C++17/Fortran2008 科学编程入门 ++17/Fortran2008 高性能计算的艺术，第三卷

Victor Eijkhout

2017–2022，格式化于 2023 年 8 月 19 日
Book and slides download: https://tinyurl.com/vle322course
Public repository: https://bitbucket.org/VictorEijkhout/intro-programming-public

目录

第一部分

引言

第一章

引言

1.1 编程与计算思维

在本章中，我们回顾了计算机和计算机编程的历史，并稍微思考了编程涉及的内容。

1.1.1 历史

在计算机的早期，硬件设计被认为是具有挑战性的，而编程不过是数据录入。事实上，最早的编程语言之一被称为 ‘Fortran’，因为

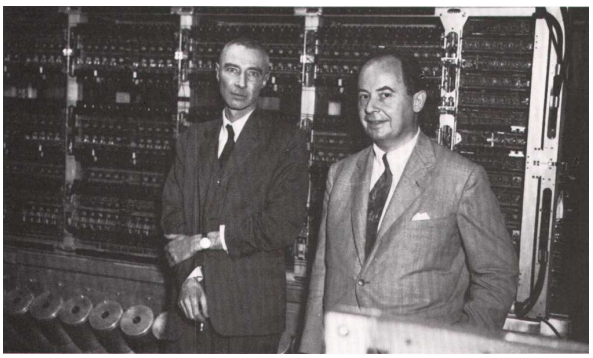


图 1.1：罗伯特 · 奥本海默和约翰 · 冯 · 诺依曼

‘公式翻译’，谈到这一点：一旦你有了数学，编程被认为不过是把数学翻译成代码。程序可能有微妙的错误，或 *bug*，这对最早的计算机设计者来说相当意外。

编程不被高度重视也产生了副作用，即许多早期的程序员是女性。在电子计算机出现之前，‘计算机’ 是一个执行计算的人，可能使用机械计算设备，而且通常是女性。因此，最早编程电子计算机进行这些计算的人，通常是受过数学教育的女性。两个著名的例子是海军少将格蕾丝 · 霍珀，她是 Cobol 语言的发明者，以及领导阿波罗计划软件开发的玛格丽特 · 汉密尔顿。这种情况在 20 世纪 60 年代后发生了变化，尤其是在个人计算机出现后 [1]。

1.

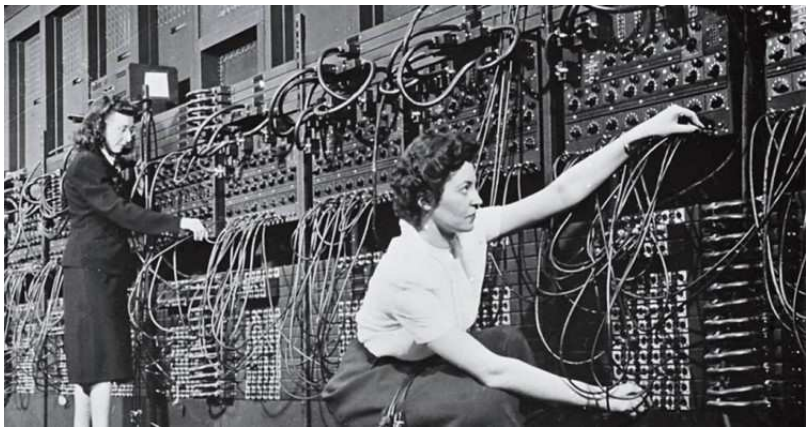


图 1.2：编程 ENIAC

1.1.2 编程是科学、艺术还是手艺?

如前一节所述，编程不仅仅是将数学翻译成硬件指令。它可以是科学吗？当然，编程确实有科学的一面：

- 算法和复杂度理论包含大量数学。
- 编程语言设计是另一个与数学相关的主题。

然而，编程本身并非一门科学。

“ 软件工程 ” 这个术语可能会让你以为设计和生产软件是一门工程学科，但这也不完全正确。软件工程师没有认证，而且也没有像土木工程和其他学科那样公认的技术体系。

编程在很大程度上是一门学科。构成一个良好程序的标准是个人喜好。但这并不意味着没有推荐的实践方法。在本课程中，我们将强调某些我们认为能写出良好代码的实践方法，同时也会禁止某些特定的编程习惯。

以上这些都不是一门精确的科学。存在多个程序可以给出正确的结果。然而，程序很少是静态的。它们往往需要被修改、扩展，甚至修复，如果发现错误行为，在这种情况下，一个写得不好的程序可能会降低程序员的效率。因此，一个重要的考虑因素是程序的可读性，无论是对于另一位程序员、本课程的教授，甚至是你两周后的自己。

1.1.3 计算思维

数学思维：

- 每天的人数，电梯的速度 $\Rightarrow$ 是的，可以让大家到达正确的楼层。
- 到达的人的分布等等。 $\Rightarrow$ 平均等待时间。

充分条件 = 存在性证明。

计算思维：解决方案的实际设计

- 电梯调度：地面有人按了按钮，5 层和 10 层有车；该派哪一辆下去?

Coming up with a strategy takes creativity!

练习 1.1。 一个简单的计算是最简单的算法示例 hm.

计算美国可以维持多少所美发师学校。确定相关因素，估计其规模，并进行计算。

Exercise 1.2. Algorithms are usually not uniquely determined:
there is more than one way solve a problem.

四辆自动驾驶汽车同时到达一个十字路口。请想出一个汽车可以遵循的算法来安全地穿过十字路口。如果你能想出不止一个算法，当使用不同算法的两辆车相遇时会发生什么?

在电话簿中查找姓名

- 从第 1 页开始，然后尝试第 2 页，等等
- 或者从中间开始，继续使用其中的一半。

这两种情况下的平均搜索时间是什么?

有一个正确的解决方案是不够的!

> 一种强大的编程语言作为一个框架，我们在这个框架中组织我们的想法。每一种编程语言都有三种机制来完成这个任务：
>
> - 原始表达式
> - 组合方式
> - 抽象方式
>
> *Abelson* 和 *Sussman*，《计算机程序的构造和解释》

- 电梯程序员可能想：‘如果按钮被按下’，而不是 ‘如果那根导线上的电压是 5 伏特’。
- 谷歌汽车程序员可能写的是：‘如果在我前面的车减速’，而不是 ‘如果我看到汽车图像在变大’。
- … 但可能另一个程序员不得不写那个翻译。

程序有抽象的层次。

抽象意味着你的程序谈论的是你的应用概念，而不是数字和字符之类的东西。

你的程序应该像是一个关于你的应用的故事；而不是关于比特和字节。 .

良好的编程风格使代码易于理解和维护。

(不良的编程风格可能导致成绩较低。)

> 有能力的程序员完全意识到自己头颅大小的严格限制；因此他以完全谦卑的态度对待编程任务，并且避免像瘟疫一样使用巧妙的技巧 —— 艾德格 · 迪科斯彻

你的程序中的数据结构是什么?

栈：你只能访问顶部元素

队列：元素在队尾添加，在 the front

程序包含支持算法的结构。你可能需要自己设计它们。

1.1.4 硬件

是的，它在那里，但我们在这个课程中不太考虑它。

高级程序员知道硬件影响执行速度；参见 HPC 书籍 [11], 章节 6.1。

1.1.5 算法

> 算法是一系列明确的指令，用于解决某个问题，即，在有限时间内为任何合法输入获得所需输出 [A。Levitin，《算法设计与分析导论》，Addison-Wesley， 2003]

这些指令用某种语言编写：

- 我们将教您 C++ 和 Fortran ；
- 编译器将这些语言翻译成机器语言

- 简单的指令：算术。
- 复杂的指令：控制结构 – 条件

- 循环

- 输入和输出数据：文件、用户输入、屏幕输出、图形。
- 程序运行期间的数据：
  - 简单变量：字符、整数、浮点数
  - 数组：索引字符集和类似
  - 数据结构：树、队列
    * 由用户定义，特定于应用程序
    * 存在于库中（ C/C++ 之间存在显著差异！）

1.2 关于语言选择

存在多种编程语言，并非每种语言都适用于所有目的。在本书中，您将学习 C++ 和 Fortran，因为它们特别适用于科学计算。而 ‘好’，我们指的是

- 它们可以表达科学计算中您想要解决的问题类型，并且
- 它们高效地执行你的程序。

还有其他语言可能不太方便或高效地表达科学问题。例如，*python* 是一种流行的语言，但如果你在编写科学程序，通常不是首选。作为一个例子，这里是一个简单的排序算法，用 C++ 和 python 编写。

Python vs C++ 在冒泡排序中：

```
f or i in range ( n -1) : for j in range (
n_i -1) : if numbers [j+1]<numbers [
j ]: swaptmp = numbers [j +1]
numbers [j +1] = numbers [j ]
numbers [j ] = swaptmp
```

```
for (int i=0; i<n-1; i++)
  for (int j=0; j<n-1-i; j++)
    if (numbers[j+1]<numbers[j]) {
      int swaptmp = numbers[j+1];
      numbers[j+1] = numbers[j];
      numbers[j] = swaptmp;
    }
```

$ pythonbubblesort.py5000 运行时间：12.1030311584 $ ./bubblesort 5000 运行时间：0.24121

但这忽略了一个问题：我们刚刚实现的排序算法实际上并不是一个很好的算法，事实上 python 内建了一个更好的算法。

快速排序算法的 Python：

```
numpy.sort(numbers,kind=_’quicksort’)
```

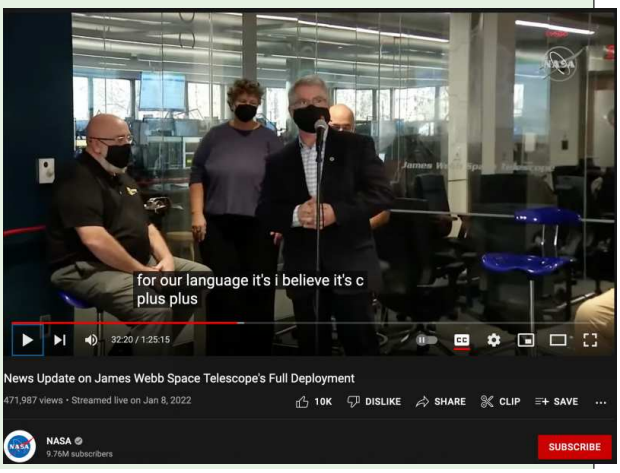
[ ]python arraysort.py 5000

```
        耗时：0.00210881233215
```

所以，在选择语言时，另一个考虑因素是：是否有现成的工具满足你的需求。这意味着你的应用程序可能会决定语言的选择。如果你被限制使用一种语言，不要重新发明轮子！如果有人已经编写过它，或者它是语言的一部分，就不要自己重新做。

C++ 规则的应用领域：

- 科学计算；与 C/Python 代码的互操作性。
- Embedded processors
- 游戏引擎



第 2 章

后勤

2.1 编程环境

编程可以用多种方式完成。可以使用集成开发环境（IDE），例如 *Xcode* 或 *Visual Studio*，但对于如果你要进行一些计算科学，你应该真正学习一个 *Unix* 变体。

- 如果你有一台 *Linux* 计算机，你就都准备好了。
- 如果你有一台 *Apple* 计算机，很容易让你开始。你可以使用标准的 *Terminal* 程序，或者你可以使用完整的 *X windows* 安装，例如 XQuartz，这使得 Unix 图形成为可能。这和其他 Unix 程序可以通过一个 包管理器，例如 *homebrew* 或 *macports* 获得。
- *Microsoft Windows* 用户可以使用 *putty* 但安装一个虚拟环境，例如 *VMware* (http://www.vmware.com/)或 *Virtualbox* (https://www.virtualbox.org/)可能是更好的解决方案。

接下来，你应该知道一个文本编辑器。最常见的两个是 *vi* 和 *emacs* .

2.1.1 你的编辑器中的语言支持

两个最流行的编辑器是 *emacs* 和 *vi* 或 *vim*。两者都支持编程语言，进行语法着色，并帮助你正确缩进。大多数情况下，你的编辑器会根据文件的 <code> 扩展 </code> 来检测文件所写的语言：

- cxx,cpp,cc for C++, and
  f90,F90 for Fortran.

如果您的编辑器无法检测到语言，您可以在文件顶部添加一行：

```
// -*- c++ -*-
```

for C++ mode, and

```
! -*- f90 -*-
```

用于 Fortran 模式。

主要优点是自动缩进（C++ 和 Fortran）以及提供块结束语句（Fortran）。编辑器还会应用 “ 语法着色 ” 来区分关键字和变量。

2. 物流

2.2 编译

‘程序’ 这个词含义模糊。有时它指的是 源代码：你用文本编辑器输入的文本。有时它指的是 可执行文件，你源代码的一个完全无法阅读的版本，计算机可以理解并执行它。将你的源代码转换为可执行文件的过程称为 编译，它需要一种称为 编译器 的东西。（那么谁写了编译器的源代码？好问题。）

以下是程序开发的工作流程

1. 你思考如何解决你的程序 2. 你使用编辑器编写代码。这会给你一个源文件。 3. 你编译你的代码。这会给你一个可执行文件。哦，更正一下：你尝试编译，因为很可能会有编译器错误：你违反语言语法的地方。 4. 你运行你的代码。很可能它不会完全按你的意图工作，所以你回到编辑步骤。

2.3 您的环境

以下练习适用于有中央类计算机的情况。为了证明您已经解决了连接问题，请执行以下操作。

练习 2.1。 进行在线搜索，了解计算机编程的历史。如果可能，用插图写一页，并将其转换为 PDF 文件。提交给您的老师。

第 3 章

教师指南

本书是为德克萨斯大学奥斯汀分校的一个学期 introductory 编程课程编写的，主要面向物理和工程科学专业的学生。因此，示例和练习尽可能具有科学动机。这个目标受众也解释了为什么包含 Fortran。

本书不是百科全书。作者没有全面讲解每个主题，而是采取了 “ 推荐实践 ” 的方法，让学生学习每个主题足够的内容，以成为一名称职的程序员。（你可能会问，由谁来推荐？作者坦率地承认自己受个人喜好的指导。然而，他让自己受到大量其他当前文献的启发。）这有助于将本书的篇幅控制在可管理的范围内，并尽量减少课堂讲授时间，强调实验练习。
即使如此，这里还有更多材料无法在一个学期内覆盖和练习。如果只教 C++ ，可能可以覆盖整个第二部分；对于同时教 C++ 和 Fortran 的情况，我们建议的时间表如下。

3.1 理由

第二部分 和第三部分 的章节以建议的教学顺序呈现。在这里，我们简要说明我们（非标准）主题排序的理由，并概述一个学期要涵盖的内容的时间表。值得注意的是，面向对象编程在数组之前涵盖，而指针则非常晚，甚至不涵盖。

背后有几个想法。首先，C 中的动态数组最容易通过 ++std::vector 机制来实现，这需要理解类。同样的，对于 `std::string`。

其次，在传统方法中，面向对象技术是在课程后期教授的，在所有基本机制（包括数组）之后。我们认为 OOP 是程序设计中的一个重要概念，是 C++ 的核心，而不是传统 C 机制的一种装饰，因此我们尽可能早地引入它。

甚至更基础的是，我们尽可能强调基于范围的循环而不是索引循环，因为范围在最近的语言版本中越来越重要。

3.1.1 算法

本课程中的一些编程练习要求学生复现存在于 `std::algorithm` 头文件中的算法。因此，本课程可能会受到批评，认为学生应该学习标准模板库（ STL ）中的算法，而不是自己重新创建它们。（参见第 14.3节）

我的辩护是，程序员应该知道的不仅仅是标准库中可以选用的算法。学生应该理解这些算法背后的机制，并能够复现它们，这样，当需要时，他们可以编写这些算法的变体。

3.2 C++/F03 课程的时间线

如前所述，本书基于一个同时教授 C++ 和 Fortran2003 的课程。这里我们给出所用的时间线，包括一些指定的练习。

对于一个略超过三个月的一个学期课程，两个月将花在 C++(见表 3.1)，之后一个月就足够解释 Fortran ； 见表 3.3。剩余时间将用于考试和选修主题。

3.2.1 高级主题

我们还概述了一个 ‘C++ 101.5’ 课程：介于入门和真正高级之间。在这里，我们假设学生已经学习了大约 8 个 C++ 讲座，涵盖了

1. 基本控制结构，2. 简单函数，包括参数通过引用传递，
3. 通过 `std::vector`。

Based on this, the topics in table 3.2 can be taught in that order.

3.2.2 基于项目的教学

在一定程度上，学生不可避免地会做一系列与任何先前或后续内容无关的练习。然而，为了提供一些连续性，本书包含一些编程项目，学生逐渐构建这些项目。

以下是一些简单的项目，这些项目有一系列相互依赖的练习：

Prime 素数测试，最终形成素数序列对象，并测试哥德巴赫猜想的一个推论。章节 45。

Geom 与几何相关的概念；这主要是一个面向对象编程的练习。章节 46。

Root 数值零点查找方法。章节 47。

以下项目是半严肃的研究项目：

Infect 传染病的传播；这些是面向对象设计的练习。学生可以探索各种现实场景。章节 49。

PageRank 谷歌 PageRank 算法。学生编程模拟互联网，并探索 PageRank，包括 ‘搜索引擎优化’ 。此练习使用大量指针。第 50 章

Gerrymandering Redistricting through dynamic programming. Chapter 51.

lesson	Topic	练习 课堂内	作业	质数	geom	感染
1	语句和表达式 - 式	4.5	4.10	45.1		
2	条件语句	5.2(S),5.3	5.4 ^(T)	45.2		
3	循环	6.5 （S）	6.6	45.3, 45.4		
4	continue	 6.4				
5	函数	7.1 7.2	7.6	45.6, 45.7 (T)		
6	continue	7.11			46.1	
7	I/O		12.1			
8	对象			9.8 (S), 45.8 (T), 45.10	46.3	49.1
9	继续					
10	has-a relation				46.9 (T), 46.10, 46.1, 46.12	49.4
11	继承				46.14, 46.15	
12	向量	10.1 (S), 10.2	10.21	45.18		49.4 and 进一步
13	继续					
14	字符串					

Table 3.1: Two-month lesson plan for C++; the annotation ‘(S)’ indicates that a skeleton code is available; ‘(T)’ indicates that a tester script is available.

				练习	
课程主题		书籍先决条件		课堂	作业
1	欢迎，账户				论文，coe_ 历史
2,3	Unix，编译，检查先前知识				Collatz: 6.13; swap: 7.5; 向量：10.19，coe_catchup
4, 5	测试驱动开发	68	分离编译 19	68.1 <S> 47.1–47.6	47.8 二分法 _
6, 7	对象	9		9.3 （S） 9.8 （S）	45.8 coe_primes
8	类包含	9.2		46.9, 46.12	
9	继承	9.3		46.14	45.9, 45.10 coe_goldbach
10	向量	10		10.1 <S>, 10.8, 10.9	
11	类中的向量	10.6		10.14 （S）	10.21 coe_pascal
12, 13	Lambda 函数	13		47.10 (S), 47.11	47.12, 47.13 coe newton_
14,15	STL，variant 可选	24.6.2		45.17	八皇后问题： 48
16	指针	16, 66.1.2			66.3–66.6
17	C 指针	17			
18	库和 cmake cxxopts fmt random 异常 正式 方法	63.1 24.7 23		63.2	

Table 3.2: Advanced lessons for C++; the annotation ‘(S)’ indicates that a skeleton code is available; ‘(T)’ indicates that a tester script is available.

lesson	Topic	Book	幻灯片	课堂内	作业
1	语句和表达 - sions	30	4F		
	条件 31		5F		
2	循环	32	6F		32.1
	函数	33	7F		
3	I/O	41	8F		
	数组	39	14F	39.1, 39.3, 39.5	
4	对象	38	10F	38.2	
	模块	37	9F (?)		37.2
5	指针	40	15F		

Table 3.3: Accelerated lesson plan for Fortran; the annotation '(S)' indicates that a skeleton code is available; '(T)' indicates that a tester script is available.

调度 探索与多旅行商问题（ MTSP ）相关的概念，建模亚马逊 *Prime*。章节 52。

Lapack 探索高性能线性代数。章节 53。
与其将项目练习包含在教学内容中，这些项目的每个部分列出了先决的基本部分。

项目作业提供了一个相当详细的逐步建议方法。这承认了认知负荷理论 [12]。

我们的项目非常基于计算。一种更接近 GUI 的项目教学方法在 [7] 中进行了描述。

3.2.3 选择：Fortran 或高级主题

在 C++ 中进行了两个月面向对象编程的基础训练后，Fortran 讲座和练习重演了这一序列，让学生在 Fortran 中做他们在 C++ 中做的相同练习。然而，Fortran 中的数组机制值得单独开一堂课。

3.3 框架

在介绍新主题时，我们尽可能提供可运行的代码，第一个练习是对这段代码的修改。在代码仓库的根目录下有一个名为 skeletons 的目录，其中包含示例程序。在上述表格中，给出了骨架代码的练习用 '(S)' 标记。

3.4 评分指南

一个程序不仅仅是得到正确的输出。人们常说程序被阅读的次数比被执行的次数多。因此它需要以某种方式编写，以说服读者其正确性。

3.4.1 编程规范

在本节中，我们概述了一些风格要点：使用它们的代码可能会给出正确答案，但它们违背了通常被认为是良好或干净代码的原则。

以下是评分时应视为负面因素的一些一般性指导原则。

总体指导原则应该是：

> 代码主要是供人类阅读，其次才是供计算机执行。这意味着除了正确之外，
> 代码还必须说服读者结果是正确的。

作为推论：

> 代码只有在正确时才应该被优化。巧妙的技巧会降低可读性，并且只有在确实需要时
> 才应该应用。

在本书的章节中，我们也将参考 *C* 的核心指南
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines 针对特定主题的指南。

3.4.2 代码布局和命名

代码应该使用正确的缩进。错误的缩进会误导读者。

不明显的代码片段应该被注释，但变量和函数的正确命名在很大程度上有助于减少这种紧迫性。

3.4.3 基本元素

- 变量应该具有描述性名称。例如，`count` 不是描述性的：一半的所有整数都用于计数。计数什么？
- 不要使用全局变量：

```
int i;int
main() {cin >>
i;}
```

3.4.4 循环

循环变量应该在循环头部声明，如果没有压倒性的理由将其声明为全局的。

局部声明：

```
for ( int i=0; i<N;i++)
{// 对 i 进行某些操作 }
```

global:

```
int i;
for ( i=0; i<N; i++) {
  // something with i
  if (something) break;
}
// look at i to see where the break
    was
```

如果索引不是严格必需的，请使用基于范围的循环。

3.4.5 函数

在 main 之前完全定义函数，或在 main 之后仅声明它，没有偏好。

Defined before:

```
bool f( double x ) {
  return x>0;
}
int main() {
  cout << f(5.1);
}
```

Only declaration given before:

```
bool f(double x);
int main() {
  cout << f(5.1);
}
bool f( double x ) {
  return x>0;
}
```

仅使用 C++- 风格的按值或引用传递参数：不要使用 C 风格的 `int*` 参数等。

3.4.6 作用域

变量应在使用它们的最近局部作用域中声明。在 Fortran 中需要在子程序的开始处声明所有变量，但在 C++ 中应避免这样做。

3.4.7 类

- 不要声明数据成员 `public`。
- 仅在他们确实需要时才编写访问器函数。
- 确保方法名称能够描述该方法的作用。
- 关键字 `this` 几乎很少需要使用。这通常意味着学生查阅了过多的 stackoverflow。

3.4.8 数组与向量

- 不要使用旧式 C 数组。

  ```
  int 索引[5];
  ```

- 绝对不要使用 *malloc* 或 `new`。
- Iterator (*begin*, *erase*) are seldom used in this course, and should only be used if strictly needed, for instance with ‘algorithms’.

3.4.9 字符串

不要使用旧式 C 字符串：

char *words "=" "and another thing";

3.4.10 其他说明

3.4.10.1 未初始化的变量

未初始化的变量可能导致未定义或不确定的行为。换句话说，就是错误。

int i_solution; int j_solution; bool found_solution; for(int i = 0; i < 10; i++){ for(int j = 0; j < 10; j++){ if(i*j >n){ i_solution= i; j_solution= j; found_solution = true; break; } } if(found_solution){break;} } cout <<i_solution << "," << j_solution << endl; %% icpc -o missinginit missinginit.cpp && echo 40 | ./missinginit 0, -917009232 6, 7

如果编译器或运行时系统可以检测到这个问题，那么整个问题都可以避免。代码结构通常会阻止检测，但在原则上总是可以进行运行时检测。

例如，Intel 编译器可以安装 arun-timecheck: %% icpc -check_uninit -o missinginit missinginit.cpp && echo 40 |./missin 运行时检查失败：变量 ’found_solution’ 正在被 mis 使用 Aborted (core dumped)

3.4.10.2 清理对象和向量

以下习语经常出现：

```
vector<int> testvector;for ( /*可能性
*/) {testvector.push_back(
something);if(
testvector.sometest() )// 记住这是最好
的 testvector.clear();}
```

类似的习语出现在类中，学生们为其赋予了一个 `reset()` 方法 .

通过在循环内部声明 testvector 来更优雅地完成：然后重置将自动处理。

在此处的总体原则是实体需要尽可能声明为局部变量。

第二部分

C++

第 4 章

C++ 的基本元素

4.1 从头开始：编译 C++

在本章和下一章中，你将学习 C++ 语言。但首先我们需要一些外部工具：你从哪里获取程序以及如何处理它?

In programming you have two kinds of files:

- 源文件，这些文件对你来说是可理解的，并且你使用编辑器（如 vi 或 emacs ）创建它们；以及
- 二进制文件，这些文件对计算机来说是可理解的，但对你来说是不可读的。

您的源文件由编译器翻译成二进制，该编译器 ‘ 编译 ’ 您的源文件。

让我们看一个例子：

```
icpc -o myprogram myprogram.cpp
```

这意味着：

- 您有一个源代码文件 myprogram.cpp ；
- 并且您希望输出一个名为 myprogram 的可执行文件，
- 并且你的编译器是 Intel 编译器 *icpc*。（如果你想使用 GNU 项目中的 C++ 编译器，你指定 *g++* ；clang 项目的编译器是 *clang++*。）

我们来做一个例子。

练习 4.1。 创建一个文件 zero.cc，包含以下行：

```
// /null.cpp
#include <iostream>
using std::cout;

int main() {
  return 0;
}
```

然后编译它。Intel 编译器：

```
icpc -o zeroprogram zero.cc
```

运行此程序（它不会输出任何内容）：

```
./zeroprogram
```

In the above program:

1. 前两行是魔法，目前先这样。始终包含它们。好的，如果你想了解：`#include` 这一行是一个 预处理 (章节 21) 指令；它将一个 头文件 包含到你的程序中，使某些功能可用。2. `main` 行指示程序开始；在它的括号内将是程序语句。 3. `return` 语句表示你的程序成功完成。（如果你想知道这些的细节，请参阅章节 4.6.1。）

如果你按照上述说明操作，并且正如你可能猜到的，当你运行它时，你会发现这个程序不会输出任何内容。

如果你在当前目录中执行 ls，你会看到你现在有两个文件：源文件 zero.cc 和可执行文件 zeroprogram。在如何选择这些名称方面，你有一定的自由度。

文件名可以有扩展名：点号后的部分。（点号前的部分完全由你决定。）

- program.cpp 或 program.cc 或 program.cxx 是 C++ 源代码的典型扩展名。
- program.cpp 可能与 ‘C 预处理器’ 产生混淆，但它似乎是标准用法，因此我们将在本课程中使用它。
- 使用没有扩展名的 program 通常表示一个可执行文件。（如果你在编译行中省略了 -o myprogram 部分，会发生什么？）

让我们让程序做些事情：在屏幕上显示一条 ‘hello world’ 消息。目前，只需复制这一行；它的具体含义稍后会解释。

练习 4.2。 添加这一行：

```
// /hello.cppcout <<"Hello world!"<< ’ \
n’ {v9};
```

(从 pdf 文件中复制是危险的！请自行输入)

编译并重新运行。输出是什么？

测试你对编程中涉及的文件类型的了解程度！

复习 4.1. 对或错？

1. 程序员只编写源文件，不产生二进制文件。2. 计算机只执行二进制文件，不处理人类可读文件。

4.1.1 关于 Unix 命令的简要介绍

编译命令行

```
g++ -o myprogram myprogram.cpp
```

可以认为由三个部分组成：

- 开始行的命令 g++，它决定了将要发生什么；
- 行末的参数 myprogram.cpp 是命令处理的主要对象；和
- 选项 / 值对 -o myprogram。大多数 Unix 命令都有各种选项，正如其名称所示，这些选项是可选的。例如，你可以告诉编译器尽力使程序运行得很快：

```
g++ -O3 -o myprogram myprogram.cpp
```

选项可以按任何顺序出现，因此最后一个命令等效于 g++ -o myprogram -O3 myprogram.cpp

小心不要将参数和选项混淆。如果你输入

```
g++-o myprogram.cpp myprogram
```

那么 Unix 会这样推理：“myprogram.cpp 是输出文件，所以如果该文件已经存在（是的，它确实存在），我们就在做任何其他事情之前先清空它”。这样你就丢失了你的程序。幸好像 *emacs* 这样的编辑器会保留你的文件的备份。

4.1.2 构建环境

在上面我们并没有说明如何编译你的源文件。此外，虽然你看到了一个编译命令，但你可能不需要手动执行这些命令。这完全取决于你的构建环境有多复杂。这里我们提供一些可能性，但本课程将选择权留给你自己。

4.1.2.1 命令行和编辑器

传统的软件开发方式是使用编辑器 —— 例如 *emacs, vim, nano* —— 并在 Unix 或其他操作系统的命令行上输入编译命令。

如果你精通 Unix，这并不是一个坏策略。你可以通过使用 `Make` 或 `CMake` 来简化生活，但这主要适用于本课程前几章中你不会遇到的那种更复杂的程序。

4.1.2.2 集成构建系统

有一些带有图形用户界面的非常好用的程序，如果你不是一个 Unix 爱好者，它们可以让软件开发不那么痛苦。它们可以帮助你编辑文件，编译只是一个按按钮的事情。

针对 Apple，有 *XCode*，而微软的商业产品中有 *Visual Studio*。后者有一个免费（且有限）的版本 *VSCode*。另一个广泛使用的商业产品是 *CLion*。

最后，还有开源 *Eclipse* 软件。

4.1.3 C++ 是一个移动目标

The C++ 语言已经经历了一系列标准。((这在第 26.7 节中有详细描述。) 在本课程中，我们关注最近的标准：C++17，以及一定程度上 C++20。不幸的是，你的编译器默认会假设一个较早的标准，因此在这里教授的结构可能被标记为语法错误。

你可以告诉你的编译器使用一个现代标准：

```
icpc -std=c++17 [other options]
```

但要节省大量输入，你可以定义

```
alias icpc='icpc -std=c++17'
```

在你的 shell 启动文件中。在类 isp 机器上，这个别名已经被默认定义。

4.2 语句

每种编程语言都有其自己的（非常精确！）关于源文件中可以包含什么内容的规则。总体来说，程序包含计算机执行的指令，这些指令以一系列 “ 语句 ” 的形式存在。以下是一些关于语句的规则；你将在阅读本书的过程中更详细地学习它们。

- 程序包含语句，每个语句以分号结束。
- “花括号 ” 可以包含多个语句。
- 一个语句可以在程序执行时对应某些操作 .
- 一些语句是定义，定义数据或可能的操作。
- 注释是 “ 自我提醒 ”，简短：

```
cout << "Hello world"<< '\n'; // say hi!
```

and arbitrary:

```
cout << /* we are now going
            to say hello
         */ "Hello!" << /* with newline: */ '\n';
```

在之前的示例中，您看到输出语句以以下方式终止：

cout <<something << "\n";

其中 “ 反斜杠 -n” 表示一个换行。

练习 4.3。 从你的打印语句中移除换行符。编译并运行。你观察到什么？

有时你也会看到：

```
// 在程序的顶部: using
std::endl;

// among the statements:
cout << something << endl;
```

它具有发出换行符的相同行为。目前，这种区别对你来说并不重要；如果你好奇，请参阅第 12.5 节中的讨论。

练习 4.4. 复制你上面编写的 ‘hello world’ 程序中的 hello 行。编译并运行。

将两个 hello 放在文件的同一行上，还是放在不同行上，是否有区别?

尝试对你的源代码布局进行其他更改。找到一个至少会导致编译器错误的更改。你能将消息与错误联系起来吗?

您的程序源代码可以有几种类型的错误。根据您发现错误的时间，我们大致将它们分为以下几类。(有关错误处理的详细信息，请参阅第 23 章)

> 1. 语法 或编译时 错误：这些错误发生在您编写的代码不符合语言规范时。编译器会捕获这些错误，并且拒绝生成二进制文件。2. 运行时 错误：这些错误发生在您的代码在语法上正确，编译器已经生成了一个可执行文件，但程序的行为不符合您的预期或预见。例如除零错误或数组索引超出范围。3. 设计错误：您的程序没有做您认为它应该做的事情。

复习 4.2. 对还是错?

- 如果您的程序编译正确，它就是正确的。
- 如果您运行您的程序并且得到正确的输出，它就是正确的。

在您刚刚编写的程序中，您显示的字符串完全由您决定。其他元素，例如 `cout` 关键字，是语言中固定的部分。大多数程序都包含它们。

您看到您的程序中某些部分是不可侵犯的:

- 存在 关键字 如 **`return`** 或 `cout`; 您不能改变它们的定义。
- 花括号和括号需要匹配。
- 必须有一个 *`main`* 关键字。
- 通常需要 *`iostream`* 和 `std`。

4.2.1 语言与库 and 关于：使用

上述示例中有一条行

```
#include <iostream>
```

这允许你编写 cout 在你的程序中用于输出。 iostream 是一个 头文件，它为基本语言添加了标准库 功能。
像 cout 这样的功能可以用多种方式使用：

You can spell it out as std::cout:

```
#include <iostream>
int main() {
  std::cout "hello\n";
  return 0;
  }
```

You can add a **using** statement:

```
#include <iostream>
using std::cout;
int main() {
  cout "hello\n";
  return 0;
  }
```

而不是为每个库函数使用单独的 **using** 语句，你也可以使用一行

```
using namespace std;
```

在你的程序中。虽然在线示例中常见这种情况，但这是不被提倡的；参见 20.4 节进行讨论。

练习 4.5. 尝试使用 cout 语句。将字符串替换为一个数字或数学表达式。你能猜到如何打印多个东西，例如：

- 字符串 One third is，和
- 计算的结果 1/3,

使用相同的 cout 语句？你得到任何意外的东西吗？

4.3 变量

没有存储数据，程序将无法做太多：输入数据、用于中间结果的临时数据以及最终结果。数据存储在变量中，它们具有

- 一个名称，以便您可以引用它们，
- 一个 数据类型，以及
- 一个值。

将变量视为内存中的一个标记位置。

- 变量在一个变量声明中定义，
- 其中可以包含一个变量初始化。
- 变量定义后，赋予值，就可以使用，
- 或在变量赋值中赋予（新的）值。

```
int i, j; // 声明 i   = 5; // 赋值 i   = 6; //
赋新值 j =i+1; // 使用 i 的值 i   = 8; // 改
变 i 的值 // 但这不会影响 j:
```

```
            // it is still 7.
```

4.3.1 变量声明

变量被定义，在 变量声明。这将其名称和类型关联起来，并且可能提供一个初始值；参见章节 4.3.2。

让我们首先讨论变量名可以是什么。

- 变量名必须以字母开头；
- 名称可以包含字母和数字，但不能包含大多数特殊字符，除了下划线。

- 对于字母来说，使用大写还是小写很重要：该语言是区分大小写的。
- 诸如 `main` 或 **`return`** 之类的单词是 保留字。
- 通常 `i` 和 `j` 不是最好的变量名：使用行 和 列，或其他有意义的名称，而不是它们。

- 虽然你可以用下划线开头命名，但在使用下划线时有一些限制：不要连续使用两个下划线，也不要以下划线开头，后面跟一个大写字母命名。

接下来，一个 变量声明 声明了变量的类型和名称。如果你有多个相同类型的变量，你可以将声明组合在一起。

变量声明建立了变量的名称和类型：

int n_elements; float value;
*int row, col; double re_*
part, im_part;

你不能像这样重新声明变量：

int value=5;*cout value*<<
value<< '\n';*float value*=1.
3;cout value<< *value*<< '\n';

但是允许的内容规则有点难以陈述。你将在第 8 章 中看到。

声明可以放在程序的几乎任何地方，但它们必须在变量的第一次使用之前。

注意：在主程序之前定义变量是合法的，但这样的 全局变量 通常不是一个好主意。请仅在主程序（或在函数中等）中声明变量内部 。

Review4.3. 以下哪些是合法的变量名?

1.
mainprogram 2.
main 3. Main

4. 1forall 5.
one4all 6. one
for all_ _7.
onefor{all}

4.3.2 初始化

可以在创建变量时立即给它赋值。这被称为 初始化 ， 它与创建变量然后后来再赋值（第 4.3.3 节）不同。

初始化变量有两种（至少）方法

```
int i = 5;
int j{6};
```

注意编写

```
int i; i
 = 7;
```

is 不是初始化：它是一个声明后跟一个赋值 nment.

如果你声明一个变量但没有初始化，你无法指望它的值是什么，特别是零。虽然一些编译器会这样做（有时），但这种隐式初始化也经常因为性能原因被省略。

4.3.3 赋值

设置一个变量

```
i = 5;
```

意味着将值存储在内存位置。它与定义数学等式不同

let $i = 5$.

一旦声明了变量，就需要为其赋值。这通过一个赋值语句来完成。在上述声明之后，以下是一些合法的赋值：

```
n ; x.5 -n ;
n1 = 7; n
2 = n1* 3
```

这些不是数学方程式：左侧变量获得右侧表达式的值。

你可以赋值一个简单值或一个表达式。

你可以多次设置变量的值。

```
int i; i= = 5; // do
something with i i= = 6; //
do something with i
```

You can also update the value of a variable, using its current value:

```
i = 2*i + 1;
```

某些在左右两侧使用相同变量的赋值可以简化：

```
x =x+2; y = y/3;// 可
以写成 x  += 2; y/= 3;
```

整数加 / 减一：

```
i=i+1; j=j-1;//
重写为: ++i; --
j;// 或 i++; j--;
```

Exercise 4.6. 以下哪些是合法的？如果是，它们的意思是什么？

```
1. n =
n; 2. n  = 2
n; 3. n =
n2; 4. n  = 2
*k; 5. n/2 =
k; 6. n /=k;
```

编程错误有多种级别。以下程序使用了变量 `i` 而没有给它赋值。

Exercise 4.7.

```
#include <<iostream>><<using>{std}{v9}::{v11}cout{v15};<int>{v17}main{v19}() {<int>{v23}i{v25};<int>{v29}j =<int>{v34}i+1;<int>{v37}cout<<
<int>{v42}j<<  "\n"{v45};<return>{v49}<int>{v51}0; }</return>
```

会发生什么？

1. 编译错误 2. 输出：13. 输出未定义

4. 运行程序时出错。

4.3.4 数据类型

你已经见过一些变量可以有的数据类型。我们将更详细地讨论数据类型的问题。

变量有不同的类型；

- 我们称类型为 `int` 、 `float` 、 `double` 的变量为 < 样式 id='10'> 数值变量 </ 样式 >。
- 复数将在后面讨论。
- 对于字符：`char`。字符串比较复杂；后面会讲。
- 布尔值：`bool`
- 你可以创建自己的类型。后面会讲。

对于复数，请参见第 24.1 节。对于字符串，请参见第 11 章。

4.3.4.1 Integers

从数学上讲，整数是实数的一个特例。在计算机中，整数与实数（或从技术上讲，浮点数）的存储方式非常不同。

你可能认为 C++ 整数是以带符号位的二进制数存储的，但事实更为微妙。目前，只需知道在一定范围内，大约以零为中心对称，所有整数值都可以表示。

练习 4.8。 如今，`int` 的默认存储量是 32 位。使用 1 位表示符号后，还剩下 31 位表示数字。可表示的整数范围是多少?

C 语言中的整数类型 ++ 是 `int`:

```
int my_整数;my_整数 = 5;
cout <<my_整数<< "\n";
```

更多整数类型，请参见章节 26.3 ；如果你想知道整数和类似类型可以有多大，请特别参见节 24.2。

整数常量可以在几种进制中表示。

整数通常以十进制形式书写，并以 32 位存储。如果你需要其他形式:

```
intd = 42;into = 052; // 以零开
始 intx = 0x2a;intX = 0
X2A;intb = 0b101010;long ell
 = 42L;
```

二进制数对 C++17 来说是新的。

4.3.4.2 浮点数

浮点数 是计算机科学中的名称，用于科学计数法：像这样写的数字

$$+6 \cdot 022 \times 10^{23}$$

with:

- 一个可选符号；
- 一个整数部分；
- 一个小数点，或者更一般地基数点 在其他数制中；
- 一个分数部分，也称为 尾数 或 有效数字；
- 以及一个指数部分：基数到某个幂。

浮点数默认为 **double** 类型，其标准为 ‘双精度’。双精度是什么？我们将在第 24.2 节中讨论。目前，我们只讨论它们如何表示。

如果没有进一步指定，浮点字面量是 **double** 类型：

```
1.5 1.5e
+5
```

使用后缀 `1.5f` 表示类型 **float**，它代表 ‘单精度’：

```
1.5f1.5e+5
f
```

使用后缀 `1.5L` 表示 **long double**：四倍精度。

```
1.5L1.5e+5
L
```

有一种方法可以给出浮点数的十六进制表示，但这很复杂。

4.3.4.2.1 **存储大小** 浮点类型的精确定义如下。

最常用的浮点类型是：

- **float**，即IEEE 32 位单精度类型，
- **double**, IEEE 64 位双精度类型，
- **long double**, IEEE 四倍精度类型。

C++23 标准添加了（可选）固定宽度浮点数在 *stdfloat* 头文件中：

```
float16_t // 16-bit half precision
float32_t // 32-bit single precision
float64_t // 64-bit double precision
float128_t // 128-bit double precision

bfloat16_t// ‘bfloat’ 半精度
```

注意存在两种 16 位浮点数类型。*float16_t* 是 IEEE 754 标准最初定义的类型；*bfloat16_t* 是 IEEE 32 位类型，但截断了最后两个字节。这在机器学习（ML）应用中是有意义的，无论是从精度角度，还是因为使用 32 位浮点数带宽增加的角度来看。

注意 1 半精度和四倍精度类型可能在语言实现中可用，但没有硬件支持，这会使它们的执行非常慢。

4.3.4.2.2 **局限性** 浮点数也被称为 ‘实数’ （实际上，在 Fortran 语言中它们是用关键字 `Real` 定义的），但这是一种不严谨的说法。由于只有有限的位数 / 数字可用，因此只能表示有限小数。例如，由于计算机数字是二进制的，$1/2$ 是可表示的，但 $1/3$ 是不可表示的。

练习 4.9. 你能想到一种方法，使得非有限小数（包括像 $\sqrt{2}$ 这样的数）仍然可以被表示吗?

- 你可以将一种类型的变量赋值给另一种类型的变量，但这可能会导致截断（将浮点数赋值给整数）或意外的位（将单精度浮点数赋值给双精度）。

浮点数的行为不像数学中的数；关于广泛的讨论，请参阅 HPC 书籍 [11]，节 3.3 及以后。

Floating point arithmetic is full of pitfalls.

- 不要指望 `3*(1./3)` 正好等于 1。
- 甚至不满足结合律。

复数存在，参见第 24.1 节。

4.3.4.3 Boolean 值

到目前为止，你已经看到了整数和实数变量。还有布尔值，它们表示真值。只有两个值：`true` 和 `false`。

```
bool 类型{false};类
型=true;
```

4.4 输入 / 输出，或我们说的 I/O

程序通常会产生输出。目前我们只会将输出显示在屏幕上，但也可以将输出到文件。关于输入，有时程序拥有其计算所需的所有信息，但也可以基于用户输入进行计算。

Terminal (console) output with `cout`:

```
float x = 5;cout << "Here is the root: " << sqrt(x) << '\
n';
```

注意换行字符。或者：`std::endl`，
效率较低。

您可以通过 `cin` 从键盘获取输入，它接受任意字符串，只要它们不包含空格。

```
// /cin.cpp
   字符串 姓名; int 年龄;
   cout <<"Your name?\n";
   cin >>name;
   cout <<"age?\n";
   cin >>年龄 ;
   cout <<年龄  << " is a nice age, "
        << name << '\n';
```

```
> ./cin
你的名字?
维克多
age?
18
18 是一个很棒的年龄，维克多
> ./cin
你的名字?
THX 1138
age?
1138 是一个不错的年龄，THX
```

为了更灵活的输入，请参阅第 12.6 节。

为了对输出进行细粒度控制，请参阅第 12.2 节。对于其他与 I/O 相关的事项，例如文件 I/O，请参阅第 12 章。

4.5 Expressions

计算中最基本的步骤是形成表达式，例如求和、乘积、逻辑合取、字符串连接，这些表达式来自变量和常量。

让我们从讨论常量开始：数字、真值、字符串。

4.5.1 数值表达式

编程语言中的表达式大多看起来符合你的预期。

- 数学运算符 :+ - / 和 * 用于乘法。
- 整数取模 : 5%2
- 你可以使用括号 : 5*(x+y)。如果不确定运算符的优先级规则，请使用括号。

- C++ 没有幂运算符（ Fortran 有）： ‘幂’ 和各种数学函数通过库调用实现。

数学函数位于 *cmath*:

```
#include <cmath>
.....
x = pow(3,.5);
```

For squaring, usually better to write `x*x` than `pow(x,2)`.

Exercise 4.10. Write a program that :

- 显示消息类型 a 数字,
- 接受一个整数数字（使用 `cin` ），

- 使另一个变量成为该整数的 3 倍加 1，
- 然后打印出第二个变量。

整数和整数表达式的细节在 26.3.1 节中讨论。

4.5.2 布尔值

除了数值类型之外，还有布尔值，`true` 和 `false`，它们定义了所有常用的逻辑运算符。

- 关系运算符 :== != < > <= >=
- 布尔运算符：not，and，or （旧式：! && ||）;

4.5.3 类型转换

由于变量有一个类型，并且始终是那种类型，你可能想知道当

```
float x = 1.5;
int i;
i = x;
```

or

```
int i = 6;
float x;
x = i;
```

- 将浮点值赋给整数会截断后者。
- 将一个整数赋值给浮点变量时，会在小数点后用零填充它。

练习 4.11。 尝试以下内容：

- 当你将一个正浮点值赋值给整数变量时会发生什么？当你将一个负浮点值赋值给整数变量时会发生什么？你的编译器会给出警告吗？有没有什么方法可以欺骗编译器，让它不理解你在做什么？
- 当你将一个 `float` 赋值给 `double` 时会发生什么？尝试对原始的 float 使用不同的数字。打印出结果，如果它们看起来相同，看看差异是否实际为零。

表达式中的类型转换规则并不完全符合逻辑。考虑

```
float x; int i=5,j=2;
x = i/j;
```

这将给出 2 而不是 2.5，因为 i/j 是一个整型表达式，因此完全按整型计算，截断后给出 2。最终赋值给浮点型变量这一事实并不会导致它按浮点数计算。

You 可以强制表达式按浮点数计算 numbe rs by writing

```
x = (1.*i)/j;
```

或任何其他强制转换的机制，而不改变结果。另一种机制是强制类型转换；这将在第 26.2 节中讨论。

练习 4.12。 编写一个程序，要求输入两个整数 n1,n2。

- 将整数比例 n_1/n_2 赋值给一个整数变量。
- 你能使用这个变量来计算模数

$$n_1 \mod n_2$$

(不使用 %modulus 运算符 !) 打印出你得到的结果。

- 还要打印出使用 modulus 运算符：% 的结果。
- 调查你的程序对于负输入的行为。你是否得到了你期望的结果?

练习 4.13。 编写两个程序，一个读取摄氏温度并转换为华氏温度，另一个执行相反的转换。

$$C = (F - 32) \cdot 5/9, \qquad F = 9/5\,C + 32$$

检查你的程序对于水的冰点和沸点。(你知道摄氏温度和华氏温度相同的温度吗?)

Can you use Unix pipes to make one accept the output of the other?

复习 4.4. 真或假?

1. 在一定范围内，所有整数都可以作为整数变量的值。 2. 在一定范围内，所有实数都可以作为浮点变量的值。 3. 5(7+2) 等价于 45。 4. 1--1 等价于零。 5. int i = 5/3.; 变量 i 是 2。 6. float x = 2/3; 变量 x 大约是 0.6667。

4.5.4 字符和字符串

在本课程中，我们主要关注数值数据，但字符串和字符数据对于输出目的也是有用的。

4.5.4.1 字符串

字符串，也就是说，字符序列，不是内置数据类型。因此，它们需要一些额外的设置才能使用。参见第 ++ 章的完整讨论。

对于字符，有 `char` 数据类型，而对于字符串 *string*，如果你想要使用字符串：

- Add the following at the top of your file:

```
#include <string>
using std::string;
```

- Declare string variables as

```
字符串名称;
```

- And you can now `cin` and `cout` them.

字符用单引号括起来：

```
'x'
```

而一般字符串用双引号括起来：

```
"The quick brown fox"
```

练习 4.14. 编写一个程序，要求用户输入其名字，使用 `cin` 读取该名字，并打印类似 Hello, Susan! 的响应。

如果你输入名和姓会怎样?

4.6 高级主题

4.6.1 主程序和返回语句

The 主 程序必须为 **int** 类型；然而，许多编译器允许偏离这一规则，例如接受 **void**，这并非语言标准。

main 的参数可以是：

int main()int main(int argc, char argv*

`[]` *)int main(int argc, char** argv)*

The *argc/argv* 变量包含命令行作为一组字符串。

- *argc* 是字符串的数量：程序的名称和空格分隔的参数数量；
- *argv* 包含命令行参数 作为字符串数组。

你可能想自己解析命令行，但为此有专门的库；参见章节 63.2.2。

返回的 **int** 可以以多种方式指定：

- 如果没有 **return** 语句，则隐式 **return** `0` 。
- 如果你显式使用 **return** 并带有整数值。
- 您可以使用隐式整数值，而不是显式整数值 *EXIT_SUCCESS* 和 *EXIT_FAILURE*，这些值在 *cstdlib* 中定义。通常，零表示成功，而非零值表示失败。
- 您还可以使用 *exit* 函数：

```
void exit(int);
```

设置返回代码的目的是因为它会被传递给操作系统作为 返回代码，然后可以在 *shell* 中查询。

代码：

```
// /return.cpp
int main() {
  返回 1;
}
```

Output [基本] 返回：

```
./返回; \
        如果 [ $? -不等于 0 ]; 则 \
            echo"程序失败" ;          \
        fi
程序失败
```

4.6.2 标识符名称

变量名称，或者更准确地说：标识符，必须以非数字开头。具体来说，这可以是

- 一个拉丁字母，这是最常见的情况；
- 一个 下划线，这是类私有成员的约定，以及其他 ‘内部’ 名称；或
- *Unicode* 类 XID Start 的字符。

任何后续字符可以是类 XID Continue 的 Unicode 字符。

关于下划线，不应使用以两个下划线开头的 双下划线，因为这些名称被保留给 <code> 编译 </code>。

备注 2 通用 *Unicode* 字符在 *C++23* 中被允许使用，但这一惯例后来被追溯应用于早期标准。

4.7 C 差异

4.7.1 布尔值

传统上，C 没有用于布尔值的类型；相反 `int` 和 `short` 被使用，其中零表示假，而非零值表示真。在 C99 中引入了类型 `_Bool`。这仅用于提高可读性：没有 true/false 常量，类型为 `_Bool` 的变量在 `printf` 中仍然必须被视为整数。

```
// /ctypes.c_ 布尔 = 1;
printf("True: %d\n",tf);
```

The `stdbool.h` 定义 `bool`、 `true` 和 `false` 为别名。

4.8 复习问题

Review4.5. 的输出是什么：

```
int m=32, n=17; cout n %
m << "\n";
```

Review4.6. 给定

```
int n;
```

给出一个使用基本数学运算符来计算 n 立方的表达式： n^3。对于所有 n，你是否得到正确的结果？解释。

计算机执行多少次基本操作来计算这个结果？

你现在能计算 n^6，同时尽量减少计算机执行的操作次数吗？

第 5 章

条件语句

一个仅由赋值语句和表达式组成的程序不会太灵活。至少你希望能够说 ‘如果某个条件，执行一个计算，否则执行其他计算’，或者：‘直到某个测试为真，迭代以下计算’。用于测试并相应选择操作的机制称为一个 条件语句。（迭代将在第 6 章讨论。）

5.1 条件语句

条件语句可以有多种形式。

单个语句，如果测试为真则执行：

```
if (x<0)
  x = -x;
```

在真分支中有一条语句，同样在假分支中有一条语句：

```
if (x>=0)
  x = 1;
else
  x = -1;
```

在真分支和假分支中都可以包含多条语句，只要你用大括号括起来：

```
if (x<0) {
  x = 2*x; y = y/2;
} else {
  x = 3*x; y = y/3;
}
```

你可以通过扩展 `else` 部分来链式条件语句。在这个例子中，点代表省略的代码：

```
if (x>0) {
  ....
} else if (x<0) {
  ....
} else {
  ....
}
```

条件也可以嵌套：

```
if (x>0) {
  if (y>0) {
    ....
  } else {
    ....
  }
} else {
  ....
}
```

- 在最后一个示例中，真值分支中的外层花括号是可选的。但无论如何使用它们更安全。
- 当你开始嵌套结构时，使用缩进来明确每一行所在的级别。一个好的编辑器会帮助你完成这些。

Exercise 5.1. For what values of x will the left code print 'b'?
For what values of x will the right code print 'b'?

```
float x = /* something */
if ( x > 1 ) {
   cout << "a" << endl;
   if ( x > 2 )
    cout << "b" << endl;
}
```

```
float x = /* something */
if ( x > 1 ) {
   cout << "a" << endl;
} else if ( x > 2 ) {
    cout << "b" << endl;
}
```

5.2 运算符

你已经看到了算术表达式；现在我们需要查看逻辑表达式：在条件语句中可以测试什么。总的来说，逻辑表达式是直观的。但是请注意，它们只能以特定方式链接：

```
bool x,y,z;
if ( x or y or z ) ; //good
int i,j,k;
if ( i < j < k ) ; // WRONG
```

Here are the most common 逻辑运算符s and 比较运算符 s:

运算符	含义	示例
==	等于	x==y-1
!=	不等于	x*x!=5
>	大于	y>x-1
>=	greater or equal	sqrt(y)>=7
<,<=	小于 , 小于等于	
&&,||	and, or	x<1 && x>0
and,or	and, or	x<1 and x>0
!	not	!(x>1 && x<2)
not		not (x>1 and x<2)

优先级 运算符的规则是常识。如有疑问，使用括号。

练习 5.2. 以下代码声称检测一个整数是否超过两位数。

代码：

```
1 // /if.cpp
2   int i;
3   cin >>i;
4   if ( i>100 )
5     cout << "That number" << i
6          "<< " 有超过 2 位数字
7          << '\n';
```

Output
[基本] if

```
... 以 50 为输入 .... 以 150
... w        as  npu   ....
那个数字 150 要多于 2
    digits
```

修复此代码中的小错误。另外，添加一个 “else” 部分，如果数字为负数则打印。

您可以根据代码仓库中的 *if.cpp* 文件进行参考

Exercise 5.3. Read in an integer. If it is even, print ‘even’, otherwise print ’odd’:

```
if ( /* 你的测试在这里 */ )
cout <<"even" << '\n';
elsecout <<"odd" << '\n';
```

然后，将您的测试重写为真实分支对应于奇数情况 .

在练习 5.3 中，使用偶数测试还是奇数测试并不重要，但有时如何排列复杂的条件会影响结果。在接下来的练习中，思考如何安排测试。方法不止一种。

练习 5.4。 读取一个正整数。如果是 3 的倍数则打印 “Fizz！”；如果是 5 的倍数则打印 “Buzz!”。如果是 3 和 5 的公倍数则打印 “Fizzbuzz!”。否则不打印任何内容。

Note:

大写。
感叹号。
- 您的程序应最多显示一行输出。

5.2.1 按位逻辑

前面我们只考虑了 **和** 以及 **或** 逻辑运算符，也写作 && 和 ||。还有按位运算符，看起来与后者的记法非常相似：

Code:

```
1// /bitor.cpp2intx=6,y=3;3cout""6|3 = ""<<
(x|y)<< ' \n' ;4cout"""6&"3 = ""<< (x&y)
<< ' \n' ;
```

Output
[basic] bitor:

```
6|3 = 7
6&3 = 2
```

要理解这里发生了什么，请意识到

$$6_{10} \equiv 110_2 \quad \text{and} \quad 3_{10} \equiv 011_2$$

其中下标表示基数。

你可能不会经常使用位运算符，但以下惯用法有时会遇到：

const int STATE_*1* = 1, STATE_*2* = 1<<1, STATE _*3* = 1<<2;int state /* stuff */;if (state & (STATE_*1* | STATE_*3*)) cout <<"We are in state 1 or 3";

整数可以进行的位操作也可以用专门的 *bitset* 类完成。

Code:

```
// /bitset.cpp
bitset<8> xb(6);
bitset<8> yb(3);
auto xory = (xb|yb).to_ulong();
cout << "6|3 = " << (xb|yb) << " = "
     << xory << '\n';
cout << "6&3 = " << (xb&yb) << '\n';
```

Output

```
[basic] bitset:
6|3 = 00000111 = 7
6&3 = 00000010
```

在 C++20 中，还有 *bit* 头文件，它提供了额外的操作，例如旋转一个值的位。

5.2.2 复习

复习 5.1. 是或否?

- 测试 `if (i>0)` 和 `if (0<i)` 是等价的。
- 测试

```
if (i<0 && i>1)
  cout << "foo"
```

  如果 $i < 0$ 并且 $i > 1$，则打印 foo。
- The test

```
if (0<i<1)
  cout << "foo"
```

  如果 i 在零和一之间，则打印 foo。

Review5.2. 对以下内容有任何评论吗?

```
bool x;// ... 带有 x 的代码 ...
if( x == true )// 做一些
事情
```

5.3 Switch 语句

If you have a number of cases corresponding to specific integer values, there is the **switch** statement.

case 语句会按顺序执行，直到你 ‘跳出’ 该 switch 语句：

代码：

```
// /switch.cpp
  switch (n) {
  case 1 :
  case 2 :
    cout << "very small" << '\n';
    break;
  case 3:
    cout << "trinity"<<'\n';
    break;
  默认:
    cout << "large"<< '\n';
  }
```

输出

[基本] switch:

```
for v in 1 2 3 4 5; do
        echo $v | ./switch;     \
    done
非常小
非常小
trinity
large
large
```

练习 5.5. 假设变量 n 是一个非负整数。编写一个 switch 语句，使其具有与以下相同的效果：

```
if (n<5)
  cout << "Small" << endl;
else
  cout << "Not small" << endl;
```

编译器有可能从 switch 语句中生成比条件语句更高效的代码。否则，你无法用 switch 做到而条件语句做不到的事情。

5.4 作用域

条件语句的真值和假值可以由单个语句组成，也可以由花括号内的代码块组成。这样的代码块会创建一个作用域，你可以在其中定义局部变量。

```
if( something ) {int i;....
dosomething with i}// 变量 ‘i’ 已经
消失了。
```

请参见第 8 章了解更多关于作用域的内容。

5.5 高级主题

5.5.1 短路求值

C++ 逻辑运算符有一个称为短路求值的特性：当结果明确时，逻辑运算符会停止严格从左到右的求值。例如，在

```
子句 1 and 子句 2
```

如果第一个子句为真，则不会评估第二个子句假，因为此合取的真值已经确定。

同样地，在

```
子句 1 或 子句 2
```

如果第一个子句是 真，则不会评估第二个子句，因为 或 合取的值已经明确。

这种机制允许您编写

```
if ( x>=0 and sqrt(x)<10 ) { /* ... */ }
```

如果没有短路求值，平方根运算符可以应用于负数。

5.5.2 三元 if

条件语句的真值和假值分支包含完整的语句。例如

```
if (foo)
  x = 5;
else
  y = 6;
```

但如果真值和假值分支分配给同一个变量，但使用不同的表达式呢？你不能这样写

原始代码：

```
if (foo)x
 = 5;elsex
 = 6;
```

Not legal syntax for 'simplification':

```
x = if (foo) 5; else 6;
```

对于这种情况，有 三元 *if*，它本身就像一个表达式，但在两个表达式之间选择。之前的对 x 的赋值就变成了：

```
x = foo ? 5 : 6;
```

令人惊讶的是，这个表达式甚至可以出现在左侧：

```
foo ? x : y = 7;
```

5.5.3 初始化器

C++17 标准引入了一种新的 `if` 和 `switch` 语句形式：它允许在测试之前有一个单一的声明语句。这被称为初始化器。

Code:

```
// /ifinit.cpp
if ( char c = getchar(); c!='a' )
  cout << "Not an a, but: " << c
       << '\n';
else
  cout << "That was an a!"
       << '\n';
```

Output
[basic] ifinit:

```
for c in d b a z ; do \
   echo $c | ./ifinit ; \
done
Not an a, but: d
Not an a, but: b
That was an a!
Not an a, but: z
```

如果初始化语句是一个声明，那么这一点尤其优雅，因为声明的变量然后是条件局部变量。以前人们不得不写

```
char c;
c = getchar();
if ( c!='a' ) /* ... */
```

变量在条件作用域之外定义。

您可以在 `else if` 分支中有进一步的初始化器。在那里定义的变量的作用域跨越任何文本上后续的分支。

5.6 复习问题

复习 5.3. 判断题：以下是一个合法的程序：

```
#include <iostrea
m>>
<int>{v7}<
main>{v9}() {if
(true)inti  = 1;
elseinti  = 2;
std::cout <<
i;return 0;}
```

复习 5.4. 判断题：以下等价：

```
if (cond1)
  i = 1
else if (cond2)
  i = 2
else
  i = 3;
```

比较：

```
if (cond1)
  i = 1
else{
  if (cond2)
    i = 2;
  else
    i = 3;
}
```

5. 条件语句

第 6 章

循环

有许多情况下，您希望重复一个操作或一系列操作：

- 一个依赖于时间的数值模拟执行固定数量的步骤，或直到某个停止测试。
- 递归：

$$x_{i+1} = f(x_i)$$

- 检查或更改数据库表中的每个元素。

C++ 用于此类重复的结构称为 < 样式 id='2'> 循环 </ 样式 >：一组被重复的语句。C++ 中的循环语句有两种：

- 带< 样式 id='2'>for 循环 </ 样式 >，它通常与预定的重复次数相关联，并且重复的语句基于某种计数器；以及
- 带< 样式 id='2'>while 循环 </ 样式 >，其中语句会无限重复，直到满足某个条件。

然而，两者之间的区别并不明确：在许多情况下，你可以使用任何一种。

< 样式 id='1'> 我们将首先考虑 </ 样式 >< 样式 id='3'>for</ 样式 >< 样式 id='5'> 循环；</ 样式 >< 样式 id='7'>while</ 样式 >< 样式 id='9'> 循环 </ 样式 > 在 </ 样式 id='11'>6.3</ 样式 >< 样式 id='13'> 节中。</ 样式 >

6.1 ‘for’ 循环

在最常见的场景中，一个 for 循环有一个 循环计数器，范围从某个初始值到某个最终值。一个展示这种简单情况语法的例子是：

```
int sum_of_squares{0}; for(int var=low; var<
upper; var++) { sum_of_squares+= var*var; } cout
<< "The sum of squares from "<< low <<" to " <<
upper<< " is "<< sum_of_squares << endl;
```

The `for` 行被称为 循环头，而大括号之间的语句被称为 循环体。循环体的每次执行被称为一个迭代。

练习 6.1. 使用 `cin` 读取一个整数值，并打印多次 ‘Hello world’ 。

Exercise 6.2. Extend exercise 6.1: the input 17 should now give lines

Hello world 1

Hello world 2 ....

Hello world 17

Can you do this both with the loop index starting at 0 and 1?
Also, let the numbers count down.

我们将现在研究循环的组件。

6.1.1 循环变量

首先，大多数 `for` 循环有一个循环变量或循环索引。括号中的第一个表达式通常用于这个变量的初始化：它在循环迭代之前执行一次。如果你在这里声明一个变量，它就变成了循环的局部变量，也就是说，它只存在于循环头部的表达式中，并在循环迭代期间存在。

The loop variable is usually an integer:

```
for (int index=0; index<max_index{v18}; index=index+1) { ... }
```

但其他类型也是允许的：

```
for ( float x=0.0;x<10.0; x+=delta ){...
}
```

注意非整型变量的停止测试！

通常，循环变量只在循环内部有意义，因此它应该只在循环内部定义。您可以通过在循环头部定义它来实现这一点：

```
for (int var=low; var<upper; var++) {
```

然而，它也可以在循环外部定义：

```
int var; for (var=low; var<upper; var++) {
```

但是，只有当变量在循环之后确实需要时，你才应该这样做。你将在下面的 6.3 节中看到一个示例，其中这样做是有意义的。6.3.

6.1.2 停止测试

接下来有一个测试，需要评估为布尔表达式。这个测试通常被称为 “ 停止测试 ”，但严格来说，它实际上是在每次迭代的开始时执行，包括第一次，它实际上是一个 “ 当这个为真时循环 ” 测试。

Code:

```
1 // /pretest.cpp
2   cout << "before the loop" << '\n';
3   for (int i=5; i<4; ++i)
4     cout << "in iteration "
5          << i << '\n';
6   cout << "after the loop" << '\n';
```

Output
[basic] pretest:

```
before the loop
after the loop
```

- If this boolean expression is true, do the next iteration.
- Done before the first iteration too!
- Test can be empty. This means no test is applied.

```
for ( int i=0; i<N; i++) {...}
for ( int i=0; ; i++ ) {...}
```

通常，初始化和停止测试的组合决定了执行多少次迭代。如果你想要执行 N 次迭代，你可以写

```
for (int iter=0; iter<N; iter++)
```

or

```
for (int iter=1; iter<=N; iter++)
```

前者对 C++ 稍微更符合习惯，但你应该写最适合你正在编写的问题的代码。

停止测试不需要是上限。这里是一个向下计数到下限的循环的例子。

for(int var=high; var>=low; var--) { ... }

可以省略停止测试

```
for(int 变量=low; ; 变量++) { ... }
```

如果循环以其他方式结束。你稍后会看到这一点。

6.1.3 自增

最后，在每次迭代后，我们需要更新循环变量。由于这通常是将变量的值加一，我们可以非正式地将其称为 “ 自增 ”，但它可以是一个更通用的更新。

每次迭代后执行自增。最常见的:

- `i++` 用于正向计数的循环;
- `i--` 用于反向计数的循环;

其他:

- `i+=2` 用于仅覆盖奇数或偶数，具体取决于您开始的起点;
- `i*=10`用于仅覆盖 10 的幂。

可选:

```
for (int i=0; i<N; ) {
  // stuff
  if ( something ) i+=1; else i+=2;
}
```

这就是循环的执行方式。

- 执行初始化。
- 在每次迭代的开始，包括第一次迭代，都会执行停止测试。如果测试结果为真，则以循环变量（s）的当前值执行迭代。
- 在每次迭代的结束时，执行增量操作。*C 差异：* 在循环头部声明循环变量也是 C 语言的现代新增特性。使用编译器标志 -std=c99。

练习 6.3. 取此代码：

```
// /sumsquares.cpp int sum_of_ 平方 {0}; for (int var=
低 ; var< 高 ; ++var) { sum_of_ 平方 +=var*var; } cout
<<"从 "<< 低 <<  " 到 " << 高 <<" 的平方和是 " <<sum_of_
平方 <<'\n';
```

并将其修改为仅对每个其他数字的平方求和，从低开始。

你能找到一种方法来对偶数 ≥ 低的平方求和吗？

复习 6.1. 对于以下每个循环头部，循环体执行多少次？（你可以假设循环体不会改变循环变量。）

```
for(int i=0; i<7; i++)for(int i
=0; i<=7; i++)for(int i=0; i<0;
i++)
```

Review6.2. 最后执行的是哪一次迭代？

```
for (int i=1; i<=2; i=i+2)

for (int i=1; i<=5; i*=2)

for (int i=0; i<0; i--)

for (int i=5; i>=0; i--)
```

```
        for (int i=5; i>0; i--)
```

6.1.4 循环体

循环体可以是一个语句：

```
int s{0};for ( int i=0; i<N; i++ )
s+= i;
```

或者是一个代码块：

```
int s{0};for ( int i=0; i<N; i++ )
{int t =i*i;t;s;t;}
```

如果是块，它就是你可以声明局部变量的作用域。

6.2 嵌套循环

通常情况下，循环体中会包含另一个循环。例如，你可能想要遍历矩阵中的所有元素。这两个循环将拥有它们自己的唯一循环变量。

```
for (int row=0; row<m; row++)
  for (int col=0; col<n; col++)
    ...
```

这被称为 循环嵌套；`row` 循环称为 外循环，而 `col` 循环称为 内循环。 .

