所示：

```
Entropy: 32
Min value: 0, Max value: 4294967295
Random number: 3590924439
```

`random_device` 的速度通常比伪随机数引擎更慢。因此，如果需要生成大量的随机数，建议使用伪随机数引擎，使用 `random_device` 为随机数引擎生成种子。后面的生成随机数小节会具体阐述。

除 `random_device` 随机数引擎之外，还有 3 个伪随机数引擎，如下所示。

- **线性同余引擎(linear congruential engine)**: 保存状态所需的内存量最少。状态是一个包含上一次生成的随机数的整数，如果尚未生成随机数，则保存的是初始种子。这个引擎的周期取决于算法的参数，最高可达 2^{64} ，但是通常不会这么高。因此，如果需要使用高质量的随机数序列，那么不应该使用线性同余引擎。
- **梅森旋转(Mersenne twister)**: 在这 3 个伪随机数引擎中，梅森旋转算法生成的随机数质量最高。梅森旋转算法的周期是所谓的梅森素数，它是一个比 2 的幂小于 1 的素数，这个周期比线性同余引擎的周期要长得多。梅森旋转算法保存状态所需的内存量也取决于算法参数，但是比线性同余引擎的整数状态高得多。例如，预定义的梅森旋转算法 `mt19937` 的周期为 $2^{19937}-1$ ，状态包含 625 个整数，约为 2.5KB。它是最快的随机数引擎之一。
- **带进位减法引擎(subtract with carry engine)**: 要求保存大约 100 字节的状态。不过，这个随机数引擎生成的随机数质量不如梅森旋转算法。

这些随机数引擎的数学原理超出了本书的讨论范围，而且对随机数质量的定义需要一定的数学背景。如果要更深入地了解这个主题的内容，可以参阅附录 B 的“随机数”部分列出的参考文献。

`random_device` 随机数引擎易于使用，不需要任何参数。然而，创建这 3 个伪随机数生成器的实例时需要指定一些数学参数，这些参数可能会很复杂。参数的选择会极大地影响生成的随机数的质量。例如，`mersenne_twister_engine` 类的定义如下所示：

```
template<class UIntType, size_t w, size_t n, size_t m, size_t r,
        UIntType a, size_t u, UIntType d, size_t s,
        UIntType b, size_t t, UIntType c, size_t l, UIntType f>
class mersenne_twister_engine {...}
```

这个定义需要 14 个参数。`linear_congruential_engine` 类和 `subtract_with_carry_engine` 类也需要一些这样的数学参数。因此，C++ 标准定义了一些预定义的随机数引擎，比如 `mt19937` 梅森旋转算法，定义如下：

```
using mt19937 = mersenne_twister_engine<uint_fast32_t, 32, 624, 397, 31,
    0x9908b0df, 11, 0xffffffff, 7, 0x9d2c5680, 15, 0xefc60000, 18,
    1812433253>;
```

除非理解梅森旋转算法的细节，否则会觉得这些参数都是魔幻数。一般情况下不需要修改任何参数，除非你是伪随机数生成器方面的数学专家。强烈建议使用这些预定义的类型别名，例如 `mt19937`。23.1.2 节将完整列出所有预定义的随机数引擎。

23.1.2 随机数引擎适配器

随机数引擎适配器修改相关联的随机数引擎生成的结果，关联的随机数引擎称为基引擎(base engine)。这是适配器模式(详见第 33 章)的一个实例。C++ 定义了以下 3 个适配器模板：

```
template<class Engine, size_t p, size_t r> class discard_block_engine {...}
template<class Engine, size_t w, class UIntType> class independent_bits_engine {...}
template<class Engine, size_t k> class shuffle_order_engine {...}
```

`discard_block_engine` 适配器丢弃基引擎生成的一些值，以生成随机数。这个适配器模板需要 3 个参数：要适配的引擎、块大小 `p` 以及使用的块大小 `r`。基引擎用于生成 `p` 个随机数。适配器丢弃其中 `p-r` 个数，返回剩下的 `r` 个数。

`independent_bits_engine` 适配器组合由基引擎生成的一些随机数，以生成具有给定位数 `w` 的随机数。

`shuffle_order_engine` 适配器生成的随机数和基引擎生成的随机数一致，但顺序不同。模板参数 `k` 是适配器使用的内部表格的大小。根据请求从表中随机选择一个随机数，然后用基引擎生成的一个新随机数替换。

这些适配器的具体工作原理与数学知识相关，超出了本书的讨论范围。和随机数引擎一样，C++ 标准中包含一些预定义的随机数引擎适配器。23.1.3 节列出了预定义的随机数引擎和引擎适配器。

23.1.3 预定义的随机数引擎和引擎适配器

如前所述，建议不要自行指定伪随机数引擎和引擎适配器的参数，而是使用那些标准的随机数引擎。C++ 定义了表 23-1 所示的预定义生成器，它们都定义在 `<random>` 中。它们都有复杂的模板参数；不过，即使不理解这些参数，也可以使用它们。

表 23-1 预定义生成器

预定义生成器	类模板
<code>minstd_rand0</code>	<code>linear_congruential_engine</code>
<code>minstd_rand</code>	<code>linear_congruential_engine</code>
<code>mt19937</code>	<code>mersenne_twister_engine</code>
<code>mt19937_64</code>	<code>mersenne_twister_engine</code>
<code>ranlux24_base</code>	<code>linear_congruential_engine</code>
<code>ranlux48_base</code>	<code>subtract_with_carry_engine</code>
<code>ranlux24</code>	<code>discard_block_engine</code>
<code>ranlux48</code>	<code>discard_block_engine</code>
<code>knuth_b</code>	<code>shuffle_order_engine</code>
<code>default_random_engine</code>	由实现定义

`default_random_engine` 与编译器相关。
23.1.4 节将列举一个使用这些预定义引擎的例子。

23.1.4 生成随机数

在生成随机数前，首先要创建一个引擎实例。如果使用一个基于软件的引擎，那么还需要定义分布。分布是一个描述数字在特定范围内分布方式的数学公式。创建引擎的推荐方式是使用前面讨论的预定义引擎。

下面的例子使用了名为 `mt19937` 的预定义的梅森旋转算法。这是一个基于软件的生成器。与旧式的 `rand()` 生成器一样，基于软件的引擎也需要使用种子进行初始化。`srand()` 的种子常常是当前时间。在

现代 C++ 中，建议不使用任何基于时间的种子(`random_device` 不拥有熵)，而使用 `random_device` 生成种子：

```
random_device seeder;
const auto seed { seeder.entropy() ? seeder() : time(nullptr) };
mt19937 engine { static_cast<mt19937::result_type>(seed) };
```

接下来定义了分布。这个例子使用均匀整数分布，在范围 1~99 内分布。在这个例子中，均匀分布很容易使用：

```
uniform_int_distribution<int> distribution { 1, 99 };
```

定义了引擎和分布后，通过调用分布的函数调用运算符，并将引擎作为参数传入，就可以生成随机数了。在这个例子中写为 `distribution(engine)`：

```
cout << distribution(engine) << endl;
```

从这个例子可以看出，为通过基于软件的引擎生成随机数，总是需要指定引擎和分布。使用第 19 章介绍的定义在 `<functional>` 中的 `std::bind()` 实用工具，可以避免在生成随机数时指定分布和引擎。下面的例子和前面的例子一样，使用 `mt19937` 引擎和均匀分布，但是这个例子通过 `std::bind()` 将 `engine` 绑定至 `distribution()` 的第一个参数，从而定义了 `generator`。通过这种方式，调用 `generator()` 生成新的随机数时不需要提供任何参数。然后这个例子演示了如何结合使用 `generator()` 和 `generate()` 算法，为一个 10 个元素的 `vector` 填充随机数。第 20 章讨论了 `generate()` 算法，这个算法在 `<algorithm>` 中定义。

```
auto generator { bind(distribution, engine) };

vector<int> values(10);
generate(begin(values), end(values), generator);

for (auto i : values) { cout << i << " "; }
```

注意：

记住 `generate()` 算法改写现有的元素，不会插入新的元素。这意味着首先需要将 `vector` 设置为足够大，以保存所需数目的元素，然后调用 `generate()` 算法。前一个例子将 `vector` 的大小指定为构造函数的参数。

尽管不知道 `generator` 的具体类型是什么，但仍可将 `generator` 传入另一个需要使用生成器的函数。你有些选择：使用 `std::function<int>` 类型的参数，或者使用函数模板。上一个例子可改为在 `fillVector()` 函数中生成随机数。下面是使用 `std::function` 的实现：

```
void fillVector(vector<int>& values, const std::function<int()>& generator)
{
    generate(begin(values), end(values), generator);
}
```

下面是函数模板版本：

```
template<typename T>
void fillVector(vector<int>& values, const T& generator)
{
    generate(begin(values), end(values), generator);
}
```

可以使用 C++20 中的简化函数模板语法进行简化:

```
void fillVector<vector<int>& values, const auto& generator>
{
    generate(begin(values), end(values), generator);
}
```

最后按如下方式使用该函数:

```
vector<int> values(10);
fillVector(values, generator);
```

23.1.5 随机数分布

分布是一个描述数字在特定范围内分布的数学公式。随机数生成器库带有的分布可与伪随机数引擎结合使用,从而定义生成的随机数的分布。这是一种压缩的表示形式。每个分布的第一行是类的名称和类模板参数(如果有的话)。接下来的行是这个分布的构造函数。每个分布只列出了一个构造函数,以帮助读者理解这个类。标准库参考资源(见附录 B)列出了每个分布的所有构造函数和方法。

这些是可用的均匀分布:

```
template<class IntType = int> class uniform_int_distribution
    uniform_int_distribution(IntType a = 0,
                           IntType b = numeric_limits<IntType>::max());
template<class RealType = double> class uniform_real_distribution
    uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
```

这些是可用的伯努利分布(根据离散概率分布生成随机布尔值):

```
class bernoulli_distribution
    bernoulli_distribution(double p = 0.5);
template<class IntType = int> class binomial_distribution
    binomial_distribution(IntType t = 1, double p = 0.5);
template<class IntType = int> class geometric_distribution
    geometric_distribution(double p = 0.5);
template<class IntType = int> class negative_binomial_distribution
    negative_binomial_distribution(IntType k = 1, double p = 0.5);
```

这些是可用的泊松分布(根据离散概率分布生成随机非负整数):

```
template<class IntType = int> class poisson_distribution
    poisson_distribution(double mean = 1.0);
template<class RealType = double> class exponential_distribution
    exponential_distribution(RealType lambda = 1.0);
template<class RealType = double> class gamma_distribution
    gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
template<class RealType = double> class weibull_distribution
    weibull_distribution(RealType a = 1.0, RealType b = 1.0);
template<class RealType = double> class extreme_value_distribution
    extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);
```

这些是可用的正态分布:

```
template<class RealType = double> class normal_distribution
    normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
template<class RealType = double> class lognormal_distribution
    lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
template<class RealType = double> class chi_squared_distribution
    chi_squared_distribution(RealType n = 1);
```

```
template<class RealType = double> class cauchy_distribution
    cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
template<class RealType = double> class fisher_f_distribution
    fisher_f_distribution(RealType m = 1, RealType n = 1);
template<class RealType = double> class student_t_distribution
    student_t_distribution(RealType n = 1);
```

这些是可用的采样分布：

```
template<class IntType = int> class discrete_distribution
    discrete_distribution(initializer_list<double> wl);
template<class RealType = double> class piecewise_constant_distribution
    template<class UnaryOperation>
        piecewise_constant_distribution(initializer_list<RealType> bl,
            UnaryOperation fw);
template<class RealType = double> class piecewise_linear_distribution
    template<class UnaryOperation>
        piecewise_linear_distribution(initializer_list<RealType> bl,
            UnaryOperation fw);
```

每个分布都需要一组参数。对这些数学参数的详细解释超出了本书的范围，本节剩余部分将列举一些例子来解释分布对生成的随机数造成的影响。

观察图形是理解分布的最简便方法。例如，下面的代码生成 100 万个介于 1 和 99 之间的随机数，然后跟踪在直方图中随机生成某个数字的次数。将结果保存在一个 map 中，这个 map 的键是介于 1 到 99 之间的数，与键关联的值是这个键被随机选择的次数。在循环之后，将结果写入一个逗号分隔值(CSV)文件，然后可在电子表格应用程序中打开这个文件。

```
const unsigned int Start { 1 };
const unsigned int End { 99 };
const unsigned int Iterations { 1'000'000 };

// Uniform Mersenne Twister
random_device seeder;
const auto seed { seeder.entropy() ? seeder() : time(nullptr) };
mt19937 engine { static_cast<mt19937::result_type>(seed) };
uniform_int_distribution<int> distribution { Start, End };
auto generator { bind(distribution, engine) };
map<int, int> histogram;
for (unsigned int i { 0 }; i < Iterations; ++i) {
    int randomNumber { generator() };
    // Search map for a key = randomNumber. If found, add 1 to the value associated
    // with that key. If not found, add the key to the map with value 1.
    ++(histogram[randomNumber]);
}

// Write to a CSV file
ofstream of { "res.csv" };
for (unsigned int i { Start }; i <= End; ++i) {
    of << i << "," << histogram[i] << endl;
}
```

结果数据可用于生成图形表示。生成的直方图如图 23-1 所示。

横轴表示所生成随机数的范围。通过图 23-1 可清晰地看出，1 到 99 之间的数字大约被随机选出 10 000 次，因此生成的这些随机数均匀地分布在整个范围内。

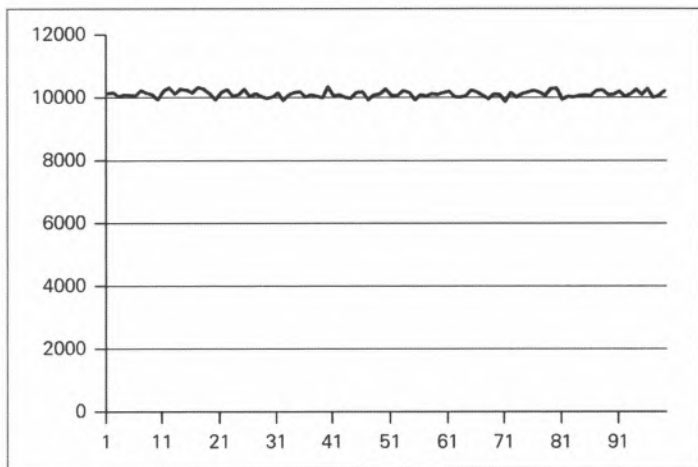


图 23-1 生成的直方图

这个例子可改为根据正态分布(而不是均匀分布)生成随机数。只需要在两个地方做微小修改。首先,将分布的创建修改为:

```
normal_distribution<double> distribution { 50, 10 };
```

由于正态分布使用的是 `double` 值而不是整数,因此需要修改 `generator()` 的调用。

```
int randomNumber { static_cast<int>(generator()) };
```

图 23-2 展示了根据正态分布生成的随机数的图形表示。

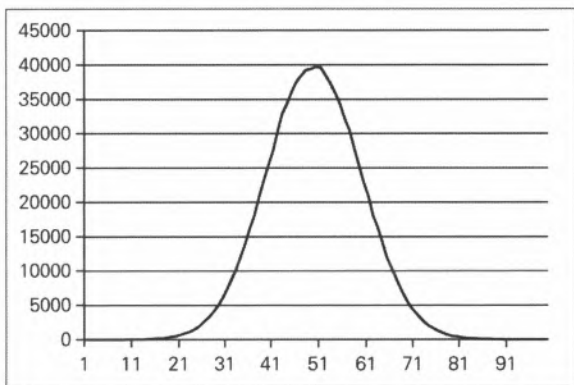


图 23-2 随机数的图形表示

图 23-2 清晰地指明生成的大部分随机数都位于范围的中间。在这个例子中,值 50 被随机选择出约 40 000 次,而值 20 和 80 则仅被随机选择出约 500 次。

23.2 本章小结

在本章中,学习了如何使用标准库提供的 C++随机数生成库来生成高质量的随机数。也看到了如何在给定范围内操作生成的数字的分布。

本书第Ⅲ部分的第 24 章介绍了一些值得讨论的较小的标准库特性。

23.3 练习

通过完成下面的习题,可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题,可以在看网站上的答案之前,先重新阅读本章的部分内容,尝试着自己找到答案。

练习 23-1 写一个循环询问是否应该扔骰子。如果是,使用均匀分布掷两次骰子,将这两个数字打印到控制台。如果否,停止程序。使用标准 `mt19337` 梅森旋转引擎。

练习 23-2 修改习题 23-1 的答案,使用 `ranlux48` 引擎代替梅森旋转。

练习 23-3 修改习题 23-1 的答案,替换 `mt19937` 梅森旋转引擎,使用 `shuffle_order_engine` 适配器调整。

练习 23-4 修改习题 23-1 的答案,替换使用均匀分布,使用负二项分布(`negative_binomial_distribution`)。

练习 23-5 使用本章前面的源代码生成直方图来绘制分布图,并对不同的分布进行试验。尝试在电子表格应用程序中绘制图表来查看分布的效果。代码可以在 `c23_code\01_Random` 目录中的源码存档中找到。可以使用 `06_uniform_int_distribution.cpp` 或者 `07_normal_distribution.cpp` 文件,这都取决于分布使用的是整数(int)还是双精度整数(double)。

第24章

其他库工具

本章内容

- 如何使用 `variant` 和 `any` 数据类型
- 元组的含义及用法

本章讨论 C++ 标准库提供的一些值得一看的附加库功能。它以另外两种词汇表类型 `variant` 和 `any` 开始讨论，它们补充了第 1 章中讨论的 `optional` 词汇表类型。本章最后讨论了元组(tuple)，它是 `pair` 的泛化。

24.1 `variant`

`std::variant` 在 `<variant>` 中定义，可用于保存给定类型集合的一个值。定义 `variant` 时，必须指定它可能包含的类型。例如，以下代码定义 `variant` 一次可以包含整数、字符串或浮点值：

```
variant<int, string, float> v;
```

`variant` 的模板类型参数必须是唯一的；例如，`variant<int, int>` 无效。这里，默认构造的 `variant` 包含第一个类型(此处是 `int`)的默认构造值。要默认构造 `variant`，务必确保 `variant` 的第一个类型是默认可构造的。例如，下面的代码无法编译，因为 `Foo` 不是默认可构造的。

```
class Foo { public: Foo() = delete; Foo(int) {} };
class Bar { public: Bar() = delete; Bar(int) {} };
...
variant<Foo, Bar> v;
```

事实上，`Foo` 和 `Bar` 都不是默认可构造的。如果仍需要默认构造 `variant`，可使用 `std::monostate`(一个表现好的空替代)作为 `variant` 的第一个类型。

```
variant<monostate, Foo, Bar> v;
```

可使用赋值运算符，在 `variant` 中存储内容：

```
variant<int, string, float> v;
v = 12;
v = 12.5f;
v = "An std::string"s;
```

`variant` 在任何给定时间只能包含一个值。因此，对于这三个赋值语句，首先将整数 12 存储在 `variant` 中，然后将 `variant` 改为包含浮点值，最后将 `variant` 改为包含字符串。

可使用 `index()` 方法来获取当前存储在 `variant` 中的值类型的索引(以 0 开始)。可以使用 `std::holds_alternative()` 函数模板来确定 `variant` 当前是否包含特定类型的值：

```
cout << "Type index: " << v.index() << endl;
cout << "Contains an int: " << holds_alternative<int>(v) << endl;
```

输出如下所示：

```
Type index: 1
Contains an int: 0
```

使用 `std::get<index>()` 或 `get<T>()` 从 `variant` 检索值，其中 `index` 是想要检索类型的索引，`T` 是想要检索的类型。如果使用类型的索引，或使用与 `variant` 的当前值不匹配的类型，这些函数抛出 `bad_variant_access` 异常。

```
cout << get<string>(v) << endl;
try {
    cout << get<0>(v) << endl;
} catch (const bad_variant_access& ex) {
    cout << "Exception: " << ex.what() << endl;
}
```

输出如下：

```
An std::string
Exception: bad variant access
```

为避免异常，可使用 `std::get_if<index>()` 或 `get_if<T>()` 辅助函数。这些函数接收指向 `variant` 的指针，返回指向请求值的指针；如果遇到错误，则返回 `nullptr`。

```
string* theString { std::get_if<string>(&v) };
int* theInt { get_if<int>(&v) };
cout << "retrieved string: " << (theString ? *theString : "null") << endl;
cout << "retrieved int: " << (theInt ? *theInt : 0) << endl;
```

输出如下：

```
retrieved string: An std::string
retrieved int: 0
```

可使用 `std::visit()` 辅助函数，将 `visitor` 模式应用于 `variant`。`visitor` 必须是可调用对象，可以接收可能存储在 `variant` 中的任何类型。假设以下类定义了多个重载的函数调用运算符，`variant` 中的每个可能类型对应一个。

```
class MyVisitor
{
public:
    void operator()(int i) { cout << "int " << i << endl; }
```

```
void operator()(const string& s) { cout << "string " << s << endl; }
void operator()(float f) { cout << "float " << f << endl; }
};
```

可将其与 `std::visit()` 一起使用，如下所示：

```
visit(MyVisitor{}, v);
```

这样就会根据 `variant` 中当前存储的值，调用适当的重载的函数调用运算符。这个示例的输出如下：

```
string An std::string
```

`variant` 不能存储数组，就像第 1 章介绍的 `optional` 一样，不能在 `variant` 中存储引用。可以存储指针、`reference_wrapper<const T>` 或 `reference_wrapper<T>` 的实例，见第 18 章。

24.2 any

`std::any` 在 `<any>` 中定义，是一个可包含任意类型值的类。可以使用任意构造函数或 `std::make_any()` 辅助函数创建实例。一旦构建，可确认 `any` 实例中是否包含值，以及所包含值的类型。要访问包含的值，需要使用 `any_cast()`，如果失败，会抛出 `bad_any_cast` 类型的异常。下面是一个示例：

```
any empty;
any anInt { 3 };
any aString { "An std::string."s };

cout << "empty.has_value = " << empty.has_value() << endl;
cout << "anInt.has_value = " << anInt.has_value() << endl << endl;

cout << "anInt wrapped type = " << anInt.type().name() << endl;
cout << "aString wrapped type = " << aString.type().name() << endl << endl;

int theInt { any_cast<int>(anInt) };
cout << theInt << endl;
try {
    int test { any_cast<int>(aString) };
    cout << test << endl;
} catch (const bad_any_cast& ex) {
    cout << "Exception: " << ex.what() << endl;
}
```

输出如下所示。注意，`aString` 的包装类型与编译器相关。

```
empty.has_value = 0
anInt.has_value = 1

anInt wrapped type = int
aString wrapped type = class std::basic_string<char,struct std::char_traits<char>,class
std::allocator<char> >

3
Exception: Bad any_cast
```

可将新值赋给 `any` 实例，甚至是不同类型的新值：

```
any something { 3 }; // Now it contains an integer.
something = "An std::string"s; // Now the same instance contains a string.
```

`any` 的实例可存储在标准库容器中。这样就可单个容器中存放异构数据。这么做的唯一缺点在于，只能通过显式执行 `any_cast` 来检索特定值，如下所示：

```
vector<any> v;
v.push_back(42);
v.push_back("An std::string"s);

cout << any_cast<string>(v[1]) << endl;
```

与 `optional` 和 `variant` 一样，无法存储 `any` 实例的引用。可存储指针，也可存储 `reference_wrapper<const T>` 或 `reference_wrapper<T>` 的实例。

24.3 元组

第 1 章介绍的在 `<utility>` 中定义的 `std::pair` 类可保存两个值，每个值都有特定的类型。每个值的类型都应该在编译期确定。下面是一个简单的例子：

```
pair<int, string> p1 { 16, "Hello World" };
pair p2 { true, 0.123f }; // Using CTAD.
cout << format("p1 = ({}, {})", p1.first, p1.second) << endl;
cout << format("p2 = ({}, {})", p2.first, p2.second) << endl;
```

输出如下所示：

```
p1 = (16, Hello World)
p2 = (1, 0.123)
```

还有 `std::tuple` 类，这个类定义在 `<tuple>` 中。`tuple` (元组) 是 `pair` 的泛化，允许存储任意数量的值，每个值都有自己特定的类型。和 `pair` 一样，`tuple` 的大小和值类型都是编译期确定的，都是固定的。

`tuple` 可通过 `tuple` 构造函数创建，需要指定模板类型和实际值。例如，下面的代码创建了一个 `tuple`，其第一个元素是一个整数，第二个元素是一个字符串，最后一个元素是一个布尔值：

```
using MyTuple = tuple<int, string, bool>;
MyTuple t1 { 16, "Test", true };
```

`std::get<i>()` 从 `tuple` 中获得第 `i` 个元素，`i` 是从 0 开始的索引；因此 `<0>` 表示 `tuple` 的第一个元素，`<1>` 表示 `tuple` 的第二个元素，以此类推。返回值的类型是 `tuple` 中那个索引位置的正确类型：

```
cout << format("t1 = ({}, {}, {})", get<0>(t1), get<1>(t1), get<2>(t1)) << endl;
// Outputs: t1 = (16, Test, 1)
```

可通过 `<typeinfo>` 中的 `typeid` 检查 `get<i>()` 是否返回了正确的类型。下面这段代码的输出表明，`get<1>(t1)` 返回的值确实是 `std::string` (如上所述，`typeid.name()` 返回的确切字符串依赖于编译器)。

```
cout << "Type of get<1>(t1) = " << typeid(get<1>(t1)).name() << endl;
// Outputs: Type of get<1>(t1) = class std::basic_string<char,
//          struct std::char_traits<char>, class std::allocator<char>
```

可以使用 `std::tuple_element` 类模板在编译期根据元素的索引获取元素的类型。`tuple_element` 要求指定元组的类型(在本例中是 `MyTuple`)，而不是像 `t1` 那样的实际元组实例。下面是示例：

```
cout << "Type of element with index 2 = "
    << typeid(tuple_element<2, MyTuple>::type).name() << endl;
// Outputs: Type of element with index 2 = bool
```

也可根据类型使用 `std::get<T>()` 从 `tuple` 中提取元素，其中 `T` 是要提取的元素(而不是索引)的类型。如果 `tuple` 有几个所需类型的元素，编译器会生成错误。例如，可从 `t1` 中提取字符串元素：

```
cout << "String = " << get<string>(t1) << endl;
// Outputs: String = Test
```

迭代 `tuple` 的值并不简单。无法编写简单循环或调用 `get<i>(mytuple)` 等，因为 `i` 的值在编译期必须是已知的。一种可能的解决方案是使用模板元编程，详见第 26 章，其中举了一个打印 `tuple` 值的示例。

可通过 `std::tuple_size` 模板查询 `tuple` 的大小。和 `tuple_element` 一样，`tuple_size` 要求指定 `tuple` 的类型，而不是实际的 `tuple` 实例：

```
cout << "Tuple Size = " << tuple_size<MyTuple>::value << endl;
// Outputs: Tuple Size = 3
```

如果不知道准确的 `tuple` 类型，始终可以使用 `decltype` 来查询类型，如下所示：

```
cout << "Tuple Size = " << tuple_size<decltype(t1)>::value << endl;
// Outputs: Tuple Size = 3
```

通过模板参数推导(CTAD)，在构造 `tuple` 时，可忽略模板类型形参，让编译器根据传递给构造函数的实参类型自动进行推导。例如，下面定义同样的 `t1` 元组，它包含一个整数、一个字符串和一个布尔值。注意，必须指定 `"Test"s`，以确保它是 `std::string`。

```
std::tuple t1 { 16, "Test"s, true };
```

因为类型的自动推导，不能通过 `&` 指定引用。如果需要通过构造函数的模板参数推导方式，生成一个包含引用或常量引用的 `tuple`，那么需要分别使用 `ref()` 和 `cref()`。它们都在 `<functional>` 中定义。例如，下面的构造会生成一个类型为 `tuple<int, double&, const double&, string&>` 的 `tuple`：

```
double d { 3.14 };
string str1 { "Test" };
std::tuple t2 { 16, ref(d), cref(d), ref(str1) };
```

为测试元组 `t2` 中的 `double` 引用，下面的代码首先将 `double` 变量的值写入控制台。然后调用 `get<1>(t2)`，这个函数实际上返回的是对 `d` 的引用，因为第二个 `tuple`(索引 1)元素使用了 `ref(d)`。第二行修改引用的变量的值，最后一行展示了 `d` 的值的确通过保存在 `tuple` 中的引用修改了。注意，第三行未能编译，因为 `cref(d)` 用于第三个 `tuple` 元素，也就是说，它是 `d` 的常量引用。

```
cout << "d = " << d << endl;
get<1>(t2) *= 2;
//get<2>(t2) *= 2;    // ERROR because of cref().
cout << "d = " << d << endl;
// Outputs: d = 3.14
//          d = 6.28
```

如果不使用构造函数的模板参数推导方法，可以使用 `std::make_tuple()` 工具函数创建一个 `tuple`。由于它是函数模板，它支持函数模板推导，因此允许通过仅指定实际值来创建元组。在编译期自动推

导类型，例如：

```
auto t2 { std::make_tuple(16, ref(d), cref(d), ref(str1) );}
```

24.3.1 分解元组

可采用两种方法，将一个元组分解为单独的元素：结构化绑定以及 `std::tie()`。

1. 结构化绑定

结构化绑定在 C++17 中可用，允许方便地将一个元组分解为多个变量。例如，下面的代码定义了一个 `tuple`，这个 `tuple` 包括一个整数、一个字符串和一个布尔值；此后，使用结构化绑定，将这个 `tuple` 分解为 3 个独立的变量：

```
tuple t1 { 16, "Test"s, true };
auto [i, str, b] { t1 };
cout << format("Decomposed: i = {}, str = \"{}\\\", b = {}\"", i, str, b) << endl;
```

还可以将元组分解为引用，允许通过引用修改元组的内容，这是示例：

```
auto& [i2, str2, b2] { t1 };
i2 *= 2;
str2 = "Hello World";
b2 = !b2;
```

使用结构化绑定，无法在分解元组时忽略特定元素。如果元组包含 3 个元素，则结构化绑定需要 3 个变量。如果想忽略元素，则必须使用 `tie()`。

2. tie()

如果在分解元组时不使用结构化绑定，可使用 `std::tie()` 工具函数，它生成一个引用的 `tuple`。下例首先创建一个 `tuple`，这个 `tuple` 包含一个整数、一个字符串和一个布尔值；然后创建 3 个变量，即整型变量、字符串变量和布尔变量，将这些变量的值写入控制台。`tie(i, str, b)` 调用会创建一个 `tuple`，其中包含对 `i` 的引用、对 `str` 的引用以及对 `b` 的引用。使用赋值运算符，将 `tuple t1` 赋给 `tie()` 的结果。由于 `tie()` 的结果是一个引用的 `tuple`，赋值实际上更改了 3 个独立变量中的值：

```
tuple t1 { 16, "Test", true };
int i { 0 };
string str;
bool b { false };
cout << format("Before: i = {}, str = \"{}\\\", b = {}\"", i, str, b) << endl;
tie(i, str, b) = t1;
cout << format("After: i = {}, str = \"{}\\\", b = {}\"", i, str, b) << endl;
```

结果如下：

```
Before: i = 0, str = "", b = 0
After: i = 16, str = "Test", b = 1
```

有了 `tie()`，可忽略一些不想分解的元素。并非使用分解值的变量名，而使用特殊的 `std::ignore` 值。例如，`t1` 元组的 `string` 元素可以被忽略，用下面的语句替换前面的 `tie()` 语句：

```
tie(i, ignore, b) = t1;
```

24.3.2 串联

通过 `std::tuple_cat()` 可将两个 `tuple` 串联为一个。在下面的例子中, `t3` 的类型为 `tuple<int, string, bool, double, string>`:

```
tuple t1 { 16, "Test"s, true };
tuple t2 { 3.14, "string 2"s };
auto t3 { tuple_cat(t1, t2) };
```

24.3.3 比较

`tuple` 支持所有比较运算符。为了能使用这些比较运算符, `tuple` 中存储的元素类型也应该支持这些操作。例如:

```
tuple t1 { 123, "def" };
tuple t2 { 123, "abc" };
if (t1 < t2) {
    cout << "t1 < t2" << endl;
} else {
    cout << "t1 >= t2" << endl;
}
```

输出如下所示:

```
t1 >= t2
```

对于包含多个数据成员的自定义类型, `tuple` 比较可用于方便地实现这些类型的按词典比较运算符。例如, 如下的类包含 3 个数据成员:

```
class Foo
{
public:
    Foo(int i, string s, bool b) : m_int { i }, m_str { move(s) }, m_bool { b }
    {
    }
private:
    int m_int;
    string m_str;
    bool m_bool;
};
```

正确实现该类的完整比较运算符集合并不简单。通过 `std::tie()` 和 C++20 的三向比较运算符 (`operator<=>`), 它就变成了简单的一行程序。下面是 `Foo` 类的运算符 `<=>` 方法的实现:

```
auto operator<=>(const Foo& rhs)
{
    return tie(m_int, m_str, m_bool) <=>
           tie(rhs.m_int, rhs.m_str, rhs.m_bool);
}
```

下面是一个使用示例:

```
Foo f1 { 42, "Hello", false };
Foo f2 { 42, "World", false };
cout << (f1 < f2) << endl; // Outputs 1
cout << (f2 > f1) << endl; // Outputs 1
```


24.3.4 make_from_tuple()

使用 `std::make_from_tuple<T>()` 可构建一个 T 类型的对象，将给定 tuple 的元素作为参数传递给 T 的构造函数。例如，假设具有以下类：

```
class Foo
{
public:
    Foo(string str, int i) : m_str { move(str) }, m_int { i } {}
private:
    string m_str;
    int m_int;
};
```

可按如下方式使用 `make_from_tuple()`：

```
auto myTuple { "Hello world.", 42 };
auto foo { make_from_tuple<Foo>(myTuple) };
```

技术上讲，提供给 `make_from_tuple()` 的实参未必是一个 tuple，但必须支持 `std::get<N>()` 和 `tuple_size`。`std::array` 和 `pair` 也满足这些要求。

在日常工作中，该函数并不实用，不过如果要编写使用模板的泛型代码，或进行模板元编程，那么这个函数可提供便利。

24.3.5 apply()

`std::apply()` 调用给定的函数、lambda 表达式和函数对象等，将给定 tuple 的元素作为实参传递。下面是一个例子：

```
int add(int a, int b) { return a + b; }
...
cout << apply(add, tuple { 39, 3 }) << endl;
```

与 `make_from_tuple()` 一样，在日常工作中，该函数并不实用；不过，如果要编写使用模板的泛型代码，或进行模板元编程，那么这个函数可提供便利。

24.4 本章小结

本章概述了 C++ 标准提供的其他功能。你在这一章学习了如何使用 `variant` 和 `any` 词汇表数据类型，它们是对 `optional` 词汇表类型的补充。你也学习了 `tuple`，它是 `pair` 的泛化。

本章是本书第 III 部分的总结。下一部分讨论一些更高级的主题，首先介绍如何通过自己的标准库兼容的算法和数据结构来定制和扩展 C++ 标准库提供的功能。

24.5 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

练习 24-1 第 14 章解释了 C++ 中的错误处理，并解释了这里基本上有两个主要选项，要么使用错误码，要么使用异常。建议使用异常进行错误处理，但在本练习中，你将使用错误码。编写一个简单的 `Error` 类，它只存储单个消息，有一个构造函数来设置消息，有一个 `getter` 来获取消息。编写一个参数名为 `fail` 的 `getData()` 函数。如果 `fail` 为 `false`，则函数返回数据的 `vector`，否则，返回 `Error` 的实例。不允许使用非 `const` 输出参数的引用。在 `main()` 函数中测试你的实现。

练习 24-2 大多数命令行应用程序接收命令行参数。在本书的大部分示例代码中，`main` 函数就是 `main()`。然而 `main()` 也可以接收参数：`main(int argc, char** argv)`，其中 `argc` 是命令行参数的数量，`argv` 是一个字符串数组，每个参数对应一个字符串。本习题假设命令行参数的形式为 `name=value`。编写一个函数，它可以解析单个参数并返回一个包含参数名称和 `variant` 的 `pair`。如果 `variant` 的值可以被解析为布尔值，则作为布尔值。如果该值被解析为整数，则为整数，否则为字符串。要拆分 `name=value` 字符串，可以使用正则表达式(参见第 21 章)。要解析整数，可以使用第 2 章介绍的某个函数。在 `main()` 函数中，循环所有命令行参数并解析它们，使用 `holds_alternative()` 将解析后的结果输出到标准输出上。

练习 24-3 修改习题 24-2 的答案，不要使用 `holds_alternative()`，使用一个访问者将解析后的结果输出到标准输出上。

练习 24-4 修改习题 24-3 的答案，使用 `tuple` 替换 `pair`。

第IV部分

掌握 C++的高级特性

- ▶ 第 25 章 自定义和扩展标准库
- ▶ 第 26 章 高级模板
- ▶ 第 27 章 C++多线程编程



第25章

自定义和扩展标准库

本章内容

- 分配器的含义
- 如何通过编写自定义算法、容器和迭代器来扩展标准库

第 16 章、第 18 章和第 20 章中提到，标准库是功能强大的通用容器和算法的集合。你目前了解的信息应该足以满足大部分应用程序的需要。然而，前面的章节只介绍了库的基本功能。可根据需要的方式任意自定义和扩展标准库。例如，可编写自己的容器、算法和迭代器；甚至可指定容器使用自定义的内存分配方案。本章通过容器 `directed_graph` 的开发过程，讲解这些高级特性。

警告：

很少需要自定义和扩展标准库。如果对前面章节讲解的基本标准库容器和算法不甚明了，可跳过这一章。不过，如果想理解标准库，而不只满足于使用标准库，那么本章值得阅读。你应该很熟悉第 15 章讲解的运算符重载内容，由于这一章大量使用了模板，因此在继续阅读本章之前还应该熟悉第 12 章讲解的模板。

25.1 分配器

每个标准库容器都接收 `Allocator` 类型作为模板参数，大部分情况下默认值就足够了。例如，`vector` 模板的定义如下所示：

```
template <class T, class Allocator = allocator<T>> class vector;
```

容器构造函数还允许指定 `Allocator` 类型的对象。通过这些额外参数可自定义容器分配内存的方式。容器执行的每一次内存分配都是通过调用 `Allocator` 对象的 `allocate()` 方法进行的。相反，每一次内存释放都是通过调用 `Allocator` 对象的 `deallocate()` 方法进行的。标准库提供了一个默认的 `Allocator` 类，名为 `allocator`，定义在 `<memory>` 中，这个类通过对 `operator new` 和 `operator delete` 进行包装实现了这些方法。

请记住，`allocate()` 只是分配了一个足够大的未初始化内存块，而没有调用任何对象构造函数。类似地，`deallocate()` 只是释放内存块，而不调用任何析构函数。一旦分配了一块内存，就可以使用

placement new 运算符(见第 15 章)在特定位置上构造一个对象。下面的代码片段显示了一个示例。第 29 章将展示分配器用于实现对象池的更现实的用法。

```
class MyClass {};
int main()
{
    // Create an allocator to use.
    allocator<MyClass> alloc;
    // Allocate uninitialized memory block for 1 instance of MyClass.
    auto* memory { alloc.allocate(1) };
    new(memory) MyClass{};
    // Destroy MyClass instance.
    destroy_at(memory);
    // Deallocate memory block.
    alloc.deallocate(memory, 1);
    memory = nullptr;
}
```

如果程序中的容器需要使用自定义的内存分配和释放方案，那么可编写自己的 Allocator 类。有几种原因需要使用自定义的分配器。例如，如果底层分配器的性能无法接受，但可构建替换的分配器。如果必须给操作系统特定的功能分配空间，例如共享内存段，那么通过自定义分配器允许在共享内存段内使用标准库容器。自定义分配器的使用很复杂，如果不小心，可能产生很多问题，因此不应该草率地使用自定义分配器。

任何提供了 allocate()、deallocate()和其他需要的方法和类型别名的类都可替换默认的 allocator 类。

C++17 引入了多态内存分配器的概念。对于指定为模板类型参数的容器的分配器，问题在于两个十分相似但具有不同分配器类型的容器区别很大。例如，具有不同分配器模板类型参数的两个 vector<int>容器是不同的，因此不能将其中一个赋给另一个。

std::pmr 名称空间的<memory_resource>中定义的多态内存分配器有助于解决这个问题。std::pmr::polymorphic_allocator 是适当的分配器类，因为它满足各种要求，如具有 allocate()和 deallocate()方法。polymorphic_allocator 的分配行为取决于构建期间的 memory_resource，而非取决于模板类型参数。因此，在分配和释放内存时，虽然具有相同的类型，但不同 polymorphic_allocator 的行为迥异。C++标准提供了一些内置的内存资源，可供初始化多态内存分配器：synchronized_pool_resource、unsynchronized_pool_resource 和 monotonic_buffer_resource。

然而，根据经验，这些自定义分配器和多态内存分配器相当高级，很少使用。因此本书略去了这些细节。要了解详情，请参阅附录B中列出的C++标准库的相关书籍。

25.2 扩展标准库

标准库包含很多有用的容器、算法和迭代器，可在应用程序中使用这些工具。然而，任何库都不可能包含所有潜在客户可能需要的所有工具。因此，库最好具有可扩展性：允许客户基于基本功能进行适配和添加功能，从而得到客户需要的准确功能。标准库本质上就是可扩展的，因为标准库的基础结构将数据和操作数据的算法分开了。提供符合标准库标准的迭代器，就可以自行编写能够用于标准库算法的容器。类似地，还可编写能操作标准容器迭代器的算法。注意不能把自己的容器和算法放在 std 名称空间中。

注意:

本书通常使用规范来命名函数和方法，没有任何下画线，除了第一个单词外，其他都大写，例如 `getIndex()`。然而本章讨论的是扩展标准库，因此它使用标准库使用的命名规范。意味着函数名和方法名都小写，单词之前用下画线分隔。例如 `get_index()`。类名也使用标准库命名规范。

25.2.1 扩展标准库的原因

准备用 C++ 编写算法或容器时，可遵循或不遵循标准库的约定。对于简单的容器和算法来说，可能不值得为遵循标准库规范而付出额外的努力。然而，对于打算重用的重要代码，这些努力是值得的。首先，代码更容易被其他 C++ 程序员理解，因为代码遵从构建良好的接口规范。其次，可将自己的容器和算法与标准库中的其他部分(算法或容器)结合使用，而不需要提供特别的修改版或适配器。最后，可强迫遵循开发稳固代码所需的严格规范。

25.2.2 编写标准库算法

第 16 章和第 20 章描述了标准库中的一组有用算法，但有时需要在自己的程序中使用新算法。这种情况下，编写和标准算法一样能操作标准库迭代器的算法并不难。

`find_all()`

假设需要在指定范围内找到满足某个谓词的所有元素，包括它们的位置。`find()` 和 `find_if()` 是最符合条件的备选算法，但这些算法返回的都是仅引用一个元素的迭代器。可使用 `copy_if()` 找出所有满足谓词的元素，但会用所找到元素的副本填充输出。如果想要避免复制，可使用 `copy_if()` 和 `back_insert_iterator`(第 17 章“理解迭代器和范围库”)到 `vector<reference_wrapper<T>>` 中，但这不能给出所找到元素的位置。事实上，无法利用任何一种标准算法找到满足谓词的所有元素的迭代器，但可自行编写能提供这个功能的版本，称为 `find_all()`。

第一个任务是定义函数原型。可遵循 `copy_if()` 采用的模型。这应该是一个带有 3 个类型参数的模板化函数：输入迭代器类型、输出迭代器类型和谓词类型。这个函数的参数为输入序列的首尾迭代器、输出序列的首迭代器以及谓词对象。与 `copy_if()` 一样，该算法给输出序列返回一个迭代器，指向输出序列中存储的最后一个元素后面的那个元素。下面是算法原型：

```
template <typename InputIterator, typename OutputIterator, typename Predicate>
OutputIterator find_all(InputIterator first, InputIterator last,
                       OutputIterator dest, Predicate pred);
```

另一种可选方案是忽略输出迭代器，给输入序列返回一个迭代器，遍历输入序列中所有匹配的元素，但是这种方案要求编写自定义的迭代器类，见稍后的讨论。

下一项任务是编写算法的实现。`find_all()` 算法遍历输入序列中的所有元素，给每个元素调用谓词，把匹配元素的迭代器存储在输出序列中。下面是算法的实现：

```
template <typename InputIterator, typename OutputIterator, typename Predicate>
OutputIterator find_all(InputIterator first, InputIterator last,
                       OutputIterator dest, Predicate pred)
{
    while (first != last) {
        if (pred(*first)) {
            *dest = first;
            ++dest;
        }
    }
}
```



```

        ++first;
    }
    return dest;
}

```

与 `copy_if()` 类似，该算法也只覆盖输出序列中的已有元素，所以确保输出序列足够大，以存储结果，或者使用迭代适配器，例如下面代码中的 `back_inserter`。找到引用所有匹配的元素后，代码计算找到的元素个数，即 `matches` 中迭代器的个数。然后，代码遍历结果，打印每个元素。

```

vector vec { 3, 4, 5, 4, 5, 6, 5, 8 };
vector<vector<int>::iterator> matches;

find_all(begin(vec), end(vec), back_inserter(matches),
    [](int i){ return i == 5; });

cout << format("Found {} matching elements: ", matches.size()) << endl;
for (const auto& it : matches) {
    cout << *it << " at position " << (it - cbegin(vec)) << endl;;
}

```

输出如下所示：

```

Found 3 matching elements:
5 at position 2
5 at position 4
5 at position 6

```

25.2.3 编写标准库容器

C++ 标准包含要把容器作为标准库容器应该满足的要求列表。此外，如果想要一个容器为顺序容器(例如 `vector`)、有序关联容器(例如 `map`)或无序关联容器(例如 `unordered_map`)，那么这个容器还必须满足额外的要求。

我们对编写自定义容器的建议是：首先编写遵循一般标准库规则的基本容器，例如编写类模板，不用太关心遵循标准库的细节。开发基本实现后，添加迭代器支持，以便它能配合标准库框架工作。接下来，添加方法和类型别名来满足所有基本的容器需求，最后满足任何附加的容器需求。本章采用这种方法开发有向图数据结构，也称为有向图。

基本的有向图

某些 C++ 标准库容器在其实现中肯定使用了图，但标准没有为用户提供任何类似图的数据结构。因此，实现自己的图听起来像是编写自己的标准库兼容容器的完美示例。

在开始编写任何代码之前，先看看有向图是什么类型的数据结构，以及如何在内存中表示它的数据。图 25-1 是一个有向图示例的可视化表示。基本上，有向图由一组节点(也成为顶点)组成，节点间由边连接。另外，每条边都有方向，由箭头表示，这就是它称为有向图的原因。

有多种方法可以在内存中存储这样的数据结构，如边表、邻接矩阵和邻接表。这个实现使用邻接表，节点存储在 `vector` 中，每个节点都有邻接表列出它的相邻节点。来看一个示例，假设有图 25-2 所示的有向图。

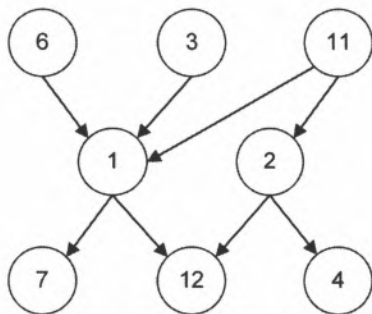


图 25-1 有向图示例的可视化表示

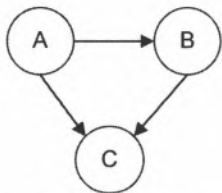


图 25-2 有向图

用邻接表表示图 25-2 会得到如表 25-1 所示的数据结构：

表 25-1 数据结构

节点	邻接表
A	B, C
B	C
C	

可以存储在 `vector` 中，`vector` 中的每个元素表示图表中的一行，也就是说，每个 `vector` 的元素代表一个节点及其对应的邻接表。我们先从一个基本的实现开始，不要太担心标准库的兼容性。第一部分实现了一个简单但功能齐全的 `directed_graph`。注意这可能不是有向图的最佳或性能最好的实现，但这不是本章的重点。重点是按照标准库的理念来完成数据结构的制作过程。

graph_node 类模板

`directed_graph` 的实现使用了节点的概念，所以要实现的第一部分代码就是表示图的单个节点的数据结构。一个节点有一个值和相邻节点的列表，存储这些相邻节点的索引集合。使用集合可以确保列表中不存储重复的相邻索引。该类具有构造函数，用于为给定值构造新 `graph_node`，具有 `value()` 方法，用于检索节点值，并支持相等运算。该定义在 `graph_node.cpp` 中的 `directed_graph::node` 模块实现分区文件中，它在 `details` 的名称空间中，但并不是从该模块导出，因为 `directed_graph` 的用户不应该直接使用 `graph_nodes` 本身。`directed_graph` 类模板需要访问相邻节点的列表，因此它是 `graph_node` 的好友。下面是 `graph_node` 的接口，注意 `[[nodiscard]]` 属性的使用，在第 1 章中介绍过。

```
module directed_graph::node;
...
namespace details
{
    template <typename T>
    class graph_node
    {
    public:
        // Constructs a graph_node for the given value.
        graph_node(directed_graph<T>* graph, const T& t);
        graph_node(directed_graph<T>* graph, T&& t);

        // Returns a reference to the stored value.
        [[nodiscard]] T& value() noexcept;
```

```

[[nodiscard]] const T& value() const noexcept;

// C++20 defaulted operator==.
bool operator==(const graph_node&) const = default;

private:
    friend class directed_graph<T>;

    // A pointer to the graph this node is in.
    directed_graph<T>* m_graph;

    // Type alias for the container type used to store the adjacency list.
    using adjacency_list_type = std::set<size_t>;

    // Returns a reference to the adjacency list.
    [[nodiscard]] adjacency_list_type& get_adjacent_nodes_indices();
    [[nodiscard]] const adjacency_list_type&
        get_adjacent_nodes_indices() const;

    T m_data;
    adjacency_list_type m_adjacentNodeIndices;
};
}

```

这些方法的实现很简单:

```

namespace details
{
    template <typename T>
    graph_node<T>::graph_node(directed_graph<T>* graph, const T& t)
        : m_graph { graph }, m_data { t } { }

    template <typename T>
    graph_node<T>::graph_node(directed_graph<T>* graph, T&& t)
        : m_graph { graph }, m_data { std::move(t) } { }

    template <typename T>
    T& graph_node<T>::value() noexcept { return m_data; }

    template <typename T>
    const T& graph_node<T>::value() const noexcept { return m_data; }

    template <typename T>
    typename graph_node<T>::adjacency_list_type&
    graph_node<T>::get_adjacent_nodes_indices()
    {
        return m_adjacentNodeIndices;
    }

    template <typename T>
    const typename graph_node<T>::adjacency_list_type&
    graph_node<T>::get_adjacent_nodes_indices() const
    {
        return m_adjacentNodeIndices;
    }
}

```

注意 `m_graph` 数据成员是一个指针。可以修改实现使其成为引用。还必须提供拷贝/移动构造函数和拷贝/移动赋值运算符的实现，因为一旦有了引用数据成员，编译器生成的运算符就会自动删除。

这个实现提供默认运算符`==`，它会自动默认运算符`!=`。这是 C++20 的一个特性，如果编译器不支持这一点，则必须为这两个运算符提供自己的实现，它们只是比较所有数据成员。

现在有了 `graph_node` 实现，再来看看 `directed_graph` 类本身。

`directed_graph` 接口

`directed_graph` 支持 3 种基本操作：插入、删除和查找；此外它是可交换的。它在 `directed_graph` 模块中定义，下面是 `directed_graph` 类模板的公共部分。

```
export module directed_graph;
...
export template <typename T>
class directed_graph
{
public:
    // For insert to be successful, the value shall not be in the graph yet.
    // Returns true if a new node with given value has been added to
    // the graph, and false if there was already a node with the given value.
    bool insert(const T& node_value);
    bool insert(T&& node_value);

    // Returns true if the given node value was erased, false otherwise.
    bool erase(const T& node_value);

    // Returns true if the edge was successfully created, false otherwise.
    bool insert_edge(const T& from_node_value, const T& to_node_value);

    // Returns true if the given edge was erased, false otherwise.
    bool erase_edge(const T& from_node_value, const T& to_node_value);

    // Removes all nodes from the graph.
    void clear() noexcept;

    // Returns a reference to the node with given index.
    // No bounds checking is done.
    T& operator[](size_t index);
    const T& operator[](size_t index) const;

    // Two directed graphs are equal if they have the same nodes and edges.
    // The order in which the nodes and edges have been added does not
    // affect equality.
    bool operator==(const directed_graph& rhs) const;
    bool operator!=(const directed_graph& rhs) const;

    // Swaps all nodes between this graph and the given graph.
    void swap(directed_graph& other_graph) noexcept;

    // Returns the number of nodes in the graph.
    [[nodiscard]] size_t size() const noexcept;

    // Returns a set with the adjacent nodes of a given node.
    // If the given node does not exist, an empty set is returned.
    [[nodiscard]] std::set<T> get_adjacent_nodes_values(
```

```

        const T& node_value) const;
    private:
        // Implementation details not shown yet.
};

```

如你所见，值类型是一个模板类型参数，类似于标准库 `vector` 容器，接口看起来很简单。注意，该接口没有任何用户自定义的拷贝和移动构造函数、拷贝和移动赋值运算符以及析构函数，也就是说，该类遵循第 9 章讨论到的零规则。

现在来看公有方法的具体实现。

实现

在完成 `directed_graph` 接口之后，需要选择实现模型。如前所述，此实现将有向图存储为节点列表，其中每个节点包含其值及其邻接节点索引集合。由于邻接节点列表包含到其他节点的索引，节点应该基于它们的索引进行访问。因此，`vector` 是存储节点的最合适容器。每个节点都表示为一个 `graph_node` 实例。因此，最终的数据结构是 `graph_node` 的 `vector`。下面是 `directed_graph` 类的第一个私有成员：

```

private:
    using nodes_container_type = std::vector<details::graph_node<T>>;
    nodes_container_type m_nodes;

```

搜索节点

`graph` 上的插入和删除操作需要代码来找到具有给定节点值的元素。因此有一个执行此任务的私有辅助方法是有帮助的。同时提供 `const` 和非 `const` 版本：

```

// Helper method to return an iterator to the given node, or the end iterator
// if the given node is not in the graph.
typename nodes_container_type::iterator findNode(const T& node_value);
typename nodes_container_type::const_iterator findNode(const T& node_value) const;

```

`findNode()` 的非 `const` 版本的实现如下：

```

template <typename T>
typename directed_graph<T>::nodes_container_type::iterator
    directed_graph<T>::findNode(const T& node_value)
{
    return std::find_if(std::begin(m_nodes), std::end(m_nodes),
        [&node_value](const auto& node) { return node.value() == node_value; });
}

```

方法的主体并不太复杂。它使用标准库中的 `find_if` 算法(第 20 章)来搜索图中的所有节点，寻找值等于 `node_value` 参数的节点。如果在途中找到这样一个节点，则返回指向该节点的迭代器；否则返回一个 `end` 迭代器。

这个方法的函数头语法有些混乱，特别是 `typename` 关键字的使用。当使用依赖于模板参数的类型时，必须使用 `typename` 关键字。具体来说，类型 `nodes_container_type::iterator` 是 `vector<details::graph_node<T>>::iterator`，它依赖 `T` 模板参数。

该方法的 `const` 版本直接转到非 `const` 版本来避免代码重复：

```

template <typename T>
typename directed_graph<T>::nodes_container_type::const_iterator
    directed_graph<T>::findNode(const T& node_value) const
{

```

```

    return const_cast<directed_graph<T>*>(this)->findNode(node_value);
}

```

插入节点

`insert()`必须首先检查给定值的节点是否已经存在于图中。如果不存在，则可以为给定值创建一个新节点。公共接口提供了一个接收 `const` 引用的 `insert()` 方法和一个接收右值引用的重载方法。前者的实现就是把工作转发到右值引用的重载版本中。对 `emplace_back()` 的调用通过将指向 `directed_graph` 的指针和节点的值传递给 `graph_node` 构造函数来构造一个新的 `graph_node`：

```

template <typename T>
bool directed_graph<T>::insert(T&& node_value)
{
    auto iter { findNode(node_value) };
    if (iter != std::end(m_nodes)) {
        // Value is already in the graph, return false.
        return false;
    }
    m_nodes.emplace_back(this, std::move(node_value));
    // Value successfully added to the graph, return true.
    return true;
}

template <typename T>
bool directed_graph<T>::insert(const T& node_value)
{
    T copy { node_value };
    return insert(std::move(copy));
}

```

插入边

一旦将节点添加到图中，这些节点之间的边就可以构建成一个有向图。为此，提供了一个 `insert_edge()` 方法，该方法需要两个参数：边的起始节点值和边应该指向的节点的值。该方法做的第一件事是在图中搜索起始节点和到达节点。如果在图中没有找到它们中的任何一个，该方法会返回 `false`。如果两者都找到，那么代码会通过调用私有辅助函数 `get_index_of_node()` 来计算包含 `to_node_value` 节点的索引，最后将该索引添加到包含 `from_node_value` 节点的邻接表中。记得在第 18 章中，`set` 的 `insert()` 方法返回 `pair<iterator, bool>`，其中布尔值表示是否插入成功，这就是在 `return` 语句中对 `insert()` 的结果使用 `.second` 的原因。

```

template <typename T>
bool directed_graph<T>::insert_edge(const T& from_node_value,
    const T& to_node_value)
{
    const auto from { findNode(from_node_value) };
    const auto to { findNode(to_node_value) };
    if (from == std::end(m_nodes) || to == std::end(m_nodes)) {
        return false;
    }
    const size_t to_index { get_index_of_node(to) };
    return from->get_adjacent_nodes_indices().insert(to_index).second;
}

```

`get_index_of_node()`辅助方法的实现如下:

```
template <typename T>
size_t directed_graph<T>::get_index_of_node(
    const typename nodes_container_type::const_iterator& node) const noexcept
{
    const auto index { std::distance(std::cbegin(m_nodes), node) };
    return static_cast<size_t>(index);
}
```

删除节点

`erase()`遵循与 `insert()`相同的模式: 它首先通过调用 `findNode()`尝试查找给定的节点。如果节点存在, 则将其从图中删除, 否则什么也不做。从图中删除已有节点有两步:

- (1) 将待删除节点从所有其他节点的邻接表中删除。
- (2) 从节点列表中删除实际节点。

对于第(1)步, 提供了一个私有辅助方法 `remove_all_links_to()`。该方法必须更新剩余的相邻节点索引, 以考虑从图中删除一个节点。首先它计算 `node_index`, 即给定节点在节点 `vector` 中的索引。然后它遍历所有节点的邻接表, 从每个表中删除 `node_index` 并更新剩余的索引。一个棘手的部分是, 邻接表是一个 `set`, 而 `set` 不允许修改它的值。因此, 第二步实现是将 `set` 转换成一个 `vector`, 使用 `for_each()` 算法更新所有需要更新的索引, 最后清空 `set` 并插入更新后的索引。这可能不是性能最佳的实现, 但是正如前面提到的, 这不是本文讨论的重点。

```
template <typename T>
void directed_graph<T>::remove_all_links_to(
    typename nodes_container_type::const_iterator node_iter)
{
    const size_t node_index { get_index_of_node(node_iter) };

    // Iterate over all adjacency lists of all nodes.
    for (auto&& node : m_nodes) {
        auto& adjacencyIndices { node.get_adjacent_nodes_indices() };
        // First, remove references to the to-be-deleted node.
        adjacencyIndices.erase(node_index);
        // Second, modify all adjacency indices to account for removal of a node.
        std::vector<size_t> indices(std::begin(adjacencyIndices),
            std::end(adjacencyIndices));
        std::for_each(std::begin(indices), std::end(indices),
            [node_index](size_t& index) {
                if (index > node_index) { --index; }
            });
        adjacencyIndices.clear();
        adjacencyIndices.insert(std::begin(indices), std::end(indices));
    }
}
```

有了这个辅助方法, 真正的 `erase()`方法的实现就会很简单:

```
template <typename T>
bool directed_graph<T>::erase(const T& node_value)
{
    auto iter { findNode(node_value) };
    if (iter == std::end(m_nodes)) {
        return false;
    }
}
```

```

    }
    remove_all_links_to(iter);
    m_nodes.erase(iter);
    return true;
}

```

删除边

删除边的过程与添加边非常相似。如果没有找到 `from` 或 `to` 节点，则什么都不做，否则，值为 `to_node_value` 的节点的索引将从值为 `from_node_value` 的节点的邻接表中删除：

```

template <typename T>
bool directed_graph<T>::erase_edge(const T& from_node_value,
    const T& to_node_value)
{
    const auto from { findNode(from_node_value) };
    const auto to { findNode(to_node_value) };
    if (from == std::end(m_nodes) || to == std::end(m_nodes)) {
        return false; // nothing to erase
    }
    const size_t to_index { get_index_of_node(to) };
    from->get_adjacent_nodes_indices().erase(to_index);
    return true;
}

```

删除所有元素

`clear()`方法简单清空了整个图：

```

template <typename T>
void directed_graph<T>::clear() noexcept
{
    m_nodes.clear();
}

```

交换图

因为 `directed_graph` 只有一个数据成员，一个 `vector` 容器，所以交换两个 `directed_graph` 就意味着交换它们的单个数据成员：

```

template <typename T>
void directed_graph<T>::swap(directed_graph& other_graph) noexcept
{
    m_nodes.swap(other_graph.m_nodes);
}

```

还提供了下面独立导出的 `swap()` 函数，它直接转发给了公有 `swap()` 方法：

```

export template <typename T>
void swap(directed_graph<T>& first, directed_graph<T>& second)
{
    first.swap(second);
}

```

访问节点

`directed_graph` 的公有接口支持通过 `operator[]` 的重载访问节点的索引。实现很简单，就像 `vector`

一样，重载的 `operator[]` 不会对请求的索引进行任何边界检查：

```
template <typename T>
T& directed_graph<T>::operator[](size_t index)
{
    return m_nodes[index].value();
}

template <typename T>
const T& directed_graph<T>::operator[](size_t index) const
{
    return m_nodes[index].value();
}
```

比较图

当且仅当两个 `directed_graph` 包含相同的节点集合并且所有节点之间具有相同的边时，它们才相等。一个稍微复杂的事实是，这两个 `directed_graph` 可以通过以稍微不同顺序添加节点的方式来创建；因此，实现不能只是比较 `m_nodes` 数据成员，还需要做更多的工作。

代码首先检查两个 `directed_graph` 的大小。如果大小不同，两个图不能相同。如果大小相同，则代码将遍历其中一个图的所有节点。对于每个节点，它试图在另一个图中找到相同的节点。如果没有找到这样的节点，则图不相同。如果它确实找到了这样的节点，则使用 `get_adjacent_values()` 辅助方法将相邻节点索引转换为相邻节点值，然后比较这些值是否相等。

```
template <typename T>
bool directed_graph<T>::operator==(const directed_graph& rhs) const
{
    if (m_nodes.size() != rhs.m_nodes.size()) { return false; }

    for (auto&& node : m_nodes) {
        const auto rhsNodeIter { rhs.findNode(node.value()) };
        if (rhsNodeIter == std::end(rhs.m_nodes)) { return false; }

        const auto adjacent_values_lhs {
            get_adjacent_nodes_values(node.get_adjacent_nodes_indices()) };
        const auto adjacent_values_rhs { rhs.get_adjacent_nodes_values(
            rhsNodeIter->get_adjacent_nodes_indices()) };
        if (adjacent_values_lhs != adjacent_values_rhs) { return false; }
    }
    return true;
}

template <typename T>
std::set<T> directed_graph<T>::get_adjacent_nodes_values(
    const typename details::graph_node<T>::adjacency_list_type& indices) const
{
    std::set<T> values;
    for (auto&& index : indices) { values.insert(m_nodes[index].value()); }
    return values;
}
```

运算符 `!=` 的实现直接转发给运算符 `==`：

```
template <typename T>
bool directed_graph<T>::operator!=(const directed_graph& rhs) const
{
    return !(*this == rhs);
}
```

```

    return !(*this == rhs);
}

```

获取相邻节点

公共接口中提供了一个 `get_adjacent_nodes_values()` 方法，接收 `T` 的 `const` 引用类型值 `node_value` 作为参数。它返回一个集合，集合中包含与给定节点相邻的节点的值。如果给定的节点不存在，则返回空集合。就像前面实现的那样，该实现使用 `get_adjacent_nodes_values()` 重载接受索引列表：

```

template <typename T>
std::set<T> directed_graph<T>::get_adjacent_nodes_values(const T& node_value) const
{
    auto iter { findNode(node_value) };
    if (iter == std::end(m_nodes)) { return std::set<T>{}; }
    return get_adjacent_nodes_values(iter->get_adjacent_nodes_indices());
}

```

查询图的大小

最后，`size()` 方法返回图中的节点数：

```

template <typename T>
size_t directed_graph<T>::size() const noexcept
{
    return m_nodes.size();
}

```

打印图

图可以用一种称为 DOT 的标准格式打印，这是一种图形描述语言。有一些可以理解 DOT 格式图的工具，它可以将图转换为图描述。为了更容易地测试 `directed_graph` 代码，可以使用下面的 `to_dot()` 转换函数。下一节将给出一个使用它的示例。

```

// Returns a given graph in DOT format.
export template <typename T>
std::wstring to_dot(const directed_graph<T>& graph, std::wstring_view graph_name)
{
    std::wstringstream wss;
    wss << format(L"digraph {} {{{", graph_name.data()) << std::endl;
    for (size_t index { 0 }; index < graph.size(); ++index) {
        const auto& node_value { graph[index] };
        const auto adjacent_nodes { graph.get_adjacent_nodes_values(node_value) };
        if (adjacent_nodes.empty()) {
            wss << node_value << std::endl;
        } else {
            for (auto&& node : adjacent_nodes) {
                wss << format(L"{} -> {}", node_value, node) << std::endl;
            }
        }
    }
    wss << "}" << std::endl;
    return wss.str();
}

```

使用基本的有向图

现在已经完成了一个基本有向图类的完整实现。现在是测试这个类的时候了，下面是一个演示基本 `directed_graph` 类模板的小程序：

```

directed_graph<int> graph;
// Insert some nodes and edges.
graph.insert(11);
graph.insert(22);
graph.insert(33);
graph.insert(44);
graph.insert(55);
graph.insert_edge(11, 33);
graph.insert_edge(22, 33);
graph.insert_edge(22, 44);
graph.insert_edge(22, 55);
graph.insert_edge(33, 44);
graph.insert_edge(44, 55);
wcout << to_dot(graph, L"Graph1");

// Remove an edge and a node.
graph.erase_edge(22, 44);
graph.erase(44);
wcout << to_dot(graph, L"Graph1");

// Print the size of the graph.
cout << "Size: " << graph.size() << endl;

```

输出如下:

```

digraph Graph1 {
    11 -> 33
    22 -> 33
    22 -> 44
    22 -> 55
    33 -> 44
    44 -> 55
    55
}
digraph Graph1 {
    11 -> 33
    22 -> 33
    22 -> 55
    33
    55
}
Size: 4

```

25.2.4 将 directed_graph 实现为标准库容器

在前几节实现的基本的 `directed_graph` 遵循标准库的精神，而不是标准库的字母。对于大多数目的，前面的实现已经足够好了。但是，如果想要在 `directed_graph` 上使用标准库算法，则需要多做一些工作。C++标准指定了数据结构作为标准库容器必须提供的方法和类型别名。

需要的类型别名

C++标准指定每个标准库容器都要有表 25-2 所示的 `public` 类型别名。

表 25-2 public 类型别名

类型名称	说明
value_type	容器中保存的元素类型
reference	容器中保存的元素类型的引用
const_reference	容器中保存的元素类型的 const 引用
iterator	遍历容器中元素的类型
const_iterator	另一个版本的 iterator，遍历容器中的 const 元素
size_type	表示容器中元素个数的类型，通常为 size_t(来自<cstddef>)
difference_type	表示用于容器的两个 iterator 间差值的类型，通常为 ptrdiff_t(来自<cstddef>)

下面是 directed_graph 实现这些类型别名的类模板定义，除了 iterator 和 const_iterator。后面将详细讲解迭代器的编写方式。

```
export template <typename T>
class directed_graph
{
    public:
        using value_type = T;
        using reference = value_type&;
        using const_reference = const value_type&;
        using size_type = size_t;
        using difference_type = ptrdiff_t;
        // Remainder of class definition omitted for brevity.
};
```

使用这些类型别名，可以稍微修改一些方法。下面是 operator[] 的早期定义：

```
T& operator[](size_t index);
const T& operator[](size_t index) const;
```

使用新的类型别名，这些可以写成如下这样：

```
reference operator[](size_type index);
const_reference operator[](size_type index) const;
```

要求容器提供的方法

除类型别名外，每个容器必须提供表 25-3 所示的方法。

表 25-3 方法

方法	说明	最坏情况下的复杂度
默认构造函数	构造一个空的容器	常量时间复杂度
拷贝构造函数	执行容器的深度复制	线性时间复杂度
移动构造函数	执行移动构造操作	常量时间复杂度
拷贝赋值运算符	执行容器的深度复制	线性时间复杂度
移动赋值运算符	执行移动赋值操作	常量时间复杂度
析构函数	销毁动态分配的内存，对容器中剩余的所有元素调用析构函数	线性时间复杂度

(续表)

方法	说明	最坏情况下的复杂度
iterator begin(); const_iterator begin() const;	返回引用容器中第一个元素的迭代器	常量时间复杂度
iterator end(); const_iterator end() const;	返回引用容器中最后一个元素后面那个位置的迭代器	常量时间复杂度
const_iterator cbegin() const;	返回引用容器中第一个元素的 const 迭代器	常量时间复杂度
const_iterator cend() const;	返回引用容器中最后一个元素后面那个位置的 const 迭代器	常量时间复杂度
operator== operator!=	逐元素比较两个容器的比较运算符	线性时间复杂度
void swap(Container&) noexcept;	对作为参数传入这个方法的容器中的内容, 以及在其中调用这个方法的对象的内容进行交换	常量时间复杂度
size_type size() const;	返回容器中元素的个数	常量时间复杂度
size_type max_size() const;	返回容器可以保存的最大元素数目	常量时间复杂度
bool empty() const;	返回容器是否包含任何元素	常量时间复杂度

如前所述, `directed_graph` 实现遵循零规则(参见第 9 章); 也就是说, 它不需要显式的拷贝/移动构造函数、拷贝/移动赋值运算符或析构函数。

下面的代码片段展示了 `size()`、`max_size()`和 `empty()`方法的声明。关于迭代器相关的方法 `begin()`、`end()`、`cbegin()`和 `cend()`方法将在下一节介绍。

```
export template <typename T>
class directed_graph
{
    public:
        [[nodiscard]] size_type size() const noexcept;
        [[nodiscard]] size_type max_size() const noexcept;
        [[nodiscard]] bool empty() const noexcept;
        // Other methods omitted for brevity.
};
```

这三个方法的实现很简单, 因为它们可以简单地转发到 `m_nodes` 容器中名称类似的方法。注意, `size_type` 是类模板中定义的类型别名之一。因为它是类模板的成员, 所以实现中的这种返回类型必须用 `typename directed_graph<T>`完全限定。

```
template <typename T>
typename directed_graph<T>::size_type directed_graph<T>::size() const noexcept
{
    return m_nodes.size();
}

template <typename T>
typename directed_graph<T>::size_type directed_graph<T>::max_size() const noexcept
```

```

{
    return m_nodes.max_size();
}

template <typename T>
bool directed_graph<T>::empty() const noexcept
{
    return m_nodes.empty();
}

```

`directed_graph` 的当前实现使用 `operator[]` 根据节点索引访问节点。这个运算符，就像 `vector` 中的 `operator[]` 一样，不进行任何边界检查。传入一个越界的索引会使程序崩溃。像 `vector` 一样，`directed_graph` 可以使用 `at()` 方法进行边界检查。如果传递的索引越界，则抛出 `std::out_of_range` 异常。以下是它们的定义：

```

reference at(size_type index);
const_reference at(size_type index) const;

```

实现只是转发到 `m_nodes` 的 `vector`：

```

template <typename T>
typename directed_graph<T>::reference
    directed_graph<T>::at(size_type index)
{
    return m_nodes.at(index).value();
}

template <typename T>
typename directed_graph<T>::const_reference
    directed_graph<T>::at(size_type index) const
{
    return m_nodes.at(index).value();
}

```

编写迭代器

容器最重要的要求是实现迭代器。为了能用于泛型算法，每个容器都必须提供一个能够访问容器中元素的迭代器。迭代器一般应该提供重载的 `operator*` 和 `operator->` 运算符，再加上其他一些取决于特定行为的操作。只要迭代器提供基本的迭代操作，就不会出现问题。

有关迭代器需要做的第一个决策是选择迭代器的类型：正向访问、双向访问或随机访问迭代器。双向迭代器支持似乎是 `directed_graph` 迭代器的一个很好选择。这意味着必须提供 `operator++`、`operator--`、`operator==` 和 `operator!=`。另一种选择是为 `directed_graph` 实现随机访问迭代器，其中包括添加运算符 `+`、`-`、`+=`、`-=`、`<`、`>`、`<=`、`>=` 和 `[]`。这可能是编写迭代器的很好练习。有关不同迭代器的具体要求，请参阅第 17 章。

第二个决策是如何对容器的元素排序。`directed_graph` 是无序的，因此执行有序迭代效率较低。相反，`directed_graph` 迭代器只能按照节点添加到图中的顺序来遍历节点。

第三个决策是选择迭代器的内部表示形式。这个实现通常和容器的内部实现紧密相关。迭代器的最主要作用是引用容器中的元素。对于 `directed_graph`，所有节点都存储在 `m_node` 的 `vector` 中，因此 `directed_graph` 迭代器可能是指向相关元素的 `vector` 迭代器的包装器。

一旦选择好实现方式，就必须为尾迭代器决定一致的表示方式。回顾一下，尾迭代器实际上应该是“越过最后一个元素”的标记，也就是对容器中最后一个元素的迭代器应用 `++` 运算符后得到的迭

代器。directed_graph 迭代器可以使用 m_nodes 的 vector 的尾迭代器作为它的尾迭代器。

最后，容器需要提供 const 迭代器和非 const 迭代器。非 const 迭代器必须能转换为 const 迭代器。这个实现用派生的 directed_graph_iterator 定义了 const_directed_graph_iterator。

const_directed_graph_iterator 类模板

根据前面做出的决策，下面开始定义 const_directed_graph_iterator 类模板。首先要注意的是，每个 const_directed_graph_iterator 对象都是 directed_graph 类的某个实例的迭代器。为提供这种一对一映射，const_directed_graph_iterator 也必须是一个类模板，并把 directed_graph 类型作为模板参数，称为 DirectedGraph。

主要问题在于如何满足双向访问迭代器的要求。任何行为上像迭代器的对象都是迭代器。自定义的迭代器不需要是另一个类的子类去满足双向访问迭代器的要求。然而，如果想让迭代器能适用于泛型算法的函数，就必须指定 iterator_traits。第 17 章已经解释了 iterator_traits 是一个类模板，它为每种迭代器类型定义了 5 个类型别名：value_type、difference_type、iterator_category、pointer 和 reference。如有必要，iterator_traits 类模板可部分特例化以满足新的迭代器类型。另外，iterator_traits 类模板的默认实现从 iterator 类本身获取了 5 个类型别名。因此，可以简单地给迭代器直接定义这些类型别名。const_directed_graph_iterator 是一个双向访问迭代器，因此将 bidirectional_iterator_tag 指定为迭代器类别。其他合法的迭代器类别是 input_iterator_tag、output_iterator_tag、forward_iterator_tag 和 random_access_iterator_tag 和 contiguous_iterator_tag(C++20)。对于 const_directed_graph_iterator，元素类型(value_type)是 typename DirectedGraph::value_type。

注意：

在过去，建议从在 <iterator> 中定义的 std::iterator 类模板中派生自定义迭代器。这个类模板已弃用，不应该继续使用。

这里是 const_directed_graph_iterator 类模板定义：

```
template <typename DirectedGraph>
class const_directed_graph_iterator
{
public:
    using value_type = typename DirectedGraph::value_type;
    using difference_type = ptrdiff_t;
    using iterator_category = std::bidirectional_iterator_tag;
    using pointer = const value_type*;
    using reference = const value_type&;
    using iterator_type =
        typename DirectedGraph::nodes_container_type::const_iterator;

    // Bidirectional iterators must supply a default constructor.
    // Using an iterator constructed with the default constructor
    // is undefined, so it doesn't matter how it's initialized.
    const_directed_graph_iterator() = default;

    // No transfer of ownership of graph.
    const_directed_graph_iterator(iterator_type it,
        const DirectedGraph* graph);

    reference operator*() const;

    // Return type must be something to which -> can be applied.
```

```

// So, return a pointer.
pointer operator->() const;
const_directed_graph_iterator& operator++();
const_directed_graph_iterator operator++(int);

const_directed_graph_iterator& operator--();
const_directed_graph_iterator operator--(int);

// C++20 defaulted operator==.
bool operator==(const const_directed_graph_iterator&) const = default;

protected:
    friend class directed_graph<value_type>;

    iterator_type m_nodeIterator;
    const DirectedGraph* m_graph { nullptr };

    // Helper methods for operator++ and operator--
    void increment();
    void decrement();
};

```

如果感觉重载运算符的定义和实现难以理解，请参阅第 15 章关于运算符重载的详细内容。`const_directed_graph_iterator` 的实现不需要拷贝/移动构造函数和拷贝/移动赋值运算符，因为默认行为就正是我们想要的。该类也不需要显式析构函数，因为默认行为，即不删除 `m_graph`，正是我们想要的。因此，该类也遵循零规则。

`const_directed_graph_iterator` 方法实现

`const_directed_graph_iterator` 构造函数初始化了 2 个数据成员：

```

template <typename DirectedGraph>
const_directed_graph_iterator<DirectedGraph>::const_directed_graph_iterator(
    iterator_type it, const DirectedGraph* graph)
    : m_nodeIterator { it }, m_graph { graph } { }

```

默认构造函数的唯一目的是允许客户声明未初始化的 `const_directed_graph_iterator` 变量。通过默认构造函数构造的迭代器可以不引用任何值，而对这个迭代器进行任何操作都可以产生未定义的行为。

解除引用运算符的实现十分简洁，但也难以理解。第 15 章讲到 `operator*` 和 `operator->` 运算符是不对称的：

- `operator*` 运算符返回的是对底层实际值的引用，在这个例子中即迭代器引用的元素。
- `operator->` 运算符返回的是某个可以再次应用箭头运算符的对象。因此，返回的是指向元素的指针。编译器对这个指针应用 `->` 运算符，从而访问元素中的字段或方法。

```

// Return a reference to the actual element.
template <typename DirectedGraph>
typename const_directed_graph_iterator<DirectedGraph>::reference
const_directed_graph_iterator<DirectedGraph>::operator*() const
{
    return m_nodeIterator->value();
}

// Return a pointer to the actual element, so the compiler can

```



```
// apply -> to it to access the actual desired field.
template <typename DirectedGraph>
typename const_directed_graph_iterator<DirectedGraph>::pointer
    const_directed_graph_iterator<DirectedGraph>::operator->() const
{
    return &(m_nodeIterator->value());
}
```

自增运算符的实现如下。它们将实际的自增操作延迟到 `increment()` 辅助方法中。递减运算符并没有显示出来，因为它们是以类似的方式实现的。

```
// Defer the details to the increment() helper.
template <typename DirectedGraph>
const_directed_graph_iterator<DirectedGraph>&
    const_directed_graph_iterator<DirectedGraph>::operator++()
{
    increment();
    return *this;
}

// Defer the details to the increment() helper.
template <typename DirectedGraph>
const_directed_graph_iterator<DirectedGraph>
    const_directed_graph_iterator<DirectedGraph>::operator++(int)
{
    auto oldIt { *this };
    increment();
    return oldIt;
}
```

`const_directed_graph_iterator` 自增表示指向容器中的下一个元素，自减表示指向容器中的前一个元素。因为 `directed_graph` 实现使用 `vector` 存储它的节点，所以递增和递减都很容易。

```
// Behavior is undefined if m_nodeIterator already refers to the past-the-end
// element, or is otherwise invalid.
template <typename DirectedGraph>
void const_directed_graph_iterator<DirectedGraph>::increment()
{
    ++m_nodeIterator;
}

// Behavior is undefined if m_nodeIterator already refers to the first
// element, or is otherwise invalid.
template <typename DirectedGraph>
void const_directed_graph_iterator<DirectedGraph>::decrement()
{
    --m_nodeIterator;
}
```

并不要求迭代器比普通指针更安全，因此不需要对“递增已经是尾迭代器的迭代器”这类操作执行错误检查。

如果编译器还不支持显式默认的 `operator==(C++20)`，那可以为 `operator==` 和 `operator!` 提供自己的实现，如下所示，它们只是比较对象的两个数据成员：

```
template <typename DirectedGraph>
bool const_directed_graph_iterator<DirectedGraph>::operator==(
```

```

    const const_directed_graph_iterator<DirectedGraph>& rhs) const
{
    // All fields, including the directed_graph to which the iterators refer,
    // must be equal.
    return (m_graph == rhs.m_graph && m_nodeIterator == rhs.m_nodeIterator);
}

template <typename DirectedGraph>
bool const_directed_graph_iterator<DirectedGraph>::operator!=(
    const const_directed_graph_iterator<DirectedGraph>& rhs) const
{
    return !(*this == rhs);
}

```

注意，`const_directed_graph_iterator` 的 `iterator_type` 类型别名使用了 `directed_graph` 的私有 `node_container_type` 类型别名。因此，`directed_graph` 类模板必须声明 `const_directed_graph_iterator` 为友元：

```

export template <typename T>
class directed_graph
{
    // Other methods omitted for brevity.
private:
    friend class const_directed_graph_iterator<directed_graph>;
};

```

directed_graph_iterator 类模板

`directed_graph_iterator` 类模板派生于 `const_directed_graph_iterator`，并且不需要重写 `operator==`、`operator!=`、`increment()` 和 `decrement()`，因为基类版本就足够了。`directed_graph_iterator` 的定义如下。注释被省略，因为它们与基类中的注释相同。下面是与 `const_iterator_graph_iterator` 的主要区别：

- 指针、引用和 `iterator_type` 类型别名是非 `const` 类型。
- `operator*` 和 `operator>` 没有被标记为 `const`。
- `operator++` 和 `operator--` 的返回类型不同。

```

template <typename DirectedGraph>
class directed_graph_iterator : public const_directed_graph_iterator<DirectedGraph>
{
public:
    using value_type = typename DirectedGraph::value_type;
    using difference_type = ptrdiff_t;
    using iterator_category = std::bidirectional_iterator_tag;
    using pointer = value_type*;
    using reference = value_type&;
    using iterator_type =
        typename DirectedGraph::nodes_container_type::iterator;

    directed_graph_iterator() = default;
    directed_graph_iterator(iterator_type it, const DirectedGraph* graph);

    reference operator*();
    pointer operator->();

    directed_graph_iterator& operator++();
    directed_graph_iterator operator++(int);

```

```

        directed_graph_iterator& operator--();
        directed_graph_iterator operator--(int);
};

```

directed_graph_iterator 方法的实现

directed_graph_iterator 方法的实现相当简单。构造函数仅调用基类模板构造函数，operator*和 operator->使用 const_cast 返回非 const 类型，operator++和 operator--只使用基类模板的 increment()和 decrement()，但返回 directed_graph_iterator 而不是 const_directed_graph_iterator。注意 C++的一个奇怪之处是，C++名称查找规则要求显式使用 this->或 const_directed_graph_iterator<DirectedGraph>::来引用基类模板中的数据成员和方法。

```

template <typename DirectedGraph>
directed_graph_iterator<DirectedGraph>::directed_graph_iterator(
    iterator_type it, const DirectedGraph* graph)
    : const_directed_graph_iterator<DirectedGraph> { it, graph } { }

// Return a reference to the actual element.
template <typename DirectedGraph>
typename directed_graph_iterator<DirectedGraph>::reference
    directed_graph_iterator<DirectedGraph>::operator*()
{
    return const_cast<reference>(this->m_nodeIterator->value());
}

// Return a pointer to the actual element, so the compiler can
// apply -> to it to access the actual desired field.
template <typename DirectedGraph>
typename directed_graph_iterator<DirectedGraph>::pointer
    directed_graph_iterator<DirectedGraph>::operator->()
{
    return const_cast<pointer>(&(this->m_nodeIterator->value()));
}

// Defer the details to the increment() helper in the base class.
template <typename DirectedGraph>
directed_graph_iterator<DirectedGraph>&
    directed_graph_iterator<DirectedGraph>::operator++()
{
    this->increment();
    return *this;
}

// Defer the details to the increment() helper in the base class.
template <typename DirectedGraph>
directed_graph_iterator<DirectedGraph>
    directed_graph_iterator<DirectedGraph>::operator++(int)
{
    auto oldIt { *this };
    this->increment();
    return oldIt;
}

// operator-- not shown as the implementation is analogous to operator++.