行、列 遍历索引空间（无论它是否对应于数组对象）称为 词典序。

Exercise 6.4. Write an i, j loop nest that prints out all pairs with

$$1 \leq i, j \leq 10, \quad j \leq i.$$

为每个 `i` 值输出一行。

现在编写一个 i, j 循环，打印所有配对，

$$1 \leq i, j \leq 10, \quad |i - j| < 2,$$

再次为每个 `i` 值打印一行。思考题：这个练习在内部循环中使用条件语句肯定更容易，但你能不能不使用条件语句完成?

你只需要遍历一个矩形范围的 i, j 索引，并不意味着你必须编写一个字典序索引的循环。图 6.1 说明了你可以按行 / 列或对角线查看 i, j 索引。就像行和列被定义为 $i =$ 常量 和 $j =$ 常量一样，对角线由 $i + j =$ 常量定义。

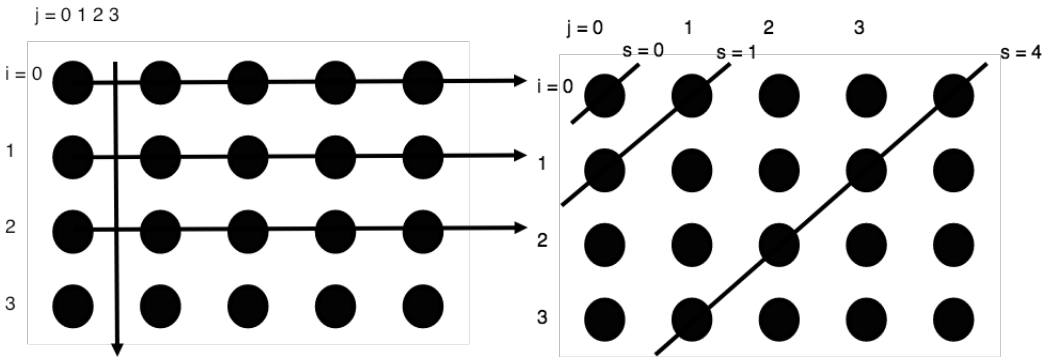


图 6.1：索引集的字典序和对角线排序

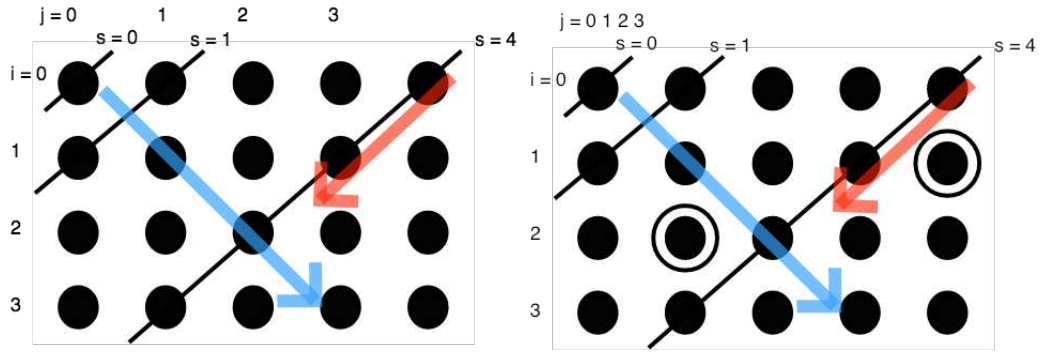


图 6.2：练习第二部分的插图 6.6

.

6.3 循环直到

基本的 for 循环看起来相当确定：循环变量通过一个或多或少规定的值集进行范围。这对于遍历数组的元素是合适的，但如果你正在编写一些需要运行直到某个动态确定的条件得到满足的过程，则不适用。在本节中，你将看到一些编写此类情况的方法。

首先，'for' 循环中的停止测试是可选的，因此你可以编写一个无限循环，如下所示：

```
for (int var=low; ; var=var+1) { ... }
```

如何结束这样的循环？为此，您使用 **break** 语句。如果执行遇到此语句，它将接着执行循环后的第一个语句。

```
for(int var=low; ; var=var+1)
{// 语句 ;if(some_test) break;// 更
多语句 ;}
```

对于以下练习，参见 6.2 以获取灵感。

练习 6.5. 编写一个双重循环遍历 $0 \leq i, j < 10$，打印所有满足乘积 $i \cdot j > 40$ 的数对 (i, j)。

你可以参考代码仓库中的文件 *ijloop.cpp*。

练习 6.6. 编写一个双重循环遍历 $0 \leq i, j < 10$，打印第一个乘积满足 $i \cdot j > N$ 的数对，其中 N 是你读入的一个数。一个良好的测试用例是 $N = 40$。
其次，找到一个满足 $i \cdot j > N$ 的数对，但 $i + j$ 的值最小。（如果有多个数对，报告 i 值较小的那个。）你能遍历 i, j 索引，使得它们首先枚举所有数对 $i + j = 1$，然后 $i + j = 2$，然后 $i + j = 3$ 等等吗？提示：编写一个遍历和值 $1, 2, 3, \ldots$ 的循环，然后找到 i, j。

你的程序应该打印出这两对，每对占一行，数字之间用逗号分隔，例如 8,5。

练习 6.7. 循环头的三个部分都是可选的。那么

```
for (;;) { /* some code */ }
```

？

假设你想知道当 break 发生时循环变量是什么。你需要循环变量是全局的：

```
int var; ... 设置 var 的代码 ...for( ; var< 上限 ;
var++) {... 语句 ...if(某个条件 ) break... 更多语句 ...}...
使用 var 的跳出值的代码 ...
```

在其他情况下：在头部定义循环变量！

示例：

代码：

```
1 // /findmin.cpp
2     float minpos{0.f};
3     for ( ; ; minpos+=.5f ) {
4       if (f(minpos)>90)
5         break;
6     }
7     cout << "Minimum satisfying value: "
8          << minpos << '\n';
```

输出
[loop] findmin:

最小满足值： 9.5 最小满足值： 9.5

练习 6.8. 你能让这个循环更紧凑吗？

Ins而不是使用一个 `break` 语句，可以有其他方法 of ending the loop.

If the test comes at the start or end of an iteration, you can move it to the loop header:

```
bool need_to_stop{false};
for (int var=low; !need_to_stop ; var++) {
  ... some code ...
  if ( some condition )
    need_to_stop = true;
}
```

在循环中改变控制流的另一种机制是 `continue` 语句。如果遇到这个语句，执行将跳到下一次迭代的开始。

```
for (int var=low; var<N; var++) {
  statement;
  if (some_test) {
    statement;
    statement;
  }
}
```

Alternative:

```
for (int var=low; var<N; var++) {
  statement;
  if (!some_test) continue;
  statement;
  statement;
}
```

The only difference is in layout.

6.3.1 While loops

‘循环直到’ 的另一种可能性是一个 `while` 循环，它会在条件满足时重复。while 循环没有计数器或更新语句；如果你需要这些，你必须自己创建它们。

Syntax:

```
while ( condition ) {
  statements;
}
```

or

```
do {
  statements;
} while ( condition );
```

两个 while 循环变体可以描述为 ‘预测试’ 和 ‘后测试’。它们之间的选择完全取决于上下文。

```
float money = 继承();while (
money < 1.e+6 )money +=on_ 年
_ 储蓄();
```

让我们考虑一个例子：我们从输入中读取数字，直到其中一个为正数。以下两个代码示例分别使用了 `do .. while` 和 `while ... do` 惯用法。

第一个解决方案可以称为 ‘预测试’：

代码：

```
1// /whiledo.cpp 2 cout <<" 输入一个正数 : " ; 3
cin >>invar; cout << ’ \n’ ; 4 cout <<" 你输入的是 : "
<< invar << ’ \n’ ; 5 while (invar<=0) { 6 cout << "
输入一个正数 : " ; 7 cin >> invar; cout <<’ \n’ ; 8
cout << " 你输入的是 :"<< invar << ’ \n’ ; 9 } 10
cout<< " 你的正数是 :"11 <<invar <<’ \n’ ;
```

输出

[基本] whiledo:

```
输入一个正数 : 您输入了 : -3 输
入一个正数 : 您输入了 : 0 输入一
个正数 : 您输入了 : 2 您的正数是
2
```

问题：代码重复。

第二个使用 ‘后测试’，您可以看到这里解决了代码重复的问题；

代码：

```
1// /dowhile.cpp 2 int invar; 3 do { 4 cout << "
Enter a positive number: " ; 5 cin >> invar; cout <<’
\n’ ; 6 cout << "You said: "<< invar << ’ \n’ ; 7 }
while (invar<=0); 8 cout <<"Your positive nu
mber was: "9 <<invar <<’ \n’ ;
```

输出

[基本] do while:

```
输入一个正数：你输入了 : -3
输入一个正数：你输入了 : 0 输
入一个正数：你输入了 : 2 你的
正数是：
                                2
```

The post-test syntax leads to more elegant code.

练习 6.9。 此时，你已准备好做素数项目中的练习，第 45.3 节。

练习 6.10. 一匹马被一根 1 米长的弹性绳索拴在一根柱子上。一只蜘蛛原本坐在柱子上，开始沿着绳索走向马，速度为 1 厘米 / 秒。突然，马受惊而以 1 米 / 秒的速度跑开。假设弹性绳索可以无限延伸，蜘蛛会到达马吗？

练习 6.11。 一个银行账户有 100 美元，年利率为 5%。另一个账户有 200 美元，但年利率仅为 2%。在两种情况下，利息都会存入账户。

多少年后，第一个账户中的金额会超过第二个账户？使用 `while` 循环来解决这个问题。

思考题：通过预测试和后测试，以及使用 for 循环来比较解决方案。

6.4 高级主题

6.4.1 并行性

在本章的开头，我们提到了以下循环的例子：

- 一个依赖于时间的数值模拟执行固定数量的步骤，或直到某个停止测试。
- 递归：

$$x_{i+1} = f(x_i).$$

- 检查或更改数据库表中的每个元素。

前两种情况实际上需要按顺序执行，而最后一种则更类似于数学中的 ‘forall’ 量词。你将在后续的数组上下文中学习到两种不同的语法。当你在学习 并行编程 时，也可以利用这种差异。Fortran 有一个 *do concurrent* 循环结构用于此目的。

6.5 练习

练习 6.12。 找出所有小于 100 的整数三元组 u,v,w，使得 $u^2 + v^2 = w^2$。确保你省略你已经找到的解的重复项。

Exercise 6.13. The integer sequence

$$u_{n+1} = \begin{cases} u_n/2 & \text{if } u_n \text{ is even} \\ 3u_n + 1 & \text{if } u_n \text{ is odd} \end{cases}$$

导致 哥德巴赫猜想：无论起始猜测 u_1，序列 $n \to u_n$ 总会终止于 1。

$$5 \to 16 \to 8 \to 4 \to 2 \to 1$$

$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \cdots$$

(如果你在达到 1 后继续迭代会发生什么？)

尝试所有初始值 $u_1 = 1, \ldots, 1000$ 以找到导致最长序列的值：每次你找到一个比之前最大值更长的序列时，打印出起始数字。

练习 6.14. 大整数通常在每三位数字之间用逗号（美国用法）或句号（欧洲用法）分隔。编写一个程序，接受从输入中获取的整数，例如 2542981，并将其打印为 2,542,981。

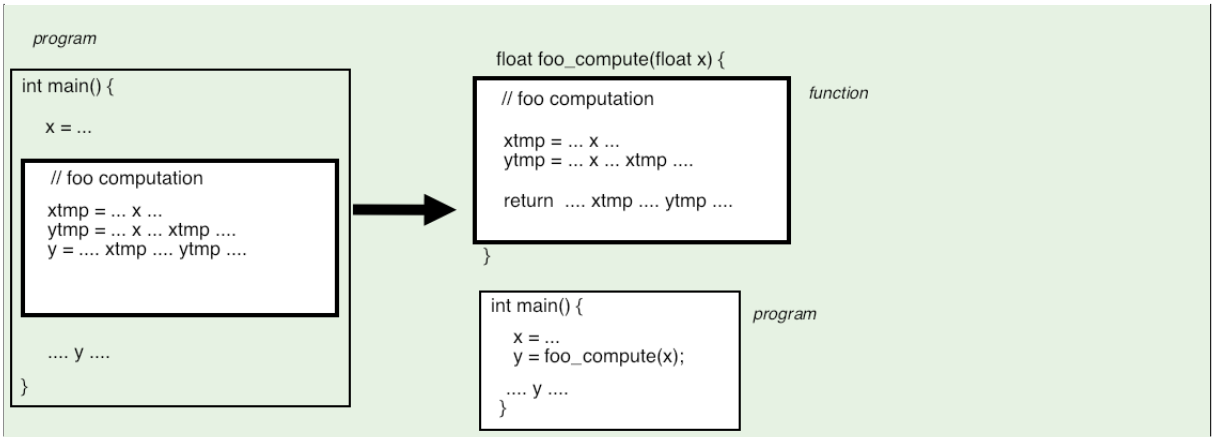
6. 循环

第 7 章

函数

一个 函数 (或 子程序) 是一种将代码块简化和用单行替换它的方法。这首先是一个代码结构工具：通过给函数一个相关的名称，你将应用程序的术语引入到你的程序中。

- 找到一个具有明确功能的代码块。
- 将其转换为函数：函数定义将包含该代码块，并有一个命名它的头部。
- 通过其名称调用函数。



通过引入函数名，你已经引入了 抽象：你的程序现在使用与你的问题相关的术语，而不仅仅是基本控制结构，例如 `for`。通过对象（第 9) 章节，你将学习进一步的抽象，这样你的程序将使用应用程序术语，例如 `Point` 或 `Line`。

7.1 Function definition and call

函数有两个方面：

- 函数定义 只需执行一次，通常在主程序之上；
- 对任何函数的 函数调用 可以多次发生，在主程序内部或其他函数内部。

让我们考虑一个简单的示例程序，其中我们介绍函数。我们编写二分法 算法来查找函数的根；参见 47.1 部分以获取详细信息。

示例：通过二分法查找零。

$$\underset{x}{?}: f(x) = 0, \qquad f(x) = x^3 - x^2 - 1$$

(其中问号量化符表示 “ 对于 x’).

第一次尝试编写这个：主程序中的所有内容。

代码：

```
1// /bisect1.cpp2
    float left{0.},right{2.},
3     mid;
4   while (right-left>.1) {
5     mid = (left+right)/2.;
6     float fmid =
7       mid*mid*mid - mid*mid-1;
8     if (fmid<0)
9       左 = 中 ;
10    else
11      right =mid;
12  }
13  cout << "Zero happens at: " << mid <<
      '\n';
```

Output [func] 二分 1:

```
零发生在： 1.4375
```

我们将分两步进行模块化。我们首先介绍的目标函数是 $f(x)$。

Introduce a function for the expression `m*m*m - m*m-1`:

```
// /bisect2.cppfloat f(float
x) {return x*x*x - x*x -1; }
```

在 main 中使用：

```
// /bisect2.cpp
  while (right-left>.1) {
    mid = (left+right)/2.;
    float fmid = f(mid);
    if (fmid<0)
      left = mid;
    else
      right = mid;
  }
```

接下来，我们介绍一个用于求零算法的函数。

函数：

```
// /bisect3.cpp
float f(float x) {
  returnx*x*x- x*x*x-1;
};
float在_     _之间查找零
    (float l,float r) {
  float mid;
  while (r-l>.1) {
    mid =(l+r)/2.;
    float fmid= f(mid);
    if (fmid<0)
      l =mid;
    else
      r = mid;
  }
  返回中间;
};
```

新主：

```
// /bisect3.cpp
int main() {
  float left{0.},right{2.};
  float zero =
    find_零_之间(左,右);
  cout <<"零发生在："
       <<零<< '\n';

  返回 0;
}
```

主程序现在不再包含任何实现细节，例如局部变量或使用的方法。这使得主程序更短更优雅：我们将中点的变量移到了函数内部。这些是实现细节，不应在主程序中。

在这个例子中，函数定义包括：

- 关键字 `float` 表示该函数向调用代码返回一个 `float` 类型的结果。
- 函数名 `find_zero_between` 由您指定。
- 括号内的部分 `(float l,float r)` 称为 ‘参数列表’：它表示该函数接受两个浮点数作为输入。对于函数而言，这些浮点数的名称将是 `l,r`，而不管主程序中使用了什么名称。
- 函数的 ‘主体’，即将要执行的代码，被花括号包围。
- 一个 ‘返回’ 语句，用于将计算结果从函数中转移出来。

7.1.1 函数的正式定义

形式上，函数定义由以下内容组成：

- 函数结果类型：你需要指明结果类型；
- 名称：你可以自行定义；
- 零个或多个 函数参数。这些描述了你需要向函数提供多少 函数参数 作为输入。参数由类型和名称组成。这使得它们看起来像变量声明，并且这就是它们的作用。参数之间用逗号分隔。然后是：
- 函数体：构成函数的语句。函数体是一个 作用域：它可以有局部变量。（你不能嵌套函数定义。）
- 一个 返回语句。不过，它不必是最后一个语句。

然后，该函数可以在主程序中或另一个函数中使用。

函数体定义了一个 作用域：函数计算中的局部变量对调用程序不可见。

函数不能嵌套：你不能在另一个函数的体内定义一个函数。

7.1.2 函数调用

The 函数调用 包括

- 函数的名称，和
- 在括号之间，任何输入参数。

函数调用可以独立存在，也可以出现在赋值语句的右侧。

练习 7.1。 通过引入函数 *new_l*，*new_r* 使二分算法更优雅，这些函数用作：

```
l = new_l(l,mid,fmid);
r = new_r(r,mid,fmid);
```

你可以参考代码仓库中的文件 *bisect.cpp* 来完成这个任务。

问题：你可以从函数中省略 *fmid*。编写这个变体。为什么这不是一个好主意?

函数调用

1. 将函数参数的值复制到 函数参数； 2. 执行函数体，3. 函数调用被替换为你 返回 的值。4. （如果函数不返回任何值，例如因为它只打印输出，你将返回类型声明为 **void**。）

介绍两个正式概念：

- 函数定义可以有零个或多个 参数，或 形式参数。这些参数作为函数内部的变量定义。
- 函数调用有相应数量的 实参，或 实际参数。

7.1.3 为什么使用函数?

在许多情况下，使用函数编写的代码也可以不使用函数编写。那么，为什么要使用函数呢？有几种原因。

可以将函数的动机描述为使代码更具结构性和可读性。使用函数调用的源代码会变得更短，而函数名使代码更具描述性。这有时被称为 “ 自文档化代码 ”。

有时引入函数的动机可以来自 代码复用：如果您的源代码中有两个地方出现相同的代码块（这被称为 代码重复），您可以通过一个函数定义来替换它，以及两个（单行）函数调用。

假设您要执行两次相同的计算：

```
doublex,y, v,w;y = ..... 从 x 的计算 .....w = .....
相同的计算，但来自 v .....
```

使用函数，这可以替换为：

```
double computation(double in) {
  return .... computation from 'in' ....
}

y = 计算(x); w = 计
算(v)
```

示例：多次范数计算：

重复代码：

```
float s = 0;
for(int i=0; i<x.size(); i++)
  {s}+= {abs}(x[i]);
cout << "One norm x: " << s << endl;
s = 0;
for (int i=0; i<y.size(); i++)
    《C++17/Fortran2008
c科学编程艺术，第3卷》引言
  7.1.3 为什么使用函数?
                          y: " << s << endl;
```

becomes:

```
float OneNorm(vecto   r<float> a ) {
  float sum = 0;
  for (int i=0; i<a.size(); i++)
    sum += abs(a{i};)
  返回求和;
   }intmain()
{//  tff
  ...     s u
  cout <<"One norm x: "
       << OneNorm(x) << endl;
  cout <<"One norm y: "
       << OneNorm(y) << endl;
```

（本例中无需担心数组内容）

A 使用函数的最终目的是代码可维护性：

- 更容易调试：如果你两次使用相同的（或大致相同的）代码块，并且你发现了一个错误，你需要修复两次。
- 维护：如果一个代码块出现两次，并且你在其中一次更改了某些内容，你必须记住也要更改其他出现的位置。
- 本地化：现在，任何仅在函数中用于计算的变量都具有有限作用域。

```
void print_mod(int n,int d) {
  int m = n%d;
  cout << "The modulus of " << n << " and " << d
       << " is " << m << endl;
```

Review7.1. 真或假?

- 函数的目的是使你的代码更短。
- 使用函数可以使您的代码更易于阅读和理解。
- 在您可以使用它们之前，必须先定义函数。
- 函数定义可以放在主程序内部或外部。

7.2 函数定义和调用的结构

大致来说，函数接收输入并计算某个结果，然后将该结果返回。以下是一些简单的例子：

```
int compute( float x, char c ) {
    /* code */
    return somevalue;
};
// in main:
i = compute(x,'c');
```

```
void compute( float x, char c ) {
    /* code */
};
// in main:
compute(x,'c');
```

因此我们需要讨论函数定义及其使用。

7.3 定义与声明

C++ 编译器 在 单次遍历 中翻译你的代码：它从上到下遍历你的代码。这意味着你不能引用任何尚未定义的内容，例如函数名。因此，在之前的示例中，我们将函数定义放在了主程序之前。

还有一个解决方案。为了让编译器判断函数调用是否合法，它不需要完整的函数定义：一旦知道函数的名称、输入和结果的类型，它就可以继续进行。这些信息有时被称为 函数头、函数原型 或 函数签名，但技术术语是 函数声明。

在以下示例中，我们将函数声明放在主程序之前，并将完整的函数定义放在它之后：

有些人喜欢以下定义函数的风格：

```
// declaration before main
int my_computation(int);

int main() {
  int result;
  result = my_computation(5);
  return 0;
};

// definition after main
int my_computation(int i) {
   return i+3;
}
```

这纯粹是一个风格问题。

参见第 19 章以获取更多详细信息。

7.4 空函数

有些函数不返回结果值，例如因为只将输出写入屏幕或文件。在这种情况下，您将函数定义为 `void`类型。

```
voidprint_header() {cout <<"
**********" << endl;cout <<"* 输出 *" <<
endl;cout <<"**********" <<
endl;}intmain() {print_header();cout <<"第
25 天的结果 :" << endl;// 打印结果的代码 ....return
0;}
```

```
void print_result(int day,float value) { cout <<"**********"
<< endl; cout <<"* 输出 *" << endl; cout <<"**********" <<
endl; cout <<" 第 " << day << ":" << endl; cout <<" " << value <<
endl; } intmain() { print_result(25,3.456); return 0; }
```

Review7.2. 是或否?

- 一个函数只能有一个输入
- 一个函数只能有一个返回结果
- 一个 void 函数不能有返回 语句。

7.5 参数传递

C++ 函数类似于数学函数：你已经看到函数可以有输入和输出。事实上，它们可以有多个输入，以逗号分隔，但只有一个输出。

$$a = f(x, y, i, j)$$

我们首先研究看起来像这些数学函数的函数。它们涉及一个称为参数传递 的机制，称为按值传递。稍后我们将查看按引用传递。

7.5.1 按值传递

以下编程风格很大程度上受到数学函数的启发，并称为函数式编程 [1]。

1. 函数式编程还有更多内容。例如，严格来说，你的整个程序需要基于函数调用；除了函数定义和调用之外，没有其他代码。

- 一个函数有一个结果，通过返回语句返回。函数调用看起来像

  y =f<x1>,x2,x3>;

- 示例：

代码：

```
1 // /passvalue.cpp
2 double 平方(          uble x ) {
3 dodouble y = x*x;
4   返回 y;
5 }
6     /* ... */
7   number = 5.1;
8   cout << "Input starts as: "
9        <<数字<<'\n';
10  其他 = 平方（数字）;
11  cout <<"输出变量是："
12       <<其他<< '\n';
13  cout << "在 put ar is n     ow: "
14       <<数字<<'\n';
```

Output[func] passvalue:

```
                    : 5.1
输出 var 是： 26.01
var 现在是： 5.1
```

- C++ 参数传递机制的定义说明输入参数被复制到函数中，这意味着它们在调用程序中不会改变：

代码：

```
1// /passvaluelocal.cpp
2doublesquared(doublex)3        {
4   返回 x*x;
5 }
6     /* ... */
7   《C++17/Fortran2008 科学编程艺术》
    第 3 卷 简介 7.5.1 值传递
8                  put starts as: "
9          <<数字<<'\n';r 平方（数字）;
10  othe  =
11  cout <<"Output var is: "
12       <<  其他<< '\n';
13  cout << "Input var is now: "
14       << number << '\n';
```

Output [函数] 按值传递局部

```
输入以： 5.1 开始
输出变量是： 26.01
输入变量现在是： 5.1
```

我们说输入参数是按值传递：它的值被复制到函数中。在这个例子中，函数参数 `x` 在函数中充当局部变量，并用主程序中 `number` 的副本进行初始化。

练习 7.2. 编写两个函数

int biggest(int i, int j); int smallest(int i, int j);

以及一个打印结果的程序：

int i = 5, j = 17;cout ...
biggest(i, j) ...cout ... s
mallest(i, j) ...

将变量传递给例程传递的是值；在例程中，变量是局部的。因此，在这个例子中，参数的值不会改变：

Code:

```
1 // /localparm.cpp
2 void change_scalar(int i) {
3   i += 1;
4 }
5     /* ... */
6   number = 3;
7   cout << "Number is 3: "
8        << number << '\n';
9   change_scalar(number);
10  cout << "is it still 3? Let's see: "
11       << number << '\n';
```

Output
[func] localparm:

```
Number is 3: 3
is it still 3? Let's see: 3
```

练习 7.3. 如果你正在进行素数项目（章节 45 ）你现在可以完成练习 45.6。

练习 7.4. 如果你正在进行零查找项目（章节 47 ）你现在可以完成练习 47.9。

7.5.2 引用传递

函数只有一个输出是一个限制。因此存在一种机制，通过改变输入参数并返回（可能多个）结果来绕过这种限制。你这样做的方法是，不将值复制到函数参数中，而是将函数参数转换为函数调用位置处变量的别名。

我们需要一个 <code> 引用 </code> 的概念：另一个变量，用来引用另一个已经存在的变量所指向的 ‘ 东西 ’ 。

引用在其定义中用与号表示，并作为它所引用的 ‘ 东西 ’ 的别名。

代码:

```
1 // /ref.cpp
2 <code>int </code><code>i</code><code>;</code>
3 int &ri= i; i 5
4     =   ;
5 cout << i << "," << ri << '\n';
6 i *= 2;
7 cout << i << "," << ri << '\n';
8 ri -= 3;
9 cout << i << "," << ri << '\n';
```

输出
[基本] 引用

```
5,5
10,10
7,7
```

(您通常不会以这种方式使用引用。)

正确:

```
float x{1.5};
float &xref = x;
```

Not correct:

```
float x{1.5};float&xref; // 错误: 需要立即初始化 xref= x;


float &threeref = 3; // 错误: 仅引用 'lvalue'
```

你可以将函数参数作为主程序中变量的引用。这使得函数参数成为另一个指向同一事物的名称。

函数参数 `n` 成为主程序中变量 `i` 的引用:

```
1 void f(int&n) {2 n= /* some expression */ ;3 };4
intmain() {5 inti;6 f(i);7 // inow has the value
that was set in the function8 }
```

引用语法比 C 的 '按引用传递' 更简洁

使用与号，参数按引用传递：不是复制值，函数接收一个引用，因此参数成为调用环境中某物的引用。

Remark3 *C*的 按引用传递机制 不同，在 *C++* 中不应使用。实际上它不是真正的按引用传递，而是按值传递地址。如果你确实需要变量的地址，请使用 `addressof` 从 `memory` 头文件，因为与号运算符可以重载。

Remark 4 *We sometimes use the following terminology for function pa rameters:*

- 输入 参数：按值传递，因此它仅作为函数的输入，不通过此参数输出结果；
- 输出 参数：按引用传递，以便它们向程序返回一个 " 输出 " 值。
- 通过 参数：这些参数按引用传递，当函数被调用时它们有初始值。在 *C++* 中，与 *Fortran* 不同，没有为这些参数的真正独立语法。

代码：

```
1// /setbyref.cpp
2 void f( int &i ) {
3   i = 5
4 }
5 int main() {
6
7   int var = 0;
8   f(var);
9   cout << var << '\n';
```

输出

[基本] 按引用设置：

```
5
```

比较省略引用时的差异。

例如，考虑一个尝试从文件中读取值的函数。对于任何与文件相关的内容，你总是要担心文件不存在的情况。因此，我们的函数返回：

- 一个布尔值，用于指示读取是否成功，和
- 如果读取成功，则返回实际值。

以下是一个常见用法，通过 返回 语句返回成功值，并通过参数传递值。

```
bool can_read_value( int &value ) {
  // this uses functions defined elsewhere
  int file_status = try_open_file();
  if (file_status==0)
    value = read_value_from_file();
  return file_status==0;
}

int main() {
  int n;
  if (!can_read_value(n)) {
    // if you can't read the value, set a default
    n = 10;
  }
  ..... do something with 'n' ....
```

这个例子也可以解决，也许更符合习惯用法，使用 `std::optional`; 章节 24.6.2。

练习 7.5. 编写一个 `void` 函数 *swap*，用于交换两个参数的输入值：

代码：

```
1// /swap.cpp
2   int i=1,j=2;t i<<
3   cou          "," << j << '\n';
4   swap(i,j);
5   cout << i << "," << j << '\n';
```

Ou tp t[func] swap:

```
1,2
2,1
```

练习 7.6. 编写一个可除性函数，该函数接受一个数字和一个除数，并给出：

- a `bool`返回结果以指示数字是否可被整除，并
- a 作为输出参数的余数。

代码：

```
1// /divisible.cpp2
    cout <<number;
3   if
       ( 是_divisible(number, divisor, remainder)
4      cout<<  " is divisible by ";
5   else
6     cout  <<  " has remainder "
7           <<  余数 <<" 来自              ";
8   cout <<  divisor <<'\n';
```

输出
[func] 可整除：

```
8 has remainder 2 from 3
8 可被 4 整除
```

练习 7.7. 如果你在做几何项目，你应该现在做第 46.1 节中的练习。

7.6 递归函数

在数学中，序列通常递归定义。例如，阶乘序列 $n \to f_n \equiv n!$ 可以定义为

$$f_0 = 1, \qquad \forall_{n>0}: f_n = n \times f_{n-1}.$$

Instead of using a subscript, we write an argument in parentheses

$$F(n) = \begin{cases} n \times F(n-1) & \text{if } n > 0 \\ 1 & \text{otherwise} \end{cases}$$

这是一种可以转换为 C++ 函数的形式。阶乘函数的头部可以像这样：

int factorial(int n)

那么函数体是什么样子呢？我们需要一个 `return` 语句，而我们返回的内容应该是 $n \times F(n-1)$：

```
int factorial(int n) {return
n*factorial(n-1);} // 几乎正确，但还
不是完全正确
```

那么如果你写了

```
int f3; f3 = factorial(3);
```

嗯，

- 表达式 `factorial(3)` 调用了 `factorial` 函数，将 `3` 作为参数 `n` 的值。
- 返回语句返回 `n*` 阶乘 `(n-1)`，在这种情况下 `3*` 阶乘 `(2)`。
- 但阶乘 `(2)` 是什么？计算这个表达式意味着阶乘 函数会被再次调用，但现在 `n` 等于 2。

- 计算 `阶乘(2)` 返回 `2*阶乘(1)`,...
- ... 返回 `1*阶乘(0)`,...
- ... 返回 ...
- 哎呀。我们忘记包含 n 为零的情况。让我们修复它:

```
int 阶乘(int n) {if (n==0)
return 1;elsereturn n*阶乘
(n-1);}
```

- 现在 `阶乘(0)` 是 1，所以 `阶乘(1)` 是 `1*阶乘(0)`，也就是 1,...
- ... 所以 `阶乘(2)` 是 2，并且 `阶乘(3)` 是 6。

Exercise 7.8. It is possible to define multiplication as repeated addition:

代码:

```
1// /mult.cpp
2int times(int number,int mult) {
3    cout <<"(" << mult << )";
4    if (mult==1)
5       返回数字 ;
6    else t
7       re urn number + times(number,mult-1);
8 }
```

Output
[func] mult:

Enter number and multiplier
递归乘法
的 7 和 5: (5)(4)(3)(2)(1)35

将这个想法扩展到定义幂为重复的乘法。

你可以基于代码仓库中的文件 *mult.cpp* 来实现这个功能。

练习 7.9。 The 埃及乘法 算法几乎有 4000 年的历史。乘以 $x \times n$ 的结果是:

如果 n 是偶数:

twice the multiplication $x \times (n/2)$;
otherwise if $n == 1$:

x

否则:
x 加上乘法 $x \times (n-1)$

扩展练习的代码 7.8 以实现这一点。

思考题：讨论此算法的计算方面与重复加法的传统算法。

Exercise 7.10. The sum of squares:

$$S_n = \sum_{n=1}^{N} n^2$$

可以递归定义为

$$S_1 = 1, \qquad S_n = n^2 + S_{n-1}.$$

编写一个递归函数来实现这个第二种定义。在交互式输入的数字上测试它。

Then write a program that prints the first 100 sums of squares.

你需要累加多少个平方才会溢出？你能在不运行程序的情况下估计这个数字吗?

练习 7.11。 编写一个递归函数来计算斐波那契数 :

$$F_0 = 1, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}$$

首先编写一个程序来计算 F_n，其中输入的值是交互式的 n。

然后编写一个程序，打印出一组斐波那契数；交互式设置数量。

注意 5 如果你让你的斐波那契程序每次计算一个值时都打印出来，你会发现大多数值都被计算了几次。*(* 数学问题：多少次 *?)* 这在运行时是浪费的。这个问题在节 *66.3.1*.

注意 6 一个函数不需要直接调用自己才能是递归的；如果它间接调用，我们可以称之为 相互递归。

```
int f(int n) { return g(n-1); }int g(int n)
{ return f(n); }
```

现在我们有一个问题：`f` 引用 `g` 在后者定义之前。参见节 *19.2* 关于 前向声明。

7.7 其他函数主题

7.7.1 相关核心指南

一些核心指南对于函数 是:

F.2 函数应该执行单个逻辑操作 F.3 保持函数简短且简单 F.16 对于 “ 输入 ” 参数，通过值传递廉价复制的类型，其他通过引用传递 const F.17 对于 “ 输入输出 ” 参数，通过引用传递非 const F.20 对于 “ 输出 ” 输出值，优先返回值而不是输出参数

7.7.2 默认参数

函数可以有 默认参数 (s):

```
double distance( double x, double y=0)
{returnsqrt( (x-y)*(x-y) );
```

```
    }
      ...
      d= distance(x); // 到原点的距离 d  dit  (  )
      //dit  sbtance x,y ;       s ance  e ween two points
```

Any default argument(s) should come last in the parameter list.

除非我指示，否则不要跟踪函数：

```
    void dosomething(double x, bool trace=false)        {
      if(trace) // 报告一些内容
    };
    int main() {
      dosomething(1); // this one I trust
      dosomething(2); // this one I trust
      d      thing(3,true); // thi  s one  I  ant to trace!
        osome wdosomething(4);// 这一个我信任
      dosomething(5); // this one I trust
```

7.7.3 多态函数

You can have multiple functions with the same name:

```
    double average(double a,double b) {
      return (a+b)/2; }
    double average(double a,double b,double c) {
      return (a+b+c)/3; }
```

通过类型或输入参数的数量区分：不能仅通过返回类型不同而不同 pe.

```
    int f(int x); string f(int x); // DOES NOT
    WORK
```

7.7.4 数学函数

Some 数学函数，例如 `abs`, 可以通过 *cmath* :

```
#include <cmath>
<using>{std}{::}{abs}{;}
```

注意 `std::abs` 是多态的（见第 7.7.3 节）。如果没有该命名空间指示，将使用整数函数 `abs`，编译器可能会建议您使用 `fabs` 来处理浮点参数。

其他数学函数，例如 *max*, 位于不太常见的 `algorithm` 头文件（见第 14.3 节）：

```
#include <algorithm><using><std>{v9}::<max>{v13};</using>
```

7.7.5 栈溢出

到目前为止，你只见过非常简单的递归函数。考虑一下这个函数

$$\forall_{n>1}: g_n = (n-1)\cdot g(n-1), \qquad g(1) = 1$$

及其实现：

```
int multifact(int n ) {if (n)==1)return 1;else {int
oneless = n-1;return
oneless*multifact(oneless);} }
```

现在函数有一个局部变量。假设我们计算 $g(3)$。这涉及到

```
int oneless = 2;
```

然后是 g_2 的计算。但这个计算涉及到

```
int oneless = 1;
```

我们是否仍然能得到正确的结果 g_3? 它是否将计算 $g_3 = 2 \cdot g_2$ 或 $g_3 = 1 \cdot g_2$?

不必担心：每次你调用 `multifact` 时，一个新的局部变量 `oneless` 会在 栈 上创建。这是好的，因为它意味着你的程序将是正确的，但它也意味着如果你的函数同时有

- 大量局部数据，和
- 大量的 递归深度 ,

它可能会导致 栈溢出。

注意 7 历史说明：非常古老的 *Fortran* 版本没有这样做，因此递归函数基本上是不可能的。

7.8 复习问题

Review 7.3. What is the output of the following programs? Assume that each program starts with

```
#include <iostream>
using std::cout;
using std::endl;
```

```
int add1(int i) {
  return i+1;
}
int main() {
  int i=5;
  i= add1());
  cout <<i << endl;
}
```

```
void add1(int i) {
  i = i+1
}
int main() {
  int i=5;
  add1(i);
  cout<< i<<endl;
}
```

```
void add1(int &i) {i i+1
     =     ;
  }intmain() {

  int i=5;
  add1((i);
  cout <<i <<  endl;
}
```

```
int add1(int &i) {
  return  i+1;
  }intmain() {

  int i=5;
  i =  add1(i);
  cout <<i <<  endl;
}
```

Review7.4. 假设有一个函数

```
bool f(int);
```

给定一个对于某些正输入值是真值的函数。编写一个主程序，找出使 `f` 为真的最小正输入值。

Review7.5. 假设有一个函数

```
bool f(int);
```

给定一个对于某些负输入值是真值的函数。编写一个代码片段，找出使 `f` 为真的绝对值最小的（负）输入值。

Review7.6. 假设有一个函数

```
bool f(int);
```

给定一个计算整数某些属性的函数。编写一个代码片段，测试 $f(i)$ 是否对某些 $0 \leq i < 100$ 为真，如果是，则打印一条消息。

Review7.7. 假设有一个函数

```
bool f(int);
```

给定一个计算整数某些属性的函数。编写一个主程序，测试 $f(i)$ 是否对所有 $0 \leq i < 100$ 为真，如果是，则打印一条消息。

7. 函数

第 8 章

作用域

8.1 作用域规则

作用域的概念 作用域 回答了这样一个问题：“ 一个名称（读作：变量）与内部实体之间的绑定何时有效? ”

8.1.1 词汇作用域

C++，与 Fortran 和其他大多数现代语言一样，使用 词汇作用域 规则。这意味着你可以通过文本确定一个变量名引用的是什么。

```
int main() { int i; if (
something ) { int j; // code
with i and j } int k; // code
with i and k }
```

- 变量的词法作用域 `i`,`k` 是主程序，包括其中的任何块，例如条件块，从定义点开始。你可以认为变量在内存中创建时，程序执行到达该语句时，之后可以按该名称引用它。

- 变量 `j` 的作用域仅限于条件语句的真分支。如果执行了真分支，则仅创建整数值，并且可以在执行期间引用它。执行离开条件语句后，该名称不再存在，内存中的整数值也随之消失。

- 一般来说，你可以说任何 使用 名称都必须在该变量的词法作用域内，并且在其 定义之后。

8.1.2 遮蔽

作用域可以由同名的变量出现来限制：

Code:

```
1 // /shadowtrue.cpp
2     bool something{true};
3     int i = 3;
4     if ( something ) {
5       int i = 5;
6       cout << "Local: " << i << '\n';
7     }
8     cout << "Global: " << i << '\n';
9     if ( something ) {
10      float i = 1.2;
11      cout << "Local again: " << i <<
      '\n';
12    }
13    cout << "Global again: " << i <<
      '\n';
```

Output
[basic] shadowtrue:

```
Local: 5
Global: 3
Local again: 1.2
Global again: 3
```

第一个 变量 i 具有整个程序的词法范围，减去两个条件语句。虽然它的 生命周期 是整个程序，但在某些地方不可达，因为它被条件语句中的 变量 i 遮蔽 。

这与动态 / 运行时行为无关！

练习 8.1. 这段代码的输出是什么？

```
// /shadowfalse.cpp
    bool something{false};
    int i = 3;
    if ( something ) {
      int i = 5;
      cout << "Local: " << i << '\n';
    }
    cout << "Global: " << i << '\n';
    if ( something ) {
      float i = 1.2;
      cout << i << '\n';
      cout << "Local again: " << i << '\n';
    }
    cout << "Global again: " << i << '\n';
```

练习 8.2. 这段代码的输出是什么？

```
for(int i=0;i<2;i++) { int j; cout
<< j<< endl{v34}; j = 2; cout
<< j<< endl{v49}; }
```

8.1.3 生命周期与可达性

函数的使用在词法作用域故事中引入了一个复杂性：一个变量可以存在于内存中，但可能无法文本上访问：

```
void f() { ... }
int main()
{int i;f();cout
<< i; }
```

f 执行期间，变量 *i* 存在于内存中，且在 *f* 执行后未改变，但它不可访问。

这种情况的一个特例是递归：

```
void f(int i) {
  int j = i;
  if (i<100)
    f(i+1);
}
```

现在，*f* 的每个实例都有一个局部变量 *i*；在递归调用期间，外部的 *i* 仍然存在，但它不可访问。

8.1.4 作用域的微妙之处

8.1.4.1 前置声明

如果你有两个函数 *f*, *g* 互相调用，

```
int f(int i) { return g(i-1); }int g(int
i) { return f(i+1); }
```

你需要一个前置声明

```
int g(int);  int f(int i) {    return g(i -
1);  }  int g(int i) {    return f(i+1);  }
```

由于 name*g* 的使用必须在其声明之后。

还有类的前向声明。如果一个类包含对另一个类的指针，你可以使用它：

```
类 B; 类 A
{private:shared_
ptr<B> myB;};类B
{private:int
myint;}
```

如果一个类是参数或返回类型，你也可以使用前向声明：

```
类 B; 类 A
{
```

8. 范围

```
public:B
GimmeaB();}; 类
B
{
p
ublic:B(int);}
```

然而，这里有一个微妙之处：在 A 的定义中，你不能给出返回 B 的函数的完整定义：

```
类 B; 类 A {public:B GimmeaB() { 返回 B(5); }; // 错误：无法编译 };
```

因为编译器尚未知道 B 构造函数的形式。

The right way:

```
类 B; 类 A {public:B GimmeaB();};
类 B {public:B(int);}B
A::GimmeaB() { 返回 B(5); };
```

8.1.4.2 闭包

使用 lambda 或 闭包 (第 13 章) <style id='10'> 会带来一般作用域规则的另一个例外。阅读关于 ‘ 捕获 ’ 的内容。

8.2 静态变量

函数中的变量具有 词法作用域 ， 仅限于该函数。通常它们还具有 动态作用域 ， 仅限于函数执行：函数结束后它们完全消失。（类对象会调用其析构函数 。）

有一个例外：一个静态变量 在函数调用之间持续存在。

```
void fun() { static int
remember; }
```

例如

intonemore() {static int remember++;;return remember;; }intmain() {for(...)cout << onemore() <<end;return 0; }

给出一个整数流。

练习 8.3. `onemore` 函数中的静态变量从未被初始化。你能找到一个机制来初始化它吗？你能用函数的默认参数来做吗?

8.3 范围和内存

范围的概念与变量对应内存中的对象这一事实相关。内存仅在实体的动态范围内为实体保留。在简单情况下，这一点很清楚：

```
intmain() {// 为’i’ 预留内存 if(true) {inti; // 现在
为整数 i 预留内存 ...code ...}// 释放’i’ 的内存 }
```

递归函数带来一个复杂性：

int f(int i) { int itmp; ... code with ‘itmp’ ... if (something) return f(i-1); else return 1; }

现在每次对 `f` 的递归调用都会为其自己的 `itmp` 实例预留空间。

在上述两种情况下，变量都被认为是在 栈 上：每一层作用域嵌套或递归函数调用都会创建新的内存空间，并且该空间在包含层释放之前被释放。

对象像变量一样表现，如上所述：当它们超出作用域时，其内存会被释放。然而，除此之外，还会在对象及其所有包含对象上调用一个析构函数：

代码:

```
1 // /destructor.cpp
2 类 SomeObject {
3 public:
4   SomeObject() {
5     cout << "calling the constructor"
6          << '\n';
7   };
8   Some 对象 () {
9     cout<< "调用析构函数"
10         << '\n';
11  };
12 };
```

输出
[对象] 析构函数:

在嵌套作用域调用之前
g 构造 uctor
在嵌套作用域内
调用析构函数
在嵌套作用域之后

8.4 复习问题

Review8.1. 这是否是一个有效的程序?

void f() { i ; }int
main() { int i; f();
return 0; }

If yes, what does it do; if no, why not?

Review8.2. 输出是什么:

#include <<iostream>>
<using
std::{v9}::cout{v15};<using
std::{v19}::endl{v25};<int
main() {<int i=5;<if(true) { i
= 6; }<cout <<i endl<< ;
<return 0; }
</int></int></int></int>

Review8.3. 输出是什么:

#include <<iostream>><
using
std::{v9}cout{v15};<using
std::{v19}endl{v25};<int
main() {<int i=5;<if (true) {
<int i = 6; }<cout <<i <<
endl{v61};<return 0;

```
}
```

Review8.4. 输出结果是什么：

```
#include <<iostream>>
<using
s
td::{v9}cout{v15};<using
std::{v19}endl{v23};<int
main() {<int i=2;<i += /*
5; i +=*/<6;cout <<i <<
endl{v56};<return 0; }
</int></int>
```

8. 范围

第 9 章

类和对象

9.1 什么是对象?

现在你已经学习了基本数据、控制结构和函数。归根结底，这就是编程：对数据进行操作。然而，为了使程序易于管理，最好对它们进行结构化，并认识到你实际上想要在更高的抽象层次上进行交流。

C++ 提供了一个重要的机制，将数据和操作统一在一起，以提供一个新的抽象层次：对象属于 类。

> 对象是一个你可以请求它执行某些操作的实体。Wh d i i l fi t
> k lfen‘ hestfgn hig h c ass, rs as yourse : w a unc ona y s ould the objects
> 支持 ’ 。

- 对象能够执行的操作是对象的方法 或 成员函数；和
- 为了使这些操作成为可能，对象可能存储数据，即数据成员。Obj t i *I* A l i lik d t t k bj t f l
- ec s comes n *c ass*es. c ass s e a a a ype: you can ma e o ec s o a c ass like you 创建数据类型的变量。
- 同一类的对象具有相同的方法。它们也具有相同的成员，但具有各自的值。

类与数据类型类似，你可以声明该类型的数据，然后可以在表达式中使用这些数据。与基本数据类型不同，它们不是预定义的，因此你首先需要定义类，才能创建该类的对象。

- 你需要一个类定义，通常放在主程序之前。
- (在较大的程序中，你会把它放在包含文件 或 模块中。)
- 然后你可以声明属于该类的多个对象。
- 对象然后可以在表达式中使用，作为参数传递，等等。

9.1.1 第一个示例：pointsintheplane

在本节中，我们将使用一个简单的示例来说明如何使用对象：我们将创建一个 点 对象，对应于 $\mathbb{R}^2$ 中的数学点。

介绍将较为自上而下：我们将更关注要做什么，而不是如何做。不用担心，所有细节都会在适当的时候覆盖。

练习 9.1. 思考练习：点对象应该能够执行哪些操作?

我们对点要做的第一件事是查询它的一些数学属性。例如，给定一个点，你可能想知道它到原点的距离或它与 x- 轴的角度。

小插图：点对象。

Code:

```
// /functionality.cpp
  Point p(1.,2.); // make point (1,2)
  cout << "distance to origin "
       << p.distance_to_origin()
       << '\n';
  p.scaleby(2.);
  cout << "distance to origin "
       << p.distance_to_origin()
       << '\n'
       << "and angle " << p.angle()
       << '\n';
```

Output

```
[object] functionality:
distance to origin 2.23607
distance to origin 4.47214
and angle 1.10715
```

Note the 'dot' notation.

练习 9.2. 思考练习:
对象需要存储哪些数据才能计算与原点的角度和距离? 有不止一种可能性吗?

值得思考：你可能会想为获取 x 和 y 坐标编写方法。然而，问问自己这些是否应该是公开可见的方法。获取 x 坐标是一种线性代数操作吗? 当你有诸如 " 获取到原点的距离 " 或 " 将此点向右移动 " 这样的方法时，你是否需要显式地使用坐标?

上面的示例使用了 点 对象，但没有说明它是如何创建的。这就是我们接下来要研究的。

- 首先定义类，包含数据和函数成员:

  ```
  类 MyObject{// 定义类成员 // 定
  义类方法 };
  ```

  (详情稍后) 通常在 *main* 之前。
- 你通过声明创建特定的对象

  ```
  我的对象
    object1( /* .. */ ),
    object2( /* .. */ );
  ```

- You let the objects do things:

  ```
  对象1. 做_这个();x = 对象2. 做_那个( /* ...
  */ );
  ```

现在让我们详细介绍所有这些步骤的细节。

9.1.2 构造函数

首先我们将查看创建类对象，我们将坚持使用点示例。

由于一个点可以通过其 x,y 坐标来定义，你可以想象到

- 点对象存储这些坐标，并且
- 当您创建一个点对象时，您是通过指定坐标来完成的。

以下是相关的代码片段：

对象的声明 `x` 类 `Point`；
点的坐标最初设置为
1.5,2.5。

```
Point x(1.5, 2.5) ;
```

```
类 点 {
private: // 数据成员
   double x,y;
public: // function members
   点
     ( double x_in,double y_in ) {
       x= x_in; y =y_in;
   };
   /* ... */
};
```

仔细研究其实现。该类名为 `Point`，并且有一个看起来像函数定义的东西，也名为 `Point`。然而，与普通函数不同，它没有返回类型，甚至 `void`。

这个函数是类的构造函数，其特点如下：

- 构造函数与类同名，并且
- 看起来像函数定义，但它没有返回类型 .

当你创建对象时，就像上面例子中看到的那样，你实际上调用了这个构造函数。

通常你会编写自己的构造函数，例如以初始化数据成员。对于类 `Point` 来说，构造函数的作用是获取点的坐标并将它们复制到 `Point` 对象的私有成员中。

如果创建的对象可以有合理的默认值，你也可以使用 默认构造函数，它没有参数。我们将在下面讨论这一点；章节 9.1.7。

9.1.3 数据成员

到目前为止的示例中，你已经从其坐标创建了一个点对象

```
点一一(1.,1.);
```

并且 `Point` 对象存储了这些坐标。然而，这种外部使用和内部实现之间的连接可以非常不同。也许你有一个在极坐标中工作的应用程序，在这种情况下，存储 r,θ 更自然，或者至少对计算更方便。但你可能仍然想从笛卡尔坐标创建一个点。

构造函数的参数并不暗示存储了哪些数据成员！

示例：创建一个在 x, y 笛卡尔坐标系中的点，但存储 $r, theta$ 极坐标：

```
1#include <cmath>2class Point {
3private: // 成员 4 double r,theta;
5public: // 方法 6 Point( double
x,double y ) { 7 r =sqrt(x*x+y*y); 8
theta= atan2(y/x); 9 }
```

注意：对外 API 没有变化。

您现在已经看到了 **private** 和 **public** 关键字。这些关键字表示类成员的可见性。

- 关键字 **private** 表示数据是内部的：不能从对象外部访问；只能在使用函数成员内部使用。
- 关键字 **public** 表示构造函数可以在程序中使用。

您可能已经注意到，私有成员都是数据成员，而所有函数成员都是公有的。这不是巧合：目前您可以将其视为一种 ‘最佳实践’。

我们将使用的最佳实践：

```
类 MyClass{ 私有: //
数据成员公有: // 方法 }
```

- 数据是私有的：在对象外部不可见。
- 方法都是公有的：可以在使用对象的代码中使用。
- 你可以有多个私有 / 公有部分，顺序任意。

9.1.4 方法

方法是你可以要求你的类对象执行的事情。例如，在 *Point* 类中，你可以要求一个点报告它到原点的距离，或者你可以要求它将它的距离按某个数进行缩放。

让我们从这两个中较简单的一个开始：测量到原点的距离。没有类和对象，你会写一个函数，以 x,y 坐标作为输入，以及一个数字作为输出

```
float x = ..., y = ...;
float d = distance_to_origin(x,y);
```

对于对象方法，它看起来像：

float x=..., y=...;Point p(x, y);float d =
p.distance(to origin());

指出差异和相似之处:

- 你仍然在使用一个具有标量输出的函数，但是
- 我们没有使用输入参数，而是使用存储在点对象中的坐标。这些充当 “ 全局变量 ”，至少在对象内部是这样。
- 要将此函数应用于一个点，我们使用 “ 点 ” 表示法。你可以将其读作 “p 到原点的距离 ”。请注意，此函数只能作为应用于点来使用；你不能在其他上下文中使用它。

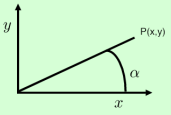
distance 函数的定义和使用:

代码:

```
1// /pointodist.cpp
2<code>class</code><code>Point</code>{
3 私有:
4    float   x,y;
5public:
6    点 (floatux,float uy) { x =      ux; y =
       uy; };
7    float distance_to_origin() {
8      return   sqrt( x*x   y*y  ;
9    };
10 };
11     /* ... */
12    点 p1(1.0,1.0);
13    float d= p1.{distance}_to_origin();
```

输出
[几何] 点到距离
到原点的距离: *1.41421*

练习 9.3. 为 `Point` 类添加一个 `angle` 方法。它需要多少个参数?



提示: 使用函数 `atan` 或 `atan2`。

你可以参考代码仓库中的文件 *pointclass.cpp*。

练习 9.4. 创建一个 `GridPoint` 类，它只能有整数坐标。实现一个函数 `manhattan_distance`，它计算从原点到该点的曼哈顿距离，即水平方向和垂直方向步数的总和。

9.1.5 初始化

设置对象数据成员的初始值有多种方法。

9.1.5.1 默认值

有时，如果未指定其他内容，对象的默认值是有意义的。这可以通过在数据成员上设置默认值来完成：

类成员可以像普通变量一样具有默认值 :

```
class Point
{private:float x=3.,
y=.14;public://等等}
```

每个对象都将被初始化为这些值。

9.1.5.2 构造函数中的初始化

前面你看到了一些用于初始化对象数据的构造函数示例。有不止一种方法可以做到这一点。

首先，你可以在构造函数体内复制构造函数参数。然而，更推荐的方法是使用成员初始化器，它采用了一种新的语法。

简单的方法：

```
1class Point{
2 私有: d bl
3     ou   e x,y;
4public:
5   Point(   double in_x,
6            double in _ y) {
7     x = in_x; y = in_y;
8   };
```

首选方法：

```
1 // /pointinit.cpp
2 类 点 {
3 私有 :
4   double x,y;
5 公有 :
6   点 (      double      用户 x
                >, double usery)
7
8       : x ( 用户 x) , y ( 用户 y) {
9   }
```

(参见第 10.6 节，了解为什么推荐使用成员初始化器。)

您甚至可以不必考虑太多名称：

代码:

```
// /pointinitxy.cpp
类 点 {
私有: d  bl
   ou  e x,y;
公有:
   点 (double x,double y      )
        : x(x),y(y) {
   }
     /* ... */
   点 p1(1.,2.);
   cou  <<  p1 = "
        <<p1. getx() <<      "," << p1.gety()
        << '\n';
```

输出
[geom] pointinitxy:

```
p1 = 1,2
```

初始化 `x(x)` 应理解为 `membername(argumentname)`。是的，`x` 出现两次有点令人困惑。

9.1.6 方法，更深入的探讨

你已经看到了类 方法 的例子：一个仅定义为该类对象中的函数，并且可以访问该对象的私有数据。

在练习 9.3 中你实现了一个 角度 函数，该函数根据存储的坐标计算角度。你可以做出其他决定。

Exercise 9.5. Discuss the pros and cons of this design:

1class Point { 2 private: 3 double x,y,r,theta; 4 public: 5 Point(double xx,double yy) { 6 x = xx; y= yy; 7 r =// sqrt something 8 theta= // something trig 9 }; 10 double angle() { return alpha; }; 11 };

通过将这些函数公开，并将数据成员设为私有，您为该类定义了一个应用程序接口（API）：

- 您正在为该类定义操作；它们是访问对象数据的唯一方式。
- 方法可以使用对象的数据，或修改它。所有数据成员，即使被声明 `private`，对方法来说都是全局的。
- 被声明 `private` 的数据成员不能从对象外部访问。

复习 9.1. T/F?

- 一个类主要由它存储的数据决定。

- 类主要由其方法决定。
- 如果你更改类数据的结构，你需要更改构造函数调用。

现在让我们看看一些不同类型的对象。这是一种非正式的分类，不一定与 C++ 标准中定义的概念相对应。

9.1.6.1 更改状态

对象通常有数据成员来维护对象的 状态 。通过更改成员的值，你更改了对象的状态。通常通过方法来完成这一操作。

例如，您可能希望按某个量缩放一个向量：

代码：

```
// /pointscaleby.cpp
class Point { //
      * ... *
   void scaleby ( double     a ) {
      x *=a; y*= a; };
      /* ... */
};
      /* ... */
   点 p1(1.,2.);
   cout <<"p1 到原点 "
        <<p1. 长度 () << '\n';
   p1.scaleby (2) . ;
   cout <<"p1 到原点 "
        << p1.length() << '\n';
```

输出
[geom] pointscaleby：

p1 到原点 *<2.23607>*
p1 到原点 *<4.47214>*

Exercise 9.6. Implement a method `shift_right` for the `Point` class.

练习 9.7。 考虑使用极坐标（见上文）设计的 点 类。实现一个 旋转 方法。

这里有一个微妙之处。提示：想象将一个点旋转足够多次。

9.1.6.2 返回对象的 方法

你之前看到的那些方法只返回了基本数据类型。也可以返回一个对象，即使是从同一个类返回。例如，与其缩放向量对象的成员，你可以基于缩放后的成员创建一个新的对象：

代码:

```
// /pointscale.cpp
类 点 {/        /
      * ... *
  点缩放 (double a ) {
    auto 缩放点 =
         点 (x*a, y*a );
    返回 缩放点 ;
  };
    /* ... */
  cout << "p1 to origin "
       << p1. 到 _ 原点的距离 _
       << '\n';
  点 p2 = p1 . 缩放 (2.);
  cout <<"p2 to origin "
       << p2. 距离 _ 到 _ 原点
       <<' \ 'n ;
```

输出 [几何] 点缩放:

p1 到原点 *<2.23607>*
p2 到原点 *<4.47214>*

通过缩放另一个点来创建一个点:

```
新 _ 点 =  旧 _ 点 . 比例 (2.81);
```

两种处理返回 语句的方式:

简单:

```
// /pointscale.cpp
点 点 :: 比例 ( double a )        {
  点缩放点 =
    Point(x*, y*);
  返回 scaledpoint;
};
```

Creates point, copies it to *new_point*

更好:

```
// /pointscale.cpp
Point Point::scale( double a ) {
  返回点 (x*a,y*a);
};
```

创建点，直接移动到新 _ 点

'move semantics' and 'copy elision':compiler is pretty good at avoiding copies

9.1.7 默认构造函数

您现在已经看到了一些类及其构造函数的例子。这些构造函数接受参数来设置对象的初始状态。

然而，如果您的对象具有合理的默认值，您可以使用一个默认构造函数。例如:

没有显式定义的构造函数；您可以通过 main 中的事实来识别默认构造函数，即对象定义时没有任何参数。

代码:

```
1// /default.cpp
2classIamOne{
3 private:
4   int i=1;
5 public:
6   void print() {
7     cout << i << '\n';
8   };
9 };
10     /* ... */
11   IamOne one;
12   一个 .print();
```

Output
[对象] 默认 no:

```
1
```

您可以自己定义默认构造函数，但之前的示例有一个 已默认 默认构造函数：它表现得好像有一个构造函数

IamZero() {};

在您学习以下代码时，请记住这一点：

// /pointdefault.cppPoint p1(1.,2.),p2;cout "p1 to origin "
p1.length() ’ \n’ ;p p1p1.scale(2.);cout "p2 to origin "
p2.length() ’ \n’ ;

使用 `Point` 类（及其构造函数）如前所示：

*// /pointodist.cpp 类 Point { private: float x,y; public: Point(float ux,float uy) { x= ux; y = uy; }; float distance_to_ origin() { return sqrt(x*x+ y*y); }; }; /* ... */ Point p1(1.0,1.0); float d= p1.distance_to_origin();*

这将在编译期间给出错误消息。原因是

Point p2{}

调用默认构造函数。既然你已经定义了自己的构造函数，默认构造函数就不再存在了。所以你需要显式地定义它：

// /pointdefault.cppPoint()
{};Point(double x,double y) :
x(x),y(y) {};

你现在有一个包含两个构造函数的类。编译器将决定使用哪一个。这是一个多态的例子。

你也可以更明确地表示默认的默认构造函数需要存在:

```
// /default.cppPoint() =
default;Point(double x, double
y) : x(x), y(y) {};
```

注意 8 默认构造函数有‘空括号’，但你使用它时不带括号。如果你在创建对象时指定空括号，会发生什么?

```
// /constructparen.cppclassMyClass {public:MyClass() { cout <<"构造!
"<< '\n'; };};intmain() {MyClass x;MyClass
y();
```

constructparen.cpp:24:12: 警告：空括号被解释为函数声明 MyClass y(); ^ constructparen.cpp:24:12: 注释：删除括号以声明变量 MyClass y(); ^ 1 警告生成。

9.1.8 数据成员访问；不变量

您可能已经注意到关键字 `public` 和 `private`。我们将数据成员设为私有，将方法设为公有。C++ 语言也有来自 C 的 `struct` 结构，在那里面，数据成员默认是公有的。为什么我们不这样做呢?

Struct data is public:

```
struct Point {
  double x;
};
int main() {
  Point andhalf;
  andhalf.x = 1.5;
}
```

```
class Point {
public: // Bad! Idea!
  double x;
};
int main() {
  Point andhalf;
  andhalf.x = 2.6;
}
```

对象应该通过其功能来访问。虽然您可以编写如 `get_x` 这样的方法（这称为一个访问器；另见第 18.4 节以了解一些细微差别）来获取 x 坐标 -

nate，问问自己这是否合理。如果你需要 x 坐标来将点向右移动，请编写一个 *shift_right* 方法，而不是其他方法。

- 接口：**public** 方法决定了对象的功能；对数据成员的影响是次要的。
- 实现：数据成员，保持 **private**：它们仅支持功能。

这种分离是一件好事：

- 保护自己免受对象数据意外更改的影响。
- 可以在不重写调用代码的情况下更改实现。

你不应该轻易编写访问函数：你应该首先考虑你的类中哪些元素应该从概念上被外部世界可检查或可更改。例如，考虑一个成员之间存在某种关系的类。在这种情况下，只允许维护该关系的更改。有时人们会说一个类满足一个 不变式。

你在练习 9.7 中已经看到了这种现象。那里的不变式是什么？让我们考虑另一个需要维护不变式的例子。

我们为单位圆上的点创建一个 <style id='1'> 类 </style>。

```
1// /unit.cpp 2 类
UnitCirclePoint { 3 private: 4 float
x,y; 5 public: 6 UnitCirclePoint(float
x) { 7 setx(x); }; 8 void setx(float
newx) { 9 x =newx; y =  sqrt(1-x*x); 10 };
```

你不希望能够只更改一个 x, y！一般来说：对成
员强制不变式。

第 9.5.5 节对直接访问内部 d 的方法进行了进一步讨论 ata.

9.1.9 示例

到目前为止，我们已经看过代表现实世界中 ‘类对象’ 事物的对象示例。然而，我们也可以为更抽象的事物创建对象。在下一个示例中，我们来看 ‘无限对象’，例如所有整数的集合。显然，没有办法存储此类对象的数据，但这里的关键问题是：所有整数的对象有哪些方法？一种可能的设计是你可以要求这个对象 ‘给我下一个整数’。

对象可以模拟相当抽象的事物：

Code:

```
1// /stream.cpp
2类 Stream {
3private:
4   int last_result{0};
5public : it
6    n  next() {
7     return last_result++; };
8};
9
10int main() {
11   Stream ints;
12   cout<<"Ne t x : "
13        << ints.next() << '\n';
14   cout <<"   Next: "
15        << ints.<next>() <<'\n';
16   cout << "Next: "
17        << ints.next() << '\n';
```

Output[

```
bpt  ec ] stream:
接下来： 0
下一节： 1
下一页： 2
```

练习 9.8。

编写一个类 `multiples`，其中每次调用 `next` 都会返回下一个 2 的倍数。

- 编写一个类 *multiples*，用于如下：

```
multiples multiples_of_three(3);
```

其中，*next* 调用给出构造函数参数的下一个倍数。

您可以根据代码仓库中的 *file_stream.cpp* 文件进行参考

练习 9.9。 如果你在做主项目（章节 45），现在是一个好时机来做 45.6 节中的练习。

9.2 类之间的包含关系

一个对象的数据成员可以是基本数据类型，也可以是对象。例如，如果你编写用于管理课程的软件，每个 *Course* 对象很可能会有一个 *Person* 对象，对应于教师。

```
类 Person {string
name;....}类Course
{private:int year;Person
the_instructor;vector<
Person> students;}
```

设计具有相互关系的对象是一种编写结构化代码的绝佳机制，因为它使代码中的对象表现得像现实世界中的对象一样。一个对象包含另一个对象的关系，称为类之间的 *has-a* 关系 。

9.2.1 字面量和隐喻 has-a

有时一个类可以表现得好像包含另一个类的对象，而实际上并不存储这个对象。考虑一条线段，即从一个起点到一个终点的线段。我们希望提供 API：

```
int main() {Segment somesegment( /* something */ );Point
somepoint =somesegment.gettheendpoint();
```

我们可以通过让 `Segment` 类实际存储起点和终点来支持这一点：

```
类 Segment{private:Point starting_点,
结束_点;}
```

或者让它存储从起点出发的距离和角度：

```
类
Segment{private:Point
starting_点;float 长度,
角度;}
```

在这两种情况下，使用对象的代码都像段对象包含两个点一样编写。这说明了面向对象编程如何将类的 API 与其实际实现解耦。

与此解耦相关，一个类也可以有两个非常不同的构造函数。

```
class Segment {
  private:
  // up to you how to implement!
  public:
  Segment( Point start,float length,float angle )
  { .... }
  Segment( Point start,Point end ) { ... }
```

根据你实际如何实现类，一个构造函数将简单地存储定义数据，而另一个将把给定数据转换为实际存储的数据。

这是面向对象编程的另一个优势：你可以改变类的实现，而无需更改使用该类的程序。

当你在类之间存在 has-a 关系时，‘默认构造函数’ 问题（第 9.1.7 节）可能会再次出现：

Class for a person:

```
类 Person {
private:
  string name;
public:
  Person( string name ) {
    /* ... */
  };
};
```

一个课程的类，其中包含一个人：

```
类课程 {
private:
  Person instructor;
  int enrollment;
public: C
   ourse( string instr,int n ) {
    /* ???? */
  };
};
```

您想使用此作为 Co

```
urse("Eijkhout",65);
```

Possible constructor:

```
课程 ( s   字符串教师姓名 ,intnstudents      ) {
  instructor =Person<teachername>;
  注册 =n 学生
};
```

Preferred:

```
Course( strin  teachername, int nstudents )
      g: in instructor(Person(teachername)),
    注册 ( 学生 ) {
};
```

```
类 内部      /* ... */ };
  { 类 外部 {
private:
  Inner inside_thing;
```

构造函数的两种可能性：

```
Outer  Inner thing) : 内部 _
 thing(thing) { };
```

在构造期间，内部对象被复制。
外部 对象。

```
外部 ( 内部事物 ) {
  内部 _ 事物 =  事物 ;
};
```

外部对象被创建，包括 i f bj h h
construct on o Inner o ect, t en t e ar ument is
g 被复制到位置 :⇒ 需要默认构造函数
Inner.

练习 9.10. 如果你在做几何项目，现在是做第 46.3 节练习的好时机。

9.2.1.1 对象的简写

对于有构造函数的类，你可以使用对象的简写，给出用大括号分隔的初始化器列表。

初始化器列表 可以用作表示。

// /rectcurly.cppPoint(float ux,float uy) {/ ...*/Rectangle(Point bl,Point tr) {/* ... */Point origin{0.,0.};Rectangle lielow(origin, {5,2});*

9.3 继承

除了 “ 有 ” 关系之外，还有是关系，也称为继承。在这里，一个类是另一个类的特殊情况。通常，派生类 (特殊情况) 也会继承基类 (一般情况) 的数据和方法。

一般的 `FunctionInterpolator` 类带有方法 `value_at`。派生类:

- *LagranceInterpolator* with*add_* 点 _ 和 _ 值 ;
- *HermiteInterpolator* with*add_* 点 _ 和 _ 导数 ;
- *SplineInterpolator* with*set_* 度 .

如何定义一个派生类? 使用基类和派生类的一般代码模式如下:

基类，一般情况:

```
类 一般 {
protected: // 注意!
 int g;
public:
 void一般 _ 方法 () {};
};
```

派生类，特殊情况:

```
class S ecial : public General {

  vo   special_method() {
    ... g ...
  };
};
```

声明派生类有以下各个方面:

- 您需要指明基类是什么:

```
class Special : public General { .... }
```

- 基类需要将其数据成员声明为 `protected`: 这与 private 类似，只是它们对派生类可见
- The methods of the derived class can then refer to data of the base class;
- 为基类定义的任何方法或数据成员对派生类对象都可用。

派生类有自己的构造函数，其名称与类名相同，但在调用时，它还会调用基类的构造函数。这可以是默认构造函数，但通常你想显式调用基类构造函数，使用描述特殊情况如何与基本情况相关的参数。

在以下示例中，我们有一个一般情况，取决于两个独立参数。特殊情况来自于这些参数之间的一种特定关系。

```
类 一般 {public: 一般 ( double x,double y )
{};}; 类 特殊 : public 一般 {public: 特殊 (
double x) : 一般 (x,x+1) {};};
```

Methods and data can be

- **私有**，因为它们仅用于内部；
- **公共**，因为它们应该可以从类对象外部使用，例如在主程序中；
- **protected**, 因为它们应该在派生类中可用。

练习 9.11。 如果你在做几何项目，你现在可以做一些第 46.4 节中的练习。

9.3.1 基类和派生类的方法

在上面，假设派生类使用基类的方法不变。然而，有时你可能希望派生类有一个不同版本的方法。这是通过 **virtual** 和 *override* 关键字完成的。

- 派生类可以继承基类的方法。
- 派生类可以定义基类没有的方法 .
- 一个派生类可以重写 一个基类方法:

```
1 类 Base {2 public:3 虚拟 f() { ... };4 };
5 类 派生 : public Base {6 public:7 虚
拟 f() 重写 { ... };8 };
```

代码：

```
// /virtual.cpp
<code> 类 </code><code> 基类 </code>{
protected:
    int i;
public:
    基类 (int i) :i(i) {};
    虚拟 int<value>() { return i; };
};

<code>class</code> <code>Deriv</code><code> : public</code> <code>Base</code><code>{</code>
public:
    派生 (int i) : 基类 (i) {};
    虚拟 int 值 () 重写 {
        int ivalue= 基类 :: 值 ();
        返回 ivalue*ivalue;

    };
};
```

输出

```
[ 对象 ]  虚拟 :
25
```

9.3.2 虚拟方法

基类和派生类的方法可以以多种方式关联。

- 基类中定义的方法：可以在任何派生类中使用。
- 派生类中定义的方法：只能在该特定的派生类中使用。
- 在基类和派生类中都定义的方法，标记 `override`：派生类方法替换（或扩展）基类方法。

- 虚拟方法：基类仅声明某个例程必须存在，但不提供基类实现。如果类包含虚拟方法，则称为 抽象类 ；如果所有方法都是虚拟的，则称为纯虚拟。不能创建抽象对象。

抽象方法的特殊语法：

```
类 基 {public: 虚拟 void f()
  = 0; };类 派生 : public 基 {
public: 虚拟 void f() { ... }; };
```

```
    // /virtualvec.hpp 类
VirtualVector{private:public: 虚拟
voidsetlinear(float)  = 0; 虚拟 float
运算符 [](int)  = 0;}


 ;
```

Suppose *DenseVector* derives from *VirtualVector*:

```
// /densevec.cpp
  DenseVector v<5>;
  v.setlinear(7.2);
  cout << v[3] << '\n';
```

```
// /densevec.cpp
类DenseVector :VirtualVector                    {
私有 :t
  vec 或 <float>  值;
public:
  DenseVector(    int  size) {
    values  = vector<float>(size,0);
  };
  void setlinear( float v ) {
    for (int i=0; i<values.size();
    ++i)
      values[i] = i*v;
  };
  float operator[](int i) {
    返回值。在 (i);
  };
};
```

Exercise9.12. 编写一个用于常微分方程（ODEs）的小型 ‘积分器’ 库。假设 ‘自洽微分方程’，即 $u' = f(t)$ 且无 u- 依赖性，则有两种简单的积分方案：

- 显式：$u_{n+1} = u_n + \Delta t\, f_n$; 和
- 隐式：$u_{n+1} = u_n + \Delta t\, f_{n+1}$.

编写一个 *Integrator* 类，其中 *nextstep* 方法（用于对另一个 Δt 进行积分）是纯虚函数；然后编写 *ExplicitIntegrator* 和 *ImplicitIntegrator* 类从此类派生。

```
// /pureint.cpp
    double stepsize = .01;
    auto integrate_linear =
      ForwardIntegrator( [] (double x) { return x*x; }, stepsize );
    double int1 = integrate_linear.to(1.);
```

你可以将待积分的函数硬编码，或者尝试传递一个函数指针。

9.3.1 友类

一个 *friend* 类即使没有继承关系也可以访问私有数据和 <style id='2'> 方法。

```
1/* 前向声明: */ 类 A;
2 类 B {
3  // A 对象可以访问 B 内部: 4 friend
classA;5 private:6   inti;7 };8class A {9
public:10 void f(B b) {b.i; }; // friend
access
```

```
11 };
```

9.3.4 多重继承

- 多重继承：一个 X 是 A，但也是 B。这种机制有点危险。
- 虚基类：你实际上不在基类中定义函数，你只是说 “ 任何派生类都必须定义这个函数 ”。

练习 9.13。 如果你在做几何项目，现在是做第 46.2 节练习的好时机 。。

9.4 更多关于构造函数的内容

9.4.1 委托构造函数

如果你有两个构造函数，其中一个是另一个的特殊情况，有一个优雅的机制来表达这一点：委托构造函数。

例如，考虑一个包含向量的类，并且你想在构造函数中设置这个向量。我们可以这样实现：

```
类 HasVector{private:: vector<int>
values;public:: HasVector(vector<int>
initvalues) : values(initvalues) {};
```

现在假设我们希望初始值向量只是存储向量的前一部分。现在我们需要一个接受初始值和一个表示完成大小的整数的构造函数。

```
HasVector(vector<int>init, int size) :
values(vector<int>(size)) {   int loc;   for (auto i :
init)      values[loc++] = i;}
```

(问题：这个构造函数有点危险。问题是什么，你将如何防范？)

We can now let the first constructor delegate to the more general one:

```
HasVector(vector<int>  init) : HasVector(init,
init. size() ) {};
```

在这里，我们使用了冒号语法来 “ 委托 ” 构造函数：一个构造函数是用另一个构造函数来表示的。(问题：在类的上下文中，冒号语法还有哪两种用途？)

所有内容一起：

类 HasVector { private: vector<int> values;
public: HasVector(vector<int>init) :
HasVector(init,init.size()) {}; HasVector(vector<
int>init,int size) : values(vector<int>(size)) { int
loc=0; for (auto i : init) values [loc++] =i; };

9.4.2 拷贝构造函数

就像如果你没有定义显式构造函数，则会定义默认构造函数一样，存在一个隐式定义的 拷贝构造函数。你可以进行拷贝的两种方式会调用两个略有不同的构造函数：

```
my_ 对象 y(something); // 普通或默认构造函数 my_ 对象 x(y);
// 拷贝构造函数 my_ 对象 x= y; // 拷贝赋值构造函数
```

通常隐式定义的拷贝构造函数会做正确的事情：它复制所有数据成员。（如果你想定义自己的拷贝构造函数，你需要知道它的原型。我们不会深入讨论。）

以拷贝构造函数的示例为例，让我们定义一个将整数存储为数据成员的类：

```
// /copyscalar.cpp
class has_int {
private:
  int mine{1};
public:
  has_int(int v) {
    cout << "set: " << v
         << '\n';
    mine = v; };
  has_int( has_int &h ) {
    auto v = h.mine;
    cout << "copy: " << v
         << '\n';
    mine = v; };
  void printme() {
    cout << "I have: " << mine
         << '\n'; };
};
```

以下代码显示了数据被复制过来：

代码：

```
1 // /copyscalar.cpp2has_int an_int(5);3has_int other_int( an_int );4an_int.printme();5 other_int.printme();6has_int yet_other = other_int;7yet_other.printme();
```

输出

```
[object] copyscalar:
: 5copy:
5I have:
5I have:
5copy: 5I have: 5
```

带向量的类：

```
1// /copyvector.cpp 2 类 has_vector { 3 private: 4 vector<int>myvector; 5 public: 6 has_vector(int v) { myvector.push_back(v); }; 7 void set(int v) { myvector.at(0) = v; }; 8 void printme() { cout9 <<"I have: " << myvector.at(0) << ’\n’; }; 10 };
```

复制是递归的，所以复制有自己的向量：

代码：

```
1// /copyvector.cpp2has_vector a_vector(5);3has_vector other_vector(a_vector);4a_vector.set(3);5a_vector.printme();6other_vector.printme();
```

Output

```
[object] copyvector:
I have: 3
I have: 5
```

9.4.3 析构函数

就像有一个构造函数来创建对象一样，也有一个 析构函数 来销毁对象。与默认构造函数的情况一样，有一个默认析构函数，你可以用自己的来替换它。

如果一个对象包含动态创建的数据，析构函数会很有用：你想使用析构函数来释放那些动态数据，以防止产生 内存泄漏。另一个例子是关闭文件，其 文件句柄 存储在对象中。

析构函数通常在你没有察觉的情况下被调用。例如，任何静态创建的对象在控制流离开其作用域时会被销毁。

示例：

代码:

9.4.3 析构函数

```
类 SomeObject {
public:
  SomeObject() {
    cout << "calling the constructor"
         << '\n';
  };
  ~SomeObject() {t
    cou  << "calling the destructor"
         << '\n';
  };
};
```

输出
[对象] 析构函数:

```
在嵌套作用域之前
calling the cons truct
    或 在嵌套作用域内
调用析构函数
在嵌套作用域之后
```

Exercise 9.14. Write a class

```
类 HasInt {private:int
mydata;public:HasInt(int v) { /* 初始
化 */ };...}
```

used as

Code:

```
// /destructexercise.cpp
  {
    HasInt v(5);
    v.set(6);
    v.set(-2);
  }
```

Output
[object] destructexercise:

```
**** object created with 5 ****
**** object set to 6 ****
**** object set to -2 ****
**** object destroyed after 2
    updates ****
```

当您抛出异常时，会调用析构函数:

Code:

```
1 // /exce_tdestruct.cpp
2 类 SomeObject {
3 public:
4    SomeObject() {
5       cout << "调用构造函数"
6            << '\n'; };
7    SomeObject() {
8       cout<<  "调用析构函数"
9            << '\n'; };
10 };
11    /* ... */
12   try  {SomeObject 对象;
13
14      cout<< "在嵌套作用域内"                <<
        \n;
15      抛出(1);
16   } catch(...) {
17      cout << "Exception caught"<< '\n';
18   }
```

输出
[对象] 异常析构:

```
调用构造函数
在嵌套作用域内
调用析构函数
异常捕获
```

9.5 高级主题

9.5.1 核心准则

Some of the 对象 的核心准则:

C.1 将相关数据组织到结构（structs 或 类）中 C.3 使用 类 表示接口和实现的区别 C.4 仅当函数需要直接访问类的表示时，才将其作为成员 C.10 优先使用具体类型而不是类层次结构 C.11 使具体类型成为常规类型

9.5.2 静态变量和方法

以 **static** 前缀的 类 成员表现得好像它们不是每个该 类 对象独有的，而是它们之间共享的。

这种标准用途是统计已创建的 类 对象数量。构造函数会递增这个静态变量，并将其赋值给一个私有变量:

Thing::Thing(int mynumber, int n, int things);

声明静态变量使用关键字 **static** 和 **inline**:

```
class Thing {private: static inline intn_
things=0; // 全局计数 intmynumber; // 我是谁?
```

9.5.2.1 静态方法

如果你想要查询上述静态变量，当然可以从任何特定的对象中查询它们，因为它们都具有相同的值。然而，你可以定义一个静态方法：

```
类 Thing{public: 静态 intnumber_ 的 _ 事物 () { 返回 n_ 事物 ;
};
```

并且这个方法可以在类本身上调用；

```
cout <<“事物数量： ”<< 类 ::number_ 的 _ 事物 () << <style id='22’ >’ \n’ ;
```

9.5.2.2 Legacy syntax for initialization

在 C 之前 ++17，初始化静态变量是以一种奇怪的方式进行的。目前，通过添加关键字内联，您可以编写：

代码：

```
1// /static.cpp2
类 事物 {
3 私有:
4    静态内联 intnumber{0};
5    int mynumber;
6 public :
7    Thing () {
8       mynumber =number++
9       cout<< "I am thing "
10            << mynumber << '\n';
11   };
12 };
```

输出
[对象] 静态：

```
我是东西 0
我是东西 1
我是东西 2
```

如果你在旧代码中遇到它，这里有静态类成员的 C++11 语法：

```
1 // /static.cpp 2 class myclass { 3 private: 4
static int count; 5 public: 6 myclass() {++count; };
7 int create_count() { return count; }; 8 }; 9 /* ...
*/ 10// 在主程序中 11int myclass::count=0;
```

9.5.3 类签名

为了组织你的代码，有时你可能不希望包含方法的完整代码，例如在头文件中。这是 声明 和 定义 之间的区别。

你已经看到了函数的这种情况：

```
float f(int);
```

是函数的 声明，声明了函数的名称、输入参数的类型和返回结果的类型。（这有时也被称为函数的 ‘签名’ 或 ‘原型’ 或 ‘头’。）另一方面

```
float f(int n) { return sqrt(n); }
```

是函数的 定义，给出了它的完整代码。

类似地，你可以编写类声明，仅给出类的数据成员和类方法的签名，并在稍后或 elsewhere 指定完整的方法。这例如发生在你将程序拆分到多个文件中时；参见章节 19 和特别地节 19.3。

Declaration:

```
class Point {
private:
  float x,y;
public:
  Point(float x,float y);
  float distance();
};
```

Definition:

```
Point::Point()
  : x(x),y(y) {};
float Point::distance() {
  return sqrt( x*x + y*y );
};
```

- Methods, including constructors, are only given by their function header in the class definition.
- Methods and constructors are then given their full definition elsewhere with ‘classname-double-colon-methodname’ as their name.
- (qualifiers like `const` are given in both places.)

9.5.4 通过引用返回

在进行本节之前，请确保您已学习 *15.3* 节。

在所有关于 API 抽象内部细节的讨论中，有时您确实需要直接访问对象的内部细节。最简单的解决方案是返回一个副本：

```
类Foo{private:int
x;public:intthe_x() {
return x; };};
```

这种方法有两个问题：

- 返回副本如果内部数据是一个大对象可能会很昂贵；并且
- 也许您实际上想要修改内部数据。

所以我们有两种情况：

- 你想获取一个数据成员的引用，以便修改它；
- 你想获取一个数据成员的引用，因为复制成本很高，但你不会修改它。

First we show 返回私有引用的一般机制 ate data.

返回一个私有成员的引用：

```
1class Point { 2 private: 3 double x,y; 4
public: 5 double &x_ 组件 () { return x; }; 6 };
7int main() { 8 Point v; 9 v.x_ 组件 () = 3.1; 10 }
```

Only define this if you need to be able to alter the internal entity.

接下来我们展示一个可以被视为性能优化的机制：我们返回私有数据的引用，但以一种方式返回，使得调用方不能修改它。

返回引用可以节省复制。通过使用 ‘const 引用’ 来防止不希望的变化 .

```
1class Grid { 2 private: 3 vector<Point>
thepoints; 4 public: 5 const vector<Point> &points()
const { 6 return thepoints; }; 7 }; 8int main() { 9
Grid grid; 10 cout<< grid.points()[0]; 11 //
grid.points()[0] =whatever ILLEGAL 12 }
```

9.5.5 访问器函数

将对象中的数据设为私有是一个好主意，这样你可以控制谁有权限访问它。

- 有时这些私有数据是辅助性的，没有必要让外部访问。
- 有时你确实想让外部访问，但你想精确控制访问方式。

访问器函数：

```
类 事物 { 私有: 浮点 x; 公共: 浮点 获取
_x() { 返回 x; };void 设置_x(浮
点 v) { x= v; }};
```

这有以下优点：

- 您可以在获取 / 设置值时随时打印出来；非常适合调试：

```
void设置_x(float v) {cout <<"设置:"
<< v << endl;x= v; }
```

- 你可以捕获特定值：如果 x 总是应该是正数，如果非正数，则打印错误（抛出异常）。

有两个访问器可能会有些笨拙。是否可以使用同一个访问器来获取和设置?

使用单个访问器来获取和设置:

代码:

```
// /accessref.cpp
类 SomeObject {
3 私有:
   float   x=0.;
public:
   SomeObject( float v ) : x(v) {};
   float &xvalue() { return x; };
};

int main() {
   SomeObject myobject(1.);
   cout <<" 对象成员初始值: "
        <<对象 .xvalue() << '\n';
   我的对象 .xvalue()  = 3.;
   cout <<" 对象成员 u        pdated   :"
        << myobject.xvalue() << '\n';
```

输出

```
[对象 ]  访问引用:

对象成员初始: 1
对象成员已更新                    :3
```

函数 xvalue 返回内部变量 x 的引用。

当然，你只有在想直接改变内部变量时才应该这样做!

9.5.6 多态

你可以有多个同名的方法，只要它们可以通过参数类型来区分。这被称为多态；参见第 7.7.3 节。

9.5.7 运算符重载

而不是写

myobject. plus(anotherobject)

你实际上可以重新定义 + 运算符，以便

```
myobject + anotherobject
```

这是合法的。这被称为 运算符重载：你为常见的算术运算符提供自己的定义。

语法:

<返回类型> 运算符 <op>(<参数>){<定义>}

例如:

代码:

```
// /pointscale.cpp
Point Point::operator*(double f) {
    返回 点(f*x,f*y);
};
    /* ... */
  cout <<"p1 到原点"
       <<p1。距离_到_原点() << '\n';
  点比例 2r= p1*2.;
  cout <<"缩放右侧:"
       <<scale2r。距离_到_原点() <<'
    '\n';// 非法点 scale2l = 2.*p1;
```

输出
[几何] 点乘:

p1 到原点 2.23607
缩放右: 4.47214

也可以:

```
void Point::operator*=(double 因子);
```

练习 9.15. 编写一个分数类，并在其上定义算术运算符。

定义 + 和 += 运算符。你能用其中一个来定义另一个吗?

Exercise 9.16. If you know about templates, you can do the exercises in section 22.3.

9.5.7.1 函数对象

运算符重载的一种特殊情况是重载括号。这使得一个对象看起来像函数；我们称这种对象为函数对象。

简单示例:

代码:

```
// /functor.cppI tP i t
class n r n unctor {
public:
  void operator()(int x) {
    cout << x << '\n';
  }
};
    /* ... */
  IntPrintFunctor intprint;
  intprint(5);
```

Output
[对象] 函数:

5

练习 9.17。 扩展那个类，如下所示：不是直接打印参数，而是应该打印它乘以一个标量。这个标量应该在构造函数中设置。使以下代码工作:

Code:

```
1 // /functor.cpp
2   IntPrintTimes printx2(2);
3   printx2(1);
4   for ( auto i : {5,6,7,8} )
5     printx2(i);
```

Output

```
[object] functor2:

2
10
12
14
16
```

（*The for each is part of algorithm*）

9.5.7.2 列出可重载运算符

算术：`+ - * / %` 注意这仅限于现有运算符：新的运算符如 `**` 不能被创建。

增量 / 减量：`++ --`

位运算：`^ & | ~ && ||`

布尔否定：`!` 及其否定 运算符 bool

比较 :`= < >` 赋值 :`+= -= *= /= %= ^= &= |=` 流提取和插入 :

`<< >> >>= <<= == != <= >=` 飞船（自 C++20 起）：`<=>`

函数调用：`()`

数组下标 :`[ ]`

更多：`, ->* ->`

9.5.7.3 对象输出

如果你能 …… 那该多好啊

```
MyObject x;
cout << x << '\n';
```

之所以这行不通，是因为 ‘双小于’ 是一个二元运算符，而它没有用你的类作为第二个操作数来定义。

参见第 12.4 节以获取解决方案。

9.5.7.4 比较和‘飞船’运算符

在上一节 9.5.7 中我们讨论了运算符重载。特别是，你可以重载比较运算符 `<, =, >` 等等。这很快就会变得非常繁琐：有六个不同的运算符。

C++20 标准通过飞船运算符简化了这一点 .

// /compare.cpp bool operator<(const Point&two)const { if (vx!= two.vx) return vx<two.vx; else return vy<two.vy; } auto operator <=>(const Point& other) const =default;

9.5.8 构造函数和包含类

假设我们有一个类，其中每个对象包含另一个非平凡的类的对象。现在我们必须了解外部对象的创建如何与内部对象的创建相关联。

最后，如果一个类包含另一个类的对象，

```
class Inner {public:
Inner(int i) { /*... */ };
class Outer {private:
Inner 类 ;public:};
```

then

```
Outer( int n ) {
   包含 = 内部 (n);
};
```

1. 这首先调用默认构造函数
2. 然后调用 `Inner(n)` 构造函数，
3. 然后将结果复制到 `contained` 成员上。

```
Outer( int n )
   : contained (Inner (n)) { //
      * ... *
};
```

1. This creates the `Inner(n)` 对象，
2. placed it in the `contained` 成员，
3. 执行其他构造函数，如果有的话。

备注 9 成员初始化列表的顺序被忽略：指定的成员将按照它们被声明的顺序进行初始化。在某些情况下，这种区别很重要，所以最好将它们放在相同的顺序中。

9.5.9 ‘this’ 指针

在对象内部，一个指向对象的指针可用作 `this`：

```
类 Myclass {私有:
intmyint;公有:
Myclass(intmyint) {
```

```
6 this->myint_= myint; // ‘this’ 冗余！7   };8 };
```

你通常不需要使用 this 指针。示例：你需要在一个方法内部调用一个需要对象作为参数的函数)

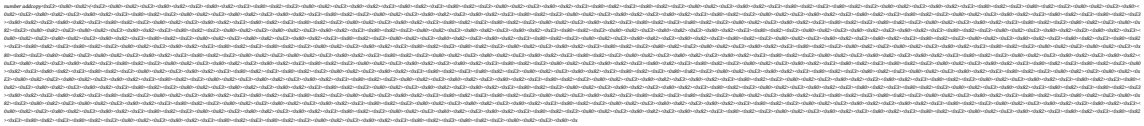
```
1 /* forward definition: */ class someclass;
2 void somefunction(const someclass &c) {
3   /* ... */ }
4 class someclass {
5 // method:
6 void somemethod() {
7   somefunction(*this);
8 };
```

(罕见的解引用星号使用)

还有一个有趣的惯用法使用了 ‘this’ 指针。定义一个简单的类

```
// /this.cppclass number
{private:: float x;public::
number(float x) : x(x) {}float
value() { return x; }};
```

定义方法



既修改了对象，又返回了一个副本。

将方法改为返回引用：

```
// /this.cppnumber& add(float y)
{x += y;return *this;};number&
multiply(float y) {x *= y;return
*this;};
```

具有一个有趣的效果，即 nocopy 被创建，但返回的 ‘对象’ 是对象本身，通过引用。这使得一个有趣的句式成为可能：

代码:

```
1 // /this.cpp
2   number mynumber(1.0);
3   mynumber.+.5
4   cout <<mynumber. value() <<  '\n';
5
6
7   cout <<mynumber value() <<  '\n';
      mynumber. 乘以 (2.). 加 (1.). 乘以 (
```

输出 [对象] this:

```
1.5
12
```

```
3.);
```

9.5.10 可变数据

假设你有一个类，并且你想通过 const-ref 返回一些复杂的数据成员:

```
类有 _ 东西 { 私有: 复杂的东西 ; 公共: const 复
杂 & 获取 _ 东西 () const{ 返回 东西 ;};};
```

为了让生活变得有趣，只有当需要时才应该构建复杂的东西。你可以尝试在 *get_ 东西* 方法中构建它:

```
private:optional<复杂> 东西 = {};public:const
复杂 &get_ 东西 () const {if (not 东西 .has_
value()) 东西 = 复杂 ( /* 当前内容 */ );return 东西 .
value();};};
```

问题是，现在的 *get_ 东西* 方法不再是 ‘const’ 。在这里，你需要意识到 **const** 意味着该例程只是外部常量: 如果内部数据被声明**可变**:

```
private: 可变optional<复杂 > 东西 = {};public:const 复
杂 &get_ 东西 () const /* 如上 */
```

代码:

```
1 // /mutable_c
  pp2 类有_东西 {

3 私有:
4   可变可选<复杂> 东西_
      {};
5 公共:
6   const复杂& 获取_事物() const          {
7     if (not 事物.有_值() )
8         事物=复杂(5);l t<< "thi l d
9     e se cou          ng a rea y there\n";
10    返回 事物.值();
11  };
12 };
```

输出
[对象] 可变:

```
制作复杂事物
事物已存在
事物已存在
```

9.5.11 懒惰求值

以下模式防止创建临时对象:

//axpy.cpp 模板< typename T>类 scaledvector { public: constT& scalar; constvector<T>& data; public: scaledvector(T x,const vector<T>& ar) :scalar(x),data(ar) {}; size_t size() const { return data.size(); }; };

```
template< typename T >
scaledvector<T> operator*( T x,const
    vector<T>& ar ) {
  return scaledvector<T>(x,ar);
};

template< typename T >
vector<T>& operator+=( vector<T>&
    y,const scaledvector<T>& ax ) {
  assert( y.size()==ax.size() );
  for ( size_t i=0; i<y.size(); ++i )
    y[i] += ax.scalar * ax.data[i];
  return y;
};
```

9.6 复习问题

Review9.2. 填充缺失的术语

- 类的功能由其 . . . 决定
- 对象的状态由其 . . . 决定

How many constructors do you need to specify in a class definition?

- 零
- 零个或多个
- 一个
- 一个或多个

Review9.3. 描述初始化对象成员的各种方法。

9. 类和对象

第 10 章

Arrays

一个 数组 [1] 是一种索引数据结构，它为每个索引存储一个整数、浮点数、字符、对象等。在科学应用中，数组通常对应于向量和矩阵，可能非常大。（如果你了解有限元方法（ FEM ），你就知道向量可以有数百万甚至更大的大小。）

在本章中你将看到 C++ 向量的 创建，它实现了事物的数组概念，无论是数字、字符串还是对象。

> *C 差异*： 虽然 C++ 可以使用 C 的数组机制，但对于几乎所有目的，使用 向量 会更好。特别是，这是一种更安全的方式来执行动态分配。旧机制在 10.10。 章节中简要讨论。

10.1 一些简单的例子

10.1.1 向量创建

要使用向量，您首先需要向量 头文件来自 STL。这允许您声明一个向量，指定它包含什么类型的元素。接下来，您可能想要决定它包含多少个元素；您可以在声明向量时指定这一点，或者稍后动态确定。

我们从创建向量的最明显方式开始：枚举其元素。

短向量可以通过枚举其元素来创建：

```
1// /shortvector.cpp 2#include <vector> 3using
std::vector; 4 5int main() { 6 vector<int>
evens{0,2,4,6,8}; 7 vector<float> halves ={0.5, 1.5,
2.5}; 8 auto halfloats= {0.5f, 1.5f, 2.5f}; 9 cout <<
evens.at(0) <<’ \n’ ; 10 return 0; 11 }
```

1. 这里 ‘array’ 一词使用得比较非正式。有一个 `array` 关键字，它在 10.4 节中简要讨论。 .

练习 10.1。

1. 取上面的代码片段，编译，运行。
2. 添加一条修改向量元素值的语句。检查它是否做了你预期的事情。
3. 添加一个与 偶数 向量长度相同的向量，其中包含奇数，这些奇数是偶数值加 1?

你可以参考代码仓库中的 *shortvector.cpp* 文件。

对于一个不太简单的例子，让我们创建一个包含对象的向量，在这种情况下是 `Point` 对象：

```
vector<Point> diagonal =
    { {0.,0.}, {1.,1.}, {1.5,1.5}, {2.,2.}, {3.,3.} };
```

10.1.2 初始化

There are various ways to declare a vector, and possibly initialize it.

更一般地，向量可以定义

- 如果没有进一步指定，创建一个空向量：`vector<float> 一些_数字;`
- 指定大小后，分配该数量的元素：`vector<float> 五个_数字(5);`(这会将元素设置为默认值；数值类型为零。)
- 您可以用常量初始化向量：`vector<float> x(25,3.15);`

  它定义了一个大小为 25 的向量 `x`，并将所有元素初始化为 3.15。

如果你的向量足够短，你可以使用一个 初始化列表 显式地设置所有元素，就像上面看到的那样。请注意，这里没有指定大小，而是根据初始化列表的长度推断出来：

代码：

```
// /dynamicinit.cpp
{
    或<int> 数字{5,6,7,8,9,10};
  vect
  cout<< 数字。at(3)<< '\n';

}
{
  vector<int> 数字=
  {5,6,7,8,9,10};
  numbers.<at>(3) t;bt(3
  cou  << num ers.a   ) << '\n';
}
```

输出 [数组] 动态初始化：

```
8
21
```

复习 10.1. T/F?

- 可以编写有效的 C++ 程序，其中定义了一个变量 *vec tor*.

10.1.3 元素访问

访问向量元素有两种方法。

1. 使用您从结构和对象中熟悉的 “ 点 ” 表示法，您可以使用 `at` 方法：

代码：

```
// /assign.cpp
    v矢量<int>  数字={1,4};
    数字.at(0)  += 3;b
    t(1)8ers.a     =  ;
    cout << numbers.at(0) << ","
         << numbers.at(1) << '\n';
```

输出
[数组] 分配函数

```
4,8
```

可以使用表达式 `a.at(i)` 来获取向量元素的值，或者它可以在赋值的左侧出现以设置值。

2. 还有一种简写符号（与 C 中的相同）：

代码：

```
// /assign.cpp
    vector<int> numbers = {1,4};
    numbers[0] += 3;
    numbers[1] = 8;
    cout << numbers[0] <<","
         << 数字[1] <<'\n';
```

输出
[array] 赋值括号：

```
4,8
```

同样，访问的元素可以在左侧和右侧使用 .

索引从零开始。因此，声明为

```
vector<int> ints(N)
```

具有元素 0，...， $N-1$。

10.1.4 越界访问

你是否想过如果你访问了向量边界之外的元素会发生什么？

vector<float>x(6); // size 6, index ranges 0..5)x.at(6) = 5; // oops!i -2;x i; // also oops, but different.

通常编译器很难确定你是否访问了向量边界之外的元素。很可能，它只能在运行时检测到。现在，两种访问方法在 *vector bounds checking* 方面有所不同。

1. 使用 `at` 方法会始终进行边界测试，如果你访问了向量边界之外的元素，程序会立即退出。（技术上，它抛出一个异常；参见第 23.2.2 节了解其工作原理以及如何处理这种情况。）

2. 括号表示法 `a[i]` 不进行边界测试：它根据向量位置和索引计算内存地址，并尝试返回那里存储的内容。正如你所想象，这种缺乏检查使你的代码稍微快一点。然而，它也使你的代码不安全：

- 您的程序可能会因段错误 或 总线错误而崩溃，但没有明确的指示说明问题出在哪里以及为什么发生了这种情况。（这种崩溃可能是由其他原因引起的，而不仅仅是向量越界访问。）

- 您的程序可能会继续运行，但会给出错误的结果，因为从向量外部读取可能会给您提供无意义的数据。向量的边界外写入甚至可能改变其他变量的数据，导致非常奇怪的错误。

目前，最好在整个程序中使用 `at` 方法。

索引越界可能在一段时间内未被检测到：

代码：

```
// /segmentation.cpp vector<int> v(10,2);
vect
for (int i=5; i<6; i--) "lt"
  cout <<  e emen    << i
       << " is " << v[i] << '\n';
```

输出

```
[array] segmentation:

元素       -5869 是 0
元素       -5870 是 2.8026e-45
元素       -5871 是 2.38221e-43
元素 -5872 是 1.00893e-41
元素 -5873 是 0
元素       -5874 是 0
元素 -5875 是 0
元素 -5876 是 0
/bin/sh: line1: 48082
    段错误 : 11
    （ 核心转储 ） ./ 段错误
```

评论 10.2. 以下代码在某些方面是不正确的。这将如何表现出来？

```
// /vecerr.cppvector<
int> a(5);a[6] = 1.;
cout <<"成功 \n";
```

```
// /vecexc.cppvector<i
nt> a<(5);a<./at>(6)  = 1.;
```

10.2 遍历所有向量元素

如果你需要考虑向量中的所有元素，你通常使用 `for` 循环。有各种方法可以做到这一点。

概念上，一个 `vector` 可以对应于一组事物，而它们被索引的事实纯粹是偶然的，或者它可以对应于一组有序的事物，而索引是必不可少的。如果你的算法要求你访问所有元素，重要的是要考虑哪种情况适用，因为有两种不同的机制。

10.2.1 遍历向量

首先考虑将向量视为元素集合的情况，并且循环函数类似于数学中的 “ 对于所有 ”。这使用 “ 冒号 ” 符号。

A *range-based for* loop gives you directly the element values:

```
vector<float> my_data(N);/* 以某
种方式设置元素 */;for ( float e :
my_data )// 关于元素 e 的语句
```

这里没有索引，因为你不需要它们。

你可以指定向量元素的类型，但这样的类型指定可能会很复杂。在这种情况下，通过 *auto* 关键字进行类型推断**非常方便**。

Same with `auto` instead of an explicit type for the elements:

```
for (auto e:my_data)
  // 相同，编译器会自动推断类型
```

例如，考虑在一个数字数组中找到最大值。由于我们只需要值，而不需要知道它的位置，因此我们使用这种基于范围的语法。（注意：实际上有一个库函数可以完成这个任务，所以这个例子主要是为了讨论基于范围的机制。）

Finding the maximum element

代码：

```
1// /dynamicmax.cpp2 vec
2     tor<int>numbers = {1,4,2,6,5}
3   int tmp_max= -2000000000;i(t,b)
4    or  au o v : num ers
5     if (v tmp max)
6       tmp_max = v;
7   cout << "Max: " << tmp_max
8        <<"（应为 6 ） "<< '\n';
```

Output
[动态数组] 最大：

Max： 6 （ 应该为 6）

另一个注意：将初始化为较大的负值可以更优雅地完成；参见第 24.2 节。

So-called *initializer list*s can also be used as a list denotation:

代码：

```
1 / /rangedenote.cpp
2   for (auto i : {2,3,5,7,9})
3     cout << i<< ",";
4   cout << '\n';
```

输出
[数组] 范围表示：

2,3,5,7,9,

10.2.2 Rangingover theindices

如果您确实需要元素的索引，可以使用传统的 `for` 循环，并使用循环变量。

您可以编写一个索引的 循环，该循环使用一个索引变量，该变量从第一个元素到最后一个元素进行范围遍历。

```
for (int i= /* 从第一个索引到最后一个索引 */ ) // 关于索
引 i 的语句
```

示例：在向量中找到最大元素，并确定其位置。

代码：

```
1 // /vectoridxmax.cpp
2   int tmp_idx = 0;
3   int tmp_max = numbers.at(tmp_idx);
4   for (int i=0; i<numbers.size(); ++i) {
5     int v= numbers. at (i);
6     if (v>tmp_max) {
7       tmp_max= v; tmp_idx= i;
8     }
9   }
10  cout << "Max: " << tmp_max
11       << " at index"<<tmp_idx<<' \n' ;
```

输出

```
[数组] vecidxmax:
最大值位于索引 : 3
```

Exercise10.2. 请针对以下向量操作，分别说明您更倾向于使用索引循环还是范围循环，并给出简短的理由。

- 统计向量中零元素的数量。
- 找到最后一个零的位置。

Exercise 10.3. 在向量中找到绝对值最大的元素。使用 e:

```
vector<int> numbers = {1,-4,2,-6,5};
```

提示：

```
#include <cmath>><..>= <abs{x}>{v12}<abs>{v14}({v16});{v18}
```

练习 10.4. 在向量中找到第一个负元素的位置。

你使用哪种机制?

练习 10.5. 检查向量是否已排序。

备注 10 在 *C++20* 中，您可以通过一个 初始化语句 在基于范围的循环中为循环索引指定一个范围：

Code:

```
// /enumerate.cpp
vector<int> values{2,4,5,7,10};
for ( size_t i=0; auto v : values )
  cout << "Element " << i++
       << ": " << v << '\n';
```

Output
[range] enumerate:

```
Element 0: 2
Element 1: 4
Element 2: 5
Element 3: 7
Element 4: 10
```

注意，你必须在循环体内显式地递增循环变量。

10.2.3 通过引用遍历

在之前的示例中，我们只读取了向量元素的值。如果你想要访问元素值以便修改它们，例如向它们添加一些内容，或乘以它们，该怎么办?

基于范围的循环可以实现这一点。你需要意识到，到目前为止的循环中

for (auto e:my_array)//
something with e

变量 `e` 实际上包含向量的元素副本。这意味着修改 `e` 不会影响向量。要实现这种效果，你需要让 `e` 成为指向向量元素的引用。

```
for ( auto &e : my_vector)
  e = ....
```

示例：将所有元素乘以二：

代码:

```
1// /vectorrangeref.cpp2ector<float>m
ecto v                yv     r
3     ={1.1, 2.2, 3.3};
4   for (auto &e : myvector) e * 2;
5         =
6   cout << myvector.at(2) << '\n';
```

输出
[数组] 向量范围引用:

```
6.6
```

练习 10.6。 如果你做素数项目，你现在可以做练习 45.1 8.

10.2.4 数组和 while 循环

在一个 `while` 循环中，如果你需要索引，你需要显式地维护该索引。然后有一些常见的惯用法。

代码:

```
// /plusplus.cpp
    I<int> 数字{3,5,7,8,9,11};
    vecto
    int 索引{0};
    while( 数字[索引++]%2==1 ) ;
    cout<< "第一个偶数\n"
        << "appears at in" ex
        <<  索引<<'\n';
```

输出
[循环] 自增

第一个偶数
出现在索引 4

练习 10.7. 练习：修改前面的代码，使得在 while 循环之后，索引是前导奇数元素的个数。

10.3 向量面积类

之前，你创建了向量并使用了函数 在 和 大小 在它们上。它们使用了类方法的点表示法，实际上向量形成一个 向量 类。你可以有一个整数的向量、浮点数的向量、双精度浮点数的向量等等；尖括号表示法指示向量中存储的具体类型。你可以认为向量类是按类型参数化的（见第 22 章的详细信息）。你也可以说向量 <int> 是一种新的数据类型，发音为 “ 整数向量 ”，你可以创建该类型的变量。

向量和其他数据类型一样可以被复制:

代码:

```
// /vectorcopy.pp
  vector<float> v(5,0),vcopy;
  v.at(2) = 3.5;
  vcopy= v;
  vcopy.at(2)*= 2;
  cout <<v. at(2),
       << vcopy.at(2) << '\n';
```

输出
[数组] 向量复制:

3.5,7

10.3.1 向量方法

有几种方法 用于 向量 类。其中一些更简单的方法是:

- *at*: 索引元素
- *size*: 获取向量的大小
- *front*: 第一个元素的值
- *back*: 最后一个元素的值

还有一些与动态存储管理相关的方法，我们接下来会讲到。

练习 10.8. 创建一个 向量 x 的 **float** 元素，并将它们设置为随机值。（目前使用 C 随机数生成器。）

现在在 L_2 范数中归一化向量，并检查你的计算的正确性，即，

1. 计算 L_2 范数：

$$\|v\| \equiv \sqrt{\sum_i v_i^2}$$

2. 将每个元素除以该范数；3. 缩放后向量的范数现在应为 1。检查一下。4. 奖励：你的程序可能输出 1，但它真的是 1 吗？调查一下。

你在使用什么类型的循环？

10.3.2 向量是动态的

向量可以在创建后增长或缩小。例如，你可以使用 `push_back` 方法在末尾添加元素。

使用 `push_back` 扩展向量的大小：

代码：

```
// /vectorend.cpp
    vector<int> mydata(5 2);

    y(.mydata.push_back(25);

    cout<< mydata. size()
        << '\n';
    cout<< mydata.().back
        << '\n';
```

Output

```
[ array  vectorend:

6
35
```

Similar functions: `pop_back`, `insert`, `erase`. Flexibility comes with a price.

使用 `push_back` 来动态创建向量的想法很有诱惑力。

Known vector size:

```
int n = get_inputsize(); vector<
float> data(n); for ( int i=0; i<n;i
++ ) { auto x= get_item(i);
data.at(i) =x; }
```

未知向量大小：

```
vector<float> data;
float x;
while ( next_item(x) ) {
  data.push_back(x);
}
```

如果你对大小有猜测：`data.reserve(n)`。

(对象数组问题：在左侧代码中，构造函数被调用了两次。)

```
向量<int> iarray;
```

创建一个大小为零的向量。然后你可以

```
iarray. push_ back(5); iarray.
push_ back(32); iarray. push_
back(4);
```

然而，这种动态调整大小涉及内存管理，并可能需要操作系统功能。这可能会很低效。因此，您只有在绝对必要时才应使用此类动态机制。如果您知道大小，请创建一个具有该大小的向量。如果大小无法精确确定，但您有一个合理的上限，您可以调用 `reserve` 来为该数量的元素预留空间：

```
vector<int> iarray;
iarray.reserve(100);
while ( ... )
  iarray.push_back( ... );
```

使用 `reserve` 和 `push_back` 的组合可能比立即用特定大小创建向量更可取。编写 `vector<X> xs(100)`，其中 `X` 是某个对象，会导致在向量每个元素上调用 `X` 的默认构造函数。对于复杂的对象，这可能并不明智。

10.4 数组类

在数组大小永远不会改变的情况下，将会有一个方便的 `vector` 类变体，它没有动态内存管理功能。 `array` 类似乎在第一眼看起来能满足这个角色。然而，它仅限于在编译时已知大小的数组，所以你不能例如将其作为参数读取。

```
// /stdarray.cpp#
include <array>
using std::array>;
```

数组对象使用静态大小声明：

```
array<float,3> coordinate;// /
stdarray.cpp{array<float,5> v5;cout << "
size: "<< v5.size() << '\n';// 错误：没有这样
的函数 // v5.push_back(2);}
```

10.4.1 初始化

There are se 多种方式初始化一个 std::array。最字面意义上的 ded way is

```
数组 <int,3> i3 = {1,2,3};//
或数组 <int,3> i3{ {1,2,3} };
```

但自 *C++17* 起，聚合初始化是允许的：

```
array<int,3> i3{1,2,3};
```

如果你觉得数组的大小在初始化中是冗余的，你可以使用 C++17 模板参数推导：

```
array i3 = {1,2,3};
```

这需要你对类型小心一些：

```
// 不编译：array
not4{1.5,2,3,4};
```

10.4.2 传递为参数

将 初始化列表 传递给一个有 std:: 数组 参数的函数是可行的 :

```
// /toarray.cppfloat first_ofthree_(array<float{v18},fff3>) {return fff
[0];};/* ... */cout first_ofthree_( { 1.5 , 2.5 , 3.5 } ) << ' \n' {v50};
```

然而，在某些情况下它不适用：

```
// /toarray.cpp模板<unsigned long intd>float first_of_bunch(
array<float,d>fff ) {return fff[0];};/* ...*/// DOES
NOTCOMPILE:// cout <<first_of_bunch( { 1.5f , 2.5f , 3.5f } )
<< ' \n' ; // 无法推导模板参数 ' d'
```

为此，我们可以使用 C++20 中的 to_array：

```
// /toarray.cpp // 模板参数 'd' 可以从 'to_array' cout <<first_of_bunch( to_
array( { 1.5f , 2.5f , 3.5f } ) ) << '\n';
```

10.5 向量和函数

向量像任何其他数据类型一样，所以它们可以与函数一起使用：你可以将向量作为参数传递，或者将其作为返回类型。我们将在本节中探讨这一点。

10.5.1 将向量传递给函数

参数传递的机制（第 7.5 节）也适用于向量：它们可以通过值传递和引用传递。

首先，有按值传递；第 7.5.1 节。在这里，向量参数被复制到函数中；函数接收向量的完整副本，并且在函数中对该向量所做的任何更改都不会影响调用环境。

C 差异： 在 C++ 向量和 C 数组之间存在很大差异！在 C 中，数组不会被复制：你通过值传递地址。不是内容。

代码：

```
1// /vectorpassnot.cpp
2voidset0
3   ( vector<float> v,</float>x )
4 {
5   v.at(0) = x;
6 }
7     /* ... */
8   vect或<浮点数> v(1) ;
9   v.at(0) = 3.5;
10  set0(v,4.6);
11  cout <<v. at(0) << ’ \n’ ;
```

输出

```
[数组] 向量传递不：
3.5
```

- 向量被复制
- 调用环境中的 ‘原始’ 值不受影响
- 复制成本如何?

练习 10.9。 回顾练习 10.8 并引入一个用于计算 L_2 范数的函数。

接下来是按引用传递；章节 7.5.2。在这里，参数向量成为调用环境中向量的别名，因此函数中对向量的修改会影响调用环境中的参数向量。

代码：

```
1 // /vectorpassref.cpp
2voidset0
3   ( vector<float> &v, floatx )
4 {
5   v.at(0) {x};
6 }
7     /* ... */
8   vect或<float> v(1);
9   v.at(0) = 3.5;
10  set0(v,4.6);
11  cout << v. at(0) <<’ \n’ ;
```

输出

```
[数组] 向量按引用传递：
4.6
```

想要通过引用传递的一个重要原因是，它可以避免通过值传递时可能存在的较大复制开销。那么，如果你想要这种效率，但又希望防止对参数向量进行无意中的修改呢？为此，你可以将函数参数声明为 “常量引用”。

传递不需要被修改的向量：

```
int f(const vector<int> & ivec ) { ... }
```

- 零拷贝成本
- 不可修改，所以：安全!

- (No need for pointers!)

参数传递的一般原则是

- 如果参数不被修改，则按值传递；
- 如果参数被修改，则按引用传递。

对于向量，这个问题又增加了一个维度：按值传递意味着复制，这对于向量来说可能是昂贵的。这里的解决方法是按 *const* 引用 传递，这既防止了复制，也防止了意外修改；参见第 18.2 节。

10.5.2 向量作为函数返回值

你可以将向量作为函数的返回类型。示例：此函数创建一个向量，并将第一个元素设置为大小：

Code:

```
// /vectorreturn.cpp
vector<int> make_vector(int n) {
  vector<int> x(n);
  x.at(0) = n;
  return x;
}
   /* ... */
  vector<int> x1 = make_vector(10);
  // "auto" also possible!
  cout << "x1 size: " << x1.size() <<
    '\n';
  cout << "zero element check: " <<
    x1.at(0) << '\n';
```

Output

```
[array] vectorreturn:

x1 size: 10
zero element check: 10
```

练习 10.10。 编写一个具有一个 `int` 参数 n 的函数，该函数返回长度为 n 的向量，并且其中包含前 n 个平方数。

练习 10.11。 编写 random_vector 和 sort 函数，以使以下 main 程序正常工作：

```
int length = 10; vector< float values > random = vector_
vector (length); vector< float sorted > sort (values);
```

这将创建一个指定长度的随机值向量，然后对其进行排序副本。

与其创建排序副本，不如原地排序（用排序数据覆盖原始数据）：

```
int length = 10; vector< float values> random_ vector=
values= random_ vector_ (length = 10); sort{v28} (values> );
// the vector is now sorted
```

找出支持 / 反对该方法的论据。

(注意：C++ 内置了排序函数。)

(参见第 24.7.5 节，以了解随机函数。)

练习 10.12。 编写代码，对一个整数向量进行处理，并构造两个向量，一个包含所有奇数元素，另一个包含所有偶数元素。因此：

输入：
5,6,2,4,5 输出：
5,5 6,2,4

你能写一个函数，接受一个向量，并产生两个向量吗?

10.6 类中的向量

你可能需要一个包含向量的类对象。例如，你可能想给你的向量命名。

```
类 命名 _ 字段 { 私有: 向量 <
双精度 > 值 ; 字符串 名称 ;
```

这里的问题在于何时以及如何创建那个向量。

- 如果向量的尺寸是静态确定的，当然可以用那个尺寸声明它：

```
类 命名 _ 字段 { 私有: 向量 < 双精度
浮点数 > 值 (25); 字符串 名称 ;
```

- ... 但在更有趣的情况下，大小在程序的运行时确定。在这种情况下，你需要声明：

```
named_field velocity_field(25,"velocity");
```

在对象的构造函数中指定大小。

那么现在的问题是，如何在对象构造函数中分配这个向量?

一个解决方案是在类定义中指定一个没有大小的向量，在构造函数中创建一个向量，并将其分配给对象的向量成员：

```
named_field( int n ) {
  values = vector<int>(n);
};
```

然而，这会产生以下效果：

- 构造函数首先创建值 作为零大小向量，
- 然后它创建一个大小为 n 的匿名向量，
- 并通过赋值，销毁了先前创建的零大小向量。

这有点低效，最佳解决方案是作为 成员初始化器 列表的一部分来创建向量：

使用初始化器来创建包含的向量：

```
1 类 命名 _ 字段 {
2 private:
3    字符串名称 ;4    向量 < 双精度浮点数 > 值 ;
5 public:
    6 _ 字段（字符串 name，int n ） 7:
  name （ name ），8 values （ vector<
  double> （ n ））； 10 ;
```

Even shorter:

```
1 _ 字段（字符串 name，int n ） 2： na
me （ name ），values （ n ） {3} ;
```

10.6.1 Timing

不同的向量访问方式可能具有截然不同的时间成本。

您可以向向量中推入元素：

```
// /arraytime.cppvector<int> flex;{/* ...
*/}for (int i=0; i<LENGTH; i++)
i{v34})flex.push_back{(i{v51});}
```

如果你静态分配向量，可以用 at：

```
// /arraytime.cppvector<int>
stat{v9}(stat){v11}(LENGTH){v13};/* ...
*/for(int =0;i<LENGTH;i++)stat.{at}(i)=
i;
```

带下标：

```
// /arraytime.cppvector<int> stat<stat>(</stat><stat>LENGTH</stat><stat>);</stat><stat>/* ... */</stat><stat>for</stat><stat>(</stat><stat>int
</stat><stat>i</stat>=0<stat>;</stat><stat>i</stat><<stat>LENGTH</stat><stat>;</stat>++<stat>i</stat><stat>)</stat><stat>stat</stat>[
<stat>i</stat>] =<stat>i</stat><stat>;</stat>
```

你也可以用 `new` 来分配 *：

```
// /arraytime.cppint *stat= new int[
LENGTH];
```

```
        /* ... */
        for (int i=0; i<LENGTH; ++i)
          stat[i] = i;
```

* 被认为是不好的实践。不要使用。

对于新内容，请参见第 17.6.2 节。但是请注意，这种分配模式基本上永远不会需要。

Timings are partly predictable, partly surprising:

```
灵活时间： 2.445 静态在时间： 1.177
静态分配时间： 0.334 静态分配时间到新：
0.467
```

The increased time for`new` 是一个谜。

那么你使用 `at` 是为了安全还是为了`[]` 的速度？嗯，你可以在开发代码时使用 `at`，然后插入

```
#define at(x) operator[](x)
```

用于生产。

10.7 将向量封装在对象中

你可能想要创建包含向量的对象。作为一个简单的例子，考虑 ‘命名向量’，我们将其实现为一个包含向量和字符串的类。

```
// /arrayprint.cpp 类 namedvector { private:
string name; vector<int> values; public:
namedvector(int n,string name="unnamed") :
name(name),values(n) { }; string 渲染 () {
stringstream render; render <<name <<":"; for (auto
v : values) render << " " << v<< ","; return
render.str(); } /* ... */ };
```

这种方法的一个问题是，你可能需要重新创建一些方法来访问向量。例如，你需要定义对象上的 `at` 方法来访问向量的元素：

```
// /arrayprint.cppint&
at(int i) {return
values.at(i);};
```

一种摆脱这种重新定义需求的方法是让类从容器类继承：

```
// /isavector.cppclass witharray : public vector<
float> {
```

现在构造函数调用向量的构造函数：

```
// /isavector.cppwitharray(
float n ) : vector<float>(n) { };
```

但是之后对象就拥有了向量的所有方法：

代码：

```
1// /isavector.cpp2
witharray x<5>
3 x[x.size()-1 ] = 3.14;
4 cout <<x.<back>() << ’ \n’ ;
```

输出 [对象] 是一个向量：

```
3.14
```

10.8 多维情况

C++ 几乎没有对多维数组的原生支持，而多维数组对于线性代数运算和许多其他物理算法是必不可少的。在 63.4 节中，我们将探讨 *Eigen* 库，但目前我们将查看一些可用于在 C++ 中处理多维对象的机制。

10.8.1 矩阵作为向量的向量

模拟多维结构的一种方法是创建一个向量的向量 s:

```
向量<float> 行(20); 向量<向量<float
>> 行(10, 行);
```

在这里，您首先创建一个代表矩阵行的向量，然后用一些副本填充第二个向量。

注意 11 您也可以这样编写这个片段：

```
向量<向量<浮点数>> 行(10);
for( auto &行:行){行= 向量<浮
点数>(20); }
```

这种公式可以实现一些特殊效果。您能想到如何创建一个三角数组吗？

这不是矩阵的最佳实现，例如因为元素不是连续的。但是，让我们暂时继续使用它并编写一个矩阵类。我们首先需要的是元素访问。

```
1// /matrix.cpp 2 类 matrix { 3 private: 4 vector<vector<double>>
elements; 5 public: 6 matrix(int m,int n) 7 : elements( 8 vector<vector<
double>>(m,vector<double>(n)) 9 ) { 10 } 11 void set(int i,int j,double v) {
12 elements.at(i).at(j) =v; 13 }; 14 double get(int i,int j) { 15 return
elements.at(i).at(j); 16 };
```

能否结合 *get/set* 方法，使用 **??**?

练习 10.13. 为这个类编写 *rows()* 和 *cols()* 方法，分别返回行数和列数。

练习 10.14. 编写一个方法 **void** *set*(**double**) 将所有矩阵元素设置为相同的值。

编写一个方法 **double** *totalsum()* 返回所有元素的总和。

代码:

```
1// /matrix.cpp2 A.set(3.);3 cout "<<""
Sum of elements: ""4          <<"A."
totalsum()"\n';
```

```
Output
[array] matrixsum:
元素之和： 30
```

您可以参考代码仓库中的 *matrix.cpp* 文件

有了这些简单的访问方法，我们可以开始实现一些线性代数运算。

练习 10.15. 为您的矩阵类添加转置、缩放等方法。

Implement matrix-matrix multiplication.

10.8.2 一个更好的矩阵类

非连续矩阵行的问题可以通过创建一个矩阵类来解决，其中对象存储一个足够长的向量:

```
// /matrixclass.cpp类
matrix {private: 向量<
double> the_ 矩阵;int
m,n;public:matrix(int
m,intn)
```

```
: m(m),n(n),the_matrix(m*n) {}; void
set(int i,int j,double v) { the_matrix.at( i*n +
j ) = v; }; doubleget(int i,int j) { return the_
matrix.at( i*n +j ); }; /* ... */ };
```

练习 10.16. 在上一张幻灯片中的矩阵类中，为什么 `m,n` 被显式存储，而与上一节中的矩阵类不同?

这种设计最重要的优点是它与许多库和代码中传统使用的存储方式兼容。此外，它提高了操作的效率，但要理解这一点，你需要了解更多的计算机架构知识。

`set` 和 `get` 的语法可以改进。

练习 10.17. 编写一个类型为 double& 的方法元素，以便你可以编写

```
A.element(2,3) = 7.24;
```

备注 12 在 *C++23* 中，可以使得语法

```
A[2,3] = 7.24;
```

工作。参见第 **??** 节。

10.9 高级主题

10.9.1 循环索引类型

在索引循环中，您可能习惯于使用 `int` 作为循环变量的类型:

```
for ( int i=0; i<some_数组.大小(); ++i ) { /* 内容 */ }
```

这里存在问题：整数通常具有大约 20 亿的最大值（参见 24.2 节详细信息），而诸如 `vector` 之类的容器可以包含比这个更多的元素。

首先，在许多情况下，您可以通过使用 基于范围的循环 或标准算法来避免使用循环变量。如果您绝对需要该索引变量，请为其指定 `size_t` 类型:

```
for ( size_t i=0; i<some_array.size(); ++i ) { /* stuff */ }
```

注意 13 确实，某些编译器会在第一个循环类型上发出警告，但这与整数类型的选择无关。相反，警告将关联到这样一个事实，即在 `i<某些_array.size()` 中，您正在将一个有符号量与一个无符号量进行比较。这被认为是一种不安全的做法。

10.9.2 容器复制

使用 拷贝构造函数 对向量等容器进行操作会调用每个元素的拷贝构造函数。像 `float` 这样的类型被称为 ‘平凡可构造’ 或 ‘平凡可拷贝’，并且它们针对以下操作进行了优化：拷贝一个向量 < `int`> 是通过一个 *memcpy* 或等效机制完成的。

10.9.3 分配失败

如果你请求的数据量超出了系统支持的范围，分配可能会失败，并抛出一个错误 _ 分配异常。

这与 C 语言中的方法不同，C 语言会返回 *NULL* 或 `nullptr`。

10.9.4 栈和堆分配

存储在内存中的实体（变量、结构、对象）可以存在于两个位置：栈 和堆。

- 每次程序进入新的 作用域时，在该作用域中声明的实体会被放置在栈的顶部，并在作用域结束时被移除。由于这种自动行为，这被称为自动分配。
- 相比之下，动态分配 会创建一个在作用域结束时不会被移除的内存块，因此这个块会被放置在堆上。这个内存块可以在任何时候返回到自由存储中，因此堆可能会遭受碎片化。

10.9.4.1 C 语言中的示例

自动内存分配，或静态内存分配，使用作用域，就像它用于创建标量一样：

```
// 假设这里没有变量 i,f,str{  // 进入作用域 int i;float
f;char str[5];// 内容 }// 这里的名称 i,f,str 再次未知。
```

遵守作用域的对象在栈上分配，因此当控制离开作用域时，它们的内存会自动释放。另一方面，过度使用自动分配可能导致栈溢出。

动态内存分配是通过调用 *malloc* 完成的，通过将返回的内存地址赋值给一个在作用域外定义的变量，该块在作用域外可见：

```
double *array;{  // 进入作用域 array =
malloc(5*sizeof(double));// 离开作用域 }array
[4] = 1.5;  // 这是合法的 free(array);  // 释放
malloc 分配的内存
```

动态创建的对象，例如指针的目标，存在于堆上，因为它们的生存期不受作用域的限制。

第二类存在的存在是 内存泄漏 的来源，因为它很容易忘记 *free* 调用。

10.9.4.2 C++ 中的示例

首先，C++ 中存在 *malloc* 和 *free* 调用，也存在稍微更方便的变体 **new/delete**：

```
double *array= new[5];//
内容 delete array;
```

然而，C 的惯用方式 ++ 来创建具有动态确定内存的数组是通过使用 std::*vector*。

```
intn = .... ;{ // 进入作
用域 vector<int> 数组
(n);// 内容 } // 退出作用域
```

This combines the best features of C allocation:

1. vector 的存储空间在堆上创建，因此无需担心栈溢出； 2. 退出作用域时，vector 的定义消失，其动态内存被释放。这在技术上是称为 *RAII*。

然而，动态分配的内存可以超越其创建的作用域：

```
vector<int> f(intn) {returnvector<int>(n); // 在函数作用域内创建的
vector};vector<int> v;v =f(5);
```

这里发生的事情如下：

1. 在函数内部创建了一个向量；2. **返回** 语句将向量及其所有数据复制到调用环境中的变量 *v* ； 3. 但由于优化，复制被省略了，实际内存现在分配给了变量 *v*。

实际上，我们已经实现了上述 C 部分函数示例的一个更安全版本。

另一种不受作用域限制的动态内存选项是使用 智能指针 机制，它也保证了防止内存泄漏。见第 16 章。

10.9.5 Vectorof bool

布尔变量占用一个完整的字节，尽管布尔值严格只需要一个位。然而，你可以通过将位打包成一个整数来优化位数组，从而 向量 <**bool**>，节省了 8 倍的存储空间。

不幸的是，这种优化意味着你不能获取元素的引用。

```
vector<bool> bits;
for ( auto& b : bits ) // DOES NOT COMPILE
  b = false;
auto& f = bits.front(); // DOES NOT COMPILE
```

10.9.6 Span 和 mdspan

旧的 C 风格数组允许进行一些用向量难以完成操作。例如，你可以创建数组的子集，对其进行操作，并使原数组受到影响。这例如在 快速排序 算法中很有用：

```
    // 警告：这是伪代码 void
qs(data ) {if(data.size()>1)
{// 转换部分省略 qs(
data.lefthalf());qs(
data.lefthalf());}}
```

这是该机制工作的图示，使用显式分配的数据。在这个例子中， *subx* 是 *x* 从第二个元素开始的子集：

```
double *x= new double[N];double *subx=
x+1;subx[1] = 5.; // same as: x[2] = 5.;
```

您不能用标准 *vector* 或 数组获得相同的效果。在 C++ 中，可以使用 *vector* 从另一个 vector 使用 迭代器 语法创建；但是，这会分配新的存储并复制原始元素，而不是创建一个真正的子 vector:

代码：

```
1 // /subiter.cpp
2   vector<int> count(5);
3   iota(count.begin(),count.end(),0);
4   vector<int> from1
5     ( 计数。开始()+1, 计数。结束()-1);
6   cout<<count[1] << ","
7       <<from1[0] << '\n';
8   from1[0] = 5;
9   cout<<count[1] << ","
10      <<from1[0] << '\n';
```

输出
[数组] subiter:

```
1,1
1,5
```

如果你真的想两个类似向量的对象共享数据，那么有 span 类，在 C 中添加的 ++20。这允许你创建一个非拥有的对向量的视图。
一个 span 只是一个指针和一个大小，所以它允许上述用例。

Create a span from a *vector*:

```
#include <span></span><vector><double>>
<v>{v13}</v>;</double><auto>{v17}<v>{v19}</v>_
<span>{v22}</span><std>{v25}::<span>{v29}</span><double><</double>>
(<v>{v35}</v>.<data>{v41}</data> (), <v>{v45}</v>.<size>{v49}</size>());
```

The span 对象具有相同 数据，以及 大小 方法，以及下标运算符，作为向量。你也可以像下面看到的那样遍历它，但它没有动态方法，例如 *push_back*，或通过 *at* 方法进行边界检查的索引。

这里是一个 span 的使用示例：我们从一部分 向量创建一个 span，更改 span 中的一个元素，并看到原始向量中对应的元素也发生了变化：

代码：

```
1// /subspan.cpp
2     vector v{1,2,3};
3     span s{v.data()+1,2};
4     tail[0] = 0
5     cout << v[0] << ' , '
6          << v[1] << ' , '
7          << v[2] << '\n';
```

输出 [span] 子跨度 1:

```
1,0,3
```

注意，span 是按值传递的，但元素被修改了，这使得它看起来更像是按引用传递。你能解决这个看似的冲突吗?

前面的例子可以变得更优雅，消除显式的元素和大小指定，并使用范围和访问方法。对向量执行稍微不同的操作：

代码：

```
1// /subspan.cpp2
2     v 向量 v{1,2,3};
3     span s{v.data()+1,2};
4     tail[0] = 0;
5     cout << v[0] << ','
6          << v[1] << ','
7          << v[2] << '\n';
```

输出 [span] 子跨度 1:

```
1,0,3
```

注意我们仅通过方括号下标访问元素： span 不支持 *at* 方法，因此不进行边界检查。

注释 14 无论是 *std::vector*，*std::array* 还是静态数组，都可以隐式转换为 *std::span*，例如在传递参数时。

```
void f( span<int> s ) {
  for ( auto e : s ) /* ... */
}
int main() {
  f( vector<int>(5) );
```

10.9.6.1 在 *C++20* 之前安装 *span*

克隆仓库：git clone https://github.com/martinmoene/gsl-lite.git

```
add to your compile line
```

-I${HOME}/Installation/gsl/gsl-lite/include (或任何其他到你的路径)

在你的源代码中：#include "
gsl/gsl-lite.hpp" using gsl::span;

10.9.6.2 mdspan

在 C++23 中存在 mdspan，一个多维 span。

一个 mdspan 对象是一个多维笛卡尔积，其中每个维度都可以以多种方式指定，最简单的方式是给出该维度中的范围 。

```
// /transpose.cpp
  vector<float> A(N*N),B(N*N);
    /* ... */
  md::mdspan Amd{ A.data(),N,N };
  md::mdspan Bmd{ B.data(),N,N };
```

or

```
vector<float> ar10203040(10*20*30*40);
auto brick10203040 =
    std::mdspan< float, extents<10,20,30,40> >( ar10203040.data() );
auto mid = brick10203040[5,10,15,20];
```

注意 15 引入 mdspan 使得重新定义了方括号内的逗号的功能。表示顺序执行的逗号运算符仍然可以在其他上下文中使用，例如循环头部。如果你绝对需要在索引表达式中使用顺序逗号，你可以写 a[(i,j)]。

该对象现在有一个范围 方法来查询范围。

在您可能在多维对象上进行的操作类型中，您可能需要多维索引。这由范围 :: 笛卡尔 _ 乘积等方法提供：

```
// /transpose.cpp
  // cartesian product only in range-v3
  auto Aij = rng::views::cartesian_product
    ( rng::iota_view(0,N), // Amd.extent(0)), argument deduction fails
      rng::iota_view(0,N) // Amd.extent(1));
      );
```

这些索引现在可用于 mdspan 对象中的矩阵样式索引。例如，这里是一个矩阵转置，使用 mdsubspan，它是为 C++26 提出的，并从这里来自 *Kokkos* 库：

```
// /transpose.cpp
s
t
d::
for_each(std::execution::par_unseq,
Aij.begin(), Aij.end(), [&](auto idx) {
auto [i, j] = idx; B[j][i] = A[i][j]; });
```

10.9.6.3 从 Kokkos 安装 mdspan

```
// /transpose.cpp#include"mdspan/mdspan.hpp"namespacemd=
Kokkos;namespaceKokkosEx=MDSPAN_IMPL_STANDARD_
NAMESPACE::MDSPAN_IMPL_PROPOSED_NAMESPACE;namespacemdx=
KokkosEx;
```

10.9.7 大小和符号

The `size` 方法返回一个无符号量，因此在使用足够高的警告级别时

```
for (int i=0; i<myarray.size(); i++ )
```

会抱怨混合了有符号和无符号的数量。

您可以

```
for (size_t i=0; i<myarray.size(); i++ )
```

或在使用 C++20 时使用 `ssize` 方法，该方法返回一个有符号的大小：

```
for (int i=0; i<myarray.ssize(); i++ )
```

https://stackoverflow.com/questions/56217283/ 为什么在 C20 中引入了 std::size#56217338

10.10 C 风格数组

静态数组实际上是 C 编程语言中数组和地址等价性的滥用。这体现在参数传递机制中，例如。

对于小数组，你可以使用不同的语法。

代码：

```
// /staticinit.cpp
{
  int 数字 [] ={5,4,3,2,1};
  cout << 数字 [3] <<'\n';
}
{
  int 数字 [5]{5,4,3,2,1};
  numbers[3] = 21;
  cout<< 数字 [3] << '\n';
}
```

输出 [数组] 静态初始化：

```
2
21
```

这有一个（最小的）优点，即没有类机制的额外开销。另一方面，它有很多缺点：

- 您不能通过数组名称查询其大小：您必须在变量中单独存储该信息。
- 将此类数组传递给函数实际上是传递其第一个元素的地址，因此它始终（某种程度）按引用传递。

10.10.1 分配

传统上，C 数组只能以

```
int a[5];float b
[6][7];
```

即，使用显式给定的数组边界。一些编译器作为扩展支持所谓的 变长数组：

```
int n; scanf("%d",&n); // this reads n from the console
double x[n];
```

该机制被添加到 C99 标准中，但由于对其支持并非普遍，C11 标准又将其变为可选。宏 `__STC_NO_VLA__` 如果确实缺乏此类支持，则设置为 `1` 。

还需要注意的是，这些数组是在 栈 上分配的，因此创建一个过大的数组可能会导致栈溢出。这将使您的代码在没有信息性错误的情况下崩溃。

10.10.2 Indexing and range-based loops

基于范围的索引与向量的工作方式相同：

代码:

```
1 // /rangemax.cpp
2   int numbers[] = {1,4,2,6,5};
3   int tmp_max = numbers[0];
4   for (auto v : numbers)
5     if (v>tmp_max)
6       tmp_max = v;
7   cout << "Max: " << tmp_max << "
      (should be 6)" << '\n';
```

Output
[array] rangemax:

```
Max: 6 (should be 6)
```

评论 10.3. 以下代码在某些方面是不正确的。这将如何体现?

int a[5];a
[6] = 1.;int a
[5];a;at(6) = 1.