```

就像 `const_directed_graph_iterator` 一样, `directed_graph` 类模板必须声明 `directed_graph_iterator` 为友元。

迭代器类型别名和访问方法

`directed_graph` 提供迭代器支持的最后一部分内容是在 `directed_graph` 类模板中提供必要的类型别名, 并编写 `begin()`、`end()`、`cbegin()` 和 `cend()` 方法。类型别名和方法原型如下所示:

```
export template <typename T>
class directed_graph
{
    public:
        // Other type aliases omitted for brevity.
        using iterator = const_directed_graph_iterator<directed_graph>;
        using const_iterator = const_directed_graph_iterator<directed_graph>;

        // Iterator methods.
        iterator begin() noexcept;
        iterator end() noexcept;
        const_iterator begin() const noexcept;
        const_iterator end() const noexcept;
        const_iterator cbegin() const noexcept;
        const_iterator cend() const noexcept;
        // Remainder of class definition omitted for brevity.
};
```

注意, `iterator` 和 `const_iterator` 都是 `const_directed_graph_iterator` 的类型别名, 这意味着用户不能修改 `directed_graph` 迭代器引用的值。这遵循与 `std::set` 相同的原则, 在 `std::set` 中也不可以修改元素。这是 `directed_graph` 当前实现的需求, 因为它使用邻接表中的索引。如果将实现修改位允许对节点修改, 则迭代器类型别名可以变为 `directed_graph_iterator`。

`directed_graph` 类模板将所有节点存储在一个简单的 `vector` 中。因此, `begin()` 和 `end()` 可以简单地将它们的工作转发给 `vector` 上同名的方法, 并将这些结果包装在一个迭代器中, 即 `const_directed_graph_iterator` 的类型别名:

```
template <typename T>
typename directed_graph<T>::iterator
    directed_graph<T>::begin() noexcept
{
    return iterator { std::begin(m_nodes), this };
}

template <typename T>
typename directed_graph<T>::iterator
    directed_graph<T>::end() noexcept
{
    return iterator { std::end(m_nodes), this };
}
```

`begin()` 和 `end()` 的 `const` 版本的实现使用了 Scott Meyers 的 `const_cast()` 模式(参见第 9 章)来调用非 `const` 版本:

```
template <typename T>
typename directed_graph<T>::const_iterator
    directed_graph<T>::begin() const noexcept
{

```

```

        return const_cast<directed_graph*>(this)->begin();
    }

    template <typename T>
    typename directed_graph<T>::const_iterator
        directed_graph<T>::end() const noexcept
    {
        return const_cast<directed_graph*>(this)->end();
    }

```

cbegin()和 cend()方法将请求转发到 begin()和 end()的 const 版本:

```

    template <typename T>
    typename directed_graph<T>::const_iterator
        directed_graph<T>::cbegin() const noexcept { return begin(); }

    template <typename T>
    typename directed_graph<T>::const_iterator
        directed_graph<T>::cend() const noexcept { return end(); }

```

修改其他方法以使用迭代器

现在 directed_graph 支持迭代器, 可以稍微修改其他方法来使用迭代器, 以便它们遵循标准库的指导原则。先看看 insert()方法, 在前面的基本实现中, 定义如下:

```

// For an insert to be successful, the value shall not be in the graph yet.
// Returns true if a new node with given value has been added to
// the graph, and false if there was already a node with the given value.
bool insert(const T& node_value);
bool insert(T&& node_value);

```

为了更接近标准库的精神, 可以将它们修改为返回一个 std::pair<iterator, bool>, 其中如果元素被添加到图中, 则布尔值为 true; 如果元素已经在图中, 则为 false。pair 的迭代器指向新添加的元素或已经在图中的元素。

```

std::pair<iterator, bool> insert(const T& node_value);
std::pair<iterator, bool> insert(T&& node_value);

```

实现如下所示, 与返回简单 bool 的版本相比较, 变化部分会被加粗显示。

```

template <typename T>
std::pair<typename directed_graph<T>::iterator, bool>
    directed_graph<T>::insert(const T& node_value)
{
    T copy { node_value };
    return insert(std::move(copy));
}

template <typename T>
std::pair<typename directed_graph<T>::iterator, bool>
    directed_graph<T>::insert(T&& node_value)
{
    auto iter { findNode(node_value) };
    if (iter != std::end(m_nodes)) {
        // Value is already in the graph.
        return { iterator { iter, this }, false };
    }
}

```

```

    m_nodes.emplace_back(this, std::move(node_value));
    // Value successfully added to the graph.
    return { iterator { --std::end(m_nodes), this }, true };
}

```

此外，还提供了接收迭代器提示的 `insert()` 的重载。这个提示对 `directed_graph` 无用，但是提供它是为了与其他标准库容器对称，比如 `std::set`。该提示被忽略，它仅仅调用没有提示的 `insert()` 版本。

```

template <typename T>
typename directed_graph<T>::iterator
    directed_graph<T>::insert(const_iterator hint, const T& node_value)
{
    // Ignore the hint, just forward to another insert().
    return insert(node_value).first;
}

template <typename T>
typename directed_graph<T>::iterator
    directed_graph<T>::insert(const_iterator hint, T&& node_value)
{
    // Ignore the hint, just forward to another insert().
    return insert(std::move(node_value)).first;
}

```

`insert()` 的最后一个版本接收迭代器范围。这个版本是一个方法模板，因此它可以接收来自任何容器的迭代器范围，而不仅仅是其他 `directed_graph` 容器。实际实现使用 `insert_iterator` (第 17 章介绍过)。

```

template <typename T>
template <typename Iter>
void directed_graph<T>::insert(Iter first, Iter last)
{
    // Copy each element in the range by using an insert_iterator adapter.
    // Give begin() as a dummy position -- insert ignores it anyway.
    std::copy(first, last, std::insert_iterator { *this, begin() });
}

```

应该修改 `erase()` 方法来使用迭代器。前面的定义有一个节点值作为参数，并返回一个 `bool` 值：

```

// Returns true if the given node value was erased, false otherwise.
bool erase(const T& node_value);

```

为了遵循标准库原则，`directed_graph` 被修改为提供两个 `erase()` 方法：一个移除迭代器指向的节点，另一个移除给定迭代器范围的节点。两者都返回指向最后一个被移除节点之后的节点的迭代器：

```

// Returns an iterator to the element after the last deleted element.
iterator erase(const_iterator pos);
iterator erase(const_iterator first, const_iterator last);

```

这里是实现：

```

template <typename T>
typename directed_graph<T>::iterator
    directed_graph<T>::erase(const_iterator pos)
{
    if (pos.m_nodeIterator == std::end(m_nodes)) {
        return iterator { std::end(m_nodes), this };
    }
    remove_all_links_to(pos.m_nodeIterator);
}

```

```

        return iterator { m_nodes.erase(pos.m_nodeIterator), this };
    }

template <typename T>
typename directed_graph<T>::iterator
    directed_graph<T>::erase(const_iterator first, const_iterator last)
{
    for (auto iter { first }; iter != last; ++iter) {
        if (iter.m_nodeIterator != std::end(m_nodes)) {
            remove_all_links_to(iter.m_nodeIterator);
        }
    }
    return iterator {
        m_nodes.erase(first.m_nodeIterator, last.m_nodeIterator), this };
}

```

最后，可以实现以下两个公有 `find()` 方法来返回迭代器。该实现留作本章最后的练习。

```

iterator find(const T& node_value);
const_iterator find(const T& node_value) const;

```

使用 `directed_graph` 迭代器

现在 `directed_graph` 支持迭代，下面可以像迭代任何标准库容器一样迭代 `directed_graph` 的元素了，并可将它的迭代器传给方法和函数。下面是一些示例：

```

// Try to insert a duplicate, and use structured bindings for the result.
auto [iter22, inserted] { graph.insert(22) };
if (!inserted) { std::cout << "Duplicate element.\n"; }

// Print nodes using a for loop and iterators.
for (auto iter { graph.cbegin() }; iter != graph.cend(); ++iter) {
    std::cout << *iter << " ";
}
std::cout << std::endl;

// Print nodes using a for loop and iterators retrieved with the non-member
// functions std::cbegin() and std::cend().
for (auto iter { std::cbegin(graph) }; iter != std::cend(graph); ++iter) {
    std::cout << *iter << " ";
}
std::cout << std::endl;

// Print nodes using a range-based for loop.
for (auto& node : graph) { std::cout << node << std::endl; }
std::cout << std::endl;

// Search a node using the std::find() Standard Library algorithm.
auto result { std::find(std::begin(graph), std::end(graph), 22) };
if (result != std::end(graph)) {
    std::cout << "Node 22 found." << std::endl;
} else {
    std::cout << "Node 22 NOT found." << std::endl;
}

// Count all nodes with values > 22.
auto count { std::count_if(std::begin(graph), std::end(graph),

```

```

[] (const auto& node) { return node > 22; }) };

// Use the iterator-based erase() method in combination with std::find().
graph.erase(std::find(std::begin(graph), std::end(graph), 44));

```

这段代码片段还表明，由于对迭代器的支持，标准库算法可以与 `directed_graph` 一起使用。但是由于 `directed_graph` 只支持 `const` 迭代器，因此只支持非修改的标准库算法，就像 `std::set` 一样。例如，下面使用 `remove-erase` 习惯用法的代码片段不能编译：

```

graph.erase(std::remove_if(std::begin(graph), std::end(graph),
    [] (const auto& node) { return node > 22; }), std::end(graph));

```

添加反向迭代器的支持

如果容器提供了双向访问或随机访问迭代器，那么可认为这个容器是可反向的。可反向容器应该提供表 25-4 所示的两个额外的类型别名。

表 25-4 类型别名

类型名称	说明
<code>reverse_iterator</code>	反向遍历容器中元素的类型
<code>const_reverse_iterator</code>	另一个版本的 <code>reverse_iterator</code> ，反向遍历容器中的 <code>const</code> 元素

此外，容器还应提供与 `begin()` 和 `end()` 对应的 `rbegin()` 和 `rend()`；还应该提供 `crbegin()` 和 `crend()`，这两个和 `cbegin()` 与 `cend()` 对应。

`directed_graph` 迭代器是双向的，这意味着它们应该支持反向迭代。下面的代码片段加粗了必要的改动。这两个新的类型别名使用标准库提供的 `std::reverse_iterator` 适配器，并在第 17 章描述，将 `directed_graph` 迭代器转换为反向迭代器。

```

export template <typename T>
class directed_graph
{
public:
    // Other type aliases omitted for brevity.
    using reverse_iterator = std::reverse_iterator<iterator>;
    using const_reverse_iterator = std::reverse_iterator<const_iterator>;

    // Reverse iterator methods.
    reverse_iterator rbegin() noexcept;
    reverse_iterator rend() noexcept;
    const_reverse_iterator rbegin() const noexcept;
    const_reverse_iterator rend() const noexcept;
    const_reverse_iterator crbegin() const noexcept;
    const_reverse_iterator crend() const noexcept;
    // Remainder of class definition omitted for brevity.
};

```

反向迭代器方法的实现如下，注意 `reverse_iterator` 适配器的使用。

```

template <typename T>
typename directed_graph<T>::reverse_iterator
    directed_graph<T>::rbegin() noexcept { return reverse_iterator { end() }; }

template <typename T>

```



```

typename directed_graph<T>::reverse_iterator
    directed_graph<T>::rend() noexcept { return reverse_iterator { begin() }; }

template <typename T>
typename directed_graph<T>::const_reverse_iterator
    directed_graph<T>::rbegin() const noexcept
{ return const_reverse_iterator { end() }; }

template <typename T>
typename directed_graph<T>::const_reverse_iterator
    directed_graph<T>::rend() const noexcept
{ return const_reverse_iterator { begin() }; }

template <typename T>
typename directed_graph<T>::const_reverse_iterator
    directed_graph<T>::crbegin() const noexcept { return rbegin(); }

template <typename T>
typename directed_graph<T>::const_reverse_iterator
    directed_graph<T>::crend() const noexcept { return rend(); }

```

下面的代码片段展示了如何以倒序打印图中的所有节点：

```

for (auto iter { graph.rbegin() }; iter != graph.rend(); ++iter) {
    std::cout << *iter << " ";
}

```

迭代邻接节点

`directed_graph` 保留一个节点的 `vector`，其中每个节点包含节点的值和相邻节点的列表。改进 `directed_graph` 接口，以支持对给定节点的相邻节点进行迭代。首先要添加 `const_adjacent_nodes_iterator` 和 `adjacent_nodes_iterator` 类模板。它们遵循与 `const_directed_graph_iterator` 和 `directed_graph_iterator` 类模板相同的原则，因此不展示代码，可去源代码归档文件中查完整的代码。

下一步是向 `directed_graph` 接口添加新的类型别名和方法，以访问相邻节点迭代器：

```

export template <typename T>
class directed_graph
{
public:
    // Other type aliases omitted for brevity.
    using iterator_adjacent_nodes = adjacent_nodes_iterator<directed_graph>;
    using const_iterator_adjacent_nodes =
        const_adjacent_nodes_iterator<directed_graph>;
    using reverse_iterator_adjacent_nodes =
        std::reverse_iterator<iterator_adjacent_nodes>;
    using const_reverse_iterator_adjacent_nodes =
        std::reverse_iterator<const_iterator_adjacent_nodes>;

    // Return iterators to the list of adjacent nodes for a given node.
    // Return a default constructed one as end iterator if value is not found.
    iterator_adjacent_nodes begin(const T& node_value) noexcept;
    iterator_adjacent_nodes end(const T& node_value) noexcept;
    const_iterator_adjacent_nodes begin(const T& node_value) const noexcept;
    const_iterator_adjacent_nodes end(const T& node_value) const noexcept;
    const_iterator_adjacent_nodes cbegin(const T& node_value) const noexcept;

```

```

const_iterator_adjacent_nodes cend(const T& node_value) const noexcept;

// Return reverse iterators to the list of adjacent nodes for a given node.
// Return a default constructed one as end iterator if value is not found.
reverse_iterator_adjacent_nodes rbegin(const T& node_value) noexcept;
reverse_iterator_adjacent_nodes rend(const T& node_value) noexcept;
const_reverse_iterator_adjacent_nodes rbegin(const T& node_value)
    const noexcept;
const_reverse_iterator_adjacent_nodes rend(const T& node_value)
    const noexcept;
const_reverse_iterator_adjacent_nodes crbegin(const T& node_value)
    const noexcept;
const_reverse_iterator_adjacent_nodes crend(const T& node_value)
    const noexcept;
// Remainder of class definition omitted for brevity.
};

```

下面是 `begin(const T&)` 方法的实现:

```

template <typename T>
typename directed_graph<T>::iterator_adjacent_nodes
    directed_graph<T>::begin(const T& node_value) noexcept
{
    auto iter { findNode(node_value) };
    if (iter == std::end(m_nodes)) {
        // Return a default constructed end iterator.
        return iterator_adjacent_nodes {};
    }
    return iterator_adjacent_nodes {
        std::begin(iter->get_adjacent_nodes_indices()), this };
}

```

相邻节点迭代器相关的其他方法要么类似于 `begin(const T&)` 的实现, 要么转发它们的工作到其他方法。可去源代码归档文件中查完整的代码。

有了对这些相邻节点迭代器的支持, 访问给定节点的所有相邻节点就很简单。下面是打印节点值为 22 的所有相邻节点的示例。记住, 默认构造的相邻节点迭代器用作结束迭代器。

```

std::cout << "Adjacency list for node 22: ";

auto iterBegin { graph.cbegin(22) };
auto iterEnd { graph.cend(22) };

if (iterBegin == directed_graph<int>::const_iterator_adjacent_nodes{}) {
    std::cout << "Value 22 not found." << std::endl;
} else {
    for (auto iter { iterBegin }; iter != iterEnd; ++iter) {
        std::cout << *iter << " ";
    }
}

```

打印图

既然 `directed_graph` 支持节点及其相邻节点的迭代器, 那么用于打印图的 `to_dot()` 辅助函数模板可以改为使用迭代器:

```

export template <typename T>
std::wstring to_dot(const directed_graph<T>& graph, std::wstring_view graph_name)

```

```

{
    std::wstringstream wss;
    wss << format(L"digraph {} {}", graph_name.data()) << std::endl;
    for (auto&& node : graph) {
        const auto b { graph.cbegin(node) };
        const auto e { graph.cend(node) };
        if (b == e) {
            wss << node << std::endl;
        } else {
            for (auto iter { b }; iter != e; ++iter) {
                wss << format(L"{} -> {}", node, *iter) << std::endl;
            }
        }
    }
    wss << "}" << std::endl;
    return wss.str();
}

```

25.2.5 添加分配器支持

根据本章前面的描述，所有标准库容器都允许指定自定义的内存分配器。`directed_graph` 的实现也应该一样。这里介绍需要做哪些更改来支持自定义分配器。分配器支持将分两个阶段实现，首先是 `native` 版本，然后是适当的最终版本。

图节点的更改

需要修改的第一个类模板是 `graph_node`。需要做出以下改变：

- 第二个模板类型参数指定要使用的分配器类型。
- 两个新构造函数传给特定分配器。
- 拷贝/移动构造函数和拷贝/移动赋值运算符，因为默认值不适合使用自定义分配器。
- 需要修改构造函数，以便使用给定分配器来为节点进行分配。
- 需要析构函数来为使用给定分配器的节点释放内存。
- 需要一个数据成员来存储分配器。
- `m_data` 现在应该是一个指针而不是一个值。
- 运算符——不能再用默认的，所以需要使用运算符=和运算符!=的显式实现。

下面是 `graph_node` 类模板的定义，其中改动部分被加粗显示：

```

template <typename T , typename A = std::allocator<T> >
class graph_node
{
public:
    // Constructs a graph_node for the given value.
    graph_node(directed_graph<T , A >* graph, const T& t);
    graph_node(directed_graph<T , A >* graph, T&& t);

    // Constructs a graph_node for the given value and with given allocator.
    graph_node(directed_graph<T , A>* graph, const T& t, const A& allocator);
    graph_node(directed_graph<T , A>* graph, T&& t, const A& allocator);

    ~graph_node();

    // Copy and move constructors.

```

```

graph_node(const graph_node& src);
graph_node(graph_node&& src) noexcept;

// Copy and move assignment operators.
graph_node& operator=(const graph_node& rhs);
graph_node& operator=(graph_node&& rhs) noexcept;

// Remainder of public part not shown, remains unchanged.
private:
    friend class directed_graph<T, A>;

    // A pointer to the graph this node is in.
    directed_graph<T, A>* m_graph;
    // ... unchanged part omitted ...
    A m_allocator;
    T* m_data{ nullptr };
    adjacency_list_type m_adjacentNodeIndices;
};

```

接收分配器的构造函数通过调用 `allocate()` 来使用该分配器为新节点分配足够的内存。这只分配了足够的内存来保存类型为 `T` 的实例，它不构造 `T`。`T` 的构造发生在 `placement new` 运算符中。正如第 15 章解释的一样，使用 `placement new` 运算符，可以在一个预先分配的内存块中构造一个对象。

```

template <typename T, typename A>
graph_node<T, A>::graph_node(directed_graph<T, A>* graph, const T& t,
    const A& allocator) : m_graph { graph }, m_allocator { allocator }
{
    m_data = m_allocator.allocate(1);
    new(m_data) T { t };
}

template <typename T, typename A>
graph_node<T, A>::graph_node(directed_graph<T, A>* graph, T&& t,
    const A& allocator) : m_graph { graph }, m_allocator { allocator }
{
    m_data = m_allocator.allocate(1);
    new(m_data) T { std::move(t) };
}

```

没有 `allocator` 参数的构造函数通过传入一个默认已构造的分配器，将其工作转发给带有 `allocator` 参数的构造函数：

```

template <typename T, typename A>
graph_node<T, A>::graph_node(directed_graph<T, A>* graph, const T& t)
    : graph_node<T, A> { graph, t, A{} } {}

template <typename T, typename A>
graph_node<T, A>::graph_node(directed_graph<T, A>* graph, T&& t)
    : graph_node<T, A> { graph, std::move(t), A{} } {}

```

析构函数显式地调用 `T` 的析构函数来销毁 `T` 实例，使用分配器释放内存块：

```

template <typename T, typename A>
graph_node<T, A>::~~graph_node()
{
    if (m_data) {
        m_data->~T();
    }
}

```

```

        m_allocator.deallocate(m_data, 1);
        m_data = nullptr;
    }
}

```

拷贝/移动构造函数和拷贝/移动赋值运算符留给读者作为练习。可去源代码归档文件中查完整的代码。

最后，`value()`方法和运算符`==`必须解引用 `m_data`，因为它现在是一个指针。

有向图的更改

`directed_graph` 类模板定义需要添加以下内容：

- 第二个模板类型参数指定要使用的分配器类型。
- `allocator_type` 类型别名。
- 默认构造函数和接收要使用的特定分配器的构造函数。
- 需要一个数据成员来存储分配器。

这里是定义，改动部分被加粗显示：

```

export template <typename T , typename A = std::allocator<T> >
class directed_graph
{
public:
    using allocator_type = A;

    directed_graph() noexcept(noexcept(A{})) = default;
    explicit directed_graph(const A& allocator) noexcept;
    // Remainder of public part not shown, remains unchanged.
private:
    friend class details::graph_node<T , A >;

    using nodes_container_type = std::vector<details::graph_node<T , A >>;

    nodes_container_type m_nodes;
    A m_allocator;
    // ... unchanged part omitted ...
    [[nodiscard]] std::set<T , std::less<T> , A > get_adjacent_nodes_values(
        const typename details::graph_node<T , A >::adjacency_list_type&
        indices) const;
};

```

默认构造函数是默认的，使用了稍微复杂的语法。基本上，这种语法将默认构造函数标记为 `noexcept`，但只有在分配器的默认构造函数为 `noexcept` 时才会如此。

所有的方法实现都需要使用 `template<typename T, typename A>` 而不是 `template<typename T>`，以及 `<T, A>` 而不是 `<T>`。

新的构造函数很简单：

```

template <typename T, typename A>
directed_graph<T, A>::directed_graph(const A& allocator) noexcept
    : m_nodes { allocator }, m_allocator { allocator } { }

```

在 `insert(T&&)` 方法中对 `emplace_back()` 的调用应该修改如下：

```

m_nodes.emplace_back(this, std::move(node_value) , m_allocator );

```

`get_adjacent_nodes_values()` 方法中的 `values` 变量的定义应修改如下。但是，因为 `allocator` 是第三

个模板类型参数，所以必须显式地提到 `std::less`。

```
std::set<T, std::less<>, A> values (m_allocator);
```

`swap()` 方法应该修改如下：

```
template <typename T, typename A>
void directed_graph<T, A>::swap(directed_graph& other_graph) noexcept
{
    using std::swap;
    m_nodes.swap(other_graph.m_nodes);
    swap(m_allocator, other_graph.m_allocator);
}
```

这些都是 `directed_graph` 类所需的改动。

异常安全

遗憾的是，上面的 `naïve` 版本有个问题。看下面的 `graph_node` 构造函数：

```
template <typename T, typename A>
graph_node<T, A>::graph_node(directed_graph<T, A>* graph, const T& t,
    const A& allocator) : m_graph { graph }, m_allocator { allocator }
{
    m_data = m_allocator.allocate(1);
    new(m_data) T { t };
}
```

这段代码有什么问题呢？这很难发现，但这段代码不是异常安全的。如果 `m_data` 被正确分配，但是 `T` 的构造函数抛出了一个异常，那么 `graph_node` 的析构函数将永远不会被调用，就会泄漏 `m_data`。

这个异常安全问题的一个可能解决方案是在基类中分配，在派生类中进行构造。基类 `graph_node_allocator` 只负责分配存储 `T` 的实例的内存。原始 `graph_node` 的 `m_allocator` 和 `m_data` 成员被推到这个新的基类中。这个基类不需要拷贝构造函数和拷贝/移动赋值运算符，因此将它们删除。下面是基类的整个代码。重要的地方被加粗显示：内存的分配和重新分配。还要注意在 `move` 构造函数中使用 `std::exchange()` 来移动 `m_data`。

```
template <typename T, typename A = std::allocator<T>>
class graph_node_allocator
{
protected:
    explicit graph_node_allocator(const A& allocator);

    // Copy and move constructors.
    graph_node_allocator(const graph_node_allocator&) = delete;
    graph_node_allocator(graph_node_allocator&& src) noexcept;

    // Copy and move assignment operators.
    graph_node_allocator& operator=(const graph_node_allocator&) = delete;
    graph_node_allocator& operator=(graph_node_allocator&&) noexcept = delete;

    ~graph_node_allocator();

    A m_allocator;
    T* m_data{ nullptr };
};
```

```

template <typename T, typename A>
graph_node_allocator<T, A>::graph_node_allocator(const A& allocator)
    : m_allocator { allocator }
{ m_data = m_allocator.allocate(1); }

template <typename T, typename A>
graph_node_allocator<T, A>::graph_node_allocator(graph_node_allocator&& src)
    noexcept
    : m_allocator { std::move(src.m_allocator) }
    , m_data { std::exchange(src.m_data, nullptr) } { }

template <typename T, typename A>
graph_node_allocator<T, A>::~graph_node_allocator()
{
    m_allocator.deallocate(m_data, 1);
    m_data = nullptr;
}

```

接下来，将 `graph_node` 类模板修改为从 `graph_node_allocator` 私有派生：

```

template <typename T, typename A = std::allocator<T>>
class graph_node : private graph_node_allocator<T, A> { /* ... */ };

```

除了这个变化，`m_allocator` 和 `m_data` 成员应该被删除，因为它们被移到了基类中。`graph_node` 的构造函数需要进行适配，以正确地调用基类中的构造函数。这里有个示例：

```

template <typename T, typename A>
graph_node<T, A>::graph_node(directed_graph<T, A>* graph, const T& t,
    const A& allocator)
    : m_graph { graph }, graph_node_allocator<T, A> { allocator }
{ new(this->m_data) T { t }; }

```

现在，假设在基类中分配成功，但是 `graph_node` 构造函数中调用的 `T` 构造函数抛出一个异常。有了这个新的实现，所有东西都会被很好地释放，因为 C++ 保证对每个完全构造的对象(在本例中是基类对象)调用析构函数。

对于接收 `T&&` 的构造函数和拷贝/移动构造函数的实现也需要进行类似的更改，因此这里不进行展示。

最后还需要修改析构函数，因为释放自己发生在基类析构函数中。因此，`graph_node` 析构函数应该只销毁 `m_data`，而不是释放它：

```

template <typename T, typename A>
graph_node<T, A>::~graph_node()
{
    if (this->m_data) { this->m_data->~T(); }
}

```

25.2.6 改善 `graph_node`

根据本章前面描述，`graph_node` 的当前实现对 `m_graph` 数据成员使用了一个指针。这可以改为引用，但如前所述，这涉及添加拷贝/移动构造函数和拷贝/移动赋值运算符，因为当类中有引用数据成员时，这些编译器生成的版本会自动删除。不过好消息是：具有自定义分配器支持的 `graph_node` 已经有了拷贝/移动构造函数和拷贝/移动赋值运算符，这意味着切换到 `m_graph` 的引用并不需要做太多工作。下面是对 `graph_node` 接口的必要改动：

```

template <typename T, typename A = std::allocator<T>>
class graph_node : private graph_node_allocator<T, A>
{
public:
    // Constructs a graph_node for the given value.
    graph_node( directed_graph<T, A>& graph , const T& t);
    graph_node( directed_graph<T, A>& graph , T&& t);

    // Constructs a graph_node for the given value and with given allocator.
    graph_node( directed_graph<T, A>& graph , const T& t, const A& allocator);
    graph_node( directed_graph<T, A>& graph , T&& t, const A& allocator);
    // Remainder of public part not shown, remains unchanged.

private:
    // A reference to the graph this node is in.
    directed_graph<T, A>& m_graph;
    // ... unchanged part omitted ...
};

```

这 4 个构造函数、拷贝/移动构造函数和拷贝/移动赋值运算符的实现需要稍做更改，但这些改动很简单，因此在本文中没有详细说明。可在源代码归档文件中查看最终代码。

此外，在 `directed_graph` 类模板中有一个方法需要稍微修改。`insert(T&&)`方法之前包含以下代码行：

```
m_nodes.emplace_back(this, std::move(node_value), m_allocator);
```

需要替换为以下内容：

```
m_nodes.emplace_back(*this , std::move(node_value), m_allocator);
```

25.2.7 附加的标准库类似功能

可以向 `directed_graph` 类模板添加更多类似标准库的特性。首先像 `vector` 一样添加 `assign()`方法。接收迭代器范围的 `assign()`方法也是一个方法模板，就像本章前面讨论的基于迭代器的 `insert()`方法一样：

```

template <typename Iter>
void assign(Iter first, Iter last);

void assign(std::initializer_list<T> init);

```

这些函数允许将给定迭代器范围或 `initializer_list`(见第 1 章)中的所有元素赋值给有向图。赋值意味着清空当前图，并插入新的节点。尽管有点语法，但实现很简单：

```

template <typename T, typename A>
template <typename Iter>
void directed_graph<T, A>::assign(Iter first, Iter last)
{
    clear();
    for (auto iter { first }; iter != last; ++iter) { insert(*iter); }
}

template <typename T, typename A>
void directed_graph<T, A>::assign(std::initializer_list<T> init)
{

```



```
    assign(std::begin(init), std::end(init));
}
```

还提供了 `insert()` 的 `initializer_list` 重载:

```
template <typename T, typename A>
void directed_graph<T, A>::insert(std::initializer_list<T> init)
{
    insert(std::begin(init), std::end(init));
}
```

有了 `insert()` 的重载, 可以按如下方式添加节点:

```
graph.insert({ 66, 77, 88 });
```

下一步, 可以添加 `initializer_list` 构造函数和赋值运算符:

```
template <typename T, typename A>
directed_graph<T, A>::directed_graph(std::initializer_list<T> init,
    const A& allocator) : m_allocator { allocator }
{
    assign(std::begin(init), std::end(init));
}

template <typename T, typename A>
directed_graph<T, A>& directed_graph<T, A>::operator=(
    std::initializer_list<T> init)
{
    // Use a copy-and-swap-like algorithm to guarantee strong exception safety.
    // Do all the work in a temporary instance.
    directed_graph new_graph { init };
    swap(new_graph); // Commit the work with only non-throwing operations.
    return *this;
}
```

有了这些, 就可以使用统一初始化来构造 `directed_graph`, 如下所示:

```
directed_graph<int> graph { 11, 22, 33 };
```

而不是下面这样:

```
directed_graph<int> graph;
graph.insert(11);
graph.insert(22);
graph.insert(33);
```

可以给图如下赋值:

```
graph = { 66, 77, 88 };
```

由于 `initializer_list` 构造函数和类模板参数推导(CTAD), 甚至可以在构造 `directed_graph` 时去掉元素类型, 就像 `vector` 一样:

```
directed_graph graph { 11, 22, 33 };
```

也可以添加接收元素迭代器范围的构造函数。这也是一个方法模板, 类似于接收迭代器范围的 `assign()` 方法。该实现只是将工作转发给 `assign()`:

```
template <typename T, typename A>
```

```
template <typename Iter>
directed_graph<T, A>::directed_graph(Iter first, Iter last, const A& allocator)
    : m_allocator { allocator }
{
    assign(first, last);
}
```

最后，可以添加接收节点值的 `erase()` 的额外重载。类似于 `std::set`，它返回被删除的节点个数，对于 `directed_graph`，它总是 0 或 1。

```
template <typename T, typename A>
typename directed_graph<T, A>::size_type directed_graph<T, A>::erase(
    const T& node_value)
{
    const auto iter { findNode(node_value) };
    if (iter != std::end(m_nodes)) {
        remove_all_links_to(iter);
        m_nodes.erase(iter);
        return 1;
    }
    return 0;
}
```

emplace 操作

`emplace` 操作在适当的地方构造对象(在第 18 章讨论)。`directed_graph` 的 `emplace` 方法如下：

```
template <typename... Args>
std::pair<iterator, bool> emplace(Args&&... args);

template <typename... Args>
iterator emplace_hint(const_iterator hint, Args&&... args);
```

这些行里的...不是打字错误。这些就是所谓的可变函数模板，即具有可变数量的模板类型参数和可变数量的函数参数的函数模板。可变参数模板会在第 26 章“高级模板”中讨论。这个 `directed_graph` 的实现忽略了 `emplace` 操作。

25.2.8 进一步改善

可以对 `directed_graph` 类模板做些改进。这里有些：

- 由于在邻接表中使用节点索引，当前的实现不允许对节点进行更改。可以对代码进行改善来允许这样的修改。这样，`directed_graph<T>::iterator` 类型别名就可以变成一个 `non-const` 迭代器。
- 当前实现不检查图中的循环。通过添加这样的检查，图变成有向无环图。
- 可以改进代码来支持随机访问迭代器，而不是支持双向迭代器。
- 参见第 18 章，C++17 为标准库关联容器添加了节点相关的功能。`directed_graph` 类模板可以修改为包含 `node_type` 类型别名和 `extract()` 等方法。

25.2.9 其他容器类型

`directed_graph` 类模板基本上是一个顺序容器，但由于图的本质，它确实从关联容器中实现了某些功能，比如 `insert()` 方法的返回类型。

可以编写纯顺序容器、无序关联容器和有序关联容器。在这种情况下，需要遵循标准库强制要求的一组特定需求。比如在这里列出它们，更容易指出的是，`deque` 容器几乎完全遵循指定的顺序容器要求。唯一的区别是它提供了一个额外的 `resize()` 方法(标准不要求)。有序关联容器的一个示例是 `map`，可以在其上建模自己的有序关联容器。`unordered_map` 是无序关联容器的一个示例。

25.3 本章小结

本章介绍了分配器的概念，它允许自定义如何为容器分配和释放内存。还展示了如何编写自己的算法来处理标准库容器的数据。最后，本章的主要部分展示了与标准库兼容的 `directed_graph` 容器的几乎完整的开发。由于其对迭代器的支持，`directed_graph` 与标准库算法兼容。

在阅读本章内容的过程中，应该理解开发容器所需的步骤。即使从来都没有编写过标准库算法或容器，也应该能更好地理解标准库的精髓和功能，并更好地运用这些知识。

这是有关标准库的最后一章。尽管本书提到了很多细节，但仍略去了很多特性。如果对这些内容感兴趣，可参阅附录 B 中的资源以找到更多信息。不要强迫自己使用这里讨论的所有特性。如果强迫自己在程序中使用不是真正需要的特性，只会让代码更加复杂。不过，我们鼓励你在必要之处考虑采用标准库中的特性。从容器开始，可能再使用几个算法，你将不知不觉地开始依赖标准库。

25.4 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

练习 25-1 编写一个名为 `transform_if()` 的算法，类似于第 20 章讨论的标准库中的 `transform()`。区别在于 `transform_if()` 应该接收一个额外的谓词，并且它只转换谓词返回 `true` 的元素。其他元素保持不变。要测试你的算法，可创建一个整数的 `vector` 并将 `vector` 中的所有奇数值乘 2。

练习 25-2 编写一个名为 `generate_fibonacci()` 的算法，它用斐波那契数列填充给定的范围。斐波那契数列从 0 和 1 开始，任何后面的值都是前面两个值的和。所以有：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 等。你的实现不允许包含任何手写的循环或者使用递归算法来实现。相反，应该使用标准库的 `generate()` 算法来完成大部分工作。

练习 25-3 为 `directed_graph` 类模板实现 `find(const T&)` 方法的 `const` 和非 `const` 重载版本。

练习 25-4 C++20 为所有关联容器添加了一个 `contains()` 方法，如果给定的元素在容器中，则返回 `true`，否则返回 `false`。因为这对 `directed_graph` 也很有用，所以将 `contains()` 的实现添加到 `directed_graph` 中。

第26章

高级模板

本章内容

- 不同类型的模板参数
- 如何使用局部特例化
- 如何编写递归模板
- 可变参数模板的含义
- 如何使用可变参数模板编写类型安全的可变参数的函数
- `constexpr if` 语句的含义
- 如何使用折叠表达式
- 模板元编程的含义和用法
- 可供使用的类型 `trait`

第 12 章讨论了类和函数模板中使用最广泛的功能。如果只是为了更好地了解标准库的工作方式或只是编写简单的类和函数模板，或者只是对模板的基础知识感兴趣，那么可以跳过本章。然而，如果对模板感兴趣，想释放模板的全部力量，那么请继续阅读本章，了解一些比较难懂但十分迷人的细节。

26.1 深入了解模板参数

实际上有 3 种模板参数：类型参数、非类型参数和 `template template` 参数(这里没有重复，确实就是这个名称)。第 12 章曾列举类型参数和非类型参数的例子，但没有见过 `template template` 参数。本章也有一些第 12 章没有涉及的有关类型参数和非类型参数的棘手问题。下面深入探讨这三类模板参数。

26.1.1 深入了解模板类型参数

模板的类型参数是模板的精髓。可声明任意数目的类型参数。例如，可给第 12 章的网格模板添加第二个类型参数，以表示这个网格构建于另一个模板化的类容器之上。标准库定义了几个模板化的容器类，包括 `vector` 和 `deque`。原始的网格类使用 `vector` 的 `vector` 存储网格的元素，`Grid` 类的用户可

能想使用 deque 的 vector。通过另一个模板的类型参数,可以让用户指定底层容器是 vector 还是 deque。该实现使用容器的 `resize()` 方法和容器的 `value_type` 类型别名。因此使用 `concept` 强制给定的容器支持这些。下面是带有额外模板参数的 `concept` 和类定义,加粗了变化部分:

```
template <typename Container>
concept ContainerType = requires(Container c)
{
    c.resize(1);
    typename Container::value_type;
};

export template <typename T, ContainerType Container>
class Grid
{
public:
    explicit Grid(size_t width = DefaultWidth, size_t height = DefaultHeight);
    virtual ~Grid() = default;

    // Explicitly default a copy constructor and assignment operator.
    Grid(const Grid& src) = default;
    Grid& operator=(const Grid& rhs) = default;

    // Explicitly default a move constructor and assignment operator.
    Grid(Grid&& src) = default;
    Grid& operator=(Grid&& rhs) = default;

    typename Container::value_type& at(size_t x, size_t y);
    const typename Container::value_type& at(size_t x, size_t y) const;

    size_t getHeight() const { return m_height; }
    size_t getWidth() const { return m_width; }

    static const size_t DefaultWidth { 10 };
    static const size_t DefaultHeight { 10 };

private:
    void verifyCoordinate(size_t x, size_t y) const;

    std::vector<Container> m_cells;
    size_t m_width { 0 }, m_height { 0 };
};
```

现在这个模板有两个参数: `T` 和 `Container`。因此,所有引用了 `Grid<T>` 的地方现在都必须指定 `Grid<T, Container>` 以表示两个模板参数。注意 `m_cells` 现在是 `Container` 的 `vector`, 而不是 `vector` 的 `vector`。 `at()` 方法的返回类型是存储在给定容器类型中元素的类型。可以通过 `typename Container::value_type` 来访问这个类型。

下面是构造函数的定义:

```
template <typename T, ContainerType Container>
Grid<T, Container>::Grid(size_t width, size_t height)
    : m_width { width }, m_height { height }
{
    m_cells.resize(m_width);
    for (auto& column : m_cells) {
        column.resize(m_height);
    }
}
```

```
    }
}
```

下面是其余方法的实现:

```
template <typename T, ContainerType Container>
void Grid<T, Container>::verifyCoordinate(size_t x, size_t y) const
{
    if (x >= m_width) {
        throw std::out_of_range {
            std::format("{} must be less than {}. ", x, m_width) };
    }
    if (y >= m_height) {
        throw std::out_of_range {
            std::format("{} must be less than {}. ", y, m_height) };
    }
}

template <typename T, ContainerType Container>
const typename Container::value_type&
Grid<T, Container>::at(size_t x, size_t y) const
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}

template <typename T, ContainerType Container>
typename Container::value_type&
Grid<T, Container>::at(size_t x, size_t y)
{
    return const_cast<typename Container::value_type&>(
        std::as_const(*this).at(x, y));
}
```

现在, 可按以下方式实例化和使用 Grid 对象:

```
Grid<int, vector<optional<int>>> myIntVectorGrid;
Grid<int, deque<optional<int>>> myIntDequeGrid;

myIntVectorGrid.at(3, 4) = 5;
cout << myIntVectorGrid.at(3, 4).value_or(0) << endl;

myIntDequeGrid.at(1, 2) = 3;
cout << myIntDequeGrid.at(1, 2).value_or(0) << endl;

Grid<int, vector<optional<int>>> grid2 { myIntVectorGrid };
grid2 = myIntVectorGrid;
```

可尝试用 double 作为容器模板类型参数来实例化 Grid 类模板:

```
Grid<int, double> test; // WILL NOT COMPILE
```

此行代码无法成功编译, 编译器会抱怨 double 类型不满足容器模板类型参数相关 concept 的约束。

与函数参数一样, 可给模板参数指定默认值。例如, 可能想表示 Grid 的默认容器是 vector。这个模板类定义如下所示:

```
export template <typename T,
```

```

        ContainerType Container = std::vector<std::optional<T>>>
class Grid
{
    // Everything else is the same as before.
};

```

可以使用第一个模板参数中的类型 `T` 作为第二个模板参数的默认值中 `optional` 模板的参数。C++ 语法要求不能在方法定义的模板标题行中重复默认值。现在有了这个默认参数后，实例化网格时，客户可指定或不指定底层容器。下面有一些示例：

```

Grid<int, deque<optional<int>>> myDequeGrid;
Grid<int, vector<optional<int>>> myVectorGrid;
Grid<int> myVectorGrid2 { myVectorGrid };

```

`stack`、`queue` 和 `priority_queue` 类模板都使用 `Container` 模板类型参数，包含默认值，并指定底层容器。

26.1.2 template template 参数介绍

26.1.1 节讨论的 `Container` 参数还存在一个问题。当实例化类模板时，这样编写代码：

```
Grid<int, vector<optional<int>>> myIntGrid;
```

请注意 `int` 类型的重复。必须在 `vector` 中同时为 `Grid` 和 `vector` 指定元素类型。如果编写了下面这样的代码，会怎样？

```
Grid<int, vector<optional<SpreadsheetCell>>> myIntGrid;
```

这不能很好地工作。如果能编写以下代码就好了，这样就不会出现此类错误：

```
Grid<int, vector> myIntGrid;
```

`Grid` 类应该能够判断出需要一个元素类型为 `int` 的 `optional vector`。不过编译器不会允许传递这样的参数给普通的类型参数，因为 `vector` 本身并不是类型，而是模板。

如果想要接收模板作为模板参数，那么必须使用一种特殊参数，称为 `template template` 参数。指定 `template template` 参数，有点像在普通函数中指定函数指针参数。函数指针的类型包括函数的返回类型和参数类型。同样，指定 `template template` 参数时，`template template` 参数的完整规范包括该模板的参数。

例如，`vector` 和 `deque` 等容器有一个模板参数列表，如下所示。`E` 参数是元素类型，`Allocator` 参数参见第 25 章“自定义和扩展标准库”。

```

template <typename E, typename Allocator = std::allocator<E>>
class vector { /* Vector definition */ };

```

要把这样的容器传递为 `template template` 参数，只能复制并粘贴类模板的声明(在本例中是 `template <typename E, typename Allocator = std::allocator<E>> class vector`)，用参数名(`Container`)替代类名(`vector`)，并把它用作另一个模板声明的 `template template` 参数(本例中的 `Grid`)，而不是简单的类型名。有了前面的模板规范，下面是接收一个容器模板作为第二个模板参数的 `Grid` 类的类模板定义：

```

export template <typename T,
    template <typename E, typename Allocator = std::allocator<E>> class Container
    = std::vector>
class Grid
{

```

```

public:
    // Omitted code that is the same as before.
    std::optional<T>& at(size_t x, size_t y);
    const std::optional<T>& at(size_t x, size_t y) const;
    // Omitted code that is the same as before.
private:
    void verifyCoordinate(size_t x, size_t y) const;

    std::vector<Container<std::optional<T>>> m_cells;
    size_t m_width { 0 }, m_height { 0 };
};

```

这里是怎么回事？第一个模板参数与以前一样：元素类型 `T`。第二个模板参数现在本身就是容器的模板，如 `vector` 或 `deque`。如前所述，这种“模板类型”必须接收两个参数：元素类型 `E` 和分配器类型。注意嵌套模板参数列表后面重复的单词 `class`。这个参数在 `Grid` 模板中的名称是 `Container`。默认值现为 `vector` 而不是 `vector<T>`，因为 `Container` 是模板而不是实际类型。

template template 参数更通用的语法规则是：

```
template <..., template <TemplateTypeParams> class ParameterName, ...>
```

注意：

从 C++17 开始，也可以用 `typename` 关键字替代 `class`，如下所示：

```
template <..., template <Params> typename ParameterName, ...>
```

在代码中不要使用 `Container` 本身，而必须把 `Container<std::optional<T>>` 指定为容器类型。例如，现在 `m_cells` 的声明如下：

```
std::vector<Container<std::optional<T>>> m_cells;
```

不需要更改方法定义，但必须更改模板行，例如：

```

template <typename T,
    template <typename E, typename Allocator = std::allocator<E>> class Container>
void Grid<T, Container>::verifyCoordinate(size_t x, size_t y) const
{
    // Same implementation as before...
}

```

可以这样使用 `Grid` 模板：

```

Grid<int, vector> myGrid;
myGrid.at(1, 2) = 3;
cout << myGrid.at(1, 2).value_or(0) << endl;
Grid<int, vector> myGrid2 { myGrid };
Grid<int, deque> myDequeGrid;

```

上述 C++ 语法有点令人费解，因为它试图获得最大的灵活性。尽量不要在这里陷入语法困境，记住主要概念：可向其他模板传入模板作为参数。

26.1.3 深入了解非类型模板参数

有时可能想让用户指定一个默认元素，用来初始化网格中的每个单元格。下面是实现这个目标的

一种完全合理的方法，它使用零初始化语法 `T{}`，作为第二个模板参数的默认值：

```
export template <typename T, const T DEFAULT = T{}>
class Grid
{
    // Identical as before.
};
```

这个定义是合法的。可使用第一个参数中的类型 `T` 作为第二个参数的类型，非类型参数可为 `const`，就像函数参数一样。可使用 `T` 的初始值来初始化网格中的每个单元格：

```
template <typename T, const T DEFAULT>
Grid<T, DEFAULT>::Grid(size_t width, size_t height)
: m_width { width }, m_height { height }
{
    m_cells.resize(m_width);
    for (auto& column : mCells) {
        column.resize(m_height);
        for (auto& element : column) {
            element = DEFAULT;
        }
    }
}
```

其他的方法定义保持不变，只是必须向模板行添加第二个模板参数，所有 `Grid<T>` 实例要变为 `Grid<T, DEFAULT>`。完成这些修改后，可实例化一个 `int` 网格，并为所有元素设置初始值：

```
Grid<int> myIntGrid;           // Initial value is 0
Grid<int, 10> myIntGrid2;     // Initial value is 10
```

初始值可以是任何整数。但是，假设尝试创建一个 `SpreadsheetCell` 网格：

```
SpreadsheetCell defaultCell;
Grid<SpreadsheetCell, defaultCell> mySpreadsheet; // WILL NOT COMPILE
```

这会导致编译错误，因为不能向非类型参数传递对象作为参数。

警告：

直到 C++20，非类型参数不能是对象，甚至不能是 `double` 和 `float` 值。非类型参数被限定为整型、枚举、指针和引用。C++20 稍微放宽了这些限制，允许浮点类型甚至某种类类型的非类型模板参数。然而，这样的类类型有很多限制，在本书中没有进一步讨论。可以说，`SpreadsheetCell` 类不遵守这些限制。

这个例子展示了模板类的一种奇怪行为：可正常用于一种类型，但另一种类型却会编译失败。

允许用户指定网格初始元素值的一种更详尽方式是使用 `T` 引用作为非类型模板参数。下面是新的类定义：

```
export template <typename T, const T& DEFAULT>
class Grid
{
    // Everything else is the same as the previous example.
};
```

现在可以为任何类型实例化这个类模板。然而，作为第二个模板参数传递的引用必须具有链接(外部链接或内部链接)。下面的示例使用内部链接的初始值声明了 `int` 和 `SpreadsheetCell` 网络：

```

namespace {
    int defaultInt { 11 };
    SpreadsheetCell defaultCell { 1.2 };
}

int main()
{
    Grid<int, defaultInt> myIntGrid;
    Grid<SpreadsheetCell, defaultCell> mySpreadsheet;
}

```

26.2 类模板部分特例化

第12章中 `const char*` 类的特例化被称为完整类模板特例化，因为它对 `Grid` 模板中的每个模板参数进行了特例化。在这个特例化中没有剩下任何模板参数。这并不是特例化类的唯一方式；还可编写部分特例化的类，这个类允许特例化部分模板参数，而不处理其他参数。例如，基本版本的 `Grid` 模板带有宽度和高度的非类型参数：

```

export template <typename T, size_t WIDTH, size_t HEIGHT>
class Grid
{
public:
    Grid() = default;
    virtual ~Grid() = default;

    // Explicitly default a copy constructor and assignment operator.
    Grid(const Grid& src) = default;
    Grid& operator=(const Grid& rhs) = default;

    std::optional<T>& at(size_t x, size_t y);
    const std::optional<T>& at(size_t x, size_t y) const;

    size_t getHeight() const { return HEIGHT; }
    size_t getWidth() const { return WIDTH; }
private:
    void verifyCoordinate(size_t x, size_t y) const;

    std::optional<T> m_cells[WIDTH][HEIGHT];
};

```

可采用这种方式为 `char* C` 风格字符串特例化这个类模板：

```

export template <size_t WIDTH, size_t HEIGHT>
class Grid<const char*, WIDTH, HEIGHT>
{
public:
    Grid() = default;
    virtual ~Grid() = default;

    // Explicitly default a copy constructor and assignment operator.
    Grid(const Grid& src) = default;
    Grid& operator=(const Grid& rhs) = default;

    std::optional<std::string>& at(size_t x, size_t y);
};

```

```

        const std::optional<std::string>& at(size_t x, size_t y) const;

        size_t getHeight() const { return HEIGHT; }
        size_t getWidth() const { return WIDTH; }
    private:
        void verifyCoordinate(size_t x, size_t y) const;

        std::optional<std::string> m_cells[WIDTH][HEIGHT];
};

```

在这个例子中，没有特例化所有模板参数。因此，模板代码行如下所示：

```

export template <size_t WIDTH, size_t HEIGHT>
class Grid<const char*, WIDTH, HEIGHT>

```

注意，这个模板只有两个参数：WIDTH 和 HEIGHT。然而，这个 Grid 类带有 3 个参数：T、WIDTH 和 HEIGHT。因此，模板参数列表包含两个参数，而显式的 Grid<const char*, WIDTH, HEIGHT> 包含 3 个参数。实例化模板时仍然必须指定 3 个参数。不能只通过高度和宽度实例化模板：

```

Grid<int, 2, 2> myIntGrid;           // Uses the original Grid
Grid<const char*, 2, 2> myStringGrid; // Uses the partial specialization
Grid<2, 3> test;                     // DOES NOT COMPILE! No type specified.

```

上述语法的确很乱。更糟糕的是，在部分特例化中，与完整特例化不同，在每个方法定义的前面要包含模板代码行，如下所示：

```

template <size_t WIDTH, size_t HEIGHT>
const std::optional<std::string>&
    Grid<const char*, WIDTH, HEIGHT>::at(size_t x, size_t y) const
{
    verifyCoordinate(x, y);
    return m_cells[x][y];
}

```

需要这一带有两个参数的模板行，以表示这个方法针对这两个参数做了参数化处理。注意，需要表示完整类名时，都要使用 Grid<const char*, WIDTH, HEIGHT>。

前面的例子并没有表现出部分特例化的真正威力。可为可能的类型子集编写特例化的实现，而不需要为每种类型特例化。例如，可为所有的指针类型编写特例化的 Grid 类。这种特例化的拷贝构造函数和赋值运算符可对指针指向的对象执行深层复制，而不是保存网格中指针的浅层复制。

下面是类的定义，假设只用一个参数特例化最早版本的 Grid。在这个实现中，Grid 成为所提供指针的拥有者，因此它在需要时自动释放内存。需要拷贝构造函数和拷贝赋值运算符，像往常一样，拷贝赋值运算符使用第 9 章讨论的拷贝交换习惯用法，它需要 noexcept swap() 方法。

```

export template <typename T>
class Grid<T*>
{
    public:
        explicit Grid(size_t width = DefaultWidth, size_t height = DefaultHeight);
        virtual ~Grid() = default;

        // Copy constructor and copy assignment operator.
        Grid(const Grid& src);
        Grid& operator=(const Grid& rhs);

        // Explicitly default a move constructor and assignment operator.

```

```

Grid(Grid&& src) = default;
Grid& operator=(Grid&& rhs) = default;

void swap(Grid& other) noexcept;

std::unique_ptr<T>& at(size_t x, size_t y);
const std::unique_ptr<T>& at(size_t x, size_t y) const;

size_t getHeight() const { return m_height; }
size_t getWidth() const { return m_width; }

static const size_t DefaultWidth { 10 };
static const size_t DefaultHeight { 10 };
private:
    void verifyCoordinate(size_t x, size_t y) const;

    std::vector<std::vector<std::unique_ptr<T>>> m_cells;
    size_t m_width { 0 }, m_height { 0 };
};

```

像往常一样，下面这两行代码是关键所在：

```

export template <typename T>
class Grid<T*>

```

上述语法表明这个类是 Grid 类模板对所有指针类型的特例化。只有 T 是指针类型的情况下才提供实现。请注意，如果像下面这样实例化网格：Grid<int*> myIntGrid，那么 T 实际上是 int 而非 int*。这不够直观，但遗憾的是，这种语法就是这样使用的。下面是一个示例：

```

Grid<int> myIntGrid;           // Uses the non-specialized grid.
Grid<int*> psGrid { 2, 2 };    // Uses the partial specialization for pointer types.

psGrid.at(0, 0) = make_unique<int>(1);
psGrid.at(0, 1) = make_unique<int>(2);
psGrid.at(1, 0) = make_unique<int>(3);

Grid<int*> psGrid2 { psGrid };
Grid<int*> psGrid3;
psGrid3 = psGrid2;

auto& element { psGrid2.at(1, 0) };
if (element) {
    cout << *element << endl;
    *element = 6;
}

cout << *psGrid.at(1, 0) << endl; // psGrid is not modified.
cout << *psGrid2.at(1, 0) << endl; // psGrid2 is modified.

```

输出如下：

```

3
3
6

```

方法的实现相当简单，但拷贝构造函数除外，拷贝构造函数使用各个元素的拷贝构造函数进行深

层复制:

```
template <typename T>
Grid<T*>::Grid(const Grid& src)
    : Grid { src.m_width, src.m_height }
{
    // The ctor-initializer of this constructor delegates first to the
    // non-copy constructor to allocate the proper amount of memory.

    // The next step is to copy the data.
    for (size_t i { 0 }; i < m_width; i++) {
        for (size_t j { 0 }; j < m_height; j++) {
            // Make a deep copy of the element by using its copy constructor.
            if (src.m_cells[i][j]) {
                m_cells[i][j].reset(new T { *(src.m_cells[i][j]) });
            }
        }
    }
}
```

26.3 通过重载模拟函数部分特例化

C++标准不允许函数的模板部分特例化。相反,可用另一个模板重载函数。区别十分微妙。假设要编写一个特例化的 Find()函数模板(参见第 12 章),这个特例化对指针解除引用,对指向的对象直接调用 operator==。实现此行为的正确方法是为 Find()方法编写第二个函数模板:

```
template <typename T>
size_t Find(T* value, T* const* arr, size_t size)
{
    for (size_t i { 0 }; i < size; i++) {
        if (*arr[i] == *value) {
            return i; // Found it; return the index
        }
    }
    return NOT_FOUND; // failed to find it; return NOT_FOUND
}
```

可在一个程序中定义原始的 Find()模板、针对指针类型的重载 Find()、针对 const char*的重载 Find()。编译器会根据推导规则选择合适的版本来调用。

注意:

在所有重载的版本之间,编译器总是选择“最具体的”函数版本。如果非模板化的版本与函数模板实例化等价,编译器更偏向非模板化的版本。

下面的代码调用了几次 Find(), 里面的注释说明了调用的是哪个版本的 Find():

```
size_t res { NOT_FOUND };

int myInt { 3 }, intArray[] { 1, 2, 3, 4 };
size_t sizeArray { size(intArray) };
res = Find(myInt, intArray, sizeArray); // calls Find<int> by deduction
res = Find<int>(myInt, intArray, sizeArray); // calls Find<int> explicitly
```

```

double myDouble { 5.6 }, doubleArray[] { 1.2, 3.4, 5.7, 7.5 };
sizeArray = size(doubleArray);
// calls Find<double> by deduction
res = Find(myDouble, doubleArray, sizeArray);
// calls Find<double> explicitly
res = Find<double>(myDouble, doubleArray, sizeArray);

const char* word { "two" };
const char* words[] { "one", "two", "three", "four" };
sizeArray = size(words);

// calls Find<const char*> explicitly
res = Find<const char*>(word, words, sizeArray);
// calls overloaded Find for const char*s
res = Find(word, words, sizeArray);

int *intPointer { &myInt }, *pointerArray[] { &myInt, &myInt };
sizeArray = size(pointerArray);
// calls the overloaded Find for pointers
res = Find(intPointer, pointerArray, sizeArray);

SpreadsheetCell cell1 { 10 };
SpreadsheetCell cellArray[] { SpreadsheetCell { 4 }, SpreadsheetCell { 10 } };
sizeArray = size(cellArray);
// calls Find<SpreadsheetCell> by deduction
res = Find(cell1, cellArray, sizeArray);
// calls Find<SpreadsheetCell> explicitly
res = Find<SpreadsheetCell>(cell1, cellArray, sizeArray);

SpreadsheetCell *cellPointer { &cell1 };
SpreadsheetCell *cellPointerArray[] { &cell1, &cell1 };
sizeArray = size(cellPointerArray);
// Calls the overloaded Find for pointers
res = Find(cellPointer, cellPointerArray, sizeArray);

```

26.4 模板递归

C++模板提供的功能比本章前面和第12章介绍的简单类和函数模板强大得多。其中一项功能称为模板递归。模板递归类似于函数递归，意思是一个函数的定义是通过调用自身实现。这一节首先讲解模板递归的动机，然后讲述如何实现模板递归。

26.4.1 N维网格：初次尝试

前面的 Grid 模板示例到现在为止只支持两个维度，这限制了它的实用性。如果想编写三维井字游戏(Tic-Tac-Toe)或四维矩阵的数学程序，该怎么办？当然，可为每个维度写一个模板类或非模板类。然而，这会重复很多代码。另一种方法是只编写一个一维网格。然后，利用另一个网格作为元素类型实例化 Grid，可创建任意维度的网格。这种 Grid 元素类型本身可以用网格作为元素类型进行实例化，以此类推。下面是 OneDGrid 类模板的实现。这只是前面例子中 Grid 模板的一维版本，添加了 `resize()` 方法，并用 `operator[]` 替换 `at()`。与诸如 `vector` 的标准库容器类似，`operator[]` 实现不执行边界检查。另外，在这个示例中，`m_elements` 存储 `T` 的实例而非 `std::optional<T>` 的实例。

```
export template <typename T>
```

```

class OneDGrid
{
public:
    explicit OneDGrid(size_t size = DefaultSize) { resize(size); }
    virtual ~OneDGrid() = default;

    T& operator[](size_t x) { return m_elements[x]; }
    const T& operator[](size_t x) const { return m_elements[x]; }

    void resize(size_t newSize) { m_elements.resize(newSize); }
    size_t getSize() const { return m_elements.size(); }

    static const size_t DefaultSize { 10 };
private:
    std::vector<T> m_elements;
};

```

有了 OneDGrid 的这个实现，就可通过如下方式创建多维网格：

```

OneDGrid<int> singleDGrid;
OneDGrid<OneDGrid<int>> twoDGrid;
OneDGrid<OneDGrid<OneDGrid<int>>> threeDGrid;
singleDGrid[3] = 5;
twoDGrid[3][3] = 5;
threeDGrid[3][3][3] = 5;

```

此代码可正常工作，但声明代码看上去有点乱。下面对其加以改进。

26.4.2 真正的 N 维网格

可使用模板递归编写“真正的” N 维网格，因为网格的维度在本质上是递归的。从如下声明中可以看出：

```
OneDGrid<OneDGrid<OneDGrid<int>>> threeDGrid;
```

可将嵌套的每层 OneDGrid 想象为一个递归步骤，int 的 OneDGrid 是递归的基本情形。换句话说，三维网格是 int 一维网格的一维网格的一维网格。用户不需要自己进行递归，可以编写一个类模板来自动进行递归。然后，可创建如下 N 维网格：

```

NDGrid<int, 1> singleDGrid;
NDGrid<int, 2> twoDGrid;
NDGrid<int, 3> threeDGrid;

```

NDGrid 类模板需要元素类型和表示维度的整数作为参数。这里的关键问题在于，NDGrid 的元素类型不是模板参数列表中指定的元素类型，而是上一层递归的维度中指定的另一个 NDGrid。换句话说，三维网格是二维网格的矢量，二维网格是一维网格的各个矢量。

使用递归时，需要处理基本情形(base case)。可编写维度为 1 的部分特例化的 NDGrid，其中元素类型不是另一个 NDGrid，而是模板参数指定的元素类型。

下面是 NDGrid 模板定义和实现，突出显示了与前面 OneDGrid 的不同之处。模板递归实现最棘手的部分不是模板递归本身，而是网格中每个维度的正确大小。这个实现创建了 N 维网格，每个维度都一样大。为每个维度指定不同的大小要困难得多。然而，即使做了这样的简化，也仍然存在一个问题：用户应该有能力创建指定大小的数组，例如 20 或 50。因此，构造函数接收一个整数作为大小参数。然而，当动态重设子网格的 vector 时，不能将这个大小参数传递给子网格元素，因为 vector

使用默认的构造函数创建对象。因此,必须对 `vector` 的每个网格元素显式调用 `resize()`。注意 `m_elements` 是 `NDGrid<T, N-1>` 的 `vector`, 这是递归步骤。此外, `operator[]` 返回类型的引用, 同样是 `NDGrid<T, N-1>`, 而不是 `T`。

```
export template <typename T, size_t N>
class NDGrid
{
public:
    explicit NDGrid(size_t size = DefaultSize) { resize(size); }
    virtual ~NDGrid() = default;

    NDGrid<T, N-1>& operator[](size_t x) { return m_elements[x]; }
    const NDGrid<T, N-1>& operator[](size_t x) const { return m_elements[x]; }
    void resize(size_t newSize)
    {
        m_elements.resize(newSize);
        // Resizing the vector calls the 0-argument constructor for
        // the NDGrid<T, N-1> elements, which constructs
        // it with the default size. Thus, we must explicitly call
        // resize() on each of the elements to recursively resize all
        // nested Grid elements.
        for (auto& element : m_elements) {
            element.resize(newSize);
        }
    }

    size_t getSize() const { return m_elements.size(); }

    static const size_t DefaultSize { 10 };
private:
    std::vector<NDGrid<T, N-1>> m_elements;
};
```

基本用例的模板定义是维度为 1 的部分特化, 下面突出显示了定义和实现。请注意, 必须重写很多代码, 因为不能在特化中继承任何实现。这里突出显示了与非特化 `NDGrid` 之间的差异:

```
export template <typename T>
class NDGrid<T, 1>
{
public:
    explicit NDGrid(size_t size = DefaultSize) { resize(size); }
    virtual ~NDGrid() = default;

    T& operator[](size_t x) { return m_elements[x]; }
    const T& operator[](size_t x) const { return m_elements[x]; }

    void resize(size_t newSize) { m_elements.resize(newSize); }
    size_t getSize() const { return m_elements.size(); }

    static const size_t DefaultSize { 10 };
private:
    std::vector<T> m_elements;
};
```

至此, 递归结束: 元素类型是 `T`, 而不是另外的模板实例化。

现在，可编写下面这样的代码：

```
NDGrid<int, 3> my3DGrid { 4 };
my3DGrid[2][1][2] = 5;
my3DGrid[1][1][1] = 5;
cout << my3DGrid[2][1][2] << endl;
```

26.5 可变参数模板

普通模板只可采取固定数量的模板参数。可变参数模板(variadic template)可接收可变数目的模板参数。例如，下面的代码定义了一个模板，它可以接收任何数目的模板参数，使用称为 Types 的参数包(parameter pack)：

```
template<typename... Types>
class MyVariadicTemplate { };
```

注意：

typename 之后的 3 个点并非错误。这是为可变参数模板定义参数包的语法。参数包可以接收可变数目的参数。在 3 个点的前后允许添加空格。

可用任何数量的类型实例化 MyVariadicTemplate，例如：

```
MyVariadicTemplate<int> instance1;
MyVariadicTemplate<string, double, list<int>>> instance2;
```

甚至可用零个模板参数实例化 MyVariadicTemplate：

```
MyVariadicTemplate<> instance3;
```

为阻止用零个模板参数实例化可变参数模板，可以像下面这样编写模板：

```
template<typename T1, typename... Types>
class MyVariadicTemplate { };
```

有了这个定义后，试图通过零个模板参数实例化 MyVariadicTemplate 会导致编译错误。

不能直接遍历传给可变参数模板的不同参数。唯一方法是借助模板递归或折叠表达式的帮助。下面将展示这两种方法的示例。

26.5.1 类型安全的变长参数列表

可变参数模板允许创建类型安全的变长参数列表。下面的例子定义了一个可变参数模板 processValues()，它允许以类型安全的方式接收不同类型的可变数目的参数。函数 processValues()会处理变长参数列表中的每个值，对每个参数执行 handleValue()函数。这意味着必须对每种要处理的类型编写 handleValue()函数，例如下例中的 int、double 和 string：

```
void handleValue(int value) { cout << "Integer: " << value << endl; }
void handleValue(double value) { cout << "Double: " << value << endl; }
void handleValue(string_view value) { cout << "String: " << value << endl; }

void processValues() // Base case to stop recursion
{ /* Nothing to do in this base case.*/ }
```

```
template<typename T1, typename... Tn>
void processValues(T1 arg1, Tn... args)
{
    handleValue(arg1);
    processValues(args...);
}
```

在这个例子中，三点运算符“...”用了两次。这个运算符出现在3个地方，有两个不同的含义。首先，用在模板参数列表中 `typename` 的后面以及函数参数列表中类型 `Tn` 的后面。在这两种情况下，它都表示参数包。参数包可接收可变数目的参数。

“...”运算符的第二种用法是在函数体中参数名 `args` 的后面。这种情况下，它表示参数包扩展。这个运算符会解包/展开参数包，得到各个参数。它基本上提取出运算符左边的内容，为包中的每个模板参数重复该内容，并用逗号隔开。从前面的例子中取出以下行：

```
processValues(args...);
```

这一行将 `args` 参数包解包(或扩展)为不同的参数，通过逗号分隔参数，然后用这些展开的参数调用 `processValues()` 函数。模板总是需要至少一个模板参数：`T1`。通过 `args...` 递归调用 `processValues()` 的结果是：每次调用都会少一个模板参数。

由于 `processValues()` 函数的实现是递归的，因此需要采用一种方法停止递归。为此，实现一个 `processValues()` 函数，要求它接收零个参数。

可通过下面的代码测试 `processValues()` 可变参数模板：

```
processValues(1, 2, 3.56, "test", 1.1f);
```

这个例子生成的递归调用是：

```
processValues(1, 2, 3.56, "test", 1.1f);
    handleValue(1);
    processValues(2, 3.56, "test", 1.1f);
        handleValue(2);
        processValues(3.56, "test", 1.1f);
            handleValue(3.56);
            processValues("test", 1.1f);
                handleValue("test");
                processValues(1.1f);
                    handleValue(1.1f);
                    processValues();
```

重要的是要记住，这种变长参数列表是完全类型安全的。`processValues()` 函数会根据实际类型自动调用正确的 `handleValue()` 重载版本。C++中也会像通常那样自动执行类型转换。例如，前面例子中 `1.1f` 的类型为 `float`。`processValues()` 函数会调用 `handleValue(double value)`，因为从 `float` 到 `double` 的转换没有任何损失。然而，如果调用 `processValues()` 时带有某种类型的参数，而这种类型没有对应的 `handleValue()` 函数，编译器会产生错误。

前面的实现存在一个小问题。由于这是一个递归的实现，因此每次递归调用 `processValues()` 时都会复制参数。根据参数的类型，这种做法的代价可能会很高。你可能会认为，向 `processValues()` 函数传递引用而不使用按值传递方法，就可以避免这种复制问题。遗憾的是，这样也无法通过字面量调用 `processValues()` 了，因为不允许使用字面量引用，除非使用 `const` 引用。

为了在使用非 `const` 引用的同时也能使用字面量值，可使用转发引用(forwarding references)。以下实现使用了转发引用 `T&&`，还使用 `std::forward()` 完美转发所有参数。“完美转发”意味着，如果把 `rvalue`

传递给 `processValues()`, 就将它作为 `rvalue` 引用转发; 如果把 `lvalue` 或 `lvalue` 引用传递给 `processValues()`, 就将它作为 `lvalue` 引用转发。

```
void processValues() // Base case to stop recursion
{ /* Nothing to do in this base case.*/ }

template<typename T1, typename... Tn>
void processValues(T1&& arg1, Tn&&... args)
{
    handleValue(std::forward<T1>(arg1));
    processValues(std::forward<Tn>(args)...);
}
```

有一行代码需要做进一步解释:

```
processValues(std::forward<Tn>(args)...);
```

“...”运算符用于解开参数包, 它在参数包中的每个参数上使用 `std::forward()`, 用逗号把它们隔开。例如, 假设 `args` 是一个参数包, 有 3 个参数(`a1`、`a2` 和 `a3`), 分别对应 3 种类型(`A1`、`A2` 和 `A3`)。扩展后的调用如下:

```
processValues(std::forward<A1>(a1),
              std::forward<A2>(a2),
              std::forward<A3>(a3));
```

在使用了参数包的函数体中, 可通过以下方法获得参数包中参数的个数:

```
int numofArguments { sizeof...(args) };
```

一个使用变长参数模板的实际例子是编写一个类似于 `printf()` 版本的安全且类型安全的函数模板。这是实践变长参数模板的一次不错的练习。

注意:

只有当 `T` 作为函数或方法模板的一个模板参数时, `T&&` 才是转发引用。如果一个类方法有个 `T&&` 参数, 但 `T` 是类的模板参数而不是方法本身的参数, 那么 `T&&` 就不是一个转发引用, 而是右值引用。这是因为当编译器开始用一个 `T&&` 参数处理那个方法时, 类模板参数 `T` 已经被解析为一个具体类型, 例如 `T`, 那时方法参数已经被 `int&&` 替换。

26.5.2 可变数目的混入类

参数包几乎可用在任何地方。例如, 下面的代码使用一个参数包为 `MyClass` 类定义了可变数目的混入类。第 5 章讨论了混入类的概念。

```
class Mixin1
{
public:
    Mixin1(int i) : m_value { i } {}
    virtual void mixin1Func() { cout << "Mixin1: " << m_value << endl; }
private:
    int m_value;
};

class Mixin2
{
```

```

public:
    Mixin2(int i) : m_value { i } {}
    virtual void mixin2Func() { cout << "Mixin2: " << m_value << endl; }
private:
    int m_value;
};

template <typename... Mixins>
class MyClass : public Mixins...
{
public:
    MyClass(const Mixins&... mixins) : Mixins { mixins }... {}
    virtual ~MyClass() = default;
};

```

上述代码首先定义了两个混入类 `Mixin1` 和 `Mixin2`。它们在这个例子中的定义非常简单。它们的构造函数接收一个整数，然后保存这个整数，这两个类有一个函数用于打印特定实例的信息。`MyClass` 可变参数模板使用参数包 `typename... Mixins` 接收可变数目的混入类。`MyClass` 继承所有的混入类，其构造函数接收同样数目的参数来初始化每一个继承的混入类。注意，扩展运算符(...)基本上接收运算符左边的内容，为参数包中的每个模板参数重复这些内容，并用逗号隔开。`MyClass` 可以这样使用：

```

MyClass<Mixin1, Mixin2> a { Mixin1 { 11 }, Mixin2 { 22 } };
a.mixin1Func();
a.mixin2Func();

MyClass<Mixin1> b { Mixin1 { 33 } };
b.mixin1Func();
//b.mixin2Func(); // Error: does not compile.

MyClass<> c;
//c.mixin1Func(); // Error: does not compile.
//c.mixin2Func(); // Error: does not compile.

```

试图对 `b` 调用 `Mixin2Func()` 会产生编译错误，因为 `b` 并非继承自 `Mixin2` 类。这个程序的输出如下：

```

Mixin1: 11
Mixin2: 22
Mixin1: 33

```

26.5.3 折叠表达式

C++ 支持折叠表达式。这样一来，将可更容易地在可变参数模板中处理参数包。表 26-1 列出了支持的 4 种折叠类型。在该表中， Θ 可以是以下任意运算符：+、-、*、/、%、^、&、|、<<、>>、+=、-=、*=、/=、%=、^=、&=、|=、<<=、>>=、=、==、!=、<、>、<=、>=、&&、||、,、*、->。

表 26-1 4 种折叠类型

名称	表达式	含义
一元右折叠	$(\text{pack } \Theta \dots)$	$\text{pack}_0 \Theta (\dots \Theta (\text{pack}_{n-1} \Theta \text{pack}_n))$
一元左折叠	$(\dots \Theta \text{pack})$	$((\text{pack}_0 \Theta \text{pack}_1) \Theta \dots) \Theta \text{pack}_n$
二元右折叠	$(\text{pack } \Theta \dots \Theta \text{Init})$	$\text{pack}_0 \Theta (\dots \Theta (\text{pack}_{n-1} \Theta (\text{pack}_n \Theta \text{Init})))$
二元左折叠	$(\text{Init } \Theta \dots \Theta \text{pack})$	$((((\text{Init } \Theta \text{pack}_0) \Theta \text{pack}_1) \Theta \dots) \Theta \text{pack}_n$

下面分析一些示例。以递归方式定义前面的 `processValues()` 函数模板，如下所示：

```
void processValues() { /* Nothing to do in this base case.*/ }

template<typename T1, typename... Tn>
void processValues(T1 arg1, Tn... args)
{
    handleValue(arg1);
    processValues(args...);
}
```

由于以递归方式定义，因此需要基本情形来停止递归。使用折叠表达式，利用一元右折叠，通过单个函数模板来实现。此时，不需要基本情形。

```
template<typename... Tn>
void processValues(const Tn&... args)
{
    (handleValue(args), ...);
}
```

基本上，函数体中的 3 个点触发折叠。扩展这一行，针对参数包中的每个参数调用 `handleValue()`，对 `handleValue()` 的每个调用用逗号分隔。例如，假设 `args` 是包含 3 个参数(`a1`、`a2` 和 `a3`)的参数包。一元右折叠扩展后的形式如下：

```
(handleValue(a1), (handleValue(a2), handleValue(a3)));
```

下面是另一个示例。`printValues()` 函数模板将所有实参写入控制台，实参之间用换行符分开。

```
template<typename... Values>
void printValues(const Values&... values)
{
    ((cout << values << endl), ...);
}
```

假设 `values` 是包含 3 个参数(`v1`、`v2` 和 `v3`)的参数包。一元右折叠扩展后的形式如下：

```
((cout << v1 << endl), ((cout << v2 << endl), (cout << v3 << endl)));
```

调用 `printValues()` 时可使用任意数量的实参，如下所示：

```
printValues(1, "test", 2.34);
```

在这些示例中，将折叠与逗号运算符结合使用，但实际上，折叠可与任何类型的运算符结合使用。例如，以下代码定义了可变参数函数模板，使用二元左折叠计算传给它的所有值之和。二元左折叠始终需要一个 `Init` 值(参见表 26-1)。因此，`sumValues()` 有两个模板类型参数：一个是普通参数，用于指定 `Init` 的类型；另一个是参数包，可接收 0 个或多个实参。

```
template<typename T, typename... Values>
double sumValues(const T& init, const Values&... values)
{
    return (init + ... + values);
}
```

假设 `values` 是包含 3 个参数(`v1`、`v2` 和 `v3`)的参数包。二元左折叠扩展后的形式如下：

```
return (((init + v1) + v2) + v3);
```

sumValues()函数模板的使用方式如下:

```
cout << sumValues(1, 2, 3.3) << endl;
cout << sumValues(1) << endl;
```

该函数模板至少需要一个参数, 因此以下代码无法编译:

```
cout << sumValues() << endl;
```

sumValues()函数模板也可以按照一元左折叠定义, 如下所示, 这仍然需要在调用 sumValues()时提供至少一个参数。

```
template <typename... Values>
double sumValues(const Values&... values) { return (... + values); }
```

长度为零的参数包允许一元折叠, 但只能和逻辑与(&&)、逻辑或(||)和逗号(,)运算符结合使用。例如:

```
template <typename... Values>
double allTrue(const Values&... values) { return (... && values); }
```

```
template <typename... Values>
double anyTrue(const Values&... values) { return (... || values); }
```

```
int main()
{
    cout << allTrue(1, 1, 0) << allTrue(1, 1) << allTrue() << endl; // 011
    cout << anyTrue(1, 1, 0) << anyTrue(0, 0) << anyTrue() << endl; // 100
}
```

26.6 模板元编程

本节讲解模板元编程。这是一个非常复杂的主题, 有一些关于模板元编程的书讲解了所有细节。本书没有足够的篇幅来讲解模板元编程的所有细节。本节通过几个例子解释最重要的概念。

模板元编程的目标是在编译期而不是运行时执行一些计算。模板元编程基本上是基于另一种语言的一种小型编程语言。下面首先讨论一个简单示例, 这个例子在编译期计算一个数的阶乘, 并在运行时能将计算结果用作简单的常数。

26.6.1 编译期阶乘

下面的代码演示了在编译期如何计算一个数的阶乘。代码使用了本章前面介绍的模板递归, 我们需要一个递归模板和用于停止递归的基本模板。根据数学定义, 0 的阶乘是 1, 所以用作基本情形:

```
template<unsigned char f>
class Factorial
{
public:
    static const unsigned long long value = (f * Factorial<f - 1>::value);
};

template<>
class Factorial<0>
{
public:
    static const unsigned long long value { 1 };
};
```

```
};

int main()
{
    cout << Factorial<6>::value << endl;
}
```

这将计算 6 的阶乘，数学表达为 $6!$ ，值为 $1 \times 2 \times 3 \times 4 \times 5 \times 6$ 或 720。

注意：

要记住，在编译期计算阶乘。在运行时，可通过 `::value` 访问编译期计算出来的值，这不过是一个静态常量值。

上面这个具体示例在编译期计算一个数的阶乘，但未必需要使用模板元编程。可不使用模板，写成 C++20 的 `constexpr` 函数形式。不过，模板实现仍然是实现递归模板的优秀示例。

```
constexpr unsigned long long factorial(unsigned char f)
{
    if (f == 0) { return 1; }
    else { return f * factorial(f - 1); }
}
```

可以像调用其他函数一样调用 `factorial()`，不同之处在于 `constexpr` 函数保证在编译期执行。

26.6.2 循环展开

模板元编程的第二个例子是在编译期展开循环，而不是在运行时执行循环。注意循环展开(loop unrolling)应仅在需要时使用，因为编译器通常足够智能，会自动展开可以展开的循环。

这个例子再次使用了模板递归，因为需要在编译期在循环中完成一些事情。在每次递归中，`Loop` 类模板都会通过 `i - 1` 实例化自身。当到达 0 时，停止递归。

```
template <int i>
class Loop
{
public:
    template <typename FuncType>
    static inline void run(FuncType func) {
        Loop<i - 1>::run(func);
        func(i);
    }
};

template <>
class Loop<0>
{
public:
    template <typename FuncType>
    static inline void run(FuncType /* func */) { }
};
```

可以像下面这样使用 `Loop` 模板：

```
void doWork(int i) { cout << "doWork(" << i << ")" << endl; }

int main()
```

```
{
    Loop<3>::Do(doWork);
}
```

这段代码将导致编译器展开循环，并连续 3 次调用 `doWork()` 函数。这个程序的输出如下所示：

```
doWork(1)
doWork(2)
doWork(3)
```

26.6.3 打印元组

这个例子通过模板元编程来打印 `std::tuple` 中的各个元素。第 24 章“其他 C++ 工具”讲解了元组。元组允许存储任何数量的值，每个值都有各自的特定类型。元组有固定的大小和值类型，这些都是在编译期确定的。然而，元组没有提供任何内置的机制来遍历其元素。下面的示例演示如何通过模板元编程在编译期遍历元组中的元素。

与模板元编程中的大部分情况一样，这个例子也使用了模板递归。`tuple_print` 类模板接收两个模板参数：`tuple` 类型和初始化为元组大小的整数。然后在构造函数中递归地实例化自身，每一次调用都将大小减小。当大小变成 0 时，`tuple_print` 的一个部分特例化停止递归。`main()` 函数演示了如何使用这个 `tuple_print` 类模板。

```
template <typename TupleType, int n>
class TuplePrint
{
public:
    TuplePrint(const TupleType& t) {
        TuplePrint<TupleType, n - 1> tp { t };
        cout << get<n - 1>(t) << endl;
    }
};

template <typename TupleType>
class TuplePrint<TupleType, 0>
{
public:
    TuplePrint(const TupleType&) { }
};

int main()
{
    using MyTuple = tuple<int, string, bool>;
    MyTuple t1 { 16, "Test", true };
    TuplePrint<MyTuple, tuple_size<MyTuple>::value> tp { t1 };
}
```

`main()` 中的 `TuplePrint` 语句看起来有点复杂，因为它要求元组的确切类型和大小作为模板参数。引入自动推导模板参数的辅助函数模板可以简化这段代码。简化的实现如下所示：

```
template <typename TupleType, int n>
class TuplePrintHelper
{
public:
    TuplePrintHelper(const TupleType& t) {
        TuplePrintHelper<TupleType, n - 1> tp { t };
        cout << get<n - 1>(t) << endl;
    }
};
```



```

    }
};

template <typename TupleType>
class TuplePrintHelper<TupleType, 0>
{
public:
    TuplePrintHelper(const TupleType&) {}
};

template <typename T>
void tuplePrint(const T& t)
{
    TuplePrintHelper<T, tuple_size<T>::value> tph { t };
}

int main()
{
    tuple t1 { 167, "Testing"s, false, 2.3 };
    tuplePrint(t1);
}

```

这里的第一个变化是将原来的 `tuplePrint` 类模板重命名为 `tuplePrintHelper`。然后，上述代码实现了一个名为 `tuplePrint()` 的小函数模板，这个函数模板接收 `tuple` 类型作为模板类型参数，并接收对元组本身的引用作为函数参数。在该函数的主体中实例化 `tuplePrintHelper` 类模板。`main()` 函数展示了如何使用这个简化的版本。不需要指定函数模板的参数，因为编译器可以根据提供的参数自动推断。

1. constexpr if

C++17 引入了 `constexpr if`。这些是在编译期(而非运行时)执行的 `if` 语句。如果 `constexpr if` 语句的分支从未到达，就不会进行编译。这可用于简化大量的模板元编程技术。例如，可按如下方式使用 `constexpr if`，简化前面的打印元组元素的代码。注意，不再需要模板递归基本情形，原因在于可通过 `constexpr if` 语句停止递归。

```

template <typename TupleType, int n>
class TuplePrintHelper
{
public:
    TuplePrintHelper(const TupleType& t) {
        if constexpr (n > 1) {
            TuplePrintHelper<TupleType, n - 1> tp { t };
        }
        cout << get<n - 1>(t) << endl;
    }
};

template <typename T>
void tuplePrint(const T& t)
{
    TuplePrintHelper<T, tuple_size<T>::value> tph { t };
}

```

现在，甚至可丢弃类模板本身，替换为简单的函数模板 `tuplePrintHelper()`：

```

template <typename TupleType, int n>
void tuplePrintHelper(const TupleType& t) {

```

```

    if constexpr (n > 1) {
        tuplePrintHelper<TupleType, n - 1>(t);
    }
    cout << get<n - 1>(t) << endl;
}

template <typename T>
void tuplePrint(const T& t)
{
    tuplePrintHelper<T, tuple_size<T>::value>(t);
}

```

可对其进一步简化。将两个方法合为一个，如下所示：

```

template <typename TupleType, int n = tuple_size<TupleType>::value>
void tuplePrint(const TupleType& t) {
    if constexpr (n > 1) {
        tuplePrint<TupleType, n - 1>(t);
    }
    cout << get<n - 1>(t) << endl;
}

```

仍然像前面那样进行调用：

```

tuple t1 { 167, "Testing"s, false, 2.3 };
tuplePrint(t1);

```

2. 使用编译期整数序列和折叠

C++使用 `std::integer_sequence` (在 `<utility>` 中定义) 支持编译期整数序列。模板元编程的一个常见用例是生成编译期索引序列，即 `size_t` 类型的整数序列。此处，可使用辅助用的 `std::index_sequence`。可使用 `std::index_sequence_for()`，生成与给定的参数包等长的索引序列。

下面使用可变参数模板、编译期索引序列和折叠表达式，实现元组打印程序：

```

template <typename Tuple, size_t... Indices>
void tuplePrintHelper(const Tuple& t, index_sequence<Indices...>)
{
    ((cout << get<Indices>(t) << endl), ...);
}

template <typename... Args>
void tuplePrint(const tuple<Args...>& t)
{
    tuplePrintHelper(t, index_sequence_for<Args...>());
}

```

可按与前面相同的方式调用：

```

tuple t1 { 167, "Testing"s, false, 2.3 };
tuplePrint(t1);

```

调用时，`tuplePrintHelper()` 函数模板中的一元右折叠表达式扩展为如下形式：

```

(((cout << get<0>(t) << endl),
 (cout << get<1>(t) << endl),
 (cout << get<2>(t) << endl),
 (cout << get<3>(t) << endl))));

```

26.6.4 类型 trait

通过类型 trait 可在编译时根据类型做出决策。例如, 可验证一个类型是否从另一个类型派生而来、是否可以转换为另一个类型、是否是整型等。C++标准提供了大量可供选择的类型 trait 类。所有与类型 trait 相关的功能都定义在<type_traits>中。类型 trait 分为几个不同类别。下面列出了每个类别的可用类型 trait 的一些例子。完整清单请参阅标准库参考资源(见附录 B)。

表 26-2 C++标准中的部分类型 trait

➤ 原始类型类别 • is_void • is_integral • is_floating_point • is_pointer • is_function • ... ➤ 类型属性 • is_const • is_polymorphic • is_unsigned • is_constructible • is_copy_constructible • is_move_constructible • is_assignable • is_trivially_copyable • is_swappable • is_nothrow_swappable • has_virtual_destructor • has_unique_object_representations • ... ➤ 属性查询 • alignment_of • rank • extent ➤ 引用修改 • remove_reference • add_lvalue_reference • add_rvalue_reference ➤ 指针修改 • remove_pointer • add_pointer ➤ 常量评估上下文 • is_constant_evaluated*	➤ 复合类型类别 • is_reference • is_object • is_scalar • ... ➤ 类型关系 • is_same • is_base_of • is_convertible • is_invocable • is_nothrow_invocable • ... ➤ const-volatile 修改 • remove_const • add_const • ... ➤ 符号修改 • make_signed • make_unsigned ➤ 数组修改 • remove_extent • remove_all_extents ➤ 逻辑运算符 trait • conjunction • disjunction • negation ➤ 其他转换 • enable_if • conditional • invoke_result • type_identity* • remove_cvref* • common_reference* • ...
---	--

标有星号的类型 `trait` 在 C++20 及其之后版本中才可用。

类型 `trait` 是一个非常高级的 C++ 功能。表 26-2 只显示了 C++ 标准中的部分类型 `trait`，从这个列表中就可以看出，本书不可能解释类型 `trait` 的所有细节。下面只列举几个用例，展示如何使用类型 `trait`。

1. 使用类型类别

在给出使用类型 `trait` 的模板示例前，首先要了解一下诸如 `is_integral` 的类的工作方式。C++ 标准对 `integral_constant` 类的定义如下所示：

```
template <class T, T v>
struct integral_constant {
    static constexpr T value { v };
    using value_type = T;
    using type = integral_constant<T, v>;
    constexpr operator value_type() const noexcept { return value; }
    constexpr value_type operator()() const noexcept { return value; }
};
```

这也定义了 `bool_constant`、`true_type` 和 `false_type` 类型别名：

```
template <bool B>
using bool_constant = integral_constant<bool, B>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

这定义了两种类型：`true_type` 和 `false_type`。当调用 `true_type::value` 时，得到的值是 `true`；调用 `false_type::value` 时，得到的值是 `false`。还可调用 `true_type::type`，这将返回 `true_type` 类型。这同样适用于 `false_type`。像 `is_integral` 这样检查类型是否为整型和 `is_class` 这样检查类型是否为类类型的类，都继承自 `true_type` 或者 `false_type`。例如，`is_integral` 为类型 `bool` 特例化，如下所示：

```
template<> struct is_integral<bool> : public true_type { };
```

这样就可编写 `is_integral<bool>::value`，并返回 `true`。注意，不需要自己编写这些特例化，这些是标准库的一部分。

下面的代码演示了使用类型类别的最简单例子：

```
if (is_integral<int>::value) { cout << "int is integral" << endl; }
else { cout << "int is not integral" << endl; }

if (is_class<string>::value) { cout << "string is a class" << endl; }
else { cout << "string is not a class" << endl; }
```

输出如下：

```
int is integral
string is a class
```

对于每一个具有 `value` 成员的 `trait`，C++ 添加了一个变量模板，它与 `trait` 同名，后跟 `_v`。不是编写 `some_trait<T>::value`，而是编写 `some_trait_v<T>`，例如 `is_integral_v<T>` 和 `is_const_v<T>` 等。下面用变量模板重写了前面的例子：

```
if (is_integral_v<int>) { cout << "int is integral" << endl; }
else { cout << "int is not integral" << endl; }
```

```
if (is_class_v<string>) { cout << "string is a class" << endl; }
else { cout << "string is not a class" << endl; }
```

当然，你可能永远都不会采用这种方式使用类型 `trait`。只有结合模板根据类型的某些属性生成代码时，类型 `trait` 才更有用。下面的模板示例演示了这一点。代码定义了函数模板 `process_helper()` 的两个重载版本，这个函数模板接收一种类型作为模板参数。第一个参数是一个值，第二个参数是 `true_type` 或 `false_type` 的实例。`process()` 函数模板接收一个参数，并调用 `processHelper()` 函数：

```
template <typename T>
void processHelper(const T& t, true_type)
{
    cout << t << " is an integral type." << endl;
}

template <typename T>
void processHelper(const T& t, false_type)
{
    cout << t << " is a non-integral type." << endl;
}

template <typename T>
void process(const T& t)
{
    processHelper(t, typename is_integral<T>::type{});
}
```

`processHelper()` 函数调用的第二个参数定义如下：

```
typename is_integral<T>::type{}
```

该参数使用 `is_integral` 判断 `T` 是否为整数类型。使用 `::type` 访问结果 `integral_constant` 类型，可以是 `true_type` 或 `false_type`。`processHelper()` 函数需要 `true_type` 或 `false_type` 的一个实例作为第二个参数，这也是 `::type` 后面有 `{}` 的原因。注意，`processHelper()` 函数的两个重载版本使用了类型为 `true_type` 或 `false_type` 的无名参数。因为在函数体的内部没有使用这些参数，所以这些参数是无名的。这些参数仅用于函数重载解析。

这些代码的测试如下：

```
process(123);
process(2.2);
process("Test"s);
```

这个例子的输出如下：

```
123 is an integral type.
2.2 is a non-integral type.
Test is a non-integral type.
```

前面的例子只使用单个函数模板来编写，但没有说明如何使用类型 `trait`，以基于类型选择不同的重载。

```
template <typename T>
void process(const T& t)
{
    if constexpr (is_integral_v<T>) {
        cout << t << " is an integral type." << endl;
    }
}
```

```

    } else {
        cout << t << " is a non-integral type." << endl;
    }
}

```

2. 使用类型关系

有 3 种类型关系: `is_same`、`is_base_of` 和 `is_convertible`。下面将给出一个例子来展示如何使用 `is_same`。其余类型关系的工作原理类似。

下面的 `same()` 函数模板通过 `is_same` 类型 trait 特性判断两个给定参数是否类型相同, 然后输出相应的信息。

```

template <typename T1, typename T2>
void same(const T1& t1, const T2& t2)
{
    bool areTypesTheSame { is_same_v<T1, T2> };
    cout << format("'{}' and '{}' are {} types.", t1, t2,
        (areTypesTheSame ? "the same" : "different")) << endl;
}

int main()
{
    same(1, 32);
    same(1, 3.01);
    same(3.01, "Test"s);
}

```

输出如下所示:

```

'1' and '32' are the same types.
'1' and '3.01' are different types
'3.01' and 'Test' are different types

```

3. 使用条件类型 trait

第 18 章“标准库容器”解释了标准库辅助函数模板 `std::move_if_noexcept()`, 它可以根据移动构造函数是否标记为 `noexcept`, 来有条件地调用移动构造函数还是拷贝构造函数。标准库没有提供类似的辅助函数模板, 根据移动赋值运算符是否标记为 `noexcept`, 选择调用移动赋值运算符还是拷贝赋值运算符。现在已经了解了模板元编程和类型 trait, 来看看如何实现自己的 `move_assign_if_noexcept()`。

记得在第 18 章中, 如果移动构造函数标记为 `noexcept`, `move_if_noexcept()` 只会将给定的引用转换为右值引用, 否则将转换为 `const` 引用。`move_assign_if_noexcept()` 需要做类似的事情, 如果移动赋值运算符标记为 `noexcept`, 则将给定的引用转换为右值引用, 否则将转换为 `const` 引用。

`std::conditional` 类型 trait 可用于实现条件, 而 `is_nothrow_move_assignable` 类型 trait 可用于判断某个类型是否有标记为 `noexcept` 的移动赋值运算符。条件类型 trait 有 3 个模板参数: 一个布尔型, 一个布尔型为 `true` 的类型以及一个布尔型为 `false` 的类型。下面是整个函数模板:

```

template <typename T>
constexpr conditional<is_nothrow_move_assignable_v<T>, T&&, const T&>::type
    move_assign_if_noexcept(T& t) noexcept
{
    return move(t);
}

```

C++ 标准为具有类型成员(比如 `conditional`)的 trait 定义了别名模板。它们与 trait 具有相同的名称,

但是附加了 `_t`。例如，与其这样：

```
conditional<is_nothrow_move_assignable_v<T>, T&&, const T&>::type
```

可以这样写：

```
conditional_t<is_nothrow_move_assignable_v<T>, T&&, const T&>
```

可以对 `move_assign_if_noexcept()` 函数模板进行以下测试：

```
class MoveAssignable
{
public:
    MoveAssignable& operator=(const MoveAssignable&) {
        cout << "copy assign" << endl; return *this; }
    MoveAssignable& operator=(MoveAssignable&&) {
        cout << "move assign" << endl; return *this; }
};

class MoveAssignableNoexcept
{
public:
    MoveAssignableNoexcept& operator=(const MoveAssignableNoexcept&) {
        cout << "copy assign" << endl; return *this; }
    MoveAssignableNoexcept& operator=(MoveAssignableNoexcept&&) noexcept {
        cout << "move assign" << endl; return *this; }
};

int main()
{
    MoveAssignable a, b;
    a = move_assign_if_noexcept(b);
    MoveAssignableNoexcept c, d;
    c = move_assign_if_noexcept(d);
}
```

输出如下：

```
copy assign
move assign
```

4. 使用 `enable_if`

使用 `enable_if` 需要了解“替换失败不是错误” (Substitution Failure Is Not An Error, SFINAE) 特性，这是 C++ 中一个复杂晦涩的特性。它规定，为一组给定的模板参数特化函数模板失败不会被视为编译错误。相反，这样的特化应该从函数重载集合中移除。下面仅讲解 SFINAE 的基础知识。

如果有一组重载函数，就可以使用 `enable_if` 根据某些类型特性有选择地禁用某些重载。`enable_if` 通常用于重载函数组的返回类型。`enable_if` 接收两个模板类型参数。第一个参数是布尔值，第二个参数是默认为 `void` 的类型。如果布尔值是 `true`，`enable_if` 类就有一种可使用 `::type` 访问的嵌套类型，这种嵌套类型由第二个模板类型参数给定。如果布尔值是 `false`，就没有嵌套类型。

通过 `enable_if`，可将前面使用 `same()` 函数模板的例子重写为一个重载的 `checkType()` 函数模板。在这个版本中，`checkType()` 函数根据给定值的类型是否相同，返回 `true` 或 `false`。如果不希望 `checkType()` 返回任何内容，可删除 `return` 语句，可删除 `enable_if` 的第二个模板类型参数，或用 `void` 替换。

```
template <typename T1, typename T2>
enable_if_t<is_same_v<T1, T2>, bool>
```

```

    checkType(const T1& t1, const T2& t2)
{
    cout << format("{}{}' and '{}{}' are the same types.", t1, t2) << endl;
    return true;
}

template <typename T1, typename T2>
enable_if_t<!is_same_v<T1, T2>, bool>
    checkType(const T1& t1, const T2& t2)
{
    cout << format("{}{}' and '{}{}' are different types.", t1, t2) << endl;
    return false;
}

int main()
{
    checkType(1, 32);
    checkType(1, 3.01);
    checkType(3.01, "Test"s);
}

```

输出与前面的相同:

```

'1' and '32' are the same types.
'1' and '3.01' are different types.
'3.01' and 'Test' are different types.

```

上述代码定义了两个重载的 `checkType()`，它们的返回类型都是 `enable_if` 的嵌套类型 `bool`。首先，通过 `is_same_v` 检查两种类型是否相同，然后通过 `enable_if_t` 获得结果。当 `enable_if_t` 的第一个参数为 `true` 时，`enable_if_t` 的类型就是 `bool`；当第一个参数为 `false` 时，将不会有返回类型。这就是 SFINAE 发挥作用的地方。

当编译器开始编译 `main()` 函数的第一行时，它试图找到接收两个整型值的 `checkType()` 函数。编译器会在源代码中找到第一个重载的 `checkType()` 函数模板，并将 `T1` 和 `T2` 都设置为整数，以推断可使用这个模板的实例。然后，编译器会尝试确定返回类型。由于这两个参数是整数，因此是相同的类型，`is_same_v<T1, T2>` 将返回 `true`，这导致 `enable_if_t<true, bool>` 返回类型 `bool`。这样实例化时一切都很好，编译器可使用该版本的 `check_type()`。

然而，当编译器尝试编译 `main()` 函数的第二行时，编译器会再次尝试找到合适的 `checkType()` 函数。编译器从第一个 `checkType()` 开始，判断出可将 `T1` 设置为 `int` 类型，将 `T2` 设置为 `double` 类型。然后，编译器会尝试确定返回类型。这一次，`T1` 和 `T2` 是不同的类型，这意味着 `is_same_v<T1, T2>` 将返回 `false`。因此 `enable_if_t<false, bool>` 不表示类型，`checkType()` 函数不会有返回类型。编译器会注意到这个错误，但由于 SFINAE，还不会产生真正的编译错误。编译器将正常回溯，并试图找到另一个 `checkType()` 函数。这种情况下，第二个 `check_type()` 可以正常工作，因为 `is_same_v<T1, T2>` 为 `true`，此时 `enable_if_t<true, bool>` 返回类型 `bool`。

如果希望在一组构造函数上使用 `enable_if`，就不能将它用于返回类型，因为构造函数没有返回类型。此时可在带默认值的额外构造函数参数上使用 `enable_if`。

建议慎用 `enable_if`，仅在需要解析重载歧义时使用，即无法使用其他技术(例如特例化、`concepts` 等)解析重载歧义时使用。例如，如果只希望在对模板使用了错误类型时编译失败，应使用第 12 章讨论的 `concepts`，或者后面介绍的 `static_assert()`，而不是 SFINAE。当然，`enable_if` 有合法的用例。一个例子是为类似于自定义矢量的类特例化复制函数，使用 `enable_if` 和 `is_trivially_copyable` 类型 trait 对普

通的可复制类型执行按位复制(例如使用 C 函数 `memcpy()`)。

警告:

依赖于 `SFINAE` 是一件很棘手和复杂的事情。如果有选择地使用 `SFINAE` 和 `enable_if` 禁用重载集中的错误重载, 就会得到奇怪的编译错误, 这些错误很难跟踪。

5. 使用 `constexpr if` 简化 `enable_if` 结构

从前面的示例可以看到, 使用 `enable_if` 将十分复杂。某些情况下, C++17 引入的 `constexpr if` 特性有助于极大地简化 `enable_if`。

例如, 假设有以下两个类:

```
class IsDoable
{
public:
    void doit() const { cout << "IsDoable::doit()" << endl; }
};

class Derived : public IsDoable { };
```

可创建一个函数模板 `callDoit()`。如果方法可用, 它调用 `doit()` 方法; 否则在控制台上打印错误消息。为此, 可使用 `enable_if`, 检查给定类型是否从 `IsDoable` 派生。

```
template <typename T>
enable_if_t<is_base_of_v<IsDoable, T>, void> callDoit(const T& t)
{
    t.doit();
}

template <typename T>
enable_if_t<!is_base_of_v<IsDoable, T>, void> callDoit(const T&)
{
    cout << "Cannot call doit()!" << endl;
}
```

下面的代码对该实现进行测试:

```
Derived d;
callDoit(d);
callDoit(123);
```

输出如下:

```
IsDoable::doit()
Cannot call doit()!
```

使用 `constexpr if` 可极大地简化 `enable_if` 实现:

```
template<typename T>
void callDoit(const T& [[maybe_unused]] t)
{
    if constexpr(is_base_of_v<IsDoable, T>) {
        t.doit();
    } else {
        cout << "Cannot call doit()!" << endl;
    }
}
```

无法使用普通 if 语句做到这一点！使用普通 if 语句，两个分支都需要编译，而如果指定并非从 `IsDoable` 派生的类型 `T`，这将失败。此时，`t.doit()` 一行无法编译。但是，使用 `constexpr if` 语句，如果提供了并非从 `IsDoable` 派生的类型，`t.doit()` 一行甚至不会编译！

注意，这里使用了 `[[maybe_unused]]` 特性。如果给定类型 `T` 不是从 `IsDoable` 派生而来，`t.doit()` 行就不会编译。因此，在 `callDoit()` 的实例化中，不会使用参数 `t`。如果具有未使用的参数，大多数编译器会给出警告，甚至发生错误。该特性可阻止参数 `t` 的此类警告或错误。

不使用 `is_base_of` 类型 trait，也可使用 `is_invocable` trait，这个 trait 可用于确定在调用给定函数时是否可以使用一组给定的参数。下面是使用 `is_invocable` trait 的 `callDoit()` 实现：

```
template<typename T>
void callDoit(const T& [[maybe_unused]] t)
{
    if constexpr(is_invocable_v<decltype(&IsDoable::doit), T>) {
        t.doit();
    } else {
        cout << "Cannot call doit()!" << endl;
    }
}
```

6. 逻辑运算符 trait

在 3 种逻辑运算符 trait：串联(conjunction)、分离(disjunction)与否定(negation)。以 `_v` 结尾的可变模板也可供使用。这些 trait 接收可变数量的模板类型参数，可用于在类型 trait 上执行逻辑操作，如下所示。

```
cout << conjunction_v<is_integral<int>, is_integral<short>> << " ";
cout << conjunction_v<is_integral<int>, is_integral<double>> << " ";

cout << disjunction_v<is_integral<int>, is_integral<double>,
    is_integral<short>> << " ";

cout << negation_v<is_integral<int>> << " ";
```

输出如下所示：

```
1 0 1 0
```

7. 静态断言

`static_assert` 允许在编译期对断言求值。断言需要是 `true`，如果断言是 `false`，编译器就会报错。`static_assert` 调用接收两个参数：编译期求值的表达式和字符串。当表达式计算为 `false` 时，编译器将给出包含指定字符串的错误提示。下例核实是否在使用 64 位编译器进行编译：

```
static_assert(sizeof(void*) == 8, "Requires 64-bit compilation.");
```

如果编译期使用 32 位编译器，指针是 4 个字符，编译器将给出错误提示，如下所示：

```
test.cpp(3): error C2338: Requires 64-bit compilation.
```

从 C++17 开始，字符串参数变为可选的，如下所示：

```
static_assert(sizeof(void*) == 8);
```

此时，如果表达式的计算结果是 `false`，将得到与编译器相关的错误消息。例如，Microsoft Visual C++ 会给出以下错误提示：

test.cpp(3): error C2607: static assertion failed

`static_assert` 可以和类型 `trait` 结合使用。示例如下:

```
template <typename T>
void foo(const T& t)
{
    static_assert(is_integral_v<T>, "T should be an integral type.");
}
```

然而, 推荐使用 C++20 的 `concepts` 替代带有类型 `traits` 的 `static_assert()`。例如:

```
template <integral T>
void foo(const T& t) { }
```

或者使用 C++20 简化的函数模板语法:

```
void foo(const integral auto& t) { }
```

26.6.5 模板元编程结论

模板元编程是一个功能非常强大的工具, 但也非常复杂。模板元编程有一个此前没有提到的问题, 那就是由于一切都发生在编译期, 因此不能通过调试器来定位问题。如果决定在代码中使用模板元编程, 一定要编写适当的注释来准确解释代码的作用, 以及解释为什么要这么做。如果没有为模板元编程的代码正确记录文档, 那么别人可能很难理解这些代码, 甚至自己都可能在未来无法理解这些代码。

26.7 本章小结

本章延续了第 12 章对模板的讨论。我们讲解了如何通过模板进行泛型编程, 以及如何使用模板元编程执行编译期计算。希望读者能理解并喜欢这些特性的强大功能, 知道怎样在自己的代码中应用这些概念。初次阅读时不用担心不能理解所有的语法或例子。第一次接触这些概念时, 可能会感觉这些概念很难, 在想要编写较复杂的模板时, 会发现语法很棘手。真正编写模板类或函数时, 可参考本章和第 12 章, 找到合适的语法。

26.8 练习

通过完成下面的习题, 可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题, 可以在看网站上的答案之前, 先重新阅读本章的部分内容, 尝试着自己找到答案。

练习 26-1 在习题 12-2 中, 为 `const char*` 的 `key` 和 `value` 编写了 `KeyValuePair` 类模板的全特化, 将全特化改为偏特化, 其中只有 `value` 是 `const char*` 类型。

练习 26-2 在编译期使用模板递归计算斐波那契数列的第 `n` 个数字, 斐波那契数列从 0 到 1 开始, 任何后面的值都是前面两个值的和, 所以有: 0、1、1、2、3、5、8、13、21、34、55、89 等等。

练习 26-3 参考习题 26-2 的解决方案并对其修改, 使计算仍在编译期进行, 但不需要使用任何模板和函数递归。

练习 26-4 编写一个名为 `push_back_values()` 的可变函数模板, 接收 `vector` 的引用和可变数量的值。函数应该使用折叠表达式将可变数量的值放到给定的 `vector` 中。

第27章

C++多线程编程

本章内容

- 多线程编程的含义
- 如何启动多线程
- 如何取消线程
- 如何从线程检索结果
- 死锁和争用条件的含义，以及如何利用互斥避免它们
- 如何使用原子类型和原子操作
- 条件变量的含义
- 如何使用 semaphore、latch 和 barrier
- 如何为线程内通信使用 future 和 promise
- 线程池的含义
- 可恢复函数或协程的含义

在多处理器的计算机系统中，多线程编程非常重要，允许编写并行利用所有处理器的程序。系统可通过多种方式获得多个处理器单元。系统可有多个处理器芯片，每个芯片都是一个独立的 CPU(中央处理单元)，系统也可只有一个处理器芯片，但该芯片内部由多个独立的 CPU(也称为核心)组成。这些处理器称为多核处理器。系统也可可是上述两种方式的组合。尽管具有多个处理器单元的系统已经存在了很长一段时间，然而，它们很少在消费系统中使用。今天，所有主要的 CPU 供应商都在销售多核处理器。如今，从服务器到 PC，甚至智能手机都在使用多核处理器。由于这种多核处理器的流行，编写多线程的应用程序变得越来越重要。专业的 C++程序员需要知道如何编写正确的多线程代码，以充分利用所有可用的处理器单元。多线程应用程序的编写曾经依赖平台和操作系统相关的 API。这使跨平台的多线程编程很困难。C++11 引入了一个标准的线程库，从而解决了这个问题。

多线程编程是一个复杂主题。本章讲解利用标准的线程库进行多线程编程，但由于篇幅受限，不可能涉及所有细节。市场上有一些关于多线程编程的专业图书。如果你对更深入的细节感兴趣，请参阅附录 B 的“多线程”部分列出的参考文献。

还可使用其他第三方 C++库，尽量编写平台独立的多线程程序，例如 pthreads 库和 boost::thread 库。然而，由于这些库不属于 C++标准的一部分，因此本书不予讨论。

27.1 多线程编程概述

通过多线程编程可并行地执行多个计算，这样可以充分利用当今大部分系统中的多个处理器单元。几十年前，CPU 市场竞争的是最高频率，对于单线程的应用程序来说主频非常重要。到 2005 年前后，由于电源管理和散热管理的问题，这种竞争已经停止。如今 CPU 市场竞争的是单个处理器芯片中的最多核心数目。在撰写本书时，双核和四核 CPU 已经非常普遍了，也有消息说要发布 12 核、16 核、18 核甚至更多核的处理器。

同样，看一下显卡中称为 GPU 的处理器，你会发现，它们是大规模并行处理器。今天，高端显卡已经拥有 4000 多个核心，这个数目还会高速增加。这些显卡不只用于游戏，还能执行计算密集型任务，例如图像和视频处理、蛋白质折叠(用于发现新的药物)和 SETI(Search for Extra-Terrestrial Intelligence, 搜寻地外智慧生命)项目中的信号处理等。

C++ 98/03 不支持多线程编程，所以必须借助第三方库或目标操作系统中的多线程 API。自 C++11 开始，C++ 有了一个标准的多线程库，使编写跨平台的多线程应用程序变得更容易了。目前的 C++ 标准仅针对 CPU，不适用于 GPU，这种情形将来可能会改变。

有两个原因促使我们应该开始编写多线程代码。首先，假设有一个计算问题，可将它分解为可互相独立运行的小块，那么在多处理器单元上运行可获得巨大的性能提升。其次，可在正交轴上对计算任务模块化；例如在线程中执行长时间的计算，而不会阻塞 UI 线程，这样在后台进行长时间计算时，用户界面仍然可以响应。

图 27-1 展示了一个非常适合并行运行的例子。图像像素处理算法不需要相邻像素的信息。该算法可将图像分为四个部分。在单核处理器上，每个部分都顺序处理；在双核处理器上，两个部分可以并行处理；在四核处理器上，四个部分可以并行处理，性能随着核心的数目而线性伸缩。

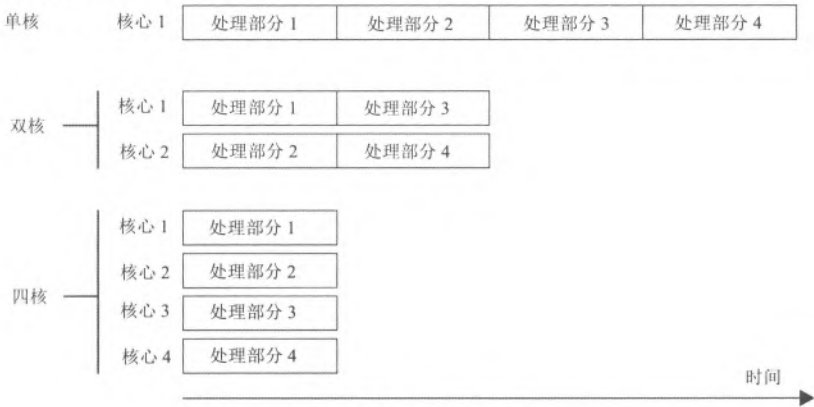


图 27-1 适合并行运行的示例

当然，并不总能将问题分解为可互相独立且并行执行的部分。但至少通常可将问题部分并行化，从而提升性能。多线程编程中的一个难点是将算法并行化，这个过程和算法的类型高度相关。其他困难之处是防止争用条件、死锁、撕裂和伪共享等。这些都可以使用原子或显式的同步机制来解决，参见本章后面的内容。

警告：

为避免这些多线程问题，应该设计程序，使多个线程不需要读写共享的内存位置。还可使用本章 27.3 节“原子操作库”中描述的原子操作，或使用 27.4 节“互斥”中描述的同步方法。

27.1.1 争用条件

当多个线程要访问任何种类的共享资源时，可能发生争用条件。共享内存上下文的争用条件称为“数据争用”。当多个线程访问共享的内存，且至少有一个线程写入共享的内存时，就会发生数据争用。例如，假设有一个共享变量，一个线程递增该变量的值，而另一个线程递减其值。递增和递减这个值，意味着需要从内存中获取当前值，递增或递减后再将结果保存回内存。在较旧的架构中，例如 PDP-11 和 VAX，这是通过一条原子的 INC 处理器指令完成的。在现代 x86 处理器中，INC 指令不再是原子的，这意味着在这个操作中，可执行其他指令，这可能导致代码获取错误值。

表 27-1 展示了递增线程在递减线程开始之前结束的结果，假设初始值是 1。

表 27-1 初始值是 1

线程 1(递增)	线程 2(递减)
加载值(值=1)	
递增值(值=2)	
存储值(值=2)	
	加载值(值=2)
	递减值(值=1)
	存储值(值=1)

存储在内存中的最终值是 1。当递减线程在递增线程开始之前完成时，最终值也是 1，如表 27-2 所示。

表 27-2 最终值是 1

线程 1(递增)	线程 2(递减)
	加载值(值=1)
	递减值(值=0)
	存储值(值= 0)
加载值(值=0)	
递增值(值=1)	
存储值(值=1)	

然而，当指令交错执行时，结果是不同的，如表 27-3 所示。

表 27-3 结果不同

线程 1(递增)	线程 2(递减)
加载值(值=1)	
递增值(值=2)	
	加载值(值=1)
	递减值(值=0)
存储值(值=2)	
	存储值(值=0)

这种情况下，最终结果是 0。换句话说，递增操作的结果丢失了。这是一个争用条件。

27.1.2 撕裂

撕裂(tearing)是数据争用的特例或结果。有两种撕裂类型：撕裂读和撕裂写。如果线程已将数据的一部分写入内存，但还有部分数据没有写入，此时读取数据的其他任何线程将看到不一致的数据，发生撕裂读。如果两个线程同时写入数据，其中一个线程可能写入数据的一部分，而另一个线程可能写入数据的另一部分，最终结果将不一致，发生撕裂写。

27.1.3 死锁

如果选择使用互斥等同步方法解决争用条件的问题，那么可能遇到多线程编程的另一个常见问题：死锁。死锁指的是两个线程因为等待访问另一个阻塞线程锁定的资源而造成无限阻塞，这也可扩展到超过两个线程的情形。例如，假设有两个线程想要访问某共享资源，它们必须拥有权限才能访问该资源。如果其中一个线程当前拥有访问该资源的权限，但由于其他一些原因而被无限期阻塞，那么此时，试图获取同一资源权限的另一个线程也将无限期阻塞。获得共享资源权限的一种机制是互斥对象，见稍后的讨论。例如，假设有两个线程和两种资源(由两个互斥对象 A 和 B 保护)。这两个线程获取这两种资源的权限，但它们以不同的顺序获得权限。表 27-4 以伪代码形式展示了这种现象。

表 27-4 伪代码形式

线程 1	线程 2
获取 A	获取 B
获取 B	获取 A
// ...计算	// ...计算
释放 B	释放 A
释放 A	释放 B

现在设想两个线程中的代码按如下顺序执行。

- 线程 1：获取 A
- 线程 2：获取 B
- 线程 1：获取 B(等待/阻塞，因为 B 被线程 2 持有)
- 线程 2：获取 A(等待/阻塞，因为 A 被线程 1 持有)

现在两个线程都在无限期地等待，这就是死锁情形。图 27-2 是这种死锁情形的图形表示。线程 1 拥有资源 A 的访问权限，并正在等待获取资源 B 的访问权限。线程 2 拥有资源 B 的访问权限，并且正在等待资源 A 的访问权限。在这种图形表示中，可以看到一个表示死锁情形的环。这两个线程将无限期地等待。

最好总是以相同的顺序获得权限，以避免这种死锁。也可在程序中包含打破这类死锁的机制。一种可行的方法是试图等待一定的时间，看看能否获得某个资源的权限。如果不能在某个时间间隔内获得这个权限，那么线程停止等待，并释放当前持有的其他锁。线程可能睡眠一小段时间，然后重新尝试获取需要的所有资源。这种方法也可能给其他线程获得必要的锁并继续执行的机会。这种方法是否可用在很大程度上取决于特定的死锁情形。

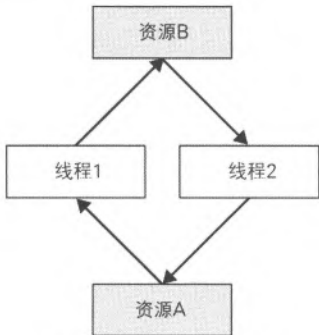


图 27-2 死锁情形的图形表示

不要使用前一段中描述的那种变通方法，而是应该尝试避免任何可能的死锁情形。如果需要获得由多个互斥对象保护的多个资源的权限，而非单独获取每个资源的权限，推荐使用 23.4 节描述的标准的 `std::lock()`或 `std::try_lock()`函数。这两个函数会通过一次调用获得或尝试获得多个资源的权限。

27.1.4 伪共享

大多数缓存都使用所谓的“缓存行(cache line)”。对于现代 CPU 而言，缓存行通常是 64 个字节。如果需要将一些内容写入缓存行，则需要锁定整行。如果代码结构设计不当，对于多线程代码而言，这会带来严重的性能问题。例如，假设有两个线程正在使用数据的两个不同部分，而那些数据共享一个缓存行，如果其中一个线程写入一些内容，那么将阻塞另一个线程，因为整个缓存行都被锁定。图 27-3 展示了这样的情况，两个线程在共享一个缓存行的同时写入两个不同的内存块。

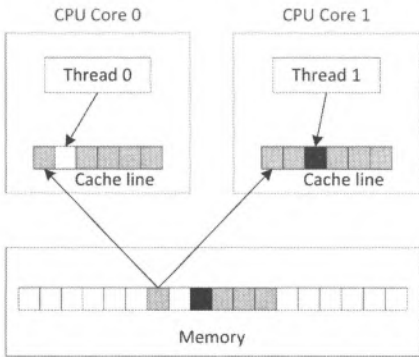


图 27-3 两个线程写入两个不同的内存块

可使用显式的内存对齐(memory alignment)方式优化数据结构，确保由多个线程处理的数据不共享任何缓存行。为了以便携方式做到这一点，C++17 引入了 `hardware_destructive_interference_size` 常量，该常量在 `<new>` 中定义，为避免共享缓存行，返回两个并发访问的对象之间的建议偏移量。可将这个值与 `alignas` 关键字结合使用，以合理地对齐数据。

27.2 线程

借助在 `<thread>` 中定义的 C++线程库，启动新的线程将变得非常容易。可通过多种方式指定新线程中需要执行的内容。可让新线程执行全局函数、函数对象的 `operator()`、`lambda` 表达式甚至某个

类实例的成员函数。

27.2.1 通过函数指针创建线程

像 Windows 上的 `CreateThread()`、`_beginthread()` 等函数, 以及 pthreads 库中的 `pthread_create()` 函数, 都要求线程函数只有一个参数。另一方面, 标准 C++ 的 `std::thread` 类使用的函数可以有任意数量的参数。

假设 `counter()` 函数接收两个整数: 第一个表示 ID, 第二个表示这个函数要循环的迭代次数。函数体是一个循环, 这个循环执行给定次数的迭代。在每次迭代中, 向标准输出打印一条消息:

```
void counter(int id, int numIterations)
{
    for (int i { 0 }; i < numIterations; ++i) {
        cout << "Counter " << id << " has value " << i << endl;
    }
}
```

可通过 `std::thread` 启动执行此函数的多个线程。可创建线程 `t1`, 使用参数 1 和 6 执行 `counter()`:

```
thread t1 { counter, 1, 6 };
```

`thread` 类的构造函数是一个可变参数模板, 也就是说, 可接收任意数目的参数。第 26 章详细讨论了可变参数模板。第一个参数是新线程要执行的函数的名称。当线程开始执行时, 将随后可变数目的参数传递给这个函数。

如果一个线程对象表示系统当前或过去的某个活动线程, 则认为它是可结合的(joinable)。即使这个线程执行完毕, 该线程对象也依然处于可结合状态。默认构造的线程对象是不可结合的。在销毁一个可结合的线程对象前, 必须调用其 `join()` 或 `detach()` 方法。对 `join()` 的调用是阻塞调用, 会一直等到线程完成工作为止。调用 `detach()` 时, 会将线程对象与底层 OS 线程分离。此时, OS 线程将继续独立运行。调用这两个方法时, 都会导致线程变得不可结合。如果一个仍可结合的线程对象被销毁, 析构函数会调用 `std::terminate()`, 这会突然间终止所有线程以及应用程序本身。

下面的代码启动两个线程来执行 `counter()` 函数。启动线程后, `main()` 调用这两个线程的 `join()` 方法。

```
thread t1 { counter, 1, 6 };
thread t2 { counter, 2, 4 };
t1.join();
t2.join();
```

这个示例的可能输出如下所示:

```
Counter 2 has value 0
Counter 1 has value 0
Counter 1 has value 1
Counter 1 has value 2
Counter 1 has value 3
Counter 1 has value 4
Counter 1 has value 5
Counter 2 has value 1
Counter 2 has value 2
Counter 2 has value 3
```

不同系统上的输出会有所不同, 很可能每次运行的结果都不同。这是因为两个线程同时执行 `counter()` 函数, 所以输出取决于系统中处理核心的数量以及操作系统的线程调度。

默认情况下,从不同线程访问 `cout` 是线程安全的,没有任何数据争用,除非在第一个输出或输入操作之前调用了 `cout.sync_with_stdio(false)`。然而,即使没有数据争用,来自不同线程的输出仍然可以交错。这意味着,前面示例的输出可能会混合在一起:

```
Counter Counter 2 has value 0
1 has value 0
Counter 1 has value 1
Counter 1 has value 2
...
```

这个问题可通过本章后面讨论的同步方法加以纠正。

注意:

线程函数的参数总是被复制到线程的某个内部存储中。通过`<functional>`中的 `std::ref()`或 `cref()`按引用传递参数。

27.2.2 通过函数对象创建线程

不使用函数指针,也可以使用函数对象在线程中执行。前面的小节使用函数指针技术,给线程传递信息的唯一方式是给函数传递参数。而使用函数对象,可向函数对象类添加成员变量,并可以采用任何方式初始化和使用这些变量。下例首先定义 `Counter` 类。这个类有两个成员变量:一个表示 ID,另一个表示循环迭代次数。这两个成员变量都通过类的构造函数进行初始化。为让 `Counter` 类成为函数对象,根据第 19 章的讨论,需要实现 `operator()`。`operator()`的实现和 `counter()`函数一样:

```
class Counter
{
public:
    Counter(int id, int numIterations)
        : m_id { id }, m_numIterations { numIterations } { }

    void operator()() const
    {
        for (int i { 0 }; i < m_numIterations; ++i) {
            cout << "Counter " << m_id << " has value " << i << endl;
        }
    }

private:
    int m_id;
    int m_numIterations;
};
```

下面的代码片段演示了通过函数对象初始化线程的两种方法。第一种方法使用了统一初始化语法。通过构造函数参数创建 `Counter` 类的一个实例,然后把这个实例放在花括号中,传递给 `thread` 类的构造函数。

第二种方法定义了 `Counter` 类的一个命名实例,并将它传递给 `thread` 类的构造函数。

```
// Using uniform initialization syntax.
thread t1 { Counter{ 1, 20 } };

// Using named variable.
Counter c { 2, 12 };
thread t2 { c };
```

```
// Wait for threads to finish.
t1.join();
t2.join();
```

注意:

函数对象总是被复制到线程的某个内部存储中。如果要在函数对象的某个特定实例上执行 `operator()` 而非进行复制, 那么应该使用 `<functional>` 中的 `std::ref()` 或 `cref()`, 通过引用传入该实例。

```
Counter c { 2, 12 };
thread t2 { ref(c) };
```

27.2.3 通过 lambda 创建线程

lambda 表达式能很好地用于标准 C++ 线程库。下例启动一个线程来执行给定的 lambda 表达式:

```
int main()
{
    int id { 1 };
    int numIterations { 5 };
    thread t1 { [id, numIterations] {
        for (int i { 0 }; i < numIterations; ++i) {
            cout << "Counter " << id << " has value " << i << endl;
        }
    } };
    t1.join();
}
```

27.2.4 通过成员函数创建线程

还可在线程中指定要执行的类的成员函数。下例定义了带有 `process()` 方法的基类 `Request`。 `main()` 函数创建 `Request` 类的一个实例, 并启动一个新的线程, 这个线程执行 `Request` 实例 `req` 的 `process()` 成员函数:

```
class Request
{
public:
    Request(int id) : m_id { id } { }
    void process() { cout << "Processing request " << m_id << endl; }
private:
    int m_id;
};

int main()
{
    Request req { 100 };
    thread t { &Request::process, &req };
    t.join();
}
```

通过这种技术, 可在不同线程中执行某个对象中的方法。如果有其他线程访问同一个对象, 那么需要确认这种访问是线程安全的, 以避免争用条件。本章稍后讨论的互斥可用作实现线程安全的同步机制。

27.2.5 线程本地存储

C++标准支持线程本地存储的概念。通过关键字 `thread_local`, 可将任何变量标记为线程本地数据, 即每个线程都有这个变量的独立副本, 而且这个变量能在线程的整个生命周期中持续存在。对于每个线程, 该变量正好初始化一次。例如, 在下面的代码中定义了两个全局变量; 每个线程都共享唯一的 `k` 副本, 而且每个线程都有自己的 `n` 副本:

```
int k;
thread_local int n;
```

下面的代码片段验证了这一点。`threadFunction()`将 `k` 和 `n` 的当前值输出到控制台, 然后将它们都加 1。`main()`函数启动第一个线程, 等待它完成, 然后启动第二个线程。

```
void threadFunction(int id)
{
    cout << format("Thread {}: k={}, n={}\\n", id, k, n);
    ++n;
    ++k;
}

int main()
{
    thread t1 { threadFunction, 1 }; t1.join();
    thread t2 { threadFunction, 2 }; t2.join();
}
```

从下面的输出可以清楚地看出, 所有线程只共享一个 `k` 实例, 而每个线程都有自己的 `n` 的拷贝。

```
Thread 1: k=0, n=0
Thread 2: k=1, n=0
```

注意, 如果 `thread_local` 变量在函数作用域内声明, 那么这个变量的行为和声明为静态变量是一致的, 只不过每个线程都有自己独立的副本, 而且不论这个函数在线程中调用多少次, 每个线程仅初始化这个变量一次。

27.2.6 取消线程

C++标准没有包含在一个线程中取消另一个已运行线程的任何机制。一种解决方案是使用 C++20 的 `jthread` 类(27.2.7 节讨论), 如果不能选择这种, 那么实现这一目标的最好方法是提供两个线程都支持的某种通信机制。最简单的机制是提供一个共享变量, 目标线程定期检查这个变量, 判断是否应该终止。其他线程可设置这个共享变量, 间接指示线程关闭。这里必须注意, 因为是由多个线程访问这个共享变量, 其中至少有一个线程向共享变量写入内容。建议使用本章后面讨论的原子变量或条件变量。

C++20

27.2.7 自动 join 线程

如前所述, 如果销毁了仍可以 `joinable` 的线程实例, C++运行时调用 `std::terminate()`来终止应用程序。C++20 引入了 `std::jthread`, 同样在 `<thread>` 中定义。 `jthread` 实际上等同于 `thread`, 除了:

- 在析构函数中自动 `join`。
- 支持所谓的协作式取消。

它被称为协作式取消,因为支持取消的线程需要定期检查它是否需要取消自己。在给出示例之前,需要引入两个关键类,它们都定义在`<stop_token>`中。

- **std::stop_token:** 支持主动检查取消的请求。一个可取消线程需要定期在 `stop_token` 上调用 `stop_requested()`, 以确定是否需要停止它的工作。`stop_token` 可以和 `condition_variable_any` 一起使用, 这样线程在需要停止时就可以被唤醒。
- **std::stop_source:** 用于请求线程取消执行。通过调用 `stop_source` 上的 `request_stop()` 方法来完成。如果 `stop_source` 被用于请求取消, 那么该停止请求对所有相关的 `stop_source` 和 `stop_token` 都可见。`stop_requested()` 方法可以用来检查是否已经请求了停止。

看个示例。下面的代码创建一个 `jthread` 来执行给定的 `lambda` 表达式。传递给 `jthread` 的可调用对象可以有 `stop_token` 作为第一个参数。可调用对象的主题可以使用 `stop_token` 来确定它是否需要取消自己。

```
jthread job { [](stop_token token) {
    while (!token.stop_requested()) {
        //...
    }
} };
```

在另一个线程中, 可以请求这个线程取消自己, 如下所示:

```
job.request_stop();
```

要从 `jthread` 中直接访问 `stop_token` 和 `stop_source`, 可以使用 `get_stop_token()` 和 `get_stop_source()` 方法。

27.2.8 从线程获得结果

如前面的例子所示, 启动新线程十分容易。然而, 大多数情况下, 你可能更感兴趣的是线程产生的结果。例如, 如果一个线程执行了一些数学计算, 你肯定想在执行结束时从这个线程获得计算结果。一种方法是向线程传入指向结果变量的指针或引用, 线程将结果保存在其中。另一种方法是将结果存储在函数对象的类成员变量中, 线程执行结束后可获得结果值。使用 `std::ref()`, 将函数对象按引用传递给 `thread` 构造函数时, 这才能生效。

然而, 还有一种更简单的方法可从线程获得结果: `future`。通过 `future` 也能更方便地处理线程中发生的错误。`future` 将在本章稍后讨论。

27.2.9 复制和重新抛出异常

整个异常机制在 C++ 中工作得很好, 当然这仅限于单线程的情况。每个线程都可抛出自己的异常, 但它们必须在自己的线程内捕获异常。如果一个线程抛出的异常不能在另一个线程中捕获, C++ 运行库将调用 `std::terminate()`, 从而终止整个应用程序。从一个线程抛出的异常不能在另一个线程中捕获。当希望将异常处理机制和多线程编程结合在一起时, 这会引入不少问题。

不使用标准线程库, 就很难在线程间正常地处理异常, 甚至根本办不到。标准线程库通过以下和异常相关的函数解决了这个问题。这些函数不仅可用于 `std::exception`, 还可以用于所有类型的异常: `int`、`string`、自定义异常等。

- `exception_ptr current_exception() noexcept;`

这个函数在 `catch` 块中调用, 返回一个 `exception_ptr` 对象, 这个对象引用目前正在处理的异常或其副本。如果没有处理异常, 则返回空的 `exception_ptr` 对象。只要存在引用异常对象的 `exception_ptr`

类型的对象，引用的异常对象就是可用的。`exception_ptr` 对象的类型是 `NullablePointer`，这意味着这个变量很容易通过简单的 `if` 语句来检查，详见后面的示例。

- `[[noreturn]] void rethrow_exception(exception_ptr p);`

这个函数重新抛出由 `exception_ptr` 参数引用的异常。未必在最开始生成引用异常的那个线程中重新抛出这个异常，因此这个特性特别适合于跨不同线程的异常处理。`[[noreturn]]`特性表示这个函数绝不会正常地返回。

- `template<class E> exception_ptr make_exception_ptr(E e) noexcept;`

这个函数创建一个引用给定异常对象副本的 `exception_ptr` 对象。这实际上是以下代码的简写形式：

```
try { throw e; }
catch(...) { return current_exception(); }
```

下面看一下如何通过这些函数实现不同线程间的异常处理。下面的代码定义了一个函数，这个函数完成一些事情并抛出异常。这个函数最终将运行在一个独立的线程中：

```
void doSomeWork()
{
    for (int i { 0 }; i < 5; ++i) {
        cout << i << endl;
    }
    cout << "Thread throwing a runtime_error exception..." << endl;
    throw runtime_error { "Exception from thread" };
}
```

下面的 `threadFunc()` 函数将上述函数包装在一个 `try/catch` 块中，捕获 `doSomeWork()` 可能抛出的所有异常。为 `threadFunc()` 传入一个参数，其类型为 `exception_ptr&`。一旦捕获到异常，就通过 `current_exception()` 函数获得正在处理的异常的引用，然后将引用赋给 `exception_ptr` 参数。之后，线程正常退出：

```
void threadFunc(exception_ptr& err)
{
    try {
        doSomeWork();
    } catch (...) {
        cout << "Thread caught exception, returning exception..." << endl;
        err = current_exception();
    }
}
```

以下 `doWorkInThread()` 函数在主线程中调用，其职责是创建一个新的线程，并开始在这个线程中执行 `threadFunc()` 函数。对类型为 `exception_ptr` 的对象的引用被作为参数传入 `threadFunc()`。一旦创建了线程，`doWorkInThread()` 函数就使用 `join()` 方法等待线程执行完毕，之后检查 `error` 对象。由于 `exception_ptr` 的类型为 `NullablePointer`，因此很容易通过 `if` 语句进行检查。如果是一个非空值，则在当前线程中重新抛出异常，在这个例子中，当前线程即主线程。在主线程中重新抛出异常，异常就从一个线程转移到另一个线程。

```
void doWorkInThread()
{
    exception_ptr error;
    // Launch thread.
    thread t { threadFunc, ref(error) };
```

```

    // Wait for thread to finish.
    t.join();
    // See if thread has thrown any exception.
    if (error) {
        cout << "Main thread received exception, rethrowing it..." << endl;
        rethrow_exception(error);
    } else {
        cout << "Main thread did not receive any exception." << endl;
    }
}

```

`main()`函数相当简单。它调用 `doWorkInThread()`，将这个调用包装在一个 `try/catch` 块中，捕获由 `doWorkInThread()`创建的任何线程抛出的异常：

```

int main()
{
    try {
        doWorkInThread();
    } catch (const exception& e) {
        cout << "Main function caught: '" << e.what() << "'" << endl;
    }
}

```

输出如下所示：

```

0
1
2
3
4
Thread throwing a runtime_error exception...
Thread caught exception, returning exception...
Main thread received exception, rethrowing it...
Main function caught: 'Exception from thread'

```

为让这个例子紧凑且更容易理解，`main()`函数通常使用 `join()`阻塞主线程，并等待线程完成。当然，在实际的应用程序中，你不想阻塞主线程。例如，在 GUI 应用程序中，阻塞主线程意味着 UI 失去响应。此时，可使用消息传递范型在线程之间通信。例如，可让前面的 `threadFunc()`函数给 UI 线程发送一条消息，消息的参数为 `current_exception()`结果的一份副本。但即使如此，如前所述，也需要确保在任何生成的线程上调用 `join()`或 `detach()`。

27.3 原子操作库

原子类型允许原子访问，这意味着不需要额外的同步机制就可执行并发的读写操作。没有原子操作，递增变量就不是线程安全的，因为编译器首先将值从内存加载到寄存器中，递增后再把结果保存回内存。另一个线程可能在这个递增操作的执行过程中接触到内存，导致数据争用。例如，下面的代码不是线程安全的，包含数据争用条件。这种争用条件在本章开头讨论过：

```

int counter { 0 };           // Global variable
...
++counter;                   // Executed in multiple threads

```

为使这个线程安全且不显式地使用任何同步机制，可使用 `std::atomic` 类型。下面是使用原子整数

的相同代码：

```
atomic<int> counter { 0 } ; // Global variable
...
++counter;                  // Executed in multiple threads
```

这些原子类型都定义在<atomic>中。C++标准为所有基本类型定义了命名的整型原子类型，如表 27-5 所示。

表 27-5 命名的整型原子类型

命名的原子类型	等效的 std::atomic 类型
atomic_bool	atomic<bool>
atomic_char	atomic<char>
atomic_uchar	atomic<unsigned char>
atomic_int	atomic<int>
atomic_uint	atomic<unsigned int>
atomic_long	atomic<long>
atomic_ulong	atomic<unsigned long>
atomic_llong	atomic<long long>
atomic_ullong	atomic<unsigned long long>
atomic_wchar_t	atomic<wchar_t>
atomic_flag	(none)

可使用原子类型，而不显式使用任何同步机制。但在底层，某些类型的原子操作可能使用同步机制(如互斥对象)。如果目标硬件缺少以原子方式执行操作的指令，则可能发生这种情况。可在原子类型上使用 is_lock_free()方法来查询它是否支持无锁操作；所谓无锁操作，是指在运行时，底层没有显式的同步机制。

可将 std::atomic 类模板与所有类型一起使用，并非仅限于整数类型。例如，可创建 atomic<double> 或 atomic<MyType>，但这要求 MyType 具有 is_trivially_copy 特点。底层可能需要显式的同步机制，具体取决于指定类型的大小。在下例中，Foo 和 Bar 具有 is_trivially_copy 特点，即 std::is_trivially_copyable_v 都等于 true。但 atomic<Foo>并非无锁操作，而 atomic<Bar>是无锁操作。

```
class Foo { private: int mArray[123]; };
class Bar { private: int mInt; };

int main()
{
    atomic<Foo> f;
    // Outputs: 1 0
    cout << is_trivially_copyable_v<Foo> << " " << f.is_lock_free() << endl;

    atomic<Bar> b;
    // Outputs: 1 1
    cout << is_trivially_copyable_v<Bar> << " " << b.is_lock_free() << endl;
}
```

在多线程中访问一段数据时，原子也可解决内存排序、编译器优化等问题。基本上，不使用原子或显式的同步机制，就不可能安全地在多线程中读写同一段数据。

注意:

内存序是访问内存的顺序。在没有任何原子和其他同步方法的情况下,只要不影响结果,编译器和硬件就可以重新对内存访问进行排序,这也称为 **as-if** 规则。在多线程环境中这可能会造成问题。

`atomic_flag` 是原子布尔值, C++标准保证了它总是无锁的。它与 `atomic<boo>` 的区别在于它不提供加载和存储值的方法。本章后面的互斥部分给出了一个使用 `atomic_flag` 实现自旋锁的示例。

27.3.1 原子操作

C++标准定义了一些原子操作。本节描述其中一些操作。完整的清单请参阅标准库参考资源(见附录 B)。

下面是一个原子操作示例:

```
bool atomic<T>::compare_exchange_strong(T& expected, T desired);
```

这个操作以原子方式实现了以下逻辑,伪代码如下:

```
if (*this == expected) {
    *this = desired;
    return true;
} else {
    expected = *this;
    return false;
}
```

这个逻辑初看起来令人感到陌生,但这是编写无锁并发数据结构的关键组件。无锁并发数据结构允许不使用任何同步机制来操作数据。但实现此类数据结构是一个高级主题,超出了本书的讨论范围。

另一个例子是 `atomic<T>::fetch_add()`。这个操作获取该原子类型的当前值,将给定的递增值添加到这个原子值,然后返回未递增的原始值。例如:

```
atomic<int> value { 10 };
cout << "Value = " << value << endl;
int fetched { value.fetch_add(4) };
cout << "Fetched = " << fetched << endl;
cout << "Value = " << value << endl;
```

如果没有其他线程操作 `fetched` 和 `value` 变量的内容,那么输出如下:

```
Value = 10
Fetched = 10
Value = 14
```

整型原子类型支持以下原子操作: `fetch_add()`、`fetch_sub()`、`fetch_and()`、`fetch_or()`、`fetch_xor()`、`++`、`--`、`+=`、`-=`、`&=`、`^=`和`|=`。原子指针类型支持 `fetch_add()`、`fetch_sub()`、`++`、`--`、`+=`和`-=`。

注意:

在 C++20 之前,对浮点类型使用 `std::atomic`,例如 `atomic<float>`和`atomic<double>`提供了原子的读写操作,但没有提供原子的算术操作。C++20 为浮点原子类型添加了 `fetch_add()`和`fetch_sub()`的支持。

大部分原子操作可接收一个额外参数,用于指定想要的内存顺序。例如:

```
T atomic<T>::fetch_add(T value, memory_order = memory_order_seq_cst);
```

可改变默认的 `memory_order`。C++标准提供了 `memory_order_relaxed`、`memory_order_consume`、`memory_order_acquire`、`memory_order_release`、`memory_order_acq_rel` 和 `memory_order_seq_cst`，这些都定义在 `std` 名称空间中。然而，很少有必要使用默认之外的顺序。尽管其他内存顺序可能会比默认顺序性能好，但根据一些标准，使用稍有不当地，就有可能再次引入争用条件或其他和线程相关的问题。如果需要了解有关内存顺序的更多信息，请参阅附录 B 中关于多线程的参考文献。

C++20

27.3.2 原子智能指针

C++20 通过 `<memory>` 引入了对 `atomic<std::shared_ptr<T>>` 的支持。在早期的 C++ 标准中不允许这样，因为 `shared_ptr` 不可拷贝。`shared_ptr` 中存储引用计数的控制块一直是线程安全的，这保证所指向的对象只被删除一次。然而，`shared_ptr` 中其他任何内容都不是线程安全的。如果在 `shared_ptr` 实例上调用非 `const` 方法(例如 `reset()`)，那么在多个线程中同时使用同一个 `shared_ptr` 实例会导致数据竞争。另一方面，当在多个线程中使用同一个 `atomic<shared_ptr<T>>` 示例时，即使调用非 `const` 的 `shared_ptr` 方法也是线程安全的。请注意，在 `shared_ptr` 所指向的对象上调用非 `const` 方法仍然不是线程安全的，需要手动同步。

C++20

27.3.3 原子引用

C++20 也引入了 `std::atomic_ref`。即使使用相同的接口，它基本上与 `std::atomic` 相同，但它使用的是引用，而 `atomic` 总是拷贝提供给它值。`atomic_ref` 实例本身的生命周期应该比它引用的对象短。`atomic_ref` 是可拷贝的，可以创建任意多个 `atomic_ref` 实例来引用同一个对象。如果 `atomic_ref` 实例引用某个对象，则不允许在没有经过其中一个 `atomic_ref` 实例的情况下接触该对象。`atomic_ref<T>` 类模板可以与任何简单的可复制类型 `T` 一起使用，就像 `std::atomic`。此外，标准库还提供了以下内容：

- 指针类型的偏特化，支持 `fetch_add()` 和 `fetch_sub()`
- 整数类型的全特化，支持 `fetch_add()`、`fetch_sub()`、`fetch_and()`、`fetch_or()` 和 `fetch_xor()`
- 浮点类型的全特化，支持 `fetch_add()` 和 `fetch_sub()`

下面的部分给出了一个如何使用 `atomic_ref` 的示例。

27.3.4 使用原子类型

本节解释为什么应该使用原子类型。假设有下面一个名为 `increment()` 的函数，它在一个循环中递增一个通过引用参数传入的整数值。这段代码使用 `std::this_thread::sleep_for()` 在每个循环中引入一小段延迟。`sleep_for()` 的参数是 `std::chrono::duration`，参见第 22 章“日期和时间工具”。

```
void increment(int& counter)
{
    for (int i { 0 }; i < 100; ++i) {
        ++counter;
        this_thread::sleep_for(1ms);
    }
}
```

现在，想要并行运行多个线程，需要在共享变量 `counter` 上执行这个 `increment()` 函数。如果不使用原子类型或任何线程同步机制，按原始方式实现这个程序，则会引入争用条件。下面的代码在加载了 10 个线程后，调用每个线程的 `join()`，等待所有线程执行完毕。

```
int main()
{
```

```

    int counter { 0 };
    vector<thread> threads;
    for (int i { 0 }; i < 10; ++i) {
        threads.push_back(thread { increment, ref(counter) });
    }

    for (auto& t : threads) {
        t.join();
    }
    cout << "Result = " << counter << endl;
}

```

由于 `increment()` 递增了这个整数 100 次，加载了 10 个线程，并且每个线程都在同一个共享变量 `counter` 上执行 `increment()`，因此期待的结果是 1000。如果执行这个程序几次，可能会得到以下输出，但值不同：

```

Result = 982
Result = 977
Result = 984

```

这段代码清楚地表现了数据争用行为。在这个例子中，可以使用原子类型解决该问题。下面的代码加粗显示了所做的修改：

```

import <atomic>;

void increment(atomic<int>& counter)
{
    for (int i { 0 }; i < 100; ++i) {
        ++counter;
        this_thread::sleep_for(1ms);
    }
}

int main()
{
    atomic<int> counter { 0 };
    vector<thread> threads;
    for (int i { 0 }; i < 10; ++i) {
        threads.push_back(thread { increment, ref(counter) });
    }
    // Remaining code omitted, same as before...
}

```

为这段代码添加了 `<atomic>`，将共享计数器的类型从 `int` 变为 `std::atomic<int>`。运行这个改进后的版本，将永远得到结果 1000：

```

Result = 1000
Result = 1000
Result = 1000

```

不用在代码中显式地添加任何同步机制，就得到了线程安全且没有争用条件的程序，因为对原子类型执行 `++counter` 操作会在原子事务中加载值、递增值并保存值，这个过程不会被打断。

通过 C++20 的 `atomic_ref`，可以像下面这样解决数据竞争问题：

```

void increment(int& counter)
{

```

```
atomic_ref<int> atomicCounter { counter };
for (int i { 0 }; i < 100; ++i) {
    ++atomicCounter;
    this_thread::sleep_for(1ms);
}

int main()
{
    int counter { 0 };
    vector<thread> threads;
    for (int i { 0 }; i < 10; ++i) {
        threads.push_back(thread { increment, ref(counter) });
    }
    // Remaining code omitted, same as before...
}
```

但是，修改后的代码会引发一个新问题：性能问题。应试着最小化同步次数，包括原子操作和显式同步，因为这会降低性能。对于这个简单示例，推荐的最佳解决方案是让 `increment()` 在一个本地变量中计算结果，并且在循环把它添加到 `counter` 引用之后再计算。注意仍需要使用原子类型，因为仍要在多线程中写入 `counter`：

```
void increment(atomic<int>& counter)
{
    int result { 0 };
    for (int i { 0 }; i < 100; ++i) {
        ++result;
        this_thread::sleep_for(1ms);
    }
    counter += result;
}
```

C++20

27.3.5 等待原子变量

C++20 在 `std::atomic` 和 `atomic_ref` 中添加了如表 27-6 所示的方法，用来有效地等待原子变量被修改。

表 27-6 方法

方法	描述
<code>wait(oldValue)</code>	阻塞线程，直到另一个线程调用 <code>notify_one()</code> 或 <code>notify_all()</code> 并且原子变量的值已经改变，即不等于 <code>oldValue</code>
<code>notify_one()</code>	唤醒一个阻塞在 <code>wait()</code> 调用上的线程
<code>notify_all()</code>	唤醒所有阻塞在 <code>wait()</code> 调用上的线程

下面是这些用法的一个示例：

```
atomic<int> value { 0 };

thread job { [&value] {
    cout << "Thread starts waiting." << endl;
    value.wait(0);
    cout << "Thread wakes up, value = " << value << endl;
} };
```

```

this_thread::sleep_for(2s);

cout << "Main thread is going to change value to 1." << endl;
value = 1;
value.notify_all();

job.join();

```

输出如下:

```

Thread starts waiting.
Main thread is going to change value to 1.
Thread wakes up, value = 1

```

27.4 互斥

如果编写的是多线程应用程序,那么必须分外留意操作顺序。如果线程读写共享数据,就可能发生问题。可采用许多方法来避免这个问题,例如绝不在线程之间共享数据。然而,如果不能避免数据共享,那么必须提供同步机制,使一次只有一个线程能更改数据。

布尔值和整数等标量经常使用上述原子操作来实现同步,但当数据更复杂且必须在多个线程中使用这些数据时,就必须提供显式的同步机制。

标准库支持互斥的形式包括互斥体(mutex)类和锁类。这些类都可以用来实现线程之间的同步,接下来讨论这些类。

27.4.1 互斥体类

互斥体(mutex, 代表 mutual exclusion)的基本使用机制如下:

- 希望与其他线程共享内存读写的一个线程试图锁定互斥体对象。如果另一个线程正在持有这个锁,希望获得访问的线程将被阻塞,直到锁被释放,或直到超时。
- 一旦线程获得锁,这个线程就可随意使用共享的内存,因为这要假定希望使用共享数据的所有线程都正确获得了互斥体对象上的锁。
- 线程读写完共享的内存后,线程将锁释放,使其他线程有机会获得访问共享内存的锁。如果两个或多个线程正在等待锁,没有机制能保证哪个线程优先获得锁,并且继续访问数据。

C++标准提供了非定时的互斥体类和定时的互斥体类。有递归和非递归两种风格。在讨论这些选项之前,先看看自旋锁的概念。

1. 自旋锁

自旋锁是互斥锁的一种形式,其中线程使用忙碌循环(自旋)方式来尝试获取锁,执行工作,并释放锁。在旋转时,线程保持活跃,但不做任何有用的工作。即便如此,自旋锁在某些情况下还是很有用,因为它们完全可以在自己的代码中实现,不需要对操作系统进行任何昂贵的调用,也不会造成线程切换的任何开销。如下面的代码片段所示,自旋锁可以使用单个原子类型实现: `atomic_flag`。下面加粗了自旋相关的代码:

```

atomic_flag spinlock = ATOMIC_FLAG_INIT;           // Uniform initialization is not allowed.
static const size_t NumberOfThreads { 50 };
static const size_t LoopsPerThread { 100 };

void dowork(size_t threadNumber, vector<size_t>& data)

```

```

{
    for (size_t i { 0 }; i < LoopsPerThread; ++i) {
        while (spinlock.test_and_set()) { } // Spins until lock is acquired.
        // Save to handle shared data...
        data.push_back(threadNumber);
        spinlock.clear(); // Releases the acquired lock.
    }
}

int main()
{
    vector<size_t> data;
    vector<thread> threads;
    for (size_t i { 0 }; i < NumberOfThreads; ++i) {
        threads.push_back(thread { dowork, i, ref(data) });
    }
    for (auto& t : threads) {
        t.join();
    }
    cout << format("data contains {} elements, expected {}.\\n", data.size(),
        NumberOfThreads * LoopsPerThread);
}

```

在这段代码中，每个线程都试图通过反复调用 `atomic_flag` 上的 `test_and_set()` 来获取一个锁，直到成功。这是忙碌循环。

警告：

由于自旋锁使用忙碌等待循环，因此只有在确定线程只会在短时间内锁定自旋锁时，才应该考虑使用这种方式。

现在看看标准库提供了哪些互斥锁类。

2. 非定时的互斥体类

标准库有 3 个非定时的互斥体类：`std::mutex`、`recursive_mutex` 和 `shared_mutex`。前两个类在 `<mutex>` 中定义，最后一个类在 `<shared_mutex>` 中定义。每个类都支持下列方法。

- `lock()`：调用线程将尝试获取锁，并阻塞直到获得锁。这个方法会无限期阻塞。如果希望设置线程阻塞的最长时间，应该使用定时的互斥体类。
- `try_lock()`：调用线程将尝试获取锁。如果当前锁被其他线程持有，这个调用会立即返回。如果成功获取锁，`try_lock()` 返回 `true`，否则返回 `false`。
- `unlock()`：释放由调用线程持有的锁，使另一个线程能获取这个锁。

`std::mutex` 是一个标准的具有独占所有权语义的互斥体类。只能有一个线程拥有互斥体。如果另一个线程想获得互斥体的所有权，那么这个线程既可通过 `lock()` 阻塞，也可通过 `try_lock()` 尝试失败。已经拥有 `std::mutex` 所有权的线程不能在这个互斥体上再次调用 `lock()` 和 `try_lock()`，否则可能导致死锁！

`std::recursive_mutex` 的行为几乎和 `std::mutex` 一致，区别在于已经获得递归互斥体所有权的线程允许在同一个互斥体上再次调用 `lock()` 和 `try_lock()`。调用线程调用 `unlock()` 方法的次数应该等于获得这个递归互斥体上锁的次数。

`shared_mutex` 支持“共享锁拥有权”的概念，这也称为 `readerswriters` 锁。线程可获取锁的独占所有权或共享所有权。独占所有权也称为写锁，仅当没有其他线程拥有独占或共享所有权时才能获得。

共享所有权也称读锁，如果其他线程都没有独占所有权，则可获得，但允许其他线程获取共享所有权。`shared_mutex` 类支持 `lock()`、`try_lock()` 和 `unlock()`。这些方法获取和释放独占锁。另外，它们具有以下与共享所有权相关的方法：`lock_shared()`、`try_lock_shared()` 和 `unlock_shared()`。这些方法与其他方法集合的工作方式相似，但尝试获取或释放共享所有权。

不允许已经在 `shared_mutex` 上拥有锁的线程在互斥体上获取第二个锁，否则会产生死锁！

在给出如何使用这些互斥锁类的示例之前，需要先讨论几个其他主题。示例会在 27.4.4 节讨论。

警告：

不要在任何互斥体类上手工调用上述锁定和解锁方法。互斥锁是资源，与所有资源一样，它们几乎总是应使用 `RAII`(Resource Acquisition Is Initialization)范型获得，参见第 32 章“结合设计技术和框架”。C++标准定义了一些 `RAII` 锁定类，这些类会在 27.4.2 节讨论。使用它们对避免死锁很重要。锁对象离开作用域时，它们会自动释放互斥体，所以不需要手工调用 `unlock()`。

3. 定时的互斥体类

当对前面讨论过的任何互斥锁类调用 `lock()` 时，该调用会阻塞，直到获得锁为止。另一方面，在那些互斥锁类上调用 `try_lock()` 会尝试获取一个锁，但如果不成功就会立刻返回。还有一些定时互斥类，可以尝试获得锁，但在一段时间后放弃。

标准库提供了 3 个定时的互斥体类：`std::timed_mutex`、`recursive_timed_mutex` 和 `shared_timed_mutex`。前两个类在 `<mutex>` 中定义，最后一个类在 `<shared_mutex>` 中定义。它们都支持 `lock()`、`try_lock()` 和 `unlock()` 方法，`shared_timed_mutex` 还支持 `lock_shared()`、`try_lock_shared()` 和 `unlock_shared()`。所有这些方法的行为与前面描述的类似。此外，它们还支持以下方法。

- `try_lock_for(rel_time)`：调用线程尝试在给定的相对时间内获得这个锁。如果不能获得这个锁，这个调用失败并返回 `false`。如果在超时之前获得了这个锁，这个调用成功并返回 `true`。将超时时间指定为 `std::chrono::duration`，见第 22 章的讨论。
- `try_lock_until(abs_time)`：调用线程将尝试获得这个锁，直到系统时间等于或超过指定的绝对时间。如果能在超时之前获得这个锁，调用返回 `true`。如果系统时间超过给定的绝对时间，将不再尝试获得锁，并返回 `false`。将绝对时间指定为 `std::chrono::time_point`，见第 22 章的讨论。

`shared_timed_mutex` 还支持 `try_lock_shared_for()` 和 `try_lock_shared_until()`。

已经拥有 `timed_mutex` 或 `shared_timed_mutex` 所有权的线程不允许再次获得这个互斥体上的锁，否则可能导致死锁！

`recursive_timed_mutex` 的行为和 `recursive_mutex` 类似，允许一个线程多次获取锁。

27.4.2 锁

锁类是 `RAII` 类，可用于更方便地正确获得和释放互斥体上的锁；锁类的析构函数会自动释放所关联的互斥体。C++标准定义了 4 种类型的锁：`std::lock_guard`、`unique_lock`、`shared_lock` 和 `scoped_lock`。

1. `lock_guard`

`lock_guard` 在 `<mutex>` 中定义，有两个构造函数。

- `explicit lock_guard(mutex_type& m)`;

接收一个互斥体引用的构造函数。这个构造函数尝试获得互斥体上的锁，并阻塞直到获得锁。第 8 章“精通类和对象”讨论了构造函数的 `explicit` 关键字。

- `lock_guard(mutex_type& m, adopt_lock_t);`

接收一个互斥体引用和一个 `std::adopt_lock_t` 实例的构造函数。C++提供了一个预定义的 `adopt_lock_t` 实例，名为 `std::adopt_lock`。该锁假定调用线程已经获得引用的互斥体上的锁，管理该锁，在销毁锁时自动释放互斥体。

2. `unique_lock`

`std::unique_lock` 定义在 `<mutex>` 中，是一类更复杂的锁，允许将获得锁的时间延迟到计算需要时，远在声明之后。使用 `owns_lock()` 方法可以确定是否获得了锁。`unique_lock` 也有 `bool` 转换运算符，可用于检查是否获得了锁。使用这个转换运算符的例子在本章后面给出。`unique_lock` 有如下几个构造函数。

- `explicit unique_lock(mutex_type& m);`

接收一个互斥体引用的构造函数。这个构造函数尝试获得互斥体上的锁，并且阻塞直到获得锁。

- `unique_lock(mutex_type& m, defer_lock_t) noexcept;`

接收一个互斥体引用和一个 `std::defer_lock_t` 实例的构造函数。C++提供了一个预定义的 `defer_lock_t` 实例，名为 `std::defer_lock`。`unique_lock` 存储互斥体的引用，但不立即尝试获得锁，锁可以稍后获得。

- `unique_lock(mutex_type& m, try_to_lock_t);`

接收一个互斥体引用和一个 `std::try_to_lock_t` 实例的构造函数。C++提供了一个预定义的 `try_to_lock_t` 实例，名为 `std::try_to_lock`。这个锁尝试获得引用的互斥体上的锁，但即便未能获得也不阻塞；此时，会在稍后获取锁。

- `unique_lock(mutex_type& m, adopt_lock_t);`

接收一个互斥体引用和一个 `std::adopt_lock_t` 实例(如 `std::adopt_lock`)的构造函数。这个锁假定调用线程已经获得引用的互斥体上的锁。锁管理互斥体，并在销毁锁时自动释放互斥体。

- `unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);`

接收一个互斥体引用和一个绝对时间的构造函数。这个构造函数试图获取一个锁，直到系统时间超过给定的绝对时间。第 22 章讨论了 `chrono` 库。

- `unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);`

接收一个互斥体引用和一个相对时间的构造函数。这个构造函数试图获得一个互斥体上的锁，直到到达给定的相对超时时间。

`unique_lock` 类也有以下方法：`lock()`、`try_lock()`、`try_lock_for()`、`try_lock_until()`和`unlock()`。这些方法的行为和前面介绍的定时的互斥体类中的方法一致。

3. `shared_lock`

`shared_lock` 类在 `<shared_mutex>` 中定义，它的构造函数和方法与 `unique_lock` 相同。区别是，`shared_lock` 类在底层的共享互斥体上调用与共享拥有权相关的方法。因此，`shared_lock` 的方法称为 `lock()`、`try_lock()`等，但在底层的共享互斥体上，它们称为 `lock_shared()`、`try_lock_shared()`等。因此，`shared_lock` 与 `unique_lock` 有相同的接口，可用作 `unique_lock` 的替代品，但获得的是共享锁，而不是独占锁。

4. 一次性获得多个锁

C++有两个泛型锁函数，可用于同时获得多个互斥体对象上的锁，而不会出现死锁。这两个泛型锁函数都在 `std` 名称空间中定义，都是可变参数模板函数。第 26 章讨论了可变参数模板函数。

第一个函数 `lock()` 不按指定顺序锁定所有给定的互斥体对象，没有出现死锁的风险。如果其中一个互斥锁调用抛出异常，则在已经获得的所有锁上调用 `unlock()`。原型如下：

```
template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);
```

`try_lock()` 函数具有类似的原型，但它通过顺序调用每个给定互斥体对象的 `try_lock()`，试图获得所有互斥体对象上的锁。如果所有 `try_lock()` 调用都成功，那么这个函数返回 `-1`。如果任何 `try_lock()` 调用失败，那么对所有已经获得的锁调用 `unlock()`，返回值是在其上调用 `try_lock()` 失败的互斥体的参数位置索引(从 0 开始计算)。

下例演示如何使用泛型函数 `lock()`。`process()` 函数首先创建两个锁，每个互斥体一个锁，然后将一个 `std::defer_lock_t` 实例作为第二个参数传入，告诉 `unique_lock` 不要在构造期间获得锁。然后调用 `std::lock()` 以获得这两个锁，而不会出现死锁：

```
mutex mut1;
mutex mut2;

void process()
{
    unique_lock lock1 { mut1, defer_lock };
    unique_lock lock2 { mut2, defer_lock };
    lock(lock1, lock2);
    // Locks acquired.
} // Locks automatically released.
```

5. `scoped_lock`

`std::scoped_lock` 在 `<mutex>` 中定义，与 `lock_guard` 类似，只是接收数量可变的互斥体。这样，就可极方便地获取多个锁。例如，可以使用 `scoped_lock`，编写刚才包含 `process()` 函数的那个示例，如下所示：

```
mutex mut1;
mutex mut2;

void process()
{
    scoped_lock locks { mut1, mut2 };
    // Locks acquired
} // Locks automatically released
```

注意：

`scoped_lock` 不仅简化了获取多个锁的过程，因为不需要担心需要以正确的顺序获取它们，而且它的性能也比手动获取锁要好。

27.4.3 `std::call_once`

结合使用 `std::call_once()` 和 `std::once_flag` 可确保某个函数或方法正好只调用一次，不论有多少个线程试图调用 `call_once()` (在同一 `once_flag` 上) 都同样如此。只有一个 `call_once()` 调用能真正调用给定的函数或方法。如果给定的函数不抛出任何异常，则这个调用称为有效的 `call_once()` 调用。如果给定的函数抛出异常，异常将传回调用者，选择另一个调用者来执行此函数。某个特定的 `once_flag` 实例的有效调用在对同一个 `once_flag` 实例的其他所有 `call_once()` 调用之前完成。在同一个 `once_flag` 实例

上调用 `call_once()` 的其他线程都会阻塞，直到有效调用结束。图 27-4 通过 3 个线程演示了这一点。线程 1 执行有效的 `call_once()` 调用，线程 2 阻塞，直到这个有效调用完成，线程 3 不会阻塞，因为线程 1 的有效调用已经完成了。

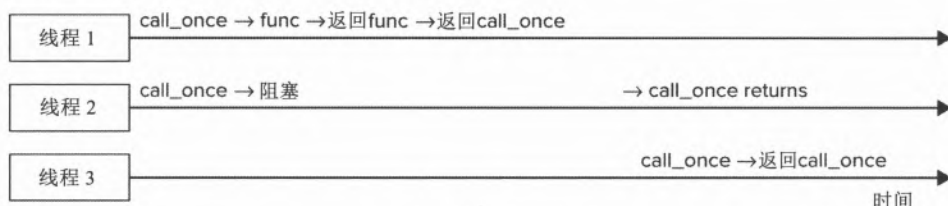


图 27-4 通过 3 个线程演示

下例演示了 `call_once()` 的使用。这个例子运行使用某个共享资源的 `processingFunction()`，启动了 3 个线程。这些线程应调用 `initializeSharedResources()` 一次。为此，每个线程用全局的 `once_flag` 调用 `call_once()`，结果是只有一个线程执行 `initializeSharedResources()`，且只执行一次。在调用 `call_once()` 的过程中，其他线程被阻塞，直到 `initializeSharedResources()` 返回：

```

once_flag g_onceFlag;

void initializeSharedResources()
{
    // ... Initialize shared resources to be used by multiple threads.
    cout << "Shared resources initialized." << endl;
}

void processingFunction()
{
    // Make sure the shared resources are initialized.
    call_once(g_onceFlag, initializeSharedResources);

    // ... Do some work, including using the shared resources
    cout << "Processing" << endl;
}

int main()
{
    // Launch 3 threads.
    vector<thread> threads { 3 };
    for (auto& t : threads) {
        t = thread { processingFunction };
    }
    // Join on all threads
    for (auto& t : threads) {
        t.join();
    }
}
  
```

这段代码的输出如下所示：

```

Shared resources initialized.
Processing
Processing
Processing
  
```

当然,在这个例子中,也可在启动线程之前,在 `main()` 函数的开头调用 `initializeSharedResources()`,但那样就无法演示 `call_once()` 的用法了。

27.4.4 互斥体对象的用法示例

下面列举几个例子,演示如何使用互斥体对象同步多个线程。

1. 以线程安全方式写入流

在本章前面有关线程的内容中,有一个例子使用了名为 `Counter` 的类。这个例子提到, C++ 中的流是不会出现争用条件的,但来自不同线程的输出仍会交错。这里有两个解决这个问题的方案:

- 使用 C++20 的同步流。
- 使用互斥对象以确保每次只有一个线程对流对象进行读写。

同步流

C++20 引入了 `std::basic_ostream`, 并分别为 `char` 流和 `wchar_t` 流预定义了类型别名 `ostream` 和 `wostream`, 它们都在 `<ostream>` 中定义。这些类名中的 `O` 表示输出。这些类保证所有通过它们完成的输出都将在同步流被销毁的那一刻出现在最终的输出流中。保证输出不会和其他线程的其他输出交错。

前面 `Counter` 类的函数调用运算符可以使用 `ostream` 来实现, 以防止交错输出:

```
class Counter
{
public:
    Counter(int id, int numIterations)
        : m_id { id }, m_numIterations { numIterations } {}

    void operator() () const
    {
        for (int i { 0 }; i < m_numIterations; ++i) {
            ostream { cout } << "Counter "
                               << m_id << " has value " << i << endl;
        }
    }

private:
    int m_id;
    int m_numIterations;
};

或:

void operator() () const
{
    for (int i { 0 }; i < m_numIterations; ++i) {
        ostream syncedCout { cout };
        syncedCout << "Counter " << m_id << " has value " << i << endl;
    }
}
```

使用锁

如果不能使用同步流, 可以使用以下代码段所示的互斥锁来同步 `Counter` 类中对 `cout` 的所有访问, 为此添加了一个静态互斥对象。它应该是静态的, 因为类的所有实例都应该使用同一个互斥对象实例。

`lock_guard` 用于在写入 `cout` 之前获取互斥锁。

```
class Counter
{
public:
    Counter(int id, int numIterations)
        : m_id { id }, m_numIterations { numIterations } { }

    void operator()() const
    {
        for (int i { 0 }; i < m_numIterations; ++i) {
            lock_guard lock { ms_mutex };
            cout << "Counter " << m_id << " has value " << i << endl;
        }
    }
private:
    int m_id;
    int m_numIterations;
    inline static mutex ms_mutex;
};
```

这段代码在 `for` 循环的每次迭代中创建了一个 `lock_guard` 实例。建议尽可能限制拥有锁的时间，否则阻塞其他线程的时间就会过长。例如，如果 `lock_guard` 实例在 `for` 循环之前创建一次，就基本上丢失了这段代码中的所有多线程特性，因为一个线程在其 `for` 循环的整个执行期间都拥有锁，所有其他线程都等待这个锁被释放。

2. 使用定时锁

下面的示例演示如何使用定时的互斥体。这与此前是同一个 `Counter` 类，但这一次结合 `unique_lock` 使用了 `timed_mutex`。将 200 毫秒的相对时间传给 `unique_lock` 构造函数，试图在 200 毫秒内获得一个锁。如果不能在这个时间间隔内获得这个锁，构造函数返回。之后，可检查这个锁是否已经获得，对这个 `lock` 变量应用 `if` 语句就可执行这种检查，因为 `unique_lock` 类定义了 `bool` 转换运算符。使用 `chrono` 库指定超时时间，第 22 章讨论了这个库。

```
class Counter
{
public:
    Counter(int id, int numIterations)
        : m_id { id }, m_numIterations { numIterations } { }

    void operator()() const
    {
        for (int i { 0 }; i < m_numIterations; ++i) {
            unique_lock lock { ms_timedMutex, 200ms };
            if (lock) {
                cout << "Counter " << m_id << " has value " << i << endl;
            } else {
                // Lock not acquired in 200ms, skip output.
            }
        }
    }
private:
    int m_id;
    int m_numIterations;
```

```

        inline static timed_mutex ms_timedMutex;
    };

```

3. 双重检查锁定

双重检查锁定(double-checked locking)实际上是一种反模式，应避免使用！这里之所以介绍它，是因为你可能在现有代码库中遇到它。双重检查锁定模式旨在尝试避免使用互斥体对象。这是编写比使用互斥体对象更有效代码的一种半途而废的尝试。如果在后续示例中想要提高速度，真的可能出错，例如使用 `relaxed atomic`(本章不讨论)，用普通的 `Boolean` 替代 `atomic<bool>` 等。该模式容易出现争用条件，很难更正。具有讽刺意味的是，使用 `call_once()` 实际上更快，使用 `magic static`(如果可用)速度更快。将函数的本地静态实例称为 `magic static`。

警告：

在新的代码中避免使用双重检查锁定模式，而使用其他机制，例如简单锁、原子变量、`call_once()` 和 `magic static` 等。

注意：

函数局部静态实例称为魔术静态。C++ 保证这样的局部静态实例以线程安全的方式初始化，因此不需要进行任何手动的线程同步。第 33 章“应用设计模式”给出了一个使用 `magic static` 的示例，其中讨论了单例模式。

例如，双重检查锁定可用于确保资源正好初始化一次。下例演示了如何实现这个功能。之所以称为双重检查锁定算法，是因为它检查 `g_initialized` 变量的值两次，一次在获得锁之前，另一次在获得锁之后。第一次检查 `g_initialized` 变量是为了防止获得不需要的锁。第二次检查用于确保没有其他线程在第一次 `g_initialized` 检查和获得锁之间执行初始化。

```

void initializeSharedResources()
{
    // ... Initialize shared resources to be used by multiple threads.
    cout << "Shared resources initialized." << endl;
}

atomic<bool> g_initialized { false };
mutex g_mutex;

void processingFunction()
{
    if (!g_initialized) {
        unique_lock lock { g_mutex };
        if (!g_initialized) {
            initializeSharedResources();
            g_initialized = true;
        }
    }
    cout << "OK" << endl;
}

int main()
{
    vector<thread> threads;

```

```

    for (int i { 0 }; i < 5; ++i) {
        threads.push_back(thread { processingFunction });
    }
    for (auto& t : threads) {
        t.join();
    }
}

```

输出清楚地表明，只有一个线程初始化了共享资源：

```

Shared resources initialized.
OK
OK
OK
OK
OK

```

注意：

对于这个例子，建议使用 `call_once()` 而不是双重检查锁定。

27.5 条件变量

条件变量允许一个线程阻塞，直到另一个线程设置某个条件或系统时间到达某个指定的时间。条件变量允许显式的线程间通信。如果熟悉 Win32 API 的多线程编程，就可将条件变量和 Windows 中的事件对象进行比较。

有两类条件变量，它们都定义在 `<condition_variable>` 中。

- **std::condition_variable:** 只能等待 `unique_lock<mutex>` 上的条件变量；根据 C++ 标准的描述，这个条件变量可在特定平台上达到最高效率。
- **std::condition_variable_any:** 可等待任何对象的条件变量，包括自定义的锁类型。

`condition_variable` 类支持以下方法。

- `notify_one();`

唤醒等待这个条件变量的线程之一。这类似于 Windows 上的 auto-reset 事件。

- `notify_all();`

唤醒等待这个条件变量的所有线程。

- `wait(unique_lock<mutex>& lk);`

调用 `wait()` 的线程应该已经获得 `lk` 上的锁。调用 `wait()` 的效果是以原子方式调用 `lk.unlock()` 并阻塞线程，等待通知。当线程被另一个线程中的 `notify_one()` 或 `notify_all()` 调用解除阻塞时，这个函数会再次调用 `lk.lock()`，可能会被这个锁阻塞，然后返回。

- `wait_for(unique_lock<mutex>& lk, const chrono::duration<Rep, Period>& rel_time);`

类似于此前的 `wait()` 方法，区别在于这个线程会被 `notify_one()` 或 `notify_all()` 调用解除阻塞，也可能在给定超时时间到达后解除阻塞。

- `wait_until(unique_lock<mutex>& lk, const chrono::time_point<Clock, Duration>& abs_time);`

类似于此前的 `wait()` 方法，区别在于这个线程会被 `notify_one()` 或 `notify_all()` 调用解除阻塞，也可能在系统时间超过给定的绝对时间时解除阻塞。

也有一些其他版本的 `wait()`、`wait_for()` 和 `wait_until()` 接收一个额外的谓词参数。例如，接收一个额外谓词的 `wait()` 等同于：

```
while (!predicate())
    wait(lk);
```

`condition_variable_any` 类支持的方法和 `condition_variable` 类相同, 区别在于 `condition_variable_any` 可接收任何类型的锁类, 而不只是 `unique_lock<mutex>`。锁类应提供 `lock()` 和 `unlock()` 方法。

27.5.1 虚假唤醒

等待条件变量的线程可在另一个线程调用 `notify_one()` 或 `notify_all()` 时醒过来, 或在系统时间超过给定时间时醒过来, 也可能不合时宜地醒过来。这意味着, 即使没有其他线程调用任何通知方法, 线程也会醒过来。因此, 当线程等待一个条件变量并醒过来时, 就需要检查它是否因为获得通知而醒过来。一种检查方法是使用接收谓词参数的 `wait()` 版本。

27.5.2 使用条件变量

条件变量可用于处理队列项的后台线程。可定义队列, 在队列中插入要处理的项。后台线程等待队列中出现项。把一项插入队列中时, 线程就醒过来, 处理项, 然后继续休眠, 等待下一项。假设有以下队列:

```
queue<string> m_queue;
```

需要确保在任何时候只有一个线程修改这个队列。可通过互斥体实现这一点:

```
mutex m_mutex;
```

为了能在添加一项时通知后台线程, 需要一个条件变量:

```
condition_variable m_condVar;
```

需要向队列中添加项的线程首先要获得这个互斥体上的锁, 然后向队列中添加项, 最后通知后台线程。无论当前是否拥有锁, 都可以调用 `notify_one()` 或 `notify_all()`, 它们都会正常工作:

```
// Lock mutex and add entry to the queue.
unique_lock lock { m_mutex };
m_queue.push(entry);
// Notify condition variable to wake up thread.
m_condVar.notify_all();
```

后台线程在一个无限循环中等待通知。注意这里使用接收谓词参数的 `wait()` 方法正确处理线程不合时宜地醒过来的情形。谓词检查队列中是否有队列项。对 `wait()` 的调用返回时, 就可以肯定队列中有队列项了。

```
unique_lock lock { m_mutex };
while (true) {
    // Wait for a notification.
    m_condVar.wait(lock, [this]{ return !m_queue.empty(); });
    // Condition variable is notified, so something is in the queue.
    // Process queue item...
}
```

第 27.7 节给出了一个完整示例, 讲解了如何通过条件变量向其他线程发送通知。

C++ 标准还定义了辅助函数 `std::notify_all_at_thread_exit(cond, lk)`, 其中 `cond` 是一个条件变量, `lk`

是一个 `unique_lock<mutex>` 实例。调用这个函数的线程应该已经获得了锁 `lk`。当线程退出时，会自动执行以下代码：

```
lk.unlock();
cond.notify_all();
```

注意：

将锁 `lk` 保持锁定，直到该线程退出为止。因此，一定要确保这不会在代码中造成任何死锁，例如由于错误的锁顺序而产生的死锁。本章前面已经讨论了死锁。

27.6 latch

`latch` 是一次性使用的线程协调点。一旦给定数量的线程达到 `latch` 点时，所有线程都会解除阻塞并继续执行。基本上它是个计数器，在每个线程到达 `latch` 点时倒数。一旦计数器达到零，`latch` 将无限期保持在一个有信号的状态，所有阻塞线程都将解除阻塞，随后到达 `latch` 点的任何线程会立刻被允许继续执行。

`latch` 由 `std::latch` 实现，在 `<latch>` 中定义。构造函数接收需要到达 `latch` 点的所需线程数。到达 `latch` 点的线程可以调用 `arrive_and_wait()`，它递减 `latch` 计数器并阻塞，直到 `latch` 有信号为止。线程也可以通过调用 `wait()` 在不减少计数器的情况下阻塞在 `latch` 点上。`try_wait()` 方法可用于检查计数器是否达到零。最后，如果需要，还可以通过调用 `count_down()` 来减少计数器，而不会阻塞。

下面演示了一个 `latch` 点的用例，其中一些数据需要加载到内存(I/O bound)中，然后在多个线程中并行处理这些数据。进一步假设线程在启动时和开始处理数据之前需要执行一些 CPU 绑定的初始化。通过先启动线程并让它们进行 CPU 绑定的初始化，并且并行加载数据(I/O bound)，性能得到了提高。代码用计数器 1 初始化一个 `latch` 对象，并启动 10 个线程，这些线程都进行一些初始化，然后阻塞 `latch`，直到 `latch` 计数器达到零。在启动 10 个线程之后，代码加载一些数据，例如从磁盘中。一旦加载了所有数据，`latch` 计数器将减为 0，这 10 个等待线程都将解除阻塞。

```
latch startLatch { 1 };
vector<jthread> threads;

for (int i { 0 }; i < 10; ++i) {
    threads.push_back(jthread { [&startLatch] {
        // Do some initialization... (CPU bound)

        // Wait until the latch counter reaches zero.
        startLatch.wait();

        // Process data...
    } });
}

// Load data... (I/O bound)

// Once all data has been loaded, decrement the latch counter
// which then reaches zero and unblocks all waiting threads.
startLatch.count_down();
```


C++20

27.7 barrier

barrier 是由一系列阶段组成的可重用线程协调机制。许多线程在 **barrier** 点阻塞。当给定数量的线程到达 **barrier** 时，将执行完成阶段的回调，解除所有阻塞线程的阻塞，重置线程计数器，并开始下一个阶段。在每个阶段中，可以调整下一阶段的预期线程数。**barrier** 对于在循环之间执行同步非常有用。例如，假设有很多线程并发执行，并在一个循环中执行一些计算。进一步假设一旦这些计算完成，需要在线程开始其循环的新迭代之前对结果进行一些处理。对于这种情况，设置 **barrier** 是完美的，所有的线程都会阻塞在 **barrier** 处。当它们全部到达时，完成阶段的回调将处理线程的结果，然后解除所有线程的阻塞，以开始它们的下一次迭代。

barrier 由 `std::barrier` 实现，在 `<barrier>` 中定义。**barrier** 最重要的方式是 `arrive_and_wait()`，它减少计数器，然后阻塞线程，直到当前阶段完成。有关其他可用方法的完整描述，请参阅标准库指南。

下面的代码片段演示了 **barrier** 的使用。它启动 4 个线程，在循环中连续执行某些操作。在每次迭代中，所有线程都是用 **barrier** 进行同步。

```
void completionFunction() noexcept { /* ... */ }

int main()
{
    const size_t numberOfThreads { 4 };
    barrier barrierPoint { numberOfThreads, completionFunction };
    vector<jthread> threads;

    for (int i { 0 }; i < numberOfThreads; ++i) {
        threads.push_back(jthread { [&barrierPoint] (stop_token token) {
            while (!token.stop_requested()) {
                // ... Do some calculations ...

                // Synchronize with other threads.
                barrierPoint.arrive_and_wait();
            }
        } });
    }
}
```

C++20

27.8 semaphore

semaphore(信号量)是轻量级同步原语，可用作其他同步机制(如 **mutex**、**latch**、**barrier**)的构建块。基本上一个 **semaphore** 由一个表示很多插槽的计数器组成。计数器在构造函数中初始化。如果获得了一个插槽，计数器将减少，而释放插槽将增加计数器。在 `<semaphore>` 中定义了两个 **semaphore** 类：`std::counting_semaphore` 和 `binary_semaphore`。前一种模型是非负资源计数。后者只有一个插槽，该槽要么是空的，要么不是空的，完全适合作为互斥的构建块。两者都提供了如表 27-7 所示的方法。

表 27-7 方法

方法	描述
<code>acquire()</code>	递减计数器。当计数器为零时阻塞，直到计数器再次递增
<code>try_acquire()</code>	尝试递减计数器，但如果计数器已经为零不会阻塞。如果计数器可以递减，则返回 <code>true</code> ，否则返回 <code>false</code>

(续表)

方法	描述
try_acquire_for()	与 try_acquire 相同，但会在给定的时间段内尝试
try_acquire_until()	与 try_acquire 相同，但是会一直尝试直到系统到达给定时间
release()	计数器增加一个给定的数，并解除在 acquire 调用中线程的阻塞

计数 semaphore 允许精确地控制希望允许并发运行的线程数量。例如，下面的代码片段允许最多 4 个线程并行运行：

```
counting_semaphore semaphore { 4 };
vector<jthread> threads;
for (int i { 0 }; i < 10; ++i) {
    threads.push_back(jthread { [&semaphore] {
        semaphore.acquire();
        // ... Slot acquired ... (at most 4 threads concurrently)
        semaphore.release();
    } });
}
```

semaphore 的另一个用例是为线程而不是为条件变量实现通知机制。例如，可以在其构造函数中将 semaphore 的计数器初始化为 0，任何调用 acquire() 的线程都会阻塞，直到其他线程对 semaphore 调用 release()。

27.9 future

根据本章前面的讨论，可通过 std::thread 启动一个线程，计算并得到一个结果，当线程结束执行时不容易取回计算的结果。与 std::thread 相关的另一个问题是处理像异常这样的错误。如果一个线程抛出一个异常，而这个异常没有被线程本身处理，C++运行时将调用 std::terminate()，这通常会终止整个应用程序。

可使用 future 更方便地获得线程的结果，并将异常转移到另一个线程中，然后另一个线程可以任意处置这个异常。当然，应该总是尝试在线程本身处理异常，不要让异常离开线程。

future 在 promise 中存储结果。可通过 future 获取 promise 中存储的结果。也就是说，promise 是结果的输入端；future 是输出端。一旦在同一线程或另一线程中运行的函数计算出希望返回的值，就把这个值放在 promise 中。然后可以通过 future 获取这个值。可将 future/promise 对想象为线程间传递结果的通信信道。

C++提供标准的 future 名为 std::future。可从 std::future 检索结果。T 是计算结果的类型。

```
future<T> myFuture { ... }; // Is discussed later.
T result { myFuture.get() };
```

调用 get() 以取出结果，并保存在变量 result 中。如果另一个线程尚未计算完结果，对 get() 的调用将阻塞，直到该结果值可用。只能在 future 上调用一次 get()。按照标准，第二次调用的行为是不确定的。

可首先通过向 future 询问结果是否可用的方式来避免阻塞：

```
if (myFuture.wait_for(0)) { // Value is available.
    T result { myFuture.get() };
} else { // Value is not yet available.
    ...
}
```

27.9.1 std::promise 和 std::future

C++提供了 `std::promise` 类，作为实现 `promise` 概念的一种方式。可在 `promise` 上调用 `set_value()` 来存储结果，也可调用 `set_exception()`，在 `promise` 中存储异常。注意，只能在特定的 `promise` 上调用 `set_value()` 或 `set_exception()` 一次。如果多次调用它，将抛出 `std::future_error` 异常。

如果线程 A 启动另一个线程 B 以执行计算，则线程 A 可创建一个 `std::promise`，将其传给已启动的线程。注意，无法复制 `promise`，但可将其移到线程中！线程 B 使用 `promise` 存储结果。将 `promise` 移入线程 B 之前，线程 A 在创建的 `promise` 上调用 `get_future()`，这样，线程 B 完成后就能访问结果。下面是一个简单示例：

```
void doWork(promise<int> thePromise)
{
    // ... Do some work ...
    // And ultimately store the result in the promise.
    thePromise.set_value(42);
}

int main()
{
    // Create a promise to pass to the thread.
    promise<int> myPromise;

    // Get the future of the promise.
    auto theFuture { myPromise.get_future() };
    // Create a thread and move the promise into it.
    thread theThread { doWork, move(myPromise) };

    // Do some more work...