10.10.3 C-style arrays and subprograms

A数组可以传递给子程序，但边界是未知的 own there.

// /arraypass.cpp void set_array(double *x,int size) {
for (int i=0; i<size; ++i) x[i] = 1.41; }; /* ... */ doublearray
[5] = {11,22,33,44,55}; set_array(array,5); cout <<array
[0] << "...." << array[4] << '\n';

练习 10.18。 重写上述练习，其中排序测试器或最大值查找器位于子程序中。

与标量参数不同，数组参数可以被子程序修改：就好像数组总是通过引用传递一样。这并不完全正确：发生的是传递数组的第一个元素的地址。因此，我们实际上是在处理按值传递，但传递的是数组地址而不是其值。

在子程序中，这样的静态数组与指针无法区分。这被称为 指针衰减。以下代码和错误信息说明了这一点:

代码：

```
1// /carray.cpp2void std::f(int
stat) {3 printf(".. in function: %
lu\n",
std::size(stat));4}5//codesnippet
end6 /* ...*/7 int stat[23];8
std::f(stat);
```

Output
[array] carray:

```
carray.cxx: In function
    'void std_f(int*)':
carray.cxx:18:43: error: no
    matching function for
    call to 'size(int*&)'
   18 |   printf(".. in
    function:
    %lu\n",std::size(stat));
       |

    ~~~~~~~~~~~~~~~
```

10.10.4 Size of arrays

What does the`sizeof` 运算符在不同类型的数组上给出什么?

Code:

```
1// /staticsize.cpp 2 int a1[10]; 3 cout <<"
static: " << sizeof(a1) <<'\n'; 4 int *a2 =
(int*) malloc( 10*sizeof(int) ); 5 cout <<"
malloc: " << sizeof(a2) <<'\n'; 6 vector<i
nt>a3(10); 7 cout <<"vector: " << sizeof(a3) <<'
\n';
```

输出
[数组] 静态大小:

```
静态： 40
malloc: 8
vector: 24
```

您可能认为 `sizeof` 在静态数组上是有用的，但这在传递到子程序时并不成立：

Code:

```
1 // /carray.c
2 void stat_f( int stat[] ) {
3   printf(".. in function:
      %lu\n",sizeof(stat));
4 }
5 //codesnippet
```

Output
[c] carraystat:

```
carray.c:16:40: warning:
    sizeof on array function
    parameter will return
    size of 'int *' instead
    of 'int []'
    [-Wsizeof-array-argument]
  printf(".. in function:
    %lu\n",sizeof(stat));

                ^
carray.c:15:18: note:
    declared here
void stat_f( int stat[] ) {
                  ^
1 warning generated.
Size of stat[23]: 92
.. in function: 8
```

注意编译器警告

```
warning: sizeof on array function parameter will return size of 'int *' in
```

10.10.5 多维数组

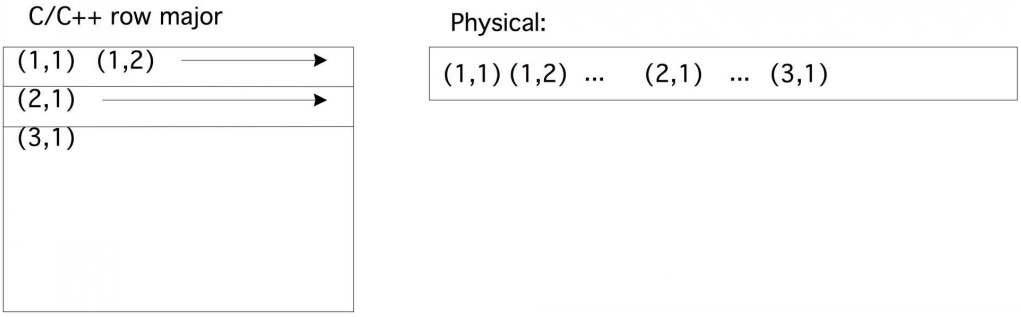
多维数组可以通过对先前语法的简单扩展来声明和使用 :

```
float matrix[15][25];

for (int i=0; i<15; i++)
  for (int j=0; j<25; j++)
    // something with matrix[i][j]
```

将多维数组传递给函数时，只有第一维可以省略不指定:

```
// /contig.cppvoid print12(int ar[][6] ) {cout <<"
Array[1][2]: " << ar[1][2] << ' \n' ;return; }/* ...
*/int array[5][6];array[1][2] = 3;
array{v48};print12(array);
```



10.10.6 内存布局

数组的一些令人困惑的方面，例如在函数调用中需要指定哪些维度以及哪些不需要，可以通过考虑数组如何在内存中存储来理解。那么问题就是如何将二维（或更高维）数组映射到内存中，这是线性的。

- 一维数组存储在连续的内存中。
- 二维数组也连续存储，首先是第一行，然后是第二行，等等。
- 更高维的数组继续这一概念，具有最高维度的连续块。

因此，超出行尾的索引将带你到下一行的开头：

```
// /contig.cppvoid print06(int ar){cout << "Array: "
<< ar << '\n';}/* ... */int array;array;print06(array);
```

现在我们也可以理解数组是如何传递给函数的：

- 传递给函数的唯一信息是数组第一个元素的地址；
- 为了能够找到第二行（以及第三行，等等）的位置，子程序需要知道每行的长度。
- 在更高维的情况下，子程序需要知道除第一维之外所有维的大小。

10.11 练习

练习 10.19。 给定一个整数向量，编写两个循环；

1. 一个用于求所有偶数元素的和，2. 一个用于求所有偶数索引元素的和。

使用正确的循环类型。

练习 10.20. 程序 冒泡排序：遍历数组，比较相邻的两个元素，如果第二个比第一个小，就交换它们。遍历完数组后，最大的元素位于最后一个位置。再次遍历数组，交换元素，将第二大的元素放在倒数第二个位置。以此类推。

帕斯卡三角形包含二项式系数：

```
Row    1:                      1
Row    2:                    1   1
Row    3:                  1   2   1
Row    4:                1   3   3   1
Row    5:              1   4   6   4   1
Row    6:            1   5  10  10   5   1
Row    7:          1   6  15  20  15   6   1
Row    8:        1   7  21  35  35  21   7   1
Row    9:      1   8  28  56  70  56  28   8   1
Row   10:    1   9  36  84 126 126  84  36   9   1
```

where

$$p_{rc} = \binom{r}{c} = \frac{r!}{c!(r-c)!}.$$

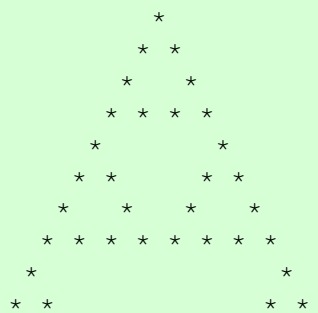
系数可以从递推关系计算得到

$$p_{rc} = \begin{cases} 1 & c \equiv 1 \vee c \equiv r \\ p_{r-1,c-1} + p_{r-1,c} \end{cases}$$

(有其他公式。为什么它们不太常用？)

练习 10.21.

- 编写一个类 pascal，使得 pascal(n) 是包含 n 行上述系数的对象。编写一个方法 get(i,j) 返回 (i, j) 系数。
- 编写一个方法 print，打印上述显示。
- 首先打印整个 pascal 三角形；然后：
- 编写一个方法 print(int m)，如果系数模 m 非零则打印星号，否则打印空格。



- 该对象需要在内部有一个数组。最简单的解决方案是创建一个大小为 $n \times n$ 的数组。
- 你的程序应该接受：

1. an integer for the size
2. any number of integers for the modulo; if this is zero, stop, otherwise print stars as described above.

练习 10.22. 扩展帕斯卡练习：优化你的代码，以使用恰好足够的空间来存储系数。

练习 10.23. 国际象棋棋盘上的骑士通过水平或垂直移动两步，并在正交方向上移动一步来移动。给定一个起始位置，找到一个移动序列，使骑士返回其起始位置。是否存在起始位置，对于这些位置不存在这样的循环?

练习 10.24. 从《保持真实》这本书，马尔可夫链的练习 3.6。

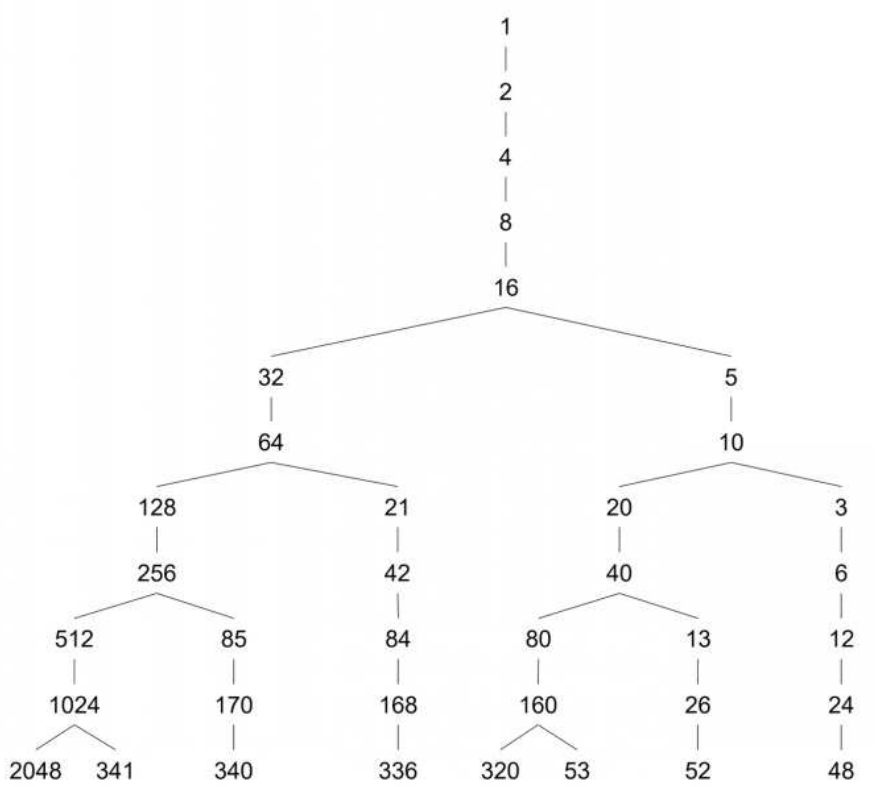


图 10.1: ‘Collatz 树’

练习 10.25. 重新审视练习 6.13，并生成 ‘Collatz 树’ （图 10.1 ）：在级别 n (基于一计数) 是在 $n-1$ 步内收敛到 1 的数字。

读入一个数 n 并打印前 n 行，每行一个新行，数字之间用空格分隔。

第 11 章

字符串

11.1 字符

- 类型 `char`;
- 表示 ‘7 位 ASCII 字符’：可打印和（某些）不可打印字符。
- 单引号：char c = ’a’

相当于（短）整数：

Code:

```
1 // /intchar.cpp
2   char ex = 'x';
3   int x_num = ex, y_num = ex+1;
4   char why = y_num;
5   cout << "x is at position " << x_num
6        << '\n';
7   cout << "one further lies " << why
8        << '\n';
```

Output

```
[string] intchar:

x is at position 120
one further lies y
```

Also: `'x'-'a'` is distance `a--x`

注意 16 从 `'x'` 到 *ascii* 码的转换，特别是具有连续值的字母，不保证由标准提供。

练习 11.1. 编写一个程序，接受一个整数 $1 \cdots 26$ 并打印第 so-many 个字母。

扩展你的程序，以便如果输入为负数，它打印第 minus-so-many 个大写字母。

11.2 Basicstring stuff

```
#include <string>
using std::string;

// .. 和现在你可以使用 'string'
```

(不要使用 C 遗留机制。)

一个 *string* 变量包含一个字符序列。

```
字符串 文本;
```

您可以初始化字符串变量或动态分配它:

```
string txt{v3}"this is text";}string
moretxt{(v3)"this is also text";}txt= "and
now it is another text";
```

通常，引号表示字符串的开始和结束。那么，如果您想在一个字符串中包含引号怎么办?

您可以转义引号，或者指示整个字符串应按字面意义处理:

代码:

```
// /quote.cpp
  strin g<one>("a b c"),
     两 ("a \"b\" c"),

     three( R"("a ""b """c)" );
  cout << one<< ' \n' ;
  cout<< 两 << ' \n' ;
  cout << three <<' \n' ;
```

输出
[字符串] 引号:

```
a b c
a "b" c
"a ""b """c
```

字符串可以 连接:

代码:

```
1// /strings.cppti
  s r ng my_strin space{" "};
    g,my_字符串 ="foo";

  my_ 字符串 += 空格 + "bar";
  cout <<my_string << ":          <<
    my_string.size() << '\n';
```

输出
[字符串] 字符串相加:

```
foo bar: 7
```

您可以查询 *size*:

代码 :

```
1// /strings.cpp
  字符串 五
文本{"五吾"};3 cout << 五 _ 文本 . 大小 () <<'\n';
```

Output
[字符串] 字符串大小 :

```
5
```

或使用下标:

代码:

```
1// /stringsub.cpp
2   字符串数字{"0123456789"};
3   cout << "char three: "
4        << digits[2] << '\n';
5   cout << "char four : "
6        << digits.at(3) << '\n';
```

Output
[字符串] 字符串子:

char three: 2
char four: 3

Same as ranging over vectors.

Range-based for:

代码:

```
1 // /stringrange.cpp
2 cout <<"按字符: ";
3   for ( char c : abc )
4     cout << c << " ";
5   cout << '\n';
```

Output
[string] stringrange:

按字符: *a b c*

按索引范围:

代码:

```
1 // /stringrange.cpp
2   string abc = "abc";
3   cout <<"By character: ";
4   for (int ic=0; ic<abc.size(); ic++)
5     cout << abc[ic] <<" ";
6   cout << '\n';
```

输出
[字符串] 字符串索引:

按字符: *a b c*

基于范围的 for 循环会复制元素
你也可以获取一个引用:

代码:

```
1// /stringrange.cpp
2 <for> <(> <char> <&> <c> <:> <abc> <)>
3 c += 1;4 cout "<<\"Shifted: \" << abc << '
\n';
```

Output
[字符串] 字符串范围集合:

移位: *bcd*

```
for (auto c: some_string)
  // do something with the character 'c'
```

Review 11.1. True or false?

1. '0' 是 achar 变量的有效值 2. "0" 是 char 变量的有效值 3. "0" 是字符串变量的有效值 4. 'a'+'b' 是 achar 变量的有效值

练习 11.2. 书写秘密信息最古老的方法是 凯撒密码。你需要一个整数 s，并将文本中的每个字符旋转该数量的位置：

$$s \equiv 3\text{: "acdz"} \Rightarrow \text{"dfgc".}$$

编写一个程序，接受一个整数和一个字符串，并显示旋转该数量的位置的原始字符串。

练习 11.3. (这个继续练习 11.2)
如果你发现一个用凯撒密码加密的消息，你能解密它吗？从 福尔摩斯 故事《跳舞的人》中获得灵感，在那里他利用了 ‘e’ 是最常见的字母这一事实。

你能实现一个更通用的字母置换密码，并用 ‘跳舞的人’ 方法破解它吗?

向量类（vector class）的其他方法应用：插入（insert）、清空（empty）、删除（erase）、追加（push_back），等等。

代码:

```
1// /strings.cpp  2 string five_chars;  3 cout <<
five_chars.size() <<  "\n";  4 for (int i=0; i<5; ++i)
5 five_chars.push_back(' ');  6 cout <<five_
chars.size() <<  "\n";
```

```
Output
[string] stringpush:

0
5
```

Methods only for `string`: `find` and such.

http://en.cppreference.com/w/cpp/string/basic_string

练习 11.4。 编写一个函数来打印数字的每一位：156 应该打印 one five six。你需要先将数字转换为字符串；你能想到更多种方法来做这件事吗?

Start by writing a program that reads a single digit and prints its name.

对于完整的程序来说，最容易的方法是按从后到前的顺序生成数字。然后想办法将它们逆序打印。

练习 11.5。 编写一个函数将整数转换为字符串：输入 215 应该输出 two hundred fifteen，等等。

练习 11.6. 编写一个模式匹配器，其中无符号点 . 匹配任何单个字符，x* 匹配任意数量的 ‘x’ 字符。

例如：

- 字符串 abc 匹配 a.c，但 abbc 不匹配。
- 字符串 abbc 匹配 ab*c，ac 也匹配，但 abzbc 不匹配。

11.3 字符串流

您可以使用 + 运算符连接字符串。小于小于运算符也进行某种形式的连接。它很吸引人，因为它可以将数量转换为字符串。有时您可能希望结合这些功能：转换为字符串，并以字符串作为结果。

为此，您可以使用来自 *sstream* 头文件的字符串流 。

像 cout （包括从数量到字符串的转换），但输出到对象，而不是屏幕 .

- 使用 << 运算符构建它；然后
- 使用 *str* 方法提取字符串。

```
1 #include <sstream>
2 stringstream s;
3 s << "text" << 1.5;
4 cout << s.str() << endl;
```

11.4 高级主题

11.4.1 字符串视图

很多时候，你将操作字符串对象而不改变它们。为此，C++17 引入了 *string_* 视图 (在 *string_* 视图 头文件中)，它为你提供了对字符串的只读视图。字符串视图有许多操作，例如通过在前面或后面截断若干个字符来获取一个新的视图。

11.4.2 原始字符串字面量

您可以通过转义来在字符串中包含引号或反斜杠等字符。这可能会很繁琐。C++11 标准有一种机制用于 原始字符串字面量。

最简单形式：

代码：

```
1 // /raw.cpp
2 cout<< R"(string with }
3 4{ \weird\ stuf)" << '\n';
```

输出

[字符串] 原始 1:

```
string with }
{\奇怪\ 东西
```

现在显而易见的问题当然是如何在字符串中包含闭合括号引号序列。为此，您可以指定自己的多字符分隔符：

Code:

```
1 // /raw.cpp
2 cout << R"limit("(string with }
3
4 { \weird\ stuff)")limit" << '\n';
```

Output

[string] raw2:

```
"(string with }
{ \weird\ stuff)"
```

11.4.3 字符串字面量后缀

一个字符串字面量 "foo" 通常与一个 std:: 字符串 兼容，但它不是那种类型。如果您需要那种类型，可以向字面量添加后缀，该后缀定义在 string_literals 命名空间中：

Code:

```
// /strings.cpp
void printfun( string s ) {
  cout << s << '\n';
}
void printfun_c( const string& s ) {
  cout << s << '\n';
}
    /* ... */
    using namespace
    std::string_literals;
    printfun( "abc" );
    printfun( "def"s );
    printfun_c( "ghi" );
    printfun_c( "jkl"s );
```

Output
[string] stringsuffix:

```
abc
def
ghi
jkl
```

11.4.4 转换到 / 从字符串

11.4.4.1 转换到字符串

在字符串和数字之间转换有多种机制 .

- C语言的遗留机制 sprintf 和 itoa。
- 到_ 字符串
- 前面你看到了 stringstream; 部分 11.3。下面是另一个使用这个头文件的方法。
- Boost 库有一个 *lexical cast*。

此外，在 C++17 中有一个 *charconv* 头文件，带有 to_chars 和 from_chars。这些是低级例程，不会抛出异常，不会分配内存，并且它们互为逆操作。低级特性体现在例如它们在字符缓冲区上工作（不是以空字符结尾）。因此，它们可以用来构建更复杂的工具。

11.4.4.2 从字符串转换

The stringstream 对象可用于将字符串转换为数字：

1. 使用字符串初始化字符串流； 2. 使用 cin- 类似的语法从流
中设置一个数值变量。

// /toint.cppstring stringnum="12345";;int num;;stringstream numstream(stringnum);numstream >>num;

11.4.5 Unicode

C++ 字符串本质上是由字符组成的向量。一个字符是一个字节。不幸的是，在这些互联网时代，有比一个字节能容纳的更多的字符。特别是，有 *Unicode* 标准涵盖了数百万个字符。它们的渲染方式是通过一个 可扩展编码，特别是 *UTF8*。这意味着有时一个 ‘字符’，或者更正确地说是 字形，需要超过一个字节。

11.5 C strings

在 C 中，字符串本质上是由字符组成的数组。C 数组不存储它们的长度，但字符串确实有函数隐式或显式地依赖于这些知识，所以它们有一个终止字符：ASCII`NULL`。C 字符串因此被称为空终止。

11. 字符串

第 12 章

输入 / 输出

大多数程序都需要某种形式的输入，并产生某种形式的输出。在像这样的入门课程中，输出比输入更重要，而且现有的输出只显示在屏幕上，所以我们从关注这一点开始。我们还将查看文件输出和输入。

12.1 流 vs theformat 库

到目前为止的示例中，你主要看到 `cout` 用于屏幕输出。这使用 流。流非常有用，但越来越应该被视为一种较低级别的机制。

在 C++20 中有 `format` 头文件，它实现了开源 *fmtlib* 库的大部分功能。与流相比，使用这个头文件有几个优点，因此我们将讨论它作为首选机制。然而，由于编译器仍在追赶 C++20 标准，我们也将详细讨论流格式化。

有点奇怪的是，使用 C++20 版本的 `format`，你只能形成格式化的字符串，但不能将其输出到屏幕。解决方案是

```
auto s = std::format( /* 格式化内容 */ );cout <<
s.str() <<'\n';
```

在 C++23 中，

```
std::print( /* 格式化内容 */ );
```

已经添加。

12.2 Screen output

在之前的示例中，你使用了 `cout` 及其默认格式。在本节中，我们将探讨自定义 `cout` 输出的方法。

备注 17 即使在你阅读下面的内容之后，你也可能发现 `cout` 并不特别优雅。事实上，如果你之前用 *C* 语言编程过，你可能会更喜欢 `printf` 机制。*C++20* 标准有 `format` 头文件，它和 `printf` 一样强大，但 *considerably more elegant*（相当优雅得多）得多。

然而，截至 *2022* 年初，这在大多数编译器中不可用，因此，在 *12.7* 章节中，我们将给出来自 fmtlib的示例，这是产生 `std::format` 的开源库。

从 *iostream*: cout 使用默认格式。

在 *iomanip* 头文件中可以进行可能的操作：填充数字、使用有限精度、以十六进制格式化等 .

通常，数字的输出占用它所需的精确空间：

代码：

```
// /io.cpp for(int i=1;i<200000000;
i*=10) cout << "数字："<<i <<'\n';

cout <<'\n';
```

输出 [io] cunformat：

Number: 1*Number*:
10*Number*: 100*Number*:
1000*Number*:
10000*Number*:
100000*Number*:
1000000*Number*:
10000000*Number*:
100000000

我们将现在查看一些非标准格式的示例。您可能想要输出多行，其中数字整齐对齐。最常见的 I/O 操作是设置统一宽度，即每个数字使用相同的位置数，无论它们需要多少。

- The setw 指定了用于以下数字的位置数。
- 注意上一句话中的单数：thesetw 指定符仅适用一次。
- 默认情况下，数字在给定的空间内右对齐，如果它们需要更多位置，它们会向右溢出。

您可以指定位置数量，输出默认在该空间内右对齐：

Code：

```
// /width.cpp
#include <iomanip>
using std::setw;
    /* ... */
  cout <<"Width is 6:"<< '\n';
  for(int i=1;i<200000000; i{i}*=10)co t"<< \"Number: \"
      u
         << setw(6) << i << '\n';
  cout <<'\n';

  // 'setw' applies only once:
  cout << "Width is 6:" << '\n';
  cout <<("
       << setw(6) << 1 << 2 << 3 << '\n';
  cout <<  '\n';
```

输出 [io] 宽度：

```
宽度是 6:
编号：     1
编号：    10
编号：   100
编号：  1000
编号： 10000
数字：100000
数字：1000000
数字：10000000
数字：100000000

宽度是 6:
>   123
```

通常，使用空格进行填充，但你也可以指定其他字符：

Code:

```
// /io.cpp
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << setfill('.') << setw(6) << i
    << '\n';
  cout << '\n';
```

Output
[io] formatpad:

```
Number: .....1
Number: ....10
Number: ...100
Number: ..1000
Number: .10000
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

注意：单引号表示字符，双引号表示字符串。

你可以使用左对齐代替右对齐：

Code:

```
// /io.cpp
  for (int i=1; i<200000000; i*=10)
    cout << "Number: "
         << left << setfill('.') <<
    setw(6) << i << '\n';
```

Output
[io] formatleft:

```
Number: 1.....
Number: 10....
Number: 100...
Number: 1000..
Number: 10000.
Number: 100000
Number: 1000000
Number: 10000000
Number: 100000000
```

最后，你可以打印不同的数字基数（非 10 ）：

Code:

```
// /io.cpp
  cout << setbase(16) << setfill(' ');
  for (int i=0; i<16; ++i) {
    for (int j=0; j<16; ++j)
      cout << i*16+j << " " ;
    cout << '\n';
  }
```

Output
[io] format16:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d 6e 6f
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d 7e 7f
80 81 82 83 84 85 86 87 88 89 8a 8b 8c 8d 8e 8f
90 91 92 93 94 95 96 97 98 99 9a 9b 9c 9d 9e 9f
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 aa ab ac ad ae af
b0 b1 b2 b3 b4 b5 b6 b7 b8 b9 ba bb bc bd be bf
c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 ca cb cc cd ce cf
d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 da db dc dd de df
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 ea eb ec ed ee ef
f0 f1 f2 f3 f4 f5 f6 f7 f8 f9 fa fb fc fd fe ff
```

练习 12.1. 使上述输出中的第一行与其他行对齐更好：

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f
etc
```

十六进制输出对于地址（第 17.2 章）：

代码：

```
// /coutpoint.cpp
  int i;
  cout << "address of i, decimal: "
       << (long)&i << '\n';
  cout << "address of i, hex    : "
       << std::hex << &i << '\n';
```

Output
[指针] coutpoint:

i 的地址，十进制：
140732703427524*i* 的地址，十六进制：

0x7ffee2cbcbc4

返回十进制：

```
cout << hex << i << dec << j;
```

没有标准的修饰符用于以二进制形式输出。但是，你可以使用 `bitset` 头文件来打印整数的位模式。

代码：

```
// /bits.cpp
#include <bitset>
using std::bitset;
    /* ... */
  auto x255 = bitset<16>(255);
  cout <<x255 << ' \n' ;
```

输出
[io] bits:

0000000011111111

12.2.1 浮点数输出

浮点数的输出更加复杂。

- 小数点前的数字有多少位?
- 小数点后打印多少位数字?
- 是否使用科学计数法?

对于浮点数，`setprecision` 修饰符决定了整数部分和小数部分一起使用的位数。如果整数部分使用的位数更多，则使用科学计数法。

使用 setprecision 设置小数点前后的数字位数：

Code:

```
// /formatfloat.cpp
#include <iomanip>
using std::left;
using std::setfill;
using std::setw;
using std::setprecision;
    /* ... */
  x = 1.234567;
  for (int i=0; i<10; ++i) {
    cout << setprecision(4) << x << '\n';
    x *= 10;
  }
```

Output
[io] formatfloat:

```
1.235
12.35
123.5
1235
1.235e+04
1.235e+05
1.235e+06
1.235e+07
1.235e+08
1.235e+09
```

此模式是固定点和浮点数的混合。请参阅科学 选项下的一致浮点数格式使用。

使用 固定修饰符时， `setprecision` 适用于小数部分。

固定精度适用于小数部分：

代码：

```
// /fix.cpp
  {x}3.234567;
  cout << fixed;
  for(int i=0; i<10; ++i){v18}co t <<
  setprecision(4)<                    < x << '\n';
    x *= 10;
  }
```

输出
[io] fix:

```
1.2346
12.3457
123.4567
1234.5670
12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

(Notice the rounding)

The `setw` 修饰符，用于定点输出，适用于整数部分和分数部分的宽度之和，包括小数点。

Combine width and precision:

代码:

```
1// /align.cpp2
    {x}3.234567;
3   cout << fixed;
4   for(int i=0; i<10; ++i) {
5      cout << setw(10) setprecision(4)
        << x
6            << '\n';
7      x = 10;
8   }
```

输出
[io] align:

```
     1.2346
    12.3457
   123.4567
  1234.5670
 12345.6700
123456.7000
1234567.0000
12345670.0000
123456700.0000
1234567000.0000
```

Exercise 12.2. Use integer output to print real numbers aligned on the decimal:

Code:

```
1 // /quasifix.cpp
2 string quasifix(double);
3 int main() {
4    for (auto x: { 1.5, 12.32, 123.456,
       1234.5678 } )
5       cout << quasifix(x) << '\n';
```

输出
io 输出伪固定格式:

```
   1.5
  12.32
 123.456
1234.5678
```

使用四个空格表示整数和小数部分；仅使用符合此格式的数字进行测试 .

你之前看到了 `setprecision` 可以同时给出固定和浮点输出。要得到严格浮点 ‘科学’ 记数法输出，请使用 `scientific`。

结合宽度与精度:

代码:

```
1// /iof.cpp
2   x = 1.234567;
3   cout << 科学;
4   for(int i=0; i<10; ++i) {
5      cout << setw(10) setprecision(4)
6           << x << '\n';
7      x *= 10;
8       }}cout <<'\n';
9
```

输出
[io] iofsci:

```
1.2346<e>+00
1.2346<e>+01
1.2346<e>+02
1.2346<e>+03
1.2346<e>+04
1.2346<e>+05
1.2346<e>+06
1.2346<e>+07
1.2346<e>+08
1.2346<e>+09
```

12.2.2 布尔输出

The*boolalpha* modifier 渲染一个 **bool** 变量作为 **true**, **false**.

12.2.3 保存和恢复设置

ios::fmtflags old_ 设置 =cout.flags();

cout.flags(old_ 设置);

int old_ 精度 = cout.precision();

```
cout.precision(old_precision);
```

12.3 File output

The *iostream* 只是一个 < 样式 id='5'> 流 </ 样式 > 的例子，而 < 样式 id='5'> 流 </ 样式 > 是一种将实体转换为可导出形式的通用机制。

特别是，文件输出与屏幕输出工作方式相同：在您创建一个流变量后，可以将其 “ 小于小于 ” 输出到其中。

```
mystream << "x: " << x << '\n';
```

以下示例使用了一个 *ofstream*：一个输出文件流。它有一个 *open* 方法来将其与文件关联，并且有一个相应的 *close* 方法。

Use:

Code:

```
1  // /fio.cpp
2  #include <fstream>
3  using std::ofstream;
4      /* ... */
5    ofstream file_out;
6    file_out.open
7      ("fio_example.out");
8      /* ... */
9    file_out << number << '\n';
10   file_out.close();
```

Output

[io] fio:

```
echo 24 | ./fio ; \
          cat fio_example.out
A number please:
Written.
24
```

Compare: `cout` is a stream that has already been opened to your terminal ‘file’.

The *open* 调用可以有标志，例如用于追加：

```
file.open(name,std::fstream::out | std::fstream::app);
g
```

二进制输出：从内存中逐字节写入数据到文件。（为什么这比可打印表示更好？）

代码：

```
1// /fiobin.cpp
2    ofstream file_out;
3    文件 _ 输出 _ 打开 ()
4            o    nary.out",ios::binary);
5        //file_out.,write((char*)(&number),4);
6
```

输出
[io] fiobin:

```
echo  25 |./fiobin ;\
              od fio_binary. out
请输入一个数字 ：已写入。
0000000          000031 000000
0000004
```

12.4 输出你自己的类

你已经使用过类似以下的语句：

cout<<" 我的值是："<<myvalue<<"\n";

这是如何工作的？ ‘双重小于’ 是一个操作符，其左操作数是一个流，右操作数定义了输出；这个操作符的结果仍然是一个流。递归地来说，这意味着你可以将任意数量的 << 应用链接在一起。

如果你想要输出你自己编写的类，你必须定义 << 操作符如何处理你的类。

在这里我们分两步解决这个问题：

定义一个函数，该函数生成一个表示对象的字符串，并且

```
1// /pointfunc.cpp  2 string as_string() { 3
stringstream ss;  4 ss <<  "(" <<  x<<  "," <<  y<<  ")
";  5 return ss.str();  6 };  7 /* ... */  8std::ostrea
m& operator<<  9(std::ostream &out,Point &p) {
10 out <<  p.as_string(); return out;  11 };
```

重新定义小于小于操作符以使用这个。

```
1 // /pointfunc.cpp
2     Point p1(1.,2.);
3     cout << "p1 " << p1
4          << " has length "
5          << p1.length() << "\n";
```

（参见第 11.3 节关于 stringstream 和 sstream 头文件的内容。）

如果你不想编写那个访问器函数，你可以将 less-less 运算符声明为 friend：

```
类 容器 { 私有 : 双精度浮点数 x; 公有 : 朋友 输出流 & 操作符 <<( 输出流 & s, 常
量容器 & c ) {s <<c.x; /* 无访问器 */ 返回 s;};};
```

12.5 输出缓冲

在 C 语言中，要在输出中获取换行符的方法是在输出中包含字符 `\n` 。这仍然在 C 语言中有效 ++，最初看起来使用 `endl` 没有区别。然而，`endl` 不仅用于断行输出：它执行一个 `std::` 刷新。

12.5.1 刷新的需求

输出通常不会立即写入屏幕或磁盘或打印机：它会被保存在缓冲区中。这可能是出于效率考虑，因为输出单个字符可能会有很大的开销，也可能是因为设备正在忙于做其他事情，而你不想让程序挂起等待设备空闲。

然而，缓冲区的问题在于屏幕上的输出可能会落后于程序的实际情况。特别是，如果你的程序在打印某条消息之前崩溃了，这意味着它是在到达那条语句之前崩溃的，还是意味着消息卡在缓冲区里了。

这种输出，在调用语句时绝对需要处理的输出，通常被称为日志 输出。事实上，`endl` 执行刷新意味着它适合用于日志输出。然而，它也会在不严格必要时刷新。事实上，有一个更好的解决方案：`std::cerr` 就像 `cout` 一样工作，只是它不会缓冲输出。

12.5.2 性能考虑

如果你想在输出（无论是屏幕还是更通用的流）中换行，使用 `endl` 可能会因为它的刷新操作而减慢你的程序。更高效的方法是直接在输出中添加换行符：

```
somestream << "Value: " << x << '\n';
otherstream << "Total " << nerrors << " reported\n";
```

换句话说，使用 `cout` 进行常规输出，`cerr` 进行日志输出，并使用 `\n` 而不是 `endl`。

12.6 输入

使用 `cin` 命令可以读取整数和浮点数格式。

Code:

```
// /cinfloat.cpp
    float input;
    cin >> input;
    cout << "(I think I got: " << input
    << ")\n";
```

Output
[io] cinfloat:

```
for n in \
            1.5  1.6  1.67
    1.67e5  2.5.6  \
            ; do \
                echo $n |
    ./cinfloat \
        ; done
(I think I got: 1.5)
(I think I got: 1.6)
(I think I got: 1.67)
(I think I got: 167000)
(I think I got: 2.5)
```

如本例中最后一位数字所示，`cin` 将读取直到第一个不符合变量格式的字符，在此情况下是第二个句点。另一方面，它前面的数字中的 e 被解释为浮点表示的指数。

最好使用 `getline`。这返回一个字符串，而不是一个值，因此你需要用以下魔法将其转换：

```
// /helloinwhat.cpp #include <iostream>
using std::cin; using std::cout; #include <
sstream>using std::stringstream; /* ...*/
std::string saymany; int howmany; cout
<<"多少次？ "; getline( cin,saymany );
stringstream saidmany(saymany);
saidmany>> howmany;
```

你不能在同一个程序中使用 `cin` 和 `getline`。

More info: http://www.cplusplus.com/forum/articles/6046/.

12.6.1 文件输入

输入文件流，使用 `open` 方法，然后使用 `getline` 逐行读取：

```
// /quickinput.cpp#include <fstream>
using {v9}std{v11}::ifstream{v15};/* ...
*/ifstream input_
file{v24};i
nput{v28}file{v31}.open{v35}("fox.txt"
{v39});string{v43}oneline{v45};
```

```
while (getline(input_file,oneline)) {
  cout << "Got line: <<" << oneline << ">>" << '\n';
}
```

检测文件结束有多种方法

对于文本文件，getline 函数在没有可读行的情况下返回 false。

- The *eof* function can be used after you have done a read.
- *EOF* is a return code of some library functions; it is not true that a file ends with an EOT character. Likewise you can not assume a Control-D or Control-Z at the end of the</style> 文件。

练习 12.3。 将以下文本放入文件：

the quick brown fox
jummps over the lazy
dog。

打开文件，读取它，并计算字母表中的每个字母在 i 中出现的频率 t

Advanced note: You may think that *getline* always returns a **bool**, but that's not true. If actually returns an *ifstream*. However, a conversion operator

```
explicit operator bool() const;
```

对于继承自基本 *_ios* 的任何内容都存在。

12.6.2 输入流

测试，主要用于文件流：*is_eof is_open*

12.6.3 C 风格文件处理

旧的 *FILE* 类型不应再使用。

12.7 Fmtlib

12.7.1 basics

- *print*用于打印：*format* 给出 *std::string*;
- 花括号指示的参数；
- 括号可以包含数字（以及修饰符，见下文）

Code:

```
// /fmtlib.cpp
  auto hello_string = fmt::format
    ("{} {}!","Hello","world");
  cout << hello_string << '\n';
  fmt::print
    ("{0}, {0} {1}!\n","Hello","world");
```

Output
[io] fmtbasic:

```
Hello world!
Hello, Hello world!
```

API documentation: https://fmt.dev/latest/api.html

12.7.2 对齐和填充

在 fmtlib 中，大于号加数字表示右对齐和字段宽度。

代码:

```
// /fmtlib.cpp
    for(int i=10; i<2000000000; i{v15}*=10)
      fmt::print("{:}|n", v11, i);
```

输出
io] fmtwidth:

```
        10
       100
      1000
     10000
    100000
   1000000
  10000000
 100000000
1000000000
1410065408
1215752192
```

Code:

```
// /fmtlib.cpp
    for (int i=10; i<2000000000; i*=10)
      fmt::print("{0:.<v11}]|n", i);
```

输出
iofmtleftpad:

```
....10
...100
..1000
.10000
100000
1000000
10000000
100000000
1000000000
1410065408
1215752192
```

12.7.3 构造字符串

如果你想要逐步构造一个字符串，例如因为它涉及对某物的循环，你可以使用一个内存 _ 缓冲区：

fmt::memory_buffer b; fmt::format_to(std::back_inserter(b),"[
"); for (auto i : indices) fmt::format_to(std::back_inserter(b),"
*{}, ",i); fmt::format_to(std::back_inserter(b),"]"); cout <<to_*
string(b) << endl;

12.7.4 Number bases

在 fmtlib 中，您可以通过指定 box 来表示二进制、八进制、十六进制等来指定要用于表示整数的基数。

代码：

```
1 // /ppfmt::print
2
3         ("{0} = {0:b} bin,                {0:o}
      八进制,        {0:x} 十六进制，
4         17);
```

输出
iofmtbase:

```
17 = 10001bin,
     21 oct,
     11 hex
```

12.7.5 输出你自己的类

With `fmtlib` 这采用不同的方法：在这里你需要特化 `formatter` struct/class。

代码：

```
1 // /fmtlib.cpp
2 template <> struct
       fmt::formatter<point> {
3  constexpr
4  auto parse(format_parse_context& ctx)
5        -> decltype(ctx.begin()) {
6    auto it = ctx.begin(),
7      end = ctx.end();
8    if (it != end && *it != '}')
9      throw format_error("invalid
       format");
10   return it;
11  }
12  template <typename FormatContext>
13  auto format
14       (const point& p, FormatContext&
       ctx)
15         -> decltype(ctx.out()) {
16    return format_to
17      (ctx.out(),
18       "{}", p.as_string());
19  }
20 };
21    /* ... */
22    point p(1.1,2.2);
23    fmt::print("{}\n",p);
```

Output
[io] fmtstream:

```
(1.1,2.2)
```

12.7.6 Output a range

在 C++23 中，fmtlib 可以立即处理范围：

```
auto rng = std::range:views::something;
std::print("{}\n",rng);
```

第 13 章

Lambda 表达式

The mechanism of *lambda* 表达式 (first added in C++11 and since expanded) makes dynamic defi- nition offunctions possible.

Traditional function usa ge:
g explicitly define a function and
apply it:

```
double sum(float x,float y) { return x+y; }
cout << sum( 1.2, 3.4 );
```

New:
apply the function recipe directly:

Code:

```
1 // /lambdaex.cpp
2   [] (float x,float y) -> float {
3     return x+y; } ( 1.5, 2.3 )
```

Output
[func] lambdadirect:

```
3.8
```

这个例子当然没什么意义，但它说明了 lambda 表达式的语法：

```
[捕获] ( 输入 ) -> 输出类型{ 定义 };[捕获] ( 输入 )
{ 定义 };
```

- 在这种情况下，但并非通常情况下，空方括号是捕获 部分；
- 然后是通常的参数列表；
- 使用一种风格化的箭头可以指示返回类型，但如果编译器可以自行推断，则这是可选的；
- 最后是通常的函数体，对于非 void 函数，需要包含返回 语句。

注释 18 *Lambda* 表达式有时被称为 闭包，但在编程语言理论中，这个术语具有技术含义，与 *C++ lambda* 表达式只有部分重合。

存在 ‘立即调用的 Lambda 表达式’ 的用法。

示例：不同的构造函数。

无法工作：

```
if (foo)
   MyClass x(5,5);
else
   MyClass x< 样式
   id='1'> (6) ; </样式
   >
```

解决方案：

```
auto x =
  [foo] ()
    if (foo)
      返回 MyClass(5,5);
    else
      返回 MyClass(6);
  }();
```

注意 auto 的使用和省略的返回类型。

为了一个稍微更有用的例子，我们可以将 lambda 表达式赋值给一个变量，并反复应用它。

代码：

```
// /lambdaex.c
auto 求和 =

  [] (float x,float y) -> float {
    返回 x+y;};o t <<s
mmin
  ,          ;cout <<求和 ( 3.7, 5.2 ) << '\n';
```

输出 [f]

```
unc  lambdavar:

3.8
8.9
```

- 这是一个变量声明。
- 使用自动 出于技术原因；见下文。

返回类型本可以省略：

```
自动求和 =
[] (float x,float y) { 返回 x+y; };
```

现在，你可以通过编写一个玩具数值库来练习这个。

练习 13.1。 做练习 47.10 的零查找项目。

13.1 Lambdaexpressionsasfunction argument

在上面，当我们把一个 lambda 表达式赋值给一个变量时，我们使用了 auto 作为类型。这样做的原因是每个 lambda 表达式都有自己的唯一类型，该类型是动态生成的。现在，如果我们想把那个变量传递给一个函数，就会有问题。

假设我们想把一个 lambda 表达式传递给一个函数：

```
int main() {
    somefun( [] (int i) { cout << i+1; } );
```

我们使用什么类型作为函数参数？

```
void somefun( /* 我们在给出什么类型？ */ f ) {f(5);}
```

由于 lambda 表达式的类型是动态生成的，我们无法在函数头中指定该类型。

解决方法是使用函数 头：

```
#include <functional>
using std::function;
```

有了这个，你可以通过它们的签名声明参数。

在以下示例中，我们编写一个函数 *apply_to_5*，它

- 接受一个函数 *f*，并且
- 应用于 5。

我们将带有 lambda 表达式作为参数的函数称为 *apply_* 到 *_5* 函数 ：

代码：

```
// /lambdaex.cpp
void apply_to_5
    ( 函数< void(int) >f )       {
  f(5);
}
  /* ... */
  将_应用于_5
    [] (int i) {
      cout <<"Int: "      << i << '\n'; } );
```

输出[函数] **lambdapass:**

Int: 5

使用此扩展您之前开始的数值库。

练习 13.2. 做练习 47.11 零查找项目的。

13.1.1 类的 Lambda 成员

lambda 表达式具有动态生成的类型，这也使得它难以存储在对象中。为此，我们再次使用 *std::function*。

在以下示例中，我们创建一个类 *SelectedInts*，它在构造函数中接受一个布尔函数：一个对象将只包含满足该函数的整数。

一组整数，以及一个测试哪些可以被接受的测试：

```
// /lambdafun.cpp
#include <functional>
using std::function;
    /* ... */
类 <selectedints>{}
私有: <t>
  vec or<int> bag;
  函数 <      bool(int) > selector;
public:
  SelectedInts(f ti
        unc  on< bool(int) > f ) {
    selector = f; };
  void add(int i) {
    if (selector(i))
      bag.push(_ back(i);
  };
  int size() {
    返回包。大小 ();};
  std::s  tring string() {
    std::::st  ring s;
    for ( int i : bag )
      {s}+= {to}_ 字符串 {(i)}+" ";</
    返回 s;
  };
};
```

我们使用上述类来构造对象，如下所示：

我们读取一个整数。除数

- 并只接受能被该数字整除的整数进入我们的对象。

为此，我们编写一个 lambda 表达式 *is_* 可被整除 的函数

- 捕获除数，然后
- 以（其唯一的）整数作为参数，
- 返回该参数是否可被整除。

代码:

```
1 /// lambdafun.c    cout <<" 给出一个除数: ";
2
3   cin >>divisor; cout << '\n';t << " i " <<
4   cou       .. us ng          v sor
5         << '\n';
6   auto is_divisible =
7     [除数] (int i) -> bool       {
8       返回 i% 除数 ==0; };
9   SelectedInts 多倍数 ( 是 _ 可被整除 );
10  for (int i=1; i<50; ++i)
11    倍数 . 加 (i);
```

输出

```
[ 函数 ] lambda 函数
给出一个除数 :i
.. us ng 7
7 的倍数 :
7 14 21 28 35 42 49
```

13.2 捕获

一个 捕获 是一种将变量 ‘ 嵌入 ’ 到函数中的方法。假设我们想要一个函数来增加其输入，并且增加量是在我们定义函数时设置的。

增量函数:

- 标量输入，标量输出;
- 增加量已经通过捕获固定。

Code:

```
1// /lambdacapture.cpp
2   int one=1;
3   auto 增量_通过_1     =
4     [一] (int input )      -> int {
5       返回输入+一个;
6   };
7   cout <<increment_by_1 (5)        << '\n';
8   cout << increment_by_1 (12)   << '\n';
9   cout <<increment_by_1 (25)        << \n;
```

输出[f]
unc lambdavalue:

```
6
13
26
```

练习 13.3。 编写一个程序，

- 读取一个 浮点数因子；
- 定义一个函数乘以，它将输入乘以该因子。

您可以捕获多个变量。显式捕获变量是通过逗号分隔的列表完成的。

示例：乘以分数。

```
int d=2,n=3;
times_fraction = [d,n] (int i) ->int {
    return (i*d)/n;
}
```

练习 13.4。

- 设置两个变量

```
float low = .5, high = 1.5;
```

- 定义一个单变量函数，该函数测试该变量是否在低，高之间。（提示：该函数的签名是什么？输入参数是什么？返回结果是什么？）

在您正在开发的数值库中再次使用它。

Exercise 13.5. Do exercises 47.12 and 47.13 of the zero-finding project.

13.2.1 按引用捕获

通常，捕获的变量按值复制。

尝试更改捕获的变量甚至无法编译：

```
auto f = // WRONG DOES NOT COMPILE
    [x] ( float &y ) -> void {
        x *= 2; y += x; };
```

如果你确实想更改捕获的参数，请按引用传递：

Code:

```
1 // /lambdacapture.cpp
2   int stride = 1;
3   auto more_and_more =
4     [&stride] ( int input ) -> void {
5       cout << input << "=>" <<
      input+stride << '\n';
6       ++stride;
7   };
8   more_and_more(5);
9   more_and_more(6);
10  more_and_more(7);
11  more_and_more(8);
12  more_and_more(9);
13  cout << "stride is now: " << stride
      << '\n';
```

Output
[func] lambdareference:

```
5=>6
6=>8
7=>10
8=>12
9=>14
stride is now: 6
```

例如，如果你正在进行某种归约操作，通过引用捕获可以很有用。捕获就是归约变量，而要归约的数字作为函数参数传递给 lambda 表达式。

在这个例子中，我们计算输入值在某个函数 `f` 下测试为真的数量：

通过引用捕获变量以便你可以更新它：

```
int count=0; auto count if f=_ _[&
count](int i) { if (f(i)) count++; } for ( int i :
int_data ) count_if_f(i); cout << "We
counted: "<< count;
```

(See the 算法 头部，第 14.3 节)

13.2.2 捕获 'this'

除了捕获特定变量（无论是否通过引用），如你所见，你还可以捕获 lambda 的整个环境。为此，以下简写形式存在：

```
[=] () {} // 通过值捕获所有内容[&] () {} // 通过引
用捕获所有内容
```

从 C++20 开始，通过值隐式捕获 `this` 已弃用：编写

```
[=] (params) { /* ... */ }
```

会通过值捕获 `this` ，但请注意 `this` 是一个指针，其成员实际上是通过引用捕获的。因此，应该编写以下之一

```
[=,this] (params} { /* ... */ );
[=,*this] (params} { /* ... */ );
```

分别通过引用和值捕获成员。

13.3 更多

13.3.1 使 lambda 具有状态

让我们考虑 lambda 表达式和可变状态的问题，这意味着一个变量被更新多次。

一个简单的例子是进行计数缩减：有多少项满足某个测试。在以下示例中，(这个使用的是 *for_* 每个 算法；章节 14.3.1) 计数的项作为参数传递，而计数通过引用捕获。

代码：

```
// /printeach.cppvec
    tor<int> moreints{8,9,10,11,12};
int count{0};
for each_
  ( 更多整数。开始 ()，更多整数。结束 ()，
    [&count] (int x) {
      if (x%2==0)
        ++count;
    } );
cout << "number of even: " << count
  << '\n';
```

输出
STL 计数每个：

```
偶数的数量 ： 3
```

如果那个计数在 lambda 表达式的调用环境中并非真正需要，我们能否将其设为内部状态呢？

Lambda 表达式通常是状态无的，意味着捕获是通过值捕获，实际上 `const`：

代码：

```
// /mutable.cppfl t 2
    oa x = = 3
     ,y ;auto f = [x] (          > void {
  float & y) -it
    n xx = x* ; 2 y = + xx;   };
  f(y);t
  cou << y << '\n';
  f(y);t
  cou << y << '\n';
```

输出
[func] 非可变：

```
7
11
```

你可以使捕获非 -`const`，从而使 lambda 表达式具有状态，使用 `mutable` 关键字：

代码：

```
1 // /mutable.cpp
2     float x = 2, y = 3;
3     auto f = [x] ( float &y ) mutable
      -> void {
4       x *= 2;y += x; }
5     f(y);
6     cout << y<< ' \n' ;
7     f(y); t <
8     cou   < y << '\n';
```

输出
[函数] 是可变的

```
7
15
```

这里有一个很实用的应用：打印一个由逗号分隔的数字列表，但末尾不带逗号：

代码：

```
1 // /lambdaexch.cpp
2   ve构造函数 x{1,2,3,4,5};
3   auto printdigit=
4     [开始 =true]      《C++17/Fortran2008 科学编程艺术，第 3 卷》第 13.3.1 节：使 lambda 具有状态
      string{
5       if (start) {
6         开始 = false;
7         返回到字符串 (xx);
8       } else
9         retru n ","+to_string(xx);
10    };
11  for ( auto xx : x )
12    cout<< printdigit(xx);
13  cout << '\n';
```

Output
[func] lambda 交换：

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

13.3.2 泛型 lambda

The `auto` 关键字可用于泛型 *lambda*:

```
auto compare = [] (auto a, auto b) {return a<b;};
```

此处返回类型明确，但输入类型是泛型的。这类似于使用模板函数：编译器会根据需要实例化表达式。

13.3.3 算法

The 算法 头包含一些自然使用 lambda 的函数。例如， 任何 _ 的都可以测试向量中的任何元素是否满足条件。然后，您可以使用 lambda 来指定测试条件的 *bool* 函数。

这使用了我们尚未讨论的机制，因此我们将其推迟到 14.3 节。

13.3.4 C 风格的函数指针

C 语言有一种 —— 有点令人困惑 —— 的函数指针表示法。如果您需要与使用它们的代码交互，可以在一定程度上使用 lambda 函数：无捕获的 lambda

可以转换为函数指针。

Code:

```
1 // /lambdacptr.cpp
2 int cfun_add1( int i ) {
3   return i+1; };
4 int apply_to_5( int(*f)(int) ) {
5   return f(5); };
6 //codesnippet  end
7     /* ... */
8   auto lambda_add1 =
9     [] (int i) { return i+1; };
10  cout << "C ptr: "
11       << apply_to_5(&cfun_add1)
12       << '\n';
13  cout << "Lambda: "
14       << apply_to_5(lambda_add1)
15       << '\n';
```

Output
[func] lambdacptr:

```
C ptr: 6
Lambda: 6
```

第 14 章

迭代器、算法、范围

你已经看到了如何遍历一个向量

- 通过索引循环遍历索引，并且
- 使用基于范围的循环遍历值。

还有一种方法，实际上是基于范围循环的基本机制。为此，你需要认识到遍历向量等对象并不是简单地保持一个计数器来告诉你当前的位置，并在需要时获取该元素。

许多 C++ 类都有一个迭代器 子类，它提供了关于 “ 你在哪里 ” 以及在那里可以找到什么的正式描述。有迭代器意味着你可以遍历没有显式索引计数的结构，而且还有许多其他的便利之处。

从某种比喻意义上讲，迭代器 是一个指向向量元素的指针：它指示容器（如向量）中的一个位置。以下是一些迭代器与索引相似的方面：

- 可迭代容器有一个 *begin* 和一个 *end* 迭代器。
- 结束迭代器 “ 指向 ” 最后一个元素之后的位置。
- ‘*’ 星号运算符给出迭代器指向的元素。
- 你可以对它们进行递增和递减（对于某些容器）。

我们从不涉及迭代器的范围讨论开始，然后进入更通用和基本的机制。

14.1 范围

C++20 标准包含一个 范围 头文件，它将可迭代对象泛化为所谓的流，这些流可以通过管道连接。

我们需要介绍两个新的概念。

一个 范围 是一个可迭代对象。C++17 之前的 STL 容器是范围，但一些新的范围已经被添加了。

首先，范围提供了一个清晰的语法：

```
vector data{2,3,1};
ranges::sort(data);
```

```
或 // /sumsquare.cppvector<float> 元素
{.5f,1.f,1.5f};auto元素之和 =_ _
rng::accumulate( 元素, 0.f);cout<< “元素之和:”
<< sum_ 的_元素 <<'\n';
```

In the examples to follow we will write

```
rng::transform
```

and such, where

```
#include <ranges>命名空间
rng = std::ranges;
```

然而，一些功能来自 *ranges-v3* 库 (https://github.com/ericniebler/range-v3)，在这种情况下

```
#include <range/v3/all.hpp>
命名空间 rng = 范围;
```

14.1.1 Views

A 视图 在某种程度上类似于一个范围，因为你可以遍历它。不同之处在于，与例如一个向量不同，视图不是一个完全形成的对象：它是一个范围的某种转换。

您通常会编写类似的内容：

```
yourcontainer | someview
```

其结果与原始容器一样可迭代。‘视图’ 通常是一个范围适配器，来自视图 命名空间。例如：

```
myvector | 视图::drop(5)
```

这就像你省略了向量中的前 5 个元素一样。

视图不拥有任何数据，你在其中查看的任何元素都会在你迭代它时形成。这有时被称为 惰性求值 或 惰性执行。换句话说，它的元素是在迭代视图时被请求时构造的。

视图是可组合的：你可以取一个视图，并将其管道到另一个视图中。如果你需要结果对象而不是作为流元素，你可以调用

```
auto newvector = myvector | views::drop(5) | to<vector<int>>();
```

视图的两个简单示例：

1. 由 `transform` 形成的视图，它按顺序将函数应用于范围内的每个元素或视图；

2. 由 *filter* 形成的，它只产生满足某些布尔测试的元素。

代码：

```
// /filtertransform.cpp
    vector<int> v{ 1,2,3,4,5,6 };
    cout<< "原始向量: "t _  _ti ( )
         << vec or as s r ng v  << '\n';
    auto times_two = v
      | rng::views::transform( [] (int
    i) {
          返回 2*i;});
    cout<< "Times two: ";
    for (auto c : times_two)
      cout << c << " "; cout << '\n';
    自动 超过五 = 倍 _              _
      | rng:: 视图:: 过滤器 ( [](int i){
          返回 i>5;});
    cout<< "超过五 :";
    for （自动 c : 超过 _ 五 ）
      cout << c << " "; cout << '\n';
```

输出 [范围] ft1:

```
原始      vector: 1, 2, 3, 4,
    5, 6,
乘以二 : 2, 4, 6, 8, 10,
    12,
超过五 : 6, 8, 10, 12,
```

接下来说明流的组成：

代码：

```
// /filtertransform.cpp
    vector<int> v{ 1,2,3,4,5,6 };
    cout<< "原始向量: "i_  _
         << vector as str ng(v) << '\n';
    auto times_two_over_five = v
      | rng::views::transform( [] (int
    i) {
          返回 2*i;})
      | rng::views::filter(int i) {
          返回 i>5;});
```

输出 [范围] ft2:

```
原始      vector: 1, 2, 3, 4,
    5, 6,
乘以五分之一 : 6, 8,
    10, 12,
```

让我们练习这个容器的管道和视图。

练习 14.1. 创建一个包含正数和负数的向量。使用范围来计算正数的最小平方根。

1. Start with a vector of numbers;
2. Make an iteratable view containing just the square roots of the positive numbers;
3. Find the minimum of these roots.

其他可用的操作：

- 丢弃初始元素：`std::views::drop`
- 反转一个向量：`std::views::reverse`

在 C++23 中添加了一些更多的视图。我们只展示一些重要的视图 s.

将范围组合在一起使用 `ranges::views::zip`，得到一个元组：

Code:

```
1 // /zip.cpp
2     vector a { 10, 20, 30, 40, 50 };
3     vector<string> b { "one", "two",
      "three", "four" };
4
5     // zip in c++23, not yet in gcc12
6     for (const auto& [num, name] :
      rng::views::zip(a, b))
7       cout << fmt::format("{} -> {}\n",
      name, num);
```

Output

[range] zip:

```
one -> 10
two -> 20
three -> 30
four -> 40
```

Return adjacent elements with *ranges::views::adjacent*, giving a tuple.

```
// /adjacent.cpp
  for ( auto [ lt,md,rt ] : values | views::adjacent<3> )
    cout << fixed
         << setw(3) << lt << ","
         << setw(3) << md << ","
         << setw(3) << rt << '\n';
```

在我的编译器中尚未可用，或在使用范围 *-v3* 库中。

14.1.2 示例：平方和

为了计算向量元素的平方和，我们可以使用转换方法来构造一个包含这些平方的 ‘惰性容器’。但是，C++20 没有 *numeric* 头文件中算法的范围版本，例如 *accumulate*。这在 C++23 中已修复。

```
// /sumsquare.cpp
    vector<float> elements{.5f,1.f,1.5f};
    auto squares =
      rng::views::transform(elements, [] (auto e) { return e*e; } );
    auto sumsq =
      rng::accumulate( squares, 0.f );
    cout << "Sum of squares: " << sumsq << '\n';
```

14.1.3 无限序列

由于视图是惰性构造的，因此可以有一个无限对象 —— 只要你不去请求它的最后一个元素。

在以下示例中，我们创建一个视图 偶数 _ 数字，它 “ 包含 ” 所有偶数，然后我们只打印其中前十个：

Code:

```
1 // /infinite.cpp
2   auto even_numbers =
3     rng::views::iota(0)
4     | rng::views::filter
5         ( [] ( auto n ) -> bool {
      return n%2==0; } );
6   for ( auto n : even_numbers |
      rng::views::take(10) )
7     cout << n << '\n';
8   //codesnippet infeven
9     /* ... */
10
11  return 0;
12 }
```

Output
[range] infinite:

```
0
2
4
6
8
10
12
14
16
18
```

在这里我们使用了 *iota* 的范围版本，在仅指定下限的变体中。

14.2 Iterators

你上面看到的大多数算法都有一个使用迭代器的旧语法。

```
vector data{2,3,1};// 迭代器语法:

sort(
begin(data),end(data));// 范围
语法 ranges::sort(data);
```

The *begin*/ *end* 函数提供了一些类似于指向容器开始和结束的指针的东西。

技术上它们是迭代器对象，它们是容器类的子类。

14.2.1 使用迭代器

容器类有一个子类 迭代器 ，可用于遍历容器的所有元素。

迭代器可以在严格遍历之外使用。你可以将迭代器视为一种 “ 指向容器的指针 ”，并且可以移动它。

让我们看看使用 *begin* 和 *end* 迭代器的例子。在以下示例中：

- 我们首先将 *begin* 和 *end* 迭代器赋值给变量；begin 迭代器指向第一个元素，但 end 迭代器指向最后一个元素之后；
- 给定一个迭代器，你可以通过对其应用 “ 星号 ” 运算符来获取相应元素的价值；
- 可以对迭代器应用一种 ‘ 指针算术 ’ 。

独立于循环使用：

代码:

```
1 // /iter.cpp
2     vector<int> v{1,3,5,7};
3     auto pointer = v.begin();
4     cout << "we start at "
5          << *pointer << '\n';
6     ++pointer;
7     cout << "after increment: "
8          << *pointer << '\n';
9
10    pointer = v.end();
11    cout << "end is not a valid element: "
12         << *pointer << '\n';
13    pointer--;
14    cout << "last element: "
15         << *pointer << '\n';
```

Output

[stl] iter:

```
we start at 1
after increment: 3
end is not a valid element: 0
last element: 7
```

请注意，星号表示法是迭代器对象上的一元星号运算符，而不是指针解引用：

```
// /iter.cppvector<int>
vec{11,22,33,44,55,66};autosecond =
vec.begin(); ++second;cout<< "解引用
second: "<< *second<< '\n';// DOES NOT
COMPILE // 迭代器不是类型 -star: // int
*subarray = second;
```

范围和迭代器代码的等价性:

The range code

```
vector<int> myvector(20);
for ( auto copy_of_int : myvector )
  s += copy_of_int;
```

实际上是以下简写:

```
for
 ( std::vector<int>::iterator
    it=myvector. 开始 ();i !
     t =myvector.end() ; ++it ) {
      int co   of int= *it;
          py s+= copy_of_
      int ; }
```

范围迭代器可用于任何可迭代对象：1！
vector, map, your own c asses

另一个迭代器的例子是从向量中获取最后一个元素：我们首先通过 `back` 方法获取该值。

Use `back` to get the value of the last element:

Code:

```
// /vectorend.cpp
    vector<int> mydata(5,2);
    mydata.push_back(25);

    cout<< mydata.size()
        << '\n';
    cout<< mydata.back()
        << '\n';
```

输出

```
array vectorend:
6
35
```

接下来我们使用迭代器机制。*end* 迭代器指向数据的 ‘末尾’，所以我们将其向左移动并获取其值。

将迭代器设置为最后一个元素并 ‘解引用’：

代码:

```
// /vectorend.cpp
    vector<int> mydata(5,2);
    mydata.push_back(35);
    cout << mydata.size() << '\n';
    cout << *( --mydata.end() ) << '\n';
```

输出[数组]

```
vectorenditerator:
6
35
```

14.2.2 为什么还有迭代器，为什么不

基于范围的算法比基于迭代器的算法更好的一个原因是，迭代器版本容易发生事故：

```
排序 ( 开始 (data1), 结束 (data2)): // 哎哟
```

有些事情你可以用迭代器做，但用基于范围的迭代器却不能做。

反向迭代不能使用基于范围的语法完成。

使用反向迭代器的一般语法：*rbegin*, *rend*.

14.2.3 形成子数组

迭代器可用于构建一个 *vector*。这可以例如用于创建一个 *subvector*。在 simplest case 中，你会使用 begin/end 迭代器复制一个 vector：

```
vector<int> 子( othervec.begin(),othervec.end());
```

注意子数组是作为原始元素的副本形成的。Vectors 完全 ‘拥有’ 它们的元素。对于非拥有子数组，你需要 span; section 10.9.6.

一些更多示例。我们形成一个子数组：

Code:

```
// /iter.cpp
    vector<int> vec{11,22,33,44,55,66};
    auto second = vec.begin(); ++second;
    auto before = vec.end(); before--;
    vector<int>
    sub(vec.data()+1,vec.data()+vec.size()-1);
    cout << "no first and last: ";
    for ( auto i : sub ) cout << i <<
    ", ";
    cout << '\n';
    /* ... */
    vector<int> vec{11,22,33,44,55,66};
    auto second = vec.begin(); ++second;
    auto before = vec.end(); before--;
    //    vector<int>
    sub(second,before);
    vector<int> sub;
    sub.assign(second,before);
    cout << "vector at " <<
    (long)vec.data() << '\n';
    cout << "sub at " <<
    (long)sub.data() << '\n';

    cout << "no first and last: ";
    for ( auto i : sub ) cout << i <<
    ", ";
    cout << '\n';
    vec.at(1) = 222;
    cout << "did we get a change in the
    sub vector? " << sub.at(0) << '\n';
```

Output
[iter] subvectorcopy:

```
no first and last: 22, 33,
    44, 55,
```

为了证明子向量确实是一个新对象，而不是原始向量的子集：

Code:

```
// /iter.cpp
    vec.at(1) = 222;
    cout << "did we get a change in the
    sub vector? "
         << sub.at(0) << '\n';
```

Output
[iter] subvectornew:

```
did we get a change in the
    sub vector? 22
```

14.2.4 通过迭代器进行向量操作

你已经看到，可以使用 *push_back* 方法通过单个元素扩展向量的长度。

使用迭代器可以进行其他操作，例如复制、删除和插入 .