    // Get the result.
    int result { theFuture.get() };
    cout << "Result: " << result << endl;

    // Make sure to join the thread.
    theThread.join();
}
```

注意：

这段代码只用于演示。这段代码在一个新的线程中启动计算，然后在 `future` 上调用 `get()`。这个线程会阻塞，直到结果计算完为止。这听起来像代价很高的函数调用。在实际应用程序中使用 `future` 模型时，可定期检查 `future` 中是否有可用的结果(通过此前描述的 `wait_for()`)，或者使用条件变量等同步机制。当结果还不可用时，可做其他事情，而不是阻塞。

27.9.2 std::packaged_task

有了 `std::packaged_task`，将可以更方便地使用 `promise`，而不是像 27.6.1 节那样显式地使用 `std::promise`。下面的代码演示了这一点。它创建一个 `packaged_task` 来执行 `calculateSum()`。通过调用 `get_future()`，从 `packaged_task` 检索 `future`。启动一个线程，并将 `packaged_task` 移入其中。无法复制

`packaged_task`! 启动线程后, 在检索到的 `future` 上调用 `get()` 来获得结果。在结果可用前, 将一直阻塞。

`calculateSum()` 不需要在任何类型的 `promise` 中显式存储任何数据。`packaged_task` 自动创建 `promise`, 自动在 `promise` 中存储被调用函数(这里是 `calculateSum()`)的结果, 并自动在 `promise` 中存储函数抛出的任何异常。

```
int calculateSum(int a, int b) { return a + b; }

int main()
{
    // Create a packaged task to run calculateSum.
    packaged_task<int(int, int)> task { calculateSum };
    // Get the future for the result of the packaged task.
    auto theFuture { task.get_future() };
    // Create a thread, move the packaged task into it, and
    // execute the packaged task with the given arguments.
    thread theThread { move(task), 39, 3 };
    // Do some more work...

    // Get the result.
    int result { theFuture.get() };
    cout << result << endl;

    // Make sure to join the thread.
    theThread.join();
}
```

27.9.3 std::async

如果想让 C++ 运行时更多地控制是否创建一个线程以进行某种计算, 可使用 `std::async()`。它接收一个将要执行的函数, 并返回可用于检索结果的 `future`。`async()` 可通过两种方法运行函数:

- 创建一个新的线程, 异步运行提供的函数。
- 在返回的 `future` 上调用 `get()` 方法时, 在主调线程上同步地运行函数。

如果没有通过额外参数来调用 `async()`, C++ 运行时会根据一些因素(例如系统中处理器的数目)从两种方法中自动选择一种方法。也可指定策略参数, 从而调整 C++ 运行时的行为。

- `launch::async`: 强制 C++ 运行时在一个不同的线程上异步地执行函数。
- `launch::deferred`: 强制 C++ 运行时在调用 `get()` 时, 在主调线程上同步地执行函数。
- `launch::async | launch::deferred`: 允许 C++ 运行时进行选择(=默认行为)。

下例演示了 `async()` 的用法:

```
int calculate() { return 123; }
int main()
{
    auto myFuture { async(calculate) };
    //auto myFuture { async(launch::async, calculate) };
    //auto myFuture { async(launch::deferred, calculate) };

    // Do some more work...

    // Get the result.
    int result { myFuture.get() };
    cout << result << endl;
}
```

从这个例子可看出, `std::async()`是以异步方式(在不同线程中)或同步方式(在同一线程中)执行一些计算并在随后获取结果的最简单方法之一。

警告:

调用 `async()` 锁返回的 `future` 会在其析构函数中阻塞,直到结果可用为止。这意味着如果调用 `async()` 时未捕获返回的 `future`, `async()` 调用将真正成为阻塞调用! 例如, 以下代码行同步调用 `calculate()`:

```
async(calculate);
```

在这条语句中, `async()` 创建和返回 `future`。未捕获这个 `future`, 因此是临时 `future`。由于是临时的, 因此将在该语句完成前调用其析构函数, 在结果可用前, 该析构函数将一直阻塞。

27.9.4 异常处理

使用 `future` 的一大优点是它们会自动在线程之间传递异常。在 `future` 上调用 `get()` 时, 要么返回计算结果, 要么重新抛出与 `future` 关联的 `promise` 中存储的任何异常。使用 `packaged_task` 或 `async()` 时, 从已启动的函数抛出的任何异常将自动存储在 `promise` 中。如果将 `std::promise` 用作 `promise`, 可调用 `set_exception()` 以在其中存储异常。下面是一个使用 `async()` 的示例:

```
int calculate()
{
    throw runtime_error { "Exception thrown from calculate()." };
}

int main()
{
    // Use the launch::async policy to force asynchronous execution.
    auto myFuture { async(launch::async, calculate) };

    // Do some more work...

    // Get the result.
    try {
        int result { myFuture.get() };
        cout << result << endl;
    } catch (const exception& ex) {
        cout << "Caught exception: " << ex.what() << endl;
    }
}
```

27.9.5 std::shared_future

`std::future<T>` 只要求 `T` 可移动构建。在 `future<T>` 上调用 `get()` 时, 结果将移出 `future`, 并返回给你。这意味着只能在 `future<T>` 上调用 `get()` 一次。

如果要多次调用 `get()`, 甚至从多个线程多次调用, 则需要使用 `std::shared_future<T>`, 此时, `T` 需要可复制构建。可使用 `std::future::share()`, 或给 `shared_future` 构造函数传递 `future`, 以创建 `shared_future`。注意, `future` 不可复制, 因此需要将其移入 `shared_future` 构造函数。

`shared_future` 可用于同时唤醒多个线程。例如, 下面的代码片段定义了两个 `lambda` 表达式, 它们在不同的线程上异步地执行。每个 `lambda` 表达式首先将值设置为各自的 `promise`, 以指示已经启动。接着在 `signalFuture` 调用 `get()`, 这一直阻塞, 直到可通过 `future` 获得参数为止; 此后将继续执行。每个 `lambda` 表达式按引用捕获各自的 `promise`, 按值捕获 `signalFuture`, 因此这两个 `lambda` 表达式都有

`signalFuture` 的副本。主线程使用 `async()`，在不同线程上执行这两个 `lambda` 表达式，一直等到线程启动，然后设置 `signalPromise` 中的参数以唤醒这两个线程。

```
promise<void> thread1Started, thread2Started;

promise<int> signalPromise;
auto signalFuture { signalPromise.get_future().share() };
//shared_future<int> signalFuture { signalPromise.get_future() };

auto function1 { [&thread1Started, signalFuture] {
    thread1Started.set_value();
    // Wait until parameter is set.
    int parameter { signalFuture.get() };
    // ...
} };

auto function2 { [&thread2Started, signalFuture] {
    thread2Started.set_value();
    // Wait until parameter is set.
    int parameter { signalFuture.get() };
    // ...
} };

// Run both lambda expressions asynchronously.
// Remember to capture the future returned by async()!
auto result1 { async(launch::async, function1) };
auto result2 { async(launch::async, function2) };

// Wait until both threads have started.
thread1Started.get_future().wait();
thread2Started.get_future().wait();

// Both threads are now waiting for the parameter.
// Set the parameter to wake up both of them.
signalPromise.set_value(42);
```

27.10 示例：多线程的 `Logger` 类

本节演示如何使用线程、互斥体对象、锁和条件变量编写一个多线程的 `Logger` 类。这个类允许不同的线程向一个队列中添加日志消息。`Logger` 类本身会在另一个后台线程中处理这个队列，将日志信息串行地写入一个文件。这个类的设计经历了两次迭代，以说明编写多线程代码时可能遇到的问题。

C++标准没有线程安全的队列。很明显，必须通过一些同步机制保护对队列的访问，避免多个线程同时读写队列。这个示例使用互斥体对象和条件变量来提供同步。在此基础上，可以这样定义 `Logger` 类：

```
export class Logger
{
public:
    // Starts a background thread writing log entries to a file.
    Logger();
    // Prevent copy construction and assignment.
    Logger(const Logger& src) = delete;
    Logger& operator=(const Logger& rhs) = delete;
```

```

        // Add log entry to the queue.
        void log(std::string entry);
private:
    // The function running in the background thread.
    void processEntries();
    // Helper method to process a queue of entries.
    void processEntriesHelper(std::queue<std::string>& queue,
                             std::ofstream& ofs) const;
    // Mutex and condition variable to protect access to the queue.
    std::mutex m_mutex;
    std::condition_variable m_condVar;
    std::queue<std::string> m_queue;
    // The background thread.
    std::thread m_thread;
};

```

实现如下。注意这个最初的设计存在几个问题，尝试运行这个程序时，它可能会行为异常甚至崩溃，在 `Logger` 类的下一次迭代中会讨论并解决这些问题。`processEntries()`方法中的 `while` 循环值得关注。它处理当前队列中的所有消息。当拥有锁时，它将当前队列的内容与栈上的一个局部空队列交换。在此之后，它释放锁，这样其他线程就不会再被阻塞，从而向现在为空的当前队列添加新条目。一旦释放锁，就会处理局部队列的所有条目。这里不再需要锁，因为其他线程不会碰到这个局部队列。

```

Logger::Logger()
{
    // Start background thread.
    m_thread = thread { &Logger::processEntries, this };
}

void Logger::log(string entry)
{
    // Lock mutex and add entry to the queue.
    unique_lock lock { m_mutex };
    m_queue.push(move(entry));
    // Notify condition variable to wake up thread.
    m_condVar.notify_all();
}

void Logger::processEntries()
{
    // Open log file.
    ofstream logFile { "log.txt" };
    if (logFile.fail()) {
        cerr << "Failed to open logfile." << endl;
        return;
    }

    // Create a lock for m_mutex, but do not yet acquire a lock on it.
    unique_lock lock { m_mutex, defer_lock };
    // Start processing loop.
    while (true) {
        lock.lock();

        // Wait for a notification.
        m_condVar.wait(lock);
    }
}

```

```

        // Condition variable is notified, so something might be in the queue.

        // While we still have the lock, swap the contents of the current queue
        // with an empty local queue on the stack.
        queue<string> localQueue;
        localQueue.swap(m_queue);

        // Now that all entries have been moved from the current queue to the
        // local queue, we can release the lock so other threads are not blocked
        // while we process the entries.
        lock.unlock();

        // Process the entries in the local queue on the stack. This happens after
        // having released the lock, so other threads are not blocked anymore.
        processEntriesHelper(localQueue, logFile);
    }
}

void Logger::processEntriesHelper(queue<string>& queue, ofstream& ofs) const
{
    while (!queue.empty()) {
        ofs << queue.front() << endl;
        queue.pop();
    }
}

```

警告：

从这个相当简单的任务中可看到，正确编写多线程代码是十分困难的。令人遗憾的是，目前，C++标准不提供任何并发数据结构，至少目前是这样的。

Logger 类是一个演示基本构建块的示例。对于生产环境中的代码而言，建议使用恰当的第三方并发数据结构，不要自行编写。例如，开源的 Boost C++库(boost.org)实现了一个无锁队列，允许并发使用，不需要任何显式的同步。

可通过下面的测试代码测试这个 Logger 类，这段代码启动一些线程，所有线程都向同一个 Logger 实例记录一些信息：

```

void logSomeMessages(int id, Logger& logger)
{
    for (int i { 0 }; i < 10; ++i) {
        logger.log(format("Log entry {} from thread {}", i, id));
    }
}

int main()
{
    Logger logger;
    vector<thread> threads;
    // Create a few threads all working with the same Logger instance.
    for (int i { 0 }; i < 10; ++i) {
        threads.emplace_back(logSomeMessages, i, ref(logger));
    }
    // Wait for all threads to finish.
    for (auto& t : threads) {
        t.join();
    }
}

```



```
    }
}
```

如果构建并运行这个原始的最初版本，你会发现应用程序突然终止。原因在于应用程序从未调用后台线程的 `join()` 或 `detach()`。回顾本章前面的内容可知，`thread` 对象的析构函数仍是可结合的，即尚未调用 `join()` 或 `detach()`，而调用 `std::terminate()` 来停止运行线程和应用程序本身。这意味着，仍在队列中的消息未写入磁盘文件。当应用程序像这样终止时，甚至一些运行时库会报错或生成崩溃转储。需要添加一种机制来正常关闭后台线程，并在应用程序本身终止之前，等待后台线程完全关闭。这可通过向类中添加一个析构函数和一个布尔数据成员来解决。新的 `Logger` 类定义如下所示：

```
export class Logger
{
public:
    // Gracefully shut down background thread.
    virtual ~Logger();
    // Other public members omitted for brevity.
private:
    // Boolean telling the background thread to terminate.
    bool m_exit { false };
    // Other members omitted for brevity.
};
```

析构函数将 `m_exit` 设置为 `true`，唤醒后台线程，并等待直到后台线程被关闭。把 `m_exit` 设置为 `true` 之前，析构函数在 `m_mutex` 上获得一个锁。这是在使用 `processEntries()` 防止争用条件和死锁。`processEntries()` 可以放在其 `while` 循环的开头，即检查 `m_exit` 之后、调用 `wait()` 之前。如果主线程此时调用 `Logger` 类的析构函数，而析构函数没有在 `m_mutex` 上获得一个锁，则析构函数在 `processEntries()` 检查 `m_exit` 之后、等待条件变量之前，把 `m_exit` 设置为 `true`，并调用 `notify_all()`，因此 `processEntries()` 看不到新值，也收不到通知。此时，应用程序处于死锁状态，因为析构函数在等待 `join()` 调用，而后台线程在等待条件变量。注意析构函数可以在仍然持有锁或释放锁之后调用 `notify_all()`，但必须在 `join()` 调用之前释放锁，这解释了使用花括号的额外代码块。

```
Logger::~~Logger()
{
    {
        unique_lock lock { m_mutex };
        // Gracefully shut down the thread by setting m_exit to true.
        m_exit = true;
    }
    // Notify condition variable to wake up thread.
    m_condVar.notify_all();
    // Wait until thread is shut down. This should be outside the above code
    // block because the lock must be released before calling join()!
    m_thread.join();
}
```

`processEntries()` 方法需要检查此布尔变量，当这个布尔变量为 `true` 时终止处理循环：

```
void Logger::processEntries()
{
    // Omitted for brevity.

    // Create a lock for m_mutex, but do not yet acquire a lock on it.
    unique_lock lock { m_mutex, defer_lock };
    // Start processing loop.
```

```

while (true) {
    lock.lock();

    if (!m_exit) { // Only wait for notifications if we don't have to exit.
        m_condVar.wait(lock);
    } else {
        // We have to exit, process the remaining entries in the queue.
        processEntriesHelper(m_queue, logFile);
        break;
    }

    // Condition variable is notified, so something might be in the queue
    // and/or we need to shut down this thread.

    // Remaining code omitted, same as before.
}
}

```

注意不能只在外层 while 循环的条件中检查 m_exit，因为即使 m_exit 是 true，队列中也可能有需要写入的日志项。

可在多线程代码的特殊位置添加人为的延迟，以触发某个行为。注意添加这种延迟应仅用于测试，并且应从最终代码中删除。例如，要测试是否解决了析构函数带来的争用条件，可在主程序中删除对 log() 的所有调用，使其几乎立即调用 Logger 类的析构函数，并添加如下延迟：

```

void Logger::processEntries()
{
    // Omitted for brevity.

    // Create a lock for m_mutex, but do not yet acquire a lock on it.
    unique_lock lock { m_mutex, defer_lock };
    // Start processing loop.
    while (true) {
        lock.lock();

        if (!m_exit) { // Only wait for notifications if we don't have to exit.
            this_thread::sleep_for(1000ms); // Needs import <chrono>;
            m_condVar.wait(lock);
        } else {
            // We have to exit, process the remaining entries in the queue.
            processEntriesHelper(m_queue, logFile);
            break;
        }

        // Remaining code omitted, same as before.
    }
}
}

```

27.11 线程池

如果不在程序的整个生命周期中动态地创建和删除线程，还可以创建可根据需要使用的线程池。这种技术通常用于需要在线程中处理某类事件的程序。在大多数环境中，线程的理想数目应该和处理器的核心数目相等。如果线程的数目多于处理器核心的数目，那么线程只有被挂起，从而允许其他线程运行，这样最终会增加开销。注意，尽管理想的线程数目和核心数目相等，但这种情况只适用于计

算密集型线程，这种情况下线程不能由于其他原因阻塞，例如 I/O。当线程可以阻塞时，往往运行数目比核心数目更多的线程更合适。在此类情况下，确定最佳线程数难度较大，可能涉及测量系统正常负载条件下的吞吐量。

由于不是所有的处理都是等同的，因此线程池中的线程经常接收一个表示要执行的计算的函数对象或 lambda 表达式作为输入的一部分。

由于线程池中的所有线程都是预先存在的，因此操作系统调度这些线程并运行的效率大大高于操作系统创建线程并响应输入的效率。此外，线程池的使用允许管理创建的线程数，因此根据平台的不同，可以少至 1 个线程，也可以多达数千个线程。

有几个库实现了线程池，例如 Intel Threading Building Blocks(TBB)、Microsoft Parallel Patterns Library(PPL)等。建议给线程池使用这样的库，而不是编写自己的实现。如果的确希望自己实现线程池，可以使用与对象池类似的方式实现。第 29 章“编写高效 C++代码”将列举一个对象池的实现示例。



27.12 协程

协程是一个可以在执行过程中挂起并在稍后的时间点恢复的函数。任何函数体中包含以下任一项都是协程：

- **co_await**：在等待一个计算完成时挂起一个协程的执行。当计算完成后，继续执行。
- **co_return**：从协程返回。在此之后，协程无法恢复。
- **co_yield**：从协程返回一个值给调用者，并挂起协程，随后再次调用协程，在它被挂起的地方继续执行。

通常有两种类型的协程：有栈协程和无栈协程。有栈协程可以从嵌套调用内部的任何地方挂起。另一方面，无栈协程只能从顶层栈帧挂起。当无栈协程挂起时，只保存函数体中具有自动存储时间的变量和临时变量；不保存调用栈。因此，无栈协程的内存使用非常少，允许数百万甚至数十亿的协程同时运行。C++只支持无栈协程的变体。

协程可以使用同步编程风格来实现异步操作。用例包括以下内容：

- 生成器(Generators)
- 异步 I/O(Asynchronous I/O)
- 延迟计算(Lazy computations)
- 事件驱动程序(Event-driven applications)

C++20 标准只提供了协程构建块，也就是语言的补充。

C++20 标准库没有提供任何标准化的高级协程，比如生成器。有一些第三方库确实提供了这样的协程，比如 `cppcoro`。Microsoft Visual C++ 2019 还提供了一些更高层次的结构，比如一个实验性生成器。以下代码演示了 Visual C++ 2019 `std::experimental::generator` 协程的使用：

```
experimental::generator<int> getSequenceGenerator(
    int startValue, int numberOfValues)
{
    for (int i { startValue }; i < startValue + numberOfValues; ++i) {
        // Print the local time to standard out, see Chapter 22.
        time_t tt { system_clock::to_time_t(system_clock::now()) };
        tm t;
        localtime_s(&t, &tt);
        cout << put_time(&t, "%H:%M:%S") << " ";
        // Yield a value to the caller, and suspend the coroutine.
        co_yield i;
    }
}
```

```

    }
}

int main()
{
    auto gen { getSequenceGenerator(10, 5) };
    for (const auto& value : gen) {
        cout << value << " (Press enter for next value)";
        cin.ignore();
    }
}

```

运行程序会得到以下输出：

```
16:35:42: 10 (Press enter for next value)
```

按 Enter 键添加另一行：

```
16:35:42: 10 (Press enter for next value)
16:36:03: 11 (Press enter for next value)
```

再按 Enter 键会添加另一行：

```
16:35:42: 10 (Press enter for next value)
16:36:03: 11 (Press enter for next value)
16:36:21: 12 (Press enter for next value)
```

每次按 Enter 键时，生成器都会请求一个新值。这会导致协程继续执行，在 `getSequenceGenerator()` 中执行 for 循环的下次迭代，输出本地时间，并返回下一个值。返回值通过 `co_yield` 完成，`co_yield` 返回值，然后挂起协程。在 `main()` 函数中打印值本身，后面跟着要按下 Enter 键获取下一个值的问题。输出清楚地显示了协程被挂起并多次恢复。

但是，在这本书中，这几乎是关于协程的所有内容。自己编写协程，例如 `experimental::generator`，是相当复杂的，而且在本书中讨论过于高级。我只想提一下这个概念，以便你们知道它的存在，也许将来的 C++ 标准会引入标准化的协程。

27.13 线程设计和最佳实践

本节简要介绍几个有关多线程编程的最佳实践。

- **使用并行标准库算法：**标准库中包含大量算法。从 C++17 开始，有 60 多个算法支持并行执行。尽量使用这些并行算法，而非编写自己的多线程代码。可参阅第 20 章，以详细了解如何为算法指定并行选项。
- **终止应用程序前，确保所有 thread 对象都不是可结合的：**确保对所有 thread 对象都调用了 `join()` 或 `detach()`。仍可结合的 thread 析构函数将调用 `std::terminate()`，从而突然间终止所有线程和应用程序。C++20 引入了 `jthread`，它在析构函数中自动 `join()`。
- **最好的同步就是没有同步：**如果采用合理的方式设计不同的线程，让所有的线程在使用共享数据时只从共享数据读取，而不写入共享数据，或者只写入其他线程不会读取的部分，那么多线程编程就会变得简单很多。这种情况下不需要任何同步，也不会有争用条件或死锁的问题。
- **尝试使用单线程的所有权模式：**这意味着同一时间拥有 1 个数据块的线程数不多于 1。拥有数据意味着不允许其他任何线程读/写这些数据。当线程处理完数据时，数据可传递到另一个线程，那个线程目前拥有这些数据的唯一且完整的责任/拥有权。这种情况下，没必要进行同步。

- **在可能时使用原子类型和操作**：通过原子类型和原子操作更容易编写没有争用条件和死锁的代码，因为它们能自动处理同步。如果在多线程设计中不可能使用原子类型和操作，而且需要共享数据，那么需要使用同步机制(如互斥)来确保同步的正确性。
- **使用锁保护可变的共享数据**：如果需要多个线程可写入的可变共享数据，而且不能使用原子类型和操作，那么必须使用锁机制，以确保不同线程之间的读写是同步的。
- **尽快释放锁**：当需要通过锁保护共享数据时，务必尽快释放锁。当一个线程持有一个锁时，会使得其他线程阻塞等待这个锁，这可能会降低性能。
- **不要手动获取多个锁，应当改用 `std::lock()` 或 `std::try_lock()`**：如果多个线程需要获取多个锁，那么所有线程都要以同样的顺序获得这些锁，以防止死锁。可通过泛型函数 `std::lock()` 或 `std::try_lock()` 获取多个锁。
- **使用支持多线程的分析器**：通过支持多线程的分析器找到多线程应用程序中的性能瓶颈，分析多个线程是否确实利用了系统中所有可用的处理能力。支持多线程的分析器的一个例子是某些 Visual Studio 版本中的 profiler。
- **使用 RAII 锁对象**：使用 `lock_guard`、`unique_lock`、`shared_lock` 或 `scoped_lock` RAII 类，在正确的时间自动释放锁。
- **了解调试器的多线程支持特性**：大部分调试器都提供对多线程应用程序调试的最基本支持。应该能得到应用程序中所有正在运行的线程列表，而且应该能切换到任意线程，查看线程的调用栈。例如，可通过这些特性检查死锁，因为可准确地看到每个线程正在做什么。
- **使用线程池，而不是动态创建和销毁大量线程**：动态地创建和销毁大量的线程会导致性能下降。这种情况下，最好使用线程池来重用现有的线程。
- **使用高级多线程库**：目前，C++标准仅提供用于编写多线程代码的基本构件。正确使用这些构件并非易事。尽可能使用高级多线程库，例如 Intel Threading Building Blocks(TBB)、Microsoft Parallel Patterns Library(PPL)等，而不是自己实现。多线程编程很难掌握，而且容易出错。另外，自己的实现不一定像预期那样正确工作。

27.14 本章小结

本章简要介绍了如何通过标准 C++线程库进行多线程编程。你学习了如何使用 `std::thread` 启动线程，以及 C++20 如何使用 `std::jthread` 更容易地编写可取消的线程。解释了如何通过原子类型和原子操作使用共享数据，而不是使用显式的同步机制。你学习了在不能使用这些原子类型和操作的情况下，如何使用互斥机制，确保需要读写访问共享数据的不同线程之间的正确同步。你了解了 C++20 的新的同步原语：semaphore、latch、barrier。你还学习了如何通过 promise 和 future 表示线程间的通信信道，通过 future 可更简单地从后台线程获得结果。本章最后介绍了多线程应用程序设计的一些最佳实践。

正如本书开头所述，本章试图涵盖标准 C++线程库提供的所有基本多线程构件，但由于篇幅受限，不能涉及多线程编程的所有细节。有很多书籍专门讨论多线程，请在附录 B 中查找参考文献。

27.15 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己

找到答案。

练习 27-1 编写一个每 3 秒发出一次蜂鸣声的程序。提示：可以通过在标注输出中写入[\a](#) 来让电脑发出蜂鸣的声音。

练习 27-2 修改练习 27-1 的解决方案，以便在用户按下 Enter 键时可以停止程序。

练习 27-3 修改练习 27-1 的解决方案，使蜂鸣声继续，直到用户按下 Enter 键。一旦按下 Enter 键，蜂鸣声应该暂停，直到用户再次按下 Enter 键。用户可以按自己的意愿暂停和恢复蜂鸣声。

练习 27-4 编写一个程序，可以并发计算多个斐波那契数列。例如，你的代码应该能够并行计算斐波那契数列中的第 4、9、14、17 个数字。斐波那契数列从 0 和 1 开始，任何后面的值都是前面两个值的和，所以有：0、1、1、2、3、5、8、13、21、34、55、89，等等。一旦所有结果都可用，将它们输出到标准输出。最后，使用标准库算法计算它们的总和。

第V部分

C++软件工程

- ▶ 第 28 章 充分利用软件工程方法
- ▶ 第 29 章 编写高效的 C++ 程序
- ▶ 第 30 章 熟练掌握测试技术
- ▶ 第 31 章 熟练掌握调试技术
- ▶ 第 32 章 使用设计技术和框架
- ▶ 第 33 章 应用设计模式
- ▶ 第 34 章 开发跨平台和跨语言的应用程序

第28章

充分利用软件工程方法

本章内容

- 以瀑布模型、生鱼片模型、螺旋模型和敏捷模型等为例讲解软件生命周期模型的含义
- 以 UP(Unified Process)、RUP(Rational Unified Process)、Scrum、XP(eXtreme Programming)和软件分流等为例讲解软件工程方法学
- 源代码控制的含义

第 28 章“充分利用软件工程方法”开启了本书的最后一部分，此部分与软件工程相关。主要介绍软件工程的方法、代码效率、测试、调试、设计技术、设计模式以及如何定位多个平台。

最初开始学习编程时，可能按照自己的进度做事情。只要乐意，可能会在最后一分钟完成所有事情，也可能在实现的过程中彻底改变最初的设计。然而，在专业编程的世界中，程序员很少有这样的灵活性。即使最开明的工程管理人员也承认，一些过程是必要的。如今，了解软件工程的过程和了解如何编写代码同等重要。

本章分析软件工程的各种方法，但没有深入讲解任何一种方法——有大量关于软件工程过程的优秀书籍，而是广泛地覆盖不同类型的过程，以对比这些方法。我们尽量不提倡或阻止读者使用某些方法，而是希望通过权衡几种不同的方法，让读者构建适合自己 and 团队的过程。不论是独立完成项目的承包商，还是由几大洲数百名工程师组成的团队，理解软件开发中的不同方法对日常工作都是有帮助的。

本章最后讨论源代码控制解决方案，以便管理源代码并追溯源代码的历史。源代码控制解决方案是每家公司避免源代码维护噩梦的必备工具；即使对于由一人完成的项目，也强烈建议采用这种方案。

28.1 过程的必要性

软件开发的历史充满了失败项目的故事。从超过预算和销售不佳的消费类应用程序，到过分宣传令人感到天花乱坠的操作系统，看上去软件开发中的任何领域都逃不过这种趋势。

即使软件成功到了用户手中，bug 也可能无处不在，最终用户不得不经常升级和打补丁。有时软件不能完成预设的任务，有时不能按照用户期望的方式工作。所有这些问题都汇聚成软件的一条真理——写软件很难。

人们不禁要问，为什么软件工程出现故障的频率和其他类型的工程不同？汽车也有 bug，但是汽车很少突然停下来，或因为缓冲区溢出要求重启(不过你可能会说，汽车上越来越多的组件都是软件驱动的)。电视机可能并不完美，但不需要为了看 6 频道而将电视升级到版本 2.3。

是不是其他工程学科比软件工程更先进？市政工程师能否借鉴桥梁建筑的悠久历史构建可用的桥？化学工程师能不能因为大部分 bug 都已经在前几代中解决而成功合成某化合物？软件是不是太新了？还是因为软件是一门完全不同的学科，本质上就有产生 bug、不可用结果和失败项目的特质？

软件看上去肯定有所不同。一方面，软件技术快速更新，使软件开发过程中产生了不确定性。即使项目中没有发生惊天动地的突破，软件工业的步伐也会导致问题。另一方面，软件往往需要迅速开发，因为软件市场的竞争异常激烈。

软件开发的进度也是不可预测的。准确的时间表几乎是不可能的，一个让人讨厌的 bug 可能需要几天甚至几周的时间才能修复。即使事情看上去在遵循时间表进行，产品定义变化(功能渐变)的普遍趋势也可能给这个过程当头一棒。如果不加以控制，这种范围蔓延会导致软件膨胀。

软件是复杂的。没有简单准确的方法能证明程序是无 bug 的。如果需要进行多个版本的维护，有 bug 或凌乱的代码会对软件带来数年的影响。软件系统往往非常复杂，以至于员工流失时，没人愿意接手前任工程师留下来的凌乱代码。这将导致无休止的修补、乱改和变通方案。

当然，软件也要面对标准的业务风险。营销压力和错误的沟通也会出现。很多程序员都试图避开办公室斗争，但是开发团队和产品营销团队之间往往会发生一些矛盾。

所有这些影响软件工程产品的因素都表明需要某种过程。软件项目很大、很复杂而且步伐快。为避免失败，工程组必须采用能控制这种棘手过程的体系。

毫无疑问，可以开发出设计精当的软件，代码清晰，易于维护；但为达到这个目的，每个团队成员都需要持续努力，并遵循适当的软件开发过程和实践。

28.2 软件生命周期模型

软件中的复杂性并不是新事物。几十年前，人们就意识到需要一个形式化的过程。对软件生命周期建模的几种方法都试图给软件开发的混沌带来一些规则，这些方法根据从最初的想法到最终产品之间的各个步骤定义了软件过程。这些模型经过多年的完善，正指导当今的软件开发。

28.2.1 瀑布模型

经典的软件生命周期模型称为瀑布模型(Waterfall Model)。这种模型依据的思想是：软件几乎可以像遵循菜谱一样构建。有一组步骤，如果正确遵循这些步骤，将得到一份极好的巧克力蛋糕，在软件工程中就是得到程序。每个阶段都必须在下一阶段可以开始之前完成，如图 28-1 所示。可将这个过程比作瀑布，因为只能从上到下进入下一个阶段。

这个过程首先要进行正式的规划，包括收集详尽的需求列表。需求列表定义了产品的功能完整性。需求越具体，项目越有可能成功。接下来，进行软件设计和完整的规范设计。设计阶段和规划阶段一样，需要尽可能具体，才能尽量提高成功的机会。所有的设计决策都是在这个时候制定的，通常包括伪代码和特定子系统的定义。子系统的所有者制定出代码和外界交互的方式，然后整个团队都要遵循这个架构的规范。接下来实现这个设计。由于设计已经说明完整，因此代码必须严格地遵循设计要求，否则不同的代码块就无法整合在一起工作。最后 4 个阶段包括单元测试、子系统测试、整合测试和评估。

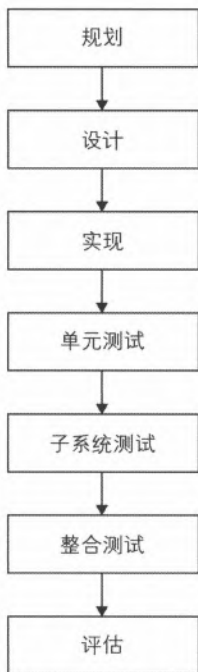


图 28-1 瀑布模型

瀑布模型的主要问题在于，在实践中，几乎不可能在完全不涉及下一个阶段的情况下完成前一个阶段。如果不写一点代码，设计很难完善。此外，如果这种模型不允许回到编码阶段，那么测试还有什么意义？

瀑布模型的各种变体采用不同的方式改进这个过程。例如，一些规划阶段包含“可行性分析”步骤，这个步骤在确定正式的需求之前通过实验进行可行性分析。

1. 瀑布模型的优点

瀑布模型的价值在于它的简单性。程序员或经理可能在之前的项目中就采取了这种方式，而没有对这种方式进行形式化或命名。瀑布模型背后的假设是：只要每个步骤都尽可能完整且准确地完成，后续步骤就会顺利进行。只要在第一步小心制定了所有需求，在第二步认真讨论了所有设计决策和问题，那么第三步中的实现只不过是设计翻译为代码而已。

瀑布模型的简单性使基于这种体系的项目规划更有组织、更易于管理。每个项目都以同样的方式开始：详尽列出所有必需的功能。例如，采用这种方法的管理人员可以在每个设计阶段结束时，要求负责某个子系统的所有工程师以正式设计文档或可用子系统规范的方式提交他们的设计。管理人员得到的好处是，要求工程师在前期定义好需求和设计，将风险最小化。

从工程师的角度看，瀑布模型迫使在前期解决各种重大问题。所有工程师在编写可观数量的代码之前，都需要理解项目并设计好子系统。理想情况下，这意味着代码只需要编写一次，而不需要通过修改将代码整合在一起，或者重写不能整合的代码。

对于需求非常具体的小项目而言，瀑布模型可工作得很好。特别是对于咨询安排来说，瀑布模型具有可以在项目开始时指定具体指标的优势。将需求形式化可以帮助咨询师准确理解客户的需求，并迫使客户更加具体地说明项目的目标。

2. 瀑布模型的缺点

在许多组织中，以及几乎所有的现代软件工程教材中，瀑布模型都已失宠。批评者贬低其基本前提，即软件工程采取的是离散的线性步骤。虽然瀑布模型允许阶段重叠，但是不允许大步后退。在如今很多项目中，需求贯穿整个产品开发过程。通常情况下，潜在的客户会要求有利于销售的功能，或者为了应对竞争对手的产品而增加新功能。

注意：

所有需求的前期规范使许多组织无法使用瀑布模型，因为这种模型不够动态。

另一个缺点是：由于尽早且形式化地做出决策，以努力降低风险，因此瀑布模型可能实际上隐藏了风险。例如，在设计阶段可能无法发现、遗忘或故意掩盖主要的设计问题；到集成测试时发现了不匹配问题，这时拯救这个项目已经为时已晚；出现严重的设计缺陷，但是根据瀑布模型，产品距离发布只有一步之遥。瀑布过程中的任何错误都可能导致过程结束时的失败。想要早期就发现既困难又罕见。

如果使用瀑布模型，那么经常需要从其他方法借鉴一些经验才能更灵活。

28.2.2 生鱼片模型

人们已经正式提出对瀑布模型的多种改进模型。其中一种改进模型称为生鱼片模型(Sashimi Model)。生鱼片模型的主要优点是引入了阶段之间重叠的概念。“生鱼片模型”得名于日本的鱼料理，在这种食物中，不同的鱼片相互重叠。尽量这种模型仍然强调严格的规划、设计、编码、测试过程，但相连的阶段可以部分重叠。图 28-2 是生鱼片模型的一个示例，演示了各个阶段的重叠。“重叠”允许两个阶段的活动同时发生。这是因为人们认识到，要真正完成一个阶段，通常必须分析下一阶段(至少是下一阶段的一部分)。

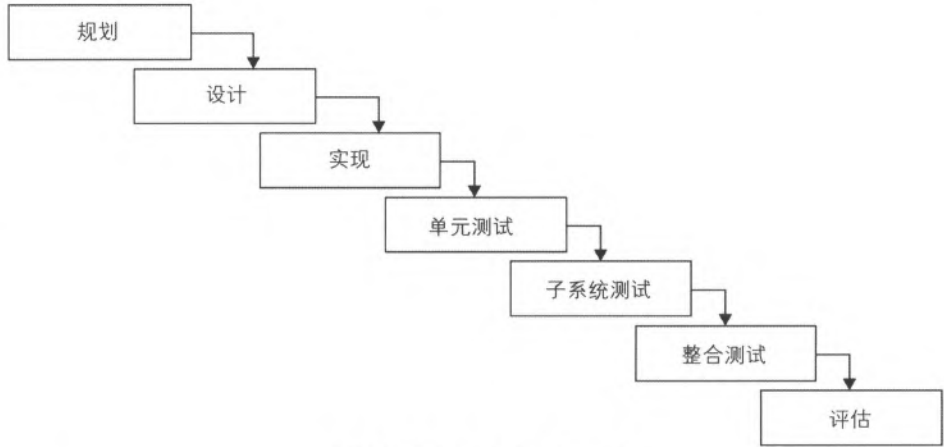


图 28-2 生鱼片模型的一个示例

28.2.3 螺旋类模型

螺旋模型(Spiral Model)在 1988 年由 Barry W. Boehm 提出为一种风险驱动的软件开发过程。还有其他螺旋模型，这些称为螺旋类模型。本节讨论的模型是迭代过程技术族的一部分。基本思想是：出错也没关系，因为在下一轮中会修复这个问题。螺旋类模型中的一次循环如图 28-3 所示。

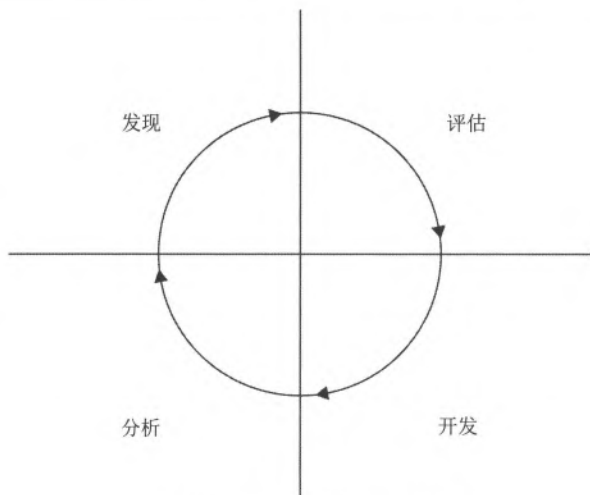


图 28-3 螺旋类模型中的一次循环

螺旋类模型中的阶段类似于瀑布模型中的步骤。在发现阶段构建需求，确定目标，确定替代方案（设计替代方案、购买第三方库等），确定其他约束。在评估阶段，会考虑评估替代方案，分析风险，构建原型系统。在螺旋类模型中，特别重视评估阶段风险的评估和解决。被视为风险最高的任务在螺旋类模型的当前周期的这个阶段实现。开发阶段的任务是根据评估阶段确定的风险决定的。例如，如果评估揭示了一个可能无法实现的具有风险的算法，那么当前周期中开发阶段的主要任务就是对这个算法进行建模、构建和测试。第 4 个阶段预留给分析和规划。在当前周期结果的基础上，形成下一个周期的计划。

图 28-4 展示了某操作系统的开发螺旋中的 3 个周期。第一个周期产生了一个规划，其中包含产品的主要需求。第二个周期得到了展示用户体验的原型。第三个周期构建了一个被认定为高风险的组件。

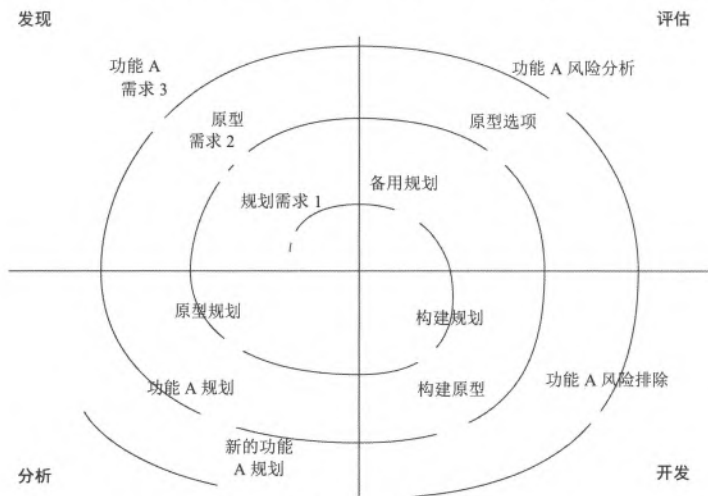


图 28-4 开发螺旋中的 3 个周期

1. 螺旋类模型的优点

螺旋类模型可看成迭代方法的应用，具有瀑布模型所能提供的优点。图 28-5 将螺旋类模型展示

为已修改为允许迭代的瀑布模型。通过迭代循环解决了瀑布模型的主要缺点：隐藏的风险和线性开发路径。

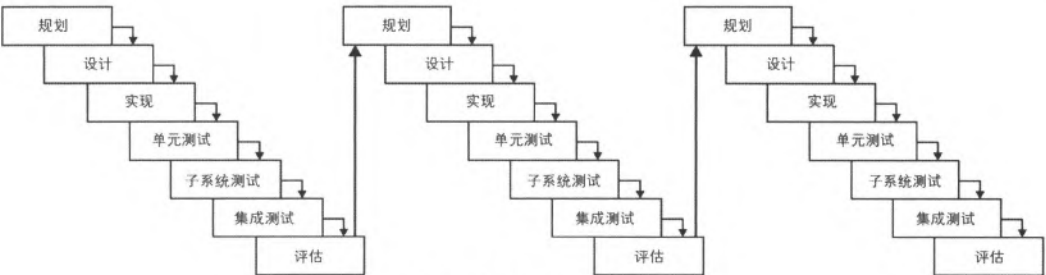


图 28-5 允许迭代的瀑布模型

首先，执行风险最大的任务是另一个好处。通过将风险提前，并承认新的情况可能随时出现，螺旋类模型避免了瀑布模型中隐藏的定时炸弹。当出现意想不到的问题时，可通过同样的 4 阶段方法解决，而过程中剩下的部分也通过这种方法完成。

其次，这种迭代方法还允许纳入测试人员的反馈。例如，产品的早期版本可在内部甚至外部发布供评估。例如，测试人员可报告某个功能丢失，或现有的功能不能按照预期的方式工作。螺旋类模型有内置的机制应对这种输入。

最后，通过每个周期后的反复分析和新设计的构建，消除了先设计再实现面临的实际困难。在每个周期中，更多关于系统的知识可以影响设计。

2. 螺旋类模型的缺点

螺旋类模型的主要缺点是，很难将每次迭代的范围界定得足够小，以获得真正的好处。在最糟糕的情况下，螺旋类模型可能因为迭代太长而退化为瀑布模型。遗憾的是，螺旋类模型只是对软件生命周期建模，不能规定一种方式来将项目分解为单周期的迭代，因为这种分解因项目而异。

其他可能的缺点是：每个周期重复这 4 个阶段的开销以及周期之间协调所面临的困难。从统筹角度看，可能很难在正确的时间召集所有的组员进行设计讨论。如果不同的团队同时开发产品中不同的部分，那么他们有可能在并行的周期中工作，而这些周期可能会不同步。例如，在开发操作系统期间，用户界面小组可能准备好开始窗口管理周期的发现阶段，但是核心操作系统小组可能仍处在内存子系统的开发阶段。

28.2.4 敏捷

为了消除瀑布模型的缺点，2001 年以敏捷软件开发宣言的形式推出了敏捷方法。

敏捷软件开发宣言

整个宣言全文如下(<http://agilemanifesto.org/>):

我们一直在实践中探寻更好的软件开发方法，身体力行的同时也帮助他人。由此，我们建立了如下价值观：

- 个体和互动高于流程和工具
- 可用的软件高于详尽的文档
- 客户合作高于合同谈判
- 响应变化高于遵循计划

也就是说，尽管每一项的后者都有价值，但我们更重视前者的价值。

根据对这份宣言的理解,敏捷这个词只是一种高层次的描述。这份宣言表达的意思基本上是:让软件开发的过程灵活,使客户需求的变化很容易在开发过程中融入项目。Scrum 是其中一种最常用的敏捷软件开发方法。

28.3 软件工程方法论

软件生命周期模型提供了回答“下一步该怎么办?”这个问题的正式方式,但是很少能够回答接下来的一个问题:“如何做这件事情?”为了回答这个问题,人们开发了很多方法论,为专业的软件开发提供了经验法则。有关软件方法论的书籍和文章比比皆是,但只有很少几个软件开发方法论值得关注:UP、RUP、Scrum、极限编程和软件分流。

28.3.1 UP

UP(Unified Process, 统一过程)是一种迭代和增量软件开发过程。UP 并非一成不变,它是一个框架,需要根据项目的特定需求进行定制。按照 UP,可将一个项目分为以下 4 个阶段。

- **起始阶段(Inception):** 该阶段通常很短暂。它包括可行性研究,编写业务案例,决定由内部开发还是交给第三方供应商,大致确定成本和时间线,定义范围。
- **细化阶段(Elaboration):** 把大多数需求都记录下来。消除风险因素,验证系统架构。为验证架构,将架构核心的最重要部分构建为可执行的交付品。这应当证实:开发的架构可以支持整个系统。
- **构建阶段(Elaboration):** 在细化阶段得到了可执行的架构交付品,该阶段将在此基础上满足所有要求。
- **交付阶段(Transition):** 将产品交付给客户。根据客户的反馈,在后续的交付迭代中进行处理。

将细化阶段、构建阶段和交付阶段分为多个时间固定的迭代,每个迭代都得到有形的结果。在每个迭代中,团队同时处理项目的多项工作:业务建模、需求、分析和设计、实现、测试、部署。在每个迭代中完成的每项工作的量是不同的。图 28-6 描述了这种迭代和重叠开发过程。在这个示例中,起始阶段有一个迭代,细化阶段有两个迭代,构建阶段有四个迭代,交付阶段有两个迭代。

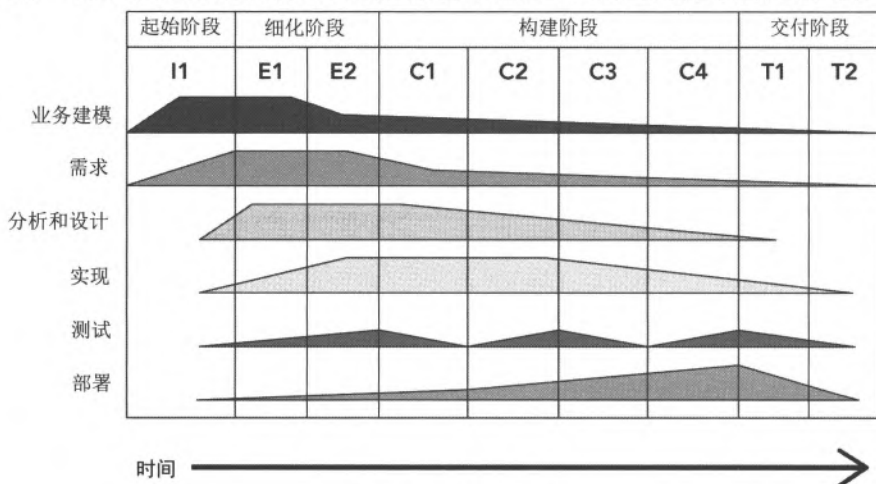


图 28-6 迭代和重叠开发过程

28.3.2 RUP

RUP(Rational Unified Process, Rational 统一过程)是统一过程(UP)的最著名改良之一,是管理软件开发过程的一种十分严格且规范的方法。RUP 最重要的特征在于,RUP 不仅是理论上的过程模型,这一点与螺旋模型和瀑布模型不一样。实际上,RUP 是软件产品,由 IBM 公司的 Rational Software 部门负责营销。把过程当成软件看待很吸引人,这有几个原因:

- 过程本身可以更新和完善,就像软件产品的定期更新。
- RUP 不仅提出了开发框架,还包含一组使用这个开发框架的软件工具。
- 作为产品,RUP 可部署在整个工程团队,使所有成员都使用完全相同的过程和工具。
- 与许多软件产品一样,RUP 可定制,以满足用户的需求。

1. RUP 作为产品

作为产品,RUP 采用的是软件应用程序套件的形式,在软件开发过程中指导开发人员。RUP 产品还提供了其他 Rational 产品的具体指导,例如 Rational Rose 可视化建模工具和 Rational ClearCase 配置管理工具。广泛的群件通信工具作为“思想的集市”的一部分,使开发人员可以共享知识。

RUP 背后的基本原则之一是:开发周期的每次迭代应该得到有形的结果。在 RUP 中,用户会创建很多设计、需求文档、报告和计划。RUP 软件提供了创建这些产物的可视化工具和开发工具。

2. RUP 作为过程

定义精确模型是 RUP 的核心原则。根据 RUP,模型能在软件开发过程中帮助解释复杂的结构和关系。在 RUP 中,模型通常以 UML(统一建模语言)格式描述。

RUP 将过程中的每个部分定义为独立的工作流。工作流表示过程中的每个步骤,表示的方式为:这个步骤的负责人、这个步骤要执行的任务、这些任务要产生的产物或结果,以及驱动任务的事件序列。几乎 RUP 的一切都是可定制的,但是 RUP 定义了一些核心的过程工作流。

核心的过程工作流和瀑布模型的阶段有一些相似之处,但每个都是可迭代的,而且定义更具体。业务建模工作流对业务流程建模,通常带有驱动软件需求的目标。需求工作流通过分析系统中的问题和遍历系统假设来创建需求定义。分析和设计工作流处理系统架构和子系统设计。实现工作流包括建模、编码和软件子系统的集成。测试工作流对软件质量测试的计划、实现和评估进行建模。部署工作流是整体规划、发布、支持和测试工作流的高层视图。配置管理工作流涉及从新的项目概念乃至迭代和最终产品的情况。最后,环境工作流通过创建和维护开发工具对工程组织提供支持。

3. 实践中的 RUP

RUP 主要面向大型组织,相比传统的生命周期模型具有一些优势。一旦团队掌握这个软件的使用法,所有成员就在统一的平台上设计、交流和实现他们的想法。这个过程可以定制,以满足团队的需求,每个阶段都展示了大量有价值的记录每个开发阶段的产物。

对于某些组织来说,像 RUP 这样的产品的量级过重。采用不同开发环境或开发预算紧张的团队可能不想或无法标准化基于软件的开发系统。学习难度也可能是因素之一——不熟悉过程软件的工程师在跟上产品进度和已有代码库的同时,还要学习如何使用 RUP 软件。

28.3.3 Scrum

敏捷模型没有规定在现实世界中应该如何实现模型。这就是 Scrum 的用途所在,Scrum 详细定义了日常使用的方法。

Scrum 是迭代过程,是一种非常流行的管理软件开发项目的手段。在 Scrum 中,每个迭代称为 sprint 周期。sprint 周期是 Scrum 过程的核心部分。sprint 周期的长度应在项目开始时决定,通常是两到四个星期。在每个 sprint 周期结束时,目标是要有一个完全可用且经过测试的软件版本,这个软件版本要表示客户需求的一个子集。Scrum 认识到,客户会在开发过程中经常改变主意,因此允许将每个 sprint 周期的结果发布给客户,让客户有机会看到软件的迭代版本,并允许他们将潜在的问题反馈给开发团队。

1. 角色

Scrum 有 3 个角色。第一个角色是 Product Owner(PO),起到客户和其他人之间桥梁的作用。PO 根据来自用户的描述编写高层次的用户故事,给每个用户故事设置优先级,然后把这些用户故事放在 Scrum 的产品需求总表中。事实上,团队中的每个成员都允许在 product 需求总表中编写高层次的用户故事,但是 PO 决定哪些用户故事可以留下,哪些用户故事要删除。

第二个角色 Scrum Master(SM)负责保持过程运行,可以是团队的一部分,不过不是团队领导,因为在 Scrum 中团队自己领导自己。SM 是团队的联络人,使团队的其他成员可专注于他们的任务。SM 确保团队正确地遵循 Scrum 过程;例如,组织日常 Scrum 会议。SM 和 PO 应由两个不同的人担任。

在 Scrum 过程中,第三个角色就是团队本身。团队负责开发软件,应该保持精简,最好不多于 10 个成员。

2. 过程

Scrum 过程强制每日例会,称为 Daily Scrum 或 Standup。在会议上,所有团队成员和 Scrum Master 坐在一起。根据 Scrum 过程,这个会议应该每天在同一时间、同一地点召开,而且不能长于 15 分钟。在会议期间,所有团队成员回答 3 个问题:

- 自从上次 Daily Scrum 会议以来做了什么?
- 在这次 Daily Scrum 会议后打算做什么?
- 为达到目标而面临什么问题?

Scrum Master 应该注意到团队成员面临的问题,在 Daily Scrum 会议之后应该尝试解决这些问题。

在每个 sprint 周期开始前都有一场 Sprint Planning 会议,在这场会议上,团队成员必须决定在新的 sprint 周期中要实现什么产品特性。这些产品特性正式记录在 sprint 需求总表(backlog)中。这些产品特性从带有优先级的用户故事的产品需求总表中挑选出来,产品需求总表保存的是新特性的高层次需求。从产品需求总表中取出的用户故事被分解为更小的任务,每个任务都带有工作量估计,并放在 sprint 需求总表中。一旦 sprint 周期开始,sprint 需求总表就被冻结,在这个 sprint 周期中不能修改。Sprint Planning 会议的时长取决于 sprint 周期的长度,如果 sprint 为期两周,则 Sprint Planning 会议应当为 4 小时。Sprint Planning 会议通常分为两部分:PO 和团队讨论产品需求总表中项的优先级,而团队本身的会议讨论 sprint 需求总表的完善。

在 Scrum 团队中,经常会发现一个真正的公告板,上面有 3 列:To Do(计划事项)、In Progress(开发中)和 Done(完成)。sprint 周期中的每个任务都写在一张小纸片上,贴在正确的列中。在会议上并不把任务分配到个人;相反,每个团队成员都可以从公告板上挑选一个 To Do 任务,然后把这个任务的纸片放在 In Progress 列中。当团队成员完成这一任务时,将纸片移到 Done 列中。这种方法使团队成员很容易了解工作的概况,了解哪些任务还需要完成、哪些任务正在进行、哪些任务已经完成。除真实的公告板外,还可使用软件解决方案提供虚拟的 Scrum 公告板。

To Do、In Progress 和 Done 这三列不是一成不变的。团队可以添加任何附加列来包含其他步骤。例如,Scrum 公告板可以包含以下列:

- To Do: 为当前 sprint 计划的尚未开始的任务。
- In Progress: 开发分支（参见本章后面“源代码控制”一节）中开发人员正在处理的任务。
- In Review: 已经实现的任务，以及正在等待另一个团队成员审查的任务，也成为 code review。
- In Testing: 已经实现过并且经过了代码评审，等待质量保证（QA）团队进行测试来获得批准的任务。
- In Integration: 代码修改已经通过评审验证并得到 QA 批准的任务，开发分支的代码可以集成到主代码库中。
- Done: 已经完全实现、评审、测试和集成的任务。

通常每天还会创建一幅 burn-down 图，在横轴上显示 sprint 周期的天数，在纵轴上显示剩下的开发小时数。这幅图可以快速概括进展，还可以用于判断 sprint 周期内所有计划的任务是否都完成了。

完成一个 sprint 周期时，有两个会议要开：Sprint Review 和 Sprint Retrospective 会议。Sprint Review 会议的时长同样取决于 sprint 的长度；如果 sprint 为两周，则会议时长通常为两小时。在 Sprint Review 会议上，会向所有感兴趣的人演示 sprint 结果和当前的软件状态。该会议还包括对 sprint 结果的讨论，包括已经完成任务、没有完成的任务以及原因。如果 sprint 为期两周，Sprint Retrospective 会议通常是 1.5 小时左右，允许团队思考前一个 sprint 周期的执行状况。例如，团队可指出过程中的缺点，并调整下一个 sprint 的过程。还要回答如下问题：“哪些进展顺利？”“哪些可以改进？”“要开始做什么，继续做什么，停止做什么？”。这称为持续改进，也就是说，在每个 sprint 后，都会分析和改进过程。

3. Scrum 的优点

Scrum 可弹性处理开发过程中遇到的不可预知的问题。当出现一个问题时，可以在后面的 sprint 中解决。团队参与项目的每一步。团队和 PO 讨论产品需求总表中的用户故事，并将用户故事转换为更小的任务，包含在 sprint 需求总表中。团队在 Scrum 任务公告板的帮助下自动将工作分配给成员。这个公告板便于快速看到团队成员的任务。最后，Daily Scrum 会议确保每个人都知道发生了什么。

对于付费用户的一个巨大好处是每个 sprint 后的 demo，demo 演示了项目的新迭代版本。客户可快速了解项目进展情况，对需求进行修改，对需求的修改通常可以融入未来的 sprint 中。

4. Scrum 的缺点

有些公司可能难以接受团队自己决定应该做什么。任务不是通过经理或团队管理者分配给团队成员的。所有成员都从 Scrum 任务公告板中挑选自己的任务。

Scrum Master(SM)是确保团队正常运作的关键人物。SM 对团队的信任非常重要。对团队成员控制过紧会导致 Scrum 过程失败。

Scrum 可能出现一个称为特性蠕动(feature creep)的问题。Scrum 允许在开发过程中向产品需求总表中添加新的用户故事。如果项目经理向产品需求总表中不断地添加新特性，则会出现危险的情况。这个问题最好的解决方法是：制定最终的发布日期，或最后一个 sprint 的终止日期。

28.3.4 极限编程

几年前，笔者的一位朋友下班回家告诉他的妻子，他们公司采取了一些极限编程(eXtreme Programming, XP)的方法，他的妻子开玩笑说“我希望你系上安全带。”尽管这个名字听上去有些做作，但极限编程实际上将现有最好的软件开发指导方针和新材料结合成了一种日益流行的方法。

XP 是 Kent Beck 在 *eXtreme Programming eXplained*(Addison-Wesley, 1999)一书中宣传的方法,号称采用了优秀软件开发的最佳实践,并且表现出色。例如,大多数程序员都认可:测试很重要。在 XP 中,测试被认为非常重要,应该在编写代码之前编写测试。

1. XP 理论

极限编程方法由 12 条主要的指导原则组成,分为 4 类。这些原则体现在软件开发过程的各个阶段,对工程师的日常任务有直接影响。

1) 精细的反馈

XP 提供与编码、规划和测试相关的 4 条细粒度指导原则。

结对编程

XP 建议,所有生产代码都应该由两个人同时编写,这种方式称为结对编程。显然,只有一个人真正负责编码。另一个人审查同伴编写的代码,采取一种高层次的方法,思考诸如测试、必要的重构和项目的整体模型等问题。

例如,假设要编写应用程序中某个功能的用户界面,就可能会询问这个功能的原始作者,而原始作者就坐在你身边。他会说明这个功能的正确用法,警告任何需要注意的“陷阱”,并从更高的层次监督你的工作。即使无法从原始作者那里获得帮助,找团队中的另一个成员也是有帮助的。该理论认为:结对工作可以构建共享的知识,确保正确的设计,构建非正式的互相制约的系统。

计划策略

在瀑布模型中,只在过程开始时做一次计划。在螺旋模型中,计划是每个迭代的第一阶段。在 XP 下,计划不仅是步骤,而且是永无止境的任务。XP 团队从一个粗略的计划开始,捕获正在开发的产品要点。在该过程的每个迭代中,有所谓的“计划策略”(planning game)会议。在整个开发过程中,计划根据需求不断地完善和修改。这个理论就是:条件在不断地变化,新的信息也在不断获取。计划过程包含两个主要部分:

- **版本计划:** 与开发人员和客户相关,旨在确定未来的版本需要包含哪些要求。
- **迭代计划:** 仅与开发人员相关,它规划开发人员的实际任务。

在 XP 下,对某个功能的估计总是由实现这个功能的人完成。这有助于避免实现人员被迫服从一张不真实的人造进度表的情况。最初的估计很粗糙,可能以周为单位安排特性。随着时间跨度的缩短,将能更精细地进行评估。功能被分解成不超过 5 天的任务。

不断测试

根据 *eXtreme Programming eXplained* 中的说法,“程序中任何不带自动化测试的功能等于不存在。”极限编程特别关心测试。XP 工程师的部分责任就是为代码编写单元测试。单元测试通常是一小块代码,以确保独立的功能正常工作。例如,基于文件的对象存储的独立单元测试可能包括 `testSaveObject`、`testLoadObject` 和 `testDeleteObject`。

XP 的单元测试更进一步,建议单元测试在实际代码之前编写。当然,测试不会通过,因为尚未编写代码。从理论上讲,如果测试是完善的,就应该知道什么时候代码才算完成,因为此时所有测试都能通过。这称为 TDD(Test Driven Development, 测试驱动开发),所以说这是“极限”。

有客户在现场

由于精明的 XP 工程团队会不断地完善产品计划,只构建当前有必要的功能,因此客户对这个过程的贡献非常有价值。尽管不可能说服客户总是停留在开发现场,但是工程师和最终用户之间有必要沟通的思想显然是非常有价值的。除了协助设计独立的功能外,客户还可根据他们的需要帮助区分任务的优先级。

2) 持续的过程

XP 倡导持续集成子系统，从而可及早检测到子系统之间的不匹配。还应当必要时重构代码，追求构建和部署小型的增量版本。

不断整合

所有程序员都熟悉可怕的代码整合。当发现对对象存储的看法与其实编写方式不匹配时，就需要执行这种任务。让子系统整合在一起时，问题就会暴露出来。XP 认识到这个问题，鼓励在开发时频繁地将代码整合进项目。

XP 提出了具体的整合方法。两个程序员(开发代码的一对搭档)坐在指定的“整合站”中，合并代码。在通过 100% 的测试之前不把代码签入。通过这种单站的方式，可以避免冲突，整合过程被清晰地定义为签入前的一个步骤。

在个人层次上仍可使用类似的方法。工程师在将代码签入代码库之前，独立或结对运行测试。指定的计算机不停地运行自动化测试。当自动化测试失败时，团队会收到一封指出问题的电子邮件，电子邮件还列出了最近的签入。

必要时重构

大部分程序员不时地重构代码。重构是对现有能工作的代码重新设计的过程，目的是引入新的知识或采用代码编写以来发现的其他用法。重构很难进入传统的软件工程进度表，因为重构的结果不如实现新功能那么有形。然而，优秀的管理者应认识到重构对代码的长期可维护性非常重要。

重构的极端方式是在开发过程中，意识到有必要重构时就开始重构。XP 程序员已经学会识别准备好重构的代码的迹象，而不是在一个版本开始时判断产品中哪些已有的部分需要设计。虽然这种做法几乎肯定会导致不可预期和不定期的任务，但在适当时重构代码应该便于未来的开发。

构建小型发布

XP 的理论之一是：当软件项目试图一次完成太多时，会变得有风险、难以处理。XP 主张的是在接近 2 个月(而不是 18 个月)的时间窗口内发布较小的版本，而不是发布涉及核心变化和数页发布说明的大版本。通过这种短小的发布周期，只有最重要的功能才能进入产品。这迫使工程人员和市场人员在哪些功能才真正重要上达成一致。

3) 寻求共识

软件由团队开发。任何代码的拥有者都不是个人，而是整个团队。XP 提供了多条指导原则，使共享代码和想法成为可能。

共同的编码标准

由于集体所有制的指导原则和结对编程的实践，如果每个工程师都有自己的命名约定和缩进约定，那么在极限环境中编程会很困难。XP 并没有提倡任何特定的风格，但是给出了一条指导原则，如果看一段代码就能立即识别出这段代码的作者，那么小组可能需要更好地定义编码标准。

有关编码风格不同方法的其他信息，请参阅第 3 章“编码风格”。

共享代码

在很多传统的开发环境中，代码的所有权是严格定义的，通常是有保障的。笔者的一位朋友曾经工作过的公司里，经理明令禁止签入对团队其他成员编写的代码的修改。XP 采取完全相反的方法，声明代码由大家集体拥有。

集体所有权是有实际效果的，这有一些原因。从管理角度看，当某个工程师突然离职时，情况不是那么不利，因为还有其他人能理解那部分代码。从工程师的角度看，集体所有权构建了系统工作方式的统一视角。这有助于设计任务，并允许单个程序员随意修改，为整个项目带来价值。

关于集体所有权的一个要点是，没必要让每个程序员熟悉每行代码。项目是借助团队力量完成的，

这更是一种心态，没有理由让任何人储藏知识。

简化设计

XP 专家的一句口头禅是“避免任意的通用性。”这违背了很多程序员的自然倾向。如果要完成的任务是设计基于文件的对象存储，可以采取以下路线：创建一个能解决所有基于文件的存储问题的完整解决方案。该设计可能很快演变为涵盖多语言和任何类型的对象。XP 认为，该设计应该倾向于通用性谱系的另一端。不要设计能获大奖、得到大家庆祝的理想对象存储，而是设计尽可能简单、能用的对象存储。必须清楚当前的需求，按照这些规范编写代码，以避免出现过于复杂的代码。

想要习惯设计的简单性可能很难。根据工作的类型，代码可能需要存在多年，由其他根本想不到的代码部分使用。根据第 6 章的讨论，构建可能在未来使用的功能的问题在于，不知道那些假想的用例是什么，无法创造出完全正确的优秀设计。相反，XP 认为，应该建立对当下有用的东西，把修改的机会留到以后。

有共同的隐喻

XP 使用术语隐喻(metaphor)，指明团队的所有成员(包括客户和经理)都应该对系统有共同的高层次看法。这不一定是对象如何通信或者 API 怎么编写的规范。相反，隐喻是系统组件的心理模型和命名模型。应当给每个组件指定一个描述性名称，这样，团队的每个成员只需要根据名称即可推断其功能。当讨论项目时，团队成员应使用隐喻来推进共享的词汇表。

4) 编程人员福利

很明显，开发人员的福利十分重要。因此，XP 的最后一条原则是关于“正常工作的小时数”的。

正常工作的小时数

XP 有一些工作小时数的说法。XP 的观点是，休息好的程序员才是开心的，才能保证生产效率。XP 提倡每周工作约 40 小时，连续两个星期加班会发出警告。

当然，不同的人需要不同的休息时间。不过，主要想法是，如果坐下来写代码时没有清醒的头脑，代码就很糟糕，会违反很多 XP 原则。

2. XP 的实践

XP 纯粹主义者声称，极限编程的 12 条原则是交织在一起的，因此如果遵循某些原则，而不遵循另一些原则，会极大地破坏这种方法论。例如，结对编程对测试是至关重要的，因为如果自己不知道如何测试某段代码，搭档可以提供帮助。此外，如果你劳累了一天，决定跳过测试，你的搭档将唤起罪恶感。

然而一些 XP 原则被证明是难以实现的。对于一些工程师来说，在编写代码之前就编写测试过于抽象。对于那些工程师，如果没有代码要测试，就不必真正编写测试代码，只需要设计测试就足够了。很多 XP 原则是严格定义的，但如果理解了背后的理论，那么应该可以设法让这些原则适用于项目的需要。

XP 的协作方面可能也有难度。结对编程有可以衡量的好处，但经理可能很难认为每天实际上只有一半的人在写代码是合理的。团队的一些成员甚至可能觉得这种密切合作不舒服，可能会觉得在有人看着的情况下很难输入代码。如果团队处于不同的物理位置，或成员倾向于定期远程办公，那么结对编程也有明显的困难。

对于一些组织，极限编程可能过于激进。XP 这类方法的速度很慢，对工程师有着正式政策的大公司可能会接受。然而，即使公司对 XP 的实现有阻力，只要理解了背后的理论，就能提升自己的工作效率。

28.3.5 软件分流

在 Edward Yourdon 的 *Death March* (Prentice Hall, 1997) 这本名字带有宿命论意味的书中, 描述了软件中频繁发生的恐怖情况: 落后于进度、人员紧缺、超过预算以及糟糕的设计。Yourdon 的理论是, 当软件项目进入这种状态时, 即使是最好的现代软件开发方法也不再适用。如本章所述, 许多软件开发方法围绕着正规化的文档来构建, 或采用以用户为中心的方式设计。在已经处于“死亡行军”模式的项目中, 根本没有时间采用这些方法。

软件分流(software triage)背后的思想是: 当项目已经处于糟糕的状态时, 资源是紧缺的, 时间是稀缺的, 工程师是稀缺的, 钱也可能是稀缺的。当项目已经远远落后于进度时, 经理和开发人员要克服的主要心理障碍是: 不可能在规定的时间内满足原来的需求了。任务就变为将剩下的功能组织为“必须有”“应该有”“可以有”的列表。

软件分流是一个艰巨而细致的过程。它往往需要一名经验丰富的老将作为“死亡行军”项目的领导, 做出艰难的决定。对于工程师, 最重要的一点是在一定条件下, 可能有必要抛弃一些熟悉的过程(遗憾的是还包括一些已有的代码), 以按时完成项目。

28.4 构建自己的过程和方法

任何书籍或工程理论都不可能完全符合项目或组织的需求。建议尽可能多地学习各种方法, 设计自己的过程。结合不同方法的概念可能比想象中容易。例如, RUP 选择性地支持一种类似 XP 的方法。下面是一些构建自己的软件工程方法的技巧。

28.4.1 对新思想采取开放态度

一些工程技术初看上去很疯狂, 或者不可能有用。看看软件工程方法中的创新, 作为改进既有过程的一种方式。在可能时尝试新事物。如果 XP 听起来很吸引人, 但不确定 XP 能否在组织中工作, 可慢慢引入 XP, 看是否可行, 每次采取几条原则, 在实验性的小项目中试用。

28.4.2 提出新想法

工程团队很可能由背景不同的人员组成。他们可能是初创企业的老总、资深顾问、应届毕业生或博士。每个人对软件项目应该如何运作都有不同的经验和想法。有时, 最好的过程往往是在各种不同的环境中不同实现方式的组合。

28.4.3 知道什么行得通、什么行不通

当项目结束时(更好的是在项目进行过程中, 例如 Scrum 方法中的 Sprint Retrospective 会议), 把团队聚在一起评估这个过程。有时, 直到整个团队停下来开始思考之后, 才有人注意到某个重大问题。也许有的问题大家都知道, 但是没人讨论。

考虑什么方法行不通, 怎样才能修复这些问题。有些组织要求签入任何源代码之前进行正式的代码审查。如果代码审查非常漫长、乏味, 没有人能很好地完成, 那么应该以小组形式讨论代码审查技术。

还要考虑哪些方面进展良好, 怎样扩展这些部分。例如, 如果以小组可编辑的网站来维护功能任务的方式行得通, 那么可以花一些时间把这个网站做得更好。

28.4.4 不要逃避

不论过程是经理强制要求的，还是团队自己定制而成的，它的存在都是有原因的。如果过程要求编写正式的设计文档，那么一定要编写这些文档。如果这个过程不够紧凑或过于复杂，看看能不能和经理讨论。不要逃避过程——它会回来继续困扰你的。

28.5 源代码控制

对所有源代码的管理对于任何公司，不论是大型公司还是小公司，甚至个人开发者来说，都是非常重要的。例如在公司，将所有代码保存在每个开发人员的计算机上，而不是由源代码控制软件来管理，是非常不切合实际的。这将导致维护噩梦，因为不是每个人都永远有最新的代码。所有的源代码应该通过源代码控制软件管理。有 3 种源代码控制软件解决方案。

- **本地解决方案**：这类解决方案将所有源代码文件及其历史保存在本地计算机上，这种方式不适合团队使用。这是 20 世纪 70 年代和 80 年代的解决方案，不应该再使用。这里不做进一步讨论。
- **客户机/服务器解决方案**：这类解决方案被分解为客户端组件和服务组件。对于个人开发者，客户端和服务组件可在同一台计算机上运行，但是这种分离很容易将服务组件转移到专用的物理服务器计算机上。
- **分布式解决方案**：这类解决方案比客户机/服务器解决方案更进一步。它没有存储所有内容的集中位置。每个开发人员都有所有文件(包含所有历史)的副本，使用对等方案来替代客户机/服务器方式。代码通过交换补丁在对等机之间同步。

客户机/服务器解决方案由两部分组成。第一部分是服务器软件，这是运行在中央服务器上的软件，负责跟踪所有的源代码文件及其历史。第二部分是客户端软件。客户端软件应该安装在每个开发人员的计算机上，负责与服务器软件通信，以获取最新版本的源文件、得到以前版本的源文件、将本地更改提交回服务器以及回滚更改到以前的版本等。

分布式解决方案不使用中心服务器。客户端软件使用对等协议，通过交换补丁与其他对等机同步。相同的操作，例如提交修改、回滚修改等，执行得很快，因为不涉及对中心服务器的网络访问。缺点是需要客户端的更多空间，因为需要存储所有文件，包括全部历史。

大部分源代码控制系统都有一些特殊术语，但遗憾的是，不是所有系统都使用同样的术语。下面列出一些常用术语。

- **分支(branch)**：源代码可以有多个分支，也就是说，不同版本的代码可以同步开发。例如，每个发布的版本都可以创建一个分支。在这些分支中，这些发布的版本可以实现 bug 修复，新功能可添加到主分支中。为发布的版本创建的 bug 修复也可以合并到主分支中。
- **签出(checkout)**：这是在开发人员的计算机上，根据中央服务器或对等机上指定版本的源代码创建本地副本的过程。
- **签入(checkin)、提交(commit)或合并(merge)**：开发人员要对本地的源代码副本进行修改。如果本机上一切工作正常，开发人员可将这些本地修改签入/提交/合并回中央服务器，或与对等机交换补丁。
- **冲突(conflict)**：当多个开发人员修改同一个源代码文件时，在提交这份源代码文件的过程中就可能发生冲突。源代码控制软件往往会尝试自动解决这些冲突。如果无法解决，客户端软件会要求用户手动解决任何冲突。

- **标签(label 或 tag)**: 任何时刻都可向所有文件或特定提交操作添加标签。这便于跳回源代码在那个时刻的版本。
- **存储库(repository)**: 由源代码控制软件管理的文件集合, 也包括历史, 称为存储库。这也包括这些文件的元数据, 例如提交的注释。
- **解决(resolve)冲突**: 发生签入冲突时, 用户只有首先解决冲突, 才能继续签入代码。
- **修订(revision)或版本(version)**: 版本指的是文件内容在特定时间点的快照。版本表示特定的点, 代码可回到这个点, 也可和这个点进行比较。
- **更新(update)或同步(sync)**: 更新或同步的意思是将开发人员计算机上的本地副本和中央服务器或对等机上的版本进行同步。请注意, 这可能需要合并, 从而可能导致需要解决的冲突。
- **工作副本(working copy)**: 工作副本是单个开发人员计算机上的本地副本。

有几个可用的源代码控制软件解决方案。其中一些是免费的, 还有一些是商业性的。表 28-1 列出了一些可用的解决方案。

表 28-1 可用的解决方案

	免费/开源	商业
仅本地解决方案	SCCS、RCS	PVCS
客户机/服务器解决方案	CVS、Subversion	IBM Rational ClearCase、Azure DevOps Server、Perforce
分布式解决方案	Git、Mercurial、Bazaar	TeamWare、BitKeeper、Plastic SCM

注意:

以上列表并不完整, 只是一组精选的解决方案, 供你了解哪些解决方案是可用的。

本书没有具体建议使用哪种解决方案。现在大多数软件公司都有源代码控制解决方案, 每个开发人员都应该采用。如果尚未采用, 那么公司绝对应该投入一些时间研究可用的解决方案, 选择适合自己的解决方案。至少, 没有使用源代码控制系统, 维护就会成为一场噩梦。即使是个人项目, 推荐研究可行的解决方案。如果找到一个喜欢的解决方案, 那么开发会简单得多。源代码控制系统会自动跟踪不同版本和更改的历史记录。如果做出的改进不能按照需要的方式工作, 就很容易回到旧版本。

28.6 本章小结

本章介绍了软件开发过程的几个模型和方法论。肯定还有很多其他构建软件的方式, 既有正式的也有非正式的。除了适用于团队的方法, 正确的软件开发方法肯定不止一种。找到正确方法的最佳方式就是自己研究一番, 学习不同的方法, 和同事交流经验, 并迭代改进自己的过程。记住, 检验过程方法的唯一重要标准就是看这种方法对团队编写代码有多大帮助。

本章最后一部分简要介绍源代码控制的概念。这应该是任何软件公司(不论大公司还是小公司)不可分割的一部分, 源代码控制甚至对个人在家做的项目也有帮助。有几个可用的源代码控制软件, 因此建议尝试几个, 看哪一个能满足自己的需要。

28.7 练习

通过完成下面的习题, 可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题, 可以在看网站上的答案之前, 先重新阅读本章的部分内容, 尝试着自己

找到答案。

练习 28-1 给出一些软件生命周期模型和软件工程方法的例子。

练习 28-2 XP 的持续集成听起来有点极端,近年来,一些公司在持续部署上有所进步。对持续部署做一些研究,然后找出它的含义。

练习 28-3 除了习题 28-2 的持续部署之外,还有持续交付。对持续交付做一些研究,将其与持续部署进行对比。

练习 28-4 研究快速应用开发(RAD)。它与本章内容有什么关系?

第29章

编写高效的 C++ 程序

本章内容

- “效率”和“性能”的意义
- 可以使用的语言层次的优化
- 设计高效程序时可遵循的设计原则
- 使用分析工具

不论应用程序属于哪个领域，程序的效率都很重要。如果产品在市场上和其他产品竞争，速度可以是一个重大的区别：在较慢和较快的程序中选择，你会选择哪一个？没有人会买一个需要两周时间才能启动的操作系统，除非这是唯一的选择。即使不打算出售产品，这些产品也会有用户。如果产品需要让用户浪费时间等待完成任务，用户会很不满意。

理解了专业 C++ 设计和编码的概念，掌握了 C++ 语言提供的一些更复杂工具后，现在应该在程序中考虑性能问题。编写高效的程序不仅要在设计层次深思熟虑，还涉及实现层次的细节。尽管本章处在本书的后半部分，但要记住，性能是程序生命周期一开始就要思考的问题。

29.1 性能和效率概述

在进一步研究细节之前，最好先定义本书使用的性能和效率这两个术语。程序的性能可能指几个方面，例如速度、内存使用、磁盘访问和网络使用。本章重点介绍速度性能。效率这个词用于程序时，指的是程序运行时不要做无用功。高效的程序能在给定条件下尽快完成其任务。如果应用程序领域本质上禁止快速执行，那么这个程序可有效率但是不快。

注意：

高效(或高性能)的程序会尽快执行特定任务。

注意，本章名字“编写高效的 C++ 程序”指的是所编写程序的运行效率，而不是所编写程序的效率。也就是说，通过学习本章，要节省用户的时间，而不是开发人员自己的时间！

29.1.1 提升效率的两种方式

语言层次的效率涉及尽量高效地使用语言，例如将按值传递对象改为按引用传递。这种做法只能达到这一步。更重要的是设计层次的效率，包括选择高效的算法、避免不必要的步骤和计算、选择恰当的设计优化。优化已有的代码往往涉及用更好、更高效的算法或数据结构替代糟糕的算法或数据结构。

29.1.2 两种程序

如前所述，效率对于所有应用程序领域来说都很重要。此外，还有一小部分程序要求极高水平的效率，例如系统级软件、嵌入式系统、计算密集型应用以及实时游戏，而大多数程序都不要求这种效率。除非编写的是高性能的应用程序，否则没必要将 C++ 代码的速度做到极致。这就是构建普通家用车和跑车之间的差别。每辆汽车都必须合理有效，但跑车对性能的要求极高。

29.1.3 C++是不是低效的语言

C 程序员经常抵制 C++ 在高性能应用程序中的使用。他们声称 C++ 语言本质上比 C 语言或类似的过程式语言低效，因为 C++ 包含高层次的概念，如异常和虚函数。然而，这种说法是有问题的。

首先，不能忽略编译器的作用。在讨论语言的效率时，必须将语言的性能和编译器优化这种语言的效果分离。计算机执行的并不是 C 或 C++ 代码。编译器首先将代码转换成机器语言，并在这个过程中进行优化。这意味着，不能简单地运行 C 和 C++ 程序的基准测试并比较结果。这实际上比较的是编译器优化语言的效果，而不是语言本身。C++ 编译器可优化掉语言中很多高层次的结构，生成类似于 C 语言生成的机器码。目前，研发投入更多集中于 C++ 编译器而非 C 编译器，因此与 C 代码相比，C++ 代码实际会得到更好的优化，运行速度可能更快。

然而，批评者仍然认为一些 C++ 特性不能被优化掉。例如，根据第 10 章“发现派生技术”的解释，虚函数需要一个 `vtable`，在运行时需要添加一个间接层次，因而比普通的非虚函数调用慢。然而，如果仔细思考，会发现这种说法仍然难以令人信服。虚函数调用不只是函数调用，还要在运行时选择调用哪个函数。对应的非虚函数调用可能需要一个条件语句来选择调用的函数。如果不需要这些额外的语义，可以使用一个非虚函数。C++ 语言的一般设计原则是：“如果不使用某项功能，则不需要付出代价。”如果不使用虚函数，那么不会因为能够使用虚函数而损失性能。因此在 C++ 中，非虚函数调用在性能上等同于 C 语言中的函数调用。然而，由于虚函数调用的开销如此之小，因此建议对于所有非 `final` 类，将所有的类方法，包括析构函数(但不包括构造函数)设计为虚方法。

更重要的是，通过 C++ 高层次的结构可编写更干净的程序，这些程序的设计层次更高效，更易于读取，更便于维护，能避免积累不必要的代码和死代码。

我们相信，如果选择 C++ 语言而不是过程式的语言(如 C 语言)，在开发、性能和维护上会有更好的结果。

还有其他更高级的面向对象语言，如 C# 和 Java，二者都在虚拟机上运行。C++ 代码由 CPU 直接执行，不存在运行代码的虚拟机。C++ 离硬件更近，这意味着大多数情况下，它的速度快于 C# 和 Java 等语言。

29.2 语言层次的效率

许多书籍、文章和程序员花费了大量时间，试图说服你对代码进行语言层次的优化。这些提示和

技巧很重要，在某些情况下可加快程序的运行速度。然而，这些优化远不如整体设计和程序选择的算法重要。可以通过引用传递需要的所有数据，但如果写磁盘的次数比实际需要的次数多一倍，那么按引用传递不会让程序更快。这很容易陷入引用和指针的优化而忘记大局。

此外，一些语言层次的技巧可通过好的优化编译器自动进行。不应花费时间自己优化某个特定领域，除非分析器指明某个领域是瓶颈，如本章后面所述。

也就是说，使用某些语言级别的优化(如按引用传递)是良好的编码风格。

本书试图展示一种平衡策略。因此，这里只包含我们认为最有用的语言层次优化。这个列表是不完整的，但若优化代码，该列表应提供一个很好的起点。然而，请务必阅读和实践本章后面描述的设计层次的效率建议。

警告：

谨慎使用语言级优化。建议先建立清晰、结构良好的设计和实现方案，再使用分析器，仅优化分析器标记为性能瓶颈的部分。

29.2.1 高效地操纵对象

C++在幕后做了很多工作，特别是和对象相关的工作。总是应该注意编写的代码对性能的影响。如果遵循一些简单的指导原则，代码将变得更有效率。注意这些原则仅与对象相关，与基本类型(例如 bool、int、float 等)无关。

1. 值传递还是引用传递

第 1 章和第 9 章讨论了何时使用值传递和引用传递的规则，这里有必要重申一次。

警告：

对于函数本身会拷贝的参数，最好使用值传递参数，但只有当参数的类型支持移动语义时这样。其他情况可使用 const 的引用参数。

对于按值传递的参数，必须记住一些事情。如果函数形参的类型是基类，而将派生类的对象作为实参按值传递，则会将派生对象切片，以符合基类类型。这导致信息丢失，详见第 10 章。按值传递还可能导致拷贝的成本，而引用传递可以避免这种成本。

然而，在某些情况下，值传递实际上是向函数传递参数的最佳方式。考虑如下表示“人”的 Person 类：

```
class Person
{
public:
    Person() = default;
    Person(string firstName, string lastName, int age)
        : m_firstName { move(firstName) }, m_lastName { move(lastName) },
          m_age { age } { }
    virtual ~Person() = default;
    const string& getFirstName() const { return m_firstName; }
    const string& getLastName() const { return m_lastName; }
    int getAge() const { return m_age; }

private:
    string m_firstName, m_lastName;
    int m_age { 0 };
};
```

正如规则所建议的, `Person` 的构造函数按值接受 `firstName` 和 `lastName`, 然后将它们分别移动到 `m_firstName` 和 `m_lastName`, 因为无论如何都会对它们进行拷贝。见第 9 章对此的解释。

现在, 看看下面的函数按值接收 `Person` 对象的函数:

```
void processPerson(Person p) { /* Process the person. */ }
```

可以像这样调用这个函数:

```
Person me { "Marc", "Gregoire", 42 };
processPerson(me);
```

像下面这样编写这个函数, 看上去并未增加多少代码:

```
void processPerson(const Person& p) { /* Process the person. */ }
```

对函数的调用保持不变。然而, 考虑一下在第一个版本的函数中按值传递会发生什么。为初始化 `processPerson()` 的 `p` 参数, `me` 必须通过调用其拷贝构造函数进行复制。即使没有为 `Person` 类编写拷贝构造函数, 编译器也会生成一个来复制每个数据成员。这看上去也没有那么糟: 只有 3 个数据成员。然而, 其中的两个成员是字符串, 都是带有拷贝构造函数的对象。因此, 也会调用它们的拷贝构造函数。通过引用接收 `p` 的 `processPerson()` 版本没有这样的复制成本。因此, 在这个例子中通过按引用传递, 可以避免代码进入这个函数时进行的 3 次构造函数调用。

这个示例至此尚未完成。在第一个版本的 `processPerson()` 中, `p` 是 `processPerson()` 函数的一个局部变量, 因此在该函数退出时必须销毁。销毁时需要调用 `Person` 类的析构函数, 而析构函数会调用所有数据成员的析构函数。`string` 类有析构函数, 因此退出这个函数时(如果按值传递)会调用 3 次析构函数。如果通过引用传递 `Person` 对象, 则不需要执行任何这种调用。

注意:

如果函数必须修改对象, 可通过非 `const` 引用传递对象。如果函数不应该修改对象, 可通过 `const` 引用传递, 如前面的例子所示。有关引用和 `const` 的详细信息, 请参阅第 1 章。

注意:

应避免通过指针传递, 因为相对按引用传递, 按指针传递相对过时, 相当于倒退到 C 语言了, 很少适合于 C++(除非在设计中传递 `nullptr` 有特殊意义)。

2. 按值返回还是按引用返回

可以通过从函数中按引用方式返回对象, 以避免对象发生不必要的复制。但有时不可能通过引用返回对象, 例如编写重载的 `operator+` 和其他类似运算符时。永远都不要返回指向局部对象的引用或指针, 局部对象会在函数退出时被销毁。

但是, 按值返回对象通常很好。这是由于(命名的)返回值优化和移动语义, 这两种方法都是按值优化返回对象, 它们会在后面的章节讨论。

3. 通过引用捕获异常

如第 14 章“错误处理”所述, 应该通过引用捕获异常, 以避免分片和额外的复制。抛出异常的性能开销很大, 因此任何提升效率的小事情都是有帮助的。

4. 使用移动语义

应该确保你的类支持移动语义, 要么编译器生成的移动构造函数和移动赋值运算符, 要么自己实

现它们。根据“零规则”(见第 9 章),设计类时,使编译器生成复制和移动构造函数以及复制和移动赋值运算符便足够了。如果编译器不能隐式定义这些类,那么在允许的情况下,可显式将它们设置为 default。如果这行不通,应当自行实现。对象有了移动语义后,许多操作都会更加高效,特别是与标准库容器和算法相结合时。

5. 避免创建临时对象

有些情况下,编译器会创建临时的无名对象。第 9 章介绍过,为一个类编写全局 operator+ 之后,可对这个类的对象和其他类型的对象进行加法运算,只要其他类型的对象可转换为这个类的对象即可。例如,第 9 章的 SpreadsheetCell 类定义,包含对算术运算符的支持,部分如下:

```
export class SpreadsheetCell
{
    public:
        // Other constructors omitted for brevity.
        SpreadsheetCell(double initialValue);
        // Remainder omitted for brevity.
};

export SpreadsheetCell operator+(const SpreadsheetCell& lhs,
    const SpreadsheetCell& rhs);
```

这个接收 double 值的构造函数允许编写下面这样的代码:

```
SpreadsheetCell myCell { 4 }, aThirdCell;
aThirdCell = myCell + 5.6;
aThirdCell = myCell + 4;
```

第二行通过参数 5.6 创建了一个临时 SpreadsheetCell 对象,然后将 myCell 和临时对象作为参数调用 operator+。把结果保存在 aThirdCell 中。第三行做了同样的事情,只不过 4 必须强制转换为 double 类型,才能调用 SpreadsheetCell 的 double 版构造函数。

这个例子中的重点是:编译器生成了代码,为两个加操作创建了一个额外的无名 SpreadsheetCell 对象。该对象必须调用其构造函数和析构函数进行构造和销毁。如果还感到怀疑,可在构造函数和析构函数中插入 cout 语句,观察输出。

一般情况下,每当代码需要在较大表达式中将一种类型的变量转换为另一种类型时,编译器都会构造临时对象。此规则主要适用于函数调用。例如,假设函数的原型如下:

```
void doSomething(const SpreadsheetCell& s);
```

可这样调用:

```
doSomething(5.56);
```

编译器会使用 double 版构造函数从 5.56 构造一个临时的 SpreadsheetCell 对象,然后把这个对象传入 doSomething()。注意,如果把 const 从 s 参数移除,那么再也不能通过常量调用 doSomething(),而是必须传入变量。

一般来说,应该避免迫使编译器构造临时对象的情况。尽管有时这是不可避免的,但是至少应该意识到这项“特性”的存在,这样才不会为实际性能和分析结果而感到惊讶。

编译器还会使用移动语义使临时对象的效率更高。这是要在类中添加移动语义的另一个原因。详见第 9 章。

6. 返回值优化

通过值返回对象的函数可能导致创建一个临时对象。继续 `Person` 示例，考虑下面的函数：

```
Person createPerson()
{
    Person newP { "Marc", "Gregoire", 42 };
    return newP;
}
```

假如像这样调用这个函数(假设 `Person` 类已经实现了 `operator<<`运算符)：

```
cout << createPerson();
```

即便这个调用没有将 `createPerson()` 的结果保存在任何地方，也必须将结果保存在某个地方，才能传递给 `operator<<`。为生成这种行为的代码，编译器允许创建一个临时变量，以保存 `createPerson()` 返回的 `Person` 对象。

即使这个函数的结果没有在任何地方使用，编译器也仍然可能会生成创建临时对象的代码。例如，考虑如下代码：

```
createPerson();
```

编译器可能生成代码，以创建一个临时对象来保存返回值，即使这个返回值没有使用也是如此。

不过，通常不必担心这个问题，因为编译器会在大多数情况下优化掉临时变量，以避免复制和移动。对于 `createPerson()` 示例，这种优化称为 NRVO(Named Return Value Optimization，命名的返回值优化)，原因是返回语句返回命名的变量。如果返回语句的参数是未命名的临时值，该优化称为 RVO(Return Value Optimization，返回值优化)。通常不会为发布版本启用此类优化。NRVO 和 RVO 都省略拷贝，使编译器避免函数返回的对象进行拷贝和移动。这会导致零拷贝的值传递语义。要使 NRVO 生效，返回语句的参数必须是一个本地变量。例如，在下面的代码中，编译器不能执行 NRVO：

```
Person createPerson()
{
    Person person1;
    Person person2;
    return getRandBool() ? person1 : person2;
}
```

根据 `someCondition()` 的调用结果，将返回 `person1` 或 `person2`。因为这不是返回对象的形式，所以 NRVO 不适用，更糟糕的是，这个表达式是一个左值，因此它不会被当作一个右值表达式。这意味着 `person1` 或 `person2` 被拷贝。你可能想要修改这个问题，重写如下：

```
return someCondition() ? move(person1) : move(person2);
```

然而，通过这样做，编译器仍然不能应用(N)RVO，实际上强制它使用移动语义。如果对象不支持移动语义，编译器就会退回到拷贝操作。

避免拷贝和移动的推荐方法不是使用 `std::move()`，而是使用允许 NRVO 的下面代码：

```
if (someCondition()) {
    return person1;
} else {
    return person2;
}
```

如果 NRVO 和 RVO 不可用，将发生复制或移动。如果从函数返回的对象支持移动语义，就将其

移出函数，而不是复制。

29.2.2 预分配内存

使用 C++ 标准库容器(第 18 章“标准库容器”中讨论)的一个重要好处是：它们自动处理内存管理。给容器添加元素时，容器会自动扩展。但有时，这会带来性能问题。例如，`std::vector` 容器在内存中连续存储元素。如果需要扩展，则需要分配新的内存块，然后将所有元素移动(或复制)到新的内存中。例如，如果在循环中使用 `push_back()` 给 `vector` 添加数百万个元素，将严重地影响性能。

如果预先知道要在 `vector` 中添加的元素数量，或大致能够评估出来，就可以在开始添加元素前预分配足够的内存。`vector` 具有容量(`capacity`)和大小(`size`)，容量指不需要重新分配的情况下可添加的元素数量，大小指容器中的实际元素数量。可以预分配内存，使用 `reserve()` 更改容量，使用 `resize()` 重新设置 `vector` 的大小。详见第 18 章。

29.2.3 使用内联方法和函数

根据第 9 章的描述，内联(`inline`)方法或函数的代码可以直接插到被调用的地方，从而避免函数调用的开销。一方面，应将所有符合这种优化条件的函数和方法标记为 `inline`。但不要过度使用该功能，因为它实际上背离了基本设计原则：基本设计原则是将接口与实现分离，这样一来，不需要更改接口即可完善实现。仅考虑为常用的基本类使用该功能。另外记住，程序员的内联请求只是给编译器提供的建议，编译器有权拒绝这些建议。

另一方面，编译器会在优化过程中内联一些适当的函数和方法，即使这些函数没有用 `inline` 关键字标记，甚至即使这些函数在源文件(而非头文件)中实现也是如此。因此，应该阅读编译器文档，以免浪费大量精力判断哪些函数应该内联。例如，在 Visual C++ 中，这个特性成为链接时代码生成(LTCG)，并支持跨模块内联。GCC 编译器将其称为链接时间优化(LTO)。

29.3 设计层次的效率

程序中的设计决策对性能的影响比语言细节(例如按引用传递)对性能的影响大多了。例如，如果为应用程序中的基础任务选择了运行时间为 $O(n^2)$ 的算法，而不是运行时间为 $O(n)$ 的更简单算法，那么可能执行的操作数是实际需要的操作数的平方。举一个具体例子，某任务使用 $O(n^2)$ 算法执行 100 万次操作，而使用 $O(n)$ 算法只执行 1000 次操作。即使这个操作已经在语言层次做了优化，这个程序也需要执行 100 万次操作，而更好的算法只需要执行 1000 次操作，所以这个程序非常低效。应总是仔细选择算法。有关算法设计决策和大 O 表示法，请参阅本书第 II 部分，特别是第 4 章“设计专业的 C++ 程序”。

除算法选择外，设计层次的效率还包括一些具体的窍门。应尽量使用已有的数据结构和算法，例如 C++ 标准库、Boost 库或其他库，而不要自己编写它们，因为它们是由专家编写的。这些库一直在使用，当前也在大量使用，因此可以预计，大多数 bug 都已被发现和纠正。还应在设计中融入多线程，以便充分利用计算机的所有处理能力，详情参见第 27 章“C++ 多线程编程”。本节剩余部分讨论另外两个优化程序的设计技术：缓存和对象池。

29.3.1 尽可能多地缓存

缓存(cache)是指将数据项保存下来供以后使用，从而避免再次获取或重新计算它们。你可能熟悉计算机硬件领域中缓存的使用原理。现代计算机处理器内建了内存缓存，它在访问速度高于主内存的

位置保存了最近访问和频繁访问的内存值。大部分被访问的内存位置在很短时间间隔内会访问多次，因此硬件层次的缓存可极大提升计算速度。

软件中的缓存遵循同样的方法。如果任务或计算特别慢，应该确保不会执行不必要的重复计算。第一次执行任务时将结果保存在内存中，使这些结果可用于未来的需求。下面是通常执行缓慢的任务清单。

- **磁盘访问**：在程序中应避免多次打开和读取同一个文件。如果内存可用，并且需要频繁访问这个文件，那么应将文件内容保存在内存中。
- **网络通信**：如果需要经由网络通信，那么程序会受网络负载的影响而行为不定。将网络访问当成文件访问处理，尽可能多地缓存静态信息。
- **数学计算**：如果需要在多个地方使用非常复杂的计算结果，那么执行这种计算一次并共享结果。但是，如果计算不是非常复杂，那么计算可能比从缓存中提取更快。如果需要确定这种情形，可使用分析器。
- **对象分配**：如果程序需要大量创建和使用短期对象，可以考虑使用本章后面讨论的对象池。
- **线程创建**：这个任务也很慢。可将线程“缓存”在线程池中，类似于在对象池中缓存对象。

常见的缓存问题是：保存的数据往往是底层信息的副本。在缓存的生命周期中，原始数据可能发生变化。例如，可能需要缓存配置文件中的值，这样就不必反复读取配置文件。但是，可能允许用户在程序运行时更改配置文件，这会使缓存版本的信息过期。这种情况下，需要“缓存失效”机制：当底层数据发生变化时，必须停止使用缓存的信息，或重新填写缓存。

缓存失效的技术之一是要求管理底层数据的实体通知“程序数据发生了变化”。可通过程序在管理器中注册回调的方式实现这一点。另外，程序还可轮询某些会触发自动重新填充缓存的事件。不论使用哪种具体的缓存失效技术，一定要在程序中使用缓存之前考虑好这些问题。

注意：

始终要记住，维护缓存需要编码、内存和处理时间。在这之上，缓存可能是难以查找的 bug 来源。在分析器清晰地说明该领域是性能瓶颈时，应仅添加该领域的缓存。首先要编写干净、正确的代码，再分析它们，仅优化其中的一部分。

29.3.2 使用对象池

存在不同类型的对象池。本节讨论一种对象池，它一次分配一大块内存，此时，对象池就地创建多个较小对象。可将这些对象分发给客户，在客户完成时重用它们，这样就不必另外调用内存管理器为各个对象分配内存或解除内存分配。

以下对象池实现的亮点(将在基准测试中演示)是针对具有大数据成员的对象。对象池是不是特定用例的正确解决方案只能通过分析代码来判断。

1. 对象池的实现

下面提供对象池的一个类模板的实现，可在程序中使用这个类模板。该实现保留了一个由类型 T 的对象块组成的 `vector`。此外，它还追踪着包含指向所有空闲对象指针的 `vector` 的对象。池通过 `acquireObject()` 方法分发对象。如果调用了 `acquireObject()`，但是已经没有空闲对象，那么池将分配另一块对象。`acquireObject()` 方法返回一个 `shared_ptr`。

注意，此实现使用的是 `vector` 标准库容器，没有任何同步操作。因此，这个版本不是线程安全的。关于多线程编程的讨论请参见第 27 章。

下面是类的定义，通过注释讲解了细节部分。类模板对将要存储在池中的类型以及用于分配和释

放内存块的分配器类型进行参数化。

```
// Provides an object pool that can be used with any class that provides a
// default constructor.
//
// acquireObject() returns an object from the list of free objects. If
// there are no more free objects, acquireObject() creates a new chunk
// of objects.
// The pool only grows: objects are never removed from the pool, until
// the pool is destroyed.
// acquireObject() returns an std::shared_ptr with a custom deleter that
// automatically puts the object back into the object pool when the
// shared_ptr is destroyed and its reference count reaches 0.
export
template <typename T, typename Allocator = std::allocator<T>>
class ObjectPool
{
public:
    ObjectPool() = default;
    explicit ObjectPool(const Allocator& allocator);
    virtual ~ObjectPool();

    // Allow move construction and move assignment.
    ObjectPool(ObjectPool&& src) noexcept = default;
    ObjectPool& operator=(ObjectPool&& rhs) noexcept = default;

    // Prevent copy construction and copy assignment.
    ObjectPool(const ObjectPool& src) = delete;
    ObjectPool& operator=(const ObjectPool& rhs) = delete;

    // Reserves and returns an object from the pool. Arguments can be
    // provided which are perfectly forwarded to a constructor of T.
    template<typename... Args>
    std::shared_ptr<T> acquireObject(Args... args);

private:
    // Contains chunks of memory in which instances of T will be created.
    // For each chunk, the pointer to its first object is stored.
    std::vector<T*> m_pool;
    // Contains pointers to all free instances of T that
    // are available in the pool.
    std::vector<T*> m_freeObjects;
    // The number of T instances that should fit in the first allocated chunk.
    static const size_t ms_initialChunkSize { 5 };
    // The number of T instances that should fit in a newly allocated chunk.
    // This value is doubled after each newly created chunk.
    size_t m_newChunkSize { ms_initialChunkSize };
    // Creates a new block of uninitialized memory, big enough to hold
    // m_newChunkSize instances of T.
    void addChunk();
    // The allocator to use for allocating and deallocating chunks.
    Allocator m_allocator;
};
```

使用这个对象池时，必须确保对象池自身的寿命超出对象池给出的所有对象的寿命。构造函数很简单，只是将给定的分配器存储在一个数据成员中：

```

template <typename T, typename Allocator>
ObjectPool<T, Allocator>::ObjectPool(const Allocator& allocator)
    : m_allocator { allocator }
{
}

```

分配新块的 `addChunk()` 方法实现如下。`addChunk()` 的第一部分进行新块的实际分配工作。一个“块”只是一个未初始化的内存块，使用分配器分配，并且足够大来保存 `T` 的 `m_newChunkSize` 实例。添加对象块，实际上没有构造对象，也就是说，对象的构造函数没有被调用。当实例被分配时，`acquireObject()` 就会完成。`addChunk()` 的第二部分创建指向 `T` 的新实例的指针。它使用定义在 `<numeric>` 中的 `iota()` 算法。要刷新内存，`iota()` 用值填充由它的前两个参数给定的范围。这些值以第 3 个参数的值开始，并对后续的值加 1。因为使用的是 `T*` 指针，所以对 `T*` 指针加 1 就会跳到内存块中的下一个 `T`。最后，`m_newChunkSize` 的值增加了一倍，因此下一个添加的块大小是当前添加块的两倍。这样做是出于性能原因，并遵循 `std::vector` 的原则。下面是它的实现：

```

template <typename T, typename Allocator>
void ObjectPool<T, Allocator>::addChunk()
{
    std::cout << "Allocating new chunk..." << std::endl;

    // Allocate a new chunk of uninitialized memory big enough to hold
    // m_newChunkSize instances of T, and add the chunk to the pool.
    auto* firstNewObject { m_allocator.allocate(m_newChunkSize) };
    m_pool.push_back(firstNewObject);

    // Create pointers to each individual object in the new chunk
    // and store them in the list of free objects.
    auto oldFreeObjectsSize { m_freeObjects.size() };
    m_freeObjects.resize(oldFreeObjectsSize + m_newChunkSize);
    std::iota(begin(m_freeObjects) + oldFreeObjectsSize, end(m_freeObjects),
              firstNewObject);

    // Double the chunk size for next time.
    m_newChunkSize *= 2;
}

```

可变参数方法模板 `acquireObject()` 从池中返回一个空闲对象，如果没有更多可用的空闲对象，则分配一个新的块。如前所述，添加一个新块只是分配了一个未初始化的内存块。在内存中正确的位置正确地构造 `T` 的实例是 `acquireObject()` 的责任。这通过 `placement new` 运算符完成。传递给 `acquireObject()` 的任何参数都被完美地转发到 `T` 的构造函数上。最后，`T*` 指针被包装在 `shared_ptr` 中，并带有一个自定义删除器。这个删除器不会释放任何内存；相反，它使用 `std::destroy_at()` 手动调用析构函数，然后将指针放回可调用对象列表中。

```

template <typename T, typename Allocator>
template <typename... Args>
std::shared_ptr<T> ObjectPool<T, Allocator>::acquireObject(Args... args)
{
    // If there are no free objects, allocate a new chunk.
    if (m_freeObjects.empty()) { addChunk(); }

    // Get a free object.
    T* object { m_freeObjects.back() };

```

```

// Initialize, i.e. construct, an instance of T in an
// uninitialized block of memory using placement new, and
// perfectly forward any provided arguments to the constructor.
new(object) T { std::forward<Args>(args)... };

// Remove the object from the list of free objects.
m_freeObjects.pop_back();

// Wrap the initialized object and return it.
return std::shared_ptr<T> { object, [this](T* object) {
    // Destroy object.
    std::destroy_at(object);
    // Put the object back in the list of free objects.
    m_freeObjects.push_back(object);
} };
}

```

最后，池的析构函数必须使用给定的分配器来释放任何已分配的内存：

```

template <typename T, typename Allocator>
ObjectPool<T, Allocator>::~~ObjectPool()
{
    // Note: this implementation assumes that all objects handed out by this
    // pool have been returned to the pool before the pool is destroyed.
    // The following statement asserts if that is not the case.
    assert(m_freeObjects.size() ==
        ms_initialChunkSize * (std::pow(2, m_pool.size()) - 1));

    // Deallocate all allocated memory.
    size_t chunkSize { ms_initialChunkSize };
    for (auto* chunk : m_pool) {
        m_allocator.deallocate(chunk, chunkSize);
        chunkSize *= 2;
    }
    m_pool.clear();
}

```

2. 使用对象池

考虑这样一个应用程序，它使用大量具有大的数据成员的短期对象，因此分配的代价很高。假设有个 `ExpensiveObject` 的类，定义如下所示：

```

class ExpensiveObject
{
public:
    ExpensiveObject() { /* ... */ }
    virtual ~ExpensiveObject() = default;
    // Methods to populate the object with specific information.
    // Methods to retrieve the object data.
    // (not shown)

private:
    // An expensive data member.
    array<double, 4 * 1024 * 1024> m_data;
    // Other data members (not shown)
};

```

不是在程序的生命周期中创建和删除大量此类对象，而是可以使用前面开发的对象池。可以使用 `chrono` 库(参见第 22 章“日期和时间工具”)对池进行基准测试，如下所示：

```
using MyPool = ObjectPool<ExpensiveObject>;

shared_ptr<ExpensiveObject> getExpensiveObject(MyPool& pool)
{
    // Obtain an ExpensiveObject object from the pool.
    auto object { pool.acquireObject() };
    // Populate the object. (not shown)
    return object;
}

void processExpensiveObject(ExpensiveObject* object) { /* ... */ }

int main()
{
    const size_t NumberOfIterations { 500'000 };

    cout << "Starting loop using pool..." << endl;
    MyPool requestPool;
    auto start1 { chrono::steady_clock::now() };
    for (size_t i { 0 }; i < NumberOfIterations; ++i) {
        auto object { getExpensiveObject(requestPool) };
        processExpensiveObject(object.get());
    }
    auto end1 { chrono::steady_clock::now() };
    auto diff1 { end1 - start1 };
    cout << format("{}ms\n", chrono::duration<double, milli>(diff1).count());

    cout << "Starting loop using new/delete..." << endl;
    auto start2 { chrono::steady_clock::now() };
    for (size_t i { 0 }; i < NumberOfIterations; ++i) {
        auto object { new ExpensiveObject{} };
        processExpensiveObject(object);
        delete object; object = nullptr;
    }
    auto end2 { chrono::steady_clock::now() };
    auto diff2 { end2 - start2 };
    cout << format("{}ms\n", chrono::duration<double, milli>(diff2).count());
}
```

`main()` 函数包含一个池性能的小型基准测试。它在循环中请求 500 000 个对象，并乘以所需的时间。执行两次循环，一次使用我们的池，一次使用标准的 `new/delete` 运算符。在测试机器上的代码发布版本的结果如下：

```
Starting loop using pool...
Allocating new chunk...
54.526ms
Starting loop using new/delete...
9463.2393ms
```

在本例中，使用对象池大约要快 170 倍。但是请记住，这个对象池是为处理具有大数据成员的对象而定制。示例中使用的 `ExpensiveObject` 类就是这种情况，它包含一个 4MB 数组作为其数据成员之一。

29.4 剖析

最好在设计和编码时考虑效率问题，如果根据常识或基于经验的直观感觉，可避免编写明显低效的程序，就不应编写它们。但在设计和编码阶段也不要过于关注效率。最好一开始就建立清晰、结构良好的设计和实现方案，再使用分析器，仅优化分析器标记为“性能瓶颈”的部分。第 4 章介绍了“90/10”法则：大部分程序中 90% 的运行时间都在执行 10% 的代码。这意味着可能优化了 90% 的代码，但程序运行时间只改进了 10%。显然，需要优化典型负载下程序中运行最多的部分。

因此，需要剖析程序，判断哪些部分的代码需要优化。有很多可用的剖析工具，可在程序运行时进行分析，并生成性能数据。大部分剖析工具都提供函数级别的分析功能，可分析程序中每个函数的运行时间(或占总执行时间的百分比)。在程序上运行剖析工具后，通常可立即判断出程序中的哪些部分需要优化。优化前后的剖析也有助于证明优化是否有效。

如果使用的是 Microsoft Visual C++ 2019，就有了一个强大的内建剖析器，详见本章后面的讨论。如果未使用 Visual C++，Microsoft 提供了 Community 版本(visualstudio.microsoft.com)，可供学生、开源开发人员和开发人员免费使用，以创建免费和付费的应用程序。对于人数不超 5 人的小公司，它也是免费的。另一个很好的剖析工具是来自 IBM 的 Rational PurifyPlus(www.almttoolbox.com/purify.php)。还有一些免费的较小剖析工具：Very Sleepy(www.codersnotes.com/sleepy)和 Luke Stackwalker(lukestackwalker.sourceforge.net)是 Windows 上流行的剖析器，Valgrind(valgrind.org)和 gprof(GNU profiler, source.org/binutils/doc/gprof)是 UNIX/Linux 系统上著名的剖析器，此外还有许多其他选择。本节演示两个剖析器：Linux 的 gprof，和 Visual C++ 2019 附带的剖析器。

29.4.1 使用 gprof 的剖析示例

最好通过一个真实的编码示例来展示剖析的强大功能。声明一下，初次尝试中的性能 bug 并不微妙。真正的效率问题可能更复杂，但一个足够长的能演示这些问题的程序对于本书来说过于冗长了。

假设你在美国社会安全局工作。美国社会安全局每年都会发布一个网站，让用户查询前一年新生儿名字的流行度。你的工作是编写一个后台程序，供用户查找名字。输入是一个包含所有新生儿名字的文件。显然，这个文件包含重复的名字。例如，在 2003 年男孩的名字文件中，Jacob 是最流行的，出现了 29 195 次。这个后台程序必须读取文件，构建一个内存数据库。用户可能请求使用给定名字的婴儿的绝对数量，或请求这个名字在所有婴儿中的使用排名。

1. 最初的设计尝试

这个后台程序的逻辑设计包含 NameDB 类，这个类包含以下公有方法：

```
export class NameDB
{
    public:
        // Reads list of baby names in nameFile to populate the database.
        // Throws invalid_argument if nameFile cannot be opened or read.
        NameDB(std::string_view nameFile);

        // Returns the rank of the name (1st, 2nd, etc).
        // Returns -1 if the name is not found.
        int getNameRank(std::string_view name) const;

        // Returns the number of babies with a given name.
        // Returns -1 if the name is not found.
```



```

        int getAbsoluteNumber(std::string_view name) const;

        // Private members not shown yet ...
};

```

困难的部分是选择合适的数据结构用于内存数据库。第一次尝试是使用包含名字/计数对的 `vector`。`vector` 中的每个条目保存一个名字，以及这个名字出现在原始数据文件中的次数。下面是使用这种设计的类的完整定义：

```

export class NameDB
{
public:
    NameDB(std::string_view nameFile);
    int getNameRank(std::string_view name) const;
    int getAbsoluteNumber(std::string_view name) const;
private:
    std::vector<std::pair<std::string, int>> m_names;

    // Helper methods
    bool nameExists(std::string_view name) const;
    void incrementNameCount(std::string_view name);
    void addNewName(std::string_view name);
};

```

注意这里使用了第 18 章讨论的标准库 `vector` 和 `pair` 类。`pair` 是一个实用工具类，它将两种不同类型的值组合在一起。

下面是构造函数和辅助方法 `nameExists()`、`incrementNameCount()` 和 `addNewName()` 的实现。`nameExists()` 和 `incrementNameCount()` 中的循环遍历 `vector` 中的所有元素。

```

// Reads the names from the file and populates the database.
// The database is a vector of name/count pairs, storing the
// number of times each name shows up in the raw data.
NameDB::NameDB(string_view nameFile)
{
    // Open the file and check for errors.
    ifstream inputFile { nameFile.data() };
    if (!inputFile) {
        throw invalid_argument { "Unable to open file" };
    }

    // Read the names one at a time.
    string name;
    while (inputFile >> name) {
        // Look up the name in the database so far.
        if (nameExists(name)) {
            // If the name exists in the database, just increment the count.
            incrementNameCount(name);
        } else {
            // If the name doesn't yet exist, add it with a count of 1.
            addNewName(name);
        }
    }
}

// Returns true if the name exists in the database, false otherwise.

```

```

bool NameDB::nameExists(string_view name) const
{
    // Iterate through the vector of names looking for the name.
    for (auto& entry : m_names) {
        if (entry.first == name) {
            return true;
        }
    }
    return false;
}

// Precondition: name exists in the vector of names.
// Postcondition: the count associated with name is incremented.
void NameDB::incrementNameCount(string_view name)
{
    for (auto& entry : m_names) {
        if (entry.first == name) {
            ++entry.second;
            return;
        }
    }
}

// Adds a new name to the database.
void NameDB::addNewName(string_view name)
{
    m_names.push_back(make_pair(name.data(), 1));
}

```

请注意, 可使用诸如 `find_if()` 的算法来完成 `nameExists()` 和 `incrementNameCount()` 中的循环要完成的工作。这里显式地展示了循环以强调性能问题。

精明的读者可能已经注意到一些性能问题。如果有成千上万个名字怎么办? 填充数据库时的大量线性搜索可能会拖慢速度。

为完成这个例子, 下面给出两个公有方法的实现:

```

// Returns the rank of the name.
// First looks up the name to obtain the number of babies with that name.
// Then iterates through all the names, counting all the names with a higher
// count than the specified name. Returns that count as the rank.
int NameDB::getNameRank(string_view name) const
{
    // Make use of the getAbsoluteNumber() method.
    int num { getAbsoluteNumber(name) };

    // Check if we found the name.
    if (num == -1) {
        return -1;
    }

    // Now count all the names in the vector that have a
    // count higher than this one. If no name has a higher count,
    // this name is rank number 1. Every name with a higher count
    // decreases the rank of this name by 1.
    int rank { 1 };
    for (auto& entry : m_names) {

```

```

        if (entry.second > num) {
            ++rank;
        }
    }
    return rank;
}

// Returns the count associated with the given name.
int NameDB::getAbsoluteNumber(string_view name) const
{
    for (auto& entry : m_names) {
        if (entry.first == name) {
            return entry.second;
        }
    }
    return -1;
}

```

2. 对初次设计尝试的剖析

为测试程序，需要 main() 函数：

```

import name_db;
import <iostream>;
using namespace std;

int main()
{
    NameDB boys { "boys_long.txt" };
    cout << boys.getNameRank("Daniel") << endl;
    cout << boys.getNameRank("Jacob") << endl;
    cout << boys.getNameRank("William") << endl;
}

```

main() 函数创建一个名为 boys 的 NameDB 数据库，要求这个数据库通过文件 boys_long.txt 填充自身，这个文件包含 500 500 个名字。

gprof 的使用有 3 个步骤：

(1) 在编译了 name_db 模块之后，应该用一个特殊标志编译程序，使它在运行程序时记录原始的执行信息。使用 GCC 作为编译器时，这个标志是 -pg，例如：

```
> gcc -lstdc++ -std=c++2a -pg -fmodules-ts -o namedb NameDB.cpp NameDBTest.cpp
```

注意：

目前，GCC 还没有对 C++20 module 的标准支持，只有通过 modules-ts。一旦 GCC 完全支持 module，请检查它的文档来学习如何编译和使用 module。

另外，现在，必须指定 -std=c++2a 来启用 C++20 特性。将来会变成 -std=c++20，而 -fmodules-ts 可能会变成 -fmodules。检查文档。

(2) 接下来运行程序。这次运行应该在工作目录下生成 gmon.out 文件。运行程序时要有耐心，因为第一个版本的程序非常慢。

(3) 最后一步是运行 gprof 命令来分析 gmon.out 剖析信息，并生成一份(大致)可读的报告。gprof 输出至标准输出，因此需要将输出重定向到一个文件：

```
> gprof namedb gmon.out > gprof_analysis.out
```

现在,可分析数据。遗憾的是,输出文件有些晦涩难懂。需要花一些时间来学习如何解释它。gprof 提供了两组独立信息。第一组信息总结了执行程序中的每个函数所花费的时间。第二组信息更实用,总结了每个函数及其后代执行所用的时间,这组信息也称为调用图。下面是一些来自 gprof_analysis.out 文件的输出,输出已经过编辑,以方便阅读。请注意在不同的计算机上,数字会有所不同。

index	%time	self	children	called	name
[1]	100.0	0.00	14.06		main [1]
		0.00	14.00	1/1	NameDB::NameDB [2]
		0.00	0.04	3/3	NameDB::getNameRank [25]
		0.00	0.01	1/1	NameDB::~~NameDB [28]

以下列表解释了上述各列。

- **index:** 通过这个索引可在调用图中检索这一条目。
- **%time:** 这个函数及其后代执行时间占程序总执行时间的百分比。
- **self:** 函数本身执行的秒数。
- **children:** 这个函数后代执行的秒数。
- **called:** 这个函数调用的频率。
- **name:** 函数的名称。如果函数名后跟一个放在方括号中的数字,那么这个数字表示调用图中的另一个索引。

上面的输出片段说明,main()及其后代的执行时间占程序总执行时间的 100%,共 14.06 秒。第二行显示,NameDB 构造函数消耗了这 14.06 秒中的 14.00 秒。因此可立即清楚地看出性能问题的出现位置。继续追查构造函数的哪一部分消耗了这么长时间,需要跳到调用图中索引为 2 的位置,因为这是最后一列中名字后面方括号中的索引。在某测试系统上,索引为 2 的调用如下所示:

[2]	99.6	0.00	14.00	1	NameDB::NameDB [2]
		1.20	6.14	500500/500500	NameDB::nameExists [3]
		1.24	5.24	499500/499500	NameDB::incrementNameCount [4]
		0.00	0.18	1000/1000	NameDB::addNewName [19]
		0.00	0.00	1/1	vector::vector [69]

NameDB::NameDB 下面的嵌套项表明了哪些后代消耗了最多的时间。在这里可看到,nameExists()花了 6.14 秒,incrementNameCount()花了 5.24 秒。请记住,这些时间是这些函数调用的时间总和。这些行的第 4 列显示了函数调用次数(nameExists()是 500 500 次,incrementNameCount()是 499 500 次)。没有其他函数会消耗这么长时间。

如果不进一步分析,你可能立即想到两件事:

- (1) 花 14 秒时间向数据库填入约 500 000 个名字是很慢的。也许需要一种更好的数据结构。
- (2) nameExists()和 incrementNameCount()几乎消耗相同的时间,调用次数也几乎相同。如果思考一下应用程序的领域,这应该是合理的:文本文件输入中的大部分名字都是重复的,因此大部分 nameExists()调用的后面都跟了一次 incrementNameCount()调用。如果再看一下代码,会发现这两个函数的代码几乎一致,应该可合并。此外,它们主要完成的事情是搜索 vector。最好使用一种排好序的数据结构以缩短搜索时间。

3. 第二次尝试

根据 gprof 输出观察到的情况,下面对这个程序重新设计。新的设计使用 map 替代 vector。第 18 章提到,标准库 map 保持项的顺序,提供 $O(\log n)$ 查找时间而不是 vector 的 $O(n)$ 查找时间。还可使

用 `std::unordered_map`, 它提供了 $O(1)$ 查找时间, 再使用剖析器确定对于这个应用程序, 它是否比 `std::map` 快。这留给读者作为练习。

这个新版本的程序还将 `nameExists()` 和 `incrementNameCount()` 合并为 `nameExistsAndIncrement()`。

下面是新的类定义:

```
export class NameDB
{
    public:
        NameDB(std::string_view nameFile);
        int getNameRank(std::string_view name) const;
        int getAbsoluteNumber(std::string_view name) const;
    private:
        std::map<std::string, int> m_names;
        bool nameExistsAndIncrement(std::string_view name);
        void addNewName(std::string_view name);
};
```

下面是新的方法实现:

```
// Reads the names from the file and populates the database.
// The database is a map associating names with their frequency.
NameDB::NameDB(string_view nameFile)
{
    // Open the file and check for errors.
    ifstream inputFile { nameFile.data() };
    if (!inputFile) {
        throw invalid_argument { "Unable to open file" };
    }

    // Read the names one at a time.
    string name;
    while (inputFile >> name) {
        // Look up the name in the database so far.
        if (!nameExistsAndIncrement(name)) {
            // If the name exists in the database, the
            // method incremented it, so we just continue.
            // We get here if it didn't exist, in which case
            // we add it with a count of 1.
            addNewName(name);
        }
    }
}

// Returns true if the name exists in the database, false
// otherwise. If it finds it, it increments it.
bool NameDB::nameExistsAndIncrement(string_view name)
{
    // Find the name in the map.
    auto res { m_names.find(name.data()) };
    if (res != end(m_names)) {
        ++res->second;
        return true;
    }
    return false;
}
```

```
// Adds a new name to the database.
void NameDB::addNewName(string_view name)
{
    m_names[name.data()] = 1;
}

int NameDB::getNameRank(string_view name) const
{
    // Implementation omitted, same as before.
}

// Returns the count associated with the given name.
int NameDB::getAbsoluteNumber(string_view name) const
{
    auto res { m_names.find(name.data()) };
    if (res != end(m_names)) {
        return res->second;
    }
    return -1;
}
```

4. 对第二次设计尝试的剖析

按照前面所示的相同步骤，可获取这个新版程序的 gprof 性能数据。这些数据相当令人鼓舞：

index	%time	self	children	called	name
[1]	100.0	0.00	0.21		main [1]
		0.02	0.18	1/1	NameDB::NameDB [2]
		0.00	0.01	1/1	NameDB::~NameDB [13]
		0.00	0.00	3/3	NameDB::getNameRank [28]
[2]	95.2	0.02	0.18	1	NameDB::NameDB [2]
		0.02	0.16	500500/500500	NameDB::nameExistsAndIncrement
[3]		0.00	0.00	1000/1000	NameDB::addNewName [24]
		0.00	0.00	1/1	map::map [87]

不同计算机上的输出会有所不同。甚至可能在输出中看不到 NameDB 方法的数据。由于第二次尝试的效率很高，导致计时太短，因此在输出中 map 方法的数据可能比 NameDB 方法的数据还要多。

在笔者的测试系统上，main()只需要 0.21 秒，速度提升了 67 倍。这个程序肯定还可做进一步改进。例如，目前构造函数执行一次查询，判断名字是否已经在 map 中，如果不在，则添加至 map。可使用以下单行代码，将这两个操作结合在一起：

```
++m_names[name];
```

如果名字已经在 map 中，该语句会递增 counter。如果还不es 在 map 中，该语句首先在 map 中添加一项，将给定名称作为该项的键，将值初始化为 0；此后递增值，得到的 counter 为 1。

为实现这个改进，可删除 nameExistsAndIncrement()和 addNewName()方法，按以下方式修改构造函数：

```
NameDB::NameDB(string_view nameFile)
{
    // Open the file and check for errors.
    ifstream inputFile { nameFile.data() };
    if (!inputFile) {
        throw invalid_argument { "Unable to open file" };
    }
}
```

```

// Read the names one at a time.
string name;
while (inputFile >> name) {
    ++m_names[name];
}
}

```

getName()方法仍使用循环，遍历 map 中的所有元素。另一项改进是使用另一个数据结构，以避免 getName()中的线性迭代，这留给读者作为练习。

29.4.2 使用 Visual C++ 2019 的剖析示例

本节简单讨论 Visual C++ 2019 大多数版本自带的强大剖析器。VC++剖析器有一个完整的图形用户界面。我们没有特别推荐某种剖析器，但最好比较和体验一下 gprof 这种命令行剖析器和 VC++这种基于 GUI 的剖析器分别能提供哪些功能。

要开始在 Visual C++ 2019 中剖析应用程序，首先需要在 Visual Studio 中打开项目。这个例子采用前面那种低效设计尝试的 NameDB 代码。这里不再重复这段代码。在 Visual Studio 中打开这个项目后，单击 Analyze 菜单，然后选择 Performance Profiler，打开一个新的窗口。图 29-1 显示了这个窗口的截图。

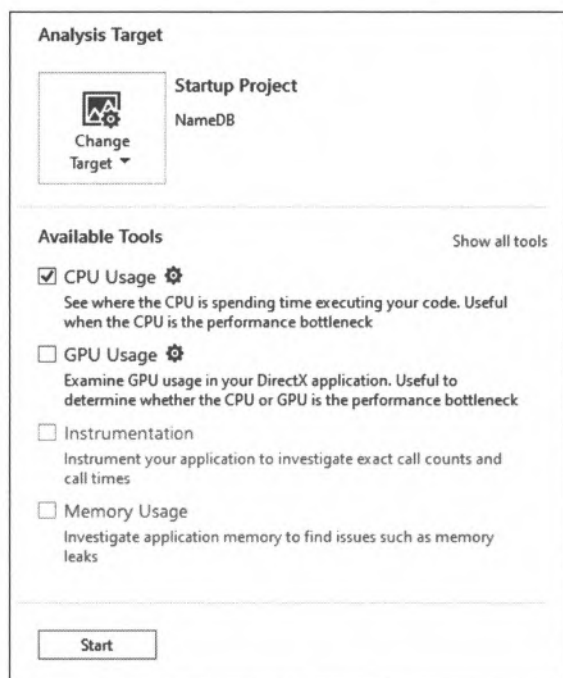


图 29-1 Analysis Target 窗口

根据 VC++版本的不同，有几种不同的剖析方法。下面解释其中的两种。

- **CPU 使用情况：**用较低的开销监测应用程序。这意味着，对应用程序的剖析不会对目标应用程序产生太大的性能影响。
- **Instrumentation：**为能准确计算函数调用的次数并为每个函数调用计时，这种工具会向应用程序添加额外的代码。然而，这种方法对应用程序具有更大的性能影响。推荐使用 CPU 使

用率工具，以对应应用程序的瓶颈有个初步认识。如果这种工具给不出足够的信息，则尝试 Instrumentation 工具。

对于这个剖析示例，只启用 CPU Usage 工具并单击 Start 按钮。会开始执行程序并剖析它的 CPU 使用情况。当程序执行完成后，Visual Studio 会自动打开剖析报告。图 29-2 显示了在第一次尝试剖析 NameDB 应用程序时这个报告的样子。

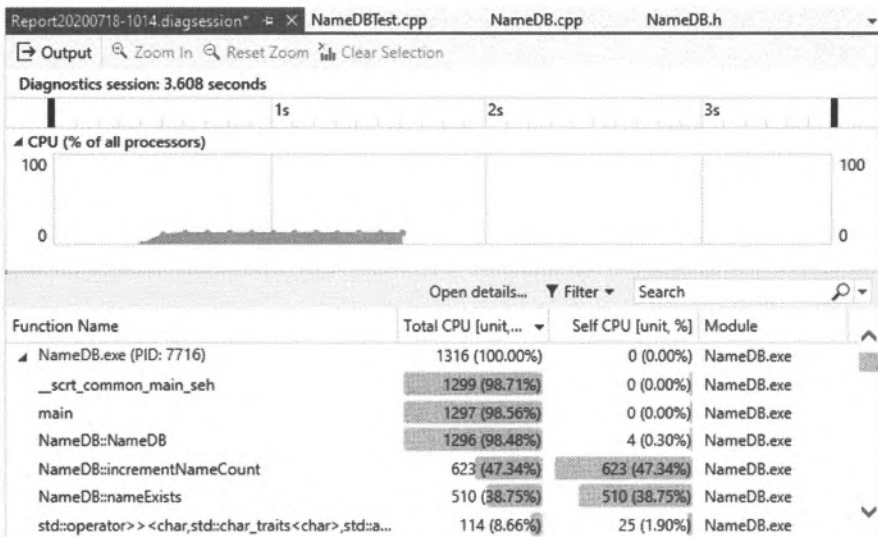


图 29-2 剖析报告

从这份报告中，马上可看到热点路径。就像 gprof 一样，展示了 NameDB 构造函数占据了程序的大量运行时间，incrementNameCount() 和 nameExists() 都占用几乎相同的时间。Visual Studio 的性能剖析报告是可交互的。例如，可单击 NameDB 构造函数，深入查看这个函数。这会得到这个函数的深入报告，如图 29-3 所示。

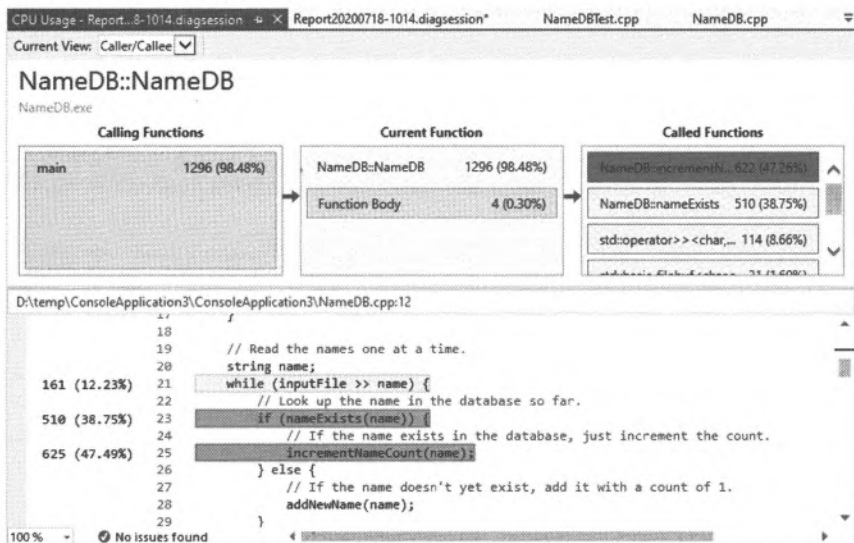


图 29-3 这个函数的深入报告

这份深入报告在顶端展示了图形分解，在底部展示了方法的实际代码。代码视图展示了那一行代码运行时间的百分比。运行时间最长的那行代码用阴影显示(见图 29-3)。

报告的顶部有一个下拉菜单，名为当前视图。从该下拉列表中选择 Call Tree 将显示代码中热路径的另一个视图。在 Call Tree 视图中，单击 Expand Hot Path 按钮，在代码中展开热路径，如图 29-4 所示。

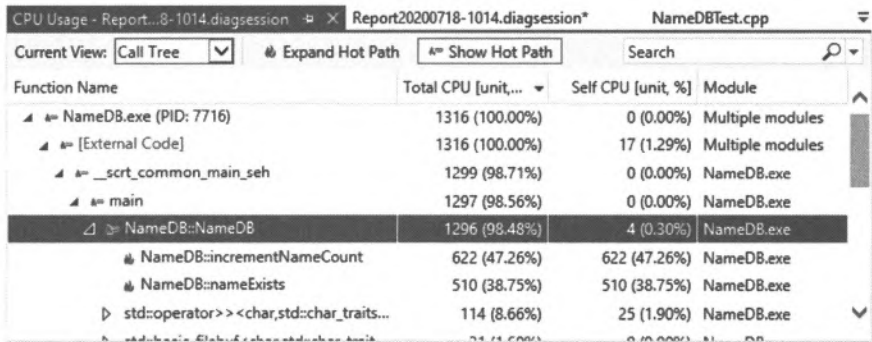


图 29-4 在代码中展开热路径

此外，在这个视图中可立即看到 main()正在调用耗时最长的 NameDB 构造函数。

29.5 本章小结

本章讨论了事关 C++程序效率和性能的关键因素，并提供了一些设计和编写更高效应用程序的技巧和技术。我们希望你能认识到性能的重要性和剖析器的强大功能。

你需要记住两件事：第一是在设计和编码时，不要过于关注性能。建议先构建正确、结构良好的设计和实现方案，再使用剖析器，仅优化剖析器标记为性能瓶颈的部分。

第二是设计层次的效率比语言层次的效率重要得多，例如不要使用复杂度很糟糕的算法或数据结构，而应使用更好的算法或数据结构。

29.6 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

练习 29-1 在下面的代码片段中你可以发现哪些效率问题？

```
class Bar { };

class Foo
{
public:
    explicit Foo(Bar b) {}
};

Foo getFoo(bool condition, Bar b1, Bar b2)
{
    return condition ? Foo { b1 } : Foo { b2 };
```

```
}

int main()
{
    Bar b1, b2;
    auto foo { getFoo(true, b1, b2) };
}
```

练习 29-2 本章中最需要记住的两件事是什么？

练习 29-3 修改 Profiling 部分的最终 NameDB 解决方案，使用 `std::unordered_map` 代替 `map`。分析更改前后的代码，并比较结果。

练习 29-4 从习题 29-3 的剖析结果来看，现在 NameDB 构造函数中的 `operator>>` 似乎是瓶颈。你能修改实现来避免使用 `operator>>` 吗？由于输入文件中的每一行都包含一个名字，也许逐行读取名字会更快？尝试这样修改你的实现，并比较更改前后的剖析结果。

第30章

熟练掌握测试技术

本章内容

- 软件质量控制的含义以及如何跟踪 bug
- 单元测试的含义
- 使用 Visual C++ 测试框架练习单元测试
- 模糊测试或模糊的含义
- 集成测试、系统测试和回归测试的含义

当程序员意识到测试是软件开发过程的一部分时，就意味着他的职业生涯已经跨过一个重要障碍。bug 并非偶然发生的，每个较大规模的项目中都存在 bug。一支良好的质量控制团队 (Quality-Assurance, QA) 是十分有价值的，但不能将测试的所有重担都压在 QA 上。程序员既负责编写可运行的代码，也负责测试代码的正确性。

测试分为白盒测试和黑盒测试两种。白盒测试中，测试者了解程序的内部原理；黑盒测试中，在测试程序功能时，不需要了解任何实现细节。在专业级项目中，这两类测试都十分重要。黑盒测试是最基本的方法，它通常建立用户行为的模型。例如，黑盒测试可分析诸如按钮的界面组件。如果测试者单击按钮，却未看到任何变化，则程序中明显存在 bug。

黑盒测试不能包罗一切。现代程序都很庞大，我们无法完全做到：模拟单击所有的按钮，提供每类输入，执行命令的所有组合。白盒测试是必需的，如果知道测试代码是在对象或子系统级别编写的，则更能方便地确保测试涵盖代码中的所有路径。这有助于确保测试范围。与黑盒测试相比，白盒测试更容易编写和自动完成。本章重点介绍白盒测试技巧，因为程序员可在开发期间使用这些技术。

本章首先简要讨论质量控制，包括查看和跟踪 bug 的一些方法。此后介绍单元测试，单元测试是最简单、最有用的测试类型。接着讲述单元测试的理论和实践，列举几个单元测试示例。之后介绍高级测试，包括集成测试、系统测试和回归测试。最后列出确保测试成功的一些提示。

30.1 质量控制

对于大型项目而言，即使到了功能完备的地步，也不能说已经完成。在主要开发阶段以及随后阶段，始终都有需要查找和修复的 bug。只有理解了质量控制以及 bug 的生命周期后，才能达到良好效果。

30.1.1 谁负责测试

软件开发组织具有多种不同的测试方法。在小公司中，可能并没有全职测试产品的团队，测试可能由单个开发人员负责；公司也可能要求所有员工伸出援手，在产品发布前测试产品的可靠性。在大公司中，由全职 QA 人员根据一系列标准对一个版本进行测试，确认是否合格。无论如何，测试的一些方面仍由开发人员负责。即使有些组织不要求开发人员参与正式测试，开发人员也仍需要知道在更大的 QA 过程中自己所扮演的角色。

30.1.2 bug 的生命周期

所有高素质的工程团队都认识到，在软件发布前后都存在 bug。可通过多种不同方式处理这些问题。图 30-1 以流程图的形式显示了正式的 bug 处理过程。在这个具体过程中，bug 总由 QA 团队的成员提出。bug 报告软件将通知发送给开发经理，开发经理设置 bug 的优先级，并将 bug 分派给相应的模块所有者。模块所有者可接收这个 bug，或解释相应的 bug 实际上属于另一个模块或认为这个 bug 是无效的，让开发经理将 bug 分派给其他人。

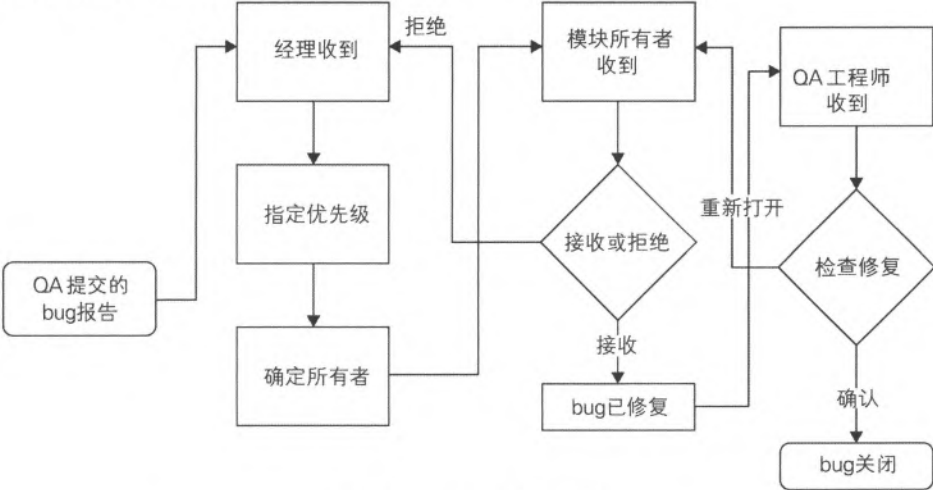


图 30-1 正式的 bug 处理过程

一旦找到 bug 的正确所有者，就进行修正，开发人员将 bug 标记为“已修复”。此时，QA 工程师如果确认 bug 不再存在，就将 bug 标记为“关闭”；如果 bug 依然存在，则再次打开相应的 bug。

图 30-2 显示了一种较不正规的处理方法。在这个工作流中，任何人都可以提交 bug，并指定初始优先级和模块。模块所有者接收 bug 报告，此后，可根据情况接收，或将其重新指定给另一个工程师或模块。在修复后，将 bug 标记为“已修复”。在测试阶段的末尾，所有开发人员和 QA 工程师会划分已修复的 bug，

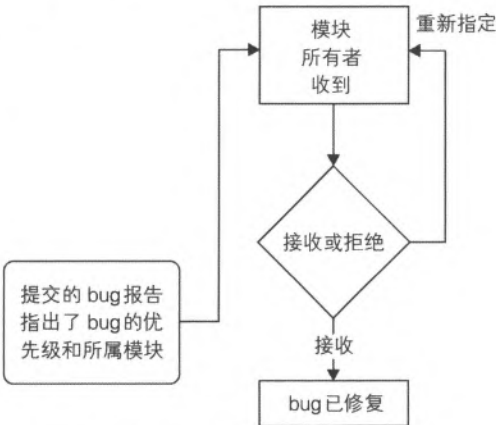


图 30-2 比较不正规的处理方法

确认每个 bug 不再存在于当前版本中。将所有 bug 标记为“关闭”时，就表明版本准备好了。

30.1.3 bug 跟踪工具

可通过多种方式跟踪软件 bug，从非正式的电子表格或电子邮件方案乃至昂贵的第三方 bug 跟踪软件。组织的相应解决方案取决于团队规模、软件的成熟度以及需要的 bug 修复正规程度。

还有很多免费的开源 bug 跟踪解决方案。一个流行的免费工具是 Bugzilla(bugzilla.org)，Bugzilla 由 Mozilla Web 浏览器的作者编写，已经积累了大量有用功能，甚至可与昂贵的 bug 跟踪软件包一决高下。它的功能十分丰富，下面列出其中一些：

- 可定制的 bug 设置，包括优先级、相关组件和状态等。
- 通过电子邮件告知新的 bug 报告，或告知现有报告的变化。
- 跟踪 bug 与重复 bug 的解决之间的依赖性。
- 报告和搜索工具。
- 用于提交和更新 bug 的基于 Web 的界面。

图 30-3 显示了输入 Bugzilla 项目中的 bug。这里，每一章都作为 Bugzilla 组件输入。bug 提交者可指定 bug 的严重程度。可添加汇总和描述信息，以便搜索 bug，或以报告格式将其列出。

Bugzilla - Enter Bug: Professional C++ Second Edition

Home | New | Browse | Search | [Search] | [?] | Reports | My Requests | Preferences | Administration
Help | Log out

Before reporting a bug, please read the [bug writing guidelines](#), please look at the list of [most frequently reported bugs](#), and please [search](#) for the bug.

[Show Advanced Fields](#) (" = Required Field)

Product: Professional C++ Second Edition

*** Component:** 01 - A Crash Course in C++
02 - Designing Professional C++ Programs
03 - Designing with Objects
04 - Designing for Reuse
05 - Coding with Style
06 - Gaining Proficiency with Classes and Objects
07 - Mastering Classes and Object

Version: 2.0

Reporter: user@some-project.com

Component Description: Chapter 6 describes the fundamental concepts involved in using classes and objects.

Severity: normal

Hardware: PC

OS: Windows 7

We've made a guess at your operating system and platform. Please check them and make any corrections if necessary.

*** Summary:** Keyword "class" is misspelled as "glass"

Description: In the first example in Chapter 6, the keyword "class" is written as "glass". I'm pretty sure there's no "glass" keyword in C++.

Attachment: [Add an attachment](#)

[Submit Bug](#)

图 30-3 输入 Bugzilla 项目中的 bug

在专业软件开发环境中，诸如 Bugzilla 的 bug 跟踪工具是必备的组件。除了集中列出当前打开的 bug 外，bug 跟踪工具还列出了以前的 bug 以及修复的重要归档信息。例如，支持工程师可使用该工具来查看与客户报告的 bug 类似的问题。如果已经修复，支持人员可告知客户需要将软件更新到哪个版本，以及如何解决问题。

30.2 单元测试

查看 bug 的唯一方式是测试。在开发人员看来，最重要的测试类型是单元测试。单元测试是代码段，对类或子系统执行特定功能。这些是可编写的粒度最细的测试。理想情况下，代码可执行的每个低级任务都应当有对应的一个或多个单元测试。例如，假设正在编写一个执行加法和乘法计算的数学库。单元测试套件应当包含以下测试：

- 测试单个加法运算
- 测试大数字的加法运算
- 测试负数的加法运算
- 测试将 0 与一个数字相加
- 测试相加的交换性
- 测试单个乘法运算
- 测试大数字的乘法运算
- 测试负数的乘法运算
- 测试将 0 与一个数字相乘
- 测试相乘的交换性

编写良好的单元测试可从多个方面为你提供保护：

- (1) 证实功能确实能够工作。只有通过一些代码来真正使用类，才能大体知道类的行为。
- (2) 如果最新引入的更改造成破坏，单元测试可首先发出警告。这种用法称为“回归测试”，将在本章后面进行介绍。
- (3) 在开发过程中使用时，将迫使开发人员从头修复问题。如果单元测试失败时无法签入，将不得不立即解决问题。
- (4) 单元测试允许在其他代码就绪前尝试自己的代码。首次开始编程时，可以编写整个程序并首次运行该程序。采用这种方法时，专业程序规模过大，因此需要能独立地测试组件。
- (5) 最后，单元测试提供了一个使用案例，这也是比较重要的。作为附带作用，单元测试为其他编程人员提供极佳的参考代码。如果一位同事需要使用你的数学库来了解如何执行矩阵乘法，你可以指导他完成适当的测试。

30.2.1 单元测试方法

使用单元测试几乎不会出错，除非未编写或编写不当。通常而言，测试数量越多，覆盖范围越大。覆盖范围越大，就越不可能漏掉 bug，使你在老板或客户面前出丑，尴尬地说：“我从未对其进行测试。”

可通过多种方式最高效地编写单元测试。第 28 章介绍的极限编程要求在编写代码前编写单元测试。首先编写测试以帮助你强化了解组件的要求，并提供可用于确定它何时完成的标准。但是，首先编写测试是一件棘手的事情，要求程序员付出努力。对于一些程序员而言，这不符合他们的编码风格。一种稍宽松的方法是在编码前设计测试，再在后来实现测试。这样，程序员仍必须了解模块的要求，但不必编写代码以使用不存在的类。

在有些团队中，具体子系统的作者不为自己的子系统编写单元测试。想法是这样的：如果为自己的代码编写测试，就可能下意识地回避已知的问题，只涵盖代码运行良好的部分。另外，如果自己刚编写了代码，就找到 bug，有时真让人高兴不起来，因此你可能三心二意。让一个开发人员为另一个开发人员的代码编写单元测试需要付出额外努力，也需要更多协调。不过，如果能完成此类协调，这种方法能提高测试效率。

为确保单元测试真正测试代码的正确部分，另一种方式是尽量增加代码覆盖范围。可以使用诸如 `gcov(gcc.gnu.org/onlinedocs/gcc/Gcov.html)` 的代码覆盖工具来了解单元测试调用的代码的百分比。只有测试完一段代码所有可能的代码路径后，才能说这段代码经过了合理测试。

30.2.2 单元测试过程

为代码提供单元测试的过程从一开始就启动了，远在编写任何代码之前。在设计阶段考虑单元测试性会影响软件设计决策。即使在编写代码前未确定编写单元测试的方法，至少也要在设计阶段花时间考虑一下要提供哪些类型的测试。这样，可将任务分解为良好定义的块，每个块都有自己的测试验证标准。例如，如果任务是编写数据库访问类，那么可以首先编写将数据插入数据库的功能。在使用单元测试套件对其进行全面测试后，可接着编写代码来支持更新、删除和选择，一边编写代码，一边测试每个片段。

下面的步骤是设计和实现单元测试的推荐方法。与任何编程方法一样，能产生最佳结果的过程就是最好的过程。建议尝试不同的单元测试使用方法，确定哪一个最合适。

1. 定义测试的粒度

编写单元测试需要耗费时间，这是无法回避的。软件开发人员的时间通常很紧。为赶在最终期限前完工，开发人员倾向于忽略编写单元测试，以便提高速度。遗憾的是，此时他们并未考虑全局。从长远看，忽略单元测试会引火烧身。在软件开发过程中，越早检测到 bug，成本越低。如果开发人员在单元测试期间找到 bug，则可以立即修复，这样其他任何人都不会再遇到这个 bug。但是，如果 bug 是由 QA 发现的，那么修复 bug 的代价更高。为处理 bug，需要额外的开发周期，需要进行 bug 管理，必须回头找开发团队进行修复，还要重新提供给 QA 来确认修复。如果在 QA 过程中漏掉一个 bug，在客户使用软件时这个 bug 依然存在，那么修复成本就更高了。

测试粒度指的是范围。如表 30-1 所示，使用单元测试时，起初只需要用几个测试函数来测试数据库，然后逐渐添加更多测试，确保一切如期工作。

表 30-1 测试粒度

大粒度测试	中等粒度测试	细粒度测试
testConnection()	testConnectionDropped()	testConnectionThroughHTTP()
testInsert()	testInsertBadData()	testConnectionLocal()
testUpdate()	testInsertStrings()	testConnectionErrorBadHost()
testDelete()	testInsertIntegers()	testConnectionErrorServerBusy()
testSelect()	testUpdateStrings()	testInsertWideCharacters()
	testUpdateIntegers()	testInsertLargeData()
	testDeleteNonexistentRow()	testInsertMalformed()
	testSelectComplicated()	testUpdateWideCharacters()
	testSelectMalformed()	testUpdateLargeData()
		testUpdateMalformed()
		testDeleteWithoutPermissions()
		testDeleteThenUpdate()
		testSelectNested()
		testSelectWideCharacters()
		testSelectLargeData()

可以看到,第2列比第1列更具体,第3列比第2列更具体。从大粒度测试到细粒度测试的移动过程中,开始考虑错误条件、不同的输入数据集以及操作模式。

当然,选择测试粒度时所做的初始决策并非一成不变。编写的数据库类可能只是概念验证,甚至不会用到。现在,很少有几个简单测试就能满足需要,始终可在后来不断增加。或者,用例可能在后来发生变化。例如,起初编写数据库类时,并未考虑国际字符。一旦添加此类功能,就应当用具体的目标单元测试进行测试。

将单元测试视为功能的实际实现的一部分。执行修改时,不要只是修改测试以使其继续工作。编写新测试,并对已有的进行重新评估。在发现和修复 bug 时,添加新的专门测试这些修复的单元测试。

注意:

单元测试是正在测试的子系统的一部分。在增强和完善子系统期间,同时增强和完善测试。

2. 构思单个测试

随着经验的积累,将可直观地感受到应将代码的哪些部分转换为单元测试。某些方法或输入看上去应当被测试。通过尝试,以及通过查看同组其他人编写的单元测试,可获得这种直觉。很容易就能看出哪位程序员是最好的单元测试人员。他们的测试编排有序,而且时常修改。

在通过直觉创建单元测试前,可构思要编写哪些测试,为此考虑以下问题:

- (1) 编写的这段代码有什么作用?
- (2) 通常采用什么方式调用每个方法?
- (3) 调用者可能破坏方法的哪些前置条件?
- (4) 可能以哪些方式误用每个方法?
- (5) 预计将哪类数据作为输入?
- (6) 预计不使用哪类数据作为输入?
- (7) 什么是边缘情形或例外情形?

不必写入上述问题的正式答案(除非经理是本书或某些测试方法的狂热爱好者),但思索这些问题有助于催生一些单元测试想法。数据库类的测试表包含测试函数,每个测试函数都是从这些问题推导而来的。

构思出要使用的一些测试后,考虑如何将它们组织成类别,即对测试进行分解。在数据库类示例中,可将测试分为以下几个类别:

- 基本测试
- 错误测试
- 本地化测试
- 错误输入测试
- 复杂的测试

将测试分为多个类别后,将更容易识别和完善。更容易确定代码的哪些部分经过良好测试,哪些部分还需要更多的单元测试。

警告:

编写大量简单的测试是很容易的,但不要忘记更复杂的情形!

3. 创建示例数据和结果

在编写单元测试时,最常掉进的陷阱是将测试与代码行为匹配,而非使用测试来验证代码。如果

编写的单元测试用于在数据库中选择一段数据，则测试是失败的，这是代码的问题还是测试的问题？有人经常假设代码是正确的，并修改测试进行匹配。这种方法通常是错误的。

为避开这个陷阱，应当在尝试前理解测试的输入以及预期输出。但说易行难。例如，假设编写一些代码，使用具体密钥来加密任意的文本块。合理的单元测试将接收固定的文本字符串，并将其传递给加密模块。此后将分析结果，看一下加密过程是否正确。

编写这样的单元测试时，有人倾向于先用加密模块尝试行为，然后查看结果。如果看似合理，就编写测试来查看相应的值。这么做什么也证明不了，因为并未真正测试代码，所编写的测试只是确认代码会返回相同的值。编写测试通常需要做些实事，需要独立于加密模块来加密文本，这样才能获得精确结果。

警告：

运行测试前，就要确定测试的正确输出。

4. 编写测试

测试的后台代码是不同的，具体取决于测试框架类型。本章后面讨论 Microsoft Visual C++ 测试框架。不管实际实现是什么，下列指导原则有助于确保测试的有效性：

- 确保每次测试只测试一点。这样，如果测试失败，将指向特定的功能片段。
- 测试中要力求具体。测试失败的原因是抛出了异常，还是返回了错误值？
- 在测试代码中广泛使用日志记录。如果有一天测试失败，必须分析发生了什么事情。
- 避免使测试依赖于更早的测试，避免使多个测试交错在一起。测试要尽量做到原子化和独立。
- 如果测试需要使用其他子系统，考虑编写这些子系统的存根或 mock 来模拟这些子系统。存根或 mock 实现与它 mock 的子系统相同的接口。它们可以用来代替任何具体的子系统实现。例如，如果单元测试需要一个数据库，但该数据库不是单元测试所测试的子系统，那么存根或 mock 可以实现数据库接口，并模拟真实的数据库。这样，运行单元测试不需要连接到实际数据库，并且实际数据库实现中的错误不会对这个特定的单元测试产生任何影响。
- 邀请代码审查者分析单元测试。在审查代码时，如果认为某处需要添加额外测试，则告诉其他工程师。

在本章后面将看到，单元测试通常很小，是简单的代码片段。大多数情况下，编写一个单元测试只需要几分钟的时间，效率极高。

5. 运行测试

编写完测试时，应当立即运行，以免生成的结果多得难以承受。看到屏幕上充满表明单元测试通过的内容，真令人感到欣慰。对于大多数程序员而言，可通过这些信息来确认代码是有用的、是正确的。

即使采用在编写代码前编写测试的方法，也应当在编写测试后立即运行一遍。这样，可自行确认测试在开始时是否会失败。一旦代码就绪，就有了实际数据，表示可以按预期完成工作。

并不能保证编写的每个测试都能在第一次就得到预期结果。从理论上讲，如果在编写代码前编写测试，所有测试都会失败。如果其中一个测试通过，要么是代码魔幻般地出现了，要么是测试存在问题。如果代码完成了，而测试失败了，将有两个可能：代码出错了，或测试出错了。

单元测试必须自动运行。可通过多种方式做到这一点。其中一种方式是使用专用系统，在每个连续集成构建后自动运行所有单元测试，或每晚至少运行一次。在单元测试失败时，此类系统必须发送电子邮件来通知开发人员。另一种方式是设置本地开发环境，每次编译代码时执行单元测试。为此，单元测试必须足够小、足够有效。如果单元测试运行时间较长，则应当将它们独立出来，由专用的测

试系统进行测试。

30.2.3 实际中的单元测试

上面介绍了单元测试的理论知识，下面将真正编写一些测试。下例将继续使用第 29 章实现的对象池。简单回顾一下，对象池是一个类，可用于避免过多的对象创建操作。通过跟踪已经创建的对象，对象池可在需要某类对象的代码与已经分配的对象之间担当代理角色。

ObjectPool 类的接口如下所示，可参阅第 29 章以了解详情。

```
export
template <typename T, typename Allocator = std::allocator<T>>
class ObjectPool
{
public:
    ObjectPool() = default;
    explicit ObjectPool(const Allocator& allocator);
    virtual ~ObjectPool();

    // Allow move construction and move assignment.
    ObjectPool(ObjectPool&& src) noexcept = default;
    ObjectPool& operator=(ObjectPool&& rhs) noexcept = default;

    // Prevent copy construction and copy assignment.
    ObjectPool(const ObjectPool& src) = delete;
    ObjectPool& operator=(const ObjectPool& rhs) = delete;

    // Reserves and returns an object from the pool. Arguments can be
    // provided which are perfectly forwarded to a constructor of T.
    template <typename... Args>
    std::shared_ptr<T> acquireObject(Args... args);
};
```

1. Visual C++测试框架简介

Visual C++内置了一个测试框架。使用单元测试框架的好处在于允许开发人员专注于编写测试，不需要耗费精力去设置测试、构建测试逻辑以及收集结果。下面针对 Visual C++ 2019 展开讨论。

注意：

除了 Visual C++，还有很多开源的单元测试框架可供使用。Google Test(<https://github.com/google/googletest>)和 Boost Test Library(http://www.boost.org/doc/libs/1_73_1/libs/test/)便是可用于 C++的框架。它们都包括对测试开发人员有用的很多实用工具，也包括用于控制结果的自动输出的选项。

开始使用 Visual C++测试框架时，必须创建一个测试项目。下面的步骤解释了如何测试 ObjectPool 类：

- (1) 启动 Visual C++，创建一个新的项目，选择 Native Unit Test Project，单击 Next 按钮。
- (2) 为项目指定名称后单击 Create 按钮。
- (3) 向导将创建一个新的测试项目，其中包含名为<ProjectName>.cpp 的文件。在 Solution Explorer 中选择该文件并删除它，因为你要添加自己的文件。如果 Solution Explorer 停靠窗口不可见，请转到 View | Solution Explorer。

- (4) 在 Solution Explorer 中右击项目并单击 Properties 按钮。进入 Configuration Properties | C/C++ | Precompiled Headers，将 Precompiled Header 选项设置为 Not Using Precompiled Headers，然后单击 OK 按钮。另外，在 Solution Explorer 中选择 pch.cpp 和 pch.h 文件并删除它们。使用预编译头文件是 Visual

C++改进构建时间的一个特性，但是这个测试项目没有使用它。

(5) 将名为 `ObjectPoolTest.h` 和 `ObjectPoolTest.cpp` 的空文件添加到测试项目中。

现在就可以开始给代码添加单元测试。

最常见的做法是将单元测试分为多个逻辑测试组，称为测试类(test class)。你将创建一个名为 `ObjectPoolTest` 的测试类。`ObjectPoolTest.h` 的基本代码起初如下所示：

```
#pragma once
#include <CppUnitTest.h>

TEST_CLASS(ObjectPoolTest)
{
    public:
};
```

上面的代码定义了测试类 `ObjectPoolTest`，但语法与标准 C++稍有不同，目的是使测试框架可自动发现所有测试。

如果在运行测试类中定义的测试前执行一些任务，或在运行测试后执行清理，那么可以实现如下加粗显示的初始化方法和清理方法：

```
TEST_CLASS(ObjectPoolTest)
{
    public:
        TEST_CLASS_INITIALIZE(setUp);
        TEST_CLASS_CLEANUP(tearDown);
};
```

由于 `ObjectPool` 的测试相对简单和独立，因此 `setUp()` 和 `tearDown()` 的空定义便足以满足需要；甚至可将它们完全删除。如果确实需要它们，`ObjectPoolTest.cpp` 源文件在开始阶段应如下所示：

```
#include "ObjectPoolTest.h"

void ObjectPoolTest::setUp() { }
void ObjectPoolTest::tearDown() { }
```

这是开始开发单元测试时需要的所有初始代码。

注意：

在实际场景中，通常将要测试的代码和测试代码放在不同项目中。为保持简洁起见，这里没有这么做。

2. 编写第一个测试

这可能是你第一次接触 Visual C++测试框架或单元测试，因此第一个测试将非常简单。它测试 $0 < 1$ 的正确性。

单元测试只是测试类的一个方法。要创建一个简单测试，只需要将其声明添加到 `ObjectPoolTest.h` 文件中。

```
TEST_CLASS(ObjectPoolTest)
{
    public:
        TEST_CLASS_INITIALIZE(setUp);
        TEST_CLASS_CLEANUP(tearDown);
```

```
TEST_METHOD(testSimple); // Your first test!  
};
```

这个测试的实现使用了 `Assert::IsTrue()`，`Assert::IsTrue()`在 `Microsoft::VisualStudio::CppUnitTestFramework` 名称空间中定义，可执行实际测试。`Assert::IsTrue()`验证给定的表达式是否返回 `true`。如果表达式返回 `false`，则测试失败。`Assert` 提供了更多的辅助函数，如 `AreEqual()`、`IsNull()`、`Fail()`、`ExpectException` 等。这里，测试生成 0 小于 1。下面是更新后的 `ObjectPoolTest.cpp` 文件：

```
#include "ObjectPoolTest.h"  
  
using namespace Microsoft::VisualStudio::CppUnitTestFramework;  
  
void ObjectPoolTest::setUp() { }  
void ObjectPoolTest::tearDown() { }  
  
void ObjectPoolTest::testSimple()  
{  
    Assert::IsTrue(0 < 1);  
}
```

这就是全部。当然，大多数单元测试所做的工作都比这更有趣，不会只是一个简单的断言。你将看到，常见模式是执行一些计算，然后断言结果就是预期的值。使用 Visual C++测试框架，甚至不需要考虑异常情况，该框架会根据需要捕获和报告异常。

3. 构建和运行测试

要构建解决方案，可单击 `Build | Build Solution`，打开 `Test Explorer`(`Test | Windows | Test Explorer`)，如图 30-4 所示。

构建解决方案后，`Test Explorer` 将自动显示所有已发现的单元测试。这里，它显示 `testSimple` 单元测试。要运行测试，可单击窗口左上角的 `Run All` 链接。此后，`Test Explorer` 显示单元测试是否成功。此处的单个单元测试成功了，如图 30-5 所示。

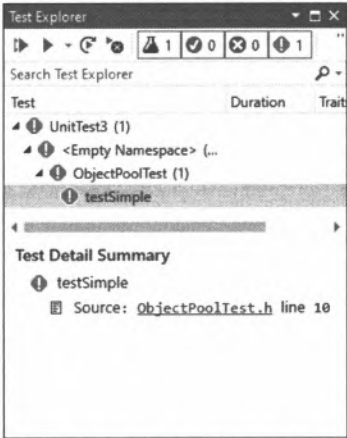


图 30-4 打开 Test Explorer 窗口

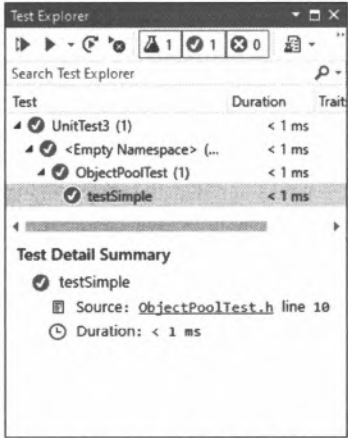


图 30-5 单个单元测试成功了

如果修改代码，将断言改成 `1 < 0`，测试将失败，`Test Explorer` 会报告这次失败，如图 30-6 所示。

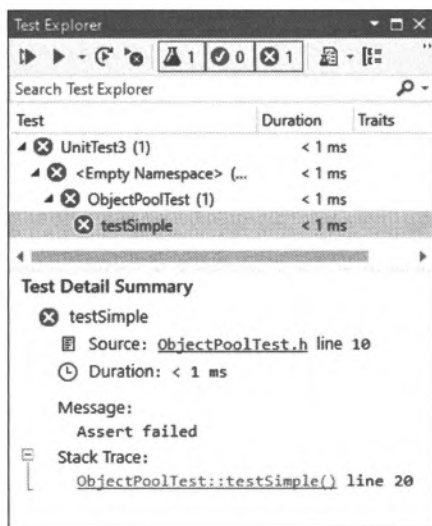


图 30-6 报告测试失败

Test Explorer 窗口的下半部分显示与所选单元测试相关的有用信息。如果单元测试失败，它会准确指出失败之处。在本例中，它指出断言失败了。在出现失败时，还捕获栈跟踪。可以单击栈跟踪中的超链接，直接跳转到出问题的行，这对调试而言十分有用。

4. 负面测试

可以编写负面测试(negative test)，即应当失败的测试。例如，可编写一个负面测试来测试某个方法抛出了预期的异常。Visual C++测试框架提供了 `Assert::ExpectException()` 函数来处理预期的异常。例如，以下单元测试使用 `ExpectException()` 执行一个 lambda 表达式，抛出 `std::invalid_argument` 异常。`ExpectException()` 的模板类型参数指定了预期异常的类型。

```
void ObjectPoolTest::testException()
{
    Assert::ExpectException<std::invalid_argument>(
        []{throw std::invalid_argument("Error"); },
        L"Unknown exception caught.");
}
```

5. 添加真正的测试

上面设置了框架，并完成一个简单测试，现在分析 `ObjectPool` 类，并编写一些代码对其进行实际测试。下面的所有测试都将添加到 `ObjectPoolTest.h` 和 `ObjectPoolTest.cpp`，与前面的初始测试一样。

首先复制创建的 `ObjectPoolTest.h` 文件旁的 `ObjectPool.cppm` 模块接口文件，然后将其添加到项目中。

在编写测试前，需要一个与 `ObjectPool` 一起使用的辅助类。`ObjectPool` 创建某种类型的对象，并在接收到请求时将它们提供给调用者。一些测试需要检查目前检索到的对象与前面检索到的对象是否相同。为此，一种方法是创建一个串行对象池(即对象有一个单调递增的序列号)。下面的代码显示了用于定义这样一个类的 `Serial.cppm` 模块接口文件：

```
module;
#include <cstdint> // For size_t
```

```

export module serial;

export class Serial
{
    public:
        // A new object gets a next serial number.
        Serial() : m_serialNumber { ms_nextSerial++ } { }
        size_t getSerialNumber() const { return m_serialNumber; }
    private:
        static inline size_t ms_nextSerial { 0 }; // The first serial number is 0.
        size_t m_serialNumber;
};

```

此时进入测试环节！作为初始的完好性检查(sanity check)，可能需要一个创建对象池的测试。如果在创建期间抛出任何异常，Visual C++测试框架将报告错误。代码根据 AAA 原则编写：安排(Arrange)、行动(Act)、断言(Assert)；测试为了可以运行设置一切，然后执行一些工作，最后断言预期结果。这也通常称为 if-when-then 原则。建议在单元测试中添加注释，这些注释实际上以 IF、WHEN 和 THEN 开头，这样测试的 3 个阶段就很明显。

```

void ObjectPoolTest::testCreation()
{
    // IF nothing

    // WHEN creating an ObjectPool
    ObjectPool<Serial> myPool;

    // THEN no exception is thrown
}

```

切记在头文件中添加 TEST_METHOD(testCreation);语句。所有后续测试同样如此。还需要给 ObjectPoolTest.cpp 源文件添加 object_pool 和 serial 模块的导入声明。

```

import object_pool;
import serial;

```

第二个测试 testAcquire()用于测试特定的一段公共功能：ObjectPool 提供对象的能力。此时，需要断言的内容不多。为证实得到的 Serial 引用的有效性，测试断言序列号大于或等于 0：

```

void ObjectPoolTest::testAcquire()
{
    // IF an ObjectPool has been created for Serial objects
    ObjectPool<Serial> myPool;
    // WHEN acquiring an object
    auto serial { myPool.acquireObject() };
    // THEN we get a valid Serial object
    Assert::IsTrue(serial->getSerialNumber() >= 0);
}

```

下一个测试更有趣。ObjectPool 不应该将同一个 Serial 对象提供两次。该测试通过从对象池中检索对象编号，检查 ObjectPool 的独有性。把检索到的对象存储在一个 vector 中，确保在每个循环迭代后，不会自动释放到对象池中。如果对象池合理分发唯一对象，则它们的序列号都不应当重复。该实现使用标准库的 vector 和 set 容器；如果不熟悉这些容器，可参见第 18 章“标准库容器”。代码需要 <memory>、<vector>和<set>。


```

void ObjectPoolTest::testExclusivity()
{
    // IF an ObjectPool has been created for Serial objects
    ObjectPool<Serial> myPool;
    // WHEN acquiring several objects from the pool
    const size_t numberOfObjectsToRetrieve { 10 };
    vector<shared_ptr<Serial>> retrievedSerials;
    set<size_t> seenSerialNumbers;

    for (size_t i { 0 }; i < numberOfObjectsToRetrieve; i++) {
        auto nextSerial { myPool.acquireObject() };

        // Add the retrieved Serial to the vector to keep it 'alive',
        // and add the serial number to the set.
        retrievedSerials.push_back(nextSerial);
        seenSerialNumbers.insert(nextSerial->getSerialNumber());
    }

    // THEN all retrieved serial numbers are different.
    Assert::AreEqual(numberOfObjectsToRetrieve, seenSerialNumbers.size());
}

```

到现在为止，最终的测试检查释放功能。一旦释放一个对象，ObjectPool 就可再次提供该对象。在回收所有已释放的对象前，对象池不应该创建额外对象。

该测试首先从对象池中检索 10 个 Serial 对象，将它们存储在 vector 中以保持活动状态，记录最后检索的 Serial 对象的序列号。一旦 10 个对象全部被检索，将所有对象返回对象池中。

该测试的第二阶段又从对象池中检索 10 个对象，将它们存储在 vector 中以保持活动状态。所有这些检索到的对象必须有一个原始指针，该指针在测试的第一阶段中就已经被看到。这验证对象是否被池正确重用。该实现使用<algorithm>中的 std::sort() 算法。

```

void ObjectPoolTest::testRelease()
{
    // IF an ObjectPool has been created for Serial objects
    ObjectPool<Serial> myPool;
    // AND we acquired and released 10 objects from the pool, while
    // remembering their raw pointers
    const size_t numberOfObjectsToRetrieve { 10 };
    // A vector to remember all raw pointers that have been handed out by the pool.
    vector<Serial*> retrievedSerialPointers;
    vector<shared_ptr<Serial>> retrievedSerials;
    for (size_t i { 0 }; i < numberOfObjectsToRetrieve; i++) {
        auto object { myPool.acquireObject() };
        retrievedSerialPointers.push_back(object.get());
        // Add the retrieved Serial to the vector to keep it 'alive'.
        retrievedSerials.push_back(object);
    }
    // Release all objects back to the pool.
    retrievedSerials.clear();

    // The above loop has created 10 Serial objects, with 10 different
    // addresses, and released all 10 Serial objects back to the pool.
    // WHEN again retrieving 10 objects from the pool, and
    // remembering their raw pointers.
    vector<Serial*> newlyRetrievedSerialPointers;

```



```

for (size_t i { 0 }; i < numberOfObjectsToRetrieve; i++) {
    auto object { myPool.acquireObject() };
    newlyRetrievedSerialPointers.push_back(object.get());
    // Add the retrieved Serial to the vector to keep it 'alive'.
    retrievedSerials.push_back(object);
}
// Release all objects back to the pool.
retrievedSerials.clear();

// THEN all addresses of the 10 newly acquired objects must have been
// seen already during the first loop of acquiring 10 objects.
// This makes sure objects are properly re-used by the pool.
sort(begin(retrievedSerialPointers), end(retrievedSerialPointers));
sort(begin(newlyRetrievedSerialPointers), end(newlyRetrievedSerialPointers));
Assert::IsTrue(retrievedSerialPointers == newlyRetrievedSerialPointers);
}

```

如果添加所有这些测试并运行它们，Test Explorer 将如图 30-7 所示。当然，如果一个或多个测试失败，将不得不面对单元测试中的典型问题：是测试存在问题，还是代码存在问题？

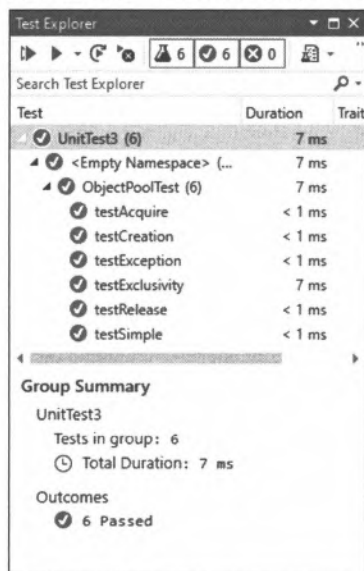


图 30-7 Test Explorer 窗口

6. 对测试进行调试

Visual C++测试框架允许方便地对失败的单元测试进行调试。Test Explorer 显示在单元测试失败时捕获的栈跟踪，并包含超链接，超链接直接指向存在问题的代码行。

但有时，有必要直接在调试器中运行单元测试，以便在运行时检测变量，一行一行地单步调试代码。为此，在单元测试的一些代码行中设置断点。然后在 Test Explorer 中右击单元测试，单击 Debug 按钮。测试框架开始在调试器中运行所选的测试，在断点处中断。此后，可以根据自己的需要单步调试代码。

7. 更多的单元测试

利用刚才介绍的测试，你将很好地理解如何开始为实际代码编写专业级测试代码。不过，这只是

冰山一角。上面的示例有助于思考应当为 `ObjectPool` 模板编写的其他测试。

例如，可以向 `ObjectPool` 添加一个 `capacity()` 方法，该方法返回已经分发的对象数量和仍然可用的对象数量总和，而无序分配新的内存块。这类似于 `vector` 中的 `capacity()` 方法，该方法返回不需要重新分配就可以存储在 `vector` 中的元素的总数。一旦拥有这个方法，可以引入一个测试，验证池的元素数量是否总是比前一次池增长时增长了两倍。

对于给定的代码片段，可编写的单元测试数量是没有穷尽的，这是单元测试的亮点。如果自己正在思考代码如何响应某种情形，那就是单元测试。如果子系统的特定方面似乎存在问题，则扩大特定领域的单元测试范围。即使只是想站在客户的角度分析类的工作情况，编写单元测试也是获得不同视角的极佳方式。

30.3 模糊测试

模糊测试，也称为 `fuzzing`，设计一个模糊器，模糊器可以自动生成程序或组件的随机输入数据，试图找到未处理的边缘情况。通常会提供一个配方，该配方指定如何构造输入数据，以便将其用作程序的输入。如果向程序提供了结构明显错误的输入，它的数据解析器很可能会立即拒绝。模糊器的工作就是试图生成没有明显结构错误的输入数据，这样就不会被程序立刻拒绝，它会在程序执行过程中进一步触发一些错误的逻辑。由于模糊器生成的是随机输入数据，它需要大量的资源来覆盖整个输入空间。一种选择是在云端运行这种模糊测试场景。有几个库可用于实现模糊测试，例如 `libFuzzer`(llvm.org/docs/LibFuzzer.html)和 `honggfuzz`(github.com/google/honggfuzz)。

30.4 高级测试

单元测试是抵御 `bug` 的第一道防线，是更大的测试过程的一部分。高级测试并非像单元测试那样关注较小焦点，而是重点关注产品的各个部分如何协同工作。在某种程度上，高级测试更难编写，因为更难确定需要编写哪些测试。只有在确认各个组件可一起工作后，才能生成程序是可工作的。

30.4.1 集成测试

集成测试的范围是组件结合区域。单元测试通常在单个类的级别上进行，而集成测试通常涉及两个类或更多类。集成测试擅长测试两个组件(往往由两个不同的编程人员所写)之间的交互。事实上，在编写集成测试期间，经常能发现设计中的重要不兼容之处。

1. 集成测试示例

并不存在有关应当编写哪些集成测试的硬性规则，这里将列举一些示例，这些示例有助于了解使用集成测试的时机。以下场景描述适于使用集成测试的情形，但并未涵盖每种可能的情形。与单元测试一样，随着练习时间的增加，直觉会告诉你哪些集成测试是有用的。

基于 JSON 的文件串行化器

假设项目包含一个持久化层，用于将某种类型的对象保存到磁盘中，并从磁盘读回对象。要序列化数据，常用方式是使用 `JSON` 格式来串行化数据；因此，添加一个位于文件 `API` 上的 `JSON` 转换层是一种合理的组件分解方式。可对这些组件进行彻底的单元测试。`JSON` 层可拥有单元测试，以确认可将不同类型的对象正确地转换为 `JSON`，并从 `JSON` 进行填充。文件 `API` 可拥有以下测试：从磁盘读取文件，将数据写入磁盘文件，以及更新和删除磁盘上的文件。

当这些模块开始共同工作时，集成测试将是适合的。至少要有一个集成测试，它通过 JSON 层将对象保存到磁盘，之后将对象读回，并与原对象进行比较。由于同时涵盖两个模块，因此它是一个基本的集成测试。

共享资源的读取和写入操作

假设程序包含一种由不同组件共享的数据结构。例如，股票交易程序可有购买和销售请求队列。与接收股票交易请求相关的组件可将订单添加到队列中，与执行股票交易相关的组件可从队列中删除订单。可针对队列类执行单元测试，但只有用使用队列类的实际组件对队列类进行测试后，才能确认一些假设是否有误。

良好的集成测试是将股票请求组件和股票交易组件用作队列类的客户端。可以编写一些示例订单，通过客户端组件确认它们可以成功地进入和离开队列。

与第三方库打包

集成测试未必总针对自己代码的集成点。有时会编写集成测试，以测试代码与第三方库之间的交互。例如，可能使用一个数据库连接库，与关系数据库系统进行交互。也可能围绕这个库构建一个面向对象的包装器，它附加支持连接缓存，或提供一个更友好的界面。这是一个需要测试的重要集成点，尽管包装器提供了更有用的数据库接口，但它也可能误用原始库。

换句话说，编写正确的包装器是一件好事，但编写引入 bug 的包装器只会招致灾难。

2. 集成测试方法

在实际编写集成测试时，集成测试和单元测试之间往往有一条明显的界线。如果修改单元测试，使其接触另一个组件，会瞬间变成集成测试吗？在某种意义上，答案是悬而未决的，原因在于，良好的测试就是良好的测试，与属于哪类测试无关。建议将集成测试和单元测试视为两种测试方法，没必要给每个单独测试贴上明确的分类标签。

在实现方面，集成测试经常使用单元测试框架编写，这使两种测试方法之间的界限变得更模糊。事实证明，单元测试框架允许更方便地编写“是/否”测试，并能提供有用的结果。从框架的角度看，无论测试是查看单个功能单元，还是查看两个组件的交互，都几乎没有什么区别。

但出于性能和编排原因，可能需要将单元测试与集成测试区分开来。例如，组织可能要求每个人必须在签入新代码前运行集成测试，但对运行无关的单元测试的要求则较为宽松。将这两类测试分开也将使结果变得更有价值。如果测试在 JSON 类测试中失败，则能明确推断 bug 存在于相应的类中，而非存在于类和文件 API 的交互中。

30.4.2 系统测试

系统测试在高于集成测试的级别操作。系统测试通盘分析程序。系统测试中会有虚拟用户，用来模拟使用程序的人。当然，必须为虚拟用户提供执行操作的脚本。其他系统测试依赖于脚本或一组固定的输入，也依赖于预期的输出。

与单元测试和集成测试类似，系统测试执行具体的测试，并预期具体的结果。经常使用系统测试来确保不同功能可结合在一起工作。

从理论上讲，完全经过系统测试的程序包含每项功能的每个测试。这种方法很快就会变得笨重不堪，但仍需要努力测试多项功能的组合。例如，图形程序的系统测试导入图像，旋转图像，应用模糊滤镜，转换为黑白图像，然后保存图像。系统测试会将保存的图像与包含预期结果的文件进行比较。

令人遗憾的是，关于系统测试的具体规则很少，因为系统测试高度依赖于实际应用程序。如果应

用程序处理文件时不存在用户交互，那么系统测试的编写与单元测试和集成测试十分类似。对于图形程序而言，虚拟用户方式是最合适的。对于服务器应用程序而言，可能需要构建存根客户端来模拟网络流量。重点在于，实际测试的是程序的实际使用情况，而非程序某部分的实际使用情况。

30.4.3 回归测试

回归测试更多是测试概念而非具体测试类型。具体想法是：一旦一个功能可以工作，开发人员就倾向于将它放在一边，并假设它将继续工作。遗憾的是，新功能和其他代码更改通常会悄无声息地破坏以前可以正常工作的功能。

回归测试常用于功能的完好性检查，这些功能在一定程度上是完整的、可工作的。如果一项更改破坏了功能，编写得当的回归测试会阻止其通过。

如果公司安排了一支 QA 测试团队，回归测试可能采用手动测试形式。测试者扮演用户，经历一系列步骤，逐步测试在前一个版本中可以工作的每项功能。如果认真执行，这种方法是彻底的、准确的，但扩展性不是很好。

另一个极端是，也可构建一个完全自动化的系统，以虚拟用户的身份执行每项功能。此时，编写脚本是一项挑战，不过，可借助几款商业软件包和非商业软件包为各类应用程序较方便地编写脚本。

冒烟测试(smoke testing)位于这两个极端之间。有些测试只测试应当可工作的最重要功能的一个子集。具体想法是：如果某个功能受损，会立即表现出来。如果冒烟测试通过，其后应当执行多个严格的手动测试或自动化测试。“冒烟测试”术语由来已久，首先用于电子行业。在构建包含真空管、电阻器等不同元件的电路后，问题是“连线是否正确？”解决方案是“连接在一起后，通电，看一下是否会冒烟。”如果冒烟，则表明设计或连线有误。通过查看哪处冒烟了，可以确定错误。

一些 bug 简直是梦魇，它们不仅可怕，还经常发生。复发的 bug 令人感到沮丧，未能利用工程资源。即使由于某些原因，决定不编写一套回归测试，也应当针对修复的 bug 编写几个回归测试。

这样一来，可证明 bug 已经修复，可以设置警报，若 bug 返回(即修改被取消或未完成，或两个分支未正确合并到主开发分支)，将触发警报。如果之前已修复 bug 的回归测试失败了，修复也较为容易，因为回归测试可参照以前的 bug 编号，并描述第一次的修复方式。

30.5 用于成功测试的建议

作为一名软件工程师，在测试方面可能负责基本的单元测试，也可能负责完整管理自动化测试系统。测试角色和风格差别很大，下面是根据经验给出的提示，它们会在不同的测试场景提供帮助：

- 花一些时间来设计自动化测试系统。整日运行的系统将能快速检测故障。此类系统会在故障发生时自动给工程师发送电子邮件，或在房间中心位置发出提示音，从而增加问题的可见度。
- 不要忘记压力测试。对于数据库访问类而言，即使一整套单元测试都通过了，但若几十个线程同时使用，也可能无力支持。应当在实际面对的极端条件下测试产品。
- 模拟客户的系统，在不同的平台上进行测试。要在多个操作系统上进行测试，一种方法是使用虚拟环境，这样，可在同一台物理计算机上运行多个不同的操作系统。
- 编写一些测试时，可故意将一些错误注入系统中。例如，可编写一个测试，在读取文件时删除文件，或模拟网络操作期间的网络中断情形。
- bug 和测试密切相关。应当编写回归测试来证实 bug 修复的正确性。测试的注释可指出原始的 bug 编号。

- 不要删除正在失败的测试。如果同事正在费力处理一个 bug，却发现你删除了测试，他会来找你。

最重要的提示是要记住：测试是软件开发的一部分。如果同意这一观点，并在开始编码前接受这一观点，那么当功能开发完毕时，你不至于大惊失色；不过，仍然需要做更多工作来证明它是可以工作的。

30.6 本章小结

本章介绍基本的测试信息，所有专业程序员都需要了解这些信息。确切地讲，单元测试是提高代码质量的最简便有效的方式。更高级的测试覆盖用例、模块之间的同步以及回归保护。无论在测试过程中扮演什么角色，现在都能在各个级别充满信心地设计、创建和审查测试。

前面介绍了如何查找 bug，下面讨论如何修复 bug。因此，第 31 章“熟练掌握调试技术”将介绍有效调试的技术和策略。

30.7 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试自己找到答案。

练习 30-1 测试的 3 种类型是什么？

练习 30-2 列出你能想到的以下代码段的单元测试列表：