首先我们展示 *copy* 的使用，它需要在一个容器中提供两个迭代器来定义要复制的范围，以及一个目标容器中的迭代器，该迭代器可以是源容器本身。复制操作会覆盖目标中的元素，但没有边界检查，因此请确保有足够的空间。

将一个容器的 begin/end 范围复制
到另一个容器的迭代器 ::

代码：

14.2.4 迭代器实现的向量运算

```
2       vecto r<int>  counts{1,2,3,4};
3       vector<int>  已复制(5);
4       复制( 计数.开始(), 计数.结束(),
5             已复制.开始()+1 );
6       cout<< 已复制[0]<<
7       " "<<     ,       copied[1]
8            << ".." << copied[4] << '\n';
```

输出
[迭代器复制：

0, 1..4

(无边界检查，请小心！)

删除操作 `erase` 需要两个迭代器，定义要删除的范围的包含下界和排除上界。

Erase from start to before-end:

代码：

14.2.4 通过迭代器进行向量运算

```
3       vect  r<int> counts{1,2,3,4,5,6};o
        r<int>::iterator second=
        counts.()begin+1;
4       auto 第四 =  第二 +2;
5       counts.erase(second,fourth);
6       cout<< counts[0]
7            << ","<<counts[1] << ' \n' ;
```

输出
[迭代器] erase2:

1,4

（同时删除单个元素而不使用结束迭代器。）

插入操作需要一个目标迭代器，插入操作将在此迭代器之后进行，以及两个迭代器，用于指定将要插入的范围。这将扩展目标容器的 sizeof。

在迭代器处插入：值、单个迭代器或范围：

代码：

```
1// /iter.cpp2v
3         向量<int> counts{1,2,3,4,5,6},
            zeros{0,0};
4       auto after_one=zeros.+1begin()();
5       zeros.insert
6          after_one,
7               counts.() +1,
8               counts.()begin+3 );
9       cout << zeros[0] << ","
10           << zeros[1] << ","
11           << zeros[2] << ","
12           << zeros[3]
13           << '\n';
```

输出
[迭代器] 插入 2:

0,2,3,0

14.2.4.1 索引和迭代

返回数组元素或位置的函数现在返回迭代器。例如：

- `find` 返回一个指向第一个等于我们查找值的元素的迭代器；
- `max_element`返回一个指向具有最大值的元素的迭代器。

基于范围的索引的一个论点是，如果我们不需要索引，我们可以得到简单的语法。是否可以在使用迭代器的同时得到索引？是的，这就是函数 `distance` 的作用。

Find 'index' by getting the distance between two iterators:

代码：

```
1// /distance.cppt
2   vec 或 int> 数字 {1,3,5,7,9};
3   自动 it= 数字。开始 ();
4   while( it!= 数字。结束 () ) {
5      auto d = distance ( numbers . begin (), it );
6      cout << "At distance " << d
7           << " ": <<*<<'\n';
8      ++ 它;
9   }
```

Output
[循环] 距离：

```
在距离 0: 1
在距离 1: 3
在距离 2: 5
At 距离 3: 7
At 距离 4: 9
```

练习 14.2. 使用上述向量方法来返回，给定一个 `std::vector<float>`, 其最大元素的整数索引。

14.3 使用迭代器的算法

许多关于数组的简单算法，例如测试 'thereis' 或 'for all'，不再需要用 C 语言手动编写 ++。现在，它们可以通过 std::algorithm 库中的单个函数来完成。这个库包含 'C 程序可以用来对容器和其他序列执行算法操作的组件 '。

因此，即使你已经在前面学习了如何手动编写特定算法，你也应该学习以下算法，或者至少知道这些算法的存在。这就是新手程序员和工业级（如果需要更好的术语的话）程序员之间的区别。

14.3.1 测试 Any/all

首先，我们来看一些将谓词应用于元素的算法。

- 测试是否任何元素满足条件： `any_of`。请注意， `std::any_of` 和 `std::ranges::any_of`都存在。接下来的两个也是如此 ._
- Test if all elements satisfy a condition: `all of`.
- Test if no elements satisfy a condition: `none_of`.
- Apply an operation to all elements: `for_each`.

函数应用的对象不是直接指定的；相反，你必须指定一个起始和结束迭代器。

（参见第 13 章了解 lambda 表达式的使用。）

作为应用谓词的一个例子，我们来看几个使用 *any_of* 的例子。这会根据谓词是否对容器的任何元素为真而返回 true 或 false；这使用 短路求值。

Reduction with boolean result:
See if any element satisfies a test

代码:

```
1 // /eachr.cpp
2 #include <ranges>
3 #include <算法>
4     /* ... */
5     vector<int> ints{1,2,3,4,5,7,8,13,14};
6     std::ranges::for_each
7       ( ints,
8         [] ( int i ) -> void {
9           cout << i<< ' \n' ;
10        }
11        );
```

输出
[迭代] 每个

```
1
2
3
4
5
7
8
13
14
```

(为什么不用 *accumulate* reduction?)

这里有一个使用 *any_of* 来查找某个元素是否出现在向量中的示例:

Code:

```
1 // /eachr.cpp
2     vector<int> ints{1,2,3,4,5,7,8,13,14};
3     bool there_was_an_8 =
4       std::ranges::any_of
5         ( ints,
6           [] (int i) -> bool {
7             return i==8;
8           }
9           );
10    cout << "There was an 8: " <<
      boolalpha << there_was_an_8 << '\n';
```

输出
[迭代] 任意 r:

有一个 8: 真实

捕获要测试的值给出:

Code:

```
1 // /eachr.cpp
2     vector<int>
      ints{1,2,3,4,5,7,8,13,14};
3     int tofind = 8;
4     bool there_was_an_8 =
5       std::ranges::any_of
6       ( ints,
7         [tofind] ( int i ) -> bool {
8           return i==tofind;
9         }
10        );
11    cout << "There was an 8: " <<
      boolalpha << there_was_an_8 << '\n';
```

Output
[iter] anyc:

```
There was an 8: true
```

注意 19 前面的示例依赖于 *C++20* 范围。使用迭代器时，它们看起来像这样：

代码：

```
1 // /each.cpp
2     vector<int>
      ints{2,3,4,5,7,8,13,14,15};
3     bool有一个 8 =_  _ _
4       any_of( ints.begin(),ints.end(),
5                 [] ( int i ) -> bool {
6                   return i==8;
7                 }
8
9     cout << "ere was an 8: " <<
      boolalpha << there_was_an_8 << '\n';
```

Output
[迭代] 任意

```
存在一个 8: true
```

代码：

```
1 ///each.cpp
2 #include <algorithm>
3     /* ... */
4     vect 或 int
      ints{2,3,4,5,7,8,13,14,15};
5     for_each( ints.begin(),ints.end(),
6                 [] ( int i ) -> void {
7                   cout<<i<< '\n';
8                 }
9                 );
```

输出
[iter] *each:*

```
2
3
4
5
7
8
13
14
15
```

14.3.2 应用于每个

The `for_each` 算法将一个函数应用于容器的每个元素。与之前的算法不同，这个可以改变元素。

为了介绍语法，我们来看一个无意义的示例，输出每个元素：

代码：

```
// /printeach.cpp
#include <algorithm>
using std::for_each;
/* ... */
vector<int> 整数{3,4,5,6,7};
for_each
  ( 整数.开始(), 整数.结束(),
    [] (整数x) -> 空 {
      if (x%2==0)
        cout << x<< '\n';
    } );
```

输出 std] 打印每个

```
4
6
```

将某物应用于每个数组元素：

代码：

14.3.2 应用于每个

```
#include <ranges>
#include <algorithm>
/* ... */
vector<int> ints{1,2,3,4,5,7,8,13,14};
std::ranges::for_each
  ints,
    [] ( int i ) -> void {
      cout << i<< '\n';
    }
    );
```

输出 [迭代] 每个

```
1
2
3
4
5
7
8
13
14
```

练习 14.3. 使用 `for each` 来对向量的元素求和 ._

提示：问题在于如何处理求和变量。不要使用全局变量！

通过引用捕获，以使用数组元素进行更新。

代码：

```
1// /each.cpp2 t<i
n   vec or   t>
    ints{2,3,4,5,7,8,13,14,15};
3   int sum=0;
4   for_each( int.begin(),int.end(),
5             [&sum] ( int i ) -> void {
6               sum+= i;
7             }
8             );
9   cout << "Sum = " << sum <<<style id='5'
                          >'\n';
```

输出 [迭代] 每个

```
2
3
4
5
7
8
13
14
15
```

14.3.3 迭代器结果

某些算法不会产生一个值，而是产生一个指向该值位置的迭代器。例如：`min`_`element` 接受一个开始和结束迭代器，并返回位于两者之间的迭代器。

找到最小元素。为了找到实际值，我们需要 “ 解引用 ” 迭代器：

```
// /minelt.cppvector<float> 元素
{.5f,1.f,1.5f};automin iter = std::min
element_                    _(元素.begin(), 元
素 .end());cout << "最小值:"<< *min_iter
<< '\n';
```

类似地 *max_ 元素*。

14.3.4 映射

The *transform* 算法将一个函数应用于每个容器元素，就地修改它：

```
std::transform( vec, vec.begin(), [] (int i) { return i*i; } );
```

14.3.5 约简

数值 约简 可以使用 *accumulate* 在 *numeric* 头文件中通过迭代器应用。如果没有指定约简运算符，则执行 求和约简 。

默认是求和约简：

代码：

```
1// /reduce.cpp
2 #include    <数值>
3 使用 std:: 累加 ;4
      /* ... */
5     vector<int> v{1,3,5,7};
6     auto 第一个 = v. 开始 ();
7     auto 最后一个 = v. 结束 ();
8     auto 求和 = 累加 ( 第一个 , 最后一个 ,0);"
9     "cout <<\' sum:    << sum <<    n ;
```

输出

STLaccumulate:

< 样式
id='1'> 求和
</ 样式 >:
16

其他可作为 *reduction operator* 使用的二进制 *arithmetic operators* 可以在 *functional* 中找到：

- 加， 减， 乘， 除，
- 仅整数： 模数
- 布尔：逻辑 _ 和 , 逻辑 _ 或

此标题还包含一元否定 运算符，当然不能用于约简。

作为显式指定约简运算符的示例：

```
= std::accumulate(x.begin(),
x.end(), 1, f,
std::multiplies<float>());
```

注意：

- 该运算符是模板化的，并且它后面跟着括号以成为函数对象，而不是类；

- accumulate 函数是模板化的，它从 init 值中获取类型。因此，在上面的示例中，1 的值会使这个操作变为整数操作。

使用 lambda 函数（第 13 章）我们可以得到更复杂的效果。

提供乘法运算符：

代码：

```
1 //pp/使用std::multiplies;3
/
       * ... */
4      vector<int> v{1,3,5,7};
5      自动 . 开始 ();
6      auto last  = v.end();
7      ++ 第一个 ; 最后 --;
8      auto product =
9         accumulate (first, last,2,
10                          乘法 <>()); 导出 :"<<
11     cout << "pro产品                    <<
       \n;
```

输出
[stl] 产品：

```
产品 : 30
```

针对最大缩减的是 最大 _ 元素。这可以在没有比较器（用于数值最大）的情况下调用，或者带有比较器进行一般最大操作。最大和最小算法返回一个迭代器，而不仅仅是最大 / 最小值。

示例：与某个量的最大相对偏差：

```
max_ 元素 (myvalue.begin(),myvalue.end(),[my_sum_ 的 _ 平方 ] (double x, double y) ->bool {
return fabs( (my_sum_ 的 _ 平方 -x)/x ) < fabs( (my_sum_ 的 _ 平方 -y)/y ); } );
```

对于在 accumulate 中使用的更复杂的 lambda 表达式，

- 第一个参数应该是归约类型，
- 第二个参数应该是迭代类型

在以下示例中，我们归约类的成员：

```
// /reduce.cpp
类 x {
public:
  int i, j;
  x() {};
  x(int i,int j) : i(i),j(j) {};
};
```

```
// /reduce.cpp
    std::vector< x > xs(5);
    auto xxx=
      std::accumulate
      (xs.begin(),xs.end(),0,)
        [] ( int init,x x1 ) -> int
    { return x1.i+init; }
      );
```

备注 20 *The* accumulate 算法没有并行版本，具有 执行策略。有关此内容，请参阅 std::reduce 在 numeric 头文件中。

14.3.6 排序

The 算法 头也有一个函数 *sort*。

使用迭代器，您可以轻松地将此应用于诸如向量之类的事物：

sort(myvec.begin(), myvec.end());

默认使用的比较方式是升序。您可以指定其他比较函数：

sort(myvec.begin(), myvec.end(), [](int i, int j)
{ return i < j; });

或排序 `( people.begin(),people.end(),[] ( const Person& lhs,const Person&rhs ) {return lhs.name <rhs.name; })`

使用迭代器，您还可以执行诸如对向量的部分进行排序等操作：

Code:

```
1 // /sort.cpp
2   vector<int>
      v{3,1,2,4,5,7,9,11,12,8,10};
3   cout << "Original vector: " <<
      vector_as_string(v) << '\n';
4
5   auto v_std(v);
6   std::sort(
      v_std.begin(),v_std.begin()+5 );
7   cout << "Five elements sorts: " <<
      vector_as_string(v_std) << '\n';
```

Output

```
[range] sortit:
Original vector: 3, 1, 2, 4,
    5, 7, 9, 11, 12, 8, 10,
Five elements sorts: 1, 2,
    3, 4, 5, 7, 9, 11, 12, 8,
    10,
```

14.4 并行执行策略

C++17 标准为标准算法添加了 执行策略 概念，描述了算法库中的元素如何并行执行。

对于 执行策略，有三个选择，定义在 执行 头文件中：

- std:: 执行:: 序: 迭代可能不会被并行化，但在评估线程中会不确定地顺序执行。
- std:: 执行:: 并: 迭代可以并行执行，但是不确定的序列；
- std:: 执行:: 并_ 非序 : 迭代可以被并行化、向量化、迁移到线程上。
- std:: 执行:: 非序(自 C++20): 允许迭代向量化。

为适应不确定的评估顺序，引入了基于现有 ‘有序算法’ 的新 ‘无序’ 算法：

- *reduce*: 类似 *accumulate*，但无序，因此可通过一个 *ExecutionPolicy* 并行化；
- *inclusive_scan*: 类似于*partial_sum*;
- *exclusive_scan*: 无有序等价物
- 转换 _ 归约， 转换 _ 包含扫描， 转换 _ 排除扫描。 转换 _ 归约（ 执行策略，第一个，最后一个，初始值， 归约 _ 操作符，转换 _ 操作符 ）；

这些的一些性能测量数据在 MPI/OpenMP 书籍 [10]，第 19.4 节。

14.5 算法分类

(根据 Dietmar K.uhl 在 CppCon 2017 演讲的内容及其版权声明整理。)

14.5.1 非并行算法

14.5.1.1 O(O)(1) 算法

clamp, *destroy_at*, *gcd*, *iter_swap*, *lcm*, *max*, *min*, *minmax*,

14.5.1.2 O($O(\log n)$ 算法

二 _ 分 _ 查找 , 等 _ 范围 , 下 _ 界 , 划分 _ 点 , 堆 _ 调整 , 堆 _ 入栈 , 上 *{v19}* 界 ,

14.5.1.3 堆算法

*make_* 堆 , *sort_* 堆 ,

14.5.1.4 排列算法

is_permutation, *next_permutation*, *prev_permutation*,

14.5.1.5 重叠算法

从 _ 处向后复制，从 _ 处向后移动,

14.5.1.6 重命名算法

accumulate, 部分 _ 和 ,

14.5.1.7 奇特的算法

iota, 样本 , 洗牌 ,

14.5.2 并行算法

14.5.2.1 映射算法

copy, *copy_n*, 销毁 , 销毁 *_n*, 填充 , 填充 *_n*, 循环 *_each*, 循环 *_each_n*, 生成 , 生成 *_n*, 移动 , 替换 , 替换 *_copy*, 替换 *_copy_if*, 替换 *_if*, 反转 , 反转 *_copy*, 交换 _ 范围 , 转换 , 未初始化 *_**,

14.5.2.2 Reduce 算法

相邻 _ 查找 , 所有 _ 的 , 任意 _ 的 , 计数 , 计数 _ 如果 , 相等 , 查找 , 查找 _ 结束 , 查找 _ 第一个 _ 的 , 查找 _ 如果 , 查找 _ 如果 _ 不 , 包含 , 内积 *_*, 是 _ 堆 , 是 _ 堆 _ 直到 , 是 _ 划分的 , 是 _ 排序的 , 是 _ 排序 _ 直到 , 字典序 _ 比较 , 最大 _ 元素 , 最小 _ 元素 , 最小最大 _ 元素 , 不匹配 , 没有 _ 的 , *Reduce*, 搜索 , 搜索 *_n*,

14.5.2.3 扫描算法

*exclusive_* 扫描，*inclusive_* 扫描

14.5.2.4 融合算法

transform_exclusive_scan, *transform_inclusive_scan*, *transform_reduce*,

14.5.2.5 Gather 算法

复制 _ 如果 < 样式 *id='2'*>,</ 样式 > 分区 _ 复制 < 样式 *id='5'*>,</ 样式 > 删除 < 样式 *id='7'*>, </ 样式 > 删除 _ 复制 < 样式 *id='10'*>, </ 样式 > 删除 _ 复制 _ 如果 < 样式 *id='14'*>, </ 样式 > 删除 _ 如果 < 样式 *id='17'*>, </ 样式 > 旋转 < 样式 *id='19'*>, </ 样式 > 唯一 < 样式 *id='21'*>,</ 样式 > 唯一 _ 复制 < 样式 *id='24'*>,

14.5.2.6 特殊算法

相邻差分，就地合并，合并，第 *n* 个元素，部分排序，部分排序复制，划分，旋转，集合差分，集合交集，集合对称差分，集合并集，排序，稳定划分，稳定排序

14.6 高级主题

14.6.1 范围类型

范围的类型:

- `std::ranges::input_` 范围 : 至少向前迭代一次，就像你用 `cin` 接受输入一样。
- `std::ranges::forward_range` : 可以向前迭代，例如使用 plus-plus，多次迭代，就像在 单链表 中一样。
- `std::ranges::bidirectional_range` : 可以双向迭代，例如使用 plus-plus 和 minus-minus。
- `std::ranges::random_access_range` 中的项可以在常数时间内找到，例如使用方括号索引。
- `std::ranges::contiguous_range` : 项在内存中连续存储，使得地址计算成为可能。

14.6.2 Makeyourown iterator

你知道你可以遍历向量对象：

```
int> myvector(20); for ( auto copy_of_int :
myvector ) s+= copy_of_int; for ( auto &r
ef_to_int : myvector ) ref_to_int =s;
```

(许多其他 STL 类也可以像这样进行迭代。)

这不是魔法：任何类都可以进行迭代：一个类 是 可迭代的 ，只要满足一些条件。

该类需要包含：

- 一个方法以原型开始 `iteratableClass iteratableClass::begin()`，该方法返回一个初始状态的对象，我们将称之为 ‘迭代器对象’ ；同样地
- 一个方法结束

  可迭代类可迭代类 ::*end*() }

  该方法返回一个最终状态的对象；此外，您还需要
- 一个增量运算符 `void iteratableClass::operator++()`，该运算符将迭代器对象推进到下一个状态；
- a test`bool` 类 :: 运算符 `!=(const` 类 `&)` to determine whether the iteration can continue;finally
- a 解引用运算符类 :: 运算符 `*()`

  that takes the iterator object and returns its state.

所有这些在 pre-C++11 迭代中都是可见的，其中遍历一个向量的循环看起来像这样：

```
for (auto elt_ptr=vec.begin(); elt_ptr!=vec.end(); ++elt_ptr)
  element = *elt_ptr;
```

Some remarks:

- 这是 C++ 中极少数需要使用星号的地方。但是，你将其应用于迭代器，而不是指针，并且这是一个你应用的运算符。
- 与普通循环一样，结束迭代器指向向量末尾之后的位置。
- 你可以对迭代器进行 指针运算 ，就像你在 `++elt_ptr` 更新循环头部的部分看到的那样。

14.6.2.1 示例 *1*

让我们创建一个名为 abag 的类，它模拟一组整数，并且我们想要枚举它们。为了简化起见，我们将创建一个连续整数的集合：

```
    // /bag.cppclass bag{// 基本数据private:int
first,last;public:bag(intfirst,intlast) :
first(first),last(last) {};
```

当你创建一个迭代器对象时，它将是你要迭代的对象的副本，只是它记得已经搜索了多远：

```
// /bag.cppclass iter {private:: int
seek{0};public:: iter(int i) : seek(i) {}int
value() { return seek; }
```

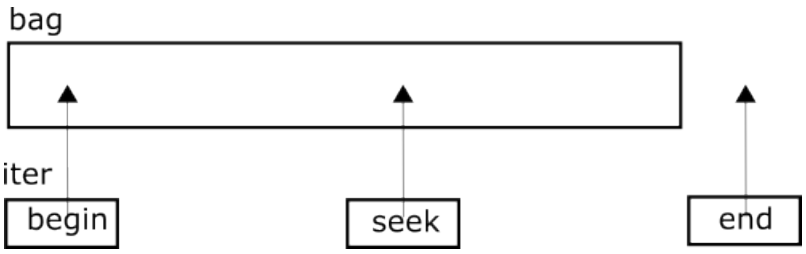


图 14.1：迭代器对象进入 *bag* 对象

begin 方法给出一个包含 seek 参数初始化的包：

```
    // /bag.cpppublic :iter begin() {return
iter(first);};iterend() { return iter(last);};
```

这些例程是公开的，因为它们被客户端代码（隐式地）调用。

终止测试方法在迭代器上调用，将其与 end 对象进行比较：

```
// /bag.cpp
    bool operator!=( const iter &test ) const { return seek!=test.seek; };
    bool operator==( const iter &test ) const { return seek==test.seek; };
```

最后，我们需要 increment 方法和 dereference。两者都访问 seek 成员：

```
// /bag.cpp
```

```
        void operator++() { ++seek; };
        int operator*() { return seek; };
```

We can iterate over our own class:

代码：

```
1// /bag.cpp
2   bag digits<0,9>;
3
4   bool find3{false};
5   for (auto seek : digits)
6       find3 = find3 || <>(style id='3'>(seek==3<>(style id='6'>);
7   cout <<"found 3: " << boolalpha
8         <<find3 << '\n';
9
10  bool find15{false};
11  for (auto seek : digits)
12     查找 15 = 查找 15 ||( 查找 ==15);
13  cout <<"找到 15: " << boolalpha
14        <<find15 << '\n';
```

输出
[循环] 包查找：

找到 3： 是
找到 15： *false*

(对于这个特定的情况，使用 std:: 任何 _ 的)

代码：

```
1 // /bag.cpp
2   // 尚未工作：需要
      迭代器 _ 类别
3   // bool find8 = any_ 的
4   //      ( digits.begin(),digits.end(),
5   //        [=] (bag::iter i) { return
        i.value()==8; } ); / t "f d 8
6   /   cou   <<   oun   : " << boolalpha
7   //        << 查找 8 << '\n';
```

Output
[循环] 袋子：

找到 8： 为 *true*

如果我们给类添加一个方法 has：

// /bag.cppbool has(int tst) {for (auto seek : *this)if (seek==tst)return true;return false; }

我们可以这样称呼它：

// /bag.cppcout "f3: " << digits.has(3) <<’ \n’ ;cout " f15: " << digits.has(15) << ’ \n’ ;

当然，我们也可以不使用基于范围的迭代来编写这个函数，但这个实现特别优雅。

练习 14.4. 你现在可以完成练习 45.24，使用这种机制实现一个素数生成器。

如果你认为你理解 const，请考虑 has 方法在概念上是成本的。但如果添加这个关键字，编译器会抱怨对 *this 的使用，因为它是通过 begin 方法改变的。

练习 14.5. 找到一个方法，使 has 成为一个 const 方法。

14.6.2.2 示例 *2*：迭代器类

回想一下

简写：

```
vector<float> v;
for ( auto e : v )
  ... e ...
```

for:

```
for ( vector<float>::iterator
    e=v.开始 (); e!=v.end(); e++)
  ... *e ...
```

如果我们想要

```
for ( auto e : my_object )
    ... e ...
```

we need an iterator class with methods such as *begin*, *end*, * and ++.

使用迭代器子类遍历一个类

Class:

```
// /iterclass.cpp
class NewVector {
protected:
  // vector data
  int *storage;
  int s;
    /* ... */
public:
  // iterator stuff
  class iter;
  iter begin();
  iter end();

};
```

Main:

```
// /iterclass.cpp
  NewVector v(s);
    /* ... */
  for ( auto e : v )
    cout << e << " ";
```

随机访问迭代器：

// /iterclass.cppNewVector::iter&
operator++();int& operator*();

```
bool operator==( constNewVector::iter &other )c          onst;
bool operator!=( const NewVector::iter &other ) const;
// needed to OpenMP
int operator-(const NewVector::iter& other)                    const;
NewVector::iter& operator = (int a);
```

练习 14.6。 Write the missing iterator methods. Here's something to get you started.

```
// iterclass.c
NewVector::iter{i  t  it
pr va e:  n  *searc er;
    /* ... */
NewVector::iter::iter( int 搜索器 )
  : 搜索器 ( 搜索器 ) {};
新向量 :: 迭代器 NewVector:: 开始 () {
  返回 NewVector:: 迭代器 ( 存储 ); };
NewVector:: 迭代器 NewVector:: 结束 ()      {
  返回 NewVector:: 迭代器 ( 存储 +NewVector::s); };
```

14. 迭代器、算法、范围

第 15 章

参考文献

15.1 参考

本节包含有关引用的更多事实，您已经将其视为参数传递的机制；7.5.2 节 。请先学习那些材料。。

将变量传递给例程会复制值；在例程中，变量是局部的 1.

Code:

```
1 // /localparm.cpp
2 void change_scalar(int i) {
3   i += 1;
4 }
5     /* ... */
6   number = 3;
7   cout << "Number is 3: "
8        << number << '\n';
9   change_scalar(number);
10  cout << "is it still 3? Let's see: "
11       << number << '\n';
```

Output
[func] localparm:

```
Number is 3: 3
is it still 3? Let's see: 3
```

如果您确实想使更改在 调用环境 中可见，请使用引用：

```
// /arraypass.cpp
void change_scalar_by_reference(int &i) { i += 1; }
```

调用程序没有变化。（有些人习惯 C 语言，不喜欢这种方式，因为从函数的使用情况无法看出是按引用传递还是按值传递。）

15.2 按引用传递

如果你使用数学风格的子程序，其中一些值输入，并输出一个新实体，实际上所有输入都可以被复制。这种风格称为函数式编程，有很多优点。例如，它使编译器能够推理你的程序。唯一需要担心的是复制的成本，如果输入的大小非平凡，例如数组。

然而，有时你想修改输入，因此你需要一种访问实际输入对象的方式，而不是复制。这就是引用的用途：允许子程序访问实际输入实体。

使用引用的一个好处是，你不会承担复制的成本。那么，如果你想要这种效率，但你的程序在功能设计上确实很棒呢？那么你可以使用一个 *const* 引用：参数通过引用传递，但你明确表示子程序不会修改它，再次允许编译器优化。

引用使函数参数成为参数的同义词。

```
void f( int &i) { i  += 1; };
intmain() {int i = 2;f(i); //
使其变为 3
```

不复制地传递一个大对象：

```
类 BigDude{
public:
    vector<double> array(5000000);
}

void f(BigDude d) {
  cout <<d. array[0];
};

int main() {
    BigDude big;
    f(big); // whole thing is copied
```

Instead write:

```
void f( BigDude &thing ) { .... };
```

防止更改：

```
void f( const BigDude &thing ) {
    .... };
```

15.3 对类成员的引用

以下是返回类成员的简单方法：

```
类对象 {private:SomeType
thing;public:SomeType
get_thing() {return
thing;};};
```

现在，返回语句会复制 `thing`，这可能就是所需的行为，也可能不是。如果你不需要实际副本，但想要访问实际的数据成员，可以通过 引用 返回成员：

```
SomeType &get_thing() {
  return thing; };
```

现在您有权限访问一个内部（可能是私有！）的成员数据。您可能需要这个权限，但如果您不使用 *const reference*：

Code:

```
1 // /constref.cpp
2 class has_int {
3 private:
4   int mine{1};
5 public:
6   const int& int_to_get() { return
      mine; };
7   int& int_to_set() { return mine; };
8   void inc() { ++mine; };
9 };
10    /* ... */
11  has_int an_int;
12  an_int.inc(); an_int.inc();
      an_int.inc();
13  cout << "Contained int is now: "
14       << an_int.int_to_get() << '\n';
15  /* Compiler error:
      an_int.int_to_get() = 5; */
16  an_int.int_to_set() = 17;
17  cout << "Contained int is now: "
18       << an_int.int_to_get() << '\n';
```

Output
[const] constref:

```
Contained int is now: 4
Contained int is now: 17
```

在上面的例子中，返回引用的函数被用于赋值语句的左侧。如果你将其用于右侧，你将不会得到引用。表达式的结果不能是引用。

让我们再次创建一个类，我们可以从中获取内部引用：

```
// /rhsref.cppclass
myclass{private:int
stored{0};public:myclass(int i) :
stored(i) {}int &data() { return stored; };};
```

现在我们探讨在右侧使用该引用的各种方法：

Code:

```
1 // /rhsref.cpp
2   myclass obj(5);
3   cout << "object data: "
4        << obj.data() << '\n';
5   int dcopy = obj.data();
6   ++dcopy;
7   cout << "object data: "
8        << obj.data() << '\n';
9   int &dref = obj.data();
10  ++dref;
11  cout << "object data: "
12       << obj.data() << '\n';
13  auto dauto = obj.data();
14  ++dauto;
15  cout << "object data: "
16       << obj.data() << '\n';
17  auto &aref = obj.data();
18  ++aref;
19  cout << "object data: "
20       << obj.data() << '\n';
```

Output
[func] rhsref:

```
object data: 5
object data: 5
object data: 6
object data: 6
object data: 7
```

(另一方面，在 `const auto &ref` 之后，引用是不可修改的。这种变体在您希望只读访问且不想承担复制成本时很有用。)

您可以看到，尽管方法 `data` 被定义为返回引用，您仍然需要指示左侧是否为引用。

15.4 对数组成员的引用

您可以定义各种运算符，例如 `+-*/` 算术运算符，以在类上执行自己的实现；参见章节 9.5.7。

您还可以定义括号和方括号运算符，以便使您的对象分别看起来像函数或数组。

这些机制也可以用于提供对对象专有的数组和 / 或向量的安全访问。

假设你有一个包含一个 `int` 数组的对象。你可以通过为该类定义下标（方括号）运算符来返回一个元素：

```
   // /getindex1.cpp 类
vector10{private:int 数组
[10];public:/* ...*/int 运
算符 ()(int i) {return 数组
[i];};int 运算符 [](int i)
{return 数组 [i];};
```

```
};/* ...*/vector10 v;cout <<v(3) << '
\n';cout <<v[2] << '\n';/* 编译错误:
v(3) = -2;*/
```

注意返回 数组 [i] 将返回数组元素的副本，因此无法写入

```
myobject[5] = 6;
```

为此，我们需要返回数组元素的引用：

```
// /getindex2.cppint
operator()(int i) {  return
array[i];}; }/* ... */cout << v <<
'\n';v -= 2;cout << v << '\n';
```

您想要返回引用的原因可能是为了防止返回语句引起的返回结果的副本 在这种情况下，您可能不希望能够修改对象内容，因此可以返回一个 const 引用：

```
// /getindex3.cppconst int&
operator[](int i) {return array[i];};};/
* ...*/cout <<v[2] << '\n';/* 编译错误:
v[2] = -2; */
```

15.5 地址

C 程序员习惯使用取地址符来获取变量的地址，以及在 C++：对象中。C++ 程序员需要注意，取地址符可能会被重新定义：

```
T operator&(){ /* stuff */ };
```

如果你绝对需要地址，可以使用 addressof 函数。

15. 参考文献

第 16 章

指针

指针是一种将实体与变量名间接关联的方式。例如，考虑列表的循环定义：

> 列表由节点组成，其中节点在其 “ 头 ” 中包含一些信息，并且节点有一个 “ 尾 ”，该尾是一个列表。

朴素代码：

```
类
N
o
d
e
{private:int
value;Node
tail;/* ...
*/};
```

这不起作用：将占用无限内存。

间接包含：仅指向尾部：

```
类 节点 { 私有: int 值 ;
PointToNode 尾 ;/
*... */};
```

本章将解释 C++ 智能指针，并给出它们的一些用途。

16.1 指针使用

最初，我们将专注于指向某些类对象的指针。

一个存储单个数字的简单类：

```
    ///
twopoint.cppclass
H
asX{private:double
x;public:}
```

```
    HasX( double x) : x(x) {};
    auto get() { return x; };
    void set(double xx) { x = xx; };
  };
```

使用类对象时，类成员的 ‘点’ 表示法在使用指针时会变成 ‘箭头’ 表示法。如果你有一个具有成员 `y` 的对象 `Obj x`，你可以通过 `x.y` 来访问它；如果你有一个指向此类对象的指针 `x`，你会写成 `X->y`。

- 如果 `x` 是一个具有成员 `y` 的对象：

  `x.y`
- 如果 `xx` 是指向具有成员 `y` 的对象的指针：

  `xx->y`
- 箭头表示法适用于 C 风格指针和新的共享 / 唯一指针。

现在，您不再单独创建对象，而是在一次调用中创建一个指向该对象的指针。

请注意，您不会像许多其他语言那样先创建对象，然后再将其指向一个指针。智能指针的工作方式不同：您在一次调用中创建对象和指向该对象的指针。

```
make_shared<ClassName >( 构造函数参数 );
```

生成的对象是 `shared_ptr<ClassName>` 类型，但您可以避免拼写它，并使用 `auto` 代替。

Object vs pointed-object:

代码：

```
// /pointx.cpp
#include <memory>
using std::make_shared;

   /* ... */
  HasX xobj(5)
  cout <<xobj. .value() << ’ \n’ ;
  xobj. 设置 (6);
  cout <<xobj.{value}() '\n';

  auto xptr = make_shared<HasX>(5);
  cou<< > l() ’ \ va ue n ;
  xptr->set(6);
  cout << xptr->value() <<’ \n’ ;
```

输出

[指针] pointx:

```
5
6
5
6
```

使用智能指针需要在文件的顶部：

#include <memory> using
std::shared_ptr; using
std::make_shared; using
std::unique_ptr; using
std::make_unique;

Why do we use pointers?
Pointers make it possible for two variables to own the same object.

代码：

```
// /two_one.cpp</p><auto><xptr>= <make>_<shared><
<HasX><>(5);
auto yptr = xptr;
cout << xptr->get() <<' \n' ;
yptr->set(6);
cout << xptr->get() <<' \n' ;
```

输出[指针] 两点：

```
5
6
```

指针使用的典型例子是在链表和图数据结构中。参见 66.1.2 节以获取代码和讨论。

练习 16.1。 如果你在做几何项目（章节 46）你现在可以做一些练习 46.16。

16.2 内存泄漏和垃圾回收

C 风格指针的主要问题是存在 内存泄漏 的风险。如果一个指向内存块的指针超出作用域，该内存块不会被返回给操作系统，但也不再可访问。

```
// 将内存附加到 'array' : double *array= new double
[25];// 对 array 进行操作 // 覆盖指针 array= new double
[26];// 首次分配的内存仍然被保留。
```

共享指针和唯一指针没有这个问题：如果它们超出作用域或被覆盖，对象的析构函数会被调用，从而释放任何已分配的内存。

一个示例。

我们需要一个带有构造函数和析构函数跟踪的类：

```
// /ptr1.cppclass thing {public:thing() { cout
<<".. 调用构造函数 \n"; };~thing() { cout <<".. 调用析构函数
\n"; };};
```

为了说明当对象超出作用域时析构函数会被调用，我们暂时展示一个没有任何指针的例子：

代码：

```
1// /ptr0.cpp2cout "
<<"Ou              tside\n";
3   {
4      thing x;
5      cout<< "create done\n";
6   }
7   cout<< "back outside\n";
```

输出

[指针] ptr0:

```
外部
.. calling constructor
创建完成
.. 调用析构函数
返回外部
```

现在我们用指针来做同样的操作，以说明当指针不再指向对象时，对象的析构函数会被调用。我们通过将 `nullptr` 赋值给指针来实现这一点。（这与 C 中的 `NULL` 非常不同：空指针实际上是一个具有类型的对象；参见章节 16.3.6。）

让我们创建一个指针并覆盖它：

代码：

```
1// /ptr1.cpp
2   cout <<"set pointer1"
3        << '\n';
4   auto 东西_ptr1 =
5     make_shared<东西>
6   cout<< "overwr te po nter"
7        << '\n';
8   东西_ptr1 =nullptr;
```

输出

[指针] ptr1:

```
设置指针 1
.. calling constructor
g 覆盖指针
.. 调用析构函数
```

然而，如果指针被复制，就会有指向同一块内存的两个指针，只有当它们都消失或指向其他地方时，对象才会被释放。

代码：

```
1// /ptr2.cpp2
2   cout <<"设置指针 2" << '\n';
3   auto 东西_ptr2 =
4     make_shared<东西>();
5   cout <<“通过拷贝设置指针 3” << '\n';
6
7   auto thin _ptr3 = thin _ptr2;
8
9        << '\n';
10  东西_ptr2 = nullptr;
11  cout<< "overwrite pointer3"
12       << '\n';
13  东西_指针 3 = nullptr;
```

输出

指针 ptr2:

```
设置 pointer2
   .. 通过拷贝调用构造函数设置 r
pointer3
覆盖 pointer2
覆盖 pointer3
.. 调用析构函数
```

- The object counts how many pointers there are:
- ‘reference counting’
- A pointed-to object is deallocated if no one points to it.

注意 21 一个更隐蔽的内存泄漏来源与异常：

```
void f() {
  double *x = new double[50];
  throw("something");
  delete x;
  }
```

由于异常（当然也可能来自嵌套函数调用），`delete` 语句永远不会被执行，因此分配的内存会泄漏。智能指针解决了这个问题：抛出异常会导致作用域结束，因此析构函数会被调用。

16.3 高级主题

16.3.1 获取指向的数据

大多数情况下，通过箭头符号访问指针的目标就足够了。但是，如果你实际上想要对象，可以用 `get` 获取。请注意，这不会给你指向的对象，而是一个传统的指针。

```
X->y;
// 是相同的
X. 获取 >y;
// 是相同的
(*X.get() ).y;
```

代码：

```
1// /pointy.cpp2
auto Y = make_shared<HasY>(5);
cout <<Y->y << '\n';
Y. 获取 >y = 6;
cout << ( *Y ).get() ).y '\n';
```

Output
[指针] 尖：

```
5
6
```

备注 22 这种机制稍微更优，因为它在 `->` 运算符被重载时也能工作。

16.3.2 唯一指针

共享指针编程相对容易，并且它们带来了许多优势，例如自动内存管理。然而，它们比严格必要的开销更大，因为它们有一个 引用计数 机制来支持内存管理。因此，存在一个唯一指针，`唯一_ptr`，用于对象将只会被一个指针 ‘ 拥有 ’ 的情况。

在这种情况下，你可以使用 C 风格的裸指针 进行非拥有引用 ：

```
auto xptr= make_唯一<对象>(); 对象
* tmp_ptr= xptr.get();
```

16.3.3 基类和派生类指针

假设你有基类和派生类：

```
类 A{}; 类 B: public A
{};
```

就像你可以将一个 B 对象赋值给一个 A 变量：

```
B b_ 对象 ; 类 a_ 对象 = b_
对象 ;
```

是否可以将一个 B 指针赋值给一个 A 指针？

以下结构可以实现这一点：

```
auto a_ptr = shared_pointer<A>( {v18}make_shared<B>() );
```

16.3.4 共享指针指向 ‘this’

在对象方法内部，对象可以作为 this 访问。在经典意义上，这是一个指针。那么，如果你想要引用 ‘this’，但需要一个共享指针呢？

例如，假设你正在编写一个链表代码，并且你的 node 类有一个方法 prepend_ 或 _append，该方法返回一个指向链表新头部的共享指针。

您的代码将开始如下，处理新节点追加到当前节点的情况：

```
shared_ 指针 < 节点 >  节点 ::prepend_ 或 _append (
shared_ptr< 节点 > other ) { if (other-> 值 >this-> 值 ) {
this-> 尾 = other; return *this; // 差不多但并不完全正确 }
else { other-> 尾 = this; // 也有点不对劲 return other; }
```

但现在您需要返回此节点，作为一个共享指针。但 this 是一个 节点 *，不是一个 共享 _ptr< 节点 >。

这里的解决方案是您可以返回

```
return this->shared_from_this();
```

如果您已经定义了您的节点类以继承自看起来像魔法的对象：

```
class node : public enable_shared_from_this<node>
```

请注意，您只能返回一个 共享 _ 来自 _this，如果已经存在一个有效的指向该对象的共享指针。

16.3.5 弱指针

除了拥有对象的共享指针和唯一指针外，还有弱 _ptr，它创建了一个弱指针。这种指针类型不拥有对象，但它至少知道何时悬垂。

```
weak_ptr<R> wp; shared_ptr<R>sp( new R );
sp->call_member(); wp= sp; // 通过新的 shared
pointer 访问: auto p =wp.lock(); if (p) { p->call_
member(); } if (!wp.expired()) { // 不是线程安全的!
wp.lock()->call_member(); }
```

弱指针和 shared 指针之间存在一个微妙之处。调用

```
auto sp = shared_ptr<Obj>( new Obj );
```

首先创建对象，然后创建一个 ‘控制块’ 来计数所有者。在另 d,

```
auto sp = make_shared<Obj>();
```

对象和控制块进行一次单独的分配。

16.3.6 空指针

在 C 语言中有一个宏 *NULL*，仅通过约定来表示空指针：不指向任何内容的指针。C++ 有 `nullptr`，它是一个类型为 `std::nullptr_t`。
有一些场景中这很有用，例如，在使用多态函数时：

```
void f(int);
void f(int*);
```

调用 `f(ptr)` 其中指针为 `NULL`，则调用第一个函数，而使用 `nullptr` 则调用第二个函数。

Unfortunately, *dereferencing a nullptr* does not give an exception.

16.3.7 不透明句柄

需要 不透明的指针 `void*` 在 C 中的需求要少得多 ++ 相比于 C。例如，上下文通常可以使用闭包中的捕获来建模（第 26.2.5 章）。如果你确实需要一个不知道它指向什么的指针，可以使用 *std*::any，它通常足够智能，在需要时调用析构函数。有关详细信息，请参阅第 24.6.5 节。

16.3.8 指向非对象的指针

在本章的介绍中，我们论证了 C 中许多指针的使用在 C++ 中已经消失，剩下的主要情况是多个对象共享某个其他对象的 ‘所有权’。

你仍然可以创建指向标量数据的共享指针，例如指向一个标量数组。这样你就可以获得内存管理的优势，但如果你使用的是 `size` 函数和类似功能，你将得不到 `vector` 对象所提供的。

这里是一个指向单个 double 的指针的示例：

代码：

```
1 // /ptrdouble.cpp
2     // shared pointer to allocated
      double
3     auto 数组 = 共享 _ptr<double>(
      new double );
4     double *ptr = array.get();
5     数组 . 获取 ()[0] = 2.;
6     cout<< ptr[0] << ' \n' ;
```

输出

[指针] ptrdouble:

```
2
```

可以初始化该 double：

代码：

```
1 / /ptrdouble.cpp // 共
2  享指针              ter to initialized
      double
3     auto 数组 =
      make_shared<double>(50);
4     double *ptr= 数组 .get();t
5     cout [0] <<r'\ '      n ;
```

输出

[指针] ptrdoubleinit:

```
50
```

你也可以有指向数组的指针。

向量的构造语法稍微复杂一些：

```
auto x = make_shared<vector<double>>(vector<double>{1.1,2.2});
```

16.4 智能指针 vs C 指针

我们指出，在 C++ 中，指针的需求比在 C 中要少。

- 通过引用传递参数，使用引用。第 *7.5* 节。
- 字符串通过 `std::string` 处理，而不是字符数组；见第 11 章。
- 数组主要通过 `std::vector` 处理，而不是 `malloc`；见第 10 章。
- 遍历数组和向量可以使用范围；第 10.2 节。
- 任何遵循作用域的内容都应该通过构造函数创建，而不是使用 `malloc`。

16.4.1 智能指针与 C 风格地址指针

旧式 `&y` 地址指针不能变为智能指针：

```
// /address.cppauto p =shared ptr_
ptr<HasY>( & y ); p - y ; cout "
Pointer’ s y: "
```

```
            << p->y << '\n';
```

gives:

address(56325,0x7fff977cc380) malloc: *** 对象 0x7ffeeb9caf08: 被释放的对象未分配

智能指针比旧式指针好得多

```
Obj *X;
*X = Obj( /* args */ );
```

有一种创建共享指针的最终方法，其中将旧式 新 对象转换为共享指针

auto p = shared ptr _ptr <Obj >(new Obj);

这不是首选的创建模式，但在 弱指针 s; 章节 16.3.5.

16. 指针

第 17 章

C 风格指针和数组

在之前的章节中，你已经学习了 `std::vector` 作为存储数组数据的首选方式、 `std::string` 用于字符数据、按值或引用传递参数，以及智能指针用于链表等数据结构。所有这些机制在 C 中都是通过不同的指针机制处理的。

本章讨论 C 机制。学习它，以防你将来需要处理 C 代码，但在你的 C++ 代码中不要使用它！

17.1 什么是指针

术语 “ 指针 ” 用于表示对某个量的引用。人们喜欢使用 C 作为高性能语言的原因是，指针实际上是内存地址。因此，你正在 “ 接近裸金属 ” 编程，并且对程序执行有极大的控制权。C++ 也有指针，但与 C 指针相比，它们的用途较少：向量和引用已经使许多 C 风格指针的用途变得过时。

17.2 指针和地址，C 风格

你已经学习了变量，也许你有一个关于变量作为 “ 命名内存位置 ” 的心理概念。这并不太远：当你在变量的（动态）作用域内时，它对应一个固定的内存位置。

练习 17.1。 什么时候变量不总是对应相同的内存位置?

有一个查找变量实际地址的机制：你用与号前缀它的名称。这个地址是整型的，但它的范围实际上大于 `int` 类型的范围。

如果你有一个 `int i`，那么 `&i` 是 `i` 的地址。

一个地址是一个（长）整数，表示一个内存地址。通常它用十六进制表示法表示。

Code:

```
// /coutpoint.cpp
  int i;
  cout << "i 的地址，十进制:              "
       <<  lon( &i) << \;"i的地
  cout << 址，十六进制 td h << &      : "
       << i<<:: ex
```

《C++17/fortran2008 科学编程艺术：HPC 卷 3》第 1 章：引言 第 17.1 节：何时变量的内存位置不总是相同?

输出

[指针] coutpoint:

i 的地址，十进制:

140732703427524

i 的地址，十六进制 :

0x7ffee2cbcbc4

Using purely C:

代码:

```
// /printfpoint.cpp
  int i;
  printf ("address of i: %ld\n",
          (长)(&i));
```

printf<0xE3><0x80><0x82>(<0xE3><0x80><0x82>" same in hex: %lx<0xE3><0x80><0x82>\n"<0xE3><0x80><0x82>,

输出

[指针] printfpoint:

i 的地址 : 140732690693076

相同的是十六进制 :7ffee2097bd4

注意，这里的 & 用法与定义引用不同；比较第 7.5.2 节。然而，由于它们在语法上不同，所以永远不会混淆。

你可以直接打印变量的地址，这在调试时有时很有用。如果你想把地址存储起来，你需要创建一个适当类型的变量。这是通过取一个类型并在其后加一个星号来完成的。

'`&i`' 的类型是 `int*`，发音为 '
int-star'，或更正式地：'指向 int 的指针'。

您可以创建这种类型的变量：

```
int i;int* addr= &
i;// 完全相同:
int*addr = &i;
```

现在 `addr` 包含 `i` 的内存地址。

现在，如果您有一个指向 int 的指针：

```
int i;int *iaddr= &i;
```

您可以使用（例如 print）该指针，这将为您提供变量的地址。如果您想获取指针指向的变量的值，您需要解引用 它。

使用 `*addr` '解引用' 指针：给出它指向的内容；内存位置中的值的值。

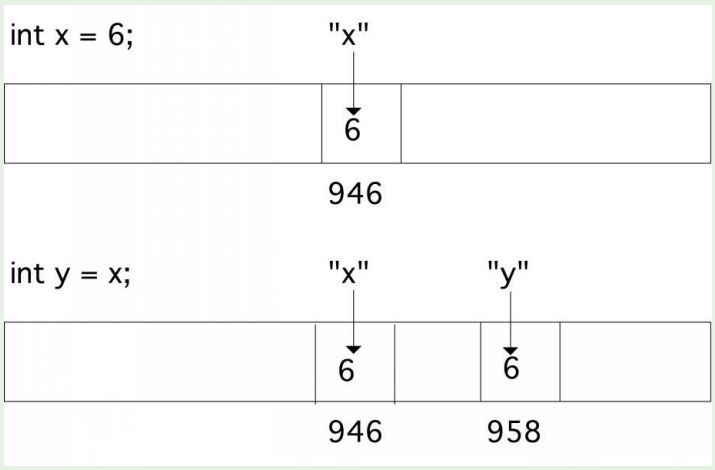
Code:

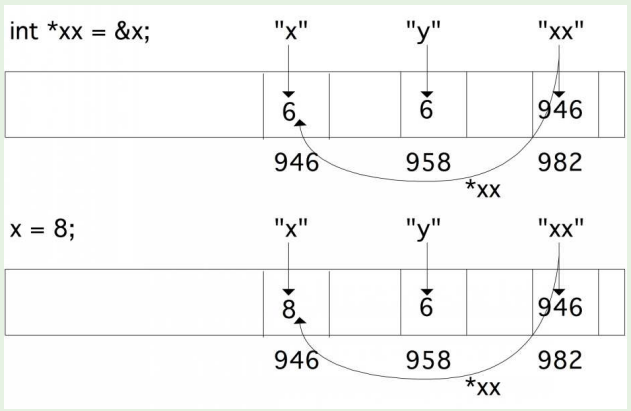
```
1 // /cintpointer.cpp
2   int i;
3   int* addr = &i;
4   i = 5;
5   cout << *addr << '\n';
6   i = 6;
7   cout << *addr << '\n';
```

Output

```
[pointer] cintpointer:

5
6
```





- *addr* 是*i* 的地址。
- 你将 *i* 设置为 5；关于 *addr* 没有任何变化。这会写入 5 到 *i* 的内存位置。

- 第一行 cout 解引用 *addr*，也就是说，查找该内存位置中的内容。
- 接下来你将 *i* 改为 6，也就是说，你写入 6 到它的内存位置。
- 第二行 cout 查找与之前相同的内存位置，现在发现 6。

声明某种类型的指针的语法允许有小的变化，这表明你可以以两种方式解释这种声明。

Equivalent:

- **int*** *addr*:*addr* 是一个 int-star，或者

- `int *addr`: `*addr` is an int.

概念 `int* addr` 等同于 `int *addr`，在语义上它们也是相同的：你可以说是 `addr` 是一个 int-star，或者你可以说是 `*addr` 是一个 int。

17.3 数组和指针

在章节 10.10 中你看到了 C 中的静态数组处理 ++。例如：

```
// /arraypass.cpp void set_array( double *x,int size) {
for (int i=0; i<size; ++i) x[i] = 1.41; }; /* ... */ doublearray
[5] = {11,22,33,44,55}; set_array(array,5); cout <<array
[0] << "...." << array[4] << ' \n' ;
```

这表明，尽管参数通常按值传递，即通过复制传递，但数组参数可以被修改。原因在于没有实际的数组类型，传递的是指向数组第一个元素的指针。因此，数组仍然按值传递，只是不是传递数组的 ‘值’，而是其位置的值。

所以你可以像这样传递一个数组：

```
// /arraypass.cppvoid array_set_star{v11}({v13}double
{v15}*/{v17}ar{v19},{v21}int {v23}idx{v25},{v27}double
{v29}val{v31}) {{{v35}ar[{v38}idx] ={v41}val{v43};}};/* ...
*/array{v47}set{v50}star{v53}({v55}array{v57},2,4.2);
```

数组和内存位置在很大程度上是相同的：

代码：

```
1 // /arraypp doublearray[5] = {11,22,33,44,55};
2
3  double *addr_of_second = &(array[1]);
4  cout << *addr_of_second << '\n';
5  数组[1] = 7.77;t
6  cout << *a  r o  second << '\n';
```

输出

[指针] 数组地址：

```
22
7.77
```

When an array is passed to a function, it behaves as an address:

Code:

```
1// /arraypass.cpp2void set
arra_
                  y( double *x int size) {
3       , for(inti=0;i<size; ++i)
4      x[i] = 1.41;
5 };     //doublearray[5] =
6       * ... *
7   {11,22,33,44,55};
8   set_array{array,5};
9   cout<<数组[0] <<     "...." <<数组[4]
      << '\n';
```

输出
[指针] 数组传递:

1.41....1.41

注意这些数组不知道它们的长度，因此您需要传递它。

您可以使用 **new** 动态分配内存，这会得到一个 something-star:

```
double *x;
x = new double[27];
```

The **new** 运算符仅适用于 C++：在 C 中，您会使用 *malloc* 来动态分配内存。上述示例将变为：

```
double *x;
x = (double*) malloc( 27 * sizeof(double) );
```

请注意，**new** 接受元素的数量，并从上下文中推断类型（因此推断每个元素的字节数）；*malloc* 接受一个参数，该参数是字节数。然后，**sizeof** 运算符帮助您确定每个元素的字节数。

17.4 指针算术

指针算术 使用它指向的对象的大小：

*double *addr_of_* 元素 = 数组; *cout* <<*
*addr_of_* 元素; *addr_of_* 元素 = *addr_of_*
元素 +1; *cout* <<**addr_of_* 元素;

增加数组元素的大小，4 或 8 字节，不 t one!

练习 17.2。 编写一个子例程，该子例程设置数组的第 i 个元素，但使用指针算术：该例程不应包含任何方括号。

17.5 多维数组

After

```
double x[10][20];
```

一行 `x[3]` 是一个 double，所以是 `x` 一个 `double**`?

Was it created as:

```
double **x = 一个 double<style
id='12'>*[10];for(int i=0;i<10; i++)
x[i] =一个<style id='44'>double[20];
```

不：多维数组是连续的。

17.6 参数传递

C++ 风格的函数，它们会改变它们的参数：

```
void inc(int &i) {
  i += 1;
}
int main() {
  int i=1;
  inc(i);
  cout << i << endl;
  return 0;
}
```

在 C 语言中，你不能像这样按引用传递。相反，你需要按值传递变量的地址 `i`：

```
void inc(int *i) {
  *i += 1;
}
int main() {
  int i=1;
  inc(&i);
  cout << i << endl;
  return 0;
}
```

现在，函数接收一个内存地址作为参数：`i` 是一个 int-star。然后它将 `*i`，即一个 int 变量，加一。

注意，这里再次使用了两种不同的与 & 符号相关的用法。虽然编译器没有问题，但对程序员来说有点困惑。

Exercise 17.3. Write another version of the *swap* function:

```
void swap( /* something with i and j */ {
  /* your code */
}
int main() {
  int i=1,j=2;
  swap( /* something with i and j */ );
  cout << "check that i is 2: " << i << endl;
  cout << "check that j is 1: " << i << endl;
  return 0;
```

```
    }
```

提示：编写 C++ 代码，然后在需要的地方插入星号。

17.6.1 分配

在 10.10 节中，你学习了如何创建局部于作用域的数组：

创建一个大小取决于某个值的数组：

```
if( something ) {double ar
[25];} else{double ar[26];}
ar[0] = // 没有数组!
```

这不起作用

The array `ar` 是创建的，取决于条件是否为真，但在条件之后它又会消失。使用 **new** (section 17.6.2) 的机制允许你分配超出其作用域的存储：

C allocates in bytes:

```
double *array;
array  = (double*) malloc( 25*sizeof(double) );
```

C++ 分配一个数组：

```
double *array;
array  = new double[25];
```

Don’t forget:

```
free(array); // C
delete array; // C++
```

现在动态分配：

```
double *array;
if (something) {
  array = new double[25];
} else {
  array = new double[26];
}
```

别忘了：

```
删除数组;
```

```
void func() { double* array = new double[large_
number]; }
```

```
  // code that uses array
}
int main() {
  func();
};
```

- The function allocates memory
- After the function ends, there is no way to get at that memory
- ⇒ *memory leak*.

```
for (int i=0; i<large_num; i++) {
  double *array = new double[1000];
  // code that uses array
}
```

每次迭代都会预留内存，并且永远不会释放：这是另一个内存泄漏。

你的代码将耗尽内存！

使用 `malloc` / new 分配的内存在你离开作用域时不会消失。因此你必须显式删除内存：

```
free(array);
delete(array);
```

The C++ `vector` does not have this problem, because it obeys scope rules.

无需使用 `malloc` 或 **new**

- 使用 `std::string` 来表示字符数组，和
- `std::vector` 来表示其他所有类型。

如果不需要动态改变大小，则不会有性能损失。

17.6.1.1 Malloc

关键字 **new** 和 **delete** 符合 C 风格编程的精神，但在 C 语言中不存在。相反，你需要使用 `malloc`，它创建一个以字节为单位的内存区域。使用函数 **sizeof** 将类型转换为字节：

```
int n;double *array;array =
malloc( n*sizeof(double) );if (!
array)// 分配失败!
```

17.6.1.2 函数中的分配

创建内存的机制，并将其分配给一个 “ 星 ” 变量，可用于在函数中分配数据并从函数返回。

```
void make_array( double **a, int n ) {
  *a = new double[n];
}
int main() {
  double *array;
  make_array(&array,17);
}
```

注意，这需要一个 ‘双重星号’ 或 ‘星号星号’ 参数：

- 变量 `a` 将包含一个数组，因此它需要是类型 `double*`；
- 但它需要通过引用传递给函数，使得参数类型 `double**`；
- 在函数内部，您将新的存储分配给 `double*` 变量，该变量是 `*a`。

这很棘手，我知道。

17.6.2 new 的用法

在进入本节之前，请确保你已经学习了第 *17.3* 节。

存在一个动态分配机制，它深受 C 语言内存管理启发。不要将其作为首选方案。

new 的用法 `new` 利用了数组和引用的等价性。

*//arraynew.cppvoid make_array{v8}(int** new_array, int length) { *new_array =new int length]; }/* ... */int* the_array;make_array(&the_array,10000);*

由于这不是作用域内的操作，你需要自己释放内存：

*//arraynew.cpp 类包含 _ 数组 { private: int *array; int array_length; public: with_ 数组 (intsize) { array_length= size; array= new int [size]; };with_ 数组 () { delete array; }; /* ... */ with_ 数组 thing_with_ 数组 (12000);*

注意你不得不自己记住数组长度吗？使用 `std::vector` 会简单得多。参见 http://www.cplusplus.com/articles/37Mf92yv/。

新的 `new` 机制是 `malloc` 的更干净的变体，后者是 C 中的动态分配机制。malloc 仍然可用，但不应该使用。甚至很少有一些合法的 `new` 用途。

17.7 内存泄漏

指针可能导致一个称为 内存泄漏 的问题：你保留了一些内存，但你已经失去了访问它的能力。

在这个例子中：

```
double *array = new double[100];
// ...
array = new double[105];
```

内存被分配了两次。第一个分配的内存永远不会被释放，因为在中间代码中可能已经设置了指向它的另一个指针。然而，如果那样没有发生，内存就会被分配，并且无法访问。这就是内存泄漏的原理。

17.8 常量指针

指针可以通过两种方式成为常量：

1. 它指向一块内存，而你无法改变它指向的位置。2. 它指向的内容是固定的，该内存的内容也无法被修改。

为了说明非常量的行为：

代码：

```
1 // /starconst.cpp
2     int value = 5;
3     int *ptr=&value;
4     *ptr += 1;
5     cout << "value:    " <<值<< '\n';
6     cout << "*ptr: " << *ptr << '\n';
7     ptr += 1;
8     cout<< "random memory:"     << *ptr
      << '\n';
```

输出

[指针] 星 const1：

```
值： 6
*ptr: 6
随机内存： 73896
```

一种在第一种意义上是常量的指针：

// /starconst.cppint value = 5;int *const ptr = ;&value;; *ptr += 1;; /* DOES NOT COMPILE: cannot assign to variable ’ptr’ with const-qualified type ’int*const’ ptr */;

你也可以创建一个指向常量整数的指针：

// /starconst.cppconst int value = 5;;// value is const /* DOES NOT COMPILE: cannot convert const int* to int* int*ptr&value; */

第 18 章

const

关键字 `const` 可用于表示各种量不能改变。这主要是为了编程安全：如果你声明一个方法不会改变任何成员，但它（间接地）做到了这一点，编译器会警告你。

18.1 const 参数

使用 const 参数是保护你自己的一种方法。如果参数在概念上应该是常量，编译器会在你错误地尝试改变它时捕获到这一点。

函数参数标记 `const` 不能被函数代码修改。以下片段会导致编译错误：

```
    ///
constchange.cppvoid
f(const int i) {i;}
```

18.2 Const references

const 的更复杂用法是 const reference：

```
void f( const int &i) {....}
```

这可能看起来很奇怪。毕竟，引用和按引用传递机制是在第 7.5 节中引入的，目的是将更改后的值返回给调用环境。const 关键字否定了更改参数的可能性。

但使用引用的第二个原因是参数按值传递，这意味着它们会被复制，包括像 `std::vector` 这样的大对象。使用引用传递 vector 在时间和空间上都代价更低，但这样一来，vector 的更改就有可能传播回调用环境。

考虑一个类，它有返回内部成员的引用的方法，一次作为 const reference，一次不是：

Code:

```
1 // /constref.cpp
2 class has_int {
3 private:
4   int mine{1};
5 public:
6   const int& int_to_get() { return
      mine; };
7   int& int_to_set() { return mine; };
8   void inc() { ++mine; };
9 };
10    /* ... */
11  has_int an_int;
12  an_int.inc(); an_int.inc();
      an_int.inc();
13  cout << "Contained int is now: "
14       << an_int.int_to_get() << '\n';
15  /* Compiler error:
      an_int.int_to_get() = 5; */
16  an_int.int_to_set() = 17;
17  cout << "Contained int is now: "
18       << an_int.int_to_get() << '\n';
```

Output
[const] constref:

```
Contained int is now: 4
Contained int is now: 17
```

如果我们定义一个类，其中拷贝构造函数 明确地报告自己，我们就可以显示出按值传递和按 const 引用传递之间的区别：

```
// /copyscalar.cpp 类有 _int{
private: int mine{1}; public: 有 _
int(int v) { cout << "set: "<< v<< '
\n'; mine= v; }; 有 _int( 有 _int &
h ) { auto v = h.mine; cout << "
copy: "<< v<< '\n'; mine= v; };
void printme() { cout << "I have: "
<<mine<< '\n'; }; };
```

现在如果我们定义两个函数，使用这两种参数传递机制，我们看到按值传递会调用拷贝构造函数，而按 const 引用传递不会：

Code:

```
1 // /constcopy.cpp
2 void f_with_copy(has_int other) {
3   cout << "function with copy" << '\n';
      };
4 void f_with_ref(const has_int &other) {
5   cout << "function with ref" << '\n';
      };
6     /* ... */
7   cout << "Calling f with copy..." <<
      '\n';
8   f_with_copy(an_int);
9
10  cout << "Calling f with ref..." <<
      '\n';
11  f_with_ref(an_int);
```

Output
[const] constcopy:

```
Calling f with copy...
(calling copy constructor)
function with copy
Calling f with ref...
function with ref
... done
```

18.2.1 Const 引用在基于范围的循环中

同样的按值 / 按引用传递问题在基于范围的 for 循环中会出现。语法

```
for ( auto v : some_vector )
```

将向量元素复制到 `v` 变量中，而

```
for ( auto& v : some_vector )
```

创建一个引用。要获得引用的好处（无复制成本）同时避免陷阱（意外更改），你也可以在这里使用一个 const 引用：

for (const auto& v: some_vector)

18.3 Const 方法

我们可以区分两种方法：那些会改变对象内部数据成员的方法，以及那些不会改变的方法。那些不会改变的方法可以被标记 `const`。虽然这并非强制要求，但它有助于形成一种干净的编程风格：

使用 `const` 可以捕获方法原型和定义之间的不匹配。例如，

```
class Things {
  private:
  int var;
  public:
  f(int &ivar,int c) const {
    var += c; // typo: should be 'ivar'
  }
}
```

这里，使用 `var` 是一个拼写错误，应该是 `ivar`。由于方法被标记为 `const`，编译器会生成错误，因为该函数在其局部作用域之外进行了修改。

It encourages a functional style, in the sense that it makes *side-effects* impossible:

```
类事物 {private:int i;public:int get() const
{return i; }intinc() { return i++; } // 侧效应!
void addto(int &thru) const {thru += i; }}
```

注意，一个方法被 const 标记并不意味着你不能在其中调用非 const 函数：这意味着 `this` 对象不能被影响。

18.4 const 重载

一个 const 方法及其非 const 变体差异足够大，以至于你可以使用这个进行重载。

代码:

```
1// /constat.cpp
2类 有 _ 数组 {
3私有: t
4   vec o r<浮点数> 值 ;;
5 public :
6   具有 _ 数组 (int    l, float v)
7     : values(      ctor<float>(l,v) {};
8   auto& at(int i) {
9     cout << "var at"<< '\n';
10    返回值。在 (i) ;
11  const auto&at(int i)const          {
12    cout << "const at"<<' \n' ;
13    返回值。在 (i) ;
14  auto sum() const {
15    float p;
16    for( int   n   i=0; i<values.size(); ++i)
17      p +=at(i);
18    返回 p;
19  };
20 };
21
22 int main() {
23
24   int l; float v;
25   cin>> l;cin>> v;
26   has_array fives(l,v);
27   cout << fives.sum() << '\n';
28   fives.at(0) = 2;
29   cout << fives.sum() << '\n';
```

输出

```
const 常量

const at
const at
const at
1.5
var at
const at
const at
const at 4
5.
```

练习 18.1. 探索这个示例的各种变体，看看哪些可以工作，哪些不能。

1. 删除第二个定义的 *at*。你能解释错误吗？ 2. 从第二个 const 关键字中删除一个 *at* 方法。你会得到什么错误?

18.5 常量和指针

让我们声明一个类 *thing* 来指向，以及一个类 *has_thing*，它包含一个指向 *thing* 的指针。

```
//constpoint.cpp
private:
  int i;
public:
  thing(int i) : i(i) {};
  void set_value(int ii){ i = ii; };
  auto value() const { return i;};
};

class has_thing {
private:
  共享_指针<事物>
      东西_ptr{nullptr};
public:
  has_东西(int i)
    : 东西_ptr
      (make_shared<东西>(i)) {};
  void print() const {
    cout<< 东西_ptr->value()<<
      '\n';};
```

如果我们定义一个返回指针的方法，我们会得到指针的副本，因此将这个指针重定向对容器没有效果：

代码：

```
1 //constpoint.cpp
2   自动获取 const 指针 ()          {
3     返回 thing ptr; };/* */
4         ...
5     has_thing 容器 (5)
6     容器.打印();
7     容器.获取_事物_指针() =
8       make_shared<thing>(6);
9     container.print();
```

输出

```
[常量] 常量点 2:
5
5
```

如果我们通过引用返回指针，我们可以改变它。但是，这要求方法不能是常量。另一方面，使用之前的常量方法，我们可以改变对象：

Code:

```
// /constpoint.cpp
  // Error: does not compile
  // auto &get_thing_ptr() const {
  auto &access_thing_ptr() {
    return thing_ptr; };
    /* ... */
    has_thing container(5);
    container.print();
    container.access_thing_ptr() =
      make_shared<thing>(7);
    container.print();

    container.get_thing_ptr()->set_value(8);
    container.print();
```

Output

```
[const] constpoint3:

5
7
8
```

如果您想防止指向的对象被修改，可以将指针声明为共享 _ptr<const 对象 >:

```
// /constpoint.cpp
私有 :h   d_
  s are   ptr<const thin >
      gconst_ 对象 {nullptr};
public:
  has_ 对象 (int i,int j)
    const_thing
      (make_shared<thing>(i+j)) {};
  自动获取 const 指针 () const {
    返回 const_ 东西 ; };
void crint() const {
  cout<< const_thing>-value<<
  \n; }/
   * ... */
  has_thing 容器 (1,2);
  // Error: does not compile

  容器 . 获取 _const_ 指针 ()-> 设置 _ 值      e(9);
```

18.5.1 Old-style const pointers

为了完整性，C 中关于 const 和指针的部分。

我们可以有 const 关键字在 C 指针声明的三个地方:

```
int *p;const int*p;int const* p; //
这与 int* const p; // 相同
```

对于解释，通常说要从 ‘右到左’ 读取声明。所以: </style>

```
int * const p;
```

是一个 ‘指向 int 的 const 指针’。这意味着它指向的 int 是 const，但 int 本身可以改变: </style>

```
// /conststarconst.cppint i=5;int *const ip= ;
&i;printf("(ptr derefs to: %d\n",
*ip);printf("(ptr derefs to: %d\n", *ip);int
j;// DOES NOT COMPILE ip &j;
```

另一方面，

const int *p;

是一个 ‘指向 const int 的指针’。这意味着你可以将它指向不同的 int，但你不能通过该指针改变这些 int 的值：

// /conststarconst.cppconst int *jp = <&i;i;
i> = 7; printf("ptr derefs to: %d\n", *jp); //
DOES NOT COMPILE*jp = 8; ; int k = 9;
jp = &k; printf("ptr derefs to: %d\n", *jp);

最后，

```
const int * const p;
```

是一个 ‘指向常量整数的常量指针’。这个指针不能重新指向其他对象，且其指向对象的值不能被改变：

// /conststarconst.cpp // DOES NOT WORK const int *
const kp; kp= &k;const int * constkp = &k;printf("
ptr derefsto: %d\n",*kp);k = 10;// DOES NOT
COMPILE*kp = 11;

因为它不能重新指向其他对象，所以在声明这种指针时你必须设置其指向对象。

18.6 Mutable

有时你可能希望方法被标记 `const`，但仍然影响其作用域外的数据。为了说明这一点，考虑一个典型的类，该类具有非 const 的更新方法，以及 const 的某些计算量的读取：

**类 Stuff { private: int i,j; public: Stuff(int
i,int j) : i(i),j(j) {}; void seti(int inew) { i =
inew; }; void setj(int jnew) { j =jnew; }; int
result ()const { return i+j; };**

现在假设在 <code>result</code> 函数中的 <code>+</code><code>j </code> 计算代表某项昂贵的操作。你可以通过

1. 缓存计算值；2. <code>result</code> 函数返回缓存的值而无需进一步计算；以及 3. 你让 <code>seti</code><code>/</code><code>setj
</code> 方法重新计算缓存的值。

```
class Stuff { private: int i,j; int 缓存 ; void compute_ 缓存 () { 缓存 =
i+j; }; public: Stuff(int i,int j) : i(i),j(j) {}; void seti(int inew) { i_ i
new; compute_ 缓存 (); }; void setj(int jnew) { j= jnew; compute_ 缓
存 (); }; int result ()const { return 缓存 ; };
```

对于下一个复杂情况，假设设置 `i, j` 比通过 `result` 方法请求计算值更频繁。那么之前的解决方案效率低下，我们改为：

1. 维护一个布尔标志，记录缓存的值是否有效； 2. `seti/j` 函数将此标志设置为 **`false`**； 以及 3. `result` 函数仅在必要时重新计算缓存。

```
class Stuff { private: int i,j; int 缓存 ; bool 缓存 _ 有效 {false}; void
update_ 缓存 () { if (! 缓存 _ 有效 ) { 缓存 = i+j; 缓存 _ 有效 =true; };
public: Stuff(int i,int j) : i(i),j(j) {}; void seti(int inew) { i =inew; 缓存 _ 有效
= false; }; void setj(int jnew) { j =jnew; 缓存 _ 有效 = false; } int result ()
const{ update_ 缓存 (); return 缓存 ; };
```

这无法编译，因为结果 是 const，但它调用了非 const 函数。

我们可以通过将缓存变量声明为 **可变的** 来解决这个问题。然后，那些在概念上不会改变对象的方法仍然可以保持 **`const`**，即使它们在改变对象的状态时也是如此。（最好不要使用 **`const_` 强制类型转换**。）

```
类 Stuff { private: int i,j; mutable int 缓存 ; mutable bool 缓存 _ 有效
{false}; void 更新 _ 缓存 ()const { if(! 缓存 _ 有效 ) { 缓存 = i+j; 缓存 _ 有效
=true; }; public: Stuff(int i,int j) : i(i),j(j) {}; void seti(int inew) { i =inew; 缓
存 _ 有效 = false; }; void setj(int jnew) { j =jnew; 缓存 _ 有效 = false; } i
nt result () const { 更新 _ 缓存 (); return 缓存 ; };
```

18.7 编译时常量

编译器长期以来就能够简化只包含常量的表达式 :

int i=5;**int**
j=**i**+4;**f(j)**

在这里，编译器将得出结论认为 `j` 是 9，故事也就到此为止。这也使得如果函数 `f` 足够简单，可以让 `f(j)` 由编译器进行计算。C++17 添加了几个 `constexpr` 的更多变体。

关键字 **`const`** 可以用来表示一个变量不能被改变：

const int i=5; // DOES
NOT COMPILE: i

模板与组合 **`if`** *`constexpr`* 一起使用很有用：

模板 <typename {v4}T>auto{v9}get_value{v14}({v15}T
t{v16}) {v17}if{v18}constexpr
{
v19}({v20}std{v21}::{v22}is{v23}pointer{v24}v{v25}
T{v26}) {v27}return {v28}*\t{v29};{v30}else return
\t{v31};{v32}}

要声明一个函数为常量，使用 *`constexpr`*。标准示例是：

```
constexpr double pi() {
  return 4.0 * atan(1.0); };
```

but also

```
constexpr int factor(int n) {
  return n <= 1 ? 1 : (n*fact(n-1));
}
```

(C 语言中的递归 ++11，循环和局部变量 ++14。)

- 可以使用条件语句，对常量数据进行测试；
- 可以使用循环，如果迭代次数是常量；
- C++20 可以分配内存，如果大小是常量。

18. 常量

第 19 章

声明和头文件

在本章中，你将学习模块化程序设计所需的技巧。

19.1 Include files

你已经看到了 `#include` 指令在 *STL* 头文件 的上下文中，尤其是 `iostream` 头文件。但你也可以包含任意文件，包括你自己的。

要包含你自己的文件，请使用稍微不同的语法：

```
#include "myfile.cpp"
```

(尖括号符号通常只适用于位于某些系统位置中的文件。) 这条语句的作用是，仿佛将文件直接插入到源代码的该位置。

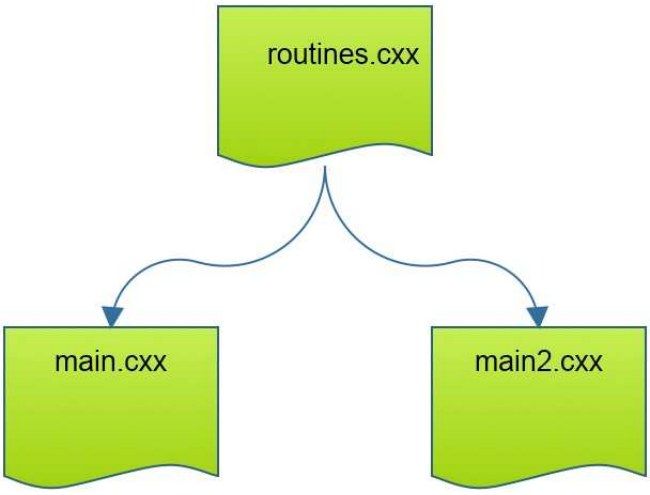


图 19.1：使用包含文件为两个主程序

这种机制为在多个主程序中使用某些函数或类提供了简单的解决方案；参见图 19.1。

这种方法的问题在于构建两个主要程序所花费的时间（大致）等于主要程序编译时间的总和和**两倍**包含文件的编译时间。此外，任何时间你更改包含文件，你都需要重新编译两个主要程序。

在一个更好的场景中，你会一次性编译所有三个文件，并花费一些额外的时间将它们全部连接起来。我们将通过一些步骤来实现这一点。

19.2 函数声明

在大多数情况下，你在这个课程中编写的程序中，你将任何函数或类放在主程序之上，以便编译器在遇到使用之前可以检查其定义。然而，编译器实际上并不需要整个定义，比如一个函数：知道它的名称、输入参数的类型和返回类型就足够了。

Such a minim一个函数的规格说明称为 函数原型 *pe*; for instance

int tester(float);

声明的一个首次使用是前向声明。

Some people like defining functions after the main:

```
int f(int);
int main() {
   f(5);
};
int f(int i) {
   返回 i;
}
```

与之前对比：

```
int f(int i) {
  返回 i;
}
int main () {
   f(5);
};
```

您还需要为相互递归函数提供前向声明：

```
int f(int);
int g(int i) { return f(i); }
int f(int i) { return g(i); }
```

如果您的程序分布在多个文件中，声明会很有用。您可以将函数放在一个文件中，而将主程序放在另一个文件中。

```
// file: def.cpp
int tester(float x) {
  .....
}
```

```
// file : main.cpp
int main() {
  int t = tester(...);
  return 0;
}
```

在这个示例中，一个测试函数在主程序文件之外定义，因此我们需要告诉主程序函数的样子，以便主程序可以编译：

```
// 文件：main.cppint
tester(float);int main()
{int t= tester(...);return
0;}
```

将代码分散到多个文件中并使用 分离编译 是良好的软件工程实践，原因有多个。

1. 如果你的代码变得很大，仅编译必要的文件可以减少编译时间。2. 你的函数可能在其他项目中很有用，无论是你自己还是其他人，因此你可以重用代码，而无需在项目之间复制粘贴。3. 它使浏览文件更容易，而不会被不相关的代码片段分心。

(然而，在实际中你不会这样做。参见关于头文件的 19.2.2 部分。)

19.2.1 分离编译

Your regular compile line

```
icpc -o yourprogram yourfile.cc
```

实际上会做两件事：编译和链接。你可以将它们分开处理：

1. 首先你编译

   ```
   icpc -c yourfile.cc
   ```

   这会给你一个 yourfile.o 文件，一个所谓的对象文件；以及
2. 然后你将编译器用作链接器来给你生成 可执行文件：icpc -o yourprogram yourfile.o

在这个特定的例子中，你可能会想知道这有什么大不了的。如果你有多个源文件，这一点就会变得清晰：现在你为每个源文件调用编译命令，并且只链接一次。

分别编译每个文件，然后链接：

```
icpc -c mainfile.cc icpc -c functionfile.cc icpc -o
yourprogram mainfile.o functionfile.o
```

此时，你应该学习 *Make* 工具来管理你的编程项目。

19.2.2 头文件

Even better than 每次需要函数 i 时都编写声明 s to have a *header file*:

使用带有函数声明的头文件。

Header file contains only declaration:

```
// file: def.h
int tester(float);
```

The header file gets included both in the definitions file and the main program:

```
// file: def.cpp
#include "def.h"
int tester(float x) {
  .....
}
```

```
// file : main.cpp
#include "def.h"

int main() {
  int t = tester(...);
  return 0;
}
```

What happens if you leave out the `#include "def.h"` in both cases?

Having a header file is important because you measure:

- 假设你更改了你的函数定义，更改了它的返回类型：
- 当你编译定义文件时，编译器会报错；
- 所以你更改头文件中的声明；
- 现在编译器会对主程序报错，所以你也编辑它。

必须在主程序中包含头文件。在定义文件中包含它并非严格必要，但这样做可以捕获潜在的错误：如果你更改了函数定义，但忘记更新头文件，编译器会捕获这种情况。

注意 23 顺便问一下，为什么编译器会重新编译主程序，即使它没有更改呢？嗯，那是因为你使用了 makefile。参见《教程》书籍 *[9]*，第 *3* 节。

注意 24 头文件在 *C* 语言中能够捕获比 *C++* 更多的错误。由于函数的多态性，不再错误地出现

```
// header.hint
somefunction(int);
```

and

```
#include "header.h"

int somefunction( float x ) { .... }
```

19.2.3 Cand C++headers

您已经看到了以下包含头文件的语法：

```
#include <header.h>#
include "header.h"
```

第一个通常用于系统文件，第二个通常用于您自己的项目文件。有一些来自 C 标准库的头文件，例如 math.h ；在 C++ 中包含它们的惯用方法是

```
#include <cmath>
```

19.3 类方法的声明

Section 9.5.3 解释了你可以如何将类声明与其定义分开。然后你可以将声明放在一个头文件中，该文件在你使用类的地方包含，而定义只需编译一次，并在你的程序的可执行文件构建时链接。

头文件：

```
// /header.hppclass
something{private::int i;;public::double
dosomething( int i, char c );};
```

实现文件：

```
// /func.cppdouble something::dosomething(int i,char c )
{// do something with i,c};
```

数据成员，即使是私有的，也需要在头文件中：

```
class something {
private:   int localvar;
public:    //声明:

double somedo(vector);};
```

实现文件：

```
// 定义 double
something::somedo(vector v) {....
something with v....    ....something
with localvar ....};
```

评论 19.1. 对于以下每个，回答：这是一个有效的函数定义还是函数声明。

- `int foo();`
- `int foo() {};`
- `int foo(int) {};`
- `int foo(int bar) {};`

```
• int foo(int) { return 0; };
int foo(int bar) { return 0; };
```

19.4 更多

19.4.1 头文件和模板

参见第 22.2.3 节。

19.4.2 命名空间和头文件

命名空间（参见第 20 章）很方便，但它们存在一个危险，即它们可能会定义函数而命名空间的使用者没有意识到。

因此，永远不应该在头文件中写 `using namespace` 。

19.4.3 全局变量和头文件

如果你有一个你想让所有地方都知道的变量，你可以把它做一个全局变量：

```
int processnumber;voidf() {...
processnumber...}intmain()
{processnumber= // 一些系统调用 };
```

它然后在主程序中定义，并在您的程序文件中定义的任何函数中定义。

警告：定义全局变量很有诱惑力，但这是一种危险的做法；参见第 20.4 节。

如果您的程序有多个文件，您不应该在其他文件中放置 “int processnumber”，因为那样会创建一个新的变量，该变量仅对该文件中的函数可见。相反，请使用：

extern int processnumber;

这表示全局变量 processnumber 在其他文件中定义。 e.

如果您将变量放在 头文件 中会发生什么？ 由于 预处理器 充当如果头文件是文本插入，这又会导致每个文件一个单独的全局变量。然后解决方案更复杂：

//file: header.h#ifndef
HEADER#define
HEADERH#endif#ifndef
EXTERN#define
EXTERNHextern#
endifEXTERNint
processnumber//file:
aux.cc#include "header.h"

```
//file: main.cc
#define EXTERN
#include "header.h"
```

(这种预处理器的魔法在章节 r 21.)

这也防止了头文件的递归包含。

19.5 Modules

The C++20 标准正在以不同的方式处理头文件，通过引入 模块。 (Fortran90 已经很长时间有了这个；见章节 37。) 这在很大程度上消除了通过 C 预处理器（ CPP ）包含头文件的需要。然而，CPP 可能仍然需要用于其他目的。

19.5.1 使用模块的程序结构

使用模块，`#include` 指令不再需要；相反，*import* 关键字指示在（子）程序中使用哪个模块：

```
// /fgmain.cppimport fg_模块;intmain()
{std::cout <<"Hello world " << f(5) << '\n';
```

模块位于不同的文件中；`export` 关键字，后跟 *module* 定义模块的名称。然后，该文件可以有任意数量的函数；只有带有 `export` 关键字的函数在导入模块的程序中可见。

```
// /fgmod.cppexport modulefg_
模块;// 内部函数 intg( int i ) {
return i/2; };// 导出函数 export
int f( int i) {return g(i+1);};
```

19.5.2 实现和接口单元

模块可以通过使用带有 模块 : 分区 结构的名称来具有分层结构。

这使得可以拥有独立的

- 接口分区，它们定义了使用程序的接口；和
- 实现分区，它们包含需要屏蔽用户的代码。

这是一个实现分区；因为没有 *import* 关键字，因为这个功能是内部的：

```
// /helperhelp.cpp // 实现单元，未导出任何内容模
块 helper_ 模块 :helper;// 内部函数 intg( inti )
{return i/2; };
```

这里是一个接口划分，它使用内部函数，并导出一个不同的函数：

```
// /helpermod.cpp导出 模块
helper_ 模块 ; 导入 :helper;// 导
出函数导出 int f( int i)
{return g(i+1);};
```

19.5.3 更多

可导入的头文件：

```
import <header.h>
import "header.h"
```

导入声明必须在其他模块规范之前，无论是模块还是函数的导入或导出。

您可以导出变量、命名空间。

第 20 章

命名空间

20.1 解决命名冲突

在 10.3 节中，你看到了 C++ STL 带有一个 *vector* 类，该类实现了动态数组。你说

```
std::vector<int> 一组_ 的_ 整数 ;
```

并且你有一个可以存储一组整数的对象。如果你经常使用这样的向量，你可以通过**使用命名空间**来节省一些输入。你放入

```
using std::vector;
```

在您的文件中较高处，并编写

```
vector<int> bunch_of_ints;
```

在文件的其余部分中。(参见第 20.4 节，了解为何不应使用 `using namespace std` 惯用法。)

但如果您正在编写一个几何包，该包包含一个类，那么是否与 STL 类冲突？如果没有 *namespace* 现象，就会冲突，该现象作为类、函数、变量的区分前缀。

当您编写 std::vector 时，您已经看到了命名空间的作用：'std' 是命名空间的名称。

您可以通过编写来创建自己的命名空间

```
命名空间 a_ 命名空间 {
    // 定义类 一个 _ 对
象 {};
```

以便您可以编写

Qualify type with namespace:

```
a_ 命名空间 :: 一个 _ 对象 myobject ();
```

or

```
using a_命名空间::一个_对象;
一个_对象 myobject();
```

甚至

```
using namespace a_namespace;
一个_对象 myobject();
```

一些包具有复杂的命名空间结构，因此您可能会发现自己编写

```
using namespace abc = space_a::space_b::space_c;
abc::func(x)
```

在 C++ 标准库中，存在两个层次是非常常见的。

20.2 命名空间头文件

如果你的命名空间将在多个程序中使用，你希望将其放在一个单独的源文件中，并附带一个头文件：

There is a `vector` in the standard namespace and in the new `geometry` namespace:

```
// /geo.cpp#include <vector>#
include {v9}"geolib.hpp"{v11}using
namespace geometry{v13};{v17}int
main{v19}() {

  std::vector< vector > vectors;
  vectors.push_back( vector( point(1,1),point(4,5) ) );
```

该头文件将包含常规函数和类头文件，但现在位于命名空间内部：

```
// /geolib.hpp
namespace geometry {
  class point {
  private:
    double xcoord,ycoord;
  public:
    point() {};
    point( double x,double y );
    double x();
    double y();
  };
  class vector {
  private:
    point from,to;
  public:
    vector( point from,point to);
    double size();
  };
}
```

以及实现文件将包含同名的命名空间中的实现：

```
// /geolib.cpp namespace geometry { point::point( double x,double y )
{ xcoord = x; ycoord= y; }; doublepoint::x() { return xcoord; }; // ' 访
问器 ' doublepoint::y() { return ycoord; }; vector::vector( point
from,point to) { this->from = from; this->to= to; };
doublevector::size() { double dx= to.x()-from.x(), dy=
to.y()-from.y(); return sqrt( dx*dx + dy*dy ); }; }
```

20.3 命名空间和库

正如本章引言所述，命名空间是防止命名冲突的好方法。这意味着为所有你的例程创建一个命名空间是个好主意。你几乎在所有已发布的 C++ 库中都能看到这种设计。

现在考虑这个场景：

1. 您编写了一个使用某个库的程序；
2. 该库发布了新版本，并安装到您的系统上； 3. 您的程序，使用共享 / 动态库，开始使用这个库，甚至可能您都没有意识到。

这意味着旧库和新库需要在以下几个方面保持兼容：

1. 早期版本中定义的所有类、函数和数据结构也需要在新版本中定义。这不是一个大问题：新库版本通常会增加功能。然而，2. 新版本的数据布局需要保持相同。

That second point is subtle. To illustrate, consider the library is upgraded:

First version:

```
namespace geometry{类
vector
{private:std::vector<float
> 数据 ;std::string 名称 ;}}
```

New version:

```
namespace geometry {
  class vector {
    private:
    std::vector<float> data;
    int id;
    std::string name;
  }
}
```

问题是，你的编译程序在类对象中有明确信息，类成员位于何处。通过改变对象的结构，这些引用就不再正确。这被称为 “ 应用程序二进制接口（ ABI ）断裂 ”，并导致未定义行为。

库可以通过以下方式防止这种情况：

```
namespace geometry{inline
namespacev1.0 {类
vector{private:std::vector<
float> data;std::string
name;}}}
```

并在未来的版本中更新版本号。使用此库的程序隐式地使用命名空间 `geometry::v1.0::vector`，因此，在库更新后，尝试执行程序

20.4 最佳实践

在本课程中，我们提倡显式地引入函数 tly:

```
#include <<iostream>>
<
using><std>{v9}::<cout>{v13};
```

It is also possible to use

```
#include <iostream>using
namespace std{v9};
```

这个问题在于它可能会引入不需要的函数。例如：

这段代码可以编译，但不应该这样：

```
// /swapname.cpp
#include <iostrea m>
using namespace std;

def swop(int i,int j) {};

int main() {
   int i=1,j=2;
   swap(i,j);
   cout <<i << ' \n' ;
   返回 0;
}
```

这会产生一个错误：

```
// /swapusing.cpp
#include <iostream >
using std::cout;

def swop(int i,int j) {};

int main() {
   int i=1,j=2;
   swap(i,j);
   cout << i <<' \n' ;
   返回 0;
}
```

即使你使用 using namespace，你也只在源文件中这样做，而不是在头文件中。任何使用头文件的人都不会知道突然定义了什么函数。

第 21 章

预处理器

在你的源文件中，你已经看到了以井号符号开头的行，例如

```
#include <iostream>
```

以井号字符开头的行被称为 指令，它们由 C 预处理器解释，也简称为 预处理器。这是一个在实际编译器之前的源到源的转换阶段。

我们在这里将看到一些预处理器更常见的用法。

最佳实践提示。

预处理程序能够实现强大的效果，这些效果实际上无法通过其他方式实现。然而，由于它也导致危险和令人困惑的编程实践，因此应尽量减少其使用。

21.1 包含文件

The `#include` 指令会导致命名文件被包含。该文件实际上是一个普通的文本文件，存储在您的系统中。因此，您的源文件被转换为另一个源文件，在源到源的转换阶段，只有第二个文件实际上由编译器编译。

通常，包含所有内容的中间文件会立即被销毁，但在罕见的情况下，您可能希望将其转储用于调试。请参阅您的编译器文档以了解如何生成此文件。

21.1.1 包含的种类

虽然你可以包含任何类型的文本文件，但预处理器通常用于在源代码的开头包含一个 头文件。

包含有两种类型

1. 文件名可以包含在尖括号中，

   #include <vector>

   这通常用于系统头文件，它们是编译器基础设施的一部分；2. 名称也可以用双引号包含，

```
#include "somelib.h"
```

这通常用于你代码的一部分文件，或者用于你自己下载并安装的库。

21.1.2 搜索路径

系统头文件通常可以被编译器找到，因为它们位于某些标准位置。包括其他文件可能需要你采取额外的操作。如果你编写

```
#include "foo.h"
```

编译器只在您编译的当前目录中查找该文件。

如果包含文件是您自己下载并安装的库的一部分，比如它在

```
/home/yourname/mylibrary/include/foo.h
```

那么您当然可以明确指定位置：

```
#include "/home/yourname/mylibrary/include/foo.h"
```

然而，这并不使您的代码很容易移植到其他系统和用户。那么您如何使

```
#include "foo.h"
```

在任何系统上都能被理解？

答案是你可以给你的编译器一个或多个 包含路径。这是通过 -I 标志完成的。

```
icpc -c yourprogram.cpp -I/home/yourname/mylibrary/include
```

你可以指定多个这样的标志，编译器会尝试在所有这些路径中找到 foo.h 文件。

你现在是不是在想每次编译都要输入这个路径？这就是学习工具 *Make* 教程书籍 [9]，章节 -3 或 *CMake* 教程书籍 [9]，章节 -4。

21.2 文本替换

假设你的程序有多个数组和循环边界，它们都是相同的。为了确保使用相同的数字，你可以创建一个变量，并在需要时将其传递给例程。

```
void dosomething(int n) { for (int i=0;
i<n;i++) .... } int main() { int n=100000;
vector<int> idxs(n); vector<float>
vals(n); dosomething(n);
```

您也可以使用 `#define` 来定义一个预处理宏：

#define N 100000void doso
mething() {for (int i=0; i{i24}N; i
{i31}) .... }int main() {vector<int>>
idxs(N);vector<float>>
vals(N);dosomething();}

通常使用全大写字母来表示此类宏。

21.2.1 动态宏定义

在源代码中定义数字会减少一些灵活性。您可以通过让编译器定义宏来恢复一些灵活性，使用 -D 标志：

```
icpc -c yourprogram.cpp -DN=100000
```

现在，如果你想在源代码中设置默认值，但希望编译器可选地优化它，该怎么办？你可以通过使用 `#ifndef` （'if not defined'，即 “ 如果未定义 ” ）来测试源代码中的定义来解决此问题：

#ifndef N#define
N 10000#endif

另见章节 21.3.2。

21.2.2 参数化宏

除了简单的文本替换，你还可以使用参数化预处理宏

例如，以下宏检查需要例程提前退出的返回代码：

```
#define CHECK_FOR_ERROR(i) if (i!=0) return i
...
ierr = some_function(a,b,c); CHECK_FOR_ERROR(ierr);
ierr = more_function(d,e);   CHECK_FOR_ERROR(ierr);
```

Y你看到这种参数化宏看起来有点像函数定义，除了

- 参数没有类型：它们是按文本替换的；
- 宏替换不是作用域，除非你显式地用花括号括起来；
- 整个定义必须在一行上：如果需要，可以用反斜杠转义行尾。

当你引入参数时，最好使用大量的括号。以下定义是危险的：

```
#define MULTIPLY(a,b) a*b
...
x = MULTIPLY(1+2,3+4);
```

This expands to

```
1+2*3+4
```

更好的版本使用括号：

```
#define MULTIPLY(a,b) (a)*(b) ...
x= MULTIPLY(1+2,3+4);
```

宏的另一个常用用途是模拟多维索引：

```
#defineINDEX2D(i,j,n) (i)*(n)+j ...
double array[m*n]; for (int i=0; i<m; i++)
for (int j=0; j<n; j++) array[
INDEX2D(i,j,n) ] =...
```

Exercise 21.1. Write a macro that simulates 1-based indexing:

```
#defineINDEX2D1BASED(i,j,n) ???? ... double
array[m*n]; for (int i=1; i<=m; i++) for (int j=n; j
<=n; j++) array[ INDEX2D1BASED(i,j,n) ] =...
```

21.2.3 类型定义

与宏相关的还有 `typedef` 关键字：

```
typedef int* intptr;
intptr my_ptr;
```

你在第 4.2.1 节看到的 using 关键字，也可以用作 `typedef` 的替代，如果它用于引入类型的同义词。

```
using Matrix=vector<vector<float>>;
```

21.3 条件语句

There are a couple of *preprocessor conditionals*.

21.3.1 检查 avalue

The #if 宏测试非零。一个常见的应用是暂时从编译中移除代码：

```
#if 0 需要禁用的代码块 #endif
```

您也可以测试数值相等：

```
#ifVARIANT == 1some
code#elifVARIANT == 2
other code#else #error
No suchvariant#endif
```

21.3.2 检查宏

The `#ifdef` 测试用于检查宏是否已定义。相反，`#ifndef` 用于检查宏是否未定义。例如，

```
#ifndef N#
define N 100#
endif
```

宏为什么已经被定义？嗯，你可以在编译行上这样做：

```
icpc-c 文件 .cc -DN=500
```

这个测试的另一个应用是防止头文件递归包含；参见第 19.4.3 节。

21.3.3 只包含文件一次

如果头文件在多个其他头文件中被包含，很容易多次包含 iostream 之类的文件。这会增加你的编译时间，或者可能导致微妙的问题。一个头文件甚至可能循环包含自己。为了防止这种情况，头文件通常具有结构

```
// 这是 foo.h#ifndef FOO_H#define FOO_
H// 你想要包含的内容 #endif
```

现在文件将只被有效包含一次：第二次包含时其内容将被跳过。

这种机制非正式地称为头文件保护 .

许多编译器支持具有相同效果的 `#once` 指令：

```
// this is foo.h
#pragma once

// 你只想包含一次的内容
```

然而，这并未标准化，其确切含义也不明确（如果将其放在文件中间会怎样？），因此核心指南 建议使用显式保护。

21.4 其他预处理器指令

通过无填充字节的打包数据结构 #pack

```
#pragma pack(push,
1)// 数据结构 #pragma
pack(pop)
```

如果你有太多 #ifdef 情况，你可能会得到无法工作的组合。有一个方便的指令来退出无意义的编译：#error。

#ifdef vax__ __#error " 在 VAX 上无法工作。" #endif

不太严重的是 #warning，它只会将一条警告字符串显示在你的屏幕上。

在 21.3.3 节中，你看到了 #pragma 指令。它有一些标准化的用途，但也可以用于特定于编译器的规范。

第 22 章

模板

在本课程中，您已经看到类型 `vector<string>` 和 `vector<float>` 非常相似：它们都有相同名称的方法，并且这些方法的行为基本相同。这种尖括号表示法称为 ‘模板化’，而 `string` 或 `float` 称为模板参数。

如果您深入挖掘 C++ 库的源代码，您会发现，在某个地方，只有一个 `vector` 类 的定义，但使用新的表示法，它将 `string` 或 `float` 作为模板参数。

就好像（当然现实更加复杂）模板类（或函数）前面有一行

```
template< typename T >
class vector {
  ...
};
```

此机制也适用于您自己的函数和类。

历史上 `typename` 是 类，但这很令人困惑。

22.1 模板函数

我们将从简要了解模板函数开始。

定义：

```
模板<typename T>void 函数(T var) { cout <<var
<< end; }
```

使用：

```
int i; function(i); double x;
function(x);
```

并且代码的行为就好像你定义了两次函数，一次为 int，一次为 double。

练习 22.1. 机器精度，或‘机器浮点数精度’，有时定义为最小的数 ϵ，使得 $1+\epsilon>1$ 在计算机算术中成立。

编写一个模板函数 *epsilon*，使得以下代码分别打印出 float 和 double 类型的机器精度值：**float** 和 **double** 类型分别：

代码：

```
// /eps.cpp
  float float_eps;
  epsilon(float_eps);
  cout<< "Epsilon float: "
      << setw(10) << setprecision(4)
      <<float_eps << '\n'{v10};

  double double_eps;
  epsilon(double_eps);
  cout<< "Epsilon double: "
      << setw(10) << setprecision(4)
      << double_eps<<'\n'{v10};
```

输出

[模板] eps:

ε 浮点数：1.0000e−07

ε 双 < 样式 id='1'>: 1.0000</ 样式 >e< 样式 id='3'>-15</ 样式 >

Hint: you may need to cast scalars to the appropriate type.

练习 22.2. 如果你在做零点查找项目，你现在可以做一些练习在节 47.2.3。

22.2 模板类

模板最常见的用途可能是定义模板类。事实上，你已经见过这个机制的实际应用：

模板化的向量类大致如下：

```
template<typename T>class vector
{private:T*vectordata; // 内部数据 public:T

at(int i) {return vectordata[i] };int
size() { /* 返回数据大小 */ };// 更多内容 }
```

让我们考虑一个实际示例。我们编写一个类 *Store*，它存储一个模板参数类型的单个元素：

代码：

```
// /example1.cpp   存储<int>
i5(5);cout <<i5.value() << '
\n';
```

Output

[template] example1i5:

5

类定义看起来相当正常，除了类型（**int** 在上述示例中）是参数化的：

```
///
example1.cpptemplate<typename
T>{class Store{private: T stored;
public: Store(T v) : stored(v) {}; T
value() {return stored;};};}
```

如果我们编写的方法引用了模板类型，事情会变得稍微复杂一些。假设我们想要两个方法 `copy` 和 `negative`，它们返回相同模板类型的对象：

代码：

```
1// /example1.cpp
2 Store<float>also314 = f314.copy(); 3 cout <<
also314.value() << '\n'; 4 Store<float>min314 =
f314.negative(); 5 cout <<min314.value() << '\n';
```

Output

```
[template] example1f314:
3.14
-3.14
```

方法定义相当直接；如果你省略模板参数，类名注入 机制将使用与被定义的类相同的模板值。在以下两行中，可以指定模板参数或省略：

```
///example1.cppStore copy() {return Store(stored);};Store<T>
negative() { returnStore<T>(-stored); };
```

22.2.1 类外方法定义

如果我们分离类签名和方法定义（例如用于分离编译；参见章节 19.3 ）事情会变得更复杂。类签名很简单：

```
///
e
x
a
mple2.cppte
mplate<typename T>class
Store{private: T stored;
public: Store(T v); T value();
Store copy(); Store(T
negative); };
```

方法定义更复杂。现在，每次提到模板类时都需要指定模板参数，除了构造函数的名称：

```
// /example2.cpp模板<typename T>Store<
T>::Store(T v) :stored(v) {}; 模板<
typename T>T Store<T>::value()
{return stored;};
```

```
template< typename T >
Store<T>  Store<T>::copy() { return Store<T>(stored); };

template< typename T >
Store<T> Store<T>::negative() { return Store<T>(-stored); };
```

22.2.2 具体实现

有时模板代码对某些类型（或值）有效，但对所有类型都无效。在这种情况下，你可以使用空尖括号指定特定类型的实例化：

```
模板 < 类型名 T>void f(T); 模板 <>void
f(char c) { /* 带有 c 的代码 */ }; 模板 <>
void f(doubled) { /* 带有 d 的代码 */ };
```

22.2.3 模板和分离编译

模板的使用通常使 分离编译 变得困难：为了编译模板定义，编译器需要知道它们将用于哪些类型。因此，许多库是 仅头文件 ： 它们必须在每个使用它们的文件中包含，而不是单独编译并链接。

在你可以预见将要实例化模板类的类型的常见情况下，有一个解决方法。假设你有一个模板类和函数：

```
模板 < 类型名 {v4}T> 类
foo<T>  {} ； 模板 < 类型名
{v24}T>double f{v29}
（ T x{v33} ） ；
```

如果它们（类和函数）只会在 `float` 和 `double` 上使用（实例化），那么在类定义之后添加以下行可以使文件分别编译：

```
模板类 foo<float>; 模板类 foo<
double>; 模板 doublef(float); 模
板 doublef(double);
```

如果类被拆分到头文件和实现文件中，这些行放在实现文件中。

22.3 示例：域上的多项式

任何数值应用都可以使用模板来允许在 单精度 `float` 和 双精度 `double` 中进行计算。然而，我们通常还可以将计算泛化到其他 域。考虑一个例子：多项式，在标量和（方）矩阵中。

让我们从一个简单的多项式类开始：

```
// /poly_eval.cpp 类 polynomial { private: vector<double>
coefficients; public: polynomial( vector<double> c ) : coefficients(c) {}; //
5 x^2 + 4 x + 3 = 5 x + 4 x + 3doubleeval( double x ) const {
double y{0.}; for_each(coefficients.rbegin(),coefficients.rend(),[x,&y]
(double c) { y *=x; y+= c; } ); return y; }; double operator()(doublex)
const { return eval(x); };
```

我们存储多项式系数，零次系数位于位置零，等等。给定输入 x 评估多项式的例程是 *Horner* 方案 的实现：

$$5x^2 + 4x + 3 \equiv ((5) \cdot x + 4) \cdot x + 3$$

(注意上面的 `eval` 方法使用 `rbegin`, `rend` 以正确的顺序遍历系数。)

例如，系数 `2,0,1` 对应多项式 $2 + 0 \cdot x + 1 \cdot x^2$:

```
// /poly_eval.cpp多项式 x2p2( {2., 0., 1.} );for ( auto x: {1., 2., 3.} )
{autoy = x2p2(x);cout << “x 的平方 ”=“<< x << “ 加上 2 得到 y”=“<< y << ’
\n’;}
```

如果我们把上述内容推广到矩阵的情况，所有多项式系数，以及 x 输入和 y 输出，都是矩阵。

上述用于对特定输入求多项式值的代码，只要定义了乘法和加法运算符，对矩阵同样适用。所以假设我们有一个类 `mat` ，并且我们有

```
mat::mat operator+(const mat& other) const; void
mat::operator+=(const mat& other); mat::mat
operator*(const mat& other) const; void mat::operator*=
(const mat& other);
```

Now we redefine the `polynomial` class, templated over the scalar type:

```
// /poly_mat.cpp template< typename Scalar>class 多项式 {
private: vector<Scalar> coefficients; public: 多项式 ( vector<Scalar>c ) :
coefficients(c) {}; int degree()const { return coefficients.size()-1; }; // 5
x^2 + 4x + 3 = 5 x + 4 x + 3
```

```
标量求值（标量 x）const {标量 y{0.};for_each(系数.
rbegin(), 系数.rend(),[x,&y] (标量 c) { y *= x;y += c; }
);return y;};
```

使用多项式的代码保持不变，只是我们在创建多项式对象时必须将标量类型作为模板参数提供。上述 $p(x) = x^2 + 2$ 的例子对于标量变为：

```
// /poly_mat.cpp
    polynomial<double> x2p2( {2., 0., 1.} );
    for ( auto x : {1., 2., 3.} ) {
      auto y = x2p2(x);
      cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
    }
```

以及对于矩阵：

```
// /poly_mat.cpp
    polynomial<mat> x2p2( {2., 0., 1.} );
    for ( auto x : {1., 2., 3.} ) {
      auto y = x2p2(x);
      cout << "Second power of x=" << x << " plus 2 gives y=" << y << '\n';
    }
```

您可以看到，在模板定义之后，多项式对象完全相同地使用。

22.4 概念

C++20 标准增加了 <code> 概念 </code> 的概念。

Te模板 ca可以是过于通用的。例如，可以编写一个模板 ed *gcd* function

<code> 模板 </code><<code> 类型名 </code><code>T</code>><code>T gcd</code><code>(</code><code>T a</code><code>, </code><code>T b</code><code>) {</code><code>if</code><code>(</code><code>b</code>==0<code>) </code><code>return </code><code>a</code><code>;</code><code>else return</code><code>gcd</code><code>(</code><code>b</code><code>,</code><code>a</code><code>%</code><code>b</code><code>); }</code>

这将适用于各种整型。为了防止它被应用于非整型，您可以为类型指定一个 <code> 概念 </code>：

<code> 模板 </code><<code> 类型名 </code><code>T</code>><code> 概念 </code><code>bool</code><code>Integral</code><code>() {</code><code>return </code><code>std</code><code>::</code><code>is</code>_<code>integral</code><<code>T</code>><code>::</code><code>value</code><code>; }</code>

用作：

```
模板<typename T>requires
Integral<T>{}T gcd( T a, T b) { /
* ...*/ }
```

or

```
template <Integral T>
T gcd( T a, T b ) { /* ... */ }
```

简化的函数模板：

```
积分自动 最大公约数（ 积分自动 a, 积分自动 b ）
{ /* ...*/ }
```

22. 模板

第 23 章

错误处理

23.1 一般讨论

在编程时，犯错几乎是不可避免的。`语法错误`，即违反语言语法规则，会被编译器捕获，并阻止生成可执行文件。因此，在本节中我们将讨论`运行时错误`：运行时与预期不符的行为。

这里是一些运行时错误来源

数组索引使用超出数组边界的索引可能会导致运行时错误 r:

```
vector<float> a(10);for (int i
=0; i<=10;i++)a.at(i) =x; // 运行
时错误
```

或未定义行为：

```
vector<float>
a(10);for (int i=0; i<=10;i++)
a[i] = x;
```

参见进一步章节 10.3。

空指针 使用未初始化的指针可能会导致程序崩溃：

```
Object *x;
if (false) x = new Object;
x->method();
```

数值错误 例如除以零不会导致程序崩溃，因此捕获它们需要一些小心。

Guarding against errors.

- 检查前置条件。
- 捕获结果。
- 检查后置条件。

错误报告：

- 消息
- 完全中止
- 异常

23.2 支持错误处理和调试的机制

23.2.1 断言

一种在错误发生前捕获错误的方法是在代码中添加断言：关于必须为真的陈述。例如，如果一个函数在数学上应该始终返回正结果，那么无论如何检查这一点可能是个好主意。您可以通过使用 *assert* 命令来实现这一点，该命令接受一个布尔值，如果布尔值为假，将停止您的程序：

Check on valid input parameters:

```
#include <cassert>

// this function requires x<y
// it computes something positive
float f(x,y) {
  assert( x<y );
  return /* some result */;
}
```

Check on valid results:

```
float positive_outcome = f(x,y);
assert( positive_outcome>0 );
```

还有 *static_assert*，它只检查编译时条件 .

由于检查断言是一个轻微的计算，您可能希望在您的程序在生产环境中运行时禁用它，通过定义 *NDEBUG* 宏：

```
#define NDEBUG
```

一种方法是将它作为编译器标志：

```
icpc -DNDEBUG yourprog.cpp
```

作为使用断言的一个例子，我们可以考虑 Collatz 练习的迭代函数 6.13。

```
// /modular.cpp
int collatz_next( int current ) {
  assert( current>0 );
  int next{-1};
  if (current%2==0) {
    next = current/2;
    assert(next<current);
  } else {
    next = 3*current+1;
    assert(next>current);
  }
  return next;
}
```

注意 25 如果一个断言失败，你的程序将调用 *std::abort*。这是一种不太优雅的退出方式，而不是 *std::exit*。

23.2.2 异常处理

断言有点粗糙：它们会终止你的程序，你能做的唯一事情是找到问题、重写你的代码并重新运行。有些错误是你在程序运行期间可能恢复的类型。在这种情况下，异常的机制是一个更好的想法，因为异常可以在程序内部进行处理。

这里的关键术语是

- 异常抛出，它使用 `throw` 关键字来表示问题；和
- 异常捕获，它使用 `catch` 关键字来指定当异常被抛出时要采取的操作。

Illustration of exception throwing and catching.

存在问题的代码：

```
if ( /* some problem */ )
  抛出 (5);
```

```
try {
  /* code that can go wrong */
} catch (...) { // literally three
  ...
  /* 处理问题的代码 */
}
```

你可以抛出各种类型的异常；在刚才的例子中我们使用了一个整数，但你也可以抛出一个错误字符串，甚至异常对象。更多内容见下文。

23.2.2.1 异常捕获

在您的程序运行期间，可能会发生错误条件，例如访问向量元素超出该向量的边界，这将使您的程序停止。您可能会在屏幕上看到

以未捕获的异常终止

这里的操作词是 异常：导致正常程序流程中断的异常情况。我们说您的程序抛出了一个 异常。

Code:

```
// /boundthrow.cpp
    vector<float> x(5);
    x.at(5) = 3.14;
```

Output

```
[except] boundthrow:

libc++abi.dylib: terminating
    with uncaught exception
    of type
    std::out_of_range: vector
```

现在您知道您的程序中有错误，但不知道它在哪里发生。您可以找到，方法是尝试捕获异常。

代码:

```
1 // /catchbounds.cpp
2     vector<float> x(5);
3     for (int i; i; i)                    {
4       try {
5          x.at(i) = 3.14;
6       } catch(...) {
7          cout <<"Exception indexing at:"
      "
8               << i << '\n';
9         break;
10      }
11    }
```

Output
[except] catchbounds:

```
Exception indexing at: 5
```

23.2.2.2 常见异常

- `std::out_of_range`: 通常由使用 `at` 无效索引引起。

23.2.2.3 抛出自定义异常

抛出 异常是表示错误或意外行为的一种方式 :

```
void do_something()
{if( Oops )
throw(5);}
```

现在可以通过捕获 异常来检测这种意外行为 n:

```
try {
  do_something();
} catch (int i) {
  cout << "doing something failed: error=" << i << endl;
}
```

如果你正在进行素数项目，现在可以做一些练习。45.11。

你可以抛出整数来表示错误代码，一个包含实际错误信息的字符串。你甚至可以创建一个错误类:

类 MyError{public :int error_;no; string error_msg;};MyError(int i,string msg):error_no(i),error_msg(msg) {};}

抛出 (MyError(27,"oops");

```
try {
  // somethingcatch
} (MyError &m) {t<<"M
  cou       y error with code=" << m.error_no
    << " msg=" << m.error_msg << endl;
}
```

你可以使用异常继承!

You can multiple `catch` statements to catch different types of errors:

```
try {
  // something
} catch (int i) {
  // 处理 int 异常
} catch(std::string c) {
  // 处理 string 异常
}
```

Catch exceptions without specifying the type:

```
try {
  // something
} catch ( ... ) { // literally: three dots
  cout << "Something went wrong!" << endl;
}
```

练习 23.1。 定义函数

$$f(x) = x^3 - 19x^2 + 79x + 100$$

并计算 $f(i)$ 的值，其中 $i = 0 \ldots 20$ 是整数。

- 首先天真地编写程序，并打印出根。 $f(i)$ 为负数时在哪里？你的程序会打印什么？
- 你会发现，像负数的根这样的浮点数错误不会让你的程序崩溃或类似的情况。修改你的程序，如果 $f(i)$ 为负数，则抛出异常，捕获异常，并打印错误消息。
- 修改你的程序以测试 `sqrt` 调用 <style id='7'> 的输出，而不是它的输入。使用函数 *isnan*

```
#include <cfenv>
using std::isnan;
```

并再次抛出异常 .

A *function try block* will catch exceptions, including in *member initializer* lists of constructors.

```
f::f( int i )try: fbase(i) {// 构造函
数体 }catch(...) { // 处理异常 }
```

- 函数可以定义它们抛出的异常：

```
void func() throw( MyError, std::string );
void funk() throw();
```

- 预定义的异常：*bad_alloc*, *bad_exception*, 等。
- 异常处理器可以抛出异常；要重新抛出相同的异常，请使用不带参数的 ‘throw;’ 。
- 异常会删除所有栈数据，但不 new 数据。此外，析构函数会被调用；章节 9.4.3。
- 你的 main 函数周围有一个隐式的 try/except 块。你可以替换该处理程序。参见 *exception* 头文件。
- 关键字 *noexcept*：

```
void f() noexcept { ... };
```

- 当解引用 nullptr 时，不会抛出异常。

23.2.3 ‘这个错误从哪里来’

C++ 定义了两个宏，*__FILE__* 和 *__LINE__* ，分别提供当前文件名和当前行号。你可以使用这些宏生成漂亮的错误消息，例如

```
Overflow occurred in line 25 of file numerics.cpp
```

C++20 标准提供了 std::*source_location* 作为机制。不幸的是，目前很少有编译器支持这个特性。

```
// /location.cpp #if defined(__cpp_lib_source_location)
std::source location loc info_ __= std::source_
location::current(); cout << " 异常发生在行 "<< loc_info.line << ’ \
n’ ; #else cout<< " 异常发生在未知源位置 \n"; #endif
```

23.2.4 遗留机制

传统的错误检查方法是每个例程返回一个整数参数，该参数指示成功或失败。如果这种方法使用不一致，就会出问题，例如用户忘记遵循库的返回代码。此外，它要求函数调用层次结构的每一级都需要检查返回代码。

The*PETSc* 库在整个过程中始终如一地使用这种机制，并取得了很好的效果 .

异常是一种更好的机制，因为

- 它们不能被忽略，并且
- 它们不需要在异常抛出和捕获之间的调用层次结构级别进行处理。

而且，还有一个事实是，异常具有自动内存管理。

23.2.5 遗留 C 机制

The *errno* 变量和 *setjmp* / *longjmp* 函数不应被使用。这些函数例如不具备异常的内存管理优势。

23.3 工具

尽管你编程得再小心，你的代码仍可能计算出错误的结果或以奇怪的错误崩溃。这时有两个工具可能提供帮助：

- *gdb* 是GNU 交互式 调试器。使用它，你可以逐步运行你的代码，沿途检查变量，并检测各种条件。它还允许你在你的代码抛出错误后检查变量。
- *valgrind* 是一个内存测试工具。它可以检测内存泄漏（参见第 16.2 节），以及未初始化数据的使用。

23. 错误处理

第 24 章

标准模板库

C++ 语言拥有一个标准模板库 (STL)，其中包含被认为是标准的函数，但实际上是使用已有的语言机制实现的。STL 非常庞大，所以我们只重点介绍几个部分。

You have already seen

- 数组（第 10 章），
- 字符串（第 11 章），
- 流（第 12 章）。

使用模板类通常涉及

```
#include <something>
using std::function;
```

请参阅第
^{20.1}
节。

24.1 复数

复数 需要 `complex` 头文件。 `complex` 类型使用模板来设置精度。

```
#include
<complex>std::complex<float>
f;f.re();f.i
m();std::complex<double> d(1.,3.);
```

定义了数学运算符如 +,*，也定义了数学函数，例如 `exp`。如果标量是复数的底层类型，则涉及复数和简单标量的表达式是定义良好的：

```
complex<float> x;x  + 1.f;// 是 x  + 1.;
// 不是，因为 '1.' 是 double
```

您可以使用虚数单位 `i` 通过字面量 `i`， 如果 (float)， `il` (long)：

```
using namespace std::complex_literals;
```

Beware: `1+1i` 无法编译：您需要编写 `1.+1i` 以进行编译 iler to deduce the types.

代码：

```
1// /veccomplex.cpp2 vector
complex< complex<d                ouble> > vec1(N,
    1.+2.5i ;
3   auto vec2(vec1); //
4       * ... *
5   for ( int i=0; i<vec1.size(); ++i ) {
6     vec2[i] = vec1[i]*( 1.+1.i );
7   }
8     /* ... */
9   自动 求和 = 累积
10     ( vec2.begin(),vec2.end(),
11       复数<双精度>(0.) );
12  cou<<"res lt " u  :      sum  < '\n';
```

输出
复数向量

结果：(-1.5e+06,3.5e+06)

支持函数：

std::complex<T> conj(const std::complex<T>& z);std::complex<T> exp(const std::complex<T>& z);

24.1.1 Complexsupport inC

C 语言自 C99 起就支持复数，其类型为 s

float _Complexdouble _Complexlong double_ Complex

头文件复杂。h 提供了同义词

float 复杂 double 复杂 long double 复杂

用于这些。

See for instance https://en.cppreference.com/w/c/numeric/complex for details.

24.2 限制

继承自 C，有一个头文件 *limits。h*，其中包含 *MAX_INT* 和 *MIN_INT* 等宏。虽然这在 C++ 中仍然可用，但 STL 在 *numeric_limits* 函数中提供了更好的解决方案，该函数位于 *numeric* 头文件中。

使用头文件 *limits*：

```
#include <limits>
使用 std:: 数值 _ 限制 ;

cout <<数值 _ 限制 <长>:: 最大值 ();
```

代码：

```
1// /unsigned.cpp
2   cout <<"最大整数          : "
3   <<      ilnumer c  imits<int>::max() <<
      ’ \n’ ;
4   cout << "max unsi ned: "
5          g<< 数值_限制<无符号
     int>::{max}() <<’ \n’ ;
```

输出
[int] 限制：

最大 *int* : 2147483647
最大无符号： 4294967295

- 最大值由 *max* 给出；使用 *lowest* 表示 ‘最负值’ 。
- 最小非规格化数由 *denorm_min* 给出。
- *min* 是最小的正数，它不是非规格化的数；
- 有一个用于机器精度的 *epsilon* 函数：

代码：

```
1// /limits.cpp
2   cout << "Single lowest "
3        << numeric_limits<float>::lowest()
4        << " and epsilon "
5        << numeric_limits<float>::epsilon()
6        << ’\n’;
7   cout << "Double lowest "
8        << numeric_limits<double>::lowest()
9        << " and epsilon "
10       <<
       数值_限制<双精度>::epsilon()
11       << ’\n’;
```

输出
[stl] eps：

单精度最低 -3.40282*e*+38 和 *epsilon*
1.19209*e*-07 双精度最低 -1.79769*e*+308 和*epsilon* 2.22045*e*-16

Code:

```
// /limits.cpp
  cout << "Signed int: "
       << numeric_limits<int>::min() << " "
       << numeric_limits<int>::max()
       << '\n';
  cout << "Unsigned    "
       << numeric_limits<unsigned int>::min()
     << " "
       << numeric_limits<unsigned int>::max()
       << '\n';
  cout << "Single      "
       << numeric_limits<float>::denorm_min()
     << " "
       << numeric_limits<float>::min() << " "
       << numeric_limits<float>::max()
       << '\n';
  cout << "Double      "
       << numeric_limits<double>::denorm_min()
     << " "
       << numeric_limits<double>::min() << " "
       << numeric_limits<double>::max()
       << '\n';
```

Output

```
[stl] limits:

Signed int: -2147483648
    2147483647
Unsigned    0 4294967295
Single      1.4013e-45
    1.17549e-38 3.40282e+38
Double      4.94066e-324
    2.22507e-308 1.79769e+308
```

练习 24.1. 编写一个程序来发现最大的 n 是多少，以便 $n!$ ，即 n- 阶乘，可以表示为 int 、long 或 long long。你能把它写成模板函数吗?

除以零等操作会导致浮点数没有有效值。为了计算效率，处理器会像处理其他浮点数一样处理这些数。

24.3 容器

C++ 有几种类型的 容器。你已经见过 `std::vector` (第 10.3 节) 和 `std::array`(第 10.4 节) 和字符串（第 11 章）。许多容器有诸如 `push_back` 和 `insert` 等方法。

在本节中，我们将看看更多的容器。

24.3.1 Maps: 关联数组

数组使用整数值索引。有时你可能希望使用一个无序的索引，或者对于其排序并不重要的索引。一个常见的例子是通过字符串查找信息，例如根据一个人的名字查找其年龄。这有时被称为“ 按内容索引 ”，支持这种数据结构的形式化名称是关联数组。

在 C++ 中，这是通过一个 `map` 实现的:

```
#include <map>
using std::map;
map<string,int> ages;
```

是一个键值对的集合，其中第一个元素（用于索引）的类型是 `string`，第二个元素（被查找）的类型是 `int`。

一个 map 是通过逐个插入元素创建的：

```
#include <<map>>
<using>{v7}<std>{v9}::{v11}<make>_
<pair>{v16};<ages>.<insert>(<make>_
<pair>{v31}(<"Alice">,29));<ages>["Bob"] = 32;
```

You can range over a map:

```
for (auto person: ages) cout << person.first << " has age " << person.second <<
endl;
```

一个更优雅的解决方案使用结构化绑定 (章节 24.5):

```
for(auto person,age:ages)cout<<person<<" has age "
<<age<<endl;
```

(在这里可以使用 const-references。)

Searching for a key gives either the iterator of the key/value pair, or the `end` iterator if not found:

```
// /countint.cpp for ( auto k: {4,5} ) { auto wherek = intcount.find(k); if
(wherek==intcount.end()) cout << "could not find key"<< k << ' \n' ; else
{ auto [kk,vk] =*wherek; cout << "found key: "<< kk << " has value " <<
vk << ' \n' ; } }
```

练习 24.2. 如果你正在进行素数项目，现在可以完成第 45.6.2 节中的练习。

24.3.2 集合

集合 容器类似于 map<`SomeType`,`bool`>, 也就是说，它只表示 “ 这个元素存在 ” 。像数学中的集合，换句话说。

```
#include <<set>><std>::<set><<int>> <my>_
<ints>{v20};<my>_
<ints>{v27}.</my><insert>{v31}(<5>);<my>_
<ints>{v38}.</my><empty>{v42}(); //
predicate<my>_
<ints>{v51}.</my><size>{v55}(); // obvious
```

你可以遍历一个集合：

```
for (auto x : my_ints) // do
something
```

这将让你以无特定顺序的方式获取元素。

你可以使用 `find` 在集合中搜索，这将返回一个迭代器。如果没有找到匹配项，将返回 `end` 迭代器。

```
auto itr= my_ints. .find(6);if ( itrmy
==ints. end() ) cout <<"not found\n";
```

你可以使用模板搜索满足谓词的元素：

```
自动 res= find_i( my ints. begin(), my ints. end(), ( auto e ) {
return e>37; } );
```

集合在许多计算机算法中很有用。你经常遇到以下模式：

```
while (notdone) {for ( x in unprocessed) {if (somethingwithx){processed.add(x);unprocessed.remove(x);}}}
```

参见 HPC 书籍 [11], 章节 10.1.1。

24.4 正则表达式

标头 `regex` 为 C++ 提供了正则表达式匹配功能。例如，`regex_match` 返回一个字符串是否完全匹配一个表达式：

代码：

```
// /regexp.cpp
auto cap = regex("[A-Z][a-z]+");
for (auto n:
      {"Victor", "aDam", "DoD"}
      ) {
  auto match =
    regex_匹配 ( n, cap);
  cout<< n; if (match) cout ": yes";

  else       cou  <<  : no  ;
  cout << '\n';
}
```

输出

```
[正则表达式] 正则表达式:

看起来像名字:
Victor: 是
aDam: 否
DoD: 否
```

(注意正则表达式匹配子字符串，但 `regex_match` 仅当整个字符串匹配时才返回 true。

要查找子字符串，使用 `regex_search`：

- 该函数本身评估为 `bool`;
- 有一个可选的返回参数类型为 *smatch* (‘字符串匹配’)，包含匹配信息。

The *smatch* 对象有这些方法：

- `smatch::position` 表示表达式匹配的位置，

- while *smatch::str* returns the string that was matched.
- *smatch::prefix* has the string preceding the match; with *smatch::prefix().size()* you get the number of characters preceding the match, that is, the location of the match.

Code:

```
1  // /regsearch.cpp
2    {
3      auto findthe = regex("the");
4      auto found = regex_search
5        ( sentence,findthe );
6      assert( found==true );
7      cout << "Found <<the>>" << '\n';
8    }
9    {
10     smatch match;
11     auto findthx = regex("o[^o]+o");
12     auto found = regex_search
13       ( sentence, match ,findthx );
14     assert( found==true );
15     cout << "Found <<o[^o]+o>>"
16          << " at " << match.position(0)
17          << " as <<" << match.str(0) <<
       ">>"
18          << " preceeded by <<" <<
       match.prefix() << ">>"
19          << '\n';
20   }
```

Output
[regexp] search:

```
Found <<the>>
Found <<o[^o]+o>> at 12 as
    <<own fo>> preceeded by
    <<The quick br>>
```

24.4.1 正则表达式语法

C++ 使用了一种国际正则表达式语法的变体。http://ecma-international.org/ecma-262/5.1/#sec-15.10. 请查阅该文档以了解转义字符和更多信息。

如果你的正则表达式因为转义字符等原因变得过于复杂，可以考虑使用原始字符串字面量 结构。

24.5 Tuples and structured binding

还记得在章节 7.5.2 中我们提到吗，如果你想要返回多个值，你不能通过返回值来实现，而必须使用一个 输出参数？ 好的，使用 STL 有一个不同的解决方案。

你可以创建一个 元组，它是一个包含多个组件（可能为不同类型）的实体，并且与 **结构体** 不同，你不需要事先定义它。（对于恰好包含两个元素的元组，使用 *pair*。）

在 C++11 中，您会使用 *get* 方法从一对或元组中提取元素：

```
#include <tuple>

std::tuple<int,double,char> id = \
    std::make_tuple<int,double,char>( 3, 5.12, 'f' );
    // or:
    std::make_tuple( 3, 5.12, 'f' );
double result = std::get<1>(id);
std::get<0>(id) += 1;

// also:
std::pair<int,char> ic = make_pair( 24, 'd' );
```

烦恼：所有‘get’ting。

这看起来不太优雅。幸运的是，C++17 可以使用指称和 `auto` 关键字来使这显著更短。考虑一个返回元组的函数。你可以使用 `auto` 来推断返回类型：

```
// /tuple.cpp auto maybe_
root1(float x) { if (x<0) return
make_tuple<bool,float>(f
alse,-1); else return make_tuple<
bool,float>(true,sqrt(x)); };
```

但更有趣的是，你可以使用一个元组指称：

```
// /tuple.cpptuple<bool,
float> maybe_root2(float x) {
if (x)      return {false, -1};
else return {true, sqrt(x)};};
```

这里指明了函数的返回类型，但返回语句并没有显式地构造这种类型。

调用代码特别优雅：

Code:

```
// /tuple.cpp
    auto [succeed,y] = maybe_root2(x);
    if (succeed)
      cout << "Root of " << x
           << " is " << y << '\n';
    else
      cout << "Sorry, " << x
           << " is negative" << '\n';
```

Output

```
[stl] tuple:
Root of 2 is 1.41421
Sorry, -2 is negative
```

This is known as *structured binding*.

上述示例，其中元组的 `bool` 组件，可以以不同的方式更好地表述，如下所示。结构化绑定更合适的用法是遍历一个 map （第 24.3.1 节）：

```
for (const auto &key, value : mymap) ....
```

24.6 组件 -like stuff: 元组、可选、变体、预期

有些情况下，你需要一个值，它可以是多种类型之一，例如，如果计算成功，返回一个数字，否则返回一个错误指示符。

让我们考虑一个平方根函数的示例，如果你用负数调用它，它会返回一个错误。最简单的解决方案是有一个函数，它既返回一个 bool，也返回一个数字：

```
// /optroot.cpp bool RootOrError(float &x) { if (x<0)
return false; else x =std::sqrt(x); return true; }; /* ... */ for (
auto x: {2.f,-2.f} ) if (RootOrError(x)) cout << "Root is " <<x <<'
\n'; else cout << "could not take root of "<< x << '\n';
```

这个解决方案存在一些不优雅之处。首先，你使用了两种不同的机制来返回两个值。此外，你实际上不需要两个值：如果有错误，返回的值无关紧要。

现在，我们将考虑一些更符合 C++17 习惯的解决方案。

24.6.1 元组

使用元组或对（章节 24.5 ）上述 ‘一个数字或一个错误’ 的解决方案现在变成了一个包含 `bool` 和一个 `float`的元组：

```
// /optroot.cpp #include <tuple>using std::tuple,
std::pair; /* ... */ pair<bool,float> RootAndValid(float x) { if (
x<0) return {false,x}; else return {true,std::sqrt(x)}; }; /* ... */
for ( auto x : {2.f,-2.f} ) if ( auto [ok, root] =
RootAndValid(x) ; ok ) cout << "Root is " <<root << '\n';
else cout << "could not take root of "<< x << '\n';
```

注意条件语句中的初始化语句 。

这种解决方案更好，因为布尔值和结果现在通过函数结果一起返回，而不是通过引用参数返回，但我们仍然存在冗余问题。

24.6.2 可选

‘一个数字或一个错误’ 的最优雅解决方案是有一个单一的数量，你可以查询它是否有效。为此，C++17 标准引入了可空类型的概念：可空类型：一种可以某种方式传达它是空的类型。

在这里我们讨论 `std::可选`。

```
#include <可选>
using std::可选;
```

24.6.2.1 创建可选

我们正在编写的函数有一个返回类型为 `可选<float>`。这是通过返回实际 `float`，或 `{}`，其是 `std::nullopt`。

```
#include <optional>
using std::optional;

optional<float> f {
  if (something) return 3.14;
  else return {};
}
```

24.6.2.2 获取可选值（如果存在）

您可以使用方法 `has_value` 测试可选量是否实际包含值，在这种情况下，您可以使用方法 `value` 提取量：

```
auto maybe_x = f(); if (f.has_
value()) // do something with f.
value();
```

尝试获取某个不存在值的对象会导致异常 _optional_ 访问异常：

代码：

```
// /optional.cpp 可选<float> 可能_数字 =
{};
try {
  cout << maybe_number.value() << '\n';
} catch (std::bad_optional_access_optional_access) {
  cout<< "failed to get value\n";
}
```

输出

[联合] 可选

failed to get value

存在一个名为 `value` 的函数 _，它返回值，或者如果可选值没有值，则返回默认值。

```
可选 <int> 一些_int; 一些
_int.值_或(-1);
```

练习 24.3. 编写 *RootOrError* 函数使用 std::optional.

练习 24.4. 如果你正在进行素数项目，现在可以完成练习 45.17.

练习 24.5. The 八皇后 问题 (章节 48) 可以使用 std::optional. 参考章节 48.3 以了解测试驱动开发 (TDD) 方法。

备注 26 如果你有一个可选类对象，你可以使用 *emplace* 方法分配该对象：

```
//optional.cppclass WithInt {public :
WithInt(int i ) {} ;void foo() {}; }/* ...
*/optional<WithInt> maybe; {
maybe.emplace(5); }cout << maybe.has_
value() <<'\n'; maybe.value().foo();
```

24.6.3 预期

上一节的 std:: 可选 对于某些情况（例如平方根）很有用，在这些情况下，如果值不存在，则很明显它的含义是什么。但是，如果不存在是由于某种错误引起的，您可能想知道其原因。

C++23 的 std:: 预期 （在 预期 标头中）允许您返回一个值，或提供更多关于为什么该错误不存在的信息。

期望 double，如果不符合则返回信息字符串：

```
std:: 期望<double,stri     ng>
      square_root( double x ) {
  auto result= sqrt(x);
  if (x<0)
  返回
  else if (x<限制<double>::min())
  返回
    std::意外的("负数");
  返回结果；
}
```

```
auto root = square_root(x);
if (x)
cout <<"Root"=" " << root.{v12}value() <<
    ' \n' ;
else if   (root.error()==/* et cetera
    */ )
/* handle the problem */
```

24.6.4 Variant

在 C 语言中，一个 **联合体** 是一种可以有多种类型的实体。类似于 C 数组不知道其大小，一个 **联合体** 也不知道它是什么类型。C++ 变体 (C++17) 不受这些限制。

在上述 可选 解决方案中，我们有一个 值 函数。由于一个 变体 可以有多种类型，我们不能直接有这样的函数。相反，我们需要测试返回的是什么类型，并使用获取 或获取 _ 如果 来按类型检索值。

作为一个第一个例子，我们考虑平方根示例。

```
// /optroot.cpp
#include   <variant>
使用 std::variant, std:: 获取 _ 如果 ;/
/     * ... *
变体 <bool,float>
    根变体 (float x)        {
  if (x<0)
    return false;
  else
    return std::sqrt(x);
};
```

```
//  /optroot.cpp
  for (auto x : {2.f,-2.f}) {
    auto okroot =  RootVariant();
    auto root =
    get_if<float>(&okroot);
    if ( root )
      cout <<"Root is"      " << *root
    << '\n';
    auto nope =
    get_if<bool>(&okroot);
    if (nope)
        cout <<"could not take root f"
    o<<    << x\"      n ;
  }
```

展示更多使用 int 、 double 、 string 变体的可能性：

```
1// /intdoublestring.cpp2 variant<int, double,
string>union_ids;
```

我们可以使用 *index* 函数来查看使用的变体是什么（在这种情况下是 0 、 1 、 2 ），并 *get* 相应地获取值：

```
1// /intdoublestring.cpp2 union_ids = 3.5;3 switch( union_ids.index() )
{4 case1 :5 cout << "Double case: "<< std::get<double>(union_ids) << '
\n';6   }
```

获取错误的变体会导致异常 _ 变体 _ 访问异常：

```
1// /intdoublestring.cpp 2 union_ids = 17; 3 cout <<" 使用选项 " << union_ids.index() << ": "
<< get<int>(union_ids)<< '\n'; 4 try { 5 cout << " 作为双精度获取 : " <<get<double>(union_
ids) << '\n'; 6 } catch ( bad_variant_access b ) { 7 cout << " 作为双精度获取时发生异常，索引 ="
<< union_ids.index() <<'\n'; 8 }
```

使用 get_if 更安全，它接受一个指向变体的指针，如果成功则返回一个指针，否则返回一个空指针：

```
1// /intdoublestring.cpp 2 union_ids ="Hello world"; 3 if ( auto union_int = get_if<int>(&union_
ids) ; union_int ) 4 cout << "Int: "<< *union_int << '\n'; 5 else if ( auto union_string = get_if<
string>(&union_ids) ; union_string ) 6 cout << "String: " <<*union_string <<'\n';
```

注意 get_if 的参数是 variant 的地址，返回结果也是指针。

Exercise 24.6. Write a routine that computes the roots of the quadratic equation

$$ax^2 + bx + c = 0.$$

该例程应返回两个根，或一个根，或表示方程无解的指示。

代码：

```
// / appfor ( auto coefficients:

        {make_tuple(2.0, 1.5, 2.5),}
          make_tuple(1.0, 4.0, 4.0),
          make_tuple(2.2, 5.1, 2.5)
        }){
    auto result =
    compute_roots(coefficients);
```

Output

[联合] 二次

With a=2 *b*=1.5*c*=2.5
无根
With a=2.2*b*=5.1*c*=2.5
根 *1*: -0.703978 根 *2*: -1.6142*ih* 1 *b* 4 4
W t a= = c=
单根 : -2

在这个练习中你可以返回一个布尔值来表示 “ 无根 ”，但布尔值可以有两个值，只有一个有意义。对于这种情况，有 单态 类，也来自 变体 头文件。然后你可以返回该类的对象，如下所示：

```
return std::monostate{};
```

24.6.4.1 The same function on all variants

假设你有一些类的变体，它们都支持一个原型相同的方法：

```
类 x_ 类型 {public:r_ 类型方法 () { ...
};}; 类y_ 类型 {public:r_ 类型方法
() { ... };
```

无法直接在变体上调用这个方法：

```
变体 <x_ 类型 , y_ 类型 > xy;// 错
误 xy.method();
```

为了一个具体的例子，我们定义了两个类，一个包含一个 double，另一个包含一个字符串。这两个类都有一个用于将对象渲染为字符串的方法。

```
1// /visit.cpp2  variant<double_ 类,string_ 类>3 联
合 _ 是 _double{ double_ 类 (2.5)},4 联合 _ 是 _
string{ string{"two-point-five"}};
```

其中我们定义了方法 `stringer` 来获取字符串表示，以及 `sizer` 来获取对象的 ‘大小’。后者：

```
// /visit.cpp 类 sizer{public:int 方法 ()(
double_ 类 d) {returnstatic_ 转换 <int>(
d.value());};int 方法 ()( string_ 类 s)
{return s.value().size();};};
```

解决这个问题的方法是 `visit`，它来自 `variant` 头文件。现在我们不再使用这些单独的方法，而是创建一个函数对象：一个带有重载方法 () 的类。这用于将定义在下方的一个对象应用于 variant 对象：

代码：

```
1// /visit.cpp2
3   cout <<"Size of "
      << 访问 (i
      str nger{} union_is_double )
4         , <<" >> 是 "
5       << 访问 (
      sizer{}, union_is_double )
6       << '\n';
7   cout << "Size of <<"
8       << visit(
      stringer{},union_is_string )
9       << ">> is "
10      << visit(
      sizer{},union_is_string )
11      << '\n';
```

输出

[联合] 访问：

```
<<2 的大小 .5>> 是 2Sif<<titf
  ze o     wo-po n - ive>>
    是 14
```

24.6.5 任意

While `variant` 可以是多种预定义类型之一， `std::any` 可以包含任何类型。因此它等同于 `void*` 在 C 中。

一个 `any` 对象可以用 `any_cast` 进行转换：

```
std::any a{12};std::any_cast<int>
(a); // 成功 std::any_cast<string>
(a); // 失败
```

24.7 随机数

STL 拥有一个 随机数生成器 ，它比 C 版本更通用、更灵活（章节 24.7.5 ）将在下文中讨论。

与其使用单个调用生成随机数，这里有一个两步过程。首先有一个随机引擎，它包含数学随机数生成器。然后，您代码中使用的随机数是通过将分布应用于此引擎生成的。

Code:

```
// /xrand.cpp
  // seed the generator
  std::random_device r;
  // set the default random number
    generator
  std::default_random_engine
    generator{r()};
  // distribution: real between 0 and 1
  std::uniform_real_distribution<float>
    distribution(0.,1.);

  for ( int i=0; i<5; i++)
    cout << "random: "
         << distribution(generator)
         << '\n';
```

Output

```
[random] xrand:

missing snippet
../code/random/xrand.runout
```

24.7.1 生成器

存在一个实现定义的生成器`默认_随机_引擎`：

```
std:: 默认_随机_引擎生成器 ;
```

以这种方式使用时，生成器每次都会从相同的值开始。要为其设置种子，请使用`随机_设备`：

```
std::random_device r;
std::default_random_engine generator{ r() };
```

这通常依赖于操作系统函数来查找随机种子。

其他生成器包括 梅森旋转算法 `mt19937` 、 `knuth_b` 以及其他几个。它们在维护内部状态所需的空间以及周期的长度上有所不同。

24.7.2 分布

生成随机数的最常见模式是从分布中选取。例如，给定范围内的均匀分布：

```
std::均匀_real_分布<float> 分布(0.,1.);
```

Random numbers from the unit interval:

```
// /xrand.cpp // 初始化生成器 std::random_device
r;// 设置默认随机数生成器

  std::default_random_引擎生成器{r()};
  // 分布：0 到 1 之间的实数
  std::均匀 _real_分布<浮点数> 分布(0.,1.);

  for (int i=0; i<5; i++) cout << "random: "
  << distribution(generator) << '\n';
```

掷骰子的结果会是：

```
std::uniform_int_分布<int> 分布(1,6);
```

```
// 设置默认生成器
std::default_random_引擎生成器;

// 分布：整数 1..6_   _
std::uniform int distribution<int> distribution(1,6);

// 将分布应用于生成器:
int dice_roll= distribution(generator);// 生
  成 1..6 范围内的数字
```

As other distributions在这里有泊松分布、伯努利分布、正态分布、卡方分布 and several more.