```
export class Foo
{
    public:
        // Constructs a Foo. Throws invalid_argument if a >= b.
        Foo(int a, int b) : m_a { a }, m_b { b }
        {
            if (a >= b) {
                throw std::invalid_argument { "a should be less than b." };
            }
        }

        int getA() const { return m_a; }
        int getB() const { return m_b; }
    private:
        int m_a, m_b;
};
```

练习 30-3 如果你正在使用 Visual C++，使用 Visual C++测试框架实现习题 30-2 中列出的单元测试。

练习 30-4 假设你写了一个计算数字阶乘的函数。 n 的阶乘写成 $n!$ ，是 1 到 n 所有数字的乘积。例如， $3! = 1 \times 2 \times 3$ 。你决定遵循本章给出的建议，为你的代码编写单元测试。运行代码来计算 $5!$ ，然后编写一个单元测试来验证。当要计算 5 的阶乘时，代码是否生成了计算出的数字。你如何看待这样的单元测试。

第31章

熟练掌握调试技术

本章内容

- 调试的基本定律和 bug 分类学
- 避免 bug 的技巧
- 如何为 bug 做好规划
- 不同的内存错误类型
- 如何通过调试器定位导致 bug 的代码

代码中肯定会有 bug。每个专业程序员都想编写没有 bug 的代码，但现实情况是，几乎没有软件工程师能成功做到这一点。计算机用户都知道，bug 是计算机软件中特有的。你编写的软件可能也不例外。因此，除非打算贿赂同事，让他们修复所有的 bug，否则不知道如何调试 C++ 代码，就不可能成为专业的 C++ 程序员。区别富有经验的程序员和新手的很重要因素之一往往是他们的调试技巧。

尽管调试的重要性显而易见，但调试往往在课程和书籍中很少引起足够重视。调试似乎是一类所有人都希望掌握的技能，但没人知道如何讲授这门技能。本章试图提供具体的指导方针和调试技术。

本章首先介绍调试的基本定律和 bug 分类学，之后讲解避免 bug 的技巧。为 bug 做好规划的技术包含错误日志、调试跟踪、断言和崩溃转储。然后讲述出现问题时调试的具体技巧，包括 bug 重现的技术、可重现 bug 的调试、不可重现 bug 的调试、内存错误的调试以及多线程程序的调试。本章最后举一个逐步调试的例子。

31.1 调试的基本定律

调试的第一原则是要对自己诚实，承认自己的程序一定会包含 bug。这种现实的评估可使自己投入最大努力，从一开始就防止 bug 进入程序，同时尽可能包含必要功能，使调试尽可能简单。

警告：

调试的基本定律：在编写代码时要注意避免 bug，但要为 bug 的出现制订好规划。

31.2 bug 分类学

计算机程序中的 bug 是指不正确的运行时行为。这种不期望的行为既包含灾难性的 bug，也包含非灾难性的 bug。灾难性 bug 的例子包括程序死亡、数据损坏、操作系统故障或其他一些可怕的结果。灾难性的 bug 还可表现在软件或运行软件的计算机系统之外，例如医疗软件可能包含灾难性的 bug，导致病人接受过量的辐射。非灾难性的 bug 以轻微方式导致程序行为不正确，例如，Web 浏览器可能返回错误的网页，或电子表格程序可能错误地计算一系列数据的标准偏差。这些也被称为逻辑错误 (cosmetic bug)。

还有一类 bug 称为 cosmetic bug，这类 bug 发生时，有些地方看上去工作不正常，但别的方面工作正常。例如，用户界面上的一个按钮在不该启用时启用了，但单击时什么也不做。所有计算都完全正确，程序不会崩溃，但看起来不如预期那么“好”。

这种 bug 的底层原因或根本原因是程序中的错误导致这种不正确行为。调试程序的过程包括找出错误根源以及修复代码，以免这样的错误再次发生。

31.3 避免 bug

编写完全无 bug 的代码是不可能的，所以调试技巧很重要。然而，通过一些技巧可以帮助减少 bug 的数量。

- **从头到尾阅读本书：**仔细学习 C++ 语言，尤其是指针和内存管理。然后，将这本书推荐给朋友和同事，让他们也避免 bug。
- **编写代码之前做好设计：**一边编码一边设计容易导致令人费解的设计，这种设计更难理解、更容易出错，也更容易忽略可能的边缘情况和错误条件。
- **审查代码：**在专业软件开发中，请同事审查代码。有时从新视角可发现问题。
- **测试、测试、再测试：**全面测试代码，再让其他人测试代码，他们更可能找到你未想到的问题。
- **编写自动单元测试：**单元测试用于测试独立功能。应给所有已实现的特性编写单元测试。自动运行这些单元测试，作为继续集成配置的一部分，或在每次本地编译后自动运行单元测试。详见第 30 章“熟练掌握测试技术”。
- **预计错误条件，并恰当地处理它们：**特别是处理文件和网络错误时，要预计并处理错误。这些情况是会发生的。详见第 13 章“解密 C++ I/O”和第 14 章“错误处理”。
- **使用智能指针避免内存泄漏：**智能指针在不再需要时，会自动释放资源。
- **不要忽略编译警告：**配置编译器，用较高的警告级别来编译。不要盲目地忽略警告。理想情况下，应在编译器中启用一个选项，将警告处理为错误。这将强制解决每个警告。在 GCC 中，可以将 `-Werror` 传给编译器，来将所有警告视为错误。在 Visual C++ 中，打开项目的属性，进入 Configuration Properties/ C/C++ | General，启用“将警告视为错误”选项。
- **使用静态代码分析：**静态代码分析器会分析源代码，有助于指出代码中的问题。理想情况下，静态代码分析是在集成开发环境 (IDE) 输入代码时实时完成的，以便及早发现问题。还可以在构建过程中设置自动执行静态代码分析。在互联网上有很多不同的分析工具，包括免费的和商业的。
- **使用良好的编码风格：**尽力提高可读性和简洁性，使用有意义的名称，不要使用简写形式，添加代码注释 (不仅仅是接口注释)，使用 `override` 关键字等，这将便于其他人理解你的代码。

31.4 为 bug 做好规划

程序应该包含一些特性，在不可避免的 bug 出现时能更容易地调试。本节介绍这些特性，并给出示例实现，可在需要时将这些实现融入自己的程序。

31.4.1 错误日志

想象这样的情景：你刚刚发布旗舰产品的新版本，首批用户反馈之一就是“这个程序‘停止工作’了”。你尝试从用户处获得更多信息，最后发现这个程序在操作期间崩溃了。用户不太记得当时在做什么，或是否有任何错误消息。你打算如何解决这个问题？

现在设想同样的情形，但除了从用户获得的有限信息之外，还能检查用户计算机上的错误日志。在日志中，有一条来自程序的错误：“错误：无法打开 config.xml 文件”。查看生成该错误消息的地方附近的代码，发现从文件读取内容的代码行并未检查是否成功打开了文件。这就是产生 bug 的根源！

记录错误日志是将错误消息写入持久存储的过程，这样可在应用程序甚至计算机崩溃时查看错误消息。看过这个示例场景后，你可能仍然怀疑这一策略。如果程序遇到错误，难道程序的行为不能很明显地表现出来吗？如果有错误，难道用户注意不到吗？正如前面的例子所示，用户反馈并不总是准确或完整。此外，还有很多程序，例如操作系统内核和长期运行的后台程序，例如 UNIX 上的 `inetd`(internet service daemon)和 `syslogd`，都是没有交互地在计算机上无人值守地运行。这些程序与用户沟通的唯一途径就是错误日志。许多情况下，程序也可能希望自动从某些错误中恢复，并对用户隐藏错误。记录这些错误对改进程序的整体稳定性是很有价值的。

因此，当程序遇到问题时，应该记录错误。这样，如果用户报告 bug，就可以检查计算机上的日志文件，看看程序在遇到 bug 之前是否报告了错误。遗憾的是，错误日志和平台相关：C++ 不包含标准的日志机制。平台相关的日志机制包括 UNIX 上的 `syslog` 工具和 Windows 上的事件报告 API。应该查阅开发平台的文档，还有一些开源的跨平台日志框架的实现，下面是两个例子：

- <http://log4cpp.sourceforge.net/> 上的 `log4cpp`
- <http://www.boost.org/> 上的 `Boost.Log`

错误日志是要添加到程序的一项卓越特性，你可能希望在代码中每隔几行就记录错误消息，这样在任何 bug 发生时，都能跟踪到正在执行的代码路径。这类错误日志消息被恰当地称为“跟踪”。

然而，基于两个原因，不能将这些跟踪写入错误日志。首先，写入持久存储的速度很慢。即使在异步写日志的系统上，记录如此大量的信息也会降低程序的运行速度。其次，也是最重要的是，放在跟踪中的大部分信息都不适合让最终用户看到。这些信息只会迷惑用户，导致不必要的服务请求。尽管如此，在正确情形下跟踪仍是一项重要的调试技术。

下面是关于程序应该记录的错误类型的具体指导方针：

- 不可恢复的错误，例如无法分配内存或系统调用意外失败。
- 管理员可采取行动的错误，例如内存不足、数据文件格式有误、不能写入磁盘或者网络连接关闭。
- 意外错误，例如没有预计到的代码路径或变量取了意料之外的值。注意，代码应该能“预料”用户会输入非法数据，并正确地处理。意外错误在程序中表现为 bug。
- 潜在的安全漏洞，例如网络连接试图访问未经授权的地址，或太多的网络连接尝试(拒绝服务)。

记录警告或可恢复的错误也是有益的，这允许调查能否避免它们。

此外，大多数日志记录 API 允许指定日志级别或错误级别、一般错误、警告和信息。可在低于“错误”日志级别的日志级别记录无错误条件。例如，记录应用程序中重要的状态变化，或程序的启动和关闭。也可以考虑允许用户在运行时调整程序的日志级别，这样他们可以自行调整日志量。

31.4.2 调试跟踪

调试复杂问题时，公共的错误消息通常不会包含足够的信息。往往需要完整的跟踪代码路径或 bug 出现之前变量的值。除了基本信息，在调试跟踪中包含以下信息也会很有帮助：

- 如果是多线程程序，将会包含线程 ID。
- 生成跟踪信息的函数名。
- 生成跟踪信息的代码所在源文件的名称。

可通过特殊的调试模式或环形缓冲区将这个跟踪添加到程序中。注意在多线程程序中，必须使跟踪日志是线程安全的。多线程编程参见第 27 章“C++多线程编程”。

注意：

可以用文本格式写入跟踪文件，但这样做时，要留意日志信息过多的情况。不要通过日志文件遗漏智能属性。另一种做法是使用只有自己能读懂的格式写入文件。

1. 调试模式

添加调试跟踪的第一种技术是给程序提供调试模式。在调试模式下，程序将跟踪输出写入到标准错误或文件中，也可能在执行过程中进行额外检查。有几种方法可以向程序添加调试模式。注意，所有这些示例都以文本格式写入跟踪信息。

启动时调试模式

启动时调试模式允许根据一个命令行参数，让应用程序在调试模式或非调试模式下运行。这种方法会在发行版二进制文件中包含调试代码，而且允许在客户现场启用调试模式。但要求用户重新启动程序，才能运行调试模式，这可能会导致无法获得某些 bug 的有用信息。

下例是一个带有启动时调试模式的简单程序。这个程序没有做任何有用的事情，只是为了演示技术。

所有日志功能都放在 `Logger` 类中，这个类有两个静态数据成员：日志文件名和布尔值。其中布尔值表示是否启用日志功能。这个类包含一个静态的公共可变参数模板方法 `log()`。可变参数模板参见第 26 章“高级模板”。注意，每次调用 `log()` 时，都会打开、写入并关闭文件。这可能降低一点性能，但能保证日志记录的正确性，后者是更重要的。

```
class Logger
{
public:
    static void enableLogging(bool enable) { ms_loggingEnabled = enable; }
    static bool isLoggingEnabled() { return ms_loggingEnabled; }

    template<typename... Args>
    static void log(const Args&... args)
    {
        if (!ms_loggingEnabled) { return; }

        ofstream logfile { ms_debugFilename, ios_base::app };
        if (logfile.fail()) {
            cerr << "Unable to open debug file!" << endl;
            return;
        }
    }
};
```

```

    }
    // Use a C++17 unary right fold, see Chapter 26.
    (logfile << args,...);
    logfile << endl;
}
private:
    static inline const string ms_debugFilename { "debugfile.out" };
    static inline bool ms_loggingEnabled { false };
};

```

下面的辅助宏用于简化记录日志。它使用了 C++ 标准定义的 `__func__`，这个预定义的变量包含当前的函数名。

```
#define log(...) Logger::log(__func__, "(): ", _VA_ARGS_)
```

这个宏将代码中对 `log()` 的每次调用都替换为对 `Logger::log()` 方法的调用。这个宏自动将函数名作为第一个参数传递给 `Logger::log()` 方法。例如，假设以如下方式调用这个宏：

```
log("The value is: ", value);
```

`log()` 宏将这一行代码替换为：

```
Logger::log(__func__, "(): ", "The value is: ", value);
```

启动时调试模式需要解析命令行参数，确定是否应启动调试模式。但 C++ 没有解析命令行参数的标准库功能。这个程序使用简单的 `isDebugSet()` 函数，在所有的命令行参数中检查调试标记，但解析所有命令行参数的函数应更加完善。

```

bool isDebugSet(int argc, char* argv[])
{
    for (int i { 0 }; i < argc; i++) {
        if (strcmp(argv[i], "-d") == 0) {
            return true;
        }
    }
    return false;
}

```

使用一些测试代码试验这个示例中的调试模式，其中定义了两个类 `ComplicatedClass` 和 `UserCommand`，这两个类都定义了 `operator<<`，以将它们的实例写入流，因为 `Logger` 类使用这个运算符将对象转存到日志文件中。

```

class ComplicatedClass { /* ... */ };
ostream& operator<<(ostream& ostr, const ComplicatedClass& src)
{
    ostr << "ComplicatedClass";
    return ostr;
}

class UserCommand { /* ... */ };
ostream& operator<<(ostream& ostr, const UserCommand& src)
{
    ostr << "UserCommand";
    return ostr;
}

```

下面是带有一些日志调用的测试代码:

```
UserCommand getNextCommand(ComplicatedClass* obj)
{
    UserCommand cmd;
    return cmd;
}

void processUserCommand(UserCommand& cmd)
{
    // Details omitted for brevity.
}

void trickyFunction(ComplicatedClass* obj)
{
    log("given argument: ", *obj);

    for (size_t i { 0 }; i < 100; ++i) {
        UserCommand cmd { getNextCommand(obj) };
        log("retrieved cmd ", i, ": ", cmd);

        try {
            processUserCommand(cmd);
        } catch (const exception& e) {
            log("exception from processUserCommand(): ", e.what());
        }
    }
}

int main(int argc, char* argv[])
{
    Logger::enableLogging(isDebugSet(argc, argv));

    if (Logger::isLoggingEnabled()) {
        // Print the command-line arguments to the trace.
        for (int i { 0 }; i < argc; i++) {
            log("Argument: ", argv[i]);
        }
    }

    ComplicatedClass obj;
    trickyFunction(&obj);

    // Rest of the function not shown.
}
```

运行这个应用程序的方法有两种:

```
> STDebug
> STDebug -d
```

只有在命令行中指定-d 参数, 才会激活调试模式。

警告:

在 C++ 中应该尽可能地避免使用宏, 因为宏很难调试。然而, 为了达到记录日志的目的, 使用简单的宏也是可以的, 这种宏可以大大简化日志记录的代码。即使如此, 有了 C++20 的 `std::source_location` 类, 可以修改示例来避免使用宏。这是本章末尾中一个练习的主题。

编译期调试模式

可使用预处理器指令 `DEBUG_MODE` 和 `#ifdef`, 选择性地将调试代码编译到程序中, 而不是通过命令行参数启用或禁用调试模式。为得到这个程序的调试版本, 在编译期应该定义 `DEBUG_MODE` 符号。编译器应该允许在编译期间定义符号, 详情请参阅编译器的文档。例如, GCC 允许通过命令行指定 `-Dsymbol`; Visual C++ 允许通过 Visual Studio IDE 指定符号, 或者通过 Visual C++ 命令行工具指定 `/D symbol`。Visual C++ 自动为调试版本定义 `_DEBUG` 符号。但该符号是 Visual C++ 特有的, 因此, 本节中的示例使用自定义符号 `DEBUG_MODE`。

这种方法的优点是, 调试代码不会被编译到发行版的二进制文件中, 因此不会增加发行版的大小。缺点是在测试或发现 bug 时, 无法在客户现场进行调试。

在可下载的源代码包中, 有一个示例实现放在 `CTDebug.cpp` 中。对于这个实现, 有一点需要专门指出, 即它包含 `log()` 宏的以下定义:

```
#ifndef DEBUG_MODE
#define log(...) Logger::log(__func__, "(): ", __VA_ARGS__)
#else
#define log(...)
#endif
```

也就是说, 如果未定义 `DEBUG_MODE`, 则会将 `log()` 的所有调用替换为“空”, 即 `no-ops`。

警告:

在 C++ 中应为使程序正确执行, 有些代码必不可少; 切不可将此类代码放入 `log()` 调用中。例如, 使用以下代码行就是自找麻烦:

```
log("Result: ", myFunctionCall());
```

若未定义 `DEBUG_MODE`, 预处理器会用 `no-ops` 替换所有 `log()` 调用, 这意味着也会删除对 `myFunctionCall()` 的调用。

运行时调试模式

提供调试模式的最灵活方式是允许运行时启用或禁用调试模式。提供这种特性的一种方法是提供一个动态控制调试模式的异步接口。在 GUI 程序中, 这个接口可采取菜单命令的形式。在 CLI(命令行界面)程序中, 这个接口可以是一条异步命令, 用于对程序进行跨进程调用(例如使用套接字、信号或远程过程调用)。该接口也可采用用户界面中的菜单命令形式。C++ 没有提供标准方式来执行进程间通信或实现用户接口, 因此这里不列举这种技术的例子。

2. 环形缓冲区

调试模式适用于调试可重现问题和运行测试。然而, bug 常出现在程序运行在非调试模式时, 而当你或客户启用调试模式时, 获得 bug 相关信息为时已晚。这个问题的解决方案之一是在任何时候都启用跟踪。通常只需要最近的跟踪来调试程序, 因此只需要保存程序执行过程中任何一点最近的跟踪。提供这种限制的一种方法是小心地使用日志文件循环。

然而, 由于性能原因, 程序最好不要不停地将这些跟踪写入磁盘, 而应将它们保存在内存中。然

后，提供一种机制，在需要时将所有跟踪消息转存到标准错误或日志文件。

一种常见方法是使用环形缓冲区(circular buffer)保存固定数目的消息，或在固定大小的内存中保存消息。当缓冲区填满时，重新开始缓冲区的开头写消息，并改写旧消息。这个循环可无限期重复。下面提供了环形缓冲区的一个实现，并说明如何在程序中使用它。

环形缓冲区接口

下面的 RingBuffer 类提供了一个简单的调试环形缓冲区。客户在构造函数中指定条目的数目，通过 addEntry()方法添加消息。一旦条目数量超过允许的数量，新条目就改写缓冲区中最老的条目。这个缓冲区还提供了一个选项，允许在向缓冲区添加条目时将条目打印出来。客户可在构造函数中指定输出流，也可用 setOutput()方法重置。最后，operator<<将整个缓存区输出到输出流。这个实现使用了可变参数模板方法，可变参数模板参见第 26 章。

```
export class RingBuffer
{
    public:
        // Constructs a ring buffer with space for numEntries.
        // Entries are written to *ostr as they are queued (optional).
        explicit RingBuffer(size_t numEntries = DefaultNumEntries,
                           std::ostream* ostr = nullptr);
        virtual ~RingBuffer() = default;

        // Adds an entry to the ring buffer, possibly overwriting the
        // oldest entry in the buffer (if the buffer is full).
        template<typename... Args>
        void addEntry(const Args&... args)
        {
            std::ostringstream os;
            // Use a C++17 unary right fold, see Chapter 26.
            ((os << args), ...);
            addStringEntry(os.str());
        }

        // Streams the buffer entries, separated by newlines, to ostr.
        friend std::ostream& operator<<(std::ostream& ostr, RingBuffer& rb);

        // Streams entries as they are added to the given stream.
        // Specify nullptr to disable this feature.
        // Returns the old output stream.

        std::ostream* setOutput(std::ostream* newOstr);
    private:
        std::vector<std::string> m_entries;
        std::vector<std::string>::iterator m_next;

        std::ostream* m_ostr { nullptr };
        bool m_wrapped { false };

        static const size_t DefaultNumEntries { 500 };

        void addStringEntry(std::string&& entry);
};
```

环形缓冲区的实现

这个环形缓冲区的实现保存固定数目的字符串对象。这肯定不是最高效的解决方案。其他可能的解决方案是为缓冲区提供固定数目字节的内存。然而，如果不是要编写高性能的应用程序，这个实现应该够用了。

对于多线程程序，可在每个跟踪项中添加线程的 ID 和时间戳。当然，在多线程程序中使用之前，必须将环形缓冲区设置为线程安全的。多线程编程参见第 27 章。

实现代码如下：

```
// Initialize the vector to hold exactly numEntries. The vector size
// does not need to change during the lifetime of the object.
// Initialize the other members.
RingBuffer::RingBuffer(size_t numEntries, ostream* ostr)
    : m_entries { numEntries }, m_ostr { ostr }, m_wrapped { false }
{
    if (numEntries == 0) {
        throw invalid_argument { "Number of entries must be > 0." };
    }
    m_next = begin(m_entries);
}

// The addStringEntry algorithm is pretty simple: add the entry to the next
// free spot, then reset m_next to indicate the free spot after
// that. If m_next reaches the end of the vector, it starts over at 0.
//
// The buffer needs to know if the buffer has wrapped or not so
// that it knows whether to print the entries past m_next in operator<<.
void RingBuffer::addStringEntry(string&& entry)
{
    // If there is a valid ostream, write this entry to it.
    if (m_ostr) { *m_ostr << entry << endl; }

    // Move the entry to the next free spot and increment
    // m_next to point to the free spot after that.
    *m_next = move(entry);
    ++m_next;

    // Check if we've reached the end of the buffer. If so, we need to wrap.
    if (m_next == end(m_entries)) {
        m_next = begin(m_entries);
        m_wrapped = true;
    }
}

// Set the output stream.
ostream* RingBuffer::setOutput(ostream* newOstr)
{
    return exchange(m_ostr, newOstr);
}

// operator<< uses an ostream_iterator to "copy" entries directly
// from the vector to the output stream.
//
// operator<< must print the entries in order. If the buffer has wrapped,
// the earliest entry is one past the most recent entry, which is the entry
```

```

// indicated by m_next. So, first print from entry m_next to the end.
//

// Then (even if the buffer hasn't wrapped) print from beginning to m_next-1.
ostream& operator<<(ostream& ostr, RingBuffer& rb)
{
    if (rb.m_wrapped) {
        // If the buffer has wrapped, print the elements from
        // the earliest entry to the end.
        copy(rb.m_next, end(rb.m_entries), ostream_iterator<string>(ostr, "\n" ));
    }

    // Now, print up to the most recent entry.
    // Go up to m_next because the range is not inclusive on the right side.
    copy(begin(rb.m_entries), rb.m_next, ostream_iterator<string>(ostr, "\n" ));

    return ostr;
}

```

使用环形缓冲区

为使用环形缓冲区，可创建它的一个实例，然后开始向其中添加消息。想打印缓冲区时，通过 `operator<<` 将缓冲区打印到相应的 `ostream`。这里将前面的启动时调试模式程序修改为使用环形缓冲区。改动的代码都加粗显示了。`ComplicatedClass` 和 `UserCommand` 类的定义，`getNextCommand()`、`processUserCommand()` 和 `trickyFunction()` 函数没有列出，它们与前面相同：

```

RingBuffer debugBuffer;

#define log(...) debugBuffer.addEntry(__func__, "(): ", __VA_ARGS__)

int main(int argc, char* argv[])
{
    // Log the command-line arguments.
    for (int i { 0 }; i < argc; i++) {
        log("Argument: ", argv[i]);
    }

    ComplicatedClass obj;
    trickyFunction(&obj);

    // Print the current contents of the debug buffer to cout.
    cout << debugBuffer;
}

```

显示环形缓冲区的内容

将跟踪调试信息保存在内存中是一个良好的开端，但为了让这些信息有用，需要有一种方法来访问这些跟踪信息，以便进行调试。

程序应该提供一个“钩子”，表示应该打印信息。这个钩子可类似于运行时用于启用调试的接口。此外，如果程序遇到致命的错误导致退出，应在退出之前将环形缓冲区自动打印至日志文件。

另一种获得这些信息的方法是获得程序的一份内存转储文件。每个平台处理内存转储的方式不同，因此应该咨询相关的平台专家，或查阅相关的书籍。

31.4.3 断言

<cassert>头文件定义了 `assert` 宏。它接收一个布尔表达式，如果表达式求值为 `false`，则打印出一条错误消息并终止程序。如果表达式求值为 `true`，则什么也不做。

警告：

一般应该避免任何可终止程序的库函数或宏，但 `assert` 宏是一个例外。如果触发了一个断言，则表示某个假设错误，或出现了灾难性的、不能恢复的错误，此时，唯一能做的就是终止应用程序，而不是继续运行。

断言可迫使程序在 `bug` 来源的确切点公开 `bug`。如果没有在这一点设置断言，那么程序可能会带着错误的值继续执行，因而 `bug` 可能在后面才显现出来。因此，断言允许尽早检测到 `bug`。

注意：

标准 `assert` 宏的行为取决于 `NDEBUG` 预处理符号：如果没有定义该符号，则发生断言，否则忽略断言。编译器通常在编译发布版本时定义这个符号。如果要在发布版本中保留断言，就必须改变编译器的设置，或者编写自己的不受 `NDEBUG` 值影响的断言。

可在代码中任何需要“假设”变量处于某些状态的地方使用断言。例如，如果调用的库函数应该返回一个指针，并且声称绝对不会返回 `nullptr`，那么在函数调用之后抛出断言，以确保指针不是 `nullptr`。

注意，假设应该尽可能少。例如，如果正在编写一个库函数，不要断言参数的合法性。相反，要对参数进行检查，如果参数非法，返回错误代码或抛出异常。

作为规则，断言应只用于真正有问题的情形，因此在开发过程中遇到的断言绝不应忽略。如果在开发过程中遇到一个断言，应修复而不是禁用它。

来看几个关于如何使用 `assert()` 的示例。下面的 `process()` 函数，要求传递给它的 `vector` 中有 3 个元素：

```
void process(const vector<int>& coordinate)
{
    assert(coordinate.size() == 3);
    // ...
}
```

如果 `process()` 函数中的 `vector` 有小于或大于 3 个元素，断言会失败，并生成如下类似的信息(确切的消息取决于所使用的编译器)：

```
Assertion failed: coordinate.size() == 3, file D:\test\test.cpp, line 12
```

如果想要一个自定义的错误消息，可以使用以下技巧，使用逗号运算符和一组额外的括号：

```
assert( ("A custom message...", coordinate.size() == 3) );
```

输出结果如下所示：

```
Assertion failed: ("A custom message...", coordinate.size() == 3), file D:\test\
test.cpp, line 106
```

如果在代码中，想要一个 `assert` 总是失败并给出错误消息，可以使用以下技巧：

```
assert(!"This should never happen.");
```


警告：

小心不要把使程序正确执行所需的任何代码放在断言中。例如，下面这行代码可能是自讨苦吃：

```
assert(myFunctionCall() != nullptr);
```

如果代码的发行版剥离了断言，那么对 `myFunctionCall()` 的调用也会被剥离。

31.4.4 崩溃转储

确保程序创建崩溃转储，也称为内存转储、核心转储等。崩溃转储是一个转储文件，会在应用程序崩溃时创建。它包含以下信息：在崩溃时哪个线程正在运行、所有线程的调用栈等。创建这种转储的方式与平台相关，所以应查阅平台的文档，或使用第三方库。Breakpad(<https://github.com/google/breakpad>)就是这样一个开源跨平台库，可用来写入和处理崩溃转储。

还要确保建立符号服务器和源代码服务器。符号服务器用于存储软件发布二进制版本的调试符号，这些符号在以后用于解释来自客户的崩溃转储。源代码服务器参见第 28 章“充分利用软件工程方法”，它存储源代码的所有修订。调试崩溃转储时，源代码服务器用于下载正确的源代码，以修订创建崩溃转储的软件。

分析崩溃转储的具体过程取决于平台和编译器，所以应查阅相关的文档。

就个人经验而论，崩溃转储的价值常比一千份 bug 报告更高。

31.5 调试技术

程序的调试可能会令人异常沮丧。然而，如果采用系统化的方法，则简单得多。尝试调试的第一步总是要重现 bug。根据能否重现 bug，后续采取的方法会有所不同。接下来讲解如何重现 bug，如何调试可重现的 bug，如何调试不可重现的 bug，以及如何调试退化。最后详细讲解内存错误和多线程程序的调试。最后几节展示了一个单步调试的示例。

31.5.1 重现 bug

如果可以一致地重现 bug，那么找到问题根源的过程就会简单得多。找到不可重现 bug 的根本原因是很困难的，但不是不可能。

要重现 bug，首先采用与 bug 第一次出现时类似的环境(硬件、操作系统等)和完全相同的输入来运行程序。一定要包含从程序启动时到出现 bug 时的所有输入。尝试重现 bug 的一个常见错误是只执行触发操作。这种技术可能无法重现 bug，因为 bug 可能是由整个操作序列产生的。

例如，如果请求某个网页时 Web 浏览器崩溃了，这可能是由那个特定请求的网络地址引发的内存损坏。另一方面，可能程序将所有请求都记录在一个队列中，这个队列只能容纳一百万个条目，而这个条目是第一百万零一个条目。此时，重新启动程序并发送一个请求肯定不会触发这个 bug。

有时不可能模拟导致这个 bug 的整个事件序列。也许报告这个 bug 的用户忘记自己采取了什么操作。还可能这个程序运行的时间太长，以至于无法模拟每个输入。这种情况下，尽你所能重现 bug。这需要一些猜测，可能非常费时，但此时做出的努力会节省之后调试过程的时间。下面是一些可以尝试的技巧：

- 在正确的环境中重复触发 bug 的操作，输入数量尽可能接近初始报告中的输入数量。
- 快速核查与 bug 相关的代码，也许会发现可能的根源，以指导如何重现该问题。

- 运行自动化测试，演练类似功能。自动化测试的一个好处是重现 bug。如果 bug 出现之前需要执行 24 小时的测试，那么最好让这些测试自己运行，而不是自己花 24 小时的时间等待重现 bug。
- 如果有必要的可用硬件资源，在不同的计算机上并发地运行带有细微变化的测试，这样可以节省时间。
- 运行压力测试，演练类似功能。如果程序是 Web 服务器，在处理某个请求时崩溃了，那么同时运行尽可能多的浏览器并发出这个请求。

能一致地重现 bug 时，应该尝试找出触发 bug 的最小序列。可从仅包含触发操作的最小序列开始，慢慢地扩大序列范围，以覆盖自从启动以来，直到 bug 被触发时的完整序列。这会得到重现 bug 的最简单高效的测试用例，简化寻找导致问题根源的过程，也更容易验证修复的 bug。

31.5.2 调试可重复的 bug

可一致高效地重现 bug 时，应开始在代码中找到导致 bug 的根源。此时的目标是找到触发这个问题的准确代码行。可采用两种不同的策略。

(1) **记录调试消息**：在程序中添加足够的调试消息并观察 bug 重现时的输出，应该能准确定位发生 bug 的那行代码。如果手边有调试器，通常不建议添加调试信息，因为这需要修改程序，而且这个过程可能会耗费时间。不过，如果已经根据前面的描述在程序中放置了调试信息，那么在调试模式下运行程序以重现 bug，也许可找到问题的根源。注意仅启用日志功能，有时 bug 会消失，因为启用日志功能可能会略微改变应用程序的计时。

(2) **使用调试器**：通过调试器可单步跟踪程序的执行，定点观察内存状态和变量的值。调试器往往是寻找 bug 问题根源的必备工具。当有权访问源代码时，可使用符号调试器：这种调试器可利用变量名、类名和代码中的其他符号。为使用符号调试器，必须通知编译器生成调试符号。为了解如何启用符号生成，可查看编译器文档。

本章最后的调试示例演示了这两种策略。

31.5.3 调试不可重现的 bug

修复不可重现的 bug 比修复可重现的 bug 困难得多。通常，能了解到的信息很少，必须进行大量猜测。不过，也有一些有帮助的策略：

(1) **尝试将不可重现的 bug 转换为可重现的 bug**。通过充分的猜测，通常可确定 bug 的大致位置。花一些时间尝试重现 bug。一旦有了可重现的 bug，就可以使用前面描述的技术找到 bug 的根源。

(2) **分析错误日志**。如果程序根据前面的描述带有生成错误日志的功能，那么这一点很容易实现。应该筛查这些信息，因为 bug 出现之前记录的任何错误都有可能对 bug 本身有贡献。如果幸运(或者程序写得很好)，程序会记录手头要处理的 bug 的准确原因。

(3) **获取和分析跟踪**。如果程序带有跟踪输出(例如之前描述的环形缓冲区)，那么这一点很容易实现。在发生 bug 时，可能获得一份跟踪的副本。通过这些跟踪，应该能找到代码中 bug 的正确位置。

(4) **如果有的话，检查崩溃/内存文件**。有些平台会在应用程序异常终止时生成内存转储文件。在 UNIX 和 Linux 上，这些内存转储文件称为核心文件(core file)。每个平台都提供了分析这些内存转储文件的工具。例如，这些工具可用来生成应用程序的栈跟踪信息，或查看应用程序崩溃之前内存中的内容。

(5) **检查代码**。遗憾的是，这往往是检查不可重现 bug 的根源的唯一策略。令人惊讶的是，这种策略往往奏效。检查代码时，甚至是检查自己编写的代码时，如果站在刚才发生的 bug 的视角，通常

可以找到之前忽视的错误。不建议花很长时间盯着代码，而手工跟踪代码执行路径往往可以直接找到问题所在。

(6) **使用内存观察工具**。这类工具往往会警告一些未必导致程序行为异常的内存错误，但这些问题可能是手头 bug 的根源。

(7) **提交或更新 bug 报告**。即使不能马上发现 bug 的根源，如果再次遇到问题，bug 报告也会是描述前面做出的尝试的有用记录。

(8) **如果无法找到引起 bug 的根本原因，务必添加额外的日志记录或跟踪**。这样，bug 下次出现时，将有更大的机会找到原因。

一旦找到不可重现 bug 的根源，就应该创建可重现的测试用例，并将其转移至“可重现 bug”类别。重要的是在实际修复 bug 之前重现这个 bug。否则，如何才能测试 bug 是否修复？调试不可重现 bug 的一个常见错误是在代码中修复错误的问题。不能重现 bug，也不知道是否真正修复了这个 bug，因此几个月后当这个 bug 再次出现时，没有什么可惊讶的。

31.5.4 调试退化

如果一个特性包含退化 bug，就表示所使用的特性工作正确，但因为引入了 bug 而停止工作。

检测退化的一种有用的调试技术是查看相关文件的更改日志(change log)。如果知道特性仍能工作的时间，就查看该时间以后的所有更改日志。在这些日志中可能会注意到某些值得怀疑的地方，跟踪它们可能找到错误的根源。

另一种方法可节省调试退化的大量时间，即对软件的旧版本使用二叉树搜索方法，尝试确定软件何时开始出错。如果保留了旧版本的二进制文件，就可以使用它们，或者通过源代码服务器使用旧版本的源代码。一旦知道软件何时开始出错，就查看自那时起的更改日志。这种机制只能在可以重现 bug 的情况下使用。

31.5.5 调试内存问题

最具灾难性的错误(例如应用程序崩溃)都是由内存错误造成的。很多非灾难性的错误也是由内存错误引起的。一些内存 bug 是显而易见的：如果程序试图解除对 nullptr 指针的引用，那么默认的行为是终止程序。然而，几乎每个平台都提供了响应灾难性错误并采取补救措施的功能。在这里投入的工作量取决于这类故障恢复对最终用户的重要性。例如，文本编辑器需要尽最大可能保存修改后的缓冲区(可能使用“恢复”作为名称)，而对于其他程序而言，用户可能会觉得这种默认行为是可以勉强接受的。

有些内存错误更难发现。如果越出 C++中数组的结尾，程序可能不会在这一刻直接崩溃。然而，如果该数组在栈上，那么程序可能写入另一个变量或数组，修改的值要在程序运行一段时间之后才显现出来。另外，如果该数组在堆上，可能导致堆中的内存损坏，从而在尝试动态分配或释放更多内存时产生错误。

第7章“内存管理”从编写代码时应该避免的行为的角度介绍了一些常见的内存错误。本节将从出现 bug 的代码中找出问题的角度讨论内存错误。继续阅读本节内容之前，应该确保熟悉第7章讨论的内容。

警告：

使用智能指针而不是普通指针可以避免以上大部分内存问题。

1. 内存错误的分类

为调试内存问题，应该熟悉可能发生的内存错误类型。下面介绍内存错误的分类。每种内存错误都包含一个演示错误的简短代码示例，并列出可能观察到的症状。注意，症状并不等同于 bug 本身：症状是由 bug 引起的可观察到的行为。

内存释放错误

表 31-1 总结了 5 种涉及释放内存的主要错误。

表 31-1 涉及释放内存的主要错误

错误类型	症状表现	示例
内存泄漏	随着时间的推移，进程的内存使用量增长 随着时间的推移，进程速度变慢 最终，由于内存不足，导致操作和系统调用失败	<pre>void memoryLeak() { int* p = new int[1000]; return; // BUG! Not freeing p. }</pre>
使用不匹配的分配和释放操作	通常不会立即引起程序崩溃 在某些平台上可能会导致内存损坏，可能表现为程序在一段时间后崩溃 某些不匹配也可能导致内存泄漏	<pre>void mismatchedFree() { int* p1 { (int*)malloc(sizeof(int)) }; int* p2 { new int }; int* p3 { new int[1000] }; delete p1; // BUG! Should use free delete[] p2; // BUG! Should use delete free(p3); // BUG! Should use delete[] }</pre>
多次释放内存	如果某个位置的内存有两次 delete 调用之间被另行分配使用，就会导致程序崩溃	<pre>void doubleFree() { int* p1 { new int[1000] }; delete[] p1; int* p2 { new int[1000] }; delete[] p1; // BUG! freeing p1 twice } // BUG! leaking memory of P2</pre>
释放未分配的内存	通常会导致程序崩溃	<pre>void freeUnallocated() { int* p { reinterpret_cast<int*>(10000) }; delete p; // BUG! p not a valid pointer. }</pre>
释放栈内存	从技术角度看，是释放未分配内存的特殊情况，通常会引起程序崩溃	<pre>void freeStack() { int x; int* p { &x }; delete p; // BUG! Freeing stack memory }</pre>

表 31-1 中提及的程序崩溃可能引起不同的症状，这取决于平台，例如段错误、总线错误或访问失败。

可以看出，有些内存释放错误不会立即导致程序终止。这些 bug 更微妙，会导致程序在运行一段时间之后出错。

内存访问错误

另一类的内存错误涉及实际的内存读写，如表 31-2 所示。

表 31-2 实际的内存读写错误

错误类型	症状表现	示例
访问无效内存	几乎总会导致程序立即崩溃	<pre>void accessInvalid() { int* p { reinterpret_cast<int*>(10000) }; *p = 5; // BUG! p is not a valid pointer. }</pre>
访问已释放的内存	通常不会导致程序崩溃 如果这段内存被另行分配使用，那么可能导致异常出现“奇怪”的值	<pre>void accessFreed() { int* p1 { new int }; delete p1; int* p2 { new int }; *p1 = 5; // BUG! The memory pointed to // by p1 has been freed. }</pre>
访问不同分配中的内存	不会导致程序崩溃 可能导致出现“奇怪”的、有潜在危险的值	<pre>void accessElsewhere() { int x, y[10], z; x = 0; z = 0; for (int i { 0 }; i <= 10; i++) { y[i] = 5; // BUG for i==10! element 10 // is past end of array. } }</pre>
读取未初始化的内存	不会导致程序崩溃，除非使用未初始化的值作为指针，并解除对指针的引用。即使这样，也不会总是导致程序崩溃	<pre>void readUninitialized() { int* p; cout << *p; // BUG! p is uninitialized }</pre>

内存访问错误并不总会让程序崩溃。相反，它们可能导致微妙错误，程序并不终止，而是产生错误的结果。错误的结果可以导致严重后果，例如使用计算机控制外部设备(例如机器手臂、X 光机、放射治疗仪和生命支撑系统等)时。

注意，这里讨论的内存释放错误和内存访问错误的症状是程序发行版的默认症状。调试版可能有不同的行为，在调试器中运行程序时，调试器可能在发生错误时进入代码。

2. 调试内存错误的技巧

每次运行程序时，内存相关的 bug 通常出现在略微不同的位置。这种情况通常表明堆内存损坏。堆内存损坏就像一颗定时炸弹，在试图分配、释放或使用堆内存时随时可能爆炸。所以，当遇到一个可重现但出现在略微不同的位置的 bug 时，可以怀疑是内存损坏。

如果怀疑是内存 bug，最好使用 C++ 的内存检查工具。调试器通常提供了选项，允许在运行程序时检查内存错误。例如，如果在 Microsoft Visual C++ 调试器中运行应用程序的调试版，它将捕获前面讨论的几乎所有错误类型。此外，还有一些优秀的第三方工具，例如来自 Rational Software(现在归 IBM 拥有)的 purify，以及 Linux 下的 valgrind(详见第 7 章的讨论)。Microsoft 还提供 Application Verifier(Windows 10 SDK 的一部分，developer.microsoft.com/windows/downloads/windows-10-sdk)的免费下载链接，这个工具可在 Windows 环境中使用。这是一个运行时验证工具，可帮你找到微妙的编程错误，例如此前讨论的内存错误。这些调试器和工具在工作时插入自己的内存分配和释放例程，以检查任何与动态内存有关的误用，例如释放未分配的内存、解除对未分配内存的引用以及越过数组结尾写入等。

如果手头没有可用的内存检查工具，普通的调试策略也没有任何帮助，那么可借助代码检查的方法。首先，将范围缩小至包含 bug 的部分代码；接着，一般应查看所有裸指针。如果处理的是中等或优等质量的代码，大多数指针应已经包含在智能指针中。如果遇到裸指针，应仔细查看它们的用法，因为它们可能是错误的根源。下面是需要检查的具体项目。

与对象和类相关的错误

- 验证带有动态分配内存的类具有以下这种析构函数：能准确地释放对象中分配的内存，不多也不少。
- 确保类能够通过复制构造函数和赋值运算符正确处理复制和赋值，详见第 9 章“精通类和对象”的讨论。确保移动构造函数和移动赋值运算符把源对象中的指针正确设置为 nullptr，这样其析构函数才不会释放该内存。
- 检查可疑的类型转换。如果将对象的指针从一种类型转换为另一种类型，确保转换是合法的。在可能的情况下，使用 `dynamic_cast`。

警告：

每当遇到使用普通指针处理资源所有权的情形时，强烈建议用智能指针替代普通指针，并按第 9 章讨论的“零规则”重构代码。这样，上面的项目符号列表中的第 1 类和第 2 类错误就不可能出现。

一般性内存错误

- 确保每个 new 调用都匹配一个 delete 调用，每个 new[] 调用也要匹配一个 delete[] 调用。同样，每个对 malloc、alloc 和 calloc 的调用都要匹配一个对 free 的调用。为避免多次释放内存或使用已释放的内存，建议释放内存后将指针设置为 nullptr。当然，最牢靠的办法是避免使用普通指针来处理资源的所有权，要使用智能指针。
- 检查缓冲区溢出。每次迭代访问数组或读写 C 风格的字符串时，验证没有越过数组或字符串的结尾访问内存。使用标准库容器和字符串通常可避免此类问题。
- 检查无效指针的解除引用。
- 在栈上声明指针时，确保总是在声明中初始化指针。例如，使用 `T* p {nullptr}` 或 `T* p {new T}`，但是绝不要使用 `T* p`。重申一次，要尽量使用智能指针。
- 同样，确保总在类的初始化器或构造函数中初始化指针数据成员，既可以在构造函数中分配内存，也可将指针设置为 nullptr。不厌其烦地重申一次，要尽量使用智能指针。

31.5.6 调试多线程程序

C++包含一个线程库，里面提供了多线程和线程间同步的机制。这个线程库在第 27 章讨论过。多线程的 C++ 程序很常见，因此考虑多线程程序调试时的特殊情况非常重要。多线程程序的 bug 往往因为操作系统调度中时序的不同而引起，很难重现。因此，调试多线程程序需要采用一套特殊技术。

(1) **使用调试器：**调试器很容易诊断某些多线程问题，例如死锁。出现死锁时，调试过程会进入调试器，检查不同的线程。在调试器中，可以看到哪些线程被阻塞，它们在哪行代码被阻塞。将这些信息与跟踪日志相比较，可以看出程序是如何进入死锁情形的，这足以解决死锁。

(2) **使用基于日志的调试：**调试多线程程序时，基于日志的调试有时在调试某些问题时比使用调试器更有效。在程序的临界区之前和之后，以及获得锁之前和释放锁之后添加日志语句。基于日志的调试对观察竞争条件极为有效，但添加日志语句会轻微改变运行时时序，这可能会隐藏 bug。

(3) **插入强制休眠和上下文切换：**如果一致地重现问题有困难，或者对问题发生的根源有感觉，但是想要验证根源，可以让线程睡眠特定的时间，强制执行特定的线程调度行为。<thread> 在 `std::this_thread` 名称空间中定义了可实现休眠的 `sleep_until()` 和 `sleep_for()` 函数。将睡眠时间分别指定为 `std::time_point` 或 `std::duration`，两者都是第 22 章“日期和时间工具”讨论的 `chrono` 库的一部分。在释放锁之前休眠几秒，或者在对某个条件变量发出信号前休眠几秒，或在访问共享数据之前休眠几秒，可能表现出争用条件(如果不休眠，则可能无法检测到)。如果通过这种调试技术找到了问题的根源，那么必须修复这个问题，这样在移除了这些强制休眠和上下文切换之后，代码就能正常工作。这种把这些强制休眠和上下文切换留在程序中，进而“解决问题”的方法是错误的。

(4) **核查代码：**核查线程同步代码有助于解决争用条件。不可能反复尝试已发生的情形，直到看出该情形是如何发生的。在代码注释中记下这些“证据”是无害的。另外，请同事与自己一起调试，他可能会看到你忽略的东西。

31.5.7 调试示例：文章引用

本节给出一个有 bug 的程序，展示调试和修复问题所采取的步骤。

假设你是一支网页编写团队的成员，这个团队编写的网页允许用户搜索引用了某篇文章的文章。这类服务对于试图找出类似文章的作者来说非常有用。一旦他们找到一篇表示相关文章的论文，就可以查找所有引用了这篇论文的文章，从而找到其他相关的文章。

在这个项目中，你负责从文本文件中读取原始引用数据的代码。为简单起见，假设每篇论文中的引用信息都可以在论文本身的文件中找到。此外，假设每个文件的第一行都包含这篇论文的作者、标题和出版信息；第二行总是为空；所有后续的行都包含这篇文章引用的论文(每行一篇)。下面是计算机科学中一篇最重要论文的示例文件：

Alan Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem",
Proceedings of the London Mathematical Society, Series 2, Vol.42 (1936-37), 230-265.

Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme,
I", Monatshefte Math. Phys., 38 (1931), 173-198.

Alonzo Church. "An unsolvable problem of elementary number theory", American J. of Math.,
58 (1936), 345-363.

Alonzo Church. "A note on the Entscheidungsproblem", J. of Symbolic Logic, 1 (1936), 40-41.
E.W. Hobson, "Theory of functions of a real variable (2nd ed., 1921)", 87-88.

1. ArticleCitations 类的存在 bug 的实现

在设计程序的结构时，决定编写 ArticleCitations 类，这个类负责读取文件和保存信息。这个类将第一行中的文章信息保存在一个字符串中，将引用信息保存在 C 风格的字符串数组中。

警告：

使用 C 风格数组的设计决定十分糟糕。最好选择一个标准库容器来保存引文信息。这里这么做是为了演示内存问题。这个实现还有其他显而易见的问题，例如使用 int 而不是 size_t，不使用第 9 章讨论的“复制和交换”惯用语法来实现赋值运算符。然而，为了演示存在 bug 的应用程序，这么做是恰当的。

在 article_citations 模块中定义 ArticleCitations 类：

```
export class ArticleCitations
{
    public:
        ArticleCitations(std::string_view filename);
        virtual ~ArticleCitations();
        ArticleCitations(const ArticleCitations& src);
        ArticleCitations& operator=(const ArticleCitations& rhs);

        const std::string& getArticle() const;
        int getNumCitations() const;
        const std::string& getCitation(int i) const;
    private:
        void readFile(std::string_view filename);
        void copy(const ArticleCitations& src);

        std::string m_article;
        std::string* m_citations;
        int m_numCitations;
};
```

实现代码如下所示。这个程序存在 bug！不要把这个程序用作参考实现或参考模式。

```
ArticleCitations::ArticleCitations(string_view filename)
{
    // All we have to do is read the file.
    readFile(filename);
}

ArticleCitations::ArticleCitations(const ArticleCitations& src)
{
    copy(src);
}

ArticleCitations& ArticleCitations::operator=(const ArticleCitations& rhs)
{
    // Check for self-assignment.
    if (this == &rhs) {
        return *this;
    }
    // Free the old memory.
    delete [] m_citations;
```



```

        // Copy the data.
        copy(rhs);
        return *this;
    }

void ArticleCitations::copy(const ArticleCitations& src)
{
    // Copy the article name, author, etc.
    m_article = src.m_article;
    // Copy the number of citations.
    m_numCitations = src.m_numCitations;
    // Allocate an array of the correct size.
    m_citations = new string[m_numCitations];
    // Copy each element of the array.
    for (int i { 0 }; i < m_numCitations; i++) {
        m_citations[i] = src.m_citations[i];
    }
}

ArticleCitations::~ArticleCitations()
{
    delete [] m_citations;
}

void ArticleCitations::readFile(string_view filename)
{
    // Open the file and check for failure.
    ifstream inputFile { filename.data() };
    if (inputFile.fail()) {
        throw invalid_argument { "Unable to open file" };
    }
    // Read the article author, title, etc. line.
    getline(inputFile, m_article);

    // Skip the whitespace before the citations start.
    inputFile >> ws;

    int count { 0 };
    // Save the current position so we can return to it.
    streampos citationsStart { inputFile.tellg() };
    // First count the number of citations.
    while (!inputFile.eof()) {
        // Skip whitespace before the next entry.
        inputFile >> ws;
        string temp;
        getline(inputFile, temp);
        if (!temp.empty()) {
            ++count;
        }
    }

    if (count != 0) {
        // Allocate an array of strings to store the citations.
        m_citations = new string[count];

        m_numCitations = count;
    }
}

```

```

        // Seek back to the start of the citations.
        inputFile.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < m_numCitations; ++count) {
            string temp;
            getline(inputFile, temp);
            if (!temp.empty()) {
                m_citations[count] = temp;
            }
        }
    } else {
        m_numCitations = -1;
    }
}

const string& ArticleCitations::getArticle() const { return m_article; }

int ArticleCitations::getNumCitations() const { return m_numCitations; }

const string& ArticleCitations::getCitation(int i) const { return m_citations[i]; }

```

2. 测试 ArticleCitations 类

下面的程序要求用户输入一个文件名，通过这个文件名构造 ArticleCitations 实例，然后将这个实例通过值传入 processCitations() 函数，这个函数打印出所有信息。

```

void processCitations(ArticleCitations cit)
{
    cout << cit.getArticle() << endl;
    for (int i { 0 }; i < cit.getNumCitations(); i++) {
        cout << cit.getCitation(i) << endl;
    }
}

int main()
{
    while (true) {
        cout << "Enter a file name (\\"STOP\\" to stop): ";
        string filename;
        cin >> filename;
        if (filename == "STOP") { break; }

        ArticleCitations cit { filename };
        processCitations(cit);
    }
}

```

现在决定通过 Alan Turing 的例子(保存在 paper1.txt 文件中)测试这个程序。输出结果如下：

```

Enter a file name ("STOP" to stop): paper1.txt
Alan Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem",
Proceedings of the London Mathematical Society, Series 2, Vol.42 (1936-37), 230-265.
[ 4 empty lines omitted for brevity ]
Enter a file name ("STOP" to stop): STOP

```

这看起来不正确。应该打印出 4 条引文信息，而不是 4 个空行。

基于消息的调试

对于这个 bug，尝试基于消息的调试；由于这是一个控制台示例，因此将消息打印至 `cout`。在这个例子中，合理的做法是首先查看从文件中读取引文信息的函数。如果这个函数工作不正常，那么很明显这个对象不存在引文信息。可对 `readFile()` 做如下修改：

```
void ArticleCitations::readFile(string_view filename)
{
    // Code omitted for brevity.

    // First count the number of citations.
    cout << "readFile(): counting number of citations" << endl;
    while (!inputFile.eof()) {
        // Skip whitespace before the next entry.
        inputFile >> ws;
        string temp;
        getline(inputFile, temp);
        if (!temp.empty()) {
            cout << "Citation " << count << ": " << temp << endl;
            ++count;
        }
    }

    cout << "Found " << count << " citations" << endl;
    cout << "readFile(): reading citations" << endl;
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        m_citations = new string[count];
        m_numCitations = count;
        // Seek back to the start of the citations.
        inputFile.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < m_numCitations; ++count) {
            string temp;
            getline(inputFile, temp);
            if (!temp.empty()) {
                cout << temp << endl;
                m_citations[count] = temp;
            }
        }
    } else {
        m_numCitations = -1;
    }
    cout << "readFile(): finished" << endl;
}
```

在这个程序上运行相同的测试，可看到以下输出：

```
Enter a file name ("STOP" to stop): paper1.txt
readFile(): counting number of citations
Citation 0: Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und
verwandter Systeme, I", Monatshefte Math. Phys., 38 (1931), 173-198.
Citation 1: Alonzo Church. "An unsolvable problem of elementary number theory",
American J. of Math., 58 (1936), 345-363.
Citation 2: Alonzo Church. "A note on the Entscheidungsproblem", J. of Symbolic
Logic, 1 (1936), 40-41.
Citation 3: E.W. Hobson, "Theory of functions of a real variable (2nd ed.,
```

```

1921)", 87-88.
Found 4 citations
readFile(): reading citations
readFile(): finished
Alan Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem",
Proceedings of the London Mathematical Society, Series 2, Vol.42 (1936-37), 230-265.
[ 4 empty lines omitted for brevity ]
Enter a file name ("STOP" to stop): STOP

```

从输出结果可以看出, 为计算文件中的引文数目, 程序第一次从文件中读取引文信息, 这些信息读取正确。然而, 第二次读取不正确。在 `readFile(): reading citations` 和 `readFile(): finished` 之间什么都没有输出。为什么? 深入钻研这个问题的方法之一是添加一些调试代码, 检查每次尝试读取一条引文信息后文件流的状态。

```

void printStreamState(const istream& inputStream)
{
    if (inputStream.good()) { cout << "stream state is good" << endl; }
    if (inputStream.bad()) { cout << "stream state is bad" << endl; }
    if (inputStream.fail()) { cout << "stream state is fail" << endl; }
    if (inputStream.eof()) { cout << "stream state is eof" << endl; }
}

void ArticleCitations::readFile(string_view filename)
{
    // Code omitted for brevity.

    // First count the number of citations.
    cout << "readFile(): counting number of citations" << endl;
    while (!inputFile.eof()) {
        // Skip whitespace before the next entry.
        inputFile >> ws;
        printStreamState(inputFile);
        string temp;
        getline(inputFile, temp);
        printStreamState(inputFile);
        if (!temp.empty()) {
            cout << "Citation " << count << ": " << temp << endl;
            ++count;
        }
    }

    cout << "Found " << count << " citations" << endl;
    cout << "readFile(): reading citations" << endl;
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        m_citations = new string[count];
        m_numCitations = count;
        // Seek back to the start of the citations.
        inputFile.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < m_numCitations; ++count) {
            string temp;
            getline(inputFile, temp);
            printStreamState(inputFile);
            if (!temp.empty()) {
                cout << temp << endl;
            }
        }
    }
}

```

```

        m_citations[count] = temp;
    }
} else {
    m_numCitations = -1;
}
cout << "readFile(): finished" << endl;
}

```

再次运行这个程序时，可看到一些有趣的信息：

```

Enter a file name ("STOP" to stop): paper1.txt
readFile(): counting number of citations
stream state is good
stream state is good
Citation 0: Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und
verwandter Systeme, I", Monatshefte Math. Phys., 38 (1931), 173-198.
stream state is good
stream state is good
Citation 1: Alonzo Church. "An unsolvable problem of elementary number theory",
American J. of Math., 58 (1936), 345-363.
stream state is good
stream state is good
Citation 2: Alonzo Church. "A note on the Entscheidungsproblem", J. of Symbolic
Logic, 1 (1936), 40-41.
stream state is good
stream state is good
Citation 3: E.W. Hobson, "Theory of functions of a real variable (2nd ed.,
1921)", 87-88.
stream state is eof
stream state is fail
stream state is eof
Found 4 citations
readFile(): reading citations
stream state is fail
stream state is fail
stream state is fail
stream state is fail
readFile(): finished

Alan Turing, "On Computable Numbers, with an Application to the
Entscheidungsproblem", Proceedings of the London Mathematical Society, Series 2,
Vol.42 (1936-37), 230-265.
[ 4 empty lines omitted for brevity ]
Enter a file name ("STOP" to stop): STOP

```

第一次读取文件时，在读入最后一引文信息前，流的状态看上去都是好的。由于 `paper1.txt` 文件的最后一行是空的，在读取最后一引文信息后，多执行一次 `while` 循环。在最后的循环中，`inputFile >> ws` 读取最后一行的空白，导致流状态变为 `eof`。此后，代码仍尝试使用 `getline()` 读取行，导致流状态变成 `fail` 和 `eof`。这是符合预期的行为。意料之外的行为是当第二次读取引文信息时，每次尝试读取一条引文信息之后流的状态都是 `fail`。初看上去不合理：这段代码在第二次读取文件之前，通过 `seekg()` 将文件指针返回引文的头部。

根据第 13 章的介绍，在显式清除流的错误状态前，流会保留这些状态。`seekg()` 不会自动清除 `fail` 状态。处于错误状态时，流无法正确读取数据，这也解释了第二次试图读取引文信息后，流的状态也

是 fail 的原因。再仔细看一下方法代码，发现代码在到达文件结尾后没有调用 istream 的 clear() 方法。如果修改这个方法，添加 clear() 调用，那么代码能正确地读取引文。

下面是改正后的 readFile() 方法，其中去掉了调试用的 cout 语句：

```
void ArticleCitations::readFile(string_view filename)
{
    // Code omitted for brevity.

    if (count != 0) {
        // Allocate an array of strings to store the citations.
        m_citations = new string[count];
        m_numCitations = count;
        // Clear the stream state.
        inputFile.clear();
        // Seek back to the start of the citations.
        inputFile.seekg(citationsStart);
        // Read each citation and store it in the new array.
        for (count = 0; count < m_numCitations; ++count) {
            string temp;
            getline(inputFile, temp);
            if (!temp.empty()) {
                m_citations[count] = temp;
            }
        }
    } else {
        m_numCitations = -1;
    }
}
```

在 paper1.txt 上运行相同的测试，现在可看到正确的 4 条引文。

在 Linux 上使用 GDB 调试器

现在 ArticleCitations 类似乎在引文文件上工作得很好，下面进一步测试一些特殊情况，从一个不带有引文信息的文件开始。这个文件的内容如下，保存在 paper2.txt 文件中：

```
Author with no citations
```

尝试针对这个文件运行程序时，可能崩溃(具体取决于 Linux 和编译器的版本)，如下所示：

```
Enter a file name ("STOP" to stop): paper2.txt
terminate called after throwing an instance of 'std::bad_alloc'
  what(): std::bad_alloc
Aborted (core dumped)
```

消息 core dumped 意味着程序崩溃了。这次要试用一下调试器。GNU 调试器是在 UNIX 和 Linux 平台上广泛使用的调试器。首先，编译程序时必须带有调试信息(在 g++ 上使用 -g)。此后，可在 GDB 下启动程序。下面是通过调试器找到这个问题根源所在的示例会话。这个例子假设编译的可执行文件名为 buggyprogram。需要输入的文本用粗体显示：

```
> gdb buggyprogram
[ Start-up messages omitted for brevity ]
Reading symbols from /home/marc/c++/gdb/buggyprogram...done.
(gdb) run
Starting program: buggyprogram
Enter a file name ("STOP" to stop): paper2.txt
```

```

terminate called after throwing an instance of 'std::bad_alloc'
  what(): std::bad_alloc
Program received signal SIGABRT, Aborted.
0x00007ffff7535c39 in raise () from /lib64/libc.so.6
(gdb)

```

当程序崩溃时，调试器中断执行，并允许查看程序在中断时的状态。`backtrace` 或 `bt` 命令显示当前的栈跟踪。栈跟踪在顶部显示最后一个操作，即编号为#0的帧。

```

(gdb) bt
#0 0x00007ffff7535c39 in raise () from /lib64/libc.so.6
#1 0x00007ffff7537348 in abort () from /lib64/libc.so.6
#2 0x00007ffff7b35f85 in __gnu_cxx::__verbose_terminate_handler() () from
/lib64/libstdc++.so.6
#3 0x00007ffff7b33ee6 in ?? () from /lib64/libstdc++.so.6
#4 0x00007ffff7b33f13 in std::terminate() () from /lib64/libstdc++.so.6
#5 0x00007ffff7b3413f in __cxa_throw () from /lib64/libstdc++.so.6
#6 0x00007ffff7b346cd in operator new(unsigned long) () from /lib64/libstdc++.so.6
#7 0x00007ffff7b34769 in operator new[](unsigned long) () from /lib64/libstdc++.so.6
#8 0x0000000004016ea in ArticleCitations::copy (this=0x7ffffffffffe090, src=...)
    at ArticleCitations.cpp:40
#9 0x0000000004015b5 in ArticleCitations::ArticleCitations (this=0x7ffffffffffe090,
src=...)
    at ArticleCitations.cpp:16
#10 0x000000000401d0c in main () at ArticleCitationsTest.cpp:20

```

得到这样的栈跟踪，应该尝试从栈顶部开始寻找属于自己代码的第一个栈帧。在这个例子中，这是栈帧#8。从这个栈帧中，可看出 `ArticleCitations` 的 `copy()` 方法中似乎存在某类问题。调用这个方法的原因是 `main()` 函数调用 `processCitations()` 时通过值传入参数，这会触发对拷贝构造函数的调用，而拷贝构造函数会调用 `copy()`。当然，在生产代码中应该传入 `const` 引用，不过这个存在 `bug` 的程序示例使用了按值传递。可通过 `frame` 命令让调试器切换到栈帧#8，`frame` 命令要求提供一个表示向上跳跃的帧索引作为参数：

```

(gdb) frame 8
#8 0x0000000004016ea in ArticleCitations::copy (this=0x7ffffffffffe090, src=...) at
ArticleCitations.cpp:40
40   m_citations = new string[m_numCitations];

```

这个输出显示是下面这行代码导致了问题：

```
M_citations = new string[m_numCitations];
```

现在，通过 `list` 命令显示在当前栈帧中出问题的那一行代码周围的代码：

```

(gdb) list
35   // Copy the article name, author, etc.
36   m_article = src.m_article;
37   // Copy the number of citations.
38   m_numCitations = src.m_numCitations;
39   // Allocate an array of the correct size.
40   m_citations = new string[m_numCitations];
41   // Copy each element of the array.
42   for (int i { 0 }; i < m_numCitations; i++) {
43       m_citations[i] = src.m_citations[i];
44   }

```

在 GDB 中, 可通过 `print` 命令查看当前作用域内可用的值。为找到问题根源, 可尝试打印一些变量。错误发生在 `copy()` 方法内, 因此检查 `src` 参数的值是一个良好开端:

```
(gdb) print src
$1 = (const ArticleCitations &) @0x7fffffff060: {
  _vptr.ArticleCitations = 0x401fb0 <vtable for ArticleCitations+16>,
  m_article = "Author with no citations", m_citations = 0x7fffffff080,
  m_numCitations = -1}
```

问题就在这里。这篇文章不应该有任何引用。为什么把 `m_numCitations` 设置为 -1? 在没有引文的情况下再看一下 `readFile()` 中的代码。这种情况下, `m_numCitations` 被错误地设置为 -1。修改它很容易, 只需要在没有引文的情况下将 `m_numCitations` 初始化为 0, 而不是设置为 -1。另一个问题是, `readFile()` 可在同一个 `ArticleCitations` 对象上调用多次, 因此还需要释放以前分配的 `m_citations` 数组。

下面是修改后的代码:

```
void ArticleCitations::readFile(string_view filename)
{
    // Code omitted for brevity.

    delete [] m_citations; // Free previously allocated citations.
    m_citations = nullptr;
    m_numCitations = 0;
    if (count != 0) {
        // Allocate an array of strings to store the citations.
        m_citations = new string[count];
        m_numCitations = count;
        // Clear the stream state.
        inputFile.clear();
        // Seek back to the start of the citations.
        inputFile.seekg(citationsStart);

        // Code omitted for brevity.
    }
}
```

这个例子说明, `bug` 并不总是会立即显现出来, 往往需要通过调试器并且有一些耐心才能显现出来。

使用 Visual C++ 2019 调试器

接下来描述与前面相同的调试过程, 但使用 Visual C++ 2019 调试器而不是 GDB。

首先, 需要创建一个项目。单击 Visual Studio 2019 欢迎屏幕中的 `Create A New Project` 按钮, 或选择 `File | New | Project`。在 `Create A New Project` 对话框中, 搜索带有 `C++`、`Windows` 和 `Console` 标签的控制台应用程序项目模板, 然后单击 `Next` 按钮。输入 `ArticleCitations` 作为项目名字, 选择一个保存项目的目录, 然后单击 `Create` 按钮。项目创建后, 可以在 `Solution Explorer` 中看到项目文件列表。如果此停靠窗口不可见, 可选择 `View | Solution Explorer`。该项目会包含一个名为 `ArticleCitations.cpp` 的文件。在 `Solution Explorer` 中选择该文件并删除它, 因为你会添加自己的文件。

现在让我们添加文件。在 `Solution Explorer` 中右击 `ArticleCitations` 项目, 并选择 `Add | Existing Item`。将源码存档中的 `06_ArticleCitations\06_VisualStudio` 目录中的所有文件添加到项目中。`Solution Explorer` 应该类似于图 31-1。

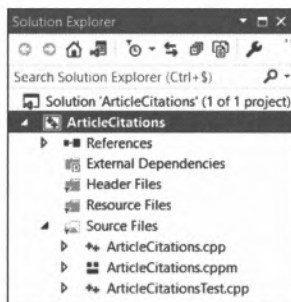


图 31-1 Solution Explorer 窗口

这个示例使用了 Visual C++2019 中未默认启用的 C++20 特性。要启用它们，在 Solution Explorer 窗口中右击 ArticleCitations 项目，然后单击 Properties。在 Properties 窗口中，选择 Configuration Properties | C/C++ | Language，将 C++语言标准选项设置为 ISO C++20 标准或 Preview-Features From The Latest C++ Working Draft，看哪个在你的 Visual C++版本中可用。（目前还没有 ISO C++20 标准选项，但它会出现在 Visual C++未来的更新中。）

确保配置设置为 Debug 而不是 Release，然后编译整个程序。选择 Build | Build Solution。然后将 paper1.txt 和 paper2.txt 测试文件复制到 ArticleCitations 项目目录中，该目录包含 ArticleCitations.vcxproj 文件。

单击 Debug | Start Debugging 运行程序，并首先指定 paper1.txt 文件来测试程序。它应该正确读取文件并将结果输出到控制台。然后测试 paper2.txt。调试器用类似于图 31-2 所示的消息中断执行。

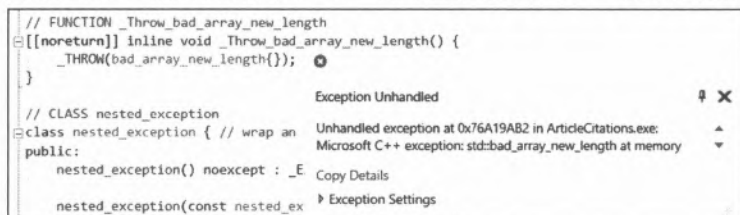


图 31-2 中断执行的消息

显示抛出了一个 std::bad_array_new_length 异常。但是，显示的代码不是你编写的代码。要找出导致异常的代码行，可以使用栈窗口(Debug | Windows | Call Stack)。在调用栈中，需要找到你所编写代码的第一行。如图 31-3 所示。

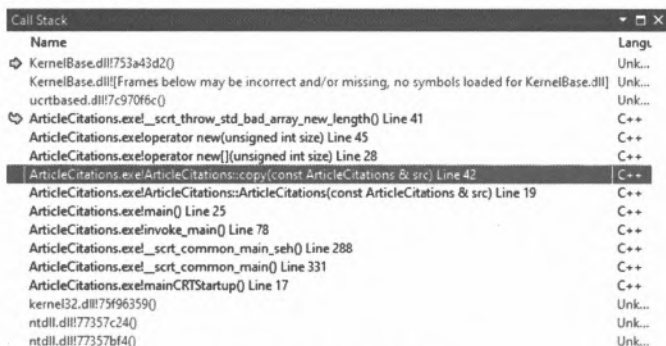


图 31-3 找到代码的第一行

就像使用 GDB 一样，可以看到问题出现在 `copy()` 中。可以双击调用栈窗口中的这一行，来跳转到代码中的正确位置。

可以通过将鼠标悬停在变量的名称上来检查变量。如果把鼠标悬停在 `src` 上，会注意到 `m_numCitations` 是 -1。原因和修复方法与 GDB 示例中的相同。

也可以使用 Debug | Windows | Autos 窗口，而不是将鼠标悬停在变量上检查它们的值，该窗口显示了变量列表。图 31-4 显示了这个列表，展开 `src` 变量来显示其数据成员。在这个窗口中，还可以看到 `m_numCitations` 是 -1。

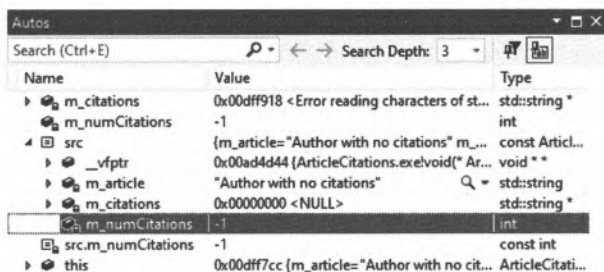


图 31-4 变量列表

31.5.8 从 ArticleCitations 示例中总结出的教训

你可能会看不上这个例子，觉得这个例子太小了，不能代表真正的调试。尽管这段有 bug 的代码并不长，但你编写的很多类并不会太多，即使是在庞大的项目中也是如此。试想一下，假设将这个例子整合到项目的其他部分之前，没有完整地测试这个例子会怎么样。如果这些错误后来出现了，工程师将不得不耗费更多时间将问题的范围缩小，才能进行这里展示的调试过程。此外，这个例子展示的技术适用于所有类型的调试，包括大规模调试和小规模调试。

31.6 本章小结

本章最重要的概念是调试的基本定律：在编写代码时注意避免 bug，但要在代码中为 bug 做好规划。编程的现实情况是一定会出现 bug。如果为程序准备好了错误日志、调试跟踪和断言，那么实际调试过程会容易得多。

本章还介绍了用于调试 bug 的具体方法。实际调试时，最重要的规则是要重现问题。然后，可使用符号调试器、基于日志的调试追查问题根源。内存错误的调试特别困难，而且占据传统 C++ 代码中 bug 的绝大部分。本章描述了不同类别的内存 bug 及其症状，并展示了在程序中调试错误的例子。

调试技术很难掌握。大量实践调试技术，才能将 C++ 技能提高到专业级别。

31.7 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

练习 31-1 调试的基本定律是什么？

练习 31-2 你能发现下面这段代码有什么问题吗？

```
import <iostream>;
using namespace std;

int* getData(int value) { return new int { value * 2 }; }

int main()
{
    int* data { getData(21) };
    cout << *data << endl;

    data = getData(42);
    cout << *data << endl;
}
```

练习 31-3 给定以下代码片段：

```
import <iostream>;
using namespace std;

int sum(int* values, int count)
{
    int total { 0 };
    for (int i { 0 }; i <= count; ++i) { total += values[i]; }
    return total;
}

int main()
{
    int values[] { 1, 2, 3 };
    int total { sum(values, size(values)) };
    cout << total << endl;
}
```

简单地计算一组值的和。对于值 1、2 和 3，期望值是 6。但是当在我的机器上运行代码的 Debug 版本时，结果是-858993454。怎么回事呢？使用符号调试器及单步执行模式来查明错误结果的根本原因。请参阅调试器的文档，了解如何单步调试代码的每一行。

练习 31-4 (高级)修改本章前面的 `start-time` 调试模式示例，使用 C++20 的 `std::source_location` 类(在第 14 章讨论)，来摆脱旧的 `log()` 宏。这比听起来要棘手。问题是 `Logger::log()` 是个可变参数函数模板，因此不能在可变参数包之后添加一个命名的 `source_location` 参数。窍门是使用一个辅助类，例如 `Log`。构造函数接受可变参数包和 `source_location`，并将工作转发给 `Logger::log()`。窍门的最后一部分是下面的推导，参见第 12 章“用模板编写泛型代码”：

```
template <typename... Ts>
log(Ts&&...) -> log<Ts...>;
```

第32章

使用设计技术和框架

本章内容

- 简述常用但包含容易忘记的语法的 C++ 语言功能
- RAII 的含义以及为什么它是一个强大的概念
- 双分派(double dispatch)技术的含义和用法
- 如何使用混入类
- 框架的含义
- MVC 范型

本书的一个重要主题是使用可重用的技术和模式。作为一名编程人员，每天都会反复面对一些类似的问题。利用各种方法，可通过为给定问题应用恰当的技术来节省时间。

在 C++ 中，设计技术是解决特定问题的标准方式。设计技术通常用于克服一些令人感到不快的功能或语言缺陷。其他时候，设计技术就是一段代码，可用在不同程序中以解决常见的 C++ 问题。

本章重点讲述设计技术，这是 C++ 惯用技术，它们未必是 C++ 固有的部分，但使用频繁。本章介绍 C++ 中包含容易忘记的语法的常用功能。大多数材料都在本书前面介绍过，但当忘记语法时，可将本章作为有用的参考工具。涵盖的主题如下：

- 从头创建类
- 通过派生方式扩展类
- 写 lambda 表达式
- 实现“复制和交换”惯用语法
- 抛出和捕获异常
- 写入文件
- 读取文件
- 定义模板类
- 约束类和函数模板参数

本章重点介绍 C++ 语言功能中的高级技术。这些技术是完成日常编程任务的较好方式，包含的主题如下：

- RAII(Resource Acquisition Is Initialization, 资源获得即初始化)

- 双分派(double dispatch)技术
- 混入类

最后简要介绍框架，这是一种编码技术，可极大地简化大型应用程序的开发。

32.1 容易忘记的语法

第1章比较了C标准和C++标准的大小。C程序员往往能记住整个C语言的语法；关键字不多，语言功能也少，行为都经过良好定义。但对于C++而言，情况并非如此。即使是C++专家也需要不时参阅资料。因此，本节列举一个编码技术示例，这个示例可用于几乎所有的C++程序。如果记着概念却忘了语法，可阅读以下内容进行复习。

32.1.1 编写类

不要忘了开头部分。下面是一个在C++20模块接口文件中定义的Simple类定义：

```
export module simple;

// A simple class that illustrates class definition syntax.
export class Simple
{
    public:
        Simple(); // Constructor
        virtual ~Simple() = default; // Defaulted virtual destructor

        // Disallow assignment and pass-by-value.
        Simple(const Simple& src) = delete;
        Simple& operator=(const Simple& rhs) = delete;

        // Explicitly default move constructor and move assignment operator.
        Simple(Simple&& src) noexcept = default;
        Simple& operator=(Simple&& rhs) noexcept = default;

        virtual void publicMethod(); // Public method
        int m_publicInteger; // Public data member

    protected:
        virtual void protectedMethod(); // Protected method
        int m_protectedInteger { 41 }; // Protected data member

    private:
        virtual void privateMethod(); // Private method
        int m_privateInteger { 42 }; // Private data member
        static const int Constant { 2 }; // Private constant
        static inline int ms_staticInt { 3 }; // Private static data member
};
```

注意：

这个类定义显示了一些可能但不推荐的编写方式。但在你自己的类定义中，应尽量避免使用public或protected数据成员。类应该封装它的数据，因此应使用private数据成员，并提供公共或受保护的getter和setter方法。

如第 10 章所述，通常，至少要将析构函数设置为 `virtual`，因为其他人可能想从这个类派生新类。也允许保留析构函数为非 `virtual`，但这只限于将类标记为 `final`，以防止其他类从其派生的情况。如果只想将析构函数设置为 `virtual`，但不需要析构函数中的任何代码，则可显式地设置为 `default`，如 `Simple` 类示例所示。

这个示例也说明，可显式地将特殊成员函数设置为 `delete` 或 `default`。将拷贝构造函数和拷贝赋值运算符设置为 `delete`，以防止赋值和按值传递，而将移动构造函数和移动赋值运算符显式设置为 `default`。

接下来的模块实现文件如下：

```
module simple;

Simple::Simple() : m_publicInteger { 40 }
{
    // Implementation of constructor
}

void Simple::publicMethod() { /* Implementation of public method */ }
void Simple::protectedMethod() { /* Implementation of protected method */ }
void Simple::privateMethod() { /* Implementation of private method */ }
```

注意：

如下一节所示，类方法定义也可以直接出现在模块接口文件中。你不需要将类拆分为模块接口文件和模块实现文件。

第 8 章“熟悉类和对象”和第 9 章“精通类和对象”提供了编写类的所有细节。

32.1.2 派生类

要从现有的类派生，可声明一个新类，这个类是另一个类的扩展类。下面是 `DerivedSimple` 类的定义，`DerivedSimple` 从 `Simple` 派生而来，定义在 `derived_simple` 模块中。

```
export module derived_simple;

export import simple;

// A class derived from the Simple class.
export class DerivedSimple : public Simple
{
    public:
    DerivedSimple() : Simple{} // Constructor
    { /* Implementation of constructor */ }

    void publicMethod() override // Overridden method
    {
        // Implementation of overridden method
        Simple::publicMethod(); // You can access base class implementations.
    }

    virtual void anotherMethod() // New method
    { /* Implementation of new method */ }
};
```

可参阅第 10 章以了解继承技术的详情。

32.1.3 编写 lambda 表达式

lambda 表达式允许编写小型匿名内联函数。它们与 C++ 标准库算法结合起来尤其强大。下面的代码片段展示了一个示例。它使用 `count_if()` 算法和 lambda 表达式来计算 `vector` 中偶数的数量。此外，lambda 表达式通过引用从闭包范围捕获 `callCount` 变量，来跟踪它被调用的次数。

```
vector values { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int callCount { 0 };
auto count { count_if(begin(values), end(values),
    [&callCount](int value) {
        ++callCount;
        return value % 2 == 0;
    })
};
cout << format("There are {} even elements in the vector.", count) << endl;
cout << format("Lambda was called {} times.", callCount) << endl;
```

第 19 章讨论了 lambda 表达式的细节。

32.1.4 使用“复制和交换”惯用语

第 9 章详细讨论了“复制和交换”惯用语。这种惯用语可在对象上实现一些可能抛出异常的操作，提供强大的异常安全保证，即“要么成功，要么什么都不做”。

只需要创建对象的一个副本，修改这个副本(可以是复杂算法，可能抛出异常)。最后，当不再抛出异常时，使用不抛出异常的 `swap()` 将这个副本与原始对象进行交换。赋值运算符是一个可使用“复制和交换”惯用语的操作示例。赋值运算符首先制作原始对象的一个本地副本，此后仅使用不抛出异常的 `swap()` 实现，将这个副本与当前对象进行交换。

下面是用于拷贝赋值运算符的“复制和交换”惯用语的一个简单示例。该类定义了一个拷贝构造函数、一个拷贝赋值运算符以及一个标记为 `noexcept` 的 `swap()` 方法。

```
export module copy_and_swap;

export class CopyAndSwap
{
public:
    CopyAndSwap() = default;
    virtual ~CopyAndSwap(); // Virtual destructor

    CopyAndSwap(const CopyAndSwap& src); // Copy constructor
    CopyAndSwap& operator=(const CopyAndSwap& rhs); // Copy assignment operator

    void swap(CopyAndSwap& other) noexcept; // noexcept swap() method

private:
    // Private data members...
};

// Standalone noexcept swap() function
export void swap(CopyAndSwap& first, CopyAndSwap& second) noexcept;
```

实现代码如下所示:

```
CopyAndSwap::~CopyAndSwap() { /* Implementation of destructor. */ }

CopyAndSwap::CopyAndSwap(const CopyAndSwap& src)
{
    // This copy constructor can first delegate to a non-copy constructor
    // if any resource allocations have to be done. See the Spreadsheet
    // implementation in Chapter 9 for an example.
    // Make a copy of all data members...
}

void swap(CopyAndSwap& first, CopyAndSwap& second) noexcept
{
    first.swap(second);
}

void CopyAndSwap::swap(CopyAndSwap& other) noexcept
{
    using std::swap; // Requires <utility>
    // Swap each data member, for example:
    // swap(m_data, other.m_data);
}

CopyAndSwap& CopyAndSwap::operator=(const CopyAndSwap& rhs)
{
    // Copy-and-swap idiom.
    auto copy { rhs }; // Do all the work in a temporary instance.
    swap(copy);        // Commit the work with only non-throwing operations.
    return *this;
}
```

详情参阅第 9 章。

32.1.5 抛出和捕获异常

如果所在的团队不使用异常(这么做是不对的!), 或习惯使用 Java 样式的异常, 那么 C++语法可能已从你的记忆中消失。下面将帮助你复习, 你将使用内置的异常类 `std::runtime_error`。在大多数大型程序中, 你将编写自己的异常类:

```
import <stdexcept>;
import <iostream>;
using namespace std;
void throwIf(bool throwIt)
{
    if (throwIt) {
        throw runtime_error { "Here's my exception" };
    }
}

int main()
{
    try {
        throwIf(false); // Doesn't throw.
        throwIf(true);  // Throws.
    } catch (const runtime_error& e) {
        cerr << "Caught exception: " << e.what() << endl;
    }
}
```



```
        return 1;
    }
}
```

第 14 章“错误处理”已详细讨论异常。

32.1.6 写入文件

以下程序向文件输出消息，此后重新打开文件，并追加另一条消息。详见第 13 章“C++ I/O 揭秘”。

```
import <iostream>;
import <fstream>;

using namespace std;

int main()
{
    ofstream outputFile { "FileWrite.out" };
    if (outputFile.fail()) {
        cerr << "Unable to open file for writing." << endl;
        return 1;
    }
    outputFile << "Hello!" << endl;
    outputFile.close();

    ofstream appendFile { "FileWrite.out", ios_base::app };
    if (appendFile.fail()) {
        cerr << "Unable to open file for appending." << endl;
        return 2;
    }
    appendFile << "World!" << endl;
}
```

32.1.7 读取文件

有关文件输入的信息，详见第 13 章。下面的简单示例程序可帮助你了解文件读取的基本方式。它读取前一节中程序写入的文件，并每次输出一个标记：

```
import <iostream>;
import <fstream>;
import <string>;

using namespace std;

int main()
{
    ifstream inputFile { "FileWrite.out" };
    if (inputFile.fail()) {
        cerr << "Unable to open file for reading." << endl;
        return 1;
    }

    string nextToken;
    while (inputFile >> nextToken) {
        cout << "Token: " << nextToken << endl;
    }
}
```

```

    }
}

```

32.1.8 写入类模板

模板语法可能会令人困惑。模板中最容易忘记的部分是：使用类模板的代码需要能够看到类模板的定义和方法实现。函数模板同样如此。实现这一点的一种技术是将类方法实现直接放在包含类模板定义的接口文件中。下面的示例对此进行了展示，实现了一个类模板，封装了对对象的引用并且包含了 `getter`。下面是模块接口文件：