泊松分布的整数：

```
std::default_random_engine generator;
float mean = 3.5;
std::poisson_distribution<int> distribution(mean);
int number = distribution(generator);
```

Exercise 24.7. Chapter 65 有一个使用随机数模拟随机游走的案例研究 .

24.7.3 使用场景

在某些情况下，你可能需要，换句话说，一堆随机数生成器。例如，在模拟中，你可以有多个对象，每个对象都采取随机行动。

类 *Thing { public: void do_something_random() { rnd =*
some_distribution(some_generator); f(rnd); }; }; int
*main() { for (many iterations) { Thing t; t.something_*
random(); } };

您可能会想，不仅不在需要它的例程中包含随机数生成器（ RNG ）的调用，还包含其定义。

错误方法：随机数生成器在函数中局部定义。

代码：

```
1// /static.cpp
2int 非随机_int(int 最大) {
3   std::默认_随机_引擎 引擎;
4   std::均匀整数分布<>_   _
      整数(1, 最大);
5   返回 整数(引擎);
6 };
```

输出
[rand] 非随机：

三个整数：15, 15, 15.

这不会起作用，因为每次调用函数时都会初始化生成器。您可以通过将生成器变量声明为静态来解决这个问题。

良好的方法：函数中的单个随机生成器静态 n.

Code:

```
1 // /static.cpp
2int realrandom_int(int max){i}_      _
3   stat c  std::default random engine
      static_引擎;
4   std::均匀整数分布<>_   _
      整数(1, 最大);
5   返回 整数(静态_引擎);
6 };
```

输出
[随机数] 真随机：

三个整数：15, 98, 70.

注释 27 类似的诊断和解决方案适用于 *ob* *jects:*

```
类 Thing {private:// 应该是静态的，两次
random_设备 r; 默认_random_引擎生成器 { r() };
public:voiddo_something_random() {
```

```
rnd = some_分布(生成器); f(rnd); }; };
```

一些随机数生成器具有大量的内部状态，所以这会使事物对象变得不必要地大。

如果你有多个使用随机数的例程怎么办？为了使所有内容在统计上合理，你可以将 RNG 移动到一个单独的例程中：

```
// /static.cppintrealrandom_int(int max) {static
std::default_random_enginestatic_
engine;std::uniform_int_分布<> ints(1,max);return
ints(static_engine);};
```

为了清理这个设计，你甚至可以将这个放入一个具有内联静态类方法的类对象中：

Note the use of **static**:

```
// /randname.cpp
class generate {
private:
  static inline std::default_random_engine engine;
public:
  static int random_int(int max) {
    std::uniform_int_distribution<> ints(1,max);
    return ints(generate::engine);
  };
};
```

Usage:

```
auto nonzero_percentage = generate::random_int(100)
```

24.7.4 排列

函数 *shuffle* 打乱一个数组。与 *iota* 函数（来自 *numeric* 头文件）结合，可以轻松得到一个排列：

Code:

```
1 // /shuffle.cpp
2     std::vector<int> idxs(20);
3     iota(idxs.begin(),idxs.end(),0);
4     /* ... */
5     std::shuffle(idxs.begin(),
      idxs.end(), g);
```

Output

```
[rand] shuffle:

Iota:
  0   1   2   3   4   5   6
    7   8   9
 10  11  12  13  14  15  16
   17  18  19
Permute:
  6   9   4   3  16  15  18
    5   2  11
 14  19  17   1   0  13   7
   10  12   8
```

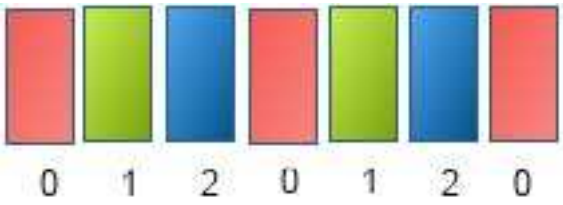


图 24.1：随机数生成器的低数字偏差（取模）

(其中 `g` 是一个随机生成器。)

24.7.5 C 随机函数

C 语言提供了多个 随机数生成器。它们包含线程安全的变体，其中状态被显式传递，因此可以存储在线程私有存储中。

- `rand48`,`rand48_r` a 48 位乘法生成器和线程安全的变体；
- `rand`, `rand_r` 是目前最常见的；
- `random`, `random_r`: 一个更好的生成器，符合最新的 *POSIX* 标准。

函数 `rand` 生成一个 `int` – 每次调用时都不同 – 在零到 `RAND_MAX` 的范围内。通过缩放和强制转换，您可以使用上述代码生成一个介于零和一之间的分数。

```
#include <<random>>
<using><std>::<rand>;</float><random
fraction>=(<float>)<rand>()/(<float>)<RAND>_
<MAX>;</
```

这个生成器存在一些问题。

- C随机数生成器的周期可以小到 2_{15}。
- 只有一个生成算法，该算法依赖于实现，并且对其质量没有任何保证。
- 没有将序列转换为范围的机制。常见的惯用法

  ```
  int under100 = rand() % 100
  ```

  倾向于小数。图 24.1 显示了对于周期为 7 的生成器模 3 的情况。

如果你运行两次程序，你会两次得到相同的随机数序列。这对于调试程序很好，但如果你希望进行一些统计分析，那就不好了。因此你可以通过 随机数种子 设置从哪个随机序列开始 `srand` 函数。示例：

```
srand(time(NULL));
```

从当前时间初始化随机数生成器。这个调用只应该发生一次，通常在主程序的高处。

24.8 时间

Header

```
#include <chrono>
```

方便:

using namespace std::chrono

但在这里我们全部说明清楚。

24.8.1 时间间隔

您可以定义以秒为单位的间隔:

```
///
second.cppseconds
s{3};auto t = 4s;
```

Remark28 使用以下之一:

```
std::chrono::seconds two{2};

using std::chrono;
seconds two{2};

using namespace std::chrono;
seconds two{2};
```

您可以对这种类型执行算术和比较操作:

代码:

```
1// /second.cpp2
3     cout <<"This lasts "
          << s.count() << "s" << '\n';
4     cout <<"This lasts ";
5     打印 _ 秒 ( s+5s );
6     auto 九 .14*3s;
7     cout << 九 . 计数 ()
8           {s} 小于 10 秒:
9           <<boolalpha << ( 九 <10s)
10          <<' \'n ;
```

输出
chrono 基本秒

```
这持续了 3
这持续 8 秒
9.42 秒小于 10 秒 : 为真
```

注意，虽然 秒 接受一个整数参数，但你可以将其乘以或除以以获得小数值。

有一个持续时间 毫秒，你可以隐式地将秒转换为毫秒，但反过来你需要持续时间 _ 计数:

```
// /second.cpp
print毫秒 ( 5s);// DOES NOT COMPILE print_ 秒(
6ms );print_ 秒 ( 持续时间 _ 转换 < 秒 >(6ms));
```

持续时间的完整列表（带后缀）是： 小时 (1h), 分钟 (1min), 秒 (1s), 毫秒 (1ms), 微秒 (1us), 纳秒 (1ns).

24.8.2 时间点

一个 时间点 可以被认为是从某个起始点开始的一段持续时间，例如 Unix 纪元的开始：1970 年的开始。

```
时间_点<系统_时钟，秒> tp{10'000s};
```

是 2 小时 +46 分钟 +40 秒 后的时间。

您通过在时间点上调用时间_自纪元以来_纪元 方法来明确这一点，从而得到一个持续时间。

24.8.3 时钟

有几种时钟。常见的提供时钟是

- 系统_时钟用于与日历有关的时间点；和
- 稳定_时钟用于精确测量。

通常，高_分辨率_时钟是其中任一的同义词。

一个时钟有属性：

- 持续时间
- 重复次数
- 周期
- 时间_点
- 是_稳定的
- 并且一个方法现在()。

如上所述，一个时间_时间点与一个时钟相关联，不同时钟的时间点不能相互比较或转换。

24.8.3.1 持续时间测量

要测量一段执行的持续时间，使用时钟的现在方法，在段之前和之后。减去时间点会得到以纳秒为单位的持续时间，你可以将其转换为任何其他类型：

Code:

```
// /clock.cpp
using clock = system_clock;
clock::time_point  before =
  clock::now();
std::this_thread::sleep_for( 1.5s );
auto after = clock::now();
cout << "Slept for "
     <<
  duration_cast<milliseconds>(after-before).count()
     << "ms\n";
```

Output

```
[chrono] clock:
Slept for 1503ms
```

(The 睡眠函数不是一个 *chrono* 函数，但它来自线程头文件；参见章节 25.1.4.)

24.8.3.2 时钟分辨率

The 时钟分辨率 可以从周期 属性中找到：

auto num = myclock::::period::num,den = myclock::::period::den; auto tick = static cast double < > (num {v51}) / static cast double _ < (den {v64});

Timing:

auto start_time= myclock::now(); auto duration= myclock::now()-start_time; auto microsec duration =_std::chrono::duration_cast<std::chrono::microseconds> (duration); cout << "This took " << microsec_duration.count() <<"usec" << endl;

计算新的时间点：

```
auto 截止日期 = myclock. 现在 () + std::chrono:: 秒 (10);
```

24.8.4 不再使用的 C 机制

让进程休眠：*sleep*

时间测量：*getrusage*

24.9 文件系统

截至 C++17 标准，存在一个文件系统头文件，`filesystem`，其中包含诸如目录遍历器之类的内容。

```
#include <filesystem>
```

24.10 枚举类

C 风格的 < 样式 id='1'>enum</ 样式 > 关键字引入了全局名称，因此容易发生名称冲突。例如：

```
/ < 样式 id='1'>//enumnot.cpp</ 样式 >< 样式 id='3'>enum </ 样式 >< 样式 id='5'>colors</ 样式 >< 样式 id='7'>{ </ 样式 >< 样式 id='9'>red</ 样式
>< 样式 id='11'>,</ 样式 >< 样式 id='13'>yellow</ 样式 >< 样式 id='15'>,</ 样式 >< 样式 id='17'>green </ 样式 >< 样式 id='19'>};</ 样式 >< 样式
id='21'>cout </ 样式 >< < 样式 id='24'>red </ 样式 >< < 样式 id='27'>"," </ 样式 >< < 样式 id='30'>yellow</ 样式 >< < 样式 id='33'>"," </
样式 >< < 样式 id='36'>green </ 样式 >< < 样式 id='39'>'\n'</ 样式 >< 样式 id='41'>;</ 样式 >< 样式 id='43'>enum</ 样式 >< 样式
id='45'>flag</ 样式 >< 样式 id='47'>{ </ 样式 >< 样式 id='49'>red</ 样式 >< 样式 id='51'>,</ 样式 >< 样式 id='53'>white</ 样式 >< 样式
id='55'>,</ 样式 >< 样式 id='57'>blue</ 样式 >< 样式 id='59'>}; </ 样式 >< 样式 id='61'>// 冲突！ </ 样式 >
```

< 样式 id='1'> 在 C</ 样式 >++ < 样式 id='4'> 中引入了 </ 样式 >< 样式 id='6'>enum class </ 样式 >(或 < 样式 id='8'>enum struct</ 样式 >)，这使得名称成为类的成员：

```
/ < 样式 id='1'>//enumclass.cpp</ 样式 >< 样式 id='3'>enum class </ 样式 >< 样式 id='5'>flowers </ 样式 >< 样式 id='7'>{ </ 样式 >< 样式 id='9'>grass</ 样
式 >< 样式 id='11'>,</ 样式 >< 样式 id='13'>poppy</ 样式 >< 样式 id='15'>, </ 样式 >< 样式 id='17'>bluebonnet </ 样式 >< 样式 id='19'>};</ 样式 >< 样式
id='21'>vector</ 样式 >< 样式 id='24'>flowers</ 样式 >> < 样式 id='27'>field</ 样式 >< 样式 id='29'>(10,</ 样式 >< 样式 id='31'>flowers</ 样式 >< 样式
id='33'>::</ 样式 >< 样式 id='35'>grass</ 样式 >< 样式 id='37'>);</ 样式 >< 样式 id='39'>field</ 样式 >[1] =< 样式 id='42'>flowers</ 样式 >< 样式
id='44'>::</ 样式 >< 样式 id='46'>poppy</ 样式 >< 样式 id='48'>;</ 样式 >< 样式 id='50'>field</ 样式 >[5] =< 样式 id='53'>flowers</ 样式 >< 样式
id='55'>::</ 样式 >< 样式 id='57'>bluebonnet</ 样式 >< 样式 id='59'>;</ 样式 >
```

默认情况下，枚举类型的变量被分配为 `int`。您可以通过让类型继承自另一个类型来更改此设置：

```
// /enumclass.cpp
  enum class flag : unsigned short { red,white,blue };
```

如果此类继承自一个整型，这并不意味着它表现得像整数；例如，你不能立即询问一个是否小于另一个。相反，它只决定了枚举项占用的空间大小。

要让它表现得像整型，你需要将其强制转换：

```
// /enumclass.cpp cout <<static_cast<unsigned short>( flag::red ) << ",
"<<static_cast<unsigned short>( flag::white ) << ","<<static_cast<
unsigned short>( flag::blue ) << ’\n’;
```

In C++23 这可以更简洁地使用 *to_underl ying*:

```
// /enumclass.cpp enum class flag : unsigned int{
red,white,blue }; // 但我们仍然需要将它们转换为 cout << std::to_
underlying( flag::red ) << ","<< std::to_underlying(
flag::white ) <<","<< std::to_underlying( flag::blue ) <<’\n’;
```

备注 29 而不是使用 ***enum class***，你通常也可以将 ***enum*** 包围在一个类的命名空间中：

```
// /spaced.cppclass Field {public :enum color { invalid{-1}, white,
red, blue, green };private : color mycolor;public :void set color( color c )
{ mycolor = c; }/* ... */Field onefield; onefield.set color( Field::color::blue );
```

24.11 排序和 ‘飞船’ 运算符

可以为类重载比较运算符 `<, <=, ==, >=, >, !=` ；对于类；章节 9.5.7<style id='25'>。但是，即使你用另一个运算符表示一个运算符，这样做通常也有很多冗余：

```
运算符>=( const MyClass& 其他 )
{return not(其他<*this);};
```

The C++20 飞船 运算符 `<=>` 可以让生活更轻松。这个运算符返回一个排序对象，该对象可以是六种关系之一。

让我们用一个简单的例子来说明。类 `Record` 的每个对象都有一个字符串和一个唯一编号：

```
// /
spaceship.cpp类
Record
{private:
```

static inline int nrecords{0};int myrecord{-1};string name;public::Record(string name) : name(name)(name){ myrecord =nrecords++; }

现在你需要一个完全基于记录编号的排序关系。

代码：

```
1 //   s /p p ppRecord alice("alice"),bob("bob");
2
3   cout <<"expect t f t t f f\n";
4   cout <<boolalpha << (alice==alice)
      << '\n';
5   cout <<boolalpha << (alice==bob) <<
      \n;
6   cout << boolalpha<< (alice<=bob) <<
      \n;
7   cout << boolalpha << (alice<bob) <<
      \n;
8   cout << boolalpha << (alice>=bob) <<
      '\n';
9   cout << boolalpha << (alice>bob) <<
      '\n';
```

输出[
l]st **spacerecord:**

期望 t f t t f f
true
false
true
true
false
false

For this we impl24.11 排序和 “ 飞船 ” 运算符 g on integers exists:

// /spaceship.cppstd::::strong_orderingoperator<=>((const Record& other) {return myrecord<=>other.myrecord; }bool operator==((const Record& other) {return myrecord==other.myrecord; }

请注意，此比较的返回类型是 std::strong_ordering，因为在这种情况下，任何两个对象 x, y 总是满足其中一条

x<y x==y x>y

为了更复杂的例子，让我们创建一个坐标类，其中两个坐标在所有组件都满足该关系时比较为 <、=、> 。这是一种偏序关系，意味着某些坐标 x、 y 不满足任何

x<y, x==y, x>y

代码：

```
// /spaceship.cpp
坐标 <int>
    p12<1,2>,p24<2,4>,p31<3,1);co t <<"
expect t t f\n"                    ;
cout << boolalpha << (p12==p12) <<
  '\n';
cout <<boolalpha << (p12<p24)<<'\ '
    n
《C++17/fortran2008 科学编程艺术，HPC 卷 3》第 24.11 章：排
序和 “ 飞船 ” 运算符；cout boolalpha (p p31 or
  12>p31 or p12==p31) '\n';
```

输出 [stl] 空间部分：

```
expect t t f
true
true
false
```

在这种情况下，spaceship 运算符返回一个 部分 _ 排序 结果。此外，我们需要再次单独定义相等性。

```
// /spaceship.cpp std::partial_ordering operator<=>( const Coordinate& other ) const {
std::strong_ordering c= the_array[0] <=> other.the_array[0]; for (int i = 1; i <the_
array.size(); ++i) { if ((the_array[i] <=> other.the_array[i]) != c) return std::partial_
ordering::unordered; } return c; }; bool operator==( constCoordinate& other ) const { for
( int ic=0; ic<the_array.size(); ++ic ) if (the_array[ic]!=other.the_array[ic]) return false;
return true; };
```

24. 标准模板库

第 25 章

并发

并发是一个复杂的话题。它与并行性既有相似之处，也有不同之处，而并行性是现代科学计算的一个主要关注点，因此它既与本书的目标读者相关，又不太相关。

从某种意义上说，并行性很明确：它是指同时发生的事情，例如在您现代处理器中的多个核心上。并行性的好处是您的代码运行得更快，挑战在于如何编写您的程序来表示确实存在可以同时完成的事情。

从词源上看，“并发”似乎也指同时发生的事情。然而，它早于实际的并行处理器：在操作系统中，即使是在单个处理器上，并发也很重要，因为您有一个程序在同时运行，而其他程序在打印、检查邮件等等。

因此，并发可以定义为对没有明确时间关系活动的学习。研究的主要问题是如何处理“共享状态”：被多个进程以没有明确顺序访问的对象。

并发性在操作系统、数据库和 *Web* 服务器等领域非常重要。并发性的主要机制是线程，我们将在本章中介绍。C++20 标准还添加了协程，但我们在这里不涉及。

对于科学计算，并行性比并发性更重要；为此，我们参考本系列的第二卷，它涵盖了 *MPI* 和 *OpenMP* 系统，这两个系统都可以轻松地从 C++ 使用。

25.1 线程创建

这使用了 *thread* 头文件：

```
#include <线程>
```

线程是类 std::*thread* 的对象，创建它就会立即开始它的执行。线程构造函数至少需要一个参数，一个可调用对象（一个函数指针或一个 lambda），以及该函数式对象的可选参数。

调用线程的环境需要调用 *join* 以确保线程的完成。在您这样做之前，无法保证线程的函数已经运行。

作为一个非常简单的例子，这里有一个睡眠一秒钟的线程。注意：这使用了操作系统 *sleep* 函数；有更好的机制；请参阅章节 25.1.4。

Code:

```
// /block.cpp
auto start_time = Clock::now();
auto waiting_thread =
  std::thread( [] () {
    sleep(1);
  }
    );
waiting_thread.join();
auto duration =
  Clock::now()-start_time;
```

Output
[thread] block:

```
This took 1.00136 sec
```

一个接受参数的函数示例：

```
#include <线程>

auto somefunc = [] (int i, int j) { /* stuff */ }; std::thread
mythread(somefunc, arg1, arg2); mythread.join()
```

25.1.1 多线程

创建单个线程用处不大。通常你会创建多个线程来细分工作，然后等待它们全部完成。

```
vector<thread> mythreads;
for ( i=/* stuff */ )
  mythreads.push_back( thread( somefunc,someargs(i) ) );
for ( i=/* stuff */ )
  mythreads[i].join();
```

注意，我们使用大小为零的 vector 创建线程，并使用 `push_back` 来填充它，因为一个 thread 开始执行。如果立即用正确的尺寸创建 vector，将会创建一堆无操作的 threads。

练习 25.1。 如果你非常关心分配的成本，如何在不创建 threads 的情况下为 threads 创建空间。

这里是一个简单的 hello world 示例。由于没有关于 threads 何时开始或结束的顺序保证，输出可能会显得混乱：

代码:

```
// /hello.cpp
    vector< std::thread > threads;
    for ( int i=0; i<NTHREADS-1; ++i ) {
      threads.push_back
        ( std::thread(hello_n,i) );
    }
    threads.emplace_back
      ( hello_n,NTHREADS-1 );
    for ( auto& t : threads )
      t.join();
```

Output
[线程] hellomess:

```
你好 你好 01

Hello 2
Hello 3
Hello 4
```

(注意对 `emplace_back` 的调用：由于完美转发，它可以在线程参数上调用构造函数。)

我们通过包含一个等待来使这条消息有序。作为一个单独的问题，线程现在执行一个由 lambda 表达式给出的函数：

代码：

```
// /hello.cpp 线程。推_入

        ( std::thread
          ( /* 函数： */
            [] (int i ) {
              std::chrono::seconds wait(i);
              std::this_thread::sleep_for(wait);
              hello_n(i); },
            /* 参数： */
            i
            )
          );
```

输出 [线程] hellonice:

```
Hello 0
Hello 1
Hello 2
你好 3
你好 4
```

Of course, in practice you don't synchronize threads through waits. Read on.

25.1.2 异步任务

线程的一个问题是如何返回数据。你可以通过引用捕获变量来解决这个问题，但这并不优雅。更好的解决方案是，如果你能问线程 " 你刚刚计算了什么 " 就好了。

为此我们有 `std::future`，来自头文件 `future`，使用返回类型进行模板化。因此，一个

```
std::future<int>
```

将在未来某个时刻是一个 `int`。你可以使用 `get` 来获取该值：

```
//async.cpp std::future<string> fut_str= std::async ( [] () ->
string { return "Hello world"; } ); auto result_str= fut_str.get();
cout << result_str <<'\n';
```

一个包含多个 future 的示例：

```
//async.cpp vector< std::future<string> >futures; for ( int ithread=0;
ithread<NTHREADS; ++ithread) { futures.push_back ( std::async ( [
ithread]() ->string { stringstream ss; ss << "Hello world "<< ithread;
return ss.str(); } ) );
```

```
    }for ( int ithread=0; ithread<NTHREADS; ++
ithread) {cout <<futures.at(ithread).get() << '\n';}
```

async 的问题在于任务不必在新的线程上执行：运行时可以在 get 调用时决定在调用线程上执行它。要立即启动新线程，请使用

auto fut= std::::async(std::::launch::async, fn, arg1, arg2);

25.1.3 返回结果：futuresandpromises

Explicit use of promises and futures is on a lower level than async.

//promise.cpp auto promise = std::promise<std::string>(); auto producer = std::thread ([&promise] {promise.set_value("Hello World"); }); auto future = promise.get_future(); auto consumer = std::thread ([&future] { std::cout <<future.get() << '\n'; }); producer.join(); consumer.join();

```
// /promise.cpp
    vector< std::thread >
    producers,consumers;
    vector< std::promise<string> >
    promises;

    for ( int i=0; i<4; ++i ) {

      promises.push_back(
    std::promise<string>() );
      producers.push_back
        ( std::thread
          ( [ i,&promises ] {
            stringstream ss;
            ss << "Hello world " <<
    i << ".";
promises.at(i).set_value(ss.str());
      } ) );

  consumers.push_back
    ( std::thread
      ( [ i,&promises ] {
        std::cout <<
promises.at(i).get_future().get()
<< '\n';
      } ) );
}

for ( auto& p : producers )
p.join();
for ( auto& c : consumers )
c.join();
```

25.1.4 当前线程

std::this_thread::get_id();

这是一个唯一的 ID，但你无法从中推导出任何其他信息。例如，它与操作系统进程 ID 或你的处理器核心数无关。

还有一个用于线程的 sleep_for 函数：

```
// /hello.cpp threads.push_back ( std::thread ( /*
function: */[ ] (int i ) { std::chrono::seconds wait(i);
std::this_thread::sleep_for(wait); hello_n(i); }, /
*argument: */ i ) );
```

25.1.5 More threadstuff

C++20 中的 jthread 启动一个线程，当其析构函数被调用时会加入。创建循环：

```
{ // 开始一个作用域 vector<thread> mythreads;for ( i=/* 内容
*/ )mythreads.push_back(thread( somefunc,someargs ) );}
// 结束作用域
```

现在，当向量超出作用域时，连接操作不再需要显式调用。这也解决了显式 `join` 调用顺序的问题。

25.2 数据竞争

并发中的一个重要主题是**数据竞争**：多个对单个数据项的访问在时间上或因果关系上未排序的现象，例如因为访问来自同时处于活动状态的线程。

```
std::mutex
alock;alock.lock();/*
临界区
*/alock.unlock();
```

这带来了一系列问题，例如如果临界区可以抛出异常。

一种解决方案是 `std::lock_guard`：

```
std::mutex alock;thread( [] () {std::lock_
guard<std::mutex> myguard(alock);/* 临界操作 */} );
```

锁卫在创建时会锁定互斥量，并在超出作用域时解锁它。

对于 C++17,`std::scoped_` 锁可以用多个互斥量实现这一点。

原子变量存在于 `原子` 头文件中：

```
#include <<atomic>><std>::<atomic><
<int>> <shared>_
<int>{v20};<shared>_<int>++;
```

线程间通信：

```
std::condition_ 变量 somecondition; //
thread 1:std::mutex alock;std::unique_
lock<std::mutex>
ulock(alock);somecondition.wait(ulock)//
thread 2:somecondition.notify_one();
```

相似但不同：

```
#include <<future>><std>{v7}::<future><<int>> <future>_
<computation>_=<std>{v23}::<async>{v27}(< []
>(
<int>x</int>{v36}){v38}){v40}<f>{v42}(<{v46}x{v48}>){v44};
100 );<future>_<computation>{v55}.</get>{v57}();

std::future_ 状态计算 _
    状态；计算 _ 状态 = 未来 _ 计算。等待 _ 对于（ /* chrono 时长 */）；如果（计算
_ 状态 ==std:: 未来 _ 状态 :: 就绪 )/* 计算已完成 */
```

25.3 同步

线程与创建它们的（或线程）环境并发运行。这意味着在同步之前，没有动作的时间顺序 `std::thread::join`。

在这个例子中，计数器的并发更新形成了一个 数据竞争：

代码：

```
1  // /race.cpp
2    auto start_time = myclock::now();
3    auto deadline = myclock::now() +
       std::chrono::seconds(1);
4    int counter{0};
5    auto add_thread =
6      std::thread( [&counter,deadline]() {
7        while (myclock::now()<deadline)
8          printf("Thread:
       %d\n",++counter);
9      }
10       );
11   while (myclock::now()<deadline)
12     printf("Main: %d\n",++counter);
13   add_thread.join();
14   cout << "Final value: " << counter <<
       '\n';
```

Output

[thread] race:

```
Three runs of <<race>>;
printing first lines only:
----
Main: 1
Thread: 51
Final value: 526851
Runtime: 1.00048 sec
----
Main: 1
Thread: 47
Final value: 617669
Runtime: 1.00243 sec
----
Main: 1
Thread: 47
Final value: 509073
Runtime: 1.00215 sec
```

形式上，此程序具有未定义行为 (UB)，您可以看到这一点反映在不同的最终值中。

最终值可以通过将计数器声明为 `std::atomic`：

Code:

```
// /atomic.cpp
auto start_time = myclock::now();
auto deadline = myclock::now() +
  std::chrono::seconds(1);
std::atomic<int> counter{0};
auto add_thread =
  std::thread( [&counter,deadline]() {
    while (myclock::now()<deadline)
      printf("Thread:
    %d\n",++counter);
  }
    );
while (myclock::now()<deadline)
  printf("Main: %d\n",++counter);
add_thread.join();
cout << "Final value: " << counter <<
  '\n';
```

Output
[thread] atomic:

```
Three runs of <<atomic>>;
printing first lines only:
----
Main: 1
Thread: 54
Final value: 495120
Runtime: 1.00282 sec
----
Thread: 1
Main: 353
Final value: 474618
Runtime: 1.00312 sec
----
Main: 1
Thread: 59
Final value: 339453
Runtime: 1.00212 sec
```

请注意，主线程和线程的访问仍然不可预测，但这是一种特性，而不是错误，绝对不是 UB。

25. 并发

第 26 章

晦涩的内容

26.1 自动

26.1.1 声明

有时变量的类型很明显：

```
std::vector<  std::shared_ptr_ myclass>>* myvar_  new std::vector<
std::shared_ptr_ myclass>>( 20, new myclass{v52}(1.3) );
```

(指向包含 20 个指向 myclass 的共享指针向量的指针，已使用唯一实例初始化。) 您可以将其写成：

```
auto myvar _new std::vector<  std::shared_ptr_
myclass >>(20, new myclass(1.3));
```

在 C++17 中，返回类型可以推断：

```
// /autofun.cppauto e
qual(int i, int j) {   return i ==
j;};
```

在 C++17 中，方法的返回类型可以推断：

```
 1 // /plainget.cpp 2 类 A { 3 private: float data; 4
public: 5 A(float i) : data(i) {}; 6 auto &access() { 7
return data; }; 8 void print() { 9 cout <<  "data: "<<
data <<  ’ \n’ ; }; 10 };
```

`auto` discards references and such:

代码：

```
1 // /pgetn et.c p
2   A my_a(5.7);t
3   au o ge a a = my_a.acess();
4   获取 _ 数据 += 1;
5   我的 _a. 打印 ();
```

输出
[自动] 平面获取：

数据 : 5.7

结合自动和引用：

代码：

```
1// /refget.cpp2
   A my_a(5.7);
3  自动 & 获取 _ 数据 = my_a.access();
4  获取 _ 数据 += 1;
5  我的 _a。print();
```

输出
[auto] refget：

data: 6.7

For good measure:

```
1 ///constrefget.cpp2    Amy_a(5.7); 3 const
auto &get_data =my_a.access();4 get_data += 1;
// WRONG does notcompile5    my_a。print () ;
```

26.1.2 Autoand functiondefinitions

函数的返回类型可以通过尾随返回类型 定义来指定：

```
auto f(int i) -> double { /* stuff */ };
```

这种表示法对于 lambda 更常见，章节 13。

26.1.3 decltype: declaredtype

在某些情况下，你希望编译器推导变量的类型，但立即无法做到。假设在

```
auto v = some_object.get_stuff();
f(v);
```

你希望在 `try ... catch` 块中仅围绕 `v` 的创建。这行不通：

```
try { auto v =some_ 对象 .get_ 东西 ();}
catch(...) {}f(v);
```

因为 `try` 块是一个作用域。它也不起作用 t o write

```
auto v;
try { v = some_object.get_stuff();
} catch (...) {}
f(v);
```

因为没有指示 v 被创建的类型。

相反，可以使用 decltype 查询创建 v 的表达式的类型。 :

```
decltype(一些_对象.get_内容()) v;try {
auto v =一些_对象.get_内容();}
catch(...) {}f(v);
```

10.9.4

26.2 转换

在 C++ 中，常量和变量具有明确的类型。对于需要强制类型为其他情况的情况，有强制类型转换 机制。通过强制类型转换，你告诉编译器：将这个事物视为某种类型，无论它如何定义。

在 C 中，只有一个强制类型转换机制：

```
sometype x; othertype y=
othertype x;
```

该机制仍然可用作 **reinterpret_强制类型转换**，它执行 ‘将这个字节视为以下类型’：

```
sometype x;auto y =reinterpret_强制类型
转换<othertype>(x);
```

继承机制需要另一种强制类型转换机制。派生类的对象包含所有基类信息。很容易获取派生类指针，更大的对象，并将其转换为基类指针。另一种方法更困难。

考虑：

```
类 基类{}; 类 派生类: public 基类
{}; 基类 * 对象 =new 派生类;
```

现在我们可以将对象转换为指向派生类的指针吗？

- **静态_转换** 假设你知道自己在做什么，并且它无论如何都会移动指针。
- **动态_转换** 检查 对象 实际上是否属于类 派生类，然后移动指针，否则返回 `nullptr`。

备注 30 *C* 风格转换的一个进一步问题是它们的语法很难被发现，例如通过在编辑器中搜索。因为 *C++* 转换有一个独特的关键字，所以在文本编辑器中更容易识别。

进一步阅读

https://www.quora.com/How-do-you-explain-the-differences-among-staticcast-reinterpret_const_cast_cast 和 dynamic_cast_cast 到新的 C 类型 ++-programmer/answer/Brian-Bi

26.2.1 静态类型转换

一种类型转换的用途是将常量转换为 ‘更大’ 的类型。例如，分配不使用整数，而 `size_t`。

```
// /longint.cpp
int hundredk = 100000; int overflow; overflow= hundredk*hundredk;
cout <<"overflow: " << overflow << '\n'; size_t bignumber =static_cast<
size_t>(hundredk)*hundredk; cout <<"bignumber: " << bignumber << '\n';
```

然而，如果转换是可能的，结果可能仍然不是 ‘正确的’ 。

代码：

```
1 // /intlong.cpp
2   long int = 100000000000;
3   cout <<"long number:"          "
4         << hundredg << '\n';
5   int overflow;
6   溢出 = 静态_转换<整数>(hundredg);t <<" i
7   cou
8         <<溢出<<"\n";
```

输出

```
[static_cast] intlong:
```

静态类型转换没有运行时测试。

静态类型转换是使 void 指针恢复其原始类型的一种好方法。

26.2.2 动态转换

考虑有一个基类和派生类的情况。

```
// /toderived.cpp 类 Base {
public: virtual voidprint() = 0; }; 类
Derived : public Base { public:
virtual void print() { cout<< "
Construct derived!"<< '\n'; }; }; 类
Erived : public Base { public: virtual
void print() { cout<< "Construct
erived!"<< '\n'; }; };
```

假设我们有一个接受基类指针的函数：

```
// /toderived.cppvoid
f(Base* obj ) {
```

```
    派生 * 派生 = 动态 _ 转换 < 派生 *>(obj);if
(der==nullptr)cout<< "无法转换为派生类型 "<< '
\n';elseder->print();};
```

该函数可以识别基类指针指向的派生类：

```
// /toderived.cpp基类 * 对象 =new
派生 ();f( 对象 ); 基类 *nobject =
new 派生 ();f(nobject);
```

如果我们有一个指向派生对象的指针，存储在一个指向基类对象的指针中，可以安全地将其转换回派生指针：

代码：

```
1 //  /toderived.cpp
2     Base * 对象      = new 派生 ();
3     f( 对象 );
4     Base * 对象       = new Erived();
5     f( 对象 );
```

输出
[动态转换] 正确：

构造派生！
无法转换为派生

另一方面，一个静态 _ 转换 不会起作用：

代码：

```
1 // /toderived.cpp
2 void g( Base *obj ) {
3   派生 Derived*der
4   =tti a  d_cas <Derived*>(obj);
5   der- 打印 ();
6 };
7     /* ... */
8     Base * 对象      = new Derived();
9     ( 对象 );
10    基本 * 对象       = new 派生 ();
11    g( 对象 );
```

输出
[动态转换] 派生错误：

构造派生！
构建 *derived*

注意：基类需要是多态的，这意味着需要纯虚方法。静态转换则不是这种情况，但正如所说，这种情况下的转换不正确。

26.2.3 Const cast

使用 const_cast，你可以给变量添加或移除 const。这是唯一可以实现这种操作的转换。

26.2.4 转义赋值

reinterpret_ cast 是最粗鲁的转换，它对应 C 语言中的 “ 取这些字节并假装它是任何类型 ” 的机制。这种转换有一个合法用途：

```
void *ptr;ptr = malloc(how_much );auto 地址 =
reinterpret_cast<long int>(ptr);
```

这样你就可以对地址进行算术运算。对于这个特定的例子，`intptr_t` 实际上更好。

26.2.5 关于 void 指针的说明

C 中 casts 的传统用途之一是处理 *void* 指针 s。在 C++ 中，这种需求不像以前那么严重。

void 指针的一个典型用途出现在 PETSc [3, 4] 库中。通常当你调用库函数时，你无法再访问该函数内部的发生了什么。然而，PETSc 提供了让你指定监控器的功能，这样你就可以打印出内部量。

```
intKSPSetMonitor(KSP ksp,int
(*monitor)(KSP,int,PetscReal,void*),void
*context,// 一个参数被省略了 );
```

在这里你可以声明自己的监控例程，它将被内部调用：库会向你的代码进行回调。由于库无法预测你的监控例程是否需要进一步的信息才能正常工作，因此有 context 参数，你可以将其作为 void 指针传递一个结构。

在 C 中不再需要这种机制，你将使用 *lambda* (章节 13):

```
KSPSetMonitor( ksp,[mycontext] (KSP k,int ,
PetscReal r) -> int{my_monitor_ 函数
(k,r,mycontext); } );
```

26.3 标量类型的细节

26.3.1 整数

存在几种整数类型，它们的不同之处在于所占的字节数，以及是否是有符号的。这里进行系统的讨论。

26.3.1.1 要点

除了 `int` 之外，还有 `short`，`long` 和 `long long` 整数。这些名称在一定程度上给出了它们的大小，因此是范围。这些类型的实际大小和范围是实现的定义。下一节将讨论具有更精确定义的类型。

- 一个短整数至少是 16 位;
- 一个整数至少是 16 位，这在 *DEC PDP-11* 的旧时代是这种情况，但现在通常是 32 位;
- 一个长整数至少是 32 位，但通常是 64 ；唯一的例外是 *Windows* 操作系统，其中 `long` 整数是 32 位的。

- A 长长整数至少有 64 位。
- 如果你只需要一个字节来存储整数，你可以使用 **char**；参见第 11.1 节。

这些类型有若干种广泛接受的 数据模型 用于定义这些类型；参见 HPC 书 [11]，第 3.7.1 节。

所有这些类型都是有符号整数：一个 k 位的整数可以表示一个范围 $-2^{k-1}\ldots 0\ldots 2^{k-1}-1$。在这些类型前加上关键字 **unsigned** 会得到非负类型，其范围是 $0\ldots 2N-1$。

如果你想要精确地确定存储在 **int** 变量中的整数（或稍后讨论的实数）的范围，你可以使用 *limits* 头文件；参见第 24.2 节。

26.3.1.2 Byte by byte

如果你想要指定使用多少位，可以使用包含 *cstdint* 的固定宽度整数类型头文件。它定义了 *int16_t* 和 *uint16_t* 等类型：

int*8*_t // 8 bitsuint*8*_t // 8 bits,
unsignedint*16*_t // 16 bitsuint*16*_t //
16 bits, unsignedint*32*_t // 32
bitsuint*32*_t // 32 bits,
unsignedint*64*_t // 64 bitsuint*64*_t //
64 bits, unsigned

以下示例通过使用 **sizeof** 函数来确认这些大小：

代码：

```
// /sizeof.cpp
  int v1t i8;
  uint8_t u8;
  int v1 t i16;
  uint16_t u16;
  cout << sizeof(i8)
       << ",  " << sizeof<u8>
       << ", " << sizeof<i16>
       << “”, << sizeof(u16)
       << '\n';
```

26.3.1.1 简而言之
[int] sizeof:

1, 1, 2, 2

自 C 以来，++20 整数保证以 “ 二进制补码 ” 形式存储；参见 HPC 书籍 [11]，第 {v7}3.2{v9} 节。{v11}

无符号值充满危险。例如，将它们与整数比较会产生反直觉的结果：

代码：

```
1// /unsigned.cpp2 unsigned intone{1};3
intmone{-1};4 cout <<"less: " <<
boolalpha <<(mone<one{v37}) <<’\n’;
```

输出 [int]
cmp:

less: false

为此，C++20 引入了实用函数 *cmp_equal* 等（在 *utility* 头文件中），这些函数可以正确执行这些比较。

26.3.1.3 为什么整数不同？

历史上，短整数是由节省空间的需求驱动的。如今，这一论点已 largely irrelevant。短整数的论点现在源于处理器的有限带宽：“流式”类型应用程序的性能取决于可用带宽，而使用短整数 effectivelydoublesthat bandwidth。

长整数也是由当前内存和 disc 空间的丰富性驱动的。现代应用程序有大量的内存可供使用，这些内存超过了 32 位整数可以寻址的范围。这种现象被 多核 处理器的存在所加剧，这些处理器可以在相同的时间内处理比早期处理器多很多倍的内存。

26.3.2 浮点类型

截断和精度是棘手的事情。作为一个简单的示例，让我们在单精度和双精度下进行相同的计算。虽然结果在默认的 cout 格式下显示相同，但如果我们相减，我们会看到非零的差异。

Code:

```
// /point3.cpp
  double point3d = .3/.7;
  float
    point3f = .3f/.7f,
    point3t = point3d;
  cout << "double precision: "
       << point3d << '\n'
       << "single precision: "
       << point3f << '\n'
       << "difference with truncation:"
       << point3t - point3f
       << '\n';
```

Output

```
[basic] point3:
double precision: 0.428571
single precision: 0.428571
difference with
    truncation:-2.98023e-08
```

实际上，您可以解释这个差异的大小，但是我们将浮点运算的细节推迟到 HPC 书籍 [11]，章节 3。

26.3.3 非数字

IEEE 754 标准对于浮点数规定，某些位模式对应于值 *NaN*：‘不是数字’。这是诸如负数的平方根或零除以零等计算的结果；您也可以使用安静 *_NaN* 或 信号 *_NaN* 显式生成它。

- NaN 仅定义于浮点类型：测试 *has_* 安静 *_NaN* 对于其他类型（如 **bool** 或 **int**）是假。
- Even through *complex* is built on top of floating point types, there is no NaN for it.

Code:

```
1 // /nan.cpp
2   cout << "Double NaNs: "
3        <<
      std::numeric_limits<double>::quiet_NaN()
          << ' ' // nan
4        <<
      std::numeric_limits<double>::signaling_NaN()
      << ' ' // nan
5        << '\n'
6        << "zero divided by zero: "
7        << 0 / 0.0          << '\n';
8   cout << boolalpha
9        << "Int has NaN: "
10       <<
      std::numeric_limits<int>::has_quiet_NaN
11       << '\n';
```

Output
[limits] nan:

```
Double NaNs: nan nan
zero divided by zero: nan
Int has NaN: false
```

26.3.3.1 测试

有测试用于检测一个数字是否 *Inf* 或 *NaN*。但是，使用这些可能会减慢计算速度。

对于浮点类型（float、 double、 long double），定义了 *isinf* 和 *isnan* 函数，返回一个 bool 值。

```
#include <math.h>
isnan{v9}(-1.0/0.0); //
falseisnan{v13}(-1.0); //
trueisinf{v25}(-1.0/0.0); //
trueisinf{v31}(-1.0); // false
```

26.3.4 常见数字

```
#include <numbers>static constexpr float pi=
std::numbers::pi;
```

26.4 左值 vs 右值

术语 ‘左值’ 和 ‘右值’ 有时会出现在编译器错误消息中。

```
int foo() {return 2;}int
main() {foo() = 2;
return 0;}
```

```
# 给出：测试。c：在函数'main'：测试。c:8:5：错误：需要左值作为左操作数
的赋值
```

看到 ‘左值’ 和 ‘左操作数’ 了吗？从第一级近似来看，你可能会认为一个 左值 是赋值左侧的某个东西。这个名字实际上意味着 ‘定位值’：与内存中特定位置相关联的东西。因此，左值也可以宽泛地说是一些可以被修改的东西。

一个 右值 是出现在赋值右侧的东西，但实际上定义为所有不是左值的东西。通常，右值不能被修改。

赋值 x=1 是合法的，因为变量 x 在内存中的某个特定位置，所以它可以被赋值。另一方面，x +1=1 是不合法的，因为 x+1 最多只是一个临时值，因此不在特定的内存位置，所以不是左值。

不太简单的例子：

```
int foo() { x = 1; return x; }int
main() { foo() = 2; }
```

不合法，因为 foo 没有返回一个左值。然而，

```
类 foo { private : int x ; public :
int & xfoo () { return x ; } ; } ; int
main () { foo x ; x . xfoo () = 2 ; }
```

合法，因为函数 xfoo 返回了对 foo 对象的非临时变量 x 的引用。

并非每个左值都可以赋值：在

```
const int a = 2;
```

变量 a 是一个左值，但不能出现在赋值语句的左侧。

26.4.1 转换

大多数左值可以快速转换为右值：

```
int a = 1;
int b = a+1;
```

在第一行中 a 作为左值，但在第二行中变成了右值。

取地址运算符接受一个左值并返回一个右值：

```
int i;int*a = &
i;&i = 5;// 错误
```

26.4.2 参考资料

The ampersand operator yields a reference. It needs to be assigned from an lvalue, so

```
std::string &s= std::string(); // 错误
```

这是非法的。s 的类型是一个 ‘左值引用’，不能从右值赋值。

另一方面

```
const std::string &s= std::string();
```

有效，因为 s 无法再被修改。

26.4.3 右值引用

C++ 的一项新特性旨在通过移动语义最小化数据复制量。

考虑一个拷贝赋值运算符

```
BigThing& 运算符=( const BigThing & 其他 )
{BigThing tmp(其他); // 标准拷贝 std::swap( /*
tmp 数据到我的数据 */);return *this;}
```

这会调用 tmp 的拷贝构造函数和析构函数。（临时对象的使用使得在异常情况下这是安全的。*swap* 方法从不抛出异常，因此不存在半拷贝内存的危险。）

然而，如果你赋值

```
thing = BigThing(stuff);
```

现在，右侧的临时右值对象会调用构造函数和析构函数。

Using a syntax that is new in *C++*, we create an *rvalue reference*:

```
BigThing& 运算符=( BigThing && 其他 )
{swap(/* 其他移动到我的位置 */);return
*this;}
```

26.5 移动语义

With an *overloaded operator*, such as addition, on matrices (or any other big object):

```
Matrix operator+(Matrix &a,Matrix &b);
```

实际加法将涉及复制：

```
Matrix c = a+b;
```

使用移动构造函数：

```
类矩阵 { 私有 : 表示 rep; 公
有 : 矩阵 ( 矩阵 &&a){rep
= a.rep;a.rep=
{};}};
```

26.6 图形

C++ 没有内置的图形功能，因此您必须使用外部库，例如 *OpenFrameworks*,https://openframeworks.cc.

26.7 标准时间线

每个标准在之前的版本中都有许多变化。

If you want to dete请说明您正在使用什么语言标准进行编译，使用 `__cpluscplus` macro:

Code:

```
1 // /version.cpp
2   cout << "C++ version: " <<
      __cplusplus << '\n';
```

Output

```
[basic] version:
C++ version: 201703
```

这返回一个**长整型**，可能的值为 199711 、 201103 、 201402 、 201703 、 202002。

这里是一些不同标准的亮点。

26.7.1 C++98/C++03

在 C++03 标准中，我们仅突出显示已弃用的特性 ures.

- 自动 _ 指针 是一个早期尝试智能指针的方案。它已被弃用，C++17 编译器实际上会在其上发出错误。对于当前的智能指针，请参阅第 16 章。

26.7.2 C++11

- auto

```
const auto count = std::count
    (begin(vec),end(vec),value);
```

The `count` 变量 now gets the type of whatever`vec` contained .
基于范围的 for 循环。我们将此视为基本情况，例如在 10.2 节中。C 语言之前的机制，使用迭代器（第 14.2.1 节），基本上被取消了。

- Lambda 表达式。参见第 13 章。
- Chrono ； 参见第 24.8 节。

- 可变参数模板。
- 智能指针；参见第 16 章。唯一 _ptr<int> iptr(new int(5)); 这解决了 auto_ptr 的问题。
- constexpr

```
constexpr int get_value() {
  return 5*3;
}
```

26.7.3 C++14

C++14 可以被视为对 C++11 的一个 bug 修复。它简化了许多事情，并使它们更加优雅。

- 自动返回类型推导：

```
autof() {SomeType
something;return
something;}
```

- 通用 lambda 表达式（节 13.3.2)const autocount =

```
std::count(begin(vec),end(vec),[] ( const auto i ) {
returni<3; });
```
还有更复杂的捕获表达式。

- constexprint get_value()

```
{intval = 5;intval2 = 3;return
val*val2}
```

26.7.4 C++17

- 可选；节 24.6.2.
- 结构化绑定声明作为一种更简单的方法来分解元组；节 24.5.
- 条件语句中的初始化语句；章节 5.5.3。

26.7.5 C++20

- 模块：这些比使用头文件提供了更好的接口规范。
- 协程，另一种并行形式。
- 概念 通过范围在标准库中包含；章节 22.4。
- *spaceship* 运算符 在标准库中包含
- 广泛使用普通 C++ 进行直接的编译时编程，而无需借助模板元编程（参见 lasttripreports ）

- 范围

- 日历 和时区
- 文本格式化
- span。参见第 10.9.6 节。
- 数字。第26.3.4 节。
- 安全整数 / 无符号比较；第 26.3.1.2 节；整数保证是二进制补码表示。

这里有一个包含示例的总结:
https://oleksandrkvl.github.io/2021/04/02/cpp-20-overview.html。

26.7.6 C++23

- mdspan 提供多维数组；这是一种对 C++17span 机制的扩展。

第 27 章

图形

C++ 语言和标准库没有图形组件。但是，以下项目存在。

https://www.sfml-dev.org/

27. 图形

第 28 章

C++for C 程序员

28.1 I/O

几乎没有用途 *printf* 和 *scanf*。使用 cout（和 *cerr*）以及 cin 。还有 *fmtlib* 库。

第 12 章。

28.2 数组

通过方括号表示法访问数组是不安全的。它们基本上是指针，这意味着它们除了内存位置之外不携带任何信息。

最好使用 *vector*。即使使用方括号表示法，也要使用基于范围的循环。

第 10 章。

向量独占其数据，因此如果有多个 C 风格指针指向相同的数据，就像有多个数组一样，这是不工作的。为此，使用 span ；章节 10.9.6。

28.2.1 从 Carrays 获取向量

假设你必须与一个使用 *malloc* 的 C 代码接口。向量有优势，比如它们知道自己的大小，你可能想将这些 C 风格数组包装在一个向量对象中。这可以使用一个范围构造函数来完成：

```
向量<双精度浮点数> x(指针_指向_第一个, 指针_在最后一个之后_最后一个);
```

这样的向量仍然可以动态使用，但这可能会导致内存泄漏和其他可能不希望的行为：

Code:

```
// /cvector.cpp
    float *x;
    x =
    (float*)malloc(length*sizeof(float));
    /* ... */
    vector<float> xvector(x,x+length);
    cout << "xvector has size: " <<
    xvector.size() << '\n';
    xvector.push_back(5);
    cout
      << "Push back was successful" <<
    '\n';
    cout << "pushed element: "
         << xvector.at(length) << '\n';
    cout << "original array: "
         << x[length] << '\n';
```

Output
[array] cvector:

```
xvector has size: 53
Push back was successful
pushed element: 5
original array: 0
```

28.3 Dynamic storage

向量和其它容器的另一个优势是 *RAII* 机制，这意味着当离开作用域时动态存储会自动被释放。参见 10.9.4。（对于跨作用域的安全动态存储，参见下文讨论的 smartpointers。）

RAII 代表 ‘资源分配即初始化’。这意味着不再可能编写

double *x; if (something1) x =
malloc(10); if (something2) x[0];

这可能会导致内存错误。相反，名称的声明和存储的分配是一个不可分割的操作。

另一方面：

```
vector<double> x(10);
```

dec定义变量 x，分配动态存储，并在 itializes it.

28.4 字符串

A *C* 字符串 是一个带 空终止符的字符数组。另一方面，一个字符串 是一个定义了其操作的 对象。

第 11 章

28.5 指针

许多 *C* 指针的用途，它们实际上是地址，已经消失了。

- 字符串通过 std::string 处理，而不是字符数组；见上文。
- 数组主要可以通过 std::vector 处理，而不是 `malloc`；见上文。
- 遍历数组和向量可以用 ranges 实现；见第 10.2 节。
- 通过引用传递参数，使用引用。第 7.5 节。
- 任何遵循作用域的内容都应该通过一个 构造函数创建，而不是使用 *malloc* .

指针有一些合法的需求，例如堆上的对象。在这种情况下，使用 共享 `_ptr` 或 唯一 `_ptr`；第 16.1 节。C 指针现在称为 裸指针 s，它们仍然可用于 “ 非拥有 ” 指针的用法。

28.5.1 参数传递

不再通过地址：现在真正的引用！第 7.5 .

28.5.2 地址

C 程序员习惯使用与号来获取变量的地址，以及在 C++：对象。C++ 程序员需要注意，与号可能会被重新定义：

```
T operator&(){ /* stuff */ };
```

如果你绝对需要一个地址，可以使用 `addressof` 函数。

28.6 对象

对象是附加了函数的结构。第 9.

28.7 命名空间

不再有加载两个包时的命名冲突：每个都可以有自己的命名空间。第 20.

28.8 模板

如果你发现自己为多种类型编写相同的函数，你会爱上模板。第 22 章。

28.9 难以理解的内容

28.9.1 Lambda

函数表达式。第 13 章。

28.9.2 Const

函数和参数可以被声明为 const。这有助于编译器。第 18.1 节。

28.9.3 左值和右值

第 26.4 节。

第 29 章

C++ 复习题

29.1 算术

1. 给定 int n;

   编写使用基本数学运算符来计算 n-cubed: n^3。对于所有 n，你是否得到正确的结果？解释。

2. 的输出是什么：int m=32, n=17;

cout<<n%m <<endl;

29.2 循环

1. 假设有一个函数 bool
f(int);

   给定一个函数，它在某些正输入值时为真。编写一个主程序，找到使 f 为真的最小正输入值。

2. 假设有一个函数

```
bool f(int);
```

   给定一个函数，它在某些负输入值时为真。编写一个主程序，找到绝对值最小的（负）输入值，使 f 为真。

29.3 函数

练习 29.1。 以下代码片段在一个循环中计算递归

$$v_{i+1} = av_i + b, \qquad v_0 \text{ given.}$$

编写一个递归函数

```
float v = value_n(n,a,b,v0);
```

该计算 v_n 的值 $n \geq 0$。

29.4 向量

练习 29.2。 以下程序存在多个语法和逻辑错误。其目的是读取一个整数 N，并将整数 1，...，N 排序到两个向量中，一个用于奇数，一个用于偶数。奇数应乘以二。

你的任务是调试这个程序。为了获得 10 分的学分，找出 10 个错误并改正它们。发现的额外错误将计入加分。对于逻辑错误，即语法正确但仍然“做错事”的地方，用几句话说明程序逻辑的问题。

```
#include <iostream>using std::cout; using std:cin;
using std::vector; int main() { vector<int>
evens,odd; cout << " 输入一个整数值 "<< endl; cin <<
N; for (i=0; i<N; i++) { if (i%2=0) { odds.push_
back(i); else evens.push_back(i); } for ( auto o:
odds ) o /= 2return 1 }
```

29.5 向量

练习 29.3. 再看看练习 29.1。现在假设你想保存这些值 v_i 到一个数组 向量 <float> 值。编写代码来完成这个任务，先使用迭代计算，然后使用递归计算。你更喜欢哪种方式?

29.6 对象

练习 29.4. 给定一个类 *Point* 类。你会如何设计一个类 *SetOfPoints* (它模拟了一组点) 以便你可以写出

```
Point p1,p2,p3;SetOfPoints
pointset;// 将点添加到集合中:

pointset.add(p1);
pointset.add(p2);
```

给出该类的相关数据成员和方法。

练习 29.5. 你正在编写一个视频游戏。有移动元素，你想为每个元素创建一个对象。移动元素需要一个名为 move 的方法，该方法有一个表示时间持续时间的参数，并且此方法会使用该对象的速度和持续时间来更新元素的位置。

补充代码的缺失部分。 class position { /* ... */ public: position() {}; position(int initial) { /* ... */ }; void move(int distance) { /* ... */ }; }; class actor { protected: int speed; position current; public: actor() { current =position(0); }; void move(int duration) { /* THIS IS THE EXERCISE: */ /* 写出此函数的体 */ }; }; class human : public actor { public: human() // EXERCISE: 写出构造函数 }; class airplane : public actor { public: airplane() // EXERCISE: 写出构造函数 }; int main() { human Alice; airplane Seven47; Alice.move(5); Seven47.move(5);

29. C++ 复习问题

PART III

FORTRAN

第 30 章

Fortran 基础

Fortran 是一种古老的编程语言，可追溯到 20 世纪 50 年代，是第一个被广泛使用的 “ 高级编程语言 ”。在某种程度上，编程语言设计和编译器编写领域是从 Fortran 开始的，而不是 Fortran 基于这些已确立的领域。因此，Fortran 的设计有一些后来设计的语言没有采用的怪癖。其中许多现在已被 “ 弃用 ” 或仅仅不建议使用。幸运的是，Fortran 可以以一种与其他当前语言一样现代和复杂的方式编写。

在本书的这一部分，你将学习编写 Fortran 的安全实践。我们偶尔不会提及你在旧 Fortran 代码中会遇到但不会建议你采用的实践。虽然 Fortran 的这种阐述可以独立存在，但我们在某些地方会明确指出与 C++ 的区别。

30.1 源格式

Fortran 诞生于程序存储在 穿孔卡片 的时代。这些卡片有 80 列，因此 Fortran 源代码的一行不能超过 80 个字符。此外，前 6 个字符有特殊含义。这被称为 固定格式。然而，从 *Fortran 90* 开始，可以拥有 自由格式，这允许更长的行，而初始列没有特殊含义。

这两种格式之间存在进一步差异（尤其是续行），但我们在本课程中只讨论自由格式。

许多编译器通过文件名扩展名约定来指示源格式：

- f 和 F 是旧式固定格式的扩展名；和
- f90 和 F90 是新式自由格式的扩展名。

大写字母表示对文件应用了 *C* 预处理器 。在本课程中，我们将使用 `F90` 扩展名。

30.2 编译 Fortran

最小的 Fortran 程序是：

```
//emptyprog.F90Program
SomeProgram! stuff goes
hereEnd
ProgramSomeProgram
```

你会编译这个：

```
yourfortrancompiler -o myprogram myprogram.F90
```

and then execute with

```
./myprogram
```

对于 Fortran 程序，编译器是 `gfortran` 用于 GNU 编译器，以及 `ifort` 用于 Intel。

练习 30.1. 添加该行

```
print*, "Hello world!"
```

在空程序中添加该行，然后编译并运行它。

Fortran 忽略 大小写 在关键字和标识符中。因此，上述程序中的 `Program` 等关键字也可以写成 PrOgRaM。

一个程序可以选择性地有一个停止语句，它可以将消息返回给操作系统 .

代码：

```
1// /stop.
2 程序 SomeProgram
3
4 结束程序 SomeProgram
```

Output
[基本 f] 停止：

```
STOP 代码在此停止
```

此外，stop 返回的数值代码 stop

```
stop 1
```

可以使用 $? shell 参数查询：

代码：

```
1 // /stopreturn.F90
2 Program SomeProgram
3     停止 17
4 结束程序        SomeProgram
```

输出
[basicf] 返回：

```
./停止返回 ||代码 =$? \
          &&echo 返回代码
     是 $code
STOP 17
返回码是 17
```

30.3 主程序

Fortran 不使用大括号来界定代码块，相反你会找到 结束语句。第一个语句出现在你开始编写程序时：一个 Fortran 程序需要以一个 `Program` 行开始，并以 `End Program` 结束。这两行都需要包含程序名：

```
// /emptyprog.F90Program
SomeProgram! stuff goes
hereEnd
ProgramSomeProgram
```

and you can not use that name for any entities in the program.

Remark31 *The* emacs 编辑器将提供块类型和名称，如果你提供 *'end'* 并按下 *TAB* 或 *RETURN* 键；参见第 *2.1.1* 节。

30.3.1 程序结构

与 C++ 不同，Fortran 不能混合变量声明和可执行语句，因此主程序和任何子程序都大致具有以下结构：

```
Program foo<声明 ><
语句 >End
Programfoo
```

还需要注意的是，Fortran 没有像 C++ 那样的 “ 标准库 ” 需要显式包含。或者你也可以说，Fortran 的标准库默认总是包含的。

30.3.2 语句

我们来谈谈布局。Fortran 有一个 “ 一行一条语句 ” 的原则，这源于它的穿孔卡片 时代。

- 只要一条语句能在一行内放下，你就不必用分号之类的符号显式结束它：

  ```
  x = 1
  y = 2
  ```

- 如果你想在一条行上放两条语句，你必须结束第一条语句：

  ```
  x = 1;y = 2
  ```
  但要注意行长度：这通常限制为 132 个字符。

- 如果一个语句跨越多行，除了第一行之外的所有行都需要有一个明确的 续行字符，即与号：

  ```
  x= 非常 & 长
  & 表达式
  ```

30.3.3 注释

Fortran knows only single-line *comments*, indicated by an exclamation point:

```
x = 1 ! set x to one
```

感叹号到行尾的内容会被忽略 .

可能不太明显：你可以在续行符后面添加注释：

```
x = f(a) & ! term1
  + g(b)   ! term2
```

备注 32 在 *Fortran77* 中，允许使用 *19* 行续行符。在 *Fortran95* 中，这个数字增加到 *40*。自 *Fortran2003* 标准以来，一行可以续行 *256* 次。

30.4 变量

与 C++ 不同，在 C++ 中，您可以在需要变量之前立即声明它，而 Fortran 希望其变量在程序的顶部附近声明：

```
Program
YourProgramimplicit
none! 变量声明 ! 可执行代码

End ProgramYourProgram
```

The `implicit none` should always be included; see section 30.4.1.1 for an explanation.

变量声明看起来像：

```
type [ , attributes ] :: name1 [ , name2, .... ]
```

where

- 我们使用常见的语法缩写，［ something ］表示一个可选的 ‘something’ ；
- 类型 最常用的是**整数**， **实数** (4)， **实数** (8)， **逻辑型**。见下文；第 30.4.1 节。
- 可选 属性 是诸如 **维度**、**可分配**、**意图**、**参数** 等等。
- 名称 是你自己想出来的。它必须以字母开头。不寻常的是，变量名是不区分大小写的。因此，`Integer ::MYVARMyVar = 2print *,myvar`

  这是完全合法的。

备注 33 在 *Fortran66* 中，变量名长度最多为六个字符，尽管许多编译器对此有扩展。自 *Fortran2003* 标准以来，变量名可以长达 *63* 个字符。

Fortran 的内置数据类型：

- 数值：**整数**，**实数**，**复数**
- 精度控制：

```
整数 ::i 整数
(4):: i4 整数
(8):: i8
```

  这通常对应于字节数；请参考教科书了解完整内容。
- 逻辑：**逻辑**。
- 字符：**字符**。字符串被实现为字符数组。
- 派生类型（如 C++ 结构或类）：**类型**

某些变量永远不打算改变，例如如果你引入一个值为 3.14159 的变量 *pi*，你可以使用 **参数** 关键字将此名称标记为值的同义词，而不是可以分配值的变量。

```
real, 参数 :: pi = 3.141592
```

在章节 39 中，你会发现**参数** 常用于定义数组的大小。

关于数值精度的进一步说明在章节 30.4.1.2 中讨论。字符串在章节 35 中讨论。

30.4.1 声明

30.4.1.1 隐式声明

Fortran 对变量类型有一种不太寻常的处理方式：如果你不指定变量的数据类型，Fortran 会根据名称的第一个字符按简单规则进行推断。这是一种非常危险的做法，因此我们提倡在任何程序或子程序头之后立即添加一行

```
implicit none
```

声明。现在每个变量都需要在声明中明确指定类型。

30.4.1.2 变量 *'kind's*

Fortran 提供了多种机制来指定数值类型的精度。

```
// / 存储。.
F90integer(2) ::
i2integer(4) ::
i4integer(8) ::i8real(4) ::
r4real(8) ::r8real(16) ::
r16complex(8) ::
c8complex(16) ::
c16complex*32 ::c32
```

这通常对应于使用的字节数，但并不总是如此。从技术上讲，它是一个数值型选择器，仅仅是一个特定类型的标识符。

30.4.2 初始化

变量可以在声明时初始化：

```
整数 :: i=2 实数
(4):: x  = 1.5
```

这是在编译时完成的，导致一个常见错误：

```
subroutine
implicit
noneinteger:: i=2
print *,ii = 3end
subroutinefoo
```

在第一次子程序调用时 i 会打印其初始化值，但在第二次调用时不会重复初始化，而是保留之前的值 3。

30.5 复数

复数是一个实数的对。复常数可以用括号表示法书写，但要从两个实数变量形成复数需要使用 **CMPLX** 函数。你也可以使用这个函数将实数强制转换为复数，以便后续计算在复数域中进行。

Real and i虚部可以用 **real** 函数提取，和 **aimag**.