```

export module simple_wrapper;

export template <typename T>
class SimpleWrapper
{
public:
    SimpleWrapper(T& object) : m_object { object } { }
    T& get() { return m_object; }
private:
    T& m_object;
};

```

该代码可以测试如下：

```

import simple_wrapper;
import <iostream>;
import <string>;

using namespace std;

int main()
{
    // Try wrapping an integer.
    int i { 7 };
    SimpleWrapper intWrapper { i }; // Using CTAD.
    // Or without class template argument deduction (CTAD).
    // SimpleWrapper<int> intWrapper { i };
    i = 2;
    cout << "wrapped value is " << intWrapper.get() << endl;

    // Try wrapping a string.
    string str { "test" };
    SimpleWrapper stringWrapper { str };
    str += "!";
    cout << "wrapped value is " << stringWrapper.get() << endl;
}

```

有关模板的细节，可参阅第 12 章和第 26 章。



32.1.9 约束模板参数

在 C++20 的 `concept` 中，可以对类模板和函数模板的模板参数添加约束。例如，下面的代码片段将前一节的 `SimpleWrapper` 类模板的模板类型参数 `T` 约束为浮点型或整型。为 `T` 指定不满足这些约束的类型将导致编译错误。

```
import <concepts>;

export template <typename T> requires std::floating_point<T> || std::integral<T>
class SimpleWrapper
{
    public:
        SimpleWrapper(T& object) : m_object { object } { }
        T& get() { return m_object; }
    private:
        T& m_object;
};
```

第12章介绍了 concept 的细节。

32.2 始终存在更好的方法

在你阅读这段文字时，全球数千名 C++ 程序员也许正在解决已经解决的问题。在位于圣何塞的办公室里，有人正在从头编写智能指针实现，该实现使用引用计数。地中海的小岛上，一位年轻程序员正在设计一种类层次结构，通过使用混入类获得极大好处。

作为一名专业的 C++ 程序员，不要将过多时间花在重新发明上，要将更多时间用于以新方式适应可重用概念。本节列举一些通用方法，可直接将这些方法应用于自己的程序中，或根据自己的需要进行定制。

32.2.1 RAII

RAII(Resource Acquisition Is Initialization，资源获得即初始化)是一个简单却十分强大的概念。它用于获得一些资源的所有权，并在 RAII 实例离开作用域时自动释放已获取的资源。这是在确定的时间点发生的。基本上，新 RAII 实例的构造函数获取特定资源的所有权，并使用资源初始化实例，因此得名 RAII。在销毁 RAII 实例时，析构函数自动释放所获取的资源。

下面的 RAII 类 File 安全地包装 C 风格的文件句柄(std::FILE)，并在 RAII 实例离开作用域时自动关闭文件。RAII 类也提供 get()、release() 和 reset() 方法，这些方法的行为类似于标准库类(如 std::unique_ptr)中的同名方法。RAII 类通常不允许拷贝构造和拷贝复制，因此，该实现删除这些成员。

```
#include <cstdio>

class File final
{
    public:
        explicit File(std::FILE* file) : m_file { file } { }
        ~File() { reset(); }

        // Prevent copy construction and copy assignment.
        File(const File& src) = delete;
        File& operator=(const File& rhs) = delete;

        // Allow move construction and move assignment.
        File(File&& src) noexcept = default;
        File& operator=(File&& rhs) noexcept = default;

        // get(), release(), and reset()
```

```

std::FILE* get() const noexcept { return m_file; }

[[nodiscard]] std::FILE* release() noexcept
{
    std::FILE* file { m_file };
    m_file = nullptr;
    return file;
}

void reset(std::FILE* file = nullptr) noexcept
{
    if (m_file) { fclose(m_file); }
    m_file = file;
}

private:
    std::FILE* m_file { nullptr };
};

```

用法如下:

```
File myFile { fopen("input.txt", "r") };
```

`myFile` 实例一旦离开作用域, 就会调用它的析构函数, 并自动关闭文件。

使用 `RAII` 类有一个需要注意的重要缺陷。你可能无意中编写了一行代码, 以为它在某个范围内正确地创建了一个 `RAII` 示例, 但实际上却创建了一个临时对象。当这个代码完成时, 对象会立刻被销毁。这个问题最好通过标准库中的 `RAII` 类 `std::unique_lock`(见第 27 章)来解释。`unique_lock` 的正确用法如下:

```

class Foo
{
public:
    void setData()
    {
        unique_lock<mutex> lock(m_mutex);
        // ...
    }
private:
    mutex m_mutex;
};

```

`setData()` 方法使用 `RAII` 对象 `unique_lock` 构建一个本地 `lock` 对象, 该对象锁定 `m_mutex` 数据成员, 并在方法结束时自动解锁互斥体。

但由于不会在定义后直接使用 `lock` 变量, 很容易犯以下错误:

```
unique_lock<mutex>(m_mutex);
```

在上述代码中, 无意中忘了给 `unique_lock` 指定名称。这是可编译的, 但行为不符合预期。它实际上将声明一个本地变量 `m_mutex`(隐藏 `m_mutex` 数据成员), 并调用 `unique_lock` 的默认构造函数对它进行初始化。结果, `m_mutex` 数据成员并未锁定! 但如果警告级别设置得足够高, 编译器会给出一个警告, 大致是:

```
warning C4458: declaration of 'm_mutex' hides class member
```

如果使用如下所示的统一初始化语法，编译器不会生成警告，但它也不会做想要的事情。下面的语句为 `m_mutex` 创建了一个临时锁，但因为它是临时的，所以这个锁会在语句结束时被立刻释放。

```
unique_lock<mutex> { m_mutex };
```

此外，可能会在作为参数传递的名称中出现错误，例如：

```
unique_lock<mutex>(m);
```

这里忘记为锁指定一个名称，并且输入了错误的参数名。这段代码只是声明了一个名为 `m` 的局部变量，并用 `unique_lock` 的默认构造函数初始化它。编译器甚至不会生成警告，除非警告 `m` 是一个未引用的局部变量。但是在这种情况下，如果像下面这样使用统一的初始化语法，编译器就会报出一个错误，抱怨一个未声明的标识符 `m`：

```
unique_lock<mutex> { m };
```

警告：

确保经常命名你的 RAII 实例。此外，建议不要在一个 RAII 类中包含默认构造函数。这避免了这里讨论的一些问题。

32.2.2 双分派

双分派(double dispatch)技术用于给多态性概念添加附加维度。如第 5 章“面向对象设计”所述，多态性允许程序在运行时基于类型确定行为。例如，`Animal` 类有 `move()` 方法。所有动物都能走动，但走动方式是不同的。为 `Animal` 类的每个派生类定义 `move()` 方法，这样，可在运行时为适当的动物调用或分派适当的方法，在编译期可不了解动物类型。第 10 章解释了如何使用虚方法实现这种运行时多态性。

但有时，方法的行为取决于两个对象(而不是一个对象)的运行时类型。例如，假设需要给 `Animal` 类添加一个方法，如果一种动物捕食另一种动物，将返回 `true`，否则返回 `false`。这个决策基于两个因素：作为捕食者的动物类型，以及作为被捕食者的动物类型。遗憾的是，C++ 没有提供相应的语言机制，以根据多个对象的运行时类型选择行为。虚方法本身不足以建立这种场景的模型，它们仅根据接收对象的运行时类型来确定方法或行为。

有些面向对象语言允许基于两个或多个对象的运行时类型，在运行时选择方法，它们将该功能称为多方法(multi-methods)。而在 C++ 中，并没有支持多方法的核心语言功能，但可以使用双分派技术，从而创建针对多个对象的虚函数。

注意：

双分派实际上是多分派的特例。所谓多分派，是指根据两个或多个对象的运行时类型来选择行为。在实践中，双分派可根据两个对象的运行时类型选择行为，这通常就能满足需要。

1. 第一次尝试：蛮力方式

要使方法的行为取决于两个不同对象的运行时类型，最直接的方法就是站在其中一个对象的立场上，使用一系列 `if/else` 构造来检查另一个对象的类型。例如，可在 `Animal` 的每个派生类中实现 `eats()` 方法，`eats()` 方法将另一种动物作为参数。在基类中将 `eats()` 方法声明为纯虚函数：

```
class Animal
{
```

```

    public:
        virtual bool eats(const Animal& prey) const = 0;
};

```

每个派生类都实现 `eats()` 方法，并基于参数类型返回适当的值。有几个派生类的 `eats()` 实现如下所示。注意，`Dinosaur` 派生类未使用一系列 `if/else` 结构，因为在笔者(T-rex)看来，和任何食肉恐龙一样，什么都吃。

```

bool Bear::eats(const Animal& prey) const
{
    if (typeid(prey) == typeid(Bear)) {
        return false;
    } else if (typeid(prey) == typeid(Fish)) {
        return true;
    } else if (typeid(prey) == typeid(Dinosaur)) {
        return false;
    }
    return false;
}

bool Fish::eats(const Animal& prey) const
{
    if (typeid(prey) == typeid(Bear)) {
        return false;
    } else if (typeid(prey) == typeid(Fish)) {
        return true;
    } else if (typeid(prey) == typeid(Dinosaur)) {
        return false;
    }
    return false;
}

bool TRex::eats(const Animal& prey) const
{
    return true;
}

```

这种蛮力方式也可行，如果类的数量不多，这可能是最直接的方式。但出于以下几种原因，最好避免使用这种方式：

- OOP(Object-Oriented Programming, 面向对象编程)纯粹主义者通常不赞成明确查询对象的类型，因为这种设计隐式地表明它不是合理的面向对象结构。
- 随着类型数量的增加，此类代码会变得杂乱不堪。
- 这种方式不强制要求派生类考虑新类型。例如，若添加了 `Donkey` 类，`Bear` 类会继续编译。让 `Bear` 捕食 `Donkey` 时，会返回 `false`；但每个人都知道熊吃驴子。熊会拒绝吃一头驴，因为没有其他 `if` 语句明确检查驴。

2. 第二次尝试：包含重载的单个多态

可尝试使用带有重载的多态，以绕过所有的 `if/else` 级联结构。不是给每个派生类都提供接收 `Animal` 引用的 `eats()` 方法，为什么不考虑重载 `Animal` 的每个派生类的方法？基类定义如下所示：

```

class Animal
{
    public:

```

```

        virtual bool eats(const Bear&) const = 0;
        virtual bool eats(const Fish&) const = 0;
        virtual bool eats(const Dinosaur&) const = 0;
};

```

由于这些方法在基类中是纯虚方法，每个派生类都必须实现其他每种 `Animal` 类型的行为。例如，`Bear` 类包含的方法如下：

```

class Bear : public Animal
{
public:
    bool eats(const Bear&) const override { return false; }
    bool eats(const Fish&) const override { return true; }
    bool eats(const Dinosaur&) const override { return false; }
};

```

这种方式初看起来是可行的，但只解决了一半问题。为了调用 `Animal` 的适当 `eats()` 方法，编译器需要了解被捕食动物的编译期类型。如下调用将能够成功，因为捕食者和被捕食者的编译期类型都是已知的：

```

Bear myBear;
Fish myFish;
cout << myBear.eats(myFish) << endl;

```

遗憾的是，这种解决方案只在一个方向上具有多态性。可以通过 `Animal` 引用访问 `myBear`，此时将调用正确的方法：

```

Bear myBear;
Fish myFish;
Animal& animalRef { myBear };
cout << animalRef.eats(myFish) << endl;

```

但反过来却行不通。如果给 `eats()` 方法传递 `Animal` 引用，将看到编译错误，因为没有接收 `Animal` 引用的 `eats()` 方法。在编译期，编译器无法确定调用哪个版本。下例无法编译：

```

Bear myBear;
Fish myFish;
Animal& animalRef = myFish;
cout << myBear.eats(animalRef) << endl; // BUG! No method Bear::eats(Animal&)

```

由于编译器需要了解在编译期调用哪个重载的 `eats()` 方法版本，因此这不是真正的多态解决方案。有时是行不通的，例如迭代 `Animal` 引用的数组，并将每一个传给 `eats()` 方法的情形。

3. 第三次尝试：双分派

对于多类型问题，双分派技术是真正的多态解决方案。在 C++ 中，通过重写派生类中的方法来获得多态性。在运行时，基于对象的实际类型调用方法。前面的单个多态尝试不可行，因为它尝试使用多态来确定调用方法的哪个重载版本，而非使用它确定调用哪个类的方法。

首先重点分析单个派生类，可能是 `Bear` 类。该类需要一个具有以下声明的方法：

```

bool eats(const Animal& prey) const override;

```

双分派的关键在于基于参数上的方法调用来确定结果。假设 `Animal` 类有一个 `eatenBy()` 方法，该方法将 `Animal` 引用作为参数。如果当前 `Animal` 会被传入的动物捕食，该方法返回 `true`。有了这个方法，`eats()` 方法的定义变得十分简单：

```
bool Bear::eats(const Animal& prey) const
{
    return prey.eatenBy(*this);
}
```

初看起来，这个解决方案给单多态方法添加了另一个方法调用层。毕竟，每个派生类都必须为每个 `Animal` 派生类实现 `eatenBy()` 版本。但有一个重要区别：多态发生了两次！当调用 `eats()` 方法时，多态性确定是调用 `Bear::eats()`、`Fish::eats()` 还是其他。当调用 `eatenBy()` 方法时，多态性再次确定要调用哪个类的方法版本，调用 `prey` 对象的运行时类型的 `eatenBy()`。注意，`*this` 的运行时类型始终与编译期类型相同，这样，编译器可为实参(这里是 `Bear`)调用 `eatenBy()` 的正确重载版本。

下面是使用双分派的 `Animal` 层次结构的类定义。前置声明是必需的，因为基类使用派生类的引用。注意，`Animal` 的派生类以相同的方式实现 `eats()` 方法，但不能向上延伸到基类；如果尝试这么做，编译器不知道要调用 `eatenBy()` 方法的哪个重载版本，因为 `*this` 是 `Animal` 而非特定的派生类。根据对象的编译期类型(而非运行时类型)来确定方法重载方案。

```
// Forward declarations.
class Fish;
class Bear;
class TRex;

class Animal
{
public:
    virtual bool eats(const Animal& prey) const = 0;

    virtual bool eatenBy(const Bear&) const = 0;
    virtual bool eatenBy(const Fish&) const = 0;
    virtual bool eatenBy(const TRex&) const = 0;
};

class Bear : public Animal
{
public:
    bool eats(const Animal& prey) const override{ return prey.eatenBy(*this); }

    bool eatenBy(const Bear&) const override { return false; }
    bool eatenBy(const Fish&) const override { return false; }
    bool eatenBy(const TRex&) const override { return true; }
};

class Fish : public Animal
{
public:
    bool eats(const Animal& prey) const override{ return prey.eatenBy(*this); }
    bool eatenBy(const Bear&) const override { return true; }
    bool eatenBy(const Fish&) const override { return true; }
    bool eatenBy(const TRex&) const override { return true; }
};

class TRex : public Animal
{
public:
    bool eats(const Animal& prey) const override{ return prey.eatenBy(*this); }
```



```
bool eatenBy(const Bear&) const override { return false; }
bool eatenBy(const Fish&) const override { return false; }
bool eatenBy(const TRex&) const override { return true; }
};
```

双分派技术需要练习一段时间才能用熟练，建议反复研究这里的代码。

32.2.3 混入类

第 5 章和第 6 章介绍了混入类的技术。他们会回答这个问题：类还能做什么？答案往往以 able 结尾。例如 Clickable、Drawable、Printable、Lovable 等。混入类是一种可以添加功能到一个类而不必提交一个完整的 is-a 关系的方式。在 C++ 中实现混入类有两种主要方式，它们会在下一节讨论：

- (1) 使用多继承
- (2) 使用类模板

1. 使用多继承

本节介绍如何使用多重继承技术设计、实现和使用混入类。

设计混入类

混入类包含可供其他类重用的实际代码。混入类往往实现一项良好定义的功能。例如，可能有一个 Playable 混入类，它混入了多种媒体对象。例如，该混入类可能包含与电脑声卡通信的大多数代码。通过混入类，媒体对象可自由地获得相应功能。

设计混入类时，需要考虑添加什么行为，以及行为属于对象层次结构还是属于单个类。使用前面的例子，如果所有媒体类都是可播放的，则基类应当从 Playable 类派生，而不是将 Playable 类混入所有派生类中。只有在某些媒体类是可播放的，并且这些媒体类分散在层次结构中时，才应当使用混入类。

如果类在一条轴上组织成层次结构，但它们还与另一条轴有相似之处，混入类将特别有用。例如，考虑一款在网格上玩的战争模拟游戏。每个网格点都可包含具有攻防能力和其他特性的 Item，有些项(如 Castle)是固定不动的。其他项(如 Knight 或 FloatingCastle)可在网格上移动。开始设计对象层次结构时，可能得到如图 32-1 所示的层次结构，根据项的攻防能力组织类。

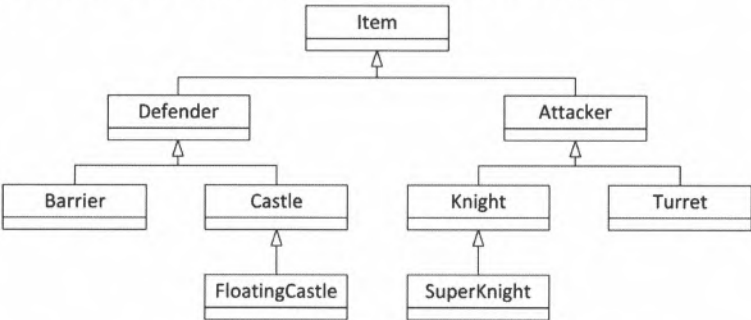


图 32-1 根据项的攻防能力组织类的层次结构

图 32-1 所示的层次结构忽略了某些类包含的移动功能。围绕移动构建层次结构将得到如图 32-2 所示的层次结构。

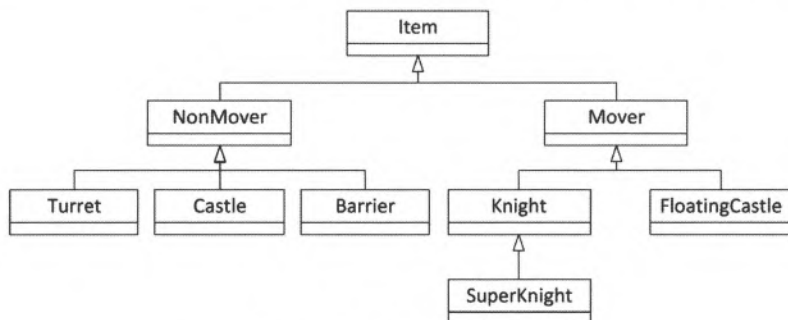


图 32-2 围绕移动构建的层次结构

当然,图 32-2 所示的设计对图 32-1 做了大刀阔斧般的改动,优秀的面向对象程序此时该怎么做?

对于这个问题有两个常用的解决方案。假设接受第一个基于攻防进行组织的层次结构,那么需要采用一些方式将移动性考虑在内。

一种方案是,虽然派生类只有一部分支持移动,但可给 Item 基类添加 move()方法。默认实现什么都不做。一部分派生类可以重写 move(),以真正更改它们在网格上的位置。

另一种方案是编写 Movable 混入类。可保留图 32-1 所示的优雅层次结构,但一部分可从 Movable 及其父类继承。图 32-3 显示了这种设计方式。

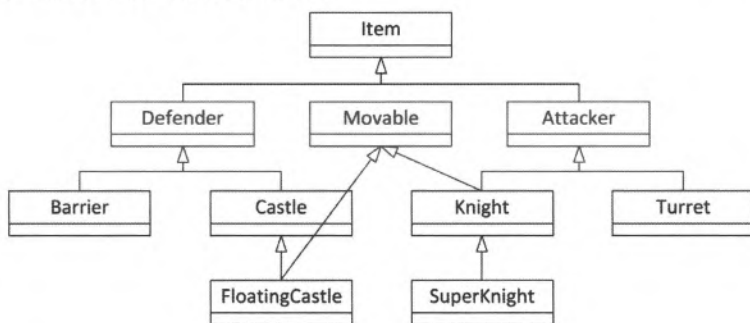


图 32-3 设计方法

实现混入类

编写混入类与编写普通的类没什么区别。实际上,编写混入类往往更简单。

使用前面的战争模拟游戏, Movable 混入类可能如下所示:

```

class Movable
{
    public:
        virtual void move() { /* Implementation to move an item... */ }
};
  
```

Movable 混入类实现实际代码,以便在网格上移动项。它还为可移动的项提供了类型。例如,可创建一组可移动项,而不必了解或关心它们所属的实际派生类。

使用混入类

使用混入类的代码的语法与多重继承相同。除了从主层次结构的父类派生外,还从混入类派生。

```

class FloatingCastle : public Castle, public Movable { /* ... */ };
  
```

这样，就将 Movable 混入类提供的功能成功混入 FloatingCastle 类。

现在有了一个类，它位于层次结构中最合理的位置，但仍与层次结构中其他位置的对象具有共性。

2. 使用类模板

在 C++ 中实现混入类的第二种方法是使用所谓奇怪的重复模板模式(CRTP)。

混入类本身是接收模板类型翻出并从该类型派生自己的类模板，这就是它被称为 recurring 的原因。

第 6 章解释了实现 SelfDrivable 混入类模板的机制，这个模板可以用来创建自动驾驶的汽车和卡车。现在你已经很熟悉类模板(第 12 章和第 26 章)，第 6 章中的 SelfDrivable 混入类示例应该不会再让你感到意外。混入类的定义如下：

```
template <typename T>
class SelfDrivable : public T { /* ... */};
```

如果你有一个 Car 类和一个 Truck 类，可以很容易地定义一个自动驾驶的汽车和卡车，如下所示。

```
SelfDrivable<Car> selfDrivingCar;
SelfDrivable<Truck> selfDrivingTruck;
```

通过这种方式，可以将功能添加到现有的 Car 类和 Truck 类中，而不必修改这些类。

32.3 面向对象的框架

图形操作系统于 20 世纪 80 年代问世，那时，过程语言最常见，编写 GUI 应用程序时，通常会操纵复杂的数据结构，并将它们传给 OS 提供的函数。例如，要在窗口中绘制一个矩形，就必须用适当信息填充 Window 结构，然后将 Window 结构传递给 drawRect()函数。

随着 OOP(Object Oriented Programming，面向对象编程)日益流行，程序员开始探索将 OOP 范型应用于 GUI 开发。面向对象的框架应运而生。

大致来讲，框架是一组类，为一些底层功能提供面向对象的接口。在讨论框架时，程序员通常指用于开发普通应用程序的庞大类库。但实际上，框架可以表示任何规模的功能。如果编写了一组类，为应用程序提供数据库功能，就可以将这些类称为框架。

32.3.1 使用框架

框架的本质特征在于提供自己的一组技术和模式。通常需要对框架进行较长时间的学习才能开始使用框架，因为它们有自己的“心智模型”。在开始使用 MFC(Microsoft Foundation Class)等大型应用程序框架前，需要了解它们与外界的交互方式。

各个框架的抽象方式和实际实现大相径庭。许多框架建立在传统的过程 API 之上，这会对设计的各个方面产生影响。其他框架则一直采用面向对象的设计方式编写。一些框架的理念可能与 C++ 语言相悖。例如，框架可能有意识地回避多重继承概念。

开始使用新框架时，首先要确定框架的工作原理是什么？遵循什么设计原理？开发人员传达的理念是什么？框架广泛使用语言的哪些方面？这些都是至关重要的问题。如果未能理解框架的设计、模型或语言功能，可能很快将逾越框架的界限。

理解框架的设计后，将可以扩展框架。例如，如果框架缺少一项功能(如支持打印)，可沿袭框架模型，编写自己的打印类。这样，可保证应用程序模型的一致性，而且代码可供其他应用程序重用。

框架可能使用某些特定的数据类型。例如，MFC 框架使用 CString 数据类型表示字符串，而不使

用标准库的 `std::string` 类。这并不意味着必须将整个代码库切换到框架提供的类型。相反，可在框架代码和其余代码的边界处转换数据类型。

32.3.2 MVC 范型

如前所述，框架采用的方法与面向对象的设计方法存在差异。一种常见范型是 MVC(Model-View-Controller，模型-视图-控制器)。这种范型的模型来源是：许多应用程序经常处理数据集，有一个或多个数据视图，还会操纵数据。

在 MVC 中，将数据集称为“模型”。在赛车模拟器中，模型将跟踪各种统计数据，如赛车的当前速度以及持续遭受的磨损。在实践中，模型通常采用类的形式，类有多个获取器和设置器。赛车模型的定义可能如下所示：

```
class RaceCar
{
public:
    RaceCar();
    virtual ~RaceCar() = default;

    virtual double getSpeed() const;
    virtual void setSpeed(double speed);

    virtual double getDamageLevel() const;
    virtual void setDamageLevel(double damage);
private:
    double m_speed { 0.0 };
    double m_damageLevel { 0.0 };
};
```

视图是模型的特定可视化形式。例如，`RaceCar` 可能有两个视图。第一个视图是赛车的图形视图，第二个视图显示随时间推移的受损程度。要点在于，这两个视图都在同一数据上操作，它们以不同方式查看相同的信息。这是 MVC 范型的一个主要优点：将数据与显示分开后，可以更好地组织代码，方便地创建其他视图。

MVC 范型的最后一部分是控制器。控制器是一段代码，通过更改模型来响应一些事件。例如，当赛车模拟器的司机遇到混凝土护栏时，控制器会告诉模型大幅提高赛车的受损级别，并降低速度。控制器也可操纵视图。例如，当用户在用户界面上移动滚动条时，控制器会告诉视图滚动其内容。

MVC 的这三个组件以反馈循环的方式进行交互。控制器处理操作，从而调整模型和(或)视图。如果模型发生变化，它会通知视图进行相应的更新。图 32-4 显示了交互方式。

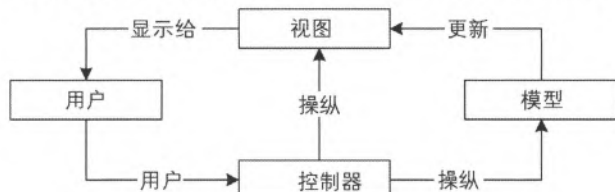


图 32-4 交互方式

MVC 范型得到很多主流框架的广泛支持。即使是有别于传统的应用程序(如 Web 应用程序)也在向 MVC 靠拢，因为它强制要求明确分离数据、数据的操纵以及显示。

MVC 设计模式已演变出多个不同变体，如 MVP(Model-View-Presenter)、MVA(Model-View-Adapter)

和 MVVM(Model-View-ViewModel)等。

32.4 本章小结

在本章中，你学习了专业 C++ 程序员在日常项目中使用的常见技术。在从事软件开发的过程中，你一定会有自己的可重用类和库的集合。了解设计技术将为开发和使用设计模式打开一扇大门，设计模式是更高级的可重用结构。第 33 章“应用设计模式”将介绍多种设计模式的运用方式。

32.5 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

练习 32-1 编写一个 RAII 类模板 `Pointer`，它可以存储指向任何类型对象的指针，并在 RAII 实例超出作用域时自动释放内存。提供 `reset()` 和 `release()` 方法以及重载 `operator*`。

练习 32-2 修改习题 32-1 中的类模板，以便构造函数的参数为 `nullptr` 时抛出异常。

练习 32-3 从习题 32-2 中拿到答案，并添加名为 `assign()` 的模板方法，接收类型为 `E` 的参数，并将这个参数赋值给包裹的指针所指向的数据。向模板方法添加一个约束，确保类型 `E` 可转换为类型 `T`。

练习 32-4 编写一个 `lambda` 表达式，返回两个参数的和。`lambda` 表达式应该处理所有种类的数据类型，比如整型和浮点型。通过计算 11 与 22，1.1 与 2.2 的和来测试你的 `lambda` 表达式。

第33章

应用设计模式

本章内容

- 设计模式的含义及其与设计技术的区别
- 如何使用以下设计模式：
 - 依赖注入
 - 抽象工厂模式
 - 工厂方法模式
 - 适配器模式
 - 代理模式
 - 迭代器模式
 - 观察者模式
 - 装饰器模式
 - 责任链模式
 - 单例模式

设计模式是组织用于解决一般性问题的标准方法。与设计技术相比，设计模式的语言专用性弱一些。设计模式和设计技术之间的区别是模糊的，各类书籍给出的定义也不尽相同。本书将“设计技术”定义为 C++ 语言的专用策略，将“设计模式”视为适用于任何面向对象语言的通用模式。例如 C++、C#、Java 或 Smalltalk。实际上，如果熟悉 C# 或 Java 编程，你就会认识其中的很多模式。

注意，很多设计模式都有多个不同名称。设计模式本身的区别有时也是模糊的，不同来源对它们的描述和分类都稍有不同。事实上，你会发现，在不同的书籍和来源中，同一个名称指代的是不同的设计模式，甚至对于应将哪些设计方法归类为设计模式仍存在异议。本书使用的名称基本沿袭重要著作 *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-wesley Professional, 1994)，但也会在适当之处指出其他名称和变体。

设计模式是一个简单却强大的概念。一旦掌握程序中反复出现的面向对象的交互方式，通过选择适当的设计模式就能找到完美的解决方案。本章将详细描述几种设计模式，并提供示例实现。

设计的任何方面都会在程序员中引起争论，这是一件好事。不要将这些设计模式视为完成任何任务的唯一途径，可巧用这些方法和想法，并加以改进，形成新的设计模式。

33.1 依赖注入

依赖注入是支持依赖倒置原则(DIP)的一种方法。通过依赖注入，可以使用接口反转依赖关系。为某个提供的服务创建接口。如果组件需要一组服务，则将这些服务的接口注入组件中。依赖注入使单元测试变得更容易，因为可以轻松地模拟服务。本节讨论一个使用依赖注入实现的日志机制的示例。

33.1.1 示例：日志机制

依赖注入 logger 示例使用了名为 `ILogger` 的接口。任何想要使用 `log` 的代码都需要使用这个 `ILogger` 接口。随后，将该接口的实现注入能够使用日志功能的任何代码中。使用此模式，单元测试可以为 `ILogger` 接口注入一个特殊的模拟实现，来验证是否记录了正确的信息。此外，可以轻松地交换具体的 logger，不需要修改任何客户端代码。

33.1.2 依赖注入 logger 的实现

该实现提供了具有以下特性的 `Logger` 类：

- 可以记录单个字符串。
- 每个日志消息都有一个关联的日志级别，该级别是日志消息的前缀。
- 可以将 logger 设置为仅记录某个级别日志的消息。
- 每个日志消息都会被刷到磁盘上，以便立即出现在文件中。

首先定义 `ILogger` 接口：

```
export class ILogger
{
    public:
        virtual ~ILogger() = default; // Virtual destructor.

        // Enumeration for the different log levels.
        enum class LogLevel {
            Error,
            Info,
            Debug
        };

        // Sets the log level.
        virtual void setLogLevel(LogLevel level) = 0;

        // Logs a single message at the given log level.
        virtual void log(std::string_view message, LogLevel logLevel) = 0;
};
```

接下来，`Logger` 类实现如下：

```
export class Logger : public ILogger
{
    public:
        explicit Logger(std::string_view logFilename);
        virtual ~Logger();
        void setLogLevel(LogLevel level) override;
        void log(std::string_view message, LogLevel logLevel) override;
    private:
```

```

        // Converts a log level to a human readable string.
        std::string_view getLogLevelString(LogLevel level) const;

        std::ofstream m_outputStream;
        LogLevel m_logLevel { LogLevel::Error };
};

```

Logger 类的实现相当简单。一旦打开日志文件，会把每个日志信息写入文件，并添加日志级别，然后将其刷新到磁盘中。方法实现如下：

```

Logger::Logger(string_view logFilename)
{
    m_outputStream.open(logFilename.data(), ios_base::app);
    if (!m_outputStream.good()) {
        throw runtime_error { "Unable to initialize the Logger!" };
    }
}

Logger::~Logger()
{
    m_outputStream << "Logger shutting down." << endl;
    m_outputStream.close();
}

void Logger::setLogLevel(LogLevel level)
{
    m_logLevel = level;
}

string_view Logger::getLogLevelString(LogLevel level) const
{
    switch (level) {
        case LogLevel::Error: return "ERROR";
        case LogLevel::Info: return "INFO";
        case LogLevel::Debug: return "DEBUG";
    }
    throw runtime_error { "Invalid log level." };
}

void Logger::log(string_view message, LogLevel logLevel)
{
    if (m_logLevel < logLevel) { return; }

    m_outputStream << format("{}: {}", getLogLevelString(logLevel), message)
        << endl;
}

```

33.1.3 使用依赖注入

假设有一个名为 Foo 的类，它想要使用日志功能。通过依赖注入模式，将具体的 ILogger 实例注入类中，例如通过构造函数：

```

class Foo
{
public:
    explicit Foo(ILogger& logger) : m_logger { logger } { }
}

```



```

        void doSomething()
        {
            m_logger.log("Hello dependency injection!", ILogger::LogLevel::Info);
        }
    private:
        ILogger& m_logger;
};

```

如果创建了一个 Foo 实例，会注入一个具体的 ILogger:

```

Logger concreteLogger { "log.out" };
concreteLogger.setLogLevel(ILogger::LogLevel::Debug);

Foo f { concreteLogger };
f.doSomething();

```

33.2 抽象工厂模式

现实中的工厂会制造有形物品，如桌子或汽车。与此类似，面向对象编程领域中的工厂会构建对象。在程序中使用工厂时，想要创建特定对象的代码片段向工厂索要对象实例，而非调用自身的对象构造函数。例如，一个室内装潢程序可能有一个 FurnitureFactory 对象。当代码的一部分需要一件家具(如桌子)时，会调用 FurnitureFactory 对象的 createTable()方法，该方法将返回新的桌子。

乍一看，工厂似乎只会增加设计复杂度，没什么明显好处。这样一来，好像程序变得更复杂了。不必调用 FurnitureFactory 对象的 createTable()方法，我们可直接创建一个新的 Table 对象。然而使用工厂的一个好处是，可将类层次结构与工厂结合使用以构建对象，不必了解具体的类。工厂可与类层次结构并行运行(当然，并非一定要并行运行)。工厂也可创建任意数量的具体类型。

工厂的另一个好处是，不必在整个代码中直接创建各种对象，通过工厂可以在程序的不同部分为特定域创建相同类型的对象。

使用工厂的另一个原因是，创建对象需要工厂拥有的某些信息、状态和资源等，工厂的用户不应该知道这些。如果创建对象时，需要按正确顺序执行一系列复杂步骤，或者需要按正确的顺序将创建的所有对象链接到其他对象，也可以使用工厂。

工厂的主要好处在于实现了创建过程的抽象化。使用依赖注入时，可方便地在程序中替换不同的工厂。就像在创建对象时可使用多态性一样，工厂也可以使用多态性。

面向对象编程中有两种与工厂相关的模式：抽象工厂模式和工厂方法模式。本节讨论抽象工厂模式，下一节讨论工厂方法模式。

33.2.1 示例：模拟汽车工厂

想象一个能够生产汽车的工厂。工厂创建它所要求的汽车类型。首先，需要一个层次结构来表示多种类型的汽车。图 33-1 介绍了一个 ICar 接口，该接口使用虚方法检索有关特定汽车的信息。Toyota 和 Ford 汽车都派生自 ICar，最后，Ford 和 Toyota 都有一款轿车和一款 SUV。

继汽车层次结构后，需要一个工厂层次结构。抽象工厂只需要一个公开接口来创建品牌无关的轿车或 SUV，具体的工厂从具体的品牌构建具体的模型。层次结构如图 33-2 所示。

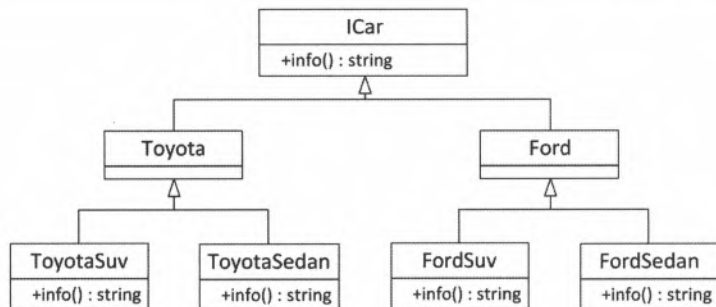


图 33-1 ICar 接口

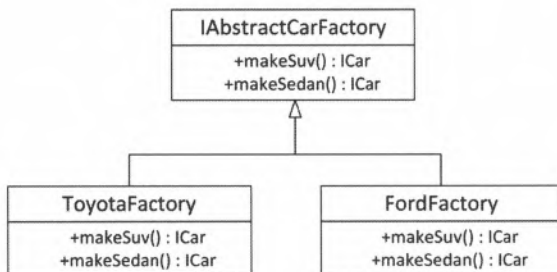


图 33-2 层次结构

33.2.2 实现抽象工厂

汽车层次结构的实现很简单：

```

export class ICar
{
    public:
        virtual ~ICar() = default; // Always a virtual destructor!
        virtual std::string info() const = 0;
};

export class Ford : public ICar { };

export class FordSedan : public Ford
{
    public:
        std::string info() const override { return "Ford Sedan"; }
};

export class FordSuv : public Ford
{
    public:
        std::string info() const override { return "Ford Suv"; }
};

export class Toyota : public ICar { };

export class ToyotaSedan : public Toyota
{

```

```

        public:
            std::string info() const override { return "Toyota Sedan"; }
    };

    export class ToyotaSuv : public Toyota
    {
        public:
            std::string info() const override { return "Toyota Suv"; }
    };

```

接下来是 `IAbstractCarFactory` 接口。它只暴露了创建轿车或 SUV 的接口，不必知道任何具体工厂或汽车。

```

    export class IAbstractCarFactory
    {
        public:
            virtual ~IAbstractCarFactory() = default; // Always a virtual destructor!
            virtual std::unique_ptr<ICar> makeSuv() = 0;
            virtual std::unique_ptr<ICar> makeSedan() = 0;
    };

```

最后有具体工厂，创建具体的汽车模型。这里只展示 Ford 的工厂，Toyota 工厂类似。

```

    export class FordFactory : public IAbstractCarFactory
    {
        public:
            std::unique_ptr<ICar> makeSuv() override {
                return std::make_unique<FordSuv>(); }
            std::unique_ptr<ICar> makeSedan() override {
                return std::make_unique<FordSedan>(); }
    };

```

本例中使用的方法称为抽象工厂，因为创建的对象类型取决于正在使用的具体工厂。

33.2.3 使用抽象工厂

下面的示例展示了如何使用已实现的工厂。它有一个函数，可以接收一个抽象的汽车工厂，使用它制造一辆轿车和一辆 SUV，并打印每一辆已生产汽车的信息。这个函数不知道任何具体工厂或任何具体汽车；也就是说，它只使用接口。`main()`函数创建两个工厂，一个用于 Ford，一个用于 Toyota，然后要求 `createSomeCars()`函数使用这些工厂创建一些汽车。

```

void createSomeCars(IAbstractCarFactory& carFactory)
{
    auto sedan { carFactory.makeSedan() };
    auto suv { carFactory.makeSuv() };
    cout << format("Sedan: {}\n", sedan->info());
    cout << format("Suv: {}\n", suv->info());
}

int main()
{
    FordFactory fordFactory;
    ToyotaFactory toyotaFactory;
    createSomeCars(fordFactory);
    createSomeCars(toyotaFactory);
}

```

代码片段的输出如下：

```
Sedan: Ford Sedan
Suv: Ford Suv
Sedan: Toyota Sedan
Suv: Toyota Suv
```

33.3 工厂方法模式

第二种与工厂相关的模式称为工厂方法模式。通过这种模式，创建哪种对象完全由具体工厂决定。在前面的抽象工厂方法示例中，`IAbstractCarFactory` 有创建 SUV 或轿车的方法。使用工厂方法模式，只需要从工厂请求一辆汽车，具体的工厂决定具体要创建什么。看看另一个汽车工厂的模拟。

33.3.1 示例：模拟第二个汽车工厂

在现实世界中，当谈到驾驶小汽车时，无论是哪类小汽车，都可以驾驶。丰田(Toyota)或福特(Ford)汽车都可以，它们都可供驾驶。现在假设要买一辆小汽车，需要指定是买丰田还是福特汽车，是这样吗？未必总是如此，也可以说：“我需要一辆小汽车。”你会得到一辆小汽车，具体型号取决于所在的位置。如果在丰田工厂附近，则很可能买到一辆丰田汽车。如果在福特工厂附近，则很可能买到一辆福特汽车。

同样的概念适用于 C++ 编程领域。第一个概念“一般小汽车都是可驾驶的”没什么新意，这是标准的多态性，见第 5 章“面向对象设计”的讨论。可编写一个抽象的 `Car` 类来定义 `drive()` 虚方法。Toyota 和 Ford 可以实现这样的接口。

程序可驾驶 `Car`，不必了解它们到底是 Toyota 还是 Ford。但在标准的面向对象编程领域，在创建 `Car` 时，必须指定是 Toyota 还是 Ford。此时，需要调用 Toyota 或 Ford 的构造函数，不能只说：“我需要一辆小汽车。”但假设还有汽车工厂的并行类层次结构。`CarFactory` 基类可以定义一个公有的非虚 `requestCar()` 方法，该方法将工作转发到受保护的虚方法 `createCar()` 中。ToyotaFactory 和 FordFactory 派生类重写 `createCar()` 方法来构建 Toyota 或 Ford。ICar 和 CarFactory 的层次结构如图 33-3 所示。

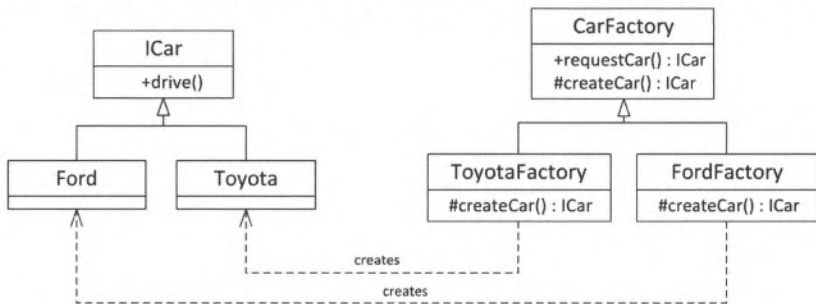


图 33-3 ICar 和 CarFactory 的层次结构

现在假设程序中有一个 `CarFactory` 对象。当程序中的代码(如汽车经销商)需要一辆新车时，会调用 `CarFactory` 对象的 `requestCar()` 方法。代码将获得 Toyota 或 Ford，具体取决于汽车工厂是 `ToyotaFactory` 还是 `FordFactory`。图 33-4 显示了使用 `ToyotaFactory` 的汽车经销商程序中的对象。



图 33-4 使用 ToyotaFactory 的汽车经销商程序中的对象

图 33-5 显示了相同的程序，但用 FordFactory 替代了 ToyotaFactory。注意，CarDealer 对象及其与工厂的关系保持不变。



图 33-5 相同的程序

这个示例演示了为工厂使用多态性。当要求汽车工厂提供汽车时，可能并不知道是丰田工厂还是福特工厂，但无论是哪家工厂，都能提供用户可驾驶的汽车。使用该方法可得到易于扩展的程序；只需要更改工厂实例，程序就可以使用完全不同的对象和类集合。

33.3.2 实现工厂

使用工厂的一个原因是：想要创建的对象类型依赖于一些条件。例如，需要一辆汽车，但想将订单递交给迄今收到订单最少的工厂，你并不介意最终获得的型号是丰田还是福特。下面的实现显示了如何在 C++ 中编写此类工厂。

首先需要 Car 类的层次结构。为简单起见，此处的 Car 类只有一个抽象方法，该方法返回汽车的描述信息：

```
export class ICar
{
    public:
        virtual ~ICar() = default; // Always a virtual destructor!
        virtual std::string info() const = 0;
};

export class Ford : public ICar
{
    public:
        std::string info() const override { return "Ford"; }
};

export class Toyota : public ICar
{
    public:
        std::string info() const override { return "Toyota"; }
};
```

CarFactory 基类更有趣。每个工厂都跟踪所生产汽车的数量。调用公有方法 requestCar() 时，对该

工厂生产的汽车数量加 1，然后调用纯虚方法 `createCar()`，`createCar()` 创建并返回新的汽车。目的是让各个工厂重写 `createCar()` 以返回适当的汽车类型。`CarFactory` 本身实现了 `requestCar()`，`requestCar()` 负责更新生产的汽车数量。`CarFactory` 还提供公有方法来查询每家工厂生成的汽车数量。

`CarFactory` 及其派生类的定义如下所示：

```
export class CarFactory
{
    public:
        virtual ~CarFactory() = default; // Always a virtual destructor!

        std::unique_ptr<ICar> requestCar()
        {
            // Increment the number of cars produced and return the new car.
            ++m_numberOfCarsProduced;
            return createCar();
        }

        size_t getNumberOfCarsProduced() const { return m_numberOfCarsProduced; }

    protected:
        virtual std::unique_ptr<ICar> createCar() = 0;
    private:
        size_t m_numberOfCarsProduced { 0 };
};

export class FordFactory : public CarFactory
{
    protected:
        std::unique_ptr<ICar> createCar() override {
            return std::make_unique<Ford>(); }
};

export class ToyotaFactory : public CarFactory
{
    protected:
        std::unique_ptr<ICar> createCar() override {
            return std::make_unique<Toyota>(); }
};
```

可以看到，派生类会简单地重写 `createCar()`，以返回它们生产的汽车类型。

注意：

工厂方法只是实现 `virtual` 构造函数的一种方式，以创建不同类型的对象。例如，`requestCar()` 方法会根据调用的工厂对象创建 `Toyota` 和 `Ford`。

33.3.3 使用工厂

使用工厂的最简单方式是实例化它并调用适当的方法，如下所示：

```
ToyotaFactory myFactory;
auto myCar { myFactory.requestCar() };
cout << myCar->info() << endl; // Outputs Toyota
```

一个更有趣的示例使用了 `virtual` 构造函数，在汽车生产数量最少的工厂生产汽车。为此，可创建

新工厂 `LeastBusyFactory`, `LeastBusyFactory` 从 `CarFactory` 派生, 其构造函数接收大量的其他 `CarFactory` 对象。与所有 `CarFactory` 类一样, `LeastBusyFactory` 重写 `createCar()` 方法。它的实现从传给构造函数的系列工厂中查找生产数量较少的工厂, 并要求那个工厂生产汽车。下面是此类工厂的实现:

```
class LeastBusyFactory : public CarFactory
{
public:
    // Constructs a LeastBusyFactory instance, taking ownership of
    // the given factories.
    explicit LeastBusyFactory(vector<unique_ptr<CarFactory>>&& factories);

protected:
    unique_ptr<ICar> createCar() override;

private:
    vector<unique_ptr<CarFactory>> m_factories;
};

LeastBusyFactory::LeastBusyFactory(vector<unique_ptr<CarFactory>>&& factories)
    : m_factories { move(factories) }
{
    if (m_factories.empty()) {
        throw runtime_error { "No factories provided." };
    }
}

unique_ptr<ICar> LeastBusyFactory::createCar()
{
    CarFactory* bestSoFar { m_factories[0].get() };

    for (auto& factory : m_factories) {
        if (factory->getNumberOfCarsProduced() <
            bestSoFar->getNumberOfCarsProduced()) {
            bestSoFar = factory.get();
        }
    }

    return bestSoFar->requestCar();
}
```

下面的代码生产 10 辆新汽车, 由生产数量最少的工厂生产, 哪种品牌都有可能。

```
vector<unique_ptr<CarFactory>> factories;

// Create 3 Ford factories and 1 Toyota factory.
factories.push_back(make_unique<FordFactory>());
factories.push_back(make_unique<FordFactory>());
factories.push_back(make_unique<FordFactory>());
factories.push_back(make_unique<ToyotaFactory>());

// To get more interesting results, preorder some cars.
factories[0]->requestCar();
factories[0]->requestCar();
factories[1]->requestCar();
factories[3]->requestCar();
```

```
// Create a factory that automatically selects the least busy
// factory from a list of given factories.
LeastBusyFactory leastBusyFactory { move(factories) };

// Build 10 cars from the least busy factory.
for (size_t i { 0 }; i < 10; i++) {
    auto theCar { leastBusyFactory.requestCar() };
    cout << theCar->info() << endl;
}
```

如果执行上述代码，程序将打印每辆汽车的生产商。

```
Ford
Ford
Ford
Ford
Ford
Ford
Ford
Ford
Ford
Ford
Ford
```

结果完全是可预测的，因为循环会依次迭代各个工厂。但是，如果有多个经销商请求提供汽车，每个工厂的当前状态就不好预测了。

33.3.4 工厂的其他类型

工厂也可以在单个类中实现，而不是在类层次中实现。在这种情况下，工厂的单个 `create()` 方法接收一个类型或字符串参数，它从中决定创建哪个对象，而不是将该工作委托给具体的子类。

33.3.5 工厂的其他用法

抽象工厂模式的使用范围并非仅限于对现实工厂的建模。例如，考虑字处理程序，需要支持多种语言的文档，每个文档都使用一种语言。在字处理程序的许多方面，所选的文档语言需要不同的支持：文档中使用的字符集(是否需要重音字符)、拼写检查器、词库和文档显示方式等。可通过编写抽象基类 `LanguageFactory`，以及每种相关语言的具体工厂，如 `EnglishLanguageFactory` 和 `FrenchLanguageFactory`，使用工厂设计清晰的字处理程序。当用户为文档指定语言时，程序使用适当的 `LanguageFactory` 创建特定域语言的特定功能的实例。例如，调用工厂中的 `createSpellchecker()` 方法来创建特定域语言的拼写检查器。然后，它用新构造的新语言拼写检查器来替换附加在文档中的上一种语言的当前拼写检查器。

工厂方法模式的另一种用法是作为 `pimpl` 习惯用法的替代，这在第 9 章“掌握类和对象”中讨论过。它在公共接口和所提供功能的具体实现之前提供了一堵墙。工厂方法模式的使用如下所示。首先，下面是暴露的公有方法：

```
// Public interface (to be included in the rest of the program,
// shared from a library, ...)
class Foo
{
public:
    virtual ~Foo() = default;           // Always a virtual destructor!
    static unique_ptr<Foo> create();     // Factory method.
```



```

        // Public functionality...
        virtual void bar() = 0;
    protected:
        Foo() = default;                                // Protected default constructor.
};

```

接下来，实现是对外界隐藏的：

```

// Implementation
class FooImpl : public Foo
{
    public:
        void bar() override { /* ... */ }
};
unique_ptr<Foo> Foo::create()
{
    return make_unique<FooImpl>();
}

```

任何需要 Foo 实例的客户端代码都可以创建这样的实例：

```
auto fooInstance { Foo::create() };
```

33.4 适配器模式

有时，类给出的抽象不适合当前的设计，不能更改。在这种情况下，可以构建一个适配器或包装类。适配器提供其余代码使用的抽象，充当期望的抽象和实际底层代码之间的桥梁。有两个主要的用例：

- 通过重用一些已有的实现来实现某个接口。在这个用例中，适配器通常在幕后创建实现的实例。
- 允许通过新接口使用现有功能。在这个用例中，适配器通常在构造函数中接收底层对象的实例。

第 18 章“标准库容器”讨论了标准库如何使用适配器模式以其他容器(如 deque 和 list)的形式实现栈和队列等容器。

33.4.1 示例：适配 Logger 类

在这个适配器模式示例中，假设有一个基本的 Logger 类。类定义如下：

```

export class Logger
{
    public:
        enum class LogLevel {
            Error,
            Info,
            Debug
        };

        Logger();
        virtual ~Logger() = default; // Always a virtual destructor!
        void log(LogLevel level, const std::string& message);
    private:

```

```

        // Converts a log level to a human readable string.
        std::string_view getLogLevelString(LogLevel level) const;
};

```

Logger 类有一个构造函数，它向标准控制台输出一行文本；还有一个 `log()` 方法，它将给定消息写到控制台，在消息前加日志级别。实现代码如下：

```

Logger::Logger()
{
    cout << "Logger constructor" << endl;
}

void Logger::log(LogLevel level, const string& message)
{
    cout << format("{}: {}", getLogLevelString(level), message) << endl;
}

string_view Logger::getLogLevelString(LogLevel level) const
{
    // See the Dependency Injection logger earlier in this chapter.
}

```

围绕这个基本 **Logger** 类编写包装类的一个原因是为了更改其接口。或许，你对日志级别不感兴趣，在调用 `log()` 方法时只想使用一个参数，即实际消息。你也可能想要更改接口，使 `log()` 方法接收 `std::string_view` (而非 `std::string`) 参数。

33.4.2 实现适配器

要实现适配器，首先要为底层功能定义一个新的接口。这个新的接口称为 `NewLoggerInterface`，如下所示：

```

export class NewLoggerInterface
{
public:
    virtual ~NewLoggerInterface() = default; // Always virtual destructor!
    virtual void log(std::string_view message) = 0;
};

```

该类是一个抽象类，它声明了想为新的 **Logger** 类使用的接口。该接口只定义了一个抽象方法 `log()`，它只接收一个 `string_view` 类型的实参。

下一步是编写实际的新 **Logger** 类 `NewLoggerAdaptor`，它实现了 `NewLoggerInterface`，使其具有你所设计的接口。该实现包装 **Logger** 实例，并且使用了组合方式。

```

export class NewLoggerAdapter : public INewLoggerInterface
{
public:
    NewLoggerAdapter();
    void log(std::string_view message) override;
private:
    Logger m_logger;
};

```

新 **Logger** 类的构造函数将一行内容写入标准输出，以跟踪调用了哪个构造函数。代码接着转发

所包装的 `Logger` 实例的 `log()` 方法调用, 从而实现 `NewLoggerInterface` 的 `log()` 方法。在这个调用中, 将给定的 `string_view` 转换为字符串, 将日志级别硬编码为 `Info`:

```
NewLoggerAdapter::NewLoggerAdapter()
{
    cout << "NewLoggerAdapter constructor" << endl;
}

void NewLoggerAdapter::log(string_view message)
{
    m_logger.log(Logger::LogLevel::Info, message.data());
}
```

33.4.3 使用适配器

由于适配器用于为底层功能提供更恰当的接口, 它们的使用应当直接明了, 用于特定目的。对于前面的实现, 下面的代码段为日志功能使用新的简化接口:

```
NewLoggerAdaptor logger;
logger.log("Testing the logger.");
```

将生成如下输出:

```
Logger constructor
NewLoggerAdaptor constructor
INFO: Testing the logger.
```

33.5 代理模式

有几种模式能将类的抽象与底层表示分离, 代理模式便是其中的一种。代理对象是实际对象的替代者。如果使用实际对象很费时, 或无法使用, 通常会使用代理对象。以文档编辑器为例。一个文档可能包含多个大对象(如图像)。不是在打开文档时加载所有这些图像, 文档编辑器可以用图像代理替代所有图像。这些代理不会立即加载图像。仅当用户在文档中滚动并到达图像位置时, 文档编辑器才会要求图像代理绘制自身。那时, 代理会将工作委托给实际的图像类, 由它们加载图像。

代理也可以用来屏蔽客户端的某些功能, 同时确保客户端甚至不能使用强制转换来绕过屏蔽。

33.5.1 示例: 隐藏网络连接问题

假设有一款网络游戏, `Player` 类表示 Internet 上参加游戏的玩家。`Player` 类包含 instant messaging(即时消息)等功能, 此类功能需要网络连接。如果一个玩家的网络连接太慢或失去响应, 表示那个玩家的 `Player` 对象将不能再收到即时消息。

由于不想将网络问题暴露给用户, 有必要使用一个独立的类来隐藏 `Player` 对象的网络部分。`PlayerProxy` 对象将替代实际的 `Player` 对象。每个客户端始终将 `PlayerProxy` 类用作实际 `Player` 类的门卫, 或当 `Player` 类不可用时替换为 `PlayerProxy` 类。网络故障期间, `PlayerProxy` 对象仍然显示玩家的姓名和最近状态, 当原始 `Player` 对象不可用时将继续运行。因此, 代理类隐藏了底层 `Player` 类的一些“令人不快”的语义。

33.5.2 实现代理

定义 `IPlayer` 接口时，首先要包含 `Player` 的公有接口。

```
class IPlayer
{
public:
    virtual ~IPlayer() = default; // Always virtual destructor.
    virtual string getName() const = 0;
    // Sends an instant message to the player over the network and
    // returns the reply as a string.
    virtual string sendInstantMessage(string_view message) const = 0;
};
```

`Player` 类定义将如下所示。想象一下，`sendInstantMessage()`方法需要连接到网络才能正常工作，如果网络连接断开，则会引发异常。

```
class Player : public IPlayer
{
public:
    string getName() const override;
    // Network connectivity is required.
    // Throws an exception if network connection is down.
    string sendInstantMessage(string_view message) const override;
};
```

`PlayerProxy` 也实现 `IPlayer` 接口，并包含另一个 `IPlayer` 实例(真正的 `Player`):

```
class PlayerProxy : public IPlayer
{
public:
    // Create a PlayerProxy, taking ownership of the given player.
    PlayerProxy(unique_ptr<IPlayer> player);
    string getName() const override;
    // Network connectivity is optional.
    string sendInstantMessage(string_view message) const override;
private:
    unique_ptr<IPlayer> m_player;
};
```

构造函数对给定的 `IPlayer` 具有所有权:

```
PlayerProxy::PlayerProxy(unique_ptr<IPlayer> player)
    : m_player { move(player) }
{
}
```

`getName()`方法只是转发给底层的 `player`:

```
string PlayerProxy::getName() const { return m_player->getName(); }
```

`PlayerProxy` 的 `sendInstantMessage()`方法的实现检查网络连接情况，返回默认字符串或转发请求。这隐藏了一个事实，即当网络连接断开时，底层的 `player` 对象上的 `sendInstantMessage()`方法会引发异常。

```
string PlayerProxy::sendInstantMessage(string_view message) const
{
    if (hasNetworkConnectivity()) { return m_player->sendInstantMessage(message); }
    else { return "The player has gone offline."; }
}
```

33.5.3 使用代理

如果代码编写得当，其用法与其他任何对象无异。在 `PlayerProxy` 示例中，使用代理的代码完全没必要了解代理的存在。当玩家赢时，将调用以下函数，该函数可处理实际 `Player` 或 `PlayerProxy`。代码能以相同的方式处理两种情形，因为代理肯定会产生有效结果。

```
bool informWinner(const IPlayer& player)
{
    auto result { player.sendInstantMessage("You have won! Play again?") };
    if (result == "yes") {
        cout << player.getName() << " wants to play again." << endl;
        return true;
    } else {
        // The player said no, or is offline.
        cout << player.getName() << " does not want to play again." << endl;
        return false;
    }
}
```

33.6 迭代器模式

迭代器模式提供了一种机制，以将算法或操作与它们所操作的数据分开。基本上，迭代器允许算法导航数据结构，而不需要知道数据的实际结构。面向对象编程的基本原理是将对象数据和行为组合在一起，并在数据上执行操作。初看起来，迭代器模式与上述原理是对立的。从某种意义上讲，这种异议是有道理的，但迭代器模式并不提倡从对象删除基本行为。实际上，迭代器模式解决了两个常见问题：数据和行为的紧密耦合。

如果将数据和行为紧密耦合，第一个问题是排斥可用于不同对象(这些对象未必位于同一个类层次结构中)的通用算法。为编写通用算法，需要使用一些标准机制导航/访问数据结构的内容，而不需要了解具体结构。

第二个问题在于有时难以添加新的行为。至少需要访问数据对象的源代码。但是，如果相关的对象层次结构属于无法更改的第三方框架或库，该怎么办？如果不需要修改保存数据的类的原始对象层次结构，就能添加作用于数据的算法和操作，效果当然更好。

从概念上讲，迭代器提供了一种机制，使操作或算法按顺序访问元素的容器。`iterator` 一词来源于 `iterate`，意即“重复”，因为它们序列中反复前移，以便到达新元素。在标准库中，通用算法使用迭代器来访问所操作容器的元素。通过定义标准的迭代器接口，标准库允许编写算法，在提供迭代器(具有适当接口)的任何容器上执行操作。甚至可以为单个数据结构提供几个不同的迭代器。这允许算法以不同的方式浏览数据，例如，树型数据结构的自顶向下和自底向上遍历。因此，迭代器允许编写泛型算法，它可以遍历数据结构的内容，而不需要了解该结构的任何信息。图 33-6 中，迭代器担当中央协调器，操作依赖于迭代器，数据对象提供迭代器。

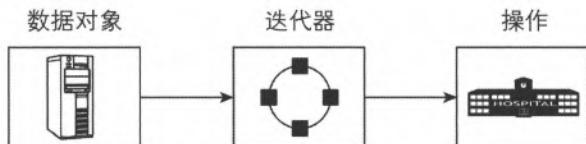


图 33-6 迭代器担当中央协调器

第 25 章“自定义和扩展标准库”列举了一个详细的示例，演示了如何为符合标准库要求的类实现迭代器，相应的迭代器可供通用标准库算法使用。

33.7 观察者模式

可利用观察者模式，使对象/观察者获得可观察对象（主题）的通知。具体的观察者被它们感兴趣的、可观察对象注册。当可观察对象的状态发生变化时，它会通知所有注册的观察者。使用观察者模式的主要好处是它减少了耦合。可观察对象类不需要知道正在观察它的具体观察者类型。

33.7.1 示例：从主题中暴露事件

该示例由具有可变数量参数的通用目的事件组成。主题可以暴露特定的事件，例如当主题的数据被修改时引发的事件，当主题的数据被删除时引发的事件等。

33.7.2 实现观察者

首先，定义一个可变类型的类模板 `Event`。可变参数类模板将在第 26 章“高级模板”中讨论。这个类保存具有可变数量参数的函数的映射。提供重载的 `operator+=`，以函数的形式注册一个新的观察者，该函数在触发事件时会被通知。该运算符返回一个 `EventHandle`，随后可以传递给重载的 `operator=` 来注销该观察者。这个 `EventHandle` 只是一个随着每个注册观察者而增加的数字。最后，`raise()` 方法通知所有注册的已经被触发的观察者事件。

```
using EventHandle = size_t;

template <typename... Args>
class Event
{
public:
    virtual ~Event() = default; // Always a virtual destructor!

    // Adds an observer. Returns an EventHandle to unregister the observer.
    [[nodiscard]] EventHandle operator+=(function<void(Args...)> observer)
    {
        auto number { ++m_counter };
        m_observers[number] = observer;
        return number;
    }

    // Unregisters the observer pointed to by the given handle.
    Event& operator-=(EventHandle handle)
    {
        m_observers.erase(handle);
        return *this;
    }
}
```

```

        // Raise event: notifies all registered observers.
        void raise(Args... args)
        {
            for (auto& observer : m_observers) { (observer.second)(args...); }
        }
    private:
        size_t m_counter { 0 };
        map<EventHandle, function<void(Args...)>> m_observers;
};

```

任何想要在它的观察者上暴露事件的对象可以注册它们自己，只需要暴露 `Event` 可变类型类模板的实例即可。由于使用可变类型的类模板，可以使用任意数量的参数创建 `Event` 实例。这允许可观察对象向观察者传递任何有关信息。下面是一个示例：

```

class ObservableSubject
{
public:
    auto& getEventDataModified() { return m_eventDataModified; }
    auto& getEventDataDeleted() { return m_eventDataDeleted; }

    void modifyData()
    {
        // ...
        getEventDataModified().raise(1, 2.3);
    }

    void deleteData()
    {
        // ...
        getEventDataDeleted().raise();
    }
private:
    Event<int, double> m_eventDataModified;
    Event<> m_eventDataDeleted;
};

```

33.7.3 使用观察者

下面是一些测试代码，演示了如何使用观察者模式。假设有以下可以处理修改事件的独立全局函数 `modified()`：

```
void modified(int, double) { cout << "modified" << endl; }
```

假设也有一个能够处理修改事件的 `Observer` 类：

```

class Observer
{
public:
    Observer(ObservableSubject& subject) : m_subject { subject }
    {
        m_subjectModifiedHandle = m_subject.getEventDataModified() +=
            [this](int i, double d) { onSubjectModified(i, d); };
    }
};

```

```

        virtual ~Observer()
        {
            m_subject.getEventDataModified() -= m_subjectModifiedHandle;
        }
    private:
        void onSubjectModified(int, double)
        {
            cout << "Observer::onSubjectModified()" << endl;
        }
        ObservableSubject& m_subject;
        EventHandle m_subjectModifiedHandle;
};

```

最后，可以构造 `ObservableSubject` 实例并注册一些观察者：

```

ObservableSubject subject;

auto handleModified { subject.getEventDataModified() += modified };
auto handleDeleted {
    subject.getEventDataDeleted() += [] { cout << "deleted" << endl; } };
Observer observer { subject };

subject.modifyData();
subject.deleteData();

cout << endl;

subject.getEventDataModified() -= handleModified;
subject.modifyData();
subject.deleteData();

```

输出如下所示：

```

modified
Observer::onSubjectModified()
deleted
Observer::onSubjectModified()
deleted

```

33.8 装饰器模式

顾名思义，装饰器模式装饰对象。我们经常将它称为包装器。装饰器模式用于在运行时添加或更改对象行为。装饰器很像派生类，但是可以动态地改变行为。折中之处是，与派生类相比，装饰器更改行为的方法更少，因为装饰器不能覆盖某些辅助方法。另一方面，装饰器的一个主要好处是，它们可以很容易地组合在一起，来精确地完成需要的任务，不必为每个组合都编写派生类。

例如，如果正在解析一个数据流，并且到达表示图像的数据处，可用 `ImageStream` 对象临时装饰流对象。`ImageStream` 构造函数将流对象作为参数，而且知道如何解析图像。解析完图像后，可继续使用原始对象解析流的其余部分。`ImageStream` 被用作装饰器，因为它给现有对象(流)添加了新功能(解析图像)。

33.8.1 示例：在网页中定义样式

网页是用简单的HTML(HyperText Markup Language, 超文本标记语言)文本结构编写的。在HTML中, 可使用样式标记设置文本的格式, 比如使用和设置粗体, 使用<I>和</I>设置斜体。下面的HTML 代码行将消息显示为粗体:

```
<B>A party? For me? Thanks!</B>
```

下面的代码行将消息显示为粗体加斜体:

```
<I><B>A party? For me? Thanks!</B></I>
```

HTML 中的段落包裹在<P>和</P>标签中。见示例:

```
<P>This is a paragraph.</P>
```

假设你正在编写一个HTML 编辑应用程序。用户能输入一段文字, 并应用一种或多种样式。可将每种段落确定为一个新的派生类, 如图33-7所示。但那样的话, 设计将十分笨拙, 在添加新样式时, 复杂度将快速增加。

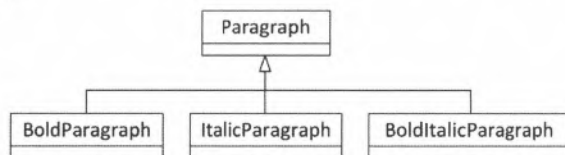


图 33-7 将每种段落确定为一个新的派生类

另一种思路是: 不将带样式的段落视为段落类型, 而视为装饰段落, 从而得到如图33-8所示的结果。其中, ItalicParagraph在BoldParagraph上操作, BoldParagraph又在Paragraph上操作。对象的递归装饰将样式嵌套在代码中, 就像将它们嵌套在HTML中一样。

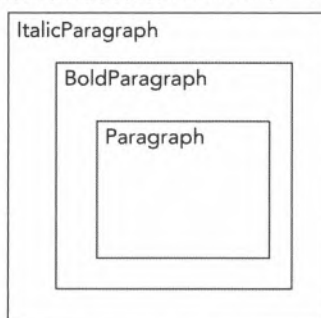


图 33-8 视为装饰段落的结果

33.8.2 装饰器的实现

首先需要有一个IParagraph接口:

```

class IParagraph
{
public:
    virtual ~IParagraph() = default; // Always a virtual destructor!
    virtual std::string getHTML() const = 0;
};
  
```

用 Paragraph 类实现这个 IParagraph 接口:

```
class Paragraph : public IParagraph
{
public:
    Paragraph(std::string text) : m_text { std::move(text) } {}
    std::string getHTML() const override { return "<P>" + m_text + "</P>"; }
private:
    std::string m_text;
};
```

要使用一种或多种样式装饰 Paragraph, 需要带样式的 IParagraph 类, 每一个都从现有的 IParagraph 构建。这样, 它们都可装饰 Paragraph 或带样式的 IParagraph。BoldParagraph 类从 IParagraph 派生, 并实现 getHTML()。关键是因为只想将其用作装饰器, 所以让它的单个公有非拷贝构造函数接收 IParagraph 的 const 引用。

```
class BoldParagraph : public IParagraph
{
public:
    BoldParagraph(const IParagraph& paragraph) : m_wrapped { paragraph } {}
    std::string getHTML() const override {
        return "<B>" + m_wrapped.getHTML() + "</B>"; }
private:
    const IParagraph& m_wrapped;
};
```

ItalicParagraph 类几乎是相同的:

```
class ItalicParagraph : public IParagraph
{
public:
    ItalicParagraph(const IParagraph& paragraph) : m_wrapped { paragraph } {}

    std::string getHTML() const override {
        return "<I>" + m_wrapped.getHTML() + "</I>"; }
private:
    const IParagraph& m_wrapped;
};
```

33.8.3 使用装饰器

在用户看来, 装饰器模式极富吸引力, 因为它应用起来十分方便, 而且一旦应用, 便是透明的。用户根本不必知道已经应用了装饰器。BoldParagraph 的行为与 Paragraph 类似。

下面的简单示例创建和输出段落, 首先是粗体, 然后是粗体加斜体:

```
Paragraph p { "A party? For me? Thanks!" };
// Bold
std::cout << BoldParagraph(p).getHTML() << std::endl;
// Bold and Italic
std::cout << ItalicParagraph{BoldParagraph(p)}.getHTML() << std::endl;
```

输出如下所示:

```
<B><P>A party? For me? Thanks!</P></B>
<I><B><P>A party? For me? Thanks!</P></B></I>
```

33.9 责任链模式

在执行特定操作时，若想使多个对象参与进来，则可使用责任链。责任链最常用于事件处理。许多现代应用程序(特别是图形用户界面)都被设计为一系列事件和响应。例如，当用户单击 File 菜单，再选择 Open 时，就会发生 Open 事件。当用户将鼠标移到绘图程序的可绘制区域时，会持续生成 mouse move 事件。如果用户按下一个鼠标按键，将生成这个按键的 mouse down 事件；此后，程序开始注意到这个 mouse down 事件，允许用户绘制一些对象，并持续到 mouse up 事件发生为止。每个操作系统都有自己的命名和使用事件的方式，但总体想法是相同的：当事件发生时，会以某种方式与程序通信，程序接着采用适当行动。

如你所知，C++并没有用于图形编程的内置工具，也没有事件、事件传输或事件处理的概念。责任链是一种合理的事件处理方法，使不同对象有机会处理某些事件。

33.9.1 示例：事件处理

考虑一个绘图程序，它是一个带有窗口的应用程序，可以在窗口中绘制形状。用户可以在程序窗口的某个地方按下鼠标按钮。如果发生这种情况，程序应该确定用户是否单击了一个形状。如果是，则要求形状处理鼠标按钮向下事件。如果它决定不需要处理该事件，它就将该事件传递给窗口，窗口会接手该事件。如果窗口也对该事件不感兴趣，则将其转发给程序本身，程序本身是链中处理该事件的最终对象。这就是一个责任链，因为每个处理程序可处理事件，或将事件上传给链中的下一个处理程序。

33.9.2 实现责任链

假设所有可能的事件都在 enum class 中定义如下：

```
enum class Event {
    LeftMouseButtonDown,
    LeftMouseButtonUp,
    RightMouseButtonDown,
    RightMouseButtonUp
};
```

接下来，定义了下面的 Handler mixin 类：

```
class Handler
{
public:
    virtual ~Handler() = default;
    explicit Handler(Handler* nextHandler) : m_nextHandler { nextHandler } {}
    virtual void handleMessage(Event message)
    {
        if (m_nextHandler) { m_nextHandler->handleMessage(message); }
    }
private:
    Handler* m_nextHandler;
};
```

接下来，Application、Window 和 Shape 类是具体的处理程序，它们都派生自 Handler 类。在这个示例中，Application 只处理 RightMouseButtonDown 消息，Window 只处理 LeftMouseButtonUp 消息，Shape 只处理 LeftMouseButtonDown 消息。如果任何一个程序接收到它不知道的消息，它就调用链中

的下一个处理程序。

```
class Application : public Handler
{
public:
    explicit Application(Handler* nextHandler) : Handler { nextHandler } { }

    void handleMessage(Event message) override
    {
        cout << "Application::handleMessage()" << endl;
        if (message == Event::RightMouseButtonDown) {
            cout << " Handling message RightMouseButtonDown" << endl;
        } else { Handler::handleMessage(message); }
    }
};

class Window : public Handler
{
public:
    explicit Window(Handler* nextHandler) : Handler { nextHandler } { }

    void handleMessage(Event message) override
    {
        cout << "Window::handleMessage()" << endl;
        if (message == Event::LeftMouseButtonUp) {
            cout << " Handling message LeftMouseButtonUp" << endl;
        } else { Handler::handleMessage(message); }
    }
};

class Shape : public Handler
{
public:
    explicit Shape(Handler* nextHandler) : Handler { nextHandler } { }

    void handleMessage(Event message) override
    {
        cout << "Shape::handleMessage()" << endl;
        if (message == Event::LeftMouseButtonDown) {
            cout << " Handling message LeftMouseButtonDown" << endl;
        } else { Handler::handleMessage(message); }
    }
};
```

33.9.3 使用责任链

上一节中实现的责任链可以通过以下方法测试：

```
Application application { nullptr };
Window window { &application };
Shape shape { &window };

shape.handleMessage(Event::LeftMouseButtonDown);
cout << endl;

shape.handleMessage(Event::LeftMouseButtonUp);
```

```
cout << endl;

shape.handleMessage(Event::RightMouseButtonDown);
cout << endl;

shape.handleMessage(Event::RightMouseButtonUp);
```

输出如下:

```
Shape::handleMessage()
    Handling message LeftMouseButtonDown

Shape::handleMessage()
Window::handleMessage()
    Handling message LeftMouseButtonUp

Shape::handleMessage()
Window::handleMessage()
Application::handleMessage()
    Handling message RightMouseButtonDown

Shape::handleMessage()
Window::handleMessage()
Application::handleMessage()
```

当然,在真实的应用程序中,必须有一些其他类,将事件分派到正确对象上,例如,在正确的对象上调用 `handleMessage()`。由于任务因框架或平台而异,差别很大,下例将显示处理 `mouse down` 事件的伪代码,以此替代平台专用的 C++ 代码:

```
MouseLocation location { getLocation() };
Shape* clickedShape { findShapeAtLocation(location) };
if (clickedShape) {
    clickedShape->handleMessage(Event::LeftMouseButtonDown);
} else {
    window.handleMessage(Event::LeftMouseButtonDown);
}
```

链式方法十分灵活,具有吸引人的面向对象层次结构。缺点在于程序员需要付出的努力较多。如果忘记从责任链的派生类向上到达基类,事件将丢失。更糟糕的是,如果到达错误的类,最终将出现无限循环。

33.10 单例模式

单例模式是最简单的设计模式。`singleton` 一词的普通意思是“一类”或“单个”,在编程中的含义与此类似。单例模式是一种策略,强制程序中只存在类的一个实例(正好一个)。为类应用单例模式将确保只创建类的一个对象。单例模式也指出,可从全局(即程序的任何位置)访问一个对象。程序员通常将遵循单例模式的类称为单例类。

如果程序要求某个类正好有一个实例,那么应当使用单例模式。

但必须知道,单例模式具有诸多缺点。如果有多个单例,往往在程序启动时难以确保按正确顺序初始化这些单例;在程序关闭期间,也很难确保在调用者需要单例时,单例仍然存在。此外,单例类引入了隐藏的依赖性,会导致紧耦合,使单元测试变得更复杂。例如,在单元测试中,可能想要为一个访问网络或数据库的单例对象编写一个模拟版本(见第 30 章“熟练掌握测试技术”),但是考虑到

典型的单例实现的特点,这很难办到。一种更合理的设计模式是本章前面讨论的依赖注入。使用依赖注入,可为提供的每个服务创建接口,将组件需要的接口注入组件。与典型的单例相比,依赖注入允许 mock(存根版本),允许在后期更方便地引入多个实例,允许采用更复杂的方式(如使用工厂模式等)构建单个对象。不过,这里还是要讨论单例模式,因为仍会在遗留代码库中遇到它。

警告:

避免在新代码中使用单例模式,因为它有很多问题。选择其他模式,比如依赖注入。

33.10.1 日志记录机制

许多应用程序都有 `Logger` 类(日志记录器),这个类负责将状态信息、调试数据以及错误写入中心位置。理想的 `Logger` 类具有以下特征:

- 任何时间都可用
- 易于使用
- 只有一个实例

单例模式可以用来完成这些需求。但是在新的代码中,建议避免引入新的单例。

33.10.2 实现单例

在 C++ 中,可通过两种基本方式实现单例。第一种方式是使用只有静态方法的类。这样的类不需要实例,可从任意位置访问。这种方式的问题在于缺少内置的构建和析构机制。但从技术角度看,全部使用静态方法的类不是真正的单例,而是静态类或 `nothington`(一个新术语)。术语“单例”是指类正好有一个实例。如果所有方法都是静态的,从未对类进行实例化,就不能称之为“单例”。本节后面将进一步讨论。

第二种方式是使用访问控制级别来控制类的实例的创建和访问。这是真正的单例,本章将通过一个简单的 `Logger` 类进一步讨论这种方式。提供与本章前面讨论的依赖注入 `Logger` 类似的特性。

要在 C++ 中构建真正的单例,可使用访问控制机制以及 `static` 关键字。真正的 `Logger` 对象在运行时是存在的,只允许类的一个对象被实例化。客户端始终可通过静态方法 `instance()` 来访问相应的对象。`Logger` 类定义如下所示:

```
export class Logger final
{
    public:
        enum class LogLevel {
            Error,
            Info,
            Debug
        };

        // Sets the name of the log file.
        // Note: needs to be called before the first call to instance()!
        static void setLogFilename(std::string_view logFilename);

        // Returns a reference to the singleton Logger object.
        static Logger& instance();

        // Prevent copy/move construction.
        Logger(const Logger&) = delete;
        Logger(Logger&&) = delete;
```

```

// Prevent copy/move assignment operations.
Logger& operator=(const Logger&) = delete;
Logger& operator=(Logger&&) = delete;

// Sets the log level.
void setLogLevel(LogLevel level);

// Logs a single message at the given log level.
void log(std::string_view message, LogLevel logLevel);

private:
// Private constructor and destructor.
Logger();
~Logger();

// Converts a log level to a human readable string.
std::string_view getLogLevelString(LogLevel level) const;

static std::string ms_logFilename;
std::ofstream m_outputStream;
LogLevel m_logLevel { LogLevel::Error };
};

```

该实现基于 Scott Meyer 的单例模式。这意味着，`instance()`方法包含 `Logger` 类的本地静态实例。C++ 确保以线程安全的方式初始化这个本地静态实例，这样，在这个单例模式的版本中，不需要手动执行任何线程同步。这些就是所谓的 `magic static`。注意，只有初始化是线程安全的！如果多个线程准备调用 `Logger` 类的方法，那么也必须使 `Logger` 方法本身变得线程安全。第 27 章“C++多线程编程”详细讨论了如何使用同步机制使类变得线程安全。

`Logger` 类的实现相当简单。一旦打开日志文件，就会写入每条日志消息，在消息的前面加上日志级别。创建和销毁 `instance()`方法中的 `Logger` 类的静态实例时，会自动调用构造函数和析构函数。由于构造函数和析构函数是私有的，外部代码不能创建或删除 `Logger` 实例。

下面是 `setLogFilename()`、`instance()`方法和构造函数的实现。其他方法的实现与本章前面的依赖注入 `Logger` 示例相同。

```

void Logger::setLogFilename(string_view logFilename)
{
    ms_logFilename = logFilename.data();
}

Logger& Logger::instance()
{
    static Logger instance; // Magic static.
    return instance;
}

Logger::Logger()
{
    m_outputStream.open(ms_logFileName, ios_base::app);
    if (!m_outputStream.good()) {
        throw runtime_error { "Unable to initialize the Logger!" };
    }
}

```

33.10.3 使用单例

可测试单例类 `Logger`，如下所示：

```
// Set the log filename before the first call to instance().
Logger::setLogFilename("log.out");
// Set log level to Debug.
Logger::instance().setLogLevel(Logger::LogLevel::Debug);
// Log some messages.
Logger::instance().log("test message", Logger::LogLevel::Debug);
// Set log level to Error.
Logger::instance().setLogLevel(Logger::LogLevel::Error);
// Now that the log level is set to Error, logging a Debug
// message will be ignored.
Logger::instance().log("A debug message", Logger::LogLevel::Debug);
```

执行后，`log.out` 文件包含的内容如下：

```
DEBUG: test message
Logger shutting down.
```

33.11 本章小结

本章介绍了使用设计模式将面向对象的概念组织成高级设计。维基百科对大量设计模式进行了讨论和分类(https://en.wikipedia.org/wiki/Software_design_pattern)。或许，这会使你投入全部时间用于查找可用于自己项目的特定设计模式。建议不要这样做，而是重点关注几个自己感兴趣的设计模式，并重点研究它们的开发方式，不要只是拘泥于分析类似设计模式的微小差别。引用一句谚语：“教会我一种设计模式，我将在一天的编码中受益；教会我如何创建设计模式，我将在一生的编码中受益。”

33.12 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试自己找到答案。

本章练习的概念与其他章节有所不同。下面的练习简要地介绍了一些新模式，并要求你对这些模式进行研究，以更多地了解它们。

练习 33-1 虽然本章讨论了一些很好的模式，当然还有更多可用的模式。其中一种模式是命令模式。它将一个或多个操作封装在对象中。该模式的一个主要用例是实现可撤销的操作。使用附录 B 中与模式相关的参考资料来研究和学习命令模式，或者参考 Wikipedia 文章 en.wikipedia.org/wiki/Software_design_pattern 来研究。

练习 33-2 另一个常用的模式是策略模式。使用此模式可以定义一系列在运行时可互换的算法。研究策略模式做更多了解。

练习 33-3 使用原型模式，可以指定不同类型的对象。这些对象可以通过构造这些对象的原型实例来创建。这些原型实例通常注册在某种注册表中。然后客户端向注册中心请求特定类型对象的原型，并随后克隆原型以供进一步使用。研究原型模式做更多了解。

练习 33-4 中介模式用于控制一组对象之间的交互。它提倡不同子系统间的松散耦合。研究中介模式做更多了解。

第34章

开发跨平台和跨语言的应用程序

本章内容

- 如何编写能在多个平台上运行的代码
- 如何混合使用不同的编程语言

从理论上讲，可将已经编译的 C++ 程序在各种计算平台上运行；已经对 C++ 进行严格的定义，确保在一个平台上编写 C++ 程序与在另一个平台上编写 C++ 程序类似。虽然对 C++ 语言已经实施了标准化，但在实际中编写专业级别的 C++ 程序时，平台差异问题终归会浮出水面。即使将开发范围限制在一个特定平台内，编译器的微小差异也会引出令人头痛的编程难题。本章分析在多平台和多编程语言环境中编程的复杂性。

本章第一部分分析 C++ 程序员遇到的与平台相关的问题。平台是构建开发系统和(或)运行时系统的所有细节的集合。例如，平台可以是：硬件是 Intel Core i7 处理器，操作系统是 Windows 10，编译器是 Microsoft Visual C++ 2019。平台也可以是：硬件是 PowerPC 处理器，操作系统是 Linux，编译器是 GCC 10.1。这两个平台都能编译和运行 C++ 程序，但它们之间差异明显。

本章第二部分介绍 C++ 如何与其他编程语言交互。尽管 C++ 是一门通用语言，但也不是适合完成所有工作的万能工具。可通过各种机制，将 C++ 与其他语言集成在一起，以更好地满足自己的需要。

34.1 跨平台开发

C++ 语言会遇到平台问题，原因是多方面的。C++ 是一门高级语言，C++ 标准并未指定某些低级细节。例如，C++ 标准并未定义内存中对象的布局，而留给编译器去处理。不同编译器可为对象使用不同的内存布局。C++ 面临的另一个挑战是：提供标准语言 and 标准库，却没有标准实现。C++ 编译器和库供应商对规范有不同解释，当从一个系统迁移到另一个系统时，会导致问题。最后，C++ 在语言所提供的标准中是有选择性的。尽管存在标准库，但程序经常需要并非由 C++ 语言或标准库提供的功能；功能通常来自第三方库或平台，而且差异很大。

34.1.1 架构问题

术语“架构”通常指运行程序的一个或一系列处理器。运行 Windows 或 Linux 的标准 PC 通常运

行在 x86 或 x64 架构上，较旧的 Mac OS 版本通常运行在 PowerPC 架构上。作为一门高级语言，C++ 会向你隐藏这些架构之间的区别。例如，Core i7 处理器的一条指令可能与 6 条 PowerPC 指令执行相同的功能。作为一名 C++ 程序员，不需要了解这种差异，甚至不必知道存在这种差异。使用高级语言的一个优点在于由编译器负责将代码转换为处理器的本地汇编代码格式。

不过，处理器差异有时会上升到 C++ 代码级别。下面首先讨论整数大小，如果编写跨平台代码，这将十分重要。其他的不常遇到，除非要执行十分低级的工作；虽然如此，但仍然需要知道它们的存在。

1. 整数大小

C++ 标准并未定义整数类型的准确大小。C++ 标准仅指出：

有 5 种标准的有符号整数类型：signed char、short int、int、long int 和 long long int。在这个列表中，后一种类型占用的存储空间大于或等于前一种类型。

C++ 标准的确进一步说明了这些类型的大小，但从未给出确切大小。实际大小取决于编译器。因此，如果要编写跨平台代码，就不能盲目地依赖这些类型。本章末尾的习题之一要求再进一步研究这个问题。

除了这些标准的整数类型，C++ 标准还定义了多种明确指定了大小的类型，其中一些类型是可选的，这些都定义在 <stdint> 头文件中。表 34-1 对此进行了概述。

表 34-1 C++ 标准定义的多种明确指定了大小的类型

类型	描述
int8_t int16_t int32_t int64_t	有符号整数，大小是 8、16、32 或 64 位。C++ 标准将这些类型定义为可选的，不过，大多数编译器都支持这些类型
int_fast8_t int_fast16_t int_fast32_t int_fast64_t	有符号整数，大小是 8、16、32 或 64 位。对于这些类型，编译器应当使用满足要求的最快的整数类型
int_least8_t int_least16_t int_least32_t int_least64_t	有符号整数，大小是 8、16、32 或 64 位。对于这些类型，编译器应当使用满足要求的最小整数类型
intmax_t	一种整数类型，大小是编译器支持的最大大小
intptr_t	一种整数类型，大小足以存储一个指针。C++ 标准将这种类型定义为可选的，不过，大多数编译器都支持这种类型

还有无符号的版本，如 uint8_t、uint_fast8_t 等。

注意：

编写跨平台代码时，建议用 <stdint> 类型替代基本整数类型。

2. 二进制兼容性

你可能已经知道, 为 Core i7 计算机编写和编译的程序不能在基于 PowerPC 的 Mac 上运行。这两个平台无法实现二进制兼容性, 因为处理器支持的不是同一组指令。在编译 C++ 程序时, 源代码将转换为计算机执行的二进制指令。二进制格式由平台(而非 C++ 语言)定义。

为支持不具备二进制兼容性的平台, 一种解决方案是在每个目标平台上使用编译器分别构建每个版本。

另一种解决方案是交叉编译(cross-compiling)。如果为开发使用平台 X, 但想使程序运行在平台 Y 和 Z 上, 可在平台 X 上使用交叉编译器, 为平台 Y 和 Z 生成二进制代码。

也可以使自己的程序成为开源程序。如果最终用户可得到源代码, 则可在用户自己的系统上对源代码进行本地编译, 针对本地计算机构建具有正确二进制格式的程序版本。如第 4 章“设计专业的 C++ 程序”所述, 开源软件已经日益流行。一个主要原因是它们允许程序员协作开发软件, 并增加可运行程序的平台数量。

3. 地址大小

当提到架构是 32 位时, 通常是说地址大小是 32 位或 4 字节。通常而言, 具有更大地址大小的系统可处理更多内存, 在复杂系统上的运行速度更快。

由于指针是内存地址, 自然与地址大小密切相关。许多程序员都认为, 指针的大小始终是 4 字节, 但这是错误的。例如, 考虑以下代码片段, 作用是输出指针的大小:

```
int *ptr { nullptr };
cout << "ptr size is " << sizeof(ptr) << " bytes" << endl;
```

如果在 32 位的 x86 系统上编译和运行程序, 输出将如下所示:

```
ptr size is 4 bytes
```

如果在 64 位的编译器上编译, 在 x64 系统上运行, 输出将如下所示:

```
ptr size is 8 bytes
```

在程序员看来, 上面显示的不同指针大小结果说明, 不能认为指针就是 4 字节。更普遍的是, 需要意识到, 大多数大小都不是 C++ 标准预先确定的。C++ 标准只是说, short integer 占用的空间小于或等于 integer, integer 占用的空间小于或等于 long integer。

指针的大小未必与整数大小相同。例如, 在 64 位平台上, 指针是 64 位, 但整数可能是 32 位。将 64 位指针强制转换为 32 位整数时, 将丢失 32 个关键位! C++ 标准在 <stdint> 中定义了 std::intptr_t 整数类型, 它的大小至少足以存储一个指针。根据 C++ 标准, 这种类型的定义是可选的, 但几乎所有编译器都支持它。

警告:

不要认为指针一定是 32 位或 4 字节。除非使用 std::intptr_t, 否则不要将指针强制转换为整数。

4. 字节顺序

所有现代计算机都以二进制形式存储数字, 但同一个数字在两个平台上的表示形式可能不同。这听起来是矛盾的, 但可以看到, 可通过两种合理方法读取数字。

现代大多数计算机都是字节可寻址的, 内存中的单个 slot 通常是一个字节。内存可寻址使这些计算机理想地用字节工作; 内存中的每个字节都有一个唯一的内存地址。C++ 中的数字类型通常是多个

字节。例如，`short` 是两字节。假设程序包含以下代码行：

```
short myShort { 513 };
```

数字 513 的二进制形式是 0000 0010 0000 0001。这个数字包含 16 位。由于 1 个字节有 8 位，因此计算机需要两个字节来存储该数字。由于每个内存地址包含 1 个字节，计算机需要将该数字分成多个字节。`short` 是两字节，该数字将被等分为两部分。数字的高部分放在高位字节，低部分放在低位字节。此处，高位字节是 0000 0010，低位字节是 0000 0001。

现在将数字分为“内存大小”部分。剩余的唯一问题是如何将它们存储在内存中。需要两个字节，但字节序是不确定的，事实上取决于相关系统的架构。

一种表示数字的方式是将高位字节首先放入内存，接着将低位字节放入内存。这种策略被称为大端序，因为数字的较大部分首先放入。`PowerPC` 和 `SPARC` 处理器使用大端序。其他一些处理器(如 `x86`)按相反顺序放置字节，首先将低序字节放入内存。这种策略被称为小端序，因为数字的较小部分首先放置。架构可能选择其中一种方式，通常取决于后向兼容性。令人好奇的是，术语“大端”和“小端”的出现时间早于现代计算机几百年。`Jonathan Swift` 于 18 世纪写的小说《格列佛游记》中，就怎样打破鸡蛋而分成了大端派与小端派。

无论特定架构使用哪种字节序，程序都可以继续使用数字值，不需要在意计算机使用的是大端序还是小端序。仅当在架构之间移动数据时才考虑字节序。例如，如果正在网络上发送二进制数据，那么需要考虑另一个系统的字节序。一种解决方案是使用标准网络字节序，即总是使用大端序。因此，在网络中发送数据前，将它们转换为大端序，当从网络上接收数据时，可将大端序转换为系统使用的字节序。

同样，如果正在将二进制数据写入文件，那么可能需要考虑：如果在使用反向字节序的系统中打开这个文件，会发生什么情况？

 C++20 标准库引入了 `<bit>` 中定义的 `std::endian` 枚举类型，它可以用来确定当前系统是大端系统还是小端系统。下面的代码片段输出了系统的本地字节顺序：

```
switch (endian::native)
{
    case endian::little:
        cout << "Native ordering is little-endian." << endl;
        break;
    case endian::big:
        cout << "Native ordering is big-endian." << endl;
        break;
}
```

34.1.2 实现问题

在编写 C++ 编译器时，编写者尝试遵循 C++ 标准。但是，C++ 标准的长度超过 1000 页，包含刻板的论述、语言语法和示例。即使两个人都按这个标准实现编译器，也不大可能以相同的方式解读预先规定的信息中的每一部分，也无法顾及每种边缘情况。因此，编译器中存在 bug。

1. 编译器的处理和扩展

并不存在查找或避免编译器 bug 的简单方法。最好的做法是一直保持编译器的版本是最新的，也可能需要订阅编译器的邮件列表或新闻组。

如果怀疑自己遇到了编译器 bug，可以在网上简单地搜索自己看到的错误消息或条件，尝试找到

变通方法或补丁。

众所周知，编译器存在的一个重要问题是难以跟上 C++ 标准最新添加或更新的语言功能。不过近几年，几个主流编译器的供应商已能较快地为最新功能添加支持。

需要了解的另一个问题是，编译器经常包含程序员注意不到的独特语言扩展。例如，VLA(Variable-Length stack-based Array，基于变长栈的数组)并非 C++ 语言的一部分，而是 C 语言的一部分。有些编译器既支持 C 标准，又支持 C++ 标准，允许在 C++ 代码中使用 VLA。g++ 就是这样一种编译器。在 g++ 编译器中，以下代码可以如期编译和运行：

```
int i { 4 };
char myStackArray[i]; // Not a standard language feature!
```

一些编译器扩展是有用的，但如果在某个时间点有机会切换编译器，可以看一下编译器是否有 strict 模式；在该模式下，会避免使用此类扩展。例如，若将 -pedantic 标志传给 g++ 来编译上面的代码，将看到以下警告消息：

```
warning: ISO C++ forbids variable length array 'myStackArray' [-Wvla]
```

C++ 规范允许由编译器定义的某些类型的语言扩展，这是通过 #pragma 机制实现的。#pragma 是一条预处理指令，其行为由实现定义。如果实现不理解该指令，则会忽略它。例如，一些编译器允许程序员使用 #pragma 临时关闭编译器警告。

2. 库实现

编译器很可能包含 C++ 标准库的实现。由于标准库用 C++ 编写，因此并非必须使用与编译器打包在一起的实现。可以使用经过性能优化的第三方标准库，甚至可自行编写。

当然，标准库实现面临着与编译器编写者同样的问题：同一个 C++ 标准，不同的解读。另外，对某些实现可能进行了一些权衡，不符合需要。例如，一种实现是针对性能优化的，而另一种实现重点关注尽量减少为容器使用的内存。

在使用标准库实现或任何第三方库时，必须考虑设计者在开发期间所做的权衡。第 4 章详细讨论了使用库时涉及的问题。

3. 处理不同的实现

正如前面所讨论的，并不是所有编译器和标准库的实现行为都完全相同。这是在跨平台开发时需要牢记的一点。具体来说，作为开发人员，很可能使用单个工具链，即单个编译器和单个标准库实现。不太可能使用产品必须使用的其他工具链来验证所有代码的改动。然而，这个问题有一个解决方案：持续集成和自动化测试。

应该设置一个持续集成环境，自动在需要支持的所有工具链上对所有代码的改动进行构建。当在某个工具链上构建中断时，应该自动通知中断构建的开发人员。

然而，并不是所有的开发环境都使用相同的项目文件来描述所有的源文件、编译器开关等。如果支持多个环境，那么手动维护每个环境的独立项目文件会是一场维护噩梦。最好使用一个类型的项目文件或一组一组构建脚本，这些脚本可以自动转换为具体的项目文件或特定工具链的具体构建脚本。其中一种工具称为 cmake。cmake 配置文件中描述了源文件、编译器开关、链接库等集合，它们也支持脚本化。然后 cmake 工具自动生成项目文件，例如用于 Windows 上开发的 Visual C++，或用于 macOS 上开发的 Xcode。

一旦持续集成环境构建，就应该触发该构建的自动化测试。这应该在生成的可执行文件上运行一套测试脚本，以验证其正确的行为。同样，在这个过程中如果出现问题，应该自动通知开发人员。

34.1.3 平台专用功能

C++是卓越的通用语言，又添加了标准库，包含的功能十分丰富；对于业余程序员来说，甚至可以多年仅凭借 C++的内置功能来轻松地编写 C++代码。但是，专业程序员需要一些 C++未提供的工具。本节将列出一些由平台或第三方库(而非由 C++语言或 C++标准库)提供的重要功能。

- 图形用户界面：当今，运行在操作系统上的大多数商业程序都有图形用户界面，其中包含诸多元素，如可单击按钮、可移动窗口和级联菜单。与 C 语言类似，C++不存在这些元素的概念。要在 C++中编写图形应用程序，可使用平台专用的库来绘制窗口、接收鼠标输入以及执行其他图形任务。更好的选择是使用第三方库，如 wxWidgets(wxwidgets.org)、Qt(qt.io)和 Uno(platform.uno)，这些库为构建图形应用程序提供了抽象层。这些库经常支持许多不同的目标平台。
- 联网：Internet 已经改变了我们编写应用程序的方式。当今，大多数应用程序都通过 Web 检查更新，游戏会通过网络提供多玩家模式。C++尚未提供联网机制，但存在几个标准库。使用抽象的套接字(socket)来编写联网软件是最常用的方式。大多数平台都包含套接字实现，允许采用简单的面向过程的方式在网络上传输数据。一些平台支持基于流的网络系统，运行方式类似于 C++中的 I/O 流。还有可用的第三方网络库提供了网络抽象层。这些库通常支持多个不同的目标平台。IPv6 正日益流行。因此，不要选择仅支持 IPv4 的网络库，最好选择独立于 IP 版本的网络库。
- OS 事件和应用程序交互：在纯粹的 C++代码中，与周边的操作系统以及其他应用程序的交互极少。在没有平台扩展的标准 C++程序中，只会接触命令行参数。C++不直接支持诸如复制和粘贴的操作。可以使用平台提供的库，也可以使用支持多平台的第三方库。例如，wxWidgets 和 Qt 库都抽象了复制和粘贴操作，并且支持多个平台。
- 低级文件：第 13 章“C++ I/O 揭秘”解释了 C++中的标准 I/O，包括读写文件。许多操作系统都提供自己的文件 API，通常与 C++中的标准文件类兼容。这些库通常提供 OS 专用文件工具，如获取当前用户主目录的机制。
- 线程：C++03 及更早版本不直接支持在单个程序中执行并发线程。从 C++11 开始，标准库添加了线程支持库，如第 27 章“C++多线程编程”所述；C++17 已经添加了并行算法，如第 20 章“掌握标准库算法”所述。如果需要标准库以外更强大的线程功能，需要使用第三方库，如 Intel TBB(Threading Building Block)和 STE||AR Group HPX(High Performance ParallelX)等。

注意：

如果正在进行跨平台开发，并且需要 C++语言或标准库未提供的功能，可尝试寻找提供所需功能的第三方跨平台库。如果开始使用平台专用的 API，跨平台应用程序会变得复杂很多，因为必须为支持的每个平台实现功能。

注意：

在使用第三方库时，如果可能的话，将这些库作为源代码，并使用所需的工具链自动构建它们。

34.2 跨语言开发

对于某些类型的程序而言，C++并非完成工作的最佳工具。例如，如果 UNIX 程序需要与 shell 环境密切交互，则最好编写 shell 脚本而非 C++程序。如果程序执行繁重的文本处理，你可能决定使

用 Perl 语言。如果需要大量的数据库交互，那么 C# 或 Java 是更好的选择。C# 结合 WPF 框架或 Uno 平台更适于编写现代图形用户界面应用程序。如果决定使用另一种语言，有时可能想要调用 C++ 代码，例如执行一些计算昂贵的操作。或者反过来，可能希望从 C++ 调用非 C++ 代码。幸运的是，可使用一些工具来取二者之长，将另一种语言的专长与 C++ 的强大功能和灵活性结合起来。

34.2.1 混用 C 和 C++

你已经知道，C++ 语言几乎就是 C 语言的超集。这意味着，几乎所有 C 程序都可在 C++ 中编译和运行。但有几处例外。一些例外与 C++ 不支持个别 C 功能有关；例如，C 支持 VLA (Variable-Length Array, 变长数组)，而 C++ 不支持。其他例外通常与保留字相关。例如 C 语言中，术语 `class` 没有特殊含义，因此可将其用作变量名，如以下 C 代码所示：

```
int class = 1; // Compiles in C, not C++
printf("class is %d\n", class);
```

上述代码在 C 中能够编译和运行，但如果编译为 C++ 代码，就会生成错误。将 C 程序转换或移植到 C++ 程序时，也将遇到此类错误。幸运的是，修复方法通常十分简单。在本例中，只需要将变量重新命名为 `classID`，代码即可编译。你将遇到的其他错误类型可能是 C++ 不支持一些 C 功能，但通常很少见。

如果遇到用 C 语言编写的有用库或遗留代码，可以方便地将 C 代码嵌入 C++ 程序中。在本书中你已多次看到，函数和类可以一起工作。类方法可以调用函数，而函数也可以使用对象。

34.2.2 改变范型

将 C 和 C++ 结合使用的一处危险在于程序会失去面向对象属性。例如，如果面向对象的 Web 浏览器是用过程化网络库实现的，程序将混合这两种范型。考虑到此类应用程序中网络任务的重要性和数量，可能考虑给过程库添加面向对象的包装器。可为此使用一种典型的设计模式，名为 *façade*。

例如，假设在用 C++ 编写一个 Web 浏览器，但使用了 C 网络库，C 网络库包含以下代码中声明的函数。注意，为简单起见，已经省略了 `HostHandle` 和 `ConnectionHandle` 数据结构。

```
// netwrklib.h
#include "HostHandle.h"
#include "ConnectionHandle.h"

// Gets the host record for a particular Internet host given
// its hostname (i.e. www.host.com )
HostHandle* lookupHostByName(char* hostName);
// Frees the given HostHandle
void freeHostHandle(HostHandle* host);

// Connects to the given host
ConnectionHandle* connectToHost(HostHandle* host);
// Closes the given connection
void closeConnection(ConnectionHandle* connection);

// Retrieves a web page from an already-opened connection
char* retrieveWebPage(ConnectionHandle* connection, char* page);
// Frees the memory pointed to by page
void freeWebPage(char* page);
```

`netwrklib.h` 接口简单直观，但并非面向对象，使用这种库的 C++ 程序员很讨厌它。这个库并未组

织到内聚类中，甚至达不到 **const-correct** 要求。当然，才华横溢的 C 程序员本可以编写一个更好的接口，但作为库用户，必须接受现实。可通过编写包装器来定制接口。

为库构建面向对象的包装器前，先分析一下，假如直接使用库会发生什么，由此理解其实际用法。在下面的程序中，使用 **netwrklib** 库检索网页 **www.wrox.com/index.html**：

```
HostHandle* myHost { lookupHostByName("www.wrox.com") };
ConnectionHandle* myConnection { connectToHost(myHost) };
char* result { retrieveWebPage(myConnection, "/index.html") };

cout << "The result is " << result << endl;

freeWebPage(result); result = nullptr;
closeConnection(myConnection); myConnection = nullptr;
freeHostHandle(myHost); myHost = nullptr;
```

要使库具有面向对象的特点，一种可能的方法是提供单个抽象，以识别链接、查找主机、连接到主机，然后检索网页。良好的面向对象的包装器会隐藏 **HostHandle** 和 **ConnectionHandle** 类型的不必要的复杂性。

下例遵循第 5 章“面向对象设计”和第 6 章“设计可重用代码”描述的设计原理：新类应当捕获该库的普通用法。上例显示了这一最常用的模式：首先查找主机，然后建立连接，最后检索网页。也可能从同一台主机检索后续页面，因此良好的设计也会考虑这种使用方式。

HostRecord 类首先包装查找主机的功能。这是一个 **RAII** 类。它的构造函数使用 **lookupHostByName()** 执行查找，它的析构函数自动释放检索的 **HostHandle**。代码如下：

```
class HostRecord
{
public:
    // Looks up the host record for the given host.
    explicit HostRecord(std::string_view host)
        : m_hostHandle { lookupHostByName(const_cast<char*>(host.data())) }
    { }

    // Frees the host record.
    virtual ~HostRecord() { if (m_hostHandle) freeHostHandle(m_hostHandle); }

    // Prevent copy construction and copy assignment.
    HostRecord(const HostRecord&) = delete;
    HostRecord& operator=(const HostRecord&) = delete;

    // Returns the underlying handle.
    HostHandle* get() const noexcept { return m_hostHandle; }

private:
    HostHandle* m_hostHandle { nullptr };
};
```

由于 **HostRecord** 类处理 C++ **string_view** 而非 C 风格的字符串，因此使用 **host** 的 **data()** 方法获取 **const char***；**netwrklib** 达不到 **const-correct** 要求，因此执行 **const_cast()** 进行弥补。

接下来实现 **WebHost** 类，该类使用 **HostRecord** 类。**HostRecord** 类创建与给定主机的连接，并且支持检索网页。**WebHost** 也是一个 **RAII** 类。在销毁 **WebHost** 对象时，它会自动关闭与主机的连接。代码如下：

```

class WebHost
{
public:
    // Connects to the given host.
    explicit WebHost(std::string_view host);
    // Closes the connection to the host.
    virtual ~WebHost();
    // Prevent copy construction and copy assignment.
    WebHost(const WebHost&) = delete;
    WebHost& operator=(const WebHost&) = delete;
    // Obtains the given page from this host.
    std::string getPage(std::string_view page);
private:
    ConnectionHandle* m_connection { nullptr };
};

WebHost::WebHost(std::string_view host)
{
    HostRecord hostRecord { host };
    if (hostRecord.get()) { m_connection = connectToHost(hostRecord.get()); }
}

WebHost::~~WebHost()
{
    if (m_connection) { closeConnection(m_connection); }
}

std::string WebHost::getPage(std::string_view page)
{
    std::string resultAsString;
    if (m_connection) {
        char* result { retrieveWebPage(m_connection,
            const_cast<char*>(page.data())) };
        resultAsString = result;
        freeWebPage(result);
    }
    return resultAsString;
}

```

WebHost 类有效地封装主机行为，并提供有用功能，不包含多余的调用和数据结构。**WebHost** 的实现广泛使用了 **netwrklib** 库，不向用户公开其工作原理。**WebHost** 的构造函数使用指定主机的 **HostRecord** RAII 对象。使用最终的 **HostRecord** 设置与主机的连接，将 **HostRecord** 存储在 **mConnection** 数据成员中供未来使用。在构造函数结束时自动销毁 **HostRecord** RAII 对象。**WebHost** 析构函数关闭连接。**getPage()** 方法使用 **retrieveWebPage()** 检索网页，将其转换为 **std::string**，使用 **freeWebPage()** 释放内存并返回 **std::string**。

WebHost 类使客户端程序员能够方便地处理一般情形：

```

WebHost myHost { "www.wrox.com" };
string result { myHost.getPage("/index.html") };
cout << "The result is " << result << endl;

```

注意：

精通网络技术的读者可能注意到，无限期地打开与主机的连接是一种不良做法，不符合 HTTP 规范。在生产质量的代码中不应该这样做。此处只是一个简化的示例。

可以看到，WebHost 类为 C 库提供了面向对象的包装器。通过提供抽象，可在不影响客户端代码的前提下更改底层实现并增加功能。这些功能可能包括：连接引用计数，按照 HTTP 规范在指定的时间过后自动关闭连接，在下次调用 `getPage()` 时自动重新打开连接等。

在本章末尾的练习中，将进一步探索如何编写包装器。

34.2.3 链接 C 代码

上例假设已经拥有初始 C 代码。该例利用了如下事实：可以用 C++ 编译器成功地编译大多数 C 代码。即便只有已经编译的 C 代码(可能是库形式的代码)，也仍可在 C++ 程序中使用它们，但需要另外执行几个步骤。

在 C++ 程序中开始使用已经编译的 C 代码之前，首先需要了解“名称改编”这个概念。为实现函数重载，会对复杂的 C++ 名称平台进行“扁平化”。例如，如果有一个 C++ 程序，则编写以下代码是合法的：

```
void myFunc(double);
void myFunc(int);
void myFunc(int, int);
```

这意味着链接器将看到 `myFunc`，但是不知道该调用哪种版本的 `myFunc` 函数。因此，所有 C++ 编译器执行名称改编操作，以生成合理的名称，如下所示：

```
myFunc_double
myFunc_int
myFunc_int_int
```

为避免与定义的其他名称发生冲突，生成的名称通常使用一些字符。对于链接器而言，这些字符是合法的；对于 C++ 源代码而言，这些字符是非法的。例如，Microsoft VC++ 生成如下名称：

```
?myFunc@@YAXN@Z
?myFunc@@YAXH@Z
?myFunc@@YAXHH@Z
```

编码十分复杂，而且是供应商专用的。C++ 标准未指定在给定的平台上如何实现函数重载，因此没有关于名称改编算法的标准。

C 语言不支持函数重载(编译器将报错，指出是重复定义)。因此，C 编译器生成的名称十分简单，例如 `_myFunc`。

因此，如果用 C++ 编译器编译一个简单的程序，即便仅有一个 `myFunc` 名称实例，也仍会生成一个请求，要求链接到改编后的名称。但是当链接 C 库时，找不到所需的已改编名称，因此链接器将报错。因此，有必要告知 C++ 编译器不要改编相应的名称。为此，需要在头文件中使用 `extern "language"` 限定，以告知客户端代码创建与指定语言兼容的名称；如果库源是 C++，还需要在定义站点使用这个限定，以告知库代码生成与指定语言兼容的名称。

`extern "language"` 的语法如下：

```
extern "language" declaration1();
extern "language" declaration2();
```

也可能如下：

```
extern "language" {
    declaration1();
    declaration2();
}
```

C++标准指出，可使用任何语言规范；因此，从原理上讲，编译器可支持以下代码：

```
extern "C" myFunc(int i);
extern "Fortran" matrixInvert(Matrix* M);
extern "Pascal" someLegacySubroutine(int n);
extern "Ada" aimMissileDefense(double angle);
```

但实际上，许多编译器只支持 C。每个编译器供应商都会告知你所支持的语言指示符。

例如，在以下代码中，将 `cFunction()` 的函数原型指定为外部 C 函数：

```
extern "C" {
    void cFunction(int i);
}

int main()
{
    cFunction(8); // Calls the C function.
}
```

在链接阶段，在附加的已编译二进制文件中提供 `doCFunction()` 的实际定义。`extern` 关键字告知编译器：链接的代码是用 C 编译的。

使用 `extern` 的更常见模式是在头文件级别。例如，如果使用 C 语言编写的图形库，很可能带有供使用的.h 文件。可以编写另一个头文件，将原始文件打包到 `extern` 块中，以指定定义函数的整个头文件是用 C 编写的。包装器的.h 文件通常使用 .hpp 名称，从而与 C 版本的头文件区分开：

```
// graphicslib.hpp
extern "C" {
    #include "graphicslib.h"
}
```

另一个常见模型是编写单个头文件，然后根据条件针对 C 或 C++对其进行编译。如果为 C++编译，C++编译器将预定义 `__cplusplus` 符号。该符号不是为 C 编译定义的。因此，可以经常看到以下形式的头文件：

```
#ifdef __cplusplus
    extern "C" {
#endif
    declaration1();
    declaration2();
#ifdef __cplusplus
    } // matches extern "C"
#endif
```

这意味着 `declaration1()` 和 `declaration2()` 是 C 编译器编译的库中的函数。使用该技术，同一个头文件可同时用于 C 和 C++客户端。

无论在 C++程序中添加 C 代码，还是针对已编译的 C 库进行链接，都要记住，虽然 C++几乎是 C 的超集，但它们是不同的语言，具有不同的设计目标。通过改编 C 代码，将其用于 C++是十分常见的，但是，最好为过程化 C 代码提供面向对象的 C++包装器。

34.2.4 从C#调用C++代码

虽然这是一本C++书籍，但不能否认其他卓越语言，比如C#。通过使用C#的互操作服务，可以方便地从C#应用程序调用C++代码。假设正在使用C#开发诸如GUI的应用程序，但使用C++实现某些性能关键组件或计算昂贵组件。为实现互操作，需要用C++编写一个库，这个库可从C#调用。在Windows上，库将会是DLL文件。以下C++示例定义了functionInDLL()函数，这个函数将编译为库。该函数接收一个Unicode字符串，并返回一个整数。这个实现将收到的字符串写入控制台，并将值42返回给调用者：

```
import <iostream>;

using namespace std;

extern "C"
{
    __declspec(dllexport) int functionInDLL(const wchar_t* p)
    {
        wcout << L"The following string was received by C++:\n ";
        wcout << p << L"" << endl;
        return 42; // Return some value...
    }
}
```

记住，是实现库中的函数而非编写程序，因此不需要main()函数。编译代码的方式取决于环境。如果正在使用Microsoft Visual C++，那么需要定位项目属性，选择Dynamic Library(.dll)作为配置类型。注意，这个示例使用__declspec(dllexport)告知链接器：这个函数应当供库的客户端使用。这是Microsoft Visual C++的处理方式。其他链接器可能使用不同的机制来导出函数。

一旦有了库，就可以使用互操作服务，从C#调用该库。首先需要添加InteropServices 名称空间：

```
using System.Runtime.InteropServices;
```

接下来定义函数原型，并告知C#从何处查找函数的实现。可使用下面的代码行，假设已经将该库编译为HelloCpp.dll：

```
[DllImport("HelloCpp.dll", CharSet = CharSet.Unicode)]
public static extern int functionInDLL(String s);
```

第一行告知C#：应当从HelloCpp.dll库导入该函数，而且应当使用Unicode字符串。第二部分指定函数的实际原型，这个函数接收一个字符串参数，并返回整数。以下代码是一个完整的示例，演示了如何从C#使用C++库：

```
using System;
using System.Runtime.InteropServices;

namespace HelloCSharp
{
    class Program
    {
        [DllImport("HelloCpp.dll", CharSet = CharSet.Unicode)]
        public static extern int functionInDLL(String s);

        static void Main(string[] args)
```

```

    {
        Console.WriteLine("Written by C#.");
        int result = functionInDLL("Some string from C#.");
        Console.WriteLine("C++ returned the value " + result);
    }
}

```

输出如下所示:

```

Written by C#.
The following string was received by C++:
'Some string from C#.'
C++ returned the value 42

```

有关上述 C# 代码的详情超出了本书的讨论范围,只需要能从这个示例了解大概即可。

本节只讨论了如何从 C# 中调用 C++ 函数,并没有提到如何从 C# 中使用 C++ 类。下一节将介绍 C++/CLI 来弥补这一点。

34.2.5 C++/CLI 在 C++ 中使用 C# 代码和在 C# 中使用 C++ 代码

要在 C++ 中使用 C# 代码,可以使用 C++/CLI。CLI 代表公共语言基础设施,是所有 .NET 语言(例如 C#、Visual Basic .NET 等)的主干。C++/CLI 是微软在 2005 年创建的,是支持 CLI 的 C++ 版本。2005 年 12 月, C++/CLI 被标准化为 ECMA-标准。可以在 C++/CLI 中编写 C++ 程序,并获得使用支持 CLI 的任何其他语言(如 C#)编写的任何其他功能。但是请记住, C++/CLI 可能落后于最新的 C++ 标准,这意味着它不一定支持所有最新的 C++ 特性。详细讨论 C++/CLI 语言超出了这本纯 C++ 书籍的范围。只给出几个小例子。

假设在一个 C# 库中有如下的 C# 类:

```

namespace MyLibrary
{
    public class MyClass
    {
        public double DoubleIt(double value) { return value * 2.0; }
    }
}

```

如下所示,可以从 C++/CLI 代码中使用这个 C# 库。重要的部分高亮显示。CLI 对象由内存垃圾收集器管理,当不再需要内存时,它会自动清理内存。因此,不能仅仅使用标准的 C++ new 运算符来创建托管对象,必须使用 gcnew,这是“garbage collect new”的缩写。不是将得到的指针存储在像 MyClass* 这样的普通指针变量中,也不是存储在像 std::unique_ptr<MyClass> 这样的智能指针中,必须使用所谓的句柄 MyClass^ 存储它,通常发音为“MyClass hat”。

```

#include <iostream>

using namespace System;
using namespace MyLibrary;

int main(array<System::String^>^ args)
{
    MyClass^ instance { gcnew MyClass() };
    auto result { instance->DoubleIt(1.2) };
    std::cout << result;
}

```

C++/CLI 也可以用于其他方向：可以编写所谓的托管 C++ ref 类，然后可以被任何其他 CLI 语言访问。下面是一个托管 C++ ref 类的简单示例：

```
#pragma once
using namespace System;

namespace MyCppClassLibrary
{
    public ref class MyCppClassRefClass
    {
    public:
        double TripleIt(double value) { return value * 3.0; }
    };
}
```

这个 C++/CLI 引用类可以在 C# 中使用，如下所示：

```
using MyCppClassLibrary;

namespace MyLibrary
{
    public class MyClass
    {
        public double TripleIt(double value)
        {
            // Ask C++ to triple it.
            MyCppClassRefClass cppRefClass = new MyCppClassRefClass();
            return cppRefClass.TripleIt(value);
        }
    }
}
```

如你所见，这些并不复杂，但这些示例都使用基本数据类型，如 `double`。如果需要处理非基本数据类型(如 `string`、`vector` 等)，那就会更加复杂，因为需要开始在 C# 和 C++/CLI 之间编组对象，反之亦然。然而，这里只是简单地介绍一下 C++/CLI。

34.2.6 在 Java 中使用 JNI 调用 C++ 代码

JNI(Java Native Interface, Java 本地接口)是 Java 语言的一部分，允许程序员访问用 Java 之外的语言编写的功能。JNI 允许程序员使用通过其他语言(如 C++)编写的库。如果 Java 程序员有性能关键或计算昂贵的代码段，或需要使用遗留代码，可以访问 C++ 库。也可以使用 JNI 在 C++ 程序中执行 Java 代码，但这种用法极少见。

由于这是一本 C++ 书籍，因此不介绍 Java 语言。如果你已经了解 Java，并且想在 Java 代码中嵌入 C++ 代码，那么建议阅读本节。

要开启 Java 跨语言开发之旅，首先看看 Java 程序。在下面这个示例中，使用了最简单的 Java 程序：

```
public class HelloCpp {
    public static void main(String[] args)
    {
        System.out.println("Hello from Java!");
    }
}
```

接下来需要声明一个用其他语言编写的 Java 方法。为此，使用 `native` 关键字并忽略实现：

```
// This will be implemented in C++.
public static native void callCpp();
```

C++代码最终会被编译为共享库，并动态加载到 Java 程序中。可在 Java 静态块中加载该库，这样，当 Java 程序开始执行时，将加载它。可为库指定任意名称，例如，在 Linux 系统上指定为 `hellocpp.so`，在 Windows 系统上指定为 `hellocpp.dll`。

```
static { System.loadLibrary("hellocpp"); }
```

最后，需要在 Java 程序中实际调用 C++代码。Java 方法 `callCpp()` 用作尚未编写的 C++代码的占位符。下面是完整的 Java 程序：

```
public class HelloCpp
{
    static { System.loadLibrary("hellocpp"); }

    // This will be implemented in C++.
    public static native void callCpp();

    public static void main(String[] args)
    {
        System.out.println("Hello from Java!");
        callCpp();
    }
}
```

这是 Java 一方需要完成的所有工作。现在，只需要像往常一样编译 Java 程序：

```
javac HelloCpp.java
```

然后使用 `javah` 程序，为本地方法创建头文件：

```
javah HelloCpp
```

运行 `javah` 后，将看到 `HelloCpp.h` 文件，这是一个完备(但有些简陋)的 C/C++头文件。这个头文件中是 `Java_HelloCpp_callCpp()` 函数的 C 函数定义。C++程序需要实现该函数。完整原型如下：

```
JNIEXPORT void JNICALL Java_HelloCpp_callCpp(JNIEnv*, jclass);
```

该函数的 C++实现可充分利用 C++语言。该例从 C++输出一些文本。首先，需要添加由 `javah` 创建的 `jni.h` 和 `HelloCpp.h` 文件。还需要添加想要使用的任何 C++头文件：

```
#include <jni.h>
#include "HelloCpp.h"
#include <iostream>
```

像往常一样编写 C++函数。函数的参数允许与 Java 环境以及称为本地代码的对象进行交互。这些超出了本例的讨论范围：

```
JNIEXPORT void JNICALL Java_HelloCpp_callCpp(JNIEnv*, jclass)
{
    std::cout << "Hello from C++!" << std::endl;
}
```

要根据环境，将该代码编译为库，很可能需要调整编译器的设置以包含 JNI 头文件。在 Linux 上

使用 GCC 编译器，编译器命令如下所示：

```
g++ -shared -I/usr/java/jdk/include/ -I/usr/java/jdk/include/linux \
HelloCpp.cpp -o hellocpp.so
```

编译器的输出是 Java 程序使用的库。只要共享库在 Java 类路径的某个位置，就可以像往常一样执行 Java 程序：

```
java HelloCpp
```

你应当会看到以下结果：

```
Hello from Java!
Hello from C++!
```

当然，本例仅触及 JNI 的皮毛。也可以使用 JNI 与 OS 专用功能或硬件驱动程序进行交互。要全面了解 JNI 信息，可参阅相关的 Java 书籍。

34.2.7 从 C++代码调用脚本

最初的 UNIX OS 包含相当有限的 C 库，不支持某些常见操作。UNIX 程序员因此从应用程序启动 shell 脚本，以完成需要 API 或库支持的任务。脚本可以用 Perl 和 Python 语言编写，但也可以是 Shell 脚本，用于在 Bash 等 Shell 中运行。

今天，许多 UNIX 程序员仍坚持将脚本用作子例程调用方式。为了启用这些类型的互操作性，C++提供了在<cstdlib>头文件中定义的 `std::system()` 函数。它只需要一个参数，表示要执行命令的字符串。示例如下：

```
system("python my_python_script.py"); // Launch a Python script.
system("perl my_perl_script.pl"); // Launch a Perl script.
system("my_shell_script.sh"); // Launch a Shell script.
```

然而，这种方式面临着重大风险。例如，如果脚本中存在错误，调用者可能获得详细的错误提示信息，也可能无法获得。`system()`调用的任务也异常繁重，因为必须创建整个过程来执行脚本。这样，最终会在应用程序中产生严重的性能瓶颈。

本书不再进一步讨论如何使用 `system()` 启动脚本。总体而言，应当研究 C++库的功能，看一下是否有完成任务的更好方式。有一些独立于平台的包装器，用于包装大量平台专用的库，如 Boost Asio 提供可移植的网络和其他低级 I/O，包括套接字、计时器和串行端口等。如果需要使用文件系统，可以使用平台无关的<filesystem>API，该 API 从 C++17 开始就作为标准库的一部分，在第 13 章进行了讨论。诸如使用 `system()`、启动 Perl 脚本来处理文本数据的做法并非最佳选择。为满足处理字符串的需要，诸如 C++正则表达式库的技术是更好的选择，详见第 21 章“字符串的本地化与正则表达式”。

34.2.8 从脚本调用 C++代码

C++内置了与其他语言和环境进行交互的通用机制。前面已经多次见过这种用法，你或许并未注意到，`main()`函数的实参和返回值就属于这类情况。

C 和 C++的设计基于命令行接口。`main()`函数从命令行接收实参，并返回可由调用者解释的状态码。在脚本环境中，传给程序的实参以及程序返回的状态码是一种强大机制，允许与环境进行交互。

示例：对密码进行加密

假设系统允许写入用户看到的所有内容，并输入一个文件用于审计。这个文件只能由系统管理员

读取，这样，如果某处出错，管理员可能确定谁是造错者。这个文件的摘要如下：

```
Login: bucky-bo
Password: feldspar

bucky-bo> mail
bucky-bo has no mail
bucky-bo> exit
```

尽管系统管理员可能希望记录所有用户活动，但也想遮盖每个人的密码，以防文件落入黑客手中。系统管理员决定编写一个脚本来解析日志文件，使用 C++ 进行实际加密。此后，该脚本调用 C++ 程序来执行加密。

以下脚本使用 Perl 语言，当然，其他大多数脚本都能完成这项任务。另外注意，当今，Perl 可使用库执行加密。但为了便于叙述，本例假设在 C++ 中完成加密。即便不了解 Perl，也仍可完成以下步骤。在这个示例中，最重要的 Perl 语法元素是 `字符。` 字符指示 Perl 脚本转向外部命令，这里，脚本将转向 C++ 程序 `encryptString`。

注意：

启动外部过程将产生大量性能开销，因为必须创建一个完备的新进程。如果需要调用一个外部进程，就不应该采用这种做法。但在这个密码加密示例中，这种做法是可行的，因为我们假设日志文件仅包含少数几个密码行。

这个脚本的策略是遍历 `userlog.txt` 文件的每一行，查找包含密码提示的行。脚本将内容写入一个新文件 `userlog.out`，这个新文件中包含的行与源文件完全相同，只是加密了所有密码。第一步是打开输入文件进行读取，并打开输出文件用于写入内容。此后，需要迭代文件中的所有行。将每一行依次放入 `$line` 变量中。

```
open (INPUT, "userlog.txt") or die "Couldn't open input file!";
open (OUTPUT, ">userlog.out") or die "Couldn't open output file!";
while ($line = <INPUT>) {
```

接着，对照正则表达式检查当前行，看一下特定行是否包含 `Password:` 提示符；如果包含，Perl 将密码存储在变量 `$1` 中。

```
if ($line =~ m/^Password: (.*)/) {
```

如果找到匹配，脚本会使用检测到的密码调用 `encryptString` 程序，获取密码的加密版本。将程序的输出保存在 `$result` 变量中，将程序的结果状态码保存在 `$?` 变量中。如果出现问题，脚本会检查 `$?`，然后立即退出。如果一切正常，会将密码行写入输出文件中。在输出文件中，用加密的密码替换原来的密码。

```
$result = `./encryptString $1`;
if ($? != 0) { exit(-1); }
print OUTPUT "Password: $result\n";
} else {
```

如果当前行不包含 `Password:` 提示符，脚本会将这一行原样写入输出文件。在循环的最后，会关闭这两个文件，然后退出。

```
print OUTPUT "$line";
}
}
```

```
close (INPUT);
close (OUTPUT);
```

这几乎就是全部。需要的其他部分是实际的 C++ 程序。本书不讨论如何实现加密算法。重要的部分是 `main()` 函数，因为它接收的实参是要加密的字符串。

实参包含在 C 风格字符串的 `argv` 数组中。在访问 `argv` 数组的元素前，始终应当检查 `argc` 参数。如果 `argc` 是 1，则实参列表中有一个元素，可作为 `argv[0]` 访问。`argv` 数组的第 0 个元素通常是程序名，因此，实参从 `argv[1]` 开始。

下面是 C++ 程序的 `main()` 函数，用于加密输入字符串。注意，如果成功，则程序返回 0；如果失败，则返回非 0 值，这是 Linux 中的标准。

```
int main(int argc, char* argv[])
{
    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " string-to-be-encrypted" << endl;
        return -1;
    }
    cout << encrypt(argv[1]);
}
```

注意：

上述代码中实际上存在一个明显的安全漏洞。将准备加密的字符串作为命令行实参传给 C++ 程序时，其他用户可通过进程表看到。可通过多种方式强化安全，例如，可通过标准输入将信息发送给 C++ 程序。

从上面可以看到，可以方便地将 C++ 程序嵌入脚本语言，可在自己的项目中取二者之长。可使用脚本语言与操作系统进行交互，并控制脚本流，同时使用诸如 C++ 的传统语言完成繁重的任务。

注意：

该例只是演示如何结合使用 Perl 和 C++。C++ 包含正则表达式库，允许方便地将这个 Perl/C++ 解决方案转换为纯粹的 C++ 解决方案。这个纯粹的 C++ 解决方案的运行速度更快，因为不需要调用外部程序。有关正则表达式库的详情，请参阅第 19 章。

34.2.9 从 C++ 调用汇编代码

与其他语言相比，C++ 是一门性能卓越的语言。但在极少数情况下，如果速度不是最重要的，可能想使用一些原始的汇编代码。编译器从源文件生成汇编代码，这些汇编代码的速度很快，几乎可用于所有目的。编译器和链接器(需要支持链接时代码生成功能)使用优化算法，使生成的汇编代码足够快。通过使用特殊的处理器指令集，如 MMX、SSE 和 AVX，这些优化器的功能越来越强大。当今，已经很难自行编写出比编译器生成的代码性能更高的汇编代码，除非了解这些增强的指令集的所有细节。

但本例中，C++ 编译器可使用关键字 `asm`，程序员可借此插入初始汇编代码。这个关键字是 C++ 标准的一部分，但实现是由编译器定义的。在一些编译器中，可使用 `asm`，在程序中从 C++ 降级到汇编代码。有时，对 `asm` 关键字的支持取决于目标架构，有时编译器使用非标准关键字而不是 `asm` 关键字。

例如，Microsoft VC++ 2019 不支持 `asm` 关键字。相反，它在 32 位模式下编译期支持 `__asm` 关键

字，但在 64 位模式下编译期根本不支持内联汇编。

可在一些应用程序中使用汇编代码，但在大多数程序中，建议不要这样做，原因有以下几点：

- 一旦开始为平台添加原始汇编代码，代码将不再有可移植性。
- 大多数程序员都不了解汇编语言，他们无法修改或维护代码。
- 人们很难读懂汇编代码，程序的样式也会受到不良影响。
- 大多数情况下，根本没必要这么做。如果程序很慢，则查找算法问题或咨询其他一些性能建议，见第 29 章“编写高效的 C++ 程序”的讨论。

警告：

在应用程序中遇到性能问题时，使用分析器确定真正的热点，通过改进算法来加速！只有别无他法时才考虑使用汇编代码，即使使用，也要留意汇编代码的缺点。

在实践中，如果有一个计算昂贵的代码块，应该将其移至一个独立的 C++ 函数中。如果使用性能分析方式(见第 29 章)确定这个函数就是性能瓶颈，并且无法再编写更简短、更快速的代码，那么可尝试使用原始汇编代码来提高性能。

这种情况下，首先要声明函数 `extern "C"`，以便禁用 C++ 名称改编功能。此时，可用汇编代码编写一个独立模块，以更高效地执行函数。独立模块的好处在于既有 C++ 中独立于平台的“引用实现”，也有原始汇编代码中平台专用的高性能实现。使用 `extern "C"` 意味着汇编代码可使用简单的命名约定(否则，必须对编译器的名称改编算法进行逆向工程)。此后，可链接 C++ 版本或汇编代码版本。

可用汇编代码编写这个模块，并通过汇编器来运行，而非使用 C++ 中的内联 `asm` 指令。在很多主流的符合 x86 标准的 64 位编译器(它们不支持内联的 `asm` 关键字)中尤其如此。

但是，只有当明显提高性能时，才应当使用原始汇编代码。性能至少提高一倍才值得这么做。提高 10 倍则很有必要这么做。提高 10% 则不值得去做。

34.3 本章小结

从本章可了解到，C++ 是一门灵活的语言。有的语言与特定平台捆得太紧，有的语言过于高级和通用，而 C++ 正好介于它们之间。在开发 C++ 代码时，不要永远将自己完全禁锢在 C++ 语言中。可将 C++ 与其他技术一起使用；C++ 技术成熟，代码库稳定，可确保它在未来占据重要地位。

本书第 V 部分的主题包括软件工程方法、编写高效的 C++、测试和调试技术、设计技术和设计模式、跨平台和跨语言应用程序开发。这将帮助 C++ 程序员更为卓越，实现蜕变。通过认真思考设计，尝试不同的面向对象编程方法，选择性地为自己的代码库添加新技术，并练习测试和调试技术，你的 C++ 技能必将达到专业水准。

34.4 练习

通过完成下面的习题，可以巩固本章所讨论的知识点。所有习题的答案都可扫描封底二维码下载。然而如果你受困于某个习题，可以在看网站上的答案之前，先重新阅读本章的部分内容，尝试着自己找到答案。

练习 34-1 编写一个程序，输出所有标准 C++ 整数类型的大小。如果可能的话，尝试在不同平台上的不同编译器编译和执行它。

练习 34-2 本章介绍整数大端和小端编码的概念。还解释说，在网络上，建议总是使用大端编码，

并在必要时进行转换。编写一个程序，可以在小端和大端编码之间从两个方向上转换 16 位无符号整数。要特别注意所使用的数据类型。编写一个 `main()` 函数来测试你的函数。

额外练习 32 位整数可以做相同的事情吗？

练习 34-3 转换范例一节中结合 C 和 C++ 的网络示例有些抽象，在这个示例中，C 函数的实现被省略，因为它们需要 C 和 C++ 标准库都没有提供的网络代码。在这个练习中，我们看一个小得多的 C 库，你可能想在你的 C++ 代码中使用它。C 库基本由两个函数组成，一个是函数 `reverseString()`，它分配一个新的字符串，并使用给定源字符串的反转来初始化它。第二个函数 `freeString()` 用于释放由 `reverseString()` 分配的内存。下面是它们的声明和描述性注释：

```
/// <summary>
/// Allocates a new string and initializes it with the reverse of a given string.
/// </summary>
/// <param name="string">The source string to reverse.</param>
/// <returns>A newly allocated buffer filled with the reverse of the
/// given string.
/// The returned memory needs to be freed with freeString().</returns>
char* reverseString(char* string);

/// <summary>Frees the memory allocated for the given string.</summary>
/// <param name="string">The string to deallocate.</param>
void freeString(char* string);
```

你将如何在 C++ 代码中使用这个“库”？

练习 34-4 本章中所有关于混用 C 和 C++ 代码的示例都是关于从 C++ 调用 C 代码的。当然，如果只使用 C 所知道的数据类型，也可能出现相反的情况。在这个练习中，你将两个方向结合起来。编写一个名为 `writeTextFromC(const char*)` 的 C 函数，调用一个名为 `writeTextFromCpp(const char*)` 的 C++ 函数，该函数使用 `cout` 将给定的字符串打印到标准输出。为了测试代码，在 C++ 中编写 `main()` 函数，调用 C 函数 `writeTextFromC()`。

第VI部分

附录

- ▶ 附录 A C++面试
- ▶ 附录 B 标准库头文件
- ▶ 附录 C UML 简介
- ▶ 附录 D 带注解的参考文献

附录 A

C++面试

读完本书后，就可以开始 C++生涯了，但雇主在开出高薪前，求职者需要展现自己的实力。各家公司的面试方法并不相同，但技术面试的很多方面都是可预见的。完整的面试会测试基本编码技能、调试技能、设计和风格技能以及解决问题的能力。求职者面对的问题集合相当庞大。本附录将介绍求职者可能面对的各种不同类型的问题，以及获得高薪 C++编程工作的最佳战术。

本附录回顾本书的各章，讨论每章中可能出现在面试场合中的问题，还讨论可能用来测试这些技能的问题类型，以及处理这些问题的最佳方法。

第1章：C++和标准库速成

技术面试包含一些基础性的 C++问题，以筛除对 C++语言知之甚少的求职者。这些问题可能会通过电话询问的形式提出，在面试前，开发人员或招聘人员先给求职者打电话，询问问题。这些问题也可能通过电子邮件询问或当面询问。在回答这些问题时，请记住，面试官只是想了解求职者是否真正学过并用过 C++。通常情况下，获得高分并不需要顾及每个细节。

要记住的内容

- 函数的用法
- 一致性初始化
- 模块的基本用法(C++20)
- 头文件的语法，包括为标准库头文件略去".h"
- 名称空间和嵌套名称空间的用法
- 语言基础知识，如循环语法，包括基于范围的 for 循环、条件运算符和变量等
- 三向比较运算符(C++20)
- 枚举类型
- 栈和堆之间的差异
- 动态分配的数组
- const 的使用
- 指针和引用的含义，以及二者的区别
- 引用在声明时必须绑定在变量上且不能改变

- 引用传递相对于值传递的优点
- `auto` 关键字
- 标准库容器(如 `std::array` 和 `std::vector`)的基本用法
- 结构化绑定
- 使用 `std::pair` 和 `std::optional`
- 类型别名和 `typedef` 的原理
- 属性背后的一般思想

问题类型

C++基础问题可能采取词汇测试的形式。面试官可能让求职者定义一些 C++术语,例如 `auto` 和 `enum class`。面试官期待的可能是教科书答案,而给出一些示例或额外的细节,通常可以获得加分。例如,在谈到 `auto` 关键字时,除了解释 `auto` 在定义变量时的作用,求职者还可以提到它在函数返回类型推导时的用法。

测试 C++基本能力的另一种问题形式是在面试官面前编写一段简短程序。面试官可能给求职者一个热身问题,如用 C++编写 Hello World 程序。面对这样一个看上去很简单的问题时,求职者一定要展示出了解名称空间,使用流而不是 `printf()`,以及知道应该导入哪些标准模块,以获得所有加分。

解释 `const` 的定义是 C++面试中的一个经典问题。根据对这个问题的回答,面试官心中会有一个对求职者能力评估的范围。例如,一般的候选人会回答 `const` 修饰变量时的用法,好的候选人还会加上 `const` 在成员方法和成员变量中的用法,出色的候选人会提到 `const` 和引用的结合,并给出 `const` 的引用传递的语义且说明在什么情况下这会比值传递更高效。

本章提到的某些主题会出现在寻找 bug 的题目中。要警惕引用的错误使用。例如,当一个类中包含引用类型的成员变量时:

```
class Gwentyth
{
    private:
        int& m_caversham;
};
```

因为 `m_caversham` 是一个引用,当类被构造时,它必须被绑定到某个变量上。为了做到这一点,求职者需要使用一个构造函数初始化器。类可以将要引用的变量作为构造函数的参数:

```
class Gwentyth
{
    public:
        Gwentyth(int& i) : m_caversham { i } { }
    private:
        int& m_caversham;
};
```

第2章和第21章:字符串、字符串视图、本地化与正则表达式

字符串非常重要,用于几乎每种应用程序。面试官很有可能至少问一个用 C++处理字符串的问题。

要记住的内容

- `std::string` 和 `std::string_view` 类
- 尽量使用 `const string&` 或 `string` 而不是 `string_view` 作为函数的返回类型
- C++ `std::string` 类和 C 风格的字符串的区别, 包括为什么要避免使用 C 风格的字符串
- 字符串和数值类型(如整数和浮点数)的相互转换
- 使用 `std::format()` 格式化字符串
- 原始字符串字面量
- 本地化的重要性
- Unicode 背后的思想
- `locale` 和 `facet` 的概念
- 正则表达式的含义

问题类型

面试官可能会要求求职者解释如何将两个字符串串联在一起, 通过这个问题了解求职者是按照专业 C++ 程序员的方式思考还是按照 C 程序员的方式思考。如果遇到这样的问题, 求职者应该解释 `std::string` 类, 说明如何通过这个类串联两个字符串。另外值得一提的是, `string` 类能自动处理所有内存管理操作, 并且要和 C 风格的字符串进行比较。

面试官可能不会专门询问有关本地化的问题, 但求职者可在面试时使用 `wchar_t` 而不是 `char`, 以展示自己对全球化的考虑。如果遇到关于本地化经验的问题, 一定要提到从项目刚开始就考虑在世界范围内使用的重要性。

求职者可能会被问到有关 `locale` 和 `facet` 基本思想的问题。此时也许不需要解释具体语法, 但是应该解释为什么通过 `locale` 和 `facet` 可根据特定语言的规则对文本和数字进行格式化。

求职者可能会遇到一个关于 Unicode 的问题, 但这个问题很可能是要求解释 Unicode 的基本思想和概念, 而不是解释实现细节。因此, 应确保自己理解 Unicode 的高层概念, 并可以解释 Unicode 在本地化上下文中的使用。求职者也应该知道编码 Unicode 字符的不同选择, 例如 UTF-8 和 UTF-16, 而不需要具体细节。

在第 21 章中可看到, 正则表达式的语法非常难理解。面试官不太可能询问正则表达式的细节。然而, 求职者应能解释正则表达式的概念, 以及了解通过正则表达式能进行哪些类型的字符串操作。

第 3 章: 编码风格

任何在专业领域编写过代码的人都有像在麦片盒背面学习 C++ 一样的同事。没有谁愿意和编写混乱代码的人一起工作, 因此面试官有时会试图了解求职者的编码风格。

要记住的内容

- 风格很重要, 即使是在没有和风格明显相关的面试中也是如此
- 编写良好的代码并不需要大量的注释
- 注释可用于传达元信息
- 分解是将代码分成较小片段的实践
- 重构是指重新编排代码的结构, 例如清理以前编写的代码

- 命名是十分重要的技巧，要注意如何给变量和类等指定名称

问题类型

风格问题可能采取几种不同的形式。笔者的一位朋友曾被要求在白板上编写一个较复杂的算法。当他写下第一个变量名时，就被面试官打断说通过了面试。问题不在于算法本身，而在于命名变量的风格如何。更常见的情况是，求职者可能会被要求提交自己编写过的代码，或给出有关风格的意见。

当潜在的雇主要求提交代码时，求职者要小心。求职者可能不能合法地提交为前任雇主编写的代码，还必须找到一段能展示求职者技能的代码，但这段代码又不要求太多的背景知识。例如，不要向一家正在面试数据库管理职位的公司提交有关高速图像渲染的硕士论文。

如果雇主要求求职者编写一个具体程序，那么这是一次展示求职者在本书中学会的技能的绝佳机会。即使潜在的雇主没有要求编写具体程序，求职者也应该考虑写一个小程序专门提交给这家公司。不要选择某个已经写过的程序，而是从头开始编写一些和工作相关、能展示良好风格的代码。

此外，如果求职者有一些自己编写且能公开的文档(即非机密文档)，就可以用它们展示自己的沟通技能，这也能求职者加分。求职者构建或维护的网站，向 CodeGuru、CodeProject 等网站提交的文章都很有用；这表明求职者不仅会编写代码，还懂得和其他人交流如何高效地使用代码。当然，有一本署有求职者大名的书籍也是一大利好。

如果正在给处于开发中的开源项目贡献代码(例如在 GitHub 上)，求职者将获得附加分。如果有自己主动维护的开源项目，效果将更好。可利用这个大好机会，展示自己的编码风格和沟通技巧。某些雇主会查看 GitHub 等网站上的个人简介。

第4章：设计专业的 C++ 程序

面试官还要确保求职者除了知道 C++ 语言外，还能熟练地应用这门语言。求职者可能不会遇到明显的设计问题，但优秀的面试官会通过各种技术将设计暗藏在其他问题中。

潜在的雇主还想知道求职者能否使用其他人编写的代码。如果求职者在简历上列出了具体的库，就要做好准备回答关于这些库的问题。如果没有列出具体的库，那么对库的重要性的一般性理解就足够了。

要记住的内容

- 设计带有主观色彩——准备在面试过程中为自己的设计决策辩护
- 在面试前回顾一下以前做过的设计的细节，以防面试时被要求举例说明
- 准备好可视化地勾勒出一设计，包括类层次结构
- 准备好说出代码重用的好处
- 理解库的概念
- 从头开始构建和重用现有代码之间的权衡
- 大 O 表示法的基本知识，至少要记得 $O(n \log n)$ 比 $O(n^2)$ 好
- C++ 标准库中包含的功能
- 设计模式的高层次定义

问题类型

设计问题是面试官很难提问的问题——在面试场合中设计的任何程序可能都太简单，无法反映真

实的设计技能。设计问题可能采取更模糊的形式，例如“告诉我设计优秀程序应该采取哪些步骤”，或“解释一下抽象的原理”。这类问题也可以不那么明显。在讨论求职者之前工作时，面试官可能会说“请解释一下那个项目的设计”。注意，在回答问题时，不要泄露前雇主的知识产权。

如果面试官询问关于某个库的问题，那么他关注的可能是这个库的高层次方面，而不是技术细节。例如，求职者可能需要从库设计的角度，解释标准库的优缺点分别是什么。最好的求职者会说标准库的优点在于广度和标准化，而主要缺点在于有时用法比较复杂。

求职者也可能会遇到一个初看上去和库无关的问题。例如，面试官可能会问求职者，如何创建一个应用程序，用于从网上下载 MP3 音乐，然后在本地计算机上播放。此问题没有明确涉及库，但需要用库解决；这个问题真正考察的是过程。

求职者应该首先讨论如何收集需求，做出初步原型。因为这个问题涉及两个具体技术，所以面试官想知道求职者会如何处理它们。这里要用到库。如果求职者告诉面试官，需要自行编写 Web 类和 MP3 播放代码，面试并不会失败，但面试官会质疑再造这些工具的时间和开销。

更好的答案应该是调查实现 Web 和 MP3 功能的现有库，看有没有适合这个项目的库。求职者可能需要说出自己打算使用的一些技术，例如 Linux 下获取 Web 内容的 libcurl 库，或 Windows 下播放音乐的 Windows Media 库。

提及一些提供免费库的网站，以及这些网站提供的某些内容也可能获得加分，例如提供 Windows 库的 www.codeguru.com 和 www.codeproject.com，提供独立于平台的 C++ 库的 www.boost.org 和 www.github.com。提及一些可用于开源软件的许可协议，如 GPL 许可协议、Boost 许可协议、Creative Commons 许可协议、CodeGuru 许可协议和 OpenBSD 许可协议等，可能会让求职者获得额外加分。

第 5 章：面向对象设计

面向对象的设计问题用于筛选只知道什么是引用的 C 程序员，选出真正会使用语言的面向对象特性的 C++ 程序员。面试官不会想当然地假设任何事情；即使求职者有多年的面向对象语言的使用经验，他们也仍希望找出求职者掌握面向对象方法论的证据。

要记住的内容

- 过程式范式和面向对象范式之间的差异
- 类和对象之间的差异
- 从组件、属性和行为方面描述类
- “是一个”关系和“有一个”关系
- 多重继承相关的权衡

问题类型

有关面向对象的设计问题通常有两种问法。求职者可能需要定义面向对象的概念，或者勾勒出面向对象的层次结构。前者非常简单。要记住，给出例子能获得加分。

如果要求求职者勾勒出面向对象的层次结构，面试官通常会给出一个简单的应用程序，例如纸牌游戏，让求职者为它设计类层次结构。面试官常提出有关游戏的设计问题，因为游戏是大多数人都已经熟悉的应用程序。相比数据库实现之类的问题，这类问题还能稍微缓解紧张的情绪。当然，求职者给出的类层次结构因所设计的游戏或应用程序而异。下面的几点需要考虑：

- 面试官希望了解求职者的思维过程。打开思路，头脑风暴，让面试官参与讨论，不要害怕推倒重来。
- 面试官可能假设求职者对这个应用程序很熟悉。如果求职者从来没有听说过二十一点(一种牌类游戏)，面试官却提出一个关于二十一点的问题，可以请面试官解释清楚这个问题或换个问题。
- 除非面试官要求求职者使用一种特定的格式描述层次结构，否则建议求职者描述类图时，采用继承树的形式，让每个类都带有方法和数据成员的大致列表。
- 求职者可能需要对自己的设计做辩护，或修改自己的设计，以满足新需求。求职者应尝试判断面试官是否真正看到自己设计中的缺陷，或者面试官只是想让求职者辩护，以便观察求职者的说服技巧。

第6章：设计可重用代码

面试官很少问关于设计可重用代码的问题。这种疏忽是很遗憾的，因为如果小组中有成员只能编写单一作用的代码，这对编程小组是不利的。偶尔，求职者也会发现一些公司精于代码重用，会在面试中提问关于代码重用的问题。这样的问题表明为这家公司工作是很好的。

要记住的内容

- 抽象的原则
- 子系统和类层次结构的创建
- 良好接口设计的一般规则，即只有公有方法的接口，不带有实现细节
- 何时使用模板，何时使用继承

问题类型

面试官可能会要求求职者解释抽象的原则与优点，并给出一些具体的例子。

有关重用的问题几乎肯定涉及求职者以前从事过的项目。例如，假设求职者曾在一家开发消费类和专业类视频编辑应用程序的公司工作过，那么面试官可能询问这两类应用程序之间如何共享代码。即使面试官没有明确地问有关代码重用的问题，求职者也可以涉及这方面的内容。当求职者描述过去的一些工作经历时，告诉面试官自己写的模块是否在其他项目中使用。即使在回答明显很简单的编码问题时，也一定要注意考虑和提到所涉及的接口。老生常谈，在回答问题时，不要泄露前雇主的知识产权。

第7章：内存管理

面试官一定会问求职者一些和内存管理相关的问题，包括求职者对智能指针的了解。除了智能指针，还有可能有一些更底层的问题。目的是考察 C++ 的面向对象特性是否让求职者对底层的实现细节太生疏。内存管理的问题可让求职者证明自己知道底层发生了什么。

要记住的内容

- 知道如何绘制栈和自由存储区的结构，这有助于了解幕后的工作原理

- 避免使用低级的内存分配和释放函数。在现代 C++ 中，不应当调用 `new`、`delete`、`new[]`、`delete[]`、`malloc()` 和 `free()` 等，而是要使用智能指针
- 理解智能指针。默认情况下使用 `std::unique_ptr`，使用 `shared_ptr` 共享所有权
- 使用 `std::make_unique()` 创建 `unique_ptr`
- 使用 `std::make_shared()` 创建 `shared_ptr`
- 永远都不要使用 `std::auto_ptr`，自 C++17 之后它已被移除
- 如果确实需要使用低级的内存分配函数，可使用 `new`、`delete`、`new[]` 和 `delete[]`，不要使用 `malloc()` 和 `free()`
- 即使有一个指向对象的指针数组，也仍需要为每个指针分配和释放内存——数组分配语法不负责处理指针
- 了解存在一些探测内存分配问题的工具，例如 Valgrind，可用于找出内存问题

问题类型

“查找 bug”这种问题通常会包含内存问题，例如双重删除、`new/delete/new[]/delete[]` 混用以及内存泄漏等。跟踪大量使用指针和数组的代码时，求职者应该在处理每一行代码时绘制和更新内存的状态。

考察求职者是否理解内存的另一个好方法是询问指针和数组的区别。此时，如果求职者不大清楚这两者之间的区别，就会被这个问题击倒。如果是这样，请再次浏览第 7 章讨论的内容。

在回答关于内存分配的问题时，最好总是提到智能指针的概念，以及智能指针自动清理内存或其他资源的优点。求职者肯定还要提到使用 `std::vector` 这样的标准库容器比使用 C 风格数组要好得多，因为标准库容器可自动处理内存管理。

第 8 章和第 9 章：熟练掌握类和对象、精通类和对象

有关类和对象的问题是没有边界的。一些面试官迷恋于语法，可能给求职者提供一些复杂的代码。其他一些面试官对实现细节不那么关心，而对设计技能更感兴趣。

要记住的内容

- 类定义的基本语法
- 方法和数据成员的访问说明符
- `this` 指针的使用
- 名称解析的工作方式
- 对象的创建和析构，既包含栈上的对象，也包含自由存储区上的对象
- 编译器自动生成构造函数的情况
- 构造函数初始化器
- 拷贝构造函数和赋值运算符
- 委托构造函数
- `mutable` 关键字
- 方法重载和默认参数
- `const` 成员
- 友元类和友元方法

- 管理对象的动态分配内存
- 静态方法和数据成员
- 内联方法，事实上 `inline` 关键字只是编译器提示，编译器可以忽略这个提示
- 分离接口和类实现的关键思想，即接口应该只包含 `public` 方法，而且应该尽可能稳定；接口不应该包含任何数据成员或 `private/protected` 方法；因此，接口可保持稳定，而底层实现可随意变化
- 类内成员初始化器
- 显式默认和显式删除特殊成员函数
- 右值和左值的区别
- 右值引用
- 使用移动构造函数和移动赋值运算符的移动语义
- “复制和交互”惯用方法及其作用
- “零规则”和“5 规则”
- 类的三向比较运算符(C++20)

问题类型

诸如“`mutable` 关键字的含义是什么？”之类的问题是很好的电话筛选问题。面试官可能有一个 C++ 术语的清单，根据求职者正确解释这些术语的数目，将求职者安排到面试过程的下一阶段。求职者可能不知道被问到的所有术语的意义，但要记住，其他求职者也面临同样的问题，这只是面试官采用的几个指标之一。

“查找 bug”这类问题在面试官和课程教员之类的人群中非常流行。求职者要面对一些无意义的代码，指出代码中的缺陷。面试官通过量化方法分析求职者，而这是为数不多的途径之一。一般情况下，求职者应该阅读每行代码，说出思考过程，大声说出自己的想法。bug 的类型可归为以下类别。

- **语法错误**——这比较少见，面试官知道，通过编译器可以找到这些编译期 bug。
- **内存问题**——这类问题包括内存泄漏和双重删除。
- **“你不应该这么做”的问题**——包括在技术上正确，但不建议这么做的事情。例如，不要使用 C 风格的字符数组，而是改用 `std::string`。
- **风格错误**——即使面试官不把这算作 bug，求职者也要指出糟糕的注释和变量名。

下面这个“查找 bug”问题演示了上述 bug 类型。

```
class Buggy
{
    Buggy(int param);
    ~Buggy();
    double fjord(double val);
    int fjord(double val);
protected:
    void turtle(int i = 7, int j);
    int param;
    double* mGraphicDimension;
};

Buggy::Buggy(int param)
{
    param = param;
    mGraphicDimension = new double;
```



```

}

Buggy::~Buggy()
{
}

double Buggy::fjord(double val)
{
    return val * param;
}

int Buggy::fjord(double val)
{
    return (int)fjord(val);
}

void Buggy::turtle(int i, int j)
{
    cout << "i is " << i << ", j is " << j << endl;
}

```

仔细看一下代码，然后在下面修正过的版本中找答案：

```

#include <iostream>           // Streams are used in the implementation.
#include <memory>             // For std::unique_ptr.

class Buggy
{
public:
    // These should most likely be public.
    Buggy(int param);

    // Recommended to make destructors virtual. Also, explicitly
    // default it, because this class doesn't need to do anything
    // in it.
    virtual ~Buggy() = default;

    // Disallow copy construction and copy assignment operator.
    Buggy(const Buggy& src) = delete;
    Buggy& operator=(const Buggy& rhs) = delete;

    // Explicitly default move constructor and move assignment op.
    Buggy(Buggy&& src) noexcept = default;
    Buggy& operator=(Buggy&& rhs) noexcept = default;

    // int version won't compile. Overloaded
    // methods cannot differ only in return type.
    double fjord(double val);

private:
    // Use private by default.
    void turtle(int i, int j); // Only last parameters can have defaults.
    int mParam;                // Data member naming.
    std::unique_ptr<double> mGraphicDimension; // Use smart pointers!
};

Buggy::Buggy(int param) // Prefer using ctor initializer
    : mParam(param)
    , mGraphicDimension(new double)

```



```

{
}

double Buggy::fjord(double val)
{
    return val * mParam;           // Changed data member name.
}

void Buggy::turtle(int i, int j)
{
    std::cout << "i is " << i << ", j is " << j << std::endl; // Namespaces.
}

```

求职者应该解释，最好避免使用普通指针表示所有权，而是改用智能指针。还应当解释为什么要将移动构造函数和移动赋值运算符显式设置为默认，为什么会酌情删除拷贝构造函数和拷贝赋值运算符。解释如果实现拷贝构造函数和拷贝赋值运算符，会对类产生什么影响。

第10章：揭秘继承技术

有关继承的问题在形式上通常和有关类的问题一致。面试官可能要求求职者实现一个类的层次结构，以表明求职者拥有足够的 C++ 经验，不需要查书就可以编写派生类。

要记住的内容

- 派生一个类的语法
- 从派生类的角度看 `private` 和 `protected` 之间的区别
- 方法重写和 `virtual`
- 重写和重载的区别
- 析构函数应该为虚函数的原因
- 链式构造函数
- 向上转换和向下转换的来龙去脉
- C++ 中不同类型的转换
- 多态性的原则
- 纯虚方法和抽象基类
- 多重继承
- 运行时类型信息(RTTI)
- 继承构造函数
- 类的 `final` 关键字
- 方法的 `override` 关键字和 `final` 关键字

问题类型

很多继承问题的陷阱都与细节相关。求职者在编写基类时，不要忘记将方法标记为 `virtual`。如果将所有方法都标记为 `virtual`，要准备好证明这一决策。求职者应该能解释 `virtual` 的意义及其工作方式。另外，不要忘了在派生类的定义中，在父类名称的前面加上 `public` 关键字(例如，`class Derived : public Base`)。在面试时，不太可能要求求职者执行非 `public` 继承。

有关继承的更具挑战性的问题在于基类和派生类之间的关系。要确保了解不同访问级别的工作方

式，以及 `private` 和 `protected` 之间的区别。提醒自己了解切片现象，即特定的转换类型会导致类丢失其派生类信息。

第 11 章：零碎的工作

本章讨论了 C++ 语言的几个特点，它们不适合任何其他章节。除了 `static` 关键字和模块的概念外，面试官在本章中不会问太多问题。

要记住的内容

- `static` 的多种用法
- 模块是什么，为什么使用它们比使用头文件更好（C++20）
- 预处理宏的概念和缺点
- 为什么不应该使用 C 风格的变长参数列表

问题类型

要求求职者定义 `static` 是经典的 C++ 面试问题。因为这个关键字有多种用途，可以用来评价求职者知识的广度。肯定应该讨论 `static` 方法和 `static` 数据成员，并给出好的示例。如果还解释了 `static` 链接和函数中的 `static` 变量，求职者会得到额外的分数。

模块是使代码可重用并支持明确职责分离的完美方式。要确保知道如何使用模块以及如何编写基本模块。

第 12 章和第 26 章：利用模板编写泛型代码、高级模板

模板是 C++ 中最晦涩的部分，这是面试官用于区分 C++ 初学者和高手的好方法。尽管大部分面试官都会原谅求职者不记得一些高级的模板语法，但求职者起码应该了解基础知识。

要记住的内容

- 如何使用类或函数模板
- 如何编写基本的类或函数模板
- 缩写函数模板语法（C++20）
- 函数模板参数推导
- 类模板参数推导（CTAD）
- 模板别名，以及为什么模板别名比 `typedef` 更好
- 概念的思想及基本用法（C++20）
- 可变模板和折叠表达式的概念
- 元编程的思想
- 类型 `trait` 及其作用

问题类型

很多面试问题都从一个简单的问题开始，然后逐步增加难度。通常情况下，问题的复杂性可以不

断添加，他们只想看看求职者能走多远。例如，面试官在提问时，首先让求职者创建一个类，对固定数目的 `int` 值提供顺序访问功能。接下来，扩充这个类，以容纳任意数量的元素。然后，这个类需要处理任意数据类型，这里需要引入模板。从此开始，面试官可从很多不同的方向引申问题，例如要求求职者通过运算符重载提供类似数组访问的语法，或继续在模板方向深入，要求求职者为模板类型参数提供默认类型或提供类型限制。

模板更容易出现在另一个编码问题的解决方案中，而不是直接询问相关的问题。求职者应该温习基本知识，以防出现这些问题。然而，大部分面试官都知道模板语法的难度很高，在面试过程中要求编写复杂的模板代码是很残酷的。

面试官可能会问求职者一些关于元编程的高层次问题，看看求职者是否听说过这些概念。在解释概念时，求职者可给出一些小例子，例如编译期计算某数的阶乘。如果语法不完全正确，不必担心。只要解释代码本应完成的功能就好了。

第13章：C++ I/O 揭秘

如果求职者在应聘一个编写 GUI 应用程序的工作，就可能不会被问到太多关于 I/O 流的问题，因为 GUI 应用程序常使用其他 I/O 机制。然而，流会出现在其他的问题中，在面试官看来，作为标准 C++ 的一部分，这些问题对于求职者来说是公平的。

要记住的内容

- 流的定义
- 使用流的基本输入输出
- 操作算子的概念
- 流的类型(控制台、文件和字符串等)
- 错误处理技术
- 现有的标准文件系统 API

问题类型

I/O 可能出现在任何问题的上下文中。例如，面试官可能让求职者读取一个包含测试分数的文件，然后将数据放在 `vector` 中。这个问题测试基本 C++ 技能、基本的标准库以及基本 I/O。即使 I/O 只是求职者要解决的问题中的一小部分，也一定要检查是否存在错误。否则，面试官就可能给求职者本应完美的程序做出负面评论。

第14章：错误处理

一些公司有时会避免招聘应届毕业生或新手程序员来完成重要(以及高薪)的工作，因为他们假定，这些人编写不出产品级质量的代码。求职者可在面试时展示自己的错误处理技能，向面试官证明自己的代码不会轻易崩溃。

要记住的内容

- 异常的语法
- 将异常捕获为 `const` 引用

- 最好使用异常的层次结构，而不是几个通用的异常
- 当抛出异常时，栈展开的基本工作原理
- 如何处理构造函数和析构函数中的错误
- 智能指针有助于在抛出异常时避免内存泄漏
- `std::source_location` 类作为某些 C 风格预处理宏的替代品
- 绝不要在 C++ 中使用 C 函数 `setjmp()` 和 `longjmp()`

问题类型

面试官会观察求职者如何报告和处理错误。当求职者被要求编写一段代码时，求职者一定要实现恰当的错误处理。此外，求职者可能需要给出抛出异常时栈展开工作方式的高层次概述，而不要求实现的细节。

当然，并非所有程序员都理解或欣赏异常。有些程序员可能因为性能原因，对异常存在完全无根据的偏见。如果面试官要求求职者不用异常实现某个功能，求职者就必须使用传统的 `nullptr` 检查和错误代码。这是求职者展示 `nothrow new` 知识的好机会。

`std::source_location` 类的知识以及它的重要使用情况会为求职者加分。

面试官可能会问这种形式的问题：“你是否会因为性能影响而拒绝使用异常？”求职者应该解释异常会带来性能损失的认知是误解。在现代编译器上，异常开销几乎为零。

第 15 章：C++运算符重载

在面试过程中，求职者可能要执行比简单运算符重载更复杂的操作，不过可能性也不是那么大。一些面试官喜欢准备一些高级问题，他们不指望有人能正确回答这些问题。运算符重载的异常复杂性使这方面的问题几乎不可能出现，因为很少有程序员能在不参考书的情况下写出正确的语法。这意味着在面试之前最好先复习一下。

要记住的内容

- 重载流运算符，因为这些是最常重载的运算符，概念也很独特
- 仿函数的概念和创建方法
- 方法运算符或全局函数之间的选择
- 一些运算符可按其他运算符的方式表达，比如 `operator<=` 可写为对 `operator>` 运算符的结果取反
- 可以定义自己的用户定义字面量，但不需要语法细节

问题类型

我们要面对现实——运算符重载问题(除了简单的之外)很难。任何要问这类问题的人都知道这一点，如果求职者回答正确，肯定会给对方留下深刻印象。不可能准确地预测求职者会遇到什么问题，但是运算符重载问题的数目是有限的。只要求求职者阅读每个适合重载的运算符的例子，就能表现得很好！

求职者除了要求实现重载运算符之外，还可能遇到有关运算符重载的高层次问题。“查找 bug”这类问题可能包含一个重载的运算符，这个运算符执行的操作在概念上是错误的。除了考虑语法之外，还要考虑运算符重载的用例和理论。

第 16~20 章和第 25 章：标准库

如前所述，标准库的某些方面很难使用。面试官很少要求求职者背诵标准库类的细节，除非求职者自称是一名标准库专家。如果求职者知道所面试的工作要使用大量标准库，就可能要在前一天编写一些标准库代码来温习一下。否则，回顾标准库的高层次设计和基本用法就足够了。

要记住的内容

- 不同类型的容器及其和迭代器的关系
- `vector` 的基本用法，`vector` 可能是最常用的标准库类
- `span` 类以及使用它的原因（C++20）
- 关联容器的使用，如 `map`
- 关联容器和无序关联容器(如 `unordered_map`)的区别
- `<bit>` 提供的位操作（C++20）
- 如何使用函数指针、函数对象和 `lambda` 表达式
- 标准库算法的作用和一些内置算法
- `lambda` 表达式和标准库算法的结合使用
- 删除-擦除惯用方法
- 很多标准库算法都有并行执行选项
- 扩展标准库的方式(通常不必了解细节)
- `range` 库的表达能力（C++20）
- 关于标准库的看法

问题类型

如果面试官很固执地要问标准库的细节问题，就无法界定他们要问的问题的范围。不过如果求职者对语法没有什么把握，就应该在面试的过程中明确表示出来——“当然，在现实生活中，我会查阅《C++高级编程(第 5 版)》这本书，但我肯定它能够这样工作……”至少通过这种方式提醒面试官，应该不必考虑这些细节，只要求求职者的基本思路正确即可。

关于标准库的高层次问题往往用来衡量求职者使用了多少标准库，而不必要求求职者回顾所有细节。例如，标准库的普通用户熟悉关联容器和非关联容器。稍微高级一些的用户能定义迭代器，描述迭代器操作容器的方式，阐述删除-擦除惯用方法，其他高层次的问题还包括询问求职者在标准库算法方面积累的经验，以及是否自定义过标准库。面试官还有可能考察求职者对 `lambda` 表达式的了解，以及它们与标准库算法的结合使用方法。

求职者可能还需要解释使用 C++20 `range` 库的好处。基本上，它允许编写描述你想做什么而不是如何做的代码。

第 22 章：日期和时间工具

C++标准库提供了 `chrono` 库，允许使用日期和时间。面试官不太可能询问有关此功能的详细问题，但他可能会评估求职者是否听说过标准库的这一部分。

要记住的内容

- 编译期有理数的使用
- 持续时间、时钟和时点的概念
- 日期和日历的概念 (C++20)
- 在不同时区之间转换日期和时间的可能性 (C++20)

问题类型

在这些主题中不要期待细节问题。不过,了解 C++17 的 `std::optional`、`variant` 和其他类肯定能够加分。可能要求解释 `chrono` 和随机数生成库的基本用法和概念,但不会涉及语法细节。如果面试官开始关注随机数,解释真随机数和伪随机数之间的差异非常重要,还应当说明随机数生成库会使用生成器和分布等概念。

面试官可能会解释一个涉及日期和时间的问题,并询问求职者如何解决,而不是询问任何关于 `chrono` 库的详细问题。如果求职者回答将实现自己的类来处理日期和时间,那么应该解释原因。更好的方法是回答将使用 `chrono` 库提供的功能。记住,代码的重用是一个重要的编程范式。

第 23 章: 随机数工具

在软件中生成好的随机数是一个复杂的话题,面试官知道这一点。他们不会问任何关于它的语法细节,但是你需要知道 `<random>` 库的基础和概念,这是 C++ 标准库的一部分。

要记住的内容

- 使用 `<random>` 库,作为生成随机数的首选方法
- 随机数引擎、适配器和分布如何协同工作以生成随机数
- 种子的概念以及它为什么重要

问题类型

面试官可能会向求职者展示一段代码,该代码使用 C 函数 `rand()` 和 `srand()` 生成随机数,并要求求职者对代码进行评论。求职者应该解释这些 C 函数不再被推荐,以及为什么使用 `<random>` 提供的功能更好。

在与随机数相关的问题中,解释真随机数和伪随机数之间的差异很重要。如果求职者还提到可以使用 `random_device` 为伪随机数生成器生成真正的随机种子,将获得额外的分数。

第 24 章: 其他库工具

该章介绍 C++ 标准库提供的值得一提的其他工具。面试官可能问到其中一些问题,以了解求职者的标准库知识的广度。

要记住的内容

- `std::variant` 和 `std::any` 数据类型,以及它们如何对 `optional` 进行补充

- `std::pair` 的泛化 `std::tuple`

问题类型

在这些主题中不要期待细节问题。不过，面试官可能会问到 `variant`、`any` 和 `tuple` 数据结构的使用场景。当解释 `variant` 和 `any` 时，求职者还可以将其与第 1 章中的 `optional` 类型对比。

第 27 章：C++多线程编程

随着多核处理器的发布，从服务器到 PC 都在使用多核处理器，多线程编程也变得越来越重要了。甚至智能手机也有多核处理器。面试官可能会问求职者几个多线程的问题。C++ 包含一个标准的线程库，因此最好了解这个线程库的工作方式。

要记住的内容

- 竞态条件和死锁，以及如何预防这些问题
- 通过 `std::thread` 产生线程
- `std::jthread` 提供的额外功能（C++20）
- 原子类型和原子操作
- 互斥的概念，包括不同互斥体和锁类的使用，提供线程间的同步
- 条件变量，以及如何通过条件变量给其他线程发信号
- 信号量、闭锁和栅栏的概念（C++20）
- `future` 和 `promise` 的含义
- 跨越线程的边界复制和重新抛出异常

问题类型

多线程编程是一个复杂的主题，所以不要期待有这方面的细节问题，除非求职者参加的是多线程编程岗位的面试。

面试官可能要求求职者解释多线程代码的不同类型问题，如竞态条件、死锁和撕裂。求职者还可能需要解释原子类型和原子操作的必要性，以及多线程编程的一般性概念。这是一个很宽泛的问题，可以让面试官了解求职者在线程方面的知识情况。解释互斥对象、信号量、闭锁和栅栏的概念将为求职者赢得额外的分数。求职者也可提及，很多标准库算法都有用来提升性能的并行运行选项。

第 28 章：充分利用软件工程方法

如果求职者经历一家公司的面试后，发现面试官没有问任何与软件开发过程有关的问题，那么求职者肯定感到可疑——这可能意味着这家公司没有任何过程或对此根本不关心。或者，还有可能他们不想因为庞大的过程而把求职者吓跑。在任何开发过程中，源代码控制都是重要一环。

大多数情况下，求职者都有机会询问有关公司的问题。建议求职者将有关软件工程过程和源代码控制解决方案的问题当成标准问题。

要记住的内容

- 传统的生命周期模型
- 不同模型的权衡
- 极限编程的主要原则
- Scrum 是一种敏捷过程
- 过去用过的其他过程
- 源代码控制解决方案的原则

问题类型

求职者会被问到的最常见问题是描述前雇主使用的过程。在回答这个问题时，应该提到什么行之有效，什么失败了，但不要谴责任何特定的方法。求职者批评的某个方法可能正是面试官使用的方法。

现在，几乎每个求职者都将 Scrum/敏捷列为一项技能。如果面试官问及 Scrum，他很可能并不希望你只是背诵课本上的定义。相反，可说出你认为的几个 Scrum 有趣之处。解释每一处，并融入自己的想法。尝试让面试官参与对话，根据面试官给出的线索，沿着他感兴趣的方向进行。

如果求职者遇到有关源代码控制的问题，这可能是一个高层次的问题。求职者应该解释源代码控制解决方案的一般性原则，提到有商业的和免费的开源解决方案，并讲解前雇主如何使用源代码控制。

第 29 章：编写高效的 C++ 程序

有关效率的问题在面试中非常普遍，因为很多组织都面临着代码可量测性的问题，需要精通性能的程序员。

要记住的内容

- 语言层次的效率很重要，但作用有限，设计级别的选择更重要
- 避免使用复杂度不好的算法，例如平方算法
- 引用参数的效率更高，因为引用可避免复制
- 对象池有助于避免创建和销毁对象的开销
- 性能剖析对于判断哪些操作消耗了最长的运行时间极为重要，所以不要尝试优化不是性能瓶颈的代码

问题类型

通常情况下，面试官会用自己的产品作为例子引出效率的问题。有时面试官会描述一种旧式设计，以及他们见过的一些和性能相关的症状。求职者应该提出一种新设计来缓解这些问题。遗憾的是，这有一个主要问题——求职者拿出的解决方案和公司解决问题时采用的解决方案相同的概率有多大呢？由于这个概率很小，因此解释自己的设计时需要格外小心。求职者可能拿不出公司实际采用的解决方案，但答案仍然是正确的，甚至比公司使用的新设计还好。

其他类型的效率问题可能要求求职者调整 C++ 代码以提高性能，或迭代某种算法。例如，面试官可能给求职者一些带有多余复制或低效循环的代码。

面试官可能还会要求求职者给出性能剖析工具的高层次描述，以及这些工具的好处。

第 30 章：熟练掌握测试技术

潜在雇主十分看重过硬的测试能力。如果没有专职 QA 经验，你的简历很可能不会提到测试技能。你可能遇到与测试相关的面试问题。

要记住的内容

- 黑盒测试和白盒测试的区别
- 单元测试、集成测试、系统测试和回归测试的概念
- 更高级的测试技术
- 以前所处的测试环境和 QA 环境：哪些可行，哪些不可行

问题类型

面试官可能要求在面试期间编写一些测试，但期间展示的程序不大可能包含相关测试的深度信息。更可能会问一些概括性的测试问题。准备好描述上份工作如何完成测试，以及你自己的看法。重申一次，不要泄露机密信息。回答面试官有关测试的问题后，可以问一下对方公司如何执行测试。这样可启动一次有关测试的对话，使你更好地了解潜在的工作环境。

第 31 章：熟练掌握调试技术

公司要找的求职者既要能调试自己的代码，又能调试从未见过的代码。技术面试往往试图估计出求职者的调试能力。

要记住的内容

- 调试不是在出现 bug 时才开始，应该事先在代码中准备好调试，这样在 bug 出现时才能有备无患
- 日志和调试器是最好的工具
- 断言的用法
- bug 表现出来的症状可能看上去和真实的原因没有关系
- 对象图可以帮助调试，特别是在进行面试时

问题类型

在面试过程中，求职者可能需要面对模糊的调试问题。记住，调试过程是最重要的，面试官可能知道这一点。即使求职者在面试过程中没有找到 bug，也一定要让面试官知道自己为跟踪这个 bug 所采取的步骤。如果面试官给出一个函数，并说明这个函数在执行时会崩溃，那么能够提出找到这个 bug 的正确步骤的求职者的得分，可能和能立即找到这个 bug 的求职者的得分相同，甚至更高。

第 32 章：使用设计技术和框架

第 32 章讨论的每种技术都是很好的面试题。不要只是简单地重复在第 1 章学到的内容，建议在参加面试前浏览一下第 32 章的内容，确保自己真正理解每种技术。

如果正在寻找一份与 GUI 相关的工作，那么需要了解 MFC 和 Qt 等框架。

第 33 章：应用设计模式

在专业领域，设计模式是十分流行的(甚至有很多求职者将这些列为自己掌握的技能)，面试官可能要求解释一种设计模式，给出设计模式的用例，或实现一种设计模式。

要记住的内容

- 将设计模式用作可重用的面向对象设计概念
- 本书中介绍的设计模式以及你在工作中使用的其他设计模式
- 求职者和面试官可能为同一种设计模式使用不同的名称，有数百种设计模式经常使用不一致的名称

问题类型

有关设计模式的问题通常都很好回答，除非面试官希望你讲清楚每种设计模式的详情。幸运的是，大多数喜欢设计模式的面试官只会与你讨论它们，并征求你的意见。对于大多数设计模式而言，只需要通过读书或上网来理解概念，并不需要死记硬背。

第 34 章：开发跨平台和跨语言的应用程序

大多数程序员递交的简历都会列出自己了解的多门语言或技术，大多数大型应用程序都依赖于多门语言或技术。即使只谋求一个 C++ 职位，面试官也会问及其他语言的问题，特别是与 C++ 相关的问题。

要记住的内容

- 平台的差异(架构、整数大小等)
- 为完成一项任务，应当尽量尝试寻找跨平台库，而不是针对不同种类的平台自行实现功能
- C++ 与其他语言的交互

问题类型

最常见的跨语言问题是比较两种不同的语言。不能仅凭个人好恶，单纯强调某种语言的优缺点。面试官希望你能进行权衡，并做出适当决策。

面试官最可能在讨论你以前从事的工作时谈到跨平台问题。如果简历中提到以前写过在自定义硬件平台上运行的 C++ 应用程序，那么应当谈一下自己使用的编译器以及在那个平台上遇到的挑战。



“随着新的程序员转向(或回归)C++语言来编写高要求的代码，C++本身也在不断发展（C++ 20就是近十年来最具里程碑的版本），大家迫切需要与时俱进的C++教程。编写C++ 20代码时常感觉就像在使用一种全新的语言，所以非常高兴看到像Marc这样知名且经验丰富的C++导师编写的这本书，帮助大家以全新视角来掌握焕然一新的C++语言。”

— 推荐人Herb Sutter, C++大师, ISO C++标准委员会召集人

★★★★★C++初学者的卓越教材

—Ettienne Hugo

作为一名在微软工作的专业C++程序员，我强烈向任何想学习C++的人推荐本书。我甚至向我的同事推荐本书，因为大家总需要学习和复习不经常使用的C++部分——本书深入研究了这些内容，真正达到了“高级编程”的水准。

不像大多数其他书停留在基本的语法/概念，本书仅在第1章介绍了基本的语法和概念，然后就推进到成为一名专业C++程序员所需要的高阶内容方面。此外，还向读者介绍了单元测试、编写可移植/高效的代码、软件架构和设计模式这些软件工程实践——正是这些内容确定了专业程序员与中级程序员的不同，即确保你不会写出糟糕的软件设计。最后，如果你考虑申请C++编程工作，本书甚至涵盖了技术面试。

我应该指出，没有一本书可以成为C++领域详尽的参考，但本书肯定会带你在这条路上走得很远，对你不断地进行帮助或提升。

★★★★★C++高级编程的优秀指南

—Lee G

我从20世纪90年代就开始编写C++代码。《C++20高级编程(第5版)》是我读过的第3个版本的C++高级编程。本书介绍的是现代C++20，帮助读者写出更好的现代C++代码，作者Marc是C++社区的专家。你可以在cppcon之类的会议上找到Marc的教学课程，这些课程被录制下来，并在youtube上发布。

使得本书从初学者书籍中脱颖而出的是，阅读本书就像与专家对话，这使它的阅读价值非常高。读完第1章，你就能理解现代C++的力量。如果你曾使用JavaScript、Java或早期版本的C++做过编程，那么你就完全准备好了编写内存安全和高效率的现代C++程序。书中还通过很多章节介绍设计、标准库用法和代码示例来巩固你的理解。

感谢Marc为这本阅读愉悦的C++教程及时更新了新版！（注：除了Marc的书和视频，我并不认识他本人）

Cover Design: Wiley
Cover Image: ©Ben Clift Williams/
EyeEm/Getty Images

 **WROX**
A Wiley Brand
www.wiley.com

本书在线资源



扫描下载

清华大学出版社



官方微信号

ISBN 978-7-302-60213-2



9 787302 602132 >

定价：228.00元(全二册)