复常数作为一对实数写在括号中。有一些基本操作。

代码：

```
1// /roots.
复数 :: &f
tfi   our y  vedegrees = (1.,1.), &
      数字，旋转
实 :: x, y
print *,"45 = 3 de rees: ",number= fourt fivede rees
cmplx(x,y)rotated= .number * fourtyfivedegrees

打印  '  (" 旋转  数字具有实部 =",f5.2,"
   Im=,f5.2)' ,&
      实部 ( 旋转 ), 虚部 ( 旋转 )
```

输出
[基本] 复数：

```
45 度:
(1.00000000,1.00000000)
旋转后的数字有 Re= 2.00Im=
    4.00
```

虚根 i 没有预定义。使用

```
Complex,parameter :: i = (0,1)
```

In Fortran2008, **复数** 是一个派生类型，并且实部 / 虚部 s can be extracted as

```
print *, rotated %, re, , rotated %, im
```

Code:

```
1 // /complexf08.F90
2   print *,"45
      degrees:",fourtyfivedegrees
3   x = 3. ; y = 1.; number = cmplx(x,y)
4   rotated = number * fourtyfivedegrees
5   print '("Rotated number has
      Re=",f5.2," Im=",f5.2)',&
6       rotated%re,rotated%im
```

Output
[basicf] complexf08:

```
45 degrees:
    (1.00000000,1.00000000)
Rotated number has Re= 2.00
    Im= 4.00
```

练习 30.2. 编写一个程序来计算二次方程的复数根

$$ax^2 + bx + c = 0$$

变量 `a,b,c` 必须是实数，但使用 **cmplx** 函数来强制在复数域中计算根。

30.6 Expressions

Fortran 有算术、逻辑和字符串表达式。

- 算术表达式看起来或多或少是你期望的样子。唯一的特殊运算符是幂运算符：`x**.5` 是变量 `x` 的平方根。
- 对于布尔表达式，有常量 。**true**. 和 。**false**.；运算符同样用点括起来：。**and**. and such。布尔变量是类型 **Logical**。参见章节 30.19 以获取更多信息。
- 字符串处理将在第 35 章中讨论。

30.7 位操作

自 Fortran95 起，提供了用于位操作的函数：

- **btest**(*word*,*pos*) 如果*word* 中的位 *pos* 被设置，则返回一个**逻辑值**。
- **ibits**(*word*,*pos*,**len**) 返回一个整数（与 *word* 相同类型），其中位 $p, \cdots p+\ell-1$(向左扩展) 右对齐。
- **ibset**(*i*,*pos*) 接收一个整数，并返回将位 *pos* 设置为 1 后的整数。类似地，**ibclr** 清除该位。
- **iand**, **ior**, **ieor** 都操作两个整数，返回按位与 / 或 /xor 的结果。
- **mvbits**(*from*,*frompos*,**len**,**to**,*tpos*) 在两个整数之间复制一个位范围。

30.8 命令行参数

现代 Fortran 提供了查询 <code> 命令行参数 </code> 的函数。首先**命令 _ 参数 _ 计数**查询参数的数量。这不包括命令本身，所以比 main 函数的 C/C++*argc* 参数少一个。

```
// / 命令。F90 如果 ( 命令 _ 参数 _ 计数 ()==0) 则打印
*,"这个程序需要一个参数 " 停止 1 结束如果
```

命令可以通过获取 _ 命令。

命令行参数通过 获取 _ 命令 _ 参数。这些是 C/C++ 中的字符串，但你必须在预先指定它们的长度：

```
// / 命令。F90 字符 (len=10):: 大小 _
字符串整数 :: 大小 _num
```

将这个字符串转换为整数或类似操作需要一些格式技巧：

```
// / 命令。F90 调用 get_ 命令 _ 参数 ( 数字 =1, 值 = 大小 _ 字符串 )
读取 ( 大小 _ 字符串 ,'(i3)') 大小 _num
```

(参见第 41.3 节)

30.9 Fortran 类型种类

30.9.1 种类选择

种类可用于请求指定精度的类型。

- 对于整数，您可以使用选定的 _int_ 种类 (n)。
- 对于浮点数，可以指定有效数字的位数，并可选地指定十进制指数范围选定 _real_kind$(p[,r])$ 的有效数字位数。

相反地，可以使用函数 precision (不适用于整数)、 range 、 storage_size 再次检索此类类型的属性。

精度和 / 或范围的声明：

Code:

```
1// /kind.
integer, 参数:: &
      i12 = 选中_int_种类(12), &
      p6 = 选中_real_种类(6), &
      p10r100 =
  selected_real_kind(10,100), &
       400 = selected_real_kind(r=400), &
       20 = selected_real_kind(20), &
       40 = selected_real_kind(40)
integer(kind{i12}) :: i
real(kind=,p6)::x
real(kind=,p10r100) :: y
real(kind=,r400) :: z
real(kind=,p20) :: p
```

Output
[基本类型] 选择

```
 Kinds:        8     4     8    16
   16    -1
精度/范围/位:
 整数12:             0    18    64
 精度6:              6    37    32
 p=10r=100 :        15   307    64
 范围=400     :     33  4931   128
 p=20         :     33  4931   128
```

同样地，您可以指定常量的精度。编写 3.14 通常会是一个单精度实数。

添加单精度 / 双精度常量，以双精度打印：

代码:

```
1// /e0.F90
real(8) :: x, y, z
x = 1
y = .1
z = x+y
print *, z
x = 1.d0
y = .1d0
z= x+y
print *, z
```

输出
[基本] 类型大小:

```
 1.1000000014901161
 1.1000000000000001
```

您可以使用查询来查看数据类型占用多少字节 **kind**。

字节数决定了数值精度：

- 4字节计算有相对误差 $\approx 10^{-6}$
- 8字节计算具有相对误差 $\approx 10^{-15}$

Also different exponent range: max $10^{\pm 50}$ and $10^{\pm 300}$ respectively.

F08: 存储_大小 报告位数。

F95: **bit_size** works on integers only.

c_sizeof reports number of bytes, requires *iso_c_binding* module.

代码：

```
1// / 绑定.F90
2   使用 iso_c_ 绑定
3   implicit none
4   integer, 参数:: &
5         p6 = 选定的实数定的 _ 实数 kind(6) &
6
7   real(kind=,p6) :: x4
8   real(kind=,p12)::x8
910 格式 (i2" 数字占用 ",i3," 字节 ")
10  打印 10,6,c_sizeof(x4)
11  print 10,12,c_sizeof(x8)
```

输出
[基本 f] 绑定：

```
 6 位需要          4 字节
12 位需要          8 字节
```

强制常量为实型 (8)：

```
real(8) :: x, y = x .
14 d0 y =  = 3.
022 e-23
```

- 使用编译器标志（如 -r8 ）强制所有实数为 8 字节。
- 编写 3.14d0
- x = 实型 (3.14, kind=8)

30.9.2 范围

您可以使用函数 huge 来查询类型的最大值。

代码：

```
1// /deftypes.
2   整数 :: idef
3   Real :: rdef
4   实 (8):: rdouble
5
6   print 10,"integer is kind",kind(idef)
7   print 10,"integer max is",huge(idef)
8   print 10, "real is kind", , kind( rdef )
9   print 15, "real max is", , huge （rdef）
10  print 10, "real8 is kind", , kind( rdouble )
11  print 15,"real8 max is",huge(rdouble)
```

输出
[类型] 定义：

```
integer is kind                4
 integer max is 2147483647
   实数是类型                  4
    实数最大是 0.3403E+39
  实数 8 是类型                8
   实数 8 最大是 0.1798+309
```

使用 *ISO* 绑定 有一种更系统的方法。

整数：

// /inttypes..F90Integer(kind=
Int8):: i8Integer(kind=
Int16)::i16Integer(kind=
Int32)::i32Integer(kind=Int64)::i64

代码:

```
1// /inttypes.F90 2 print 10," 检查支持的类型 :"
3 print 10," 定义的 int 类型的数量 :",
size(INTEGER_KINDS) 4 print 10," 这些是支持的类
型 :",INTEGER_KINDS 5 print 15," 预定义类型
INT8,INT16,INT32,INT64:",& 6
INT8,INT16,INT32,INT64 7 print * 8 print 20,"
kind Int8 最大值是 ",huge(i8) 9 print 20,"kind
Int16 最大值是 ",huge(i16) 10 print 20,"kind
Int32 最大值是 ",huge(i32) 11 print 20,"kind
Int64 最大值是 ",huge(i64)
```

输出 [typef] int:

```
  Checking on supported
    types:
  number of defined int
    types:  5
these are the supported
    types:  1   2   4   8
 16
Pre-defined types
    INT8,INT16,INT32,INT64

       1   2   4   8

    kind Int8 max is
            127
   kind Int16 max is
          32767
   kind Int32 max is
    2147483647
   kind Int64 最大值是
 9223372036854775807
```

浮点数:

```
// /iso.F90useiso fortran env_        _
implicit nonereal(kind=real32) ::
x32real(kind=real64) ::x64print*,"32 位
最大浮点数 :",huge(x32)print*,"64 位最大浮
点数 :",huge(x64)
```

30.10 快速比较 Fortran 与 C++

30.10.1 语句

Some of it is much like C++:

- 赋值:

```
x = y
x = 2*y / (a+b)
z1 = 5; z2 = 6
```

  (注意语句末尾没有分号。)

- I/O
- 条件语句和循环

Different:

- 函数定义和调用
- 数组语法

- 面向对象编程
- 模块

30.10.2 输入 / 输出，或我们说的 I/O

- 输入：

  READ *, n

- 输出：

  <code>PRINT</code><code>*,</code><code>n</code>

There is also **Write**.

‘星号’ 表示使用默认格式。其他用于文件
和格式的读写语法。

30.10.3 表达式

- 几乎与 C++ 相同
- 异常：r**a 用于幂运算 r^a。
- 模数（C 语言中的 % 运算符 ++）是一个函数：**MOD**(7,3)。

- Long form:
  .and. .not. .or. .lt. .le. .eq. .ne. .
  ge. .gt. .true. .false.
- Short form:
  < <= == /= > >=

转换通过函数完成。

- **INT**: 截断 ;**NINT** 四舍五入
- **REAL, FLOAT, , SNGL,**
- **CMPLX**, **CONJG**,*AIMIG*

http://userweb.eng.gla.ac.uk/peter.smart/com/com/f77-conv.htm

复数存在；章节 30.5。

字符串由单引号或双引号分隔。更多内
容，请参见章节 35。

30.11 复习问题

Exercise 30.3. 这个片段的输出是什么，假设 i,j 是整数?

```
// /div.F90integer:: idiv/* ... */i = 3 ;
j = 2 ;idiv = i/jprint*,idiv
```

Exercise 30.4. What is the output for this fragment, assuming `i,j` are integers?

```
// /div.F90
  real     :: fdiv
    /* ... */
  i = 3 ; j = 2 ; fdiv = i/j
  print *,fdiv
```

练习 30.5. 在声明中

real(4) ::
xreal(8) :: y

4 和 8 代表什么?

使用其中一个或另一个的实际意义是什么?

练习 30.6. 编写一个程序:

- 显示消息类型一个数字,
- 接受一个整数数字（使用 Read ）,
- 创建另一个变量，该变量是那个整数的 3 倍加 1,
- 然后打印出第二个变量。

练习 30.7。 在以下代码中，如果 value 非零，您期望输出什么?

```
// /d0.F90 real(8) :: value8,should_be_value real(4) :: value4 /
* ...*/ print *,".. original value was:",value8 value4 = value8
print *,".. copied to single:",value4 should be value =value4
_  _print *,".. copied back to double:",should_be_value
print *,"Difference:",value8-should_be_value
```

30. Fortran 基础

第 31 章

条件语句

31.1 条件语句的形式

Fortran 条件语句使用 `if` 关键字：

单行条件语句：

```
if ( test ) statement
```

完整的 if 语句是：

```
if ( something ) then
   !! something_doing
else
   !! otherwise else_
end if
```

‘else’ 部分是可选的；你可以嵌套条件语句。

你可以标记条件语句，这对可读性有好处，但不会增加任何功能：

checkx: if (... some test on x...) thenchecky: if (... some test on y...) then... code...end if checkyelse checkx...code ... end ifcheckx

31.2 运算符

Operator	old style	meaning	example
==	.eq.	等于	x==y-1
/=	.ne.	不等于	x*x/=5
>	.gt.	大于	y>x-1
>=	.ge.	大于或等于	sqrt(y)>=7
<	.lt.	小于	
<=	.le.	小于或等于	
	.and. .or.	and, or	x<1 和 x>0
	.not.	not	.not.(x>1 .and. x<2))
	.eqv.	equiv (iff, not XOR)	
	.neqv.	非等价（异或）	

逻辑运算符如 . 与 . 在 C++ 中不是短路形式。子句可以按任意顺序求值。

练习 31.1. 读取三个成绩：代数、生物学、化学，每个成绩按 1 · · · 10 量表。计算平均成绩，条件如下：

- 代数总是包含在内。
- 生物学只有在它提高平均成绩时才包含在内。
- 化学只有在它是 6 分或更高时才包含在内。

31.3 选择语句

Fortran 的 C++ `case` 语句的等价物是 `select`。它接受单个值或范围；适用于整数和字符字符串。

测试单个值或范围，整数或字符：

```
// /select.F90Select Case(i)Case(:-1) !范围一个和
更小print *,"Negative"Case (5)print *,"Five!"
Case (0)print *,"Zero."Case(1:4,6:) !其他情况，不
能有 (1:)print *,"Positive"end Select
```

编译器会检查重叠的情况!

情况值需要是常量表达式。

默认情况使用一个 `case default` case 来覆盖。

31.4 布尔变量

布尔值的 Fortran 类型是 `Logical`。

这两个字面量是 `.true.` 和 `.false.`

练习 31.2. 打印一个布尔变量。在 true 和 false 的情况下，输出看起来像什么?

31.5 已弃用的条件语句

旧版本的 Fortran 有其他形式的 `if` 语句，你仍然可能在代码中遇到。The`if`，算术 在 Fortran90 中被宣布为已弃用，并在 Fortran2018 中被删除。

31.6 复习问题

练习 31.3. C++ `switch` 和 Fortran `Select` 语句之间的概念差异是什么?

31. 条件语句

第 32 章

循环结构

32.1 循环类型

Fortran 有常见的索引循环和 ‘while’ 循环。基本循环有变体，并且都使用 `do` 关键字。最简单的循环有一个循环变量、一个上限和一个下限。

```
integer :: i

do i=1,10! 带有 i
的代码 end do
```

你可以将步长（可以是负数）作为第三个参数包含：

按 3 的步长：

```
do i=1,10,3! 带有
i 的代码 end do
```

Counting down:

```
do i=10,1,-1
  ! code with i
end do
```

循环变量在循环外部定义，所以在循环结束后它将有一个值。

- Fortran 循环在执行前确定迭代次数；循环将执行那么多次，除非你退出。
- 你不允许修改迭代变量。
- 非整数循环变量曾经是被允许的，现在不再。

while 循环有前置测试：

```
do while (i<1000)
  print *,i
  i = i*2
end do
```

32.2 控制流的中断

对于不确定次数的循环，你可以使用 `while` 测试，或者完全省略循环参数。在这种情况下，你需要使用 `exit` 语句来停止迭代。

Loop without counter or while test:

```
do
  call random_number(x)
  if (x>.9) exit
  print *,"Nine out of ten exes agree"
end do
```

与 C++ 中的 break 对比。

Skip rest of current iteration:

```
do i=1,100
  if (isprime(i)) cycle
  ! do something with non-prime
end do
```

与 C++ 中的 continue 对比。

你可以标记循环
与退出语句:

```
// / 标记。F90 外部: do i=1, 10 内
部: doj=1, 10 测试: if (i*j>42)
thenprint *, i, j 退出外部 end
if 测试enddo 内部 end do 外部
```

标签需要与 `do` 位于同一行，如果你使用标签，你需要在 `end do` 行中提及它。

循环和退出可以应用于多个级别，如果 do 语句被标记。

```
外部: doi = 1, 10 内部:
doj = 1, 10if (i+j>15) 退出
外部 if (i==j) 循环内部
enddo 内部 enddo 外部
```

32.3 隐含的 do 循环

存在可以通过一个表达式和一个循环头来在一行中编写的 do 循环。实际上，这种 隐含的 *do* 循环 成为索引表达式的总和。这对于 I/O 很有用。例如，迭代一个简单的表达式:

如果你在一个打印语句中循环，每个打印语句都在新的一行；
使用隐式循环在一行中打印。

```
打印 *,(2*i,i=1,20)
```

你可以迭代多个表达式：

```
Print *,(2*i,2*i+1,i=1,20)
```

这些循环可以嵌套：

```
Print *,( (i*j,i=1,20), j=1,20 )
```

也适用于阅读。

这种结构特别适用于打印数组。

练习 32.1。 使用隐式 do 循环机制打印一个三角形：

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

直到输入的一个数字。

32.4 Obsoleteloop statements

旧版本的 Fortran 有其他形式的 。语句，你仍然可能在代码中遇到。自 Fortran2018 起，`do` 循环必须以 `end do` 或 `continue` 结束。共享终止也是一个已删除的特性。

Fortran 有一个 `goto` 语句。虽然这在 1950 年和 60 年代是必要的，但现在被认为是一种糟糕的编程实践。它的大部分传统用途都可以用 `cycle` 和 `exit` 语句来覆盖。`continue` 语句，通常用作 `goto` 的目标，现在也很少使用了。

32.5 复习问题

Exercise 32.2. What is the output of:

```
do i=1,11,3
print *,i end
do
```

What is the output of:

```
do i=1,3,11
print *,i end
do
```

32. 循环结构

第 33 章

过程

程序可以有子程序：代码的一部分，出于某种原因你想将其与主程序分开。这些术语称为过程。虽然这实际上是一个关键字，但你直到第 37.5 节才会看到它；在本章中，我们只考虑子程序 和 函数。

如果你将代码组织在一个文件中，则推荐的结构是：

定义过程的 simplest way:
in `Contains` part of main program.

```
程序 foo
  < declarations>
  < 可执行语句 >
  包含
    < procedure definitions >
结束程序 foo
```

有两种过程：函数和子程序。稍后介绍。

也就是说，过程位于主程序语句之后，由一个包含子句分隔。

通常，这些是过程的放置位置：

- 内部：在程序的包含 子句之后：

  ```
  程序 foo... 内容 ... 包
  含子程序 bar() 子程序结
  束 bar 程序结束 foo
  ```

  这是本章我们关注的模式。
- 在模块；参见节 37.2。
- 外部：过程不是程序 或 模块的内部部分。这可能在第三方库的情况下发生，或者从另一种语言链接的代码中发生。在这种情况下，最安全的方法是通过接口 规范；节 42.1。

33.1 子程序和函数

Fortran 有两种过程：

- 子程序，它们有点像 C++ 中的 void 函数：它们可用于组织代码，并且只能通过它们的参数将信息返回给调用环境。
- 函数，它们像 C++ 中的带返回值的函数。

这两种类型具有相同的结构，大致与主程序相同。对于子程序：

```
子程序 foo( <参数> )<变量声明>
可执行语句 >end subroutinefoo
```

以及对于函数：

```
返回类型 函数 foo( <参数> )<变量声明>< 可执
行语句 >endfunction foo
```

声明函数有另一种语法，参见章节 33.2.1。

退出过程有两种方式：

1. 控制流到达过程体的末尾：

```
subroutine foo()
  statement1
  ..
  statementn
end subroutine foo
```

or

2. execution is finished by an explicit `return` statement.

```
子程序 foo()print *,"foo"
if
(something)returnprint
*,"bar"endsubroutine
foo
```

在第一种情况下，`return` 语句是可选的。与 C++ 不同的是，`return` 语句不表示函数的返回结果值。

练习 33.1。 重写上述子程序 `foo` 不使用 返回 语句 .

子程序通过调用 语句被调用：

```
调用 foo()
```

Code:

```
1 // /printone.F90
2 program printone
3   implicit none
4   call printint(5)
5 contains
6   子程序 printint(invalue)
7     implicit none
8     integer :: invalue
9     print *,invalue
10  end subroutine printint
11 结束程序 printone
```

Output

```
[funcf] printone:
           5
```

参数类型在主体中定义，而不是在头部

代码:

```
1// /addone.
2 program addone
3   implicit none
4   整数 :: i=5
5   调用 addint(i,4)
6   print *,i
7 包含 b t
8   su rou ine addint(inoutvar,addendum)
9     implicit none
10    整数 i    ::  noutvar,addendum
11    inoutvar =inoutvar + 补充说明
12  结束子程序 addint
13 结束程序 addone
```

输出

```
[funcf] addone:
           9
```

参数始终是 ‘按引用’ 传递!

Fortran 中的递归函数需要使用 **recursive** 关键字显式声明。

将函数声明为**递归函数**

代码:

```
1// /fact.F90
2   递归整型函数
      fact(invalue) &
3       结果 (val)
4     implicit none
5     整数 ,intent(in) ::invalue
6     如果 (invalue==0) 则
7       val = 1
8     else
9       val = invalue * fact(invalue-1)
10    end if
11  end function fact
```

输出

```
[funcf] fact:
echo 7 |  ./阶乘
          7 阶乘是
     5040
```

注意**结果** 子句。这可以防止歧义。

33.2 返回结果

虽然一个 **子程序**只能通过其参数返回信息，但一个 函数 过程返回一个显式的结果：

逻辑函数测试（*x*）**隐式无类型 real**：：
x 测试 = 一些 _ 测试 _ 在（*x*）**返回**！可选，见上文**结束函数** 测试

您可以看到结果不是在 **返回** 语句中返回的，而是通过赋值给函数名。像之前一样，**返回** 语句是可选的，并且仅指示控制流结束的位置。

Fortran 中的 函数 是一个过程，它将其结果返回给调用程序，类似于 C++ 中的非 void 函数

- **子程序**与 **函数**：
  比较 *void* 函数与 C++ 中的非 void 函数
- 函数头：
  返回类型，关键字**函数**，名称，参数
- 函数体包含语句
- 结果通过赋值给函数名返回
- 使用：y = f(x)

Code:

```
1 // /plusone.F90
2 program plussing
3   implicit none
4   integer :: i
5   i = plusone(5)
6   print *,i
7 contains
8   integer function plusone(invalue)
9     implicit none
10    integer :: invalue
11    plusone = invalue+1 ! note!
12  end function plusone
13 end program plussing
```

Output

```
[funcf] plusone:
           6
```

- The function name is a variable
- . . . that you assign to.

函数不是通过**调用**来执行的，而是通过在表达式中使用来执行的：

```
if (test(3.0) .and. something_else) ...
```

You 现在有以下几个情况来使函数知名 in the main program:

- 如果函数在一个 **contains** 部分中，它的类型在主程序中是已知的。
- 如果函数在一个模块（见下节 37.2 ）中，它通过一个 **use** 语句变得知名。*F77* 注：没有模块和 **contains** 部分，你需要在调用程序中显式声明函数类型。安全的方式是通过使用一个 **interface** 规范。

练习 33.2. 编写一个程序，要求用户输入一个正数；非正数输入应被拒绝。填写这个代码片段中缺失的行：

代码：

```
1// / 读取位置 .F90
2 program readpos
3   implicit none
4   real(4) :: userinput
5   print*,"输入一个正数 :"
6   用户输入 = 读取 _ 正 () 它 "Th  k  f
7   pr n  *,   an  you  or ,userinput
8 包含
9   实 (4) 函数 读取 _ 正 ()
10    隐式无 /*
11        ... */
12  结束函数读取 _ 正
13 结束程序 读取正
```

输出 [funcf] 读取位置：

```
输入一个正数：
不， 不是      -5.00000000
不， 不是       0.00000000
没有， 不是    -3.14000010
感谢您                2.48000002
```

33.2.1 The ‘result’ keyword

除了赋值给函数名之外，还有一个返回函数结果的第二种机制，即通过**结果** 关键字。

```
函数 some_ 函数 () 结果 (x) 隐式无 ::x!!
内容 x= ！一些计算 end function
```

您在这里看到

- 赋值给名称缺失，
- 函数名没有输入；但是
- 相反，有一个输入的局部变量被标记为结果。

33.2.2 ‘contains’ 子句

```
// / 不包含。F90
程序 不包含
  implicit none
  调用 执行什么 ()
结束程序不包含

子程序 DoWhat(i)
  implicit none
  整数 :: i
  i = 5
子程序结束 DoWhat
```

Warning only, crashes.

```
// /wron contain F90
  g程序 包含作用域
  implicit none
  调用 DoWhat()
contains
  subroutine DoWhat(i)
    implicit none
    整数 ::i
    i = 5
  end subroutine DoWhat
结束程序包含范围
```

Error, does not compile

Code:

```
1// /nocontain2.
2Program NoContainTwo
3    implicit none
4    整数 :: i=5
5    调用 DoWhat(i)dP
6 enNC rogram   o ontainTwo
7 8<subroutine>DoWhat</subroutine>(x)

9    implicit none
10   real :: x
11   print *,x
12 end subroutine DoWhat
```

Output[f

```
f] unc    nocontaintype:

不包含 2.F90:15:16:

   15 |      调用 DoWhat(i)1
      |
警告： 参数'x' 在 (1); 处类型不匹配
                                    ssed
     INTEGER<4>to REAL<4>
     [-W参数 - 不匹配 ]
    7.00649232E-45
```

At best compiler warning if all in the same file

33.3 参数

Arguments are declared in procedure body:

subroutine f(x,y,i) implicit none integer,intent(in) :: i real(4),intent(out) :: x real(8),intent(inout) :: y x = 5; y = y+6end subroutinef ! 以及在主程序中调用 f(x,y,5)

声明 ‘意图’ 是可选的，但强烈建议。

- 所有内容都是通过引用传递的。不用担心大对象被复制。
- 可选的意图声明：使用 **in**, **out**, **inout** 限定符来向编译器明确语义。

术语虚拟参数 是 Fortran 在过程定义中称为的参数：

```
subroutine f(x) ! 'x' is dummy argument
```

过程调用中的参数是 实际参数：

```
调用 f(x) ! 'x' 是实际参数
```

编译器检查你的意图是否与你的实现一致。这段代码不合法：

```
// / 意图。F90 子程序 ArgIn(x) 隐
式无，意图（输入） ::xx = 5! 编译
器抱怨 end subroutineArgIn
```

自我保护：如果你声明了例程的预期行为，编译器可以检测编程错误。

允许编译器优化：

```
x =f() 调用
ArgOut(x) 打印
*,x
```

Call to `f` removed

```
do i=1,1000
  x = ! something
  y1 = .... x ....
  call ArgIn(x)
  y2 = ! same expression as y1
```

y2 与 y1 相同，因为 x 没有改变

(可能需要进一步规范，因此这不是主要的理由。)

练习 33.3. 编写一个名为 trig 的子例程，它接受一个数字 α 作为输入，并将 $\sin\alpha$ 和 $\cos\alpha$ 返回给调用环境。

33.3.1 关键字和可选参数

在过程调用中，参数总是可以用它们对应的参数名给出。这称为 关键字参数，有时用于防止混淆。

```
! confusing:
调用两个 1.1, 2.2, 3.3, 4.4 )! 更好：调用两个 _ 点 (
x1=1.1, x2=2.2, y1=3.3, y2=4.4 )
```

未使用关键字指定的参数称为 位置参数。你可以混合使用位置参数和关键字参数，但如果使用关键字指定了一个参数，后续的所有参数也都需要使用关键字。

- 使用 形式参数 的名称作为关键字。
- 关键字参数必须放在最后。

代码：

```
1// / 关键字。F90
2   call say_xy(1
3      ,2)调用 说_xy(x=1,y=2)
4   调用 说_xy(y=2,x=1)
5   调用 说_xy(1,y=2)!  11
6   – cба 21say xy y= , ) ! ILLEGAL
7 包含
8   子程序 说_xy(x,y)
9     implicit none
10    整数,intent(in) ::x,y
11    print*, "x"=", x, ", y"=", y
12  end subroutine say_xy
```

输出
[funcf] 关键字：

```
x=           1 , y=           2
x=           1 , y=           2
x=           1 , y=           2
x=           1 , y=           2
```

一个关系概念是 可选参数。一个参数可以被标记 可选，之后它就可以从过程调用中省略。

- 可选参数可以出现在参数列表的任何位置；
- 如果你在参数列表中省略了一个可选参数，那么后续的所有参数都需要通过关键字指定。

- 该过程可以通过函数来测试是否提供了可选参数

  Present

- 额外指定符：可选
- 参数的存在可以通过 Present

33.4 过程类型

位于主程序（或另一种程序单元）中，并通过 包含子句分隔的过程称为内部过程。这与 模块过程相对。

还有语句函数，它们是单语句函数，通常用于标识程序单元中常用的复杂表达式。推测编译器会 内联它们以提高效率。

标准库函数，如 sqrt，可以在一个 内置语句中这样声明

```
Intrinsic :: sqrt,cmplx
```

但这不是必需的。

入口语句非常奇怪，我拒绝讨论它。

33.5 局部变量 save-ing

通常，过程中局部变量表现得好像在过程调用时创建，并在其执行结束时消失。通过给它一个 *save* 属性，可以保留变量在调用之间的值。

```
subroutine whatever()
integer, save :: i
```

(这与 C++ 中的 *static* 变量大致对应。)

这里有一个主要的陷阱。如果你给局部变量一个初始化值:

```
子程序 无论什么 () 整数 ::
i  = 5
```

然后变量隐式地获得一个保存 属性，无论是否指定。初始化只执行一次，可能在编译时执行，在第二次过程调用时使用保存的值。

这可能让你困惑，如下面的示例所示:

Local variable is initialized only once,
second time it uses its retained value.

代码:

```
1 // /save.F90
2   整数函数 maxof2(i,j)
3     implicit none
4     整数,intent(in) ::i,j
5     整数 ::max=0
6     if (i>max) max = i
7     if (j>max) max = j
8     maxof2 = max
9   结束函数 maxof2
```

输出
[funcf] 保存:

```
Comparing:  1   3
            3
Comparing: -2  -4
            3
```

33. 过程

第 34 章

作用域

34.1 作用域

Fortran ‘没有大括号’：你不容易用局部变量像 C 那样创建嵌套作用域 ++。例如，`do` 和 `end do` 之间的范围不是作用域。这意味着所有变量都必须在程序或子程序的顶部声明。

34.1.1 Variables local to a program unit

Variables declared in a subprogram have similar scope rules as in C++:

- 它们的可见性由它们的文本作用域控制：

```
子程序 Foo()integer:: i! ‘i’ 现在可以
使用 call Bar()! ‘i’ 仍然存在
EndSubroutine Foo 子程序Bar() ！没有
参数！这里的 Foo 的 ‘i’ 是未知的
EndSubroutine Bar
```

- 它们的动态作用域是它们声明的程序单元的生存期：

```
子程序 Foo() 调用 Bar() 调用 Bar()EndSubroutine
Foo 子程序 Bar()Integer:: i! ‘i’ 每次调用 Bar 时
都会被创建 EndSubroutine Bar
```

（最后一个例子有一点微妙之处；参见第 33.5 节关于过程变量上的保留 属性。）

34.1.1.1 模块中的变量

模块中的变量（第 37.2 节）具有与其调用层次结构无关的生存期：它们是静态变量。

34.1.1.2 其他机制用于创建静态变量

在 Fortran 获得递归函数功能之前，每个函数的数据被放置在一个静态确定的位置。这意味着当你第二次调用一个函数时，所有变量仍然具有它们上次具有的值。要在现代 Fortran 中强制这种行为，你可以将 `Save` 规范添加到变量声明中。

创建静态数据的另一种机制是 **`Common`** 块。这不应该使用，因为一个**模块**是解决同样问题的更优雅的方案。

34.1.2 内部过程中的变量

一个 内部过程 (即放置在程序单元的 **`Contains`** 部分中的一个) 可以接收来自包含程序单元的参数。它还可以通过称为主关联的过程直接访问在包含程序单元中声明的任何变量。

这些规则很混乱，尤其是在考虑变量的隐式声明时，因此我们不建议依赖它。

第 35 章

字符串处理

35.1 字符串表示

字符串可以由单引号或双引号包围。这使得在字符串中包含另一种类型的引号更加容易。

```
//F90
  print *,'This string was in single quotes'
  print *,'This string in single quotes contains a single '' quote'
  print *,"This string was in double quotes"
  print *,"This string in double quotes contains a double "" quote"
```

35.2 字符

数据类型字符 既用于字符也用于字符串。因此，请查看下一节。

35.3 字符串

Fortran 字符串的长度在创建字符串时通过 `len` 关键字指定：

```
字符 (len=50) ::
mystringmystring= "短字符串"
```

The `len` 函数也给出字符串的长度，但请注意，那是分配的长度，而不是你放入其中的非空白内容。

```
字符串长度，带 / 不带修剪。
```

代码:

```
1 // /quote.
2   字符 (len=12):: strvar/
3   /  * ... *
4   strvar = "word"
5   print *,len(strvar),len(trim(strvar))
```

输出
[stringf] strlen:

```
          12           4
```

为了获得更直观的字符串长度，即最后一个非空白字符的位置，你需要修剪 字符串。

连接操作使用双斜杠完成:

代码:

```
1 // /quote.F90
2   character(len=10) :: firstname,lastname
3   character(len=15) :: shortname,fullname
4     /* ... */
5   firstname = "Victor"; lastname =
      "Eijkhout"
6   shortname =firstname // lastname
7   print *,"without trimming: ",shortname
8   fullname = trim(firstname) // " " //
      trim(lastname)
9   print *,"with trimming: ",fullname
```

Output
[stringf]concat:

不修剪: *Victor*
Eijkh
带修剪 : *Victor Eijkhout*

35.4 转换

有时我们希望将字符串 123 转换为数字 123。让我们从简单的地方开始，通过查看字符及其 *ascii* 码。

35.4.1 字符转换

给定一个整数， `Char` 给出具有该 ascii 码的字符。这可以是一个可打印字符或一个不可打印字符:

Code:

```
1// /convert.
2 print *, "97 is a:", char(97) 3 print *, "84
is T:", char(84) 4 print *, "53 is 5:",
char(53) 5 print *, "11 is VT :", char(11), "."
```

输出
[字符串] ASCII:

97 是 :a84 是
T:T53 是 5:5
11 是 *VT*:
.

注意最后一个！

在另一个方向上， `Iachar` 给出字符的 ascii 码 aracter.

```
字符 :: 字符 整数 :: 代码字符 = "x"
代码 = iachar(字符)print*, 字符,
"有代码", 代码
```

注意 34 还有一个函数 ***ichar***，但它返回的是本地字符集的代码。在罕见的情况下，这可能不是 *ascii*。

练习 35.1. 编写一个测试来检查字符是否为小写：

代码：

```
1// /转换.F90it   "l
2   pr n   *,   ower t",islower("t")
3   打印, 较低T,islower( T) 打印" "
4   *,"较低 3",islower("3")
```

输出
[stringf] 较低：

较低 *t T*
较低 *T F*
lower<3>3</3>F

同样，编写一个测试 *isdigit*。

35.4.2 字符串转换

将数字和字符串之间的转换依赖于 I/O 章节（章节 41 ）的概念；参见节 41.5。

35.5 更多说明

除了使用 `len` 规范定义的字符之外，还有 s

```
字符 *80 ::str 字符, 维度 (80)::
str 字符 :: str(80
```

这些不应被使用。

35. 字符串处理

第 36 章

结构，嗯，类型

Fortran 有结构用于捆绑数据，但没有 `struct` 关键字：相反，你需要用 派生类型 的 **type** 关键字声明结构类型和该类型的变量。

36.1 派生类型基础

现在你需要

- 定义类型来描述其内容；
- 声明该类型的变量；并且
- 使用这些变量，但设置类型成员或使用它们的值。

类型名称 / **类型名称结束**块。
Member declarations inside the block:

```
类型 mytype 整数 :: 字
符数字 :: 名称实数
(4):: 值类型结束
mytype
```

类型定义在可执行语句之前。

创建类型变量与 C++ 类中的对象略有不同。在 C++ 中，类名本身可以作为数据类型使用；在 Fortran 中，你需要编写**类型** (*mytype*)。否则，它看起来像任何其他变量声明。

在主程序中声明类型变量：

```
类型 (mytype) ::struct1,struct2
```

使用类型名称初始化：

```
struct1 = mytype( 1, 'my_name', 3.7 )
```

Copying:

```
struct2 =struct1
```

如果你需要访问类型中的一个字段，有一个类似于 C++ 中 ‘点’ 表示法的表示法：在 Fortran 中，你使用百分号 %。

Access structure members with %
(compare C++ dot-notation)

```
类型(mytype) ::typed_
structtyped_struct%成员= ....
```

例如，我们使用几何项目中的 ‘点’ 结构。

```
//点类型.
  类型点
      real:: x, y
  结束类型点
```

```
// /pointtype.F90
  类型(点) ::p1,p2
<p>1 =<point>(2.5,
3.7)</point></p>
  p2 =p1
  print *,p1
  print *,p2%x,p2%y
```

注意，单独打印一个类型等同于按顺序打印其组件。

您可以拥有类型的数组：

```
type(my_struct) :: 数据类型(my_struct),维度
(10):: 数据_数组
```

36.2 派生类型和过程

结构可以作为过程参数传递，就像任何其他数据类型一样。在这个例子中，函数 *length*：

- 采用类型（点）作为参数；并且
- 返回一个实数（4）结果。
- 该结构被声明为意图（输入）。

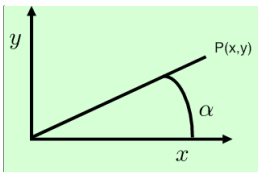
具有结构参数的函数：

```
//点类型.
  实数(4) 函数长度(p)
    implicit none t
     ype(point),intent(in) :: p
    length =sqrt( &
          p%x**2 + p%y**2 )
  end function length
```

Function call

```
//点类型.
  print *,"Length:",length(p2)
```

练习 36.1. 添加一个函数 *angle*，它接受一个 *Point* 参数，并返回 x- 轴和从原点到该点的线的角度。



您的程序应该读取 x,y 点的值并打印出以弧度为单位的角。

奖励：你能将角作为 π 的分数打印出来吗？所以

$$(1,1) \Rightarrow 0.25$$

您可以根据代码仓库中的文件 *point.F90* 来实现这一点

Exercise 36.2. Write a program that has the following:

- 一个包含实数的类型 `Point x,y`;
- 一个包含两个 `Point` 的类型 `Rectangle`，分别对应左下角和右上角。
- 一个具有一个参数的函数 面积：一个矩形。

Your program should

- 在同一行上接受两个实数，作为左下角点；
- 类似地，再次在同一行上，定义右上角的坐标；然后
- 打印出由这两个点定义的（轴平行）矩形的面积。

练习 36.3. 在之前的练习 36.2：
使用模块可以获得加分，使用面向对象的解决方案可
以获得双倍加分。

36.3 参数化类型

如果一个派生类型包含一个数组，你可能希望该数组的长度是可变的，而无需使数组动态分配。为此，Fortran 有 参数化类型：你可以定义一个类型，其包含以下组合：

- 一个具有属性 `len` 的参数，用作数组成员的长度；或
- 一个具有属性 `kind`，用作某些变量的类型；节 30.4.1.2。

示例：

```
// /parampoint.F90type
point(dim)integer,len ::dim
real,dimension(dim) ::xend
typepoint
```

我还没搞清楚如何设置变量：

```
// /parampoint.F90
  type(point(3)) :: p1,p2
  p1%x = [1.,2.,3.]
```

```
p2 = p1
print *,p2%x
```

这些类型可以正常传递：

```
// /parampoint. F90 real (4) function length( p )
implicit none type( point(3) ), intent( in ) :: p
length = sqrt( & p%x(1)**2 + p%x(2)**2 + p%
x(3)**2 ) end function length
```

第 37 章

模块

Fortran 拥有清晰的机制来导入数据（包括数值常量等）π、函数、在其他文件中定义的类型。这是通过模块完成的，模块使用 **module** 关键字定义。

Modules look like a program, but without main
(only 'stuff to be used elsewhere'):

```
// /typemod.F90
Module geometry
  type point
     real :: x,y
  end type point
  real(8),parameter :: pi = 3.14159265359
contains
  real(4) function length(p)
    implicit none
    type(point),intent(in) :: p
    length = sqrt( p%x**2 + p%y**2 )
  end function length
end Module geometry
```

注意数值常量。

通过 **use** 语句导入模块；放置在
implicit none 之前

Code:

```
1 // /typemod.F90
2 Program size
3   use geometry
4   implicit none
5
6   type(point) :: p1,p2
7   p1 = point(2.5, 3.7)
8
9   p2 = p1
10  print *,p2%x,p2%y
11  print *,"length:",length(p2)
12  print *,2*pi
13
14 end Program size
```

Output

```
[structf] typemod:
   2.50000000       3.70000005
 length:   4.46542263
   6.2831854820251465
```

练习 37.1。 做练习 36.2 并将所有类型定义和所有函数放在一个模块中。

37.1 用于程序模块化的模块

模块是 Fortran 支持分离编译的机制：你可以把你的模块放在一个文件中，主程序放在另一个文件中，并分别编译它们。

模块是子程序和类型定义以及常数和变量等数据的容器。模块不是结构或对象：只有一个实例。

注释 35 使用 语句有点类似于 *C* 中的 ***#include "stuff.h"*** 行。然而，请注意 *C++20* 也采用了模块，因为它们比基于预处理器的解决方案更干净。

37.2 模块定义

你使用模块有什么用?

- 类型定义：在多个程序单元中可以有相同的类型定义，但这不是一个好主意。在模块中只写一次定义，并以这种方式提供它。
- 函数定义：这使得函数在同一个程序的多个源文件中或多个程序中可用。
- 定义常量：对于物理模拟，将所有常量放在一个模块中并使用它，而不是每次都拼写常量。
- 全局变量：如果变量不适合一个明显的范围，请将它们放在模块中。

Any routines come after the `contains` clause.

备注 36 模块在 *Fortran90* 中被引入。在早期标准中，可以通过公共 块使信息全局可用。由于模块比公共块更干净，因此不再使用那些块了。

A module is m可以使用使用 关键字使其可用，该关键字需要放在 b efore the `implicit none`.

使用 放在前面的声明隐式

```
// /module.F90
Program ModProgram
  use FunctionsAndValues
  implicit none

  print *,"Pi is:",pi
  call SayHi()

End Program ModProgram
```

也行：

```
使用 mymodule, 仅: func1, func2 使
用 mymodule, func1 => 新_name1
```

如果你编译 amodule，你会在你的目录中找到一个 .mod 文件。（这与 C++ 中的 .h 文件有点像。）如果这个文件不存在，你就不可以 使用 该模块在另一个程序单元中，所以你需要先编译包含该模块的文件。

练习 37.2。 编写一个模块 `PointMod`，它定义一个类型 `Point` 和一个函数 `distance`，以使以下代码工作：

```
// /pointmain.F90 use pointmod implicit none
type(Point) ::p1,p2 real(8) :: p1x,p1y,p2x,p2y
read *,p1x,p1y,p2x,p2y p1 = point(p1x,p1y)
p2 = point(p2x,p2y) print *,"Distance:",
distance(p1,p2)
```

将程序和模块放在两个不同的文件中，并按如下方式编译：

```
ifort -g -c pointmod.F90 ifort -g -c pointmain.F90 ifort -g -
o pointmain pointmod.o pointmain.o
```

37.3 分离编译

本课程中的练习足够简单，以至于你可以将任何模块与你的主程序放在同一个文件中。然而，在实际应用中，你会为模块创建单独的文件，甚至可能为每个模块使用一个文件。

假设程序被分成两个文件：
theprogram.F90 和 themodule.F90。

- 编译模块：ifort -c themodule.F90；这会给出
- 一个 对象文件 (扩展：.o)，稍后会链接，并且
- 一个模块文件 modulename.mod。

- Compile the main program:
  `ifort -c theprogram.F90` will read the `.mod` file; and finally
- 将对象文件链接成一个可执行文件：
  `ifort -o myprogram theprogram.o themodule.o`
  The compiler is used as *linker*: there is no compiling in this step.

重要提示：在使用模块的任何（子）程序之前，都需要先编译该模块。

Fortran2008 标准引入了 子模块，这甚至可以进一步促进分离编译。

37.4 访问

默认情况下，使用模块的（子）程序可以访问模块的所有内容。然而，使用关键字 **private** 可以使模块内容仅在模块内部可用。您可以通过使用 **public** 关键字来显式指定默认行为。

public 和 **private** 都可以作为模块中定义的属性使用。有一个关键字 **protected** 用于数据成员，这些成员对模块外部代码是公开的，但不能被外部代码修改。

```
    // protect.F90Module 设置
implicit nonelogical,protected ::
has been initialized= .FALSE.
_    _containssubroutine
init()has been initialized= .
TRUE._    _end subroutineinit


End Module设置
```

```
// protect.F90if ( .not. has_
been_ 初始化 )thencall init()end if!!
WRONG does not compile:

  ! 已经初始化 = 。FALSE._    _
```

37.5 多态

模块 *somemodule*

```
模块 swap
模块过程 swapreal,swapint, swaplog, swappoint
END 模块

  包含子程序 swapreal... 结
  束子程序 swapreal 子程序
  swapint... 结束子程序
  swapint
```

37.6 运算符重载

您可以在类型上定义 + 或 * 等操作。

```
///Pust ype模F90
块 Typedef
  类型 int 类型整数
    :: val endtype inttype    ue
  接口运算符 <sub>(</sub>+<sub>)</sub>
  模块过程 <sub>ad</sub>          dtypes
  结束接口运算符 (+)
包含
```

```
函数 添加类型 (i1,i2)
  结果 (isum)
  隐式无 T
  ype (i'tt ype),intent(in) :: i1,i2
  类型 (int)::isum
  isum%value= i1%value+i2%value
结束函数添加类型
结束模块类型定义
```

您现在可以使代码看起来美观且简单：

代码：

```
//plustype.
Type(inttype) :: i1,i2,i3
i1 = inttype(1); i2 = inttype(2)
i3 = i1+i2
print *,"Sum:",i3%value
```

Output
[typef] plus:

```
Sum:           3
```

重载包括赋值运算符：

```
接口赋值 (=) 子程序 _ 接口 _ 体
END INTERFACE
```

您可以用点表示法定义新的运算符：

```
INTERFACEOPERATOR
(.DIST.)模块过程 calcdist
END INTERFACE
```

37. 模块

第 38 章

类和对象

38.1 类

Fortran 类基于 类型 对象。某些方面与 C++ 类似。例如，指定数据成员和方法时使用相同的语法：

```
print *, 对象 %xfield 对象 %
set_xfield(5.1)
```

其他方面略有不同：在 C++ 中，你可以在一个类定义中编写所有数据和函数成员；在 Fortran 中，数据和函数是分别声明的。

一个主要区别在于函数方法的定义方式：对象本身变成了一个额外的参数。你稍后会看到详细信息。

首先关于 Fortran 类的组织方式。类是模块内的类型定义，有一个额外的子句指示该类型可用的函数方法。

You define a type as before, with its data members, but now the type has a **contains** for the methods:

```
// /mult1.F90
模块 multmod

  类型 Scalar
     real(4) :: value
   contains
     procedure,public :: &
           printme, scaled
  end type Scalar

  包含！方法 /* ...*/
  结束模块 multmod
```

如上所述，在对象上调用方法使用与访问其数据成员相同的语法。

类似于 C++ 的方法调用

Code:

```
1 // /mult1.F90
2 Program Multiply
3   use multmod
4   implicit none
5
6   type(Scalar) :: x
7   real(4) :: y
8   x = Scalar(-3.14)
9   call x%printme()
10  y = x%scaled(2.)
11  print '(f7.3)',y
12
13 end Program Multiply
```

Output

```
[objectf] mult1:

The value is -3.140
 -6.280
```

方法定义与 C++ 略有不同，但如果你熟悉 Python，你会看到相似之处。如果一个方法使用一个参数被调用；

```
调用 对象 % 函数（参数）
```

该函数有两个参数，第一个是对象，第二个是括号内的参数。

此外，第一个参数的类型是**类型**（对象），但在方法中它被声明为 **类**（对象）。

Note the extra first parameter:
which is a **Type** but declared here as **Class**:

```
//mult1.F90
  子程序 打印（我）
    隐式无
    类（标量）:: 我
     打印'（"值是"，f7.3）'，我 % 值结束子程序打印

  函数 缩放（我，因子）
    隐式无类（标量）:: 我实型
    (4):: 缩放，因子

    缩放 = me% 值 * 因子
  结束函数缩放
```

总结：

- 类是一个 **类型**，后面跟着一个 **包含**子句，然后是**过程** 声明，
- ... 包含在模块中。
- 实际方法放在**包含** 模块的部分
- 方法的第一个参数是对象本身。

```
!!! pointexample.F90 模块
模块 PointClass
  类型 ,public:: Point
    real(8) :: x, y
  contains
    过程, 公共 :: &
        距离
  结束类型点
包含 !!
    ... distance function ...
  /* ... */
结束模块点类
```

```
!!! pointexample.F90 程序点测试
使用 点类
implicit none
type(Point) :: p1,p2

p1 = point(1.0,1.0)
p2 = 点 <4.,5.>

print *,"Distance:",&
    p1% 距离 p2

结束程序点测试
```

	C++	Fortran 在‘类型’中
成员 方法	在对象中 在对象中	接口：在类型中 实现：在模块中
构造函数 对象本身 类成员 对象的方法	默认或显式 ‘this’ global variable 周期	none 第一个参数 通过第一个参数访问 百分比

练习 38.1。 取点示例程序并添加一个距离函数 :

```
类型（点）::p1,p2! ... 初
始化 p1,p2dist =p1% 距离
(p2)! ... 打印距离
```

您可以基于存储库中的文件 *pointexample.F90* 来实现 *y*

练习 38.2。 为点类型编写一个 add 方法:

```
Type(Point) ::
p1,p2,sum! ... 初始化
p1,p2sum= p1%add(p2)
```

函数 add 的返回类型是什么?

38.1.1 Final procedures: destructors

析构函数 的 Fortran 等价物是一个 *final* 过程，由 *final* 关键字指定。

```
// /final.F90 包含
final:: &print_
finalend typeScalar
```

final 过程有一个应用于其类型的单个参数:

```
// /final.F90subroutine print_final(me)implicit none
type(Scalar) :: meprint ’("On exit: valueis",f7.3)’,
me%valueend subroutine print_final
```

当派生类型对象被删除时，最终过程会被调用，除了在程序结束时：

```
// /final.F90call tmp_
scalar()containssubroutine
tmp_scalar()type(Scalar) ::
xreal(4) :: y x=
Scalar(-3.14)endsubroutine
tmp_scalar
```

38.2 继承

继承:

```
type,extends(baseclas) ::derived_类
```

Pure virtual:

```
type, 抽象
```

http://fortranwiki.org/fortran/show/Object-oriented+ 编程

当然最好将类型定义和方法定义放在一个模块中，这样你就可以 使用它。

Mark me将方法声明为 `private` ， 以便它们只能作为部分使用 e `type`:

```
    // / 点。F90 模块 点类 /* ...
*/private contains 子程序
setzero(p) implicit none 类 (
点 ) ::pp%x  = 0.d0 ; p%y = 0.
d0end subroutinesetzero/* ...
*/End Module点类
```

38.3 运算符重载

对于许多物理量，定义一个加法运算符是有意义的。这使得可以编写

```
类型 (X) ::x,y,z!
内容 x =y+z
```

为了说明，让我们定义一个非常简单的类：

```
// / 标量.F90 类型，公共 :: 标量
场实数 (8):: 值包含

    过程，公共 :: 设置，打印
    过程，公共 :: 添加
  结束类型标量场
```

我们定义了一些明显的方法：

```
// / 标量.F90
  子程序 设置 (v,x)
    implicit none
    类（标量场） ::v
    实型 (8), 意图（输入） ::x

    v%value = x
  end subroutine set

  子程序 print<subscript>(v)</subscript>
    implicit none
```

```
  类 (ScalarField) ::v

  print '(f7.4)', v%value
结束子程序 print

// / 标量.F90
调用 u% 设置 (2.d0)
调用 v% 设置 (1.d0)!

z = u%add(v)
z = u+v
```

在定义加法运算符之前，首先需要定义一个加法函数：

```
// /scalar.F90 函数 add(in1,in2) 结果 (out) 隐式无
类 (ScalarField),intent(in) :: in1 类型
(ScalarField),intent(in) :: in2 类型
(ScalarField) :: out out%value= in1%value +
in2%value 结束函数 add
```

这个函数需要满足一些条件：

- 这个函数需要有两个输入参数。显然。
- 输入参数需要声明 `Intent(In)`。这稍微不那么明显，但它是有道理的，因为加法参数的参数不是以正常方式传递的。

将函数转换为运算符然后非常简单。

接口块：

```
// / 标量.F90 接口运算符 (+)
模块过程 add
```

结束接口操作符 (+)

Exercise38.3. 扩展上述示例程序，使类型存储数组而不是标量。

Code:

```
// field.F90
  integer,parameter :: size = 12

  Type(VectorField) :: u,v,z

  call u%alloc(size)
  call v%alloc(size)
  call u%setlinear()
  call v%setconstant(1.d0)
  !  z = u%add(v)
  z = u+v
  call z%print()
```

Output

```
[geomf] field:

 2.0000  3.0000  4.0000  5.0000
    6.0000  7.0000  8.0000
   9.0000 10.0000 11.0000
   12.0000 13.0000
```

You can base this off the file `scalar.F90` in the repository

类似地，我们可以重新定义赋值运算符；参见 https://dannyvanpoucke.be/oop-fortran-tut5-en/。这带来了一些关于 浅拷贝 和 深拷贝 的复杂性。

第 39 章

Arrays

Fortran 中的数组处理在某些方面与 C++ 相似，但存在差异，例如 Fortran 的索引从 1 开始，而不是从 0 开始。更重要的是，Fortran 对多维数组有更好的处理，并且更容易操作整个数组。

39.1 静态数组

指定数组大小的首选方式是：

通过**维度**关键字创建数组：

```
real(8), 维度(100) ::x, y
```

大小为 100 的一维数组。

```
整数, 维度(10,20) ::iarr
```

大小为 10×20 的二维数组。

这些数组是静态定义的，并且只存在于它们的程序单元（子程序、函数、模块）内部。

动态分配稍后进行。

这种数组，其大小明确指示，称为 **静态数组** 或 **自动数组**。（参见 39.4 节关于动态数组的内容。）

Fortran 中的数组索引默认为 1 基：

```
integer, 参数 ::N=8
real(4),dimension(N) ::
xdo i=1,N...x(i) ...
```

与 C/C++ 不同。

注意**参数**的使用： 编译时常数 Size 需要被编译器知道。

Unlike C++, Fortran can specify the lower bound explicitly:

```
real,dimension(-1:7) :: x
do i=-1,7
  ... x(i) ...
```

Preferred: use `lbound` and `ubound`
(see also 39.2.1)

Code:

```
1 // /query.F90
2   real,dimension(-1:7) :: array
3   integer :: idx
4     /* ... */
5   do idx=lbound(array,1),ubound(array,1)
6     array(idx) = 1+idx/10.
7     print *,array(idx)
8   end do
```

Output
[arrayf] lubound:

```
 0.899999976
  1.00000000
  1.10000002
  1.20000005
  1.29999995
  1.39999998
  1.50000000
  1.60000002
  1.70000005
```

此类数组，如在 C++ 中，遵循作用域规则：它们在程序或子程序的末尾消失。

39.1.1 初始化

存在多种数组初始化的语法，包括使用隐式 *do*- 循环：

不同的语法：

- Explicit:

```
// /init.F90
  real,dimension(5) :: real5 = [ 1.1, 2.2, 3.3, 4.4, 5.5 ]
```

- Implicit do-loop:

```
// /init.F90
  real5 = [ (1.01*i,i=1,size(real5,1)) ]
```

- Legacy syntax

```
// /init.F90
  real5 = (/ 0.1, 0.2, 0.3, 0.4, 0.5 /)
```

(This is pre-Fortran2003. Slashes were also used for some other deprecated constructs.)

39.1.2 数组片段

Fortran 在处理数组作为整体方面比 C++ 更复杂。首先，你可以将一个数组赋值给另一个数组：

real*8, dimension(10) :: x, y{x}=
y

这显然要求数组具有相同的大小。您可以分配子数组，或 数组片段，只要它们具有相同的形状。这使用冒号语法。

- `A(:)` 以获取所有索引，
- `A(:n)` 以获取到 n 的索引，
- `A(n:)` 以获取 n 及其后的索引。
- `A(m:n)` 索引在 m,...,n 的范围内。

从一个数组段赋值到另一个数组段：

代码：

```
1// /section.
2   real(8), dimension(5) :: x = &
3           [.1d0, .2d0, .3d0, .4d0, .5d
4     /* ... */
5   x(2:5) = x(1:4)
6   print '(f5.3)',x
```

输出 [arrayf] sectionassign:

```
0.100
0.100
0.200
0.300
0.400
```

Note:
格式语法将在后面讨论：浮点数，5 位，小数点后 3 位。

练习 39.1. 编写数组的赋值代码

$x(2:5) = x(1:4)$

使用显式索引循环。你会得到相同的输出吗？为什么？你对数组片段使用的内部机制有什么结论？

上述练习说明了关于数组操作的语义的一个要点：数组语句表现得好像所有输入在存储任何结果之前都汇集在一起。从概念上讲，就好像右边的部分被组装并复制到某些临时位置，然后再写入左边。在实践中，这可能需要大型临时数组（并因减少局部性而影响性能），因此你希望编译器能做些更聪明的事情。然而，练习表明，数组赋值不能简单地转换为简单的循环。

练习 39.2. 你能将那种数组语句的形式化，使得将其简单地转换为循环会改变语义吗？（在编译器术语中，这被称为依赖性。）

数组操作可以比分配给整个数组或其一部分更复杂。例如，您可以使用步长：

`X(a:b:c)` : stride `c`

类似于：`do i=a,b,c`

Copy a contiguous array to a strided subset of another:

Code:

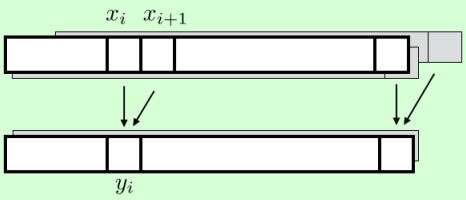
```
1 // /section.F90
2   integer,dimension(5) :: &
3         y = [0,0,0,0,0]
4   integer,dimension(3) :: &
5         z = [3,3,3]
6     /* ... */
7   y(1:5:2) = z(:)
8   print '(i3)',y
```

Output
[arrayf] sectionmg:

```
 3
 0
 3
 0
 3
```

您甚至可以对数组片段进行算术运算，例如将它们相加 .

Exercise 39.3. Code $\forall_i$: $y_i = (x_i + x_{i+1})/2$:



- 首先使用 do 循环；然后
- 通过使用片段，在单个数组赋值语句中完成

使用允许您检查代码正确性的值初始化数组 x。

39.1.3 整数数组作为索引

甚至可以使用存储在整数数组中的一组索引来访问数组中的任意位置。

索引子集：

```
integer,dimension(4) :: i = [2,3,5,7]
real(4),dimension(10) :: x
print *,x(i)
```

39.2 多维

上述数组具有 ‘秩一’。秩定义为访问元素所需的索引数量。数学上这不是秩，而是数组的维度，但这个词已经被占用。我们仍然会使用这个词，例如在谈论数组的第一个和第二个维度时。

一个秩为二的数组，或矩阵，定义如下：

声明和使用括号和逗号（与 C++ 中的 `a[i][j]` 比较）：

```
real(8), 维度(20,30) :: 数组 数组
(i,j)  = 5./2
```

一个有用的函数是 **reshape**。

Reshape：将二维数组转换为一维（或反之）在具有相同元素数量的数组之间转换。

示例：

- 初始化为一维，
- 重塑为 2D

代码：

```
1 // /multi.F90
2   real,dimension(2,2) :: x
3   x = reshape( [ ( 1.*i,i=1,size(x) ) ],
      shape(x) )
4   print *,x
```

Output

```
[arrayf] multi:

   1.00000000           2.00000000
        3.00000000
    4.00000000
```

对于多维数组，我们必须担心它们在内存中的存储方式。是按行存储还是按列存储？在 Fortran 中，后者选择，也称为 列主序 存储，被使用；参见图 39.1。

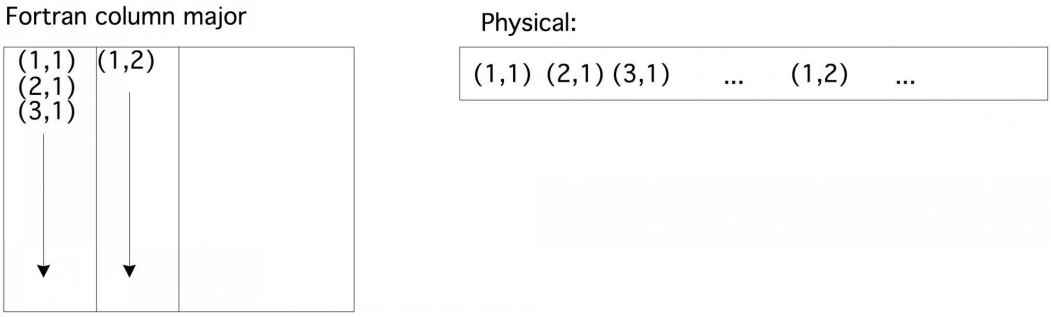


图 39.1：Fortran 中的列主序存储

要按它们在内存中的存储方式遍历元素，您需要以下代码：

```
do col=1,size(A,2)
  do row=1,size(A,1)
    .... A(row,col) .....
  end do
end do
```

这有时被描述为 “ 第一个索引变化最快 ”。有各种与性能相关的理由，为什么这种遍历比交换循环更好。

Exercise 39.4. Can you describe in words how memory elements are access if you would write

```
do row=1,size(A,1)
  do col=1,size(A,2)
    .... A(row,col) .....
```

```
      end do
    end do

?
```

您可以在多维数组中创建区域：您需要在所有维度中指示一个范围。

```
    real(8), 维度(10) ::
    a,ba(1:9) =b(2:10)

or

    逻辑 , 维度 (25,3) ::a 逻辑 , 维度
    (25)    ::ba(:,2) =b
```

您也可以使用步长。

按行填充数组，打印按列：

$$\begin{pmatrix} 1 & 2 & \dots & N \\ N+1 & \dots & & \\ & \dots & & \\ & & & MN \end{pmatrix}$$

代码：

```
1 //printarray.
2   integer, 参数:: M=4,N=5
3   real(4), dimension(M,N) :: rect
4
5   do i=1,M
6       do j=1,N
7           rect(i,j) = count
8           count= count+1
9       end do
10  end
11  do i=1   *,rect
```

Output
[数组 f] 打印数组：

```
 1.00000000           6.00000000
          11.0000000
 16.0000000           2.00000000
          7.00000000
 12.0000000           17.0000000
          3.00000000
 8.00000000           13.0000000
          18.0000000
 4.00000000           9.00000000
          14.0000000
 19.0000000           5.00000000
          10.0000000
 15.0000000           20.0000000
```

39.2.1 查询数组

我们有一个数组的以下属性：

- 边界是每个维度中的下限和上限。例如，在

  ```
  整数 , 维度 (-1:1,-2:2) :: 对称
  ```

  数组 symm 在第一维度的下界为 -1，在第二维度为 -2。函数 `Lbound` 和 `Ubound` 将这些边界作为数组或标量返回：

数组 _ 的 *Lbound(* 对称 *)* 下界和
Ubound(对称 *,2)* 上界在维度 _2 = 中

代码:

```
1 //章节。F90
2   real(8), dimension(2:N)+1 :: Afrom2 =      &
3         [1,2,3,4,5]
4     /* ... */
5   lo = 下界(Afrom1,1)
6   高 = 上界(Afrom1,1)
7   print *,'lo,高'
8   print '(i3,":",f5.3)', &
9         (i,Afrom1(i),i=lo,hi)
```

Output[arrayf] fsection2:

```
             1            5
  1:1.000
  2:2.000
  3:3.000
  4:4.000
  5:5.000
```

- The *extent* 是某个维度中元素的数量，而 **shape** 是 extents 的数组。
- The **size** 是元素的数量，无论是整个数组，还是指定维度。

integer :: x(8), y(5,4)
size(size(x)) size(size(y,2))

39.2.2 调整形状

RESHAPE

```
array = RESHAPE( list,shape )
```

示例:

```
// /shape.F90
  square = reshape( (/(i,i=1,16)/),(/4,4/) )
```

SPREAD

数组 =SPREAD(旧，维度，副本)

39.3 数组传递给子程序

子程序需要知道数组的形状，而不是实际边界:

将数组作为单个符号传递:

Code:

```
1 // /arraypass1d.F90
2     real(8),dimension(:) :: x(N) &
3          = [ (i,i=1,N) ]
4     real(8),dimension(:) :: y(0:N-1) &
5          = [ (i,i=1,N) ]
6
7     sx = arraysum(x)
8     sy = arraysum(y)
9     print '("Sum of one-based
      array:",/,4x,f6.3)', sx
10    print '("Sum of zero-based
      array:",/,4x,f6.3)', sy
```

Output

[arrayf] arraypass1d:

```
Sum of one-based array:
    55.000
Sum of zero-based array:
    55.000
```

Note declaration as `dimension(:)`
actual size is queried

```
// /arraypass1d.F90
  real(8) function arraysum(x)
    implicit none
    real(8),intent(in),dimension(:) :: x
    real(8) :: tmp
    integer i

    tmp = 0.
    do i=1,size(x)
       tmp = tmp+x(i)
    end do
    arraysum = tmp
  end function arraysum
```

子程序内的数组被称为假定形状数组 或 自动数组。

39.4 可分配数组

静态数组在小尺寸下是合适的。然而，在大型尺寸下使用它们有两个主要反对意见。

- 由于尺寸是明确声明的，这使得您的程序不够灵活，需要重新编译才能用不同的问题尺寸运行它。
- 由于它们在所谓的 栈上分配，使它们过大可能会导致 栈溢出。

更好的策略是标明数组的形状，并使用 `allocate` 稍后指定其大小，这显然是以运行时程序参数来表示的。

```
! static:
integer, 参数 ::s=100
real(8), 维度(s) ::xs,ys

! dynamic
```

```
integer :: nreal (8), dimension (:), allocatable ::
xd, ydread *, nallocate (xd (n), yd (n))
```

You can **deallocate** the array when you don't need the space anymore.

如果你有内存不足的风险，给 allocate 语句添加一个 stat=ierror 子句可能是个好主意：

```
integer ::
ierrallocate(x(n), stat=ierr )
if( ierr/=0 ) !报告错误
```

数组是否已分配：

```
Allocated( x ) ! returns logical
```

可分配数组在超出作用域时自动释放。这防止了 C++ 的内存泄漏 问题。

显式释放：

```
deallocate(<x>)
```

39.4.1 返回一个已分配的数组

为了保持主程序简洁和抽象，您可能希望将 allocate 语句委托给一个过程。对于 子程序，您可以将（未分配的）数组作为一个 参数传递。但是，您能从一个 函数返回它吗?

这需要使用 Result 关键字（第 33.2.1 节：

```
// /returnarray.F90 function create_array(n)
result(v) implicit none integer,intent(in) :: n
real,dimension(:),allocatable :: v integer :: i
allocate(v(n)) print *," 分配的形状为: ",shape(v) v
 = [(i+.5,i=1,n) ]end functioncreate_array
```

39.5 数组输出

简单的语句

```
print *,A
```

将输出内存中的 A 元素，按内存顺序；参见章节 39.2。

对于更复杂的方法，需要知道的是每次调用格式语句都会从新的一行开始。因此

```
print '(10f7.3)',A( ...stuff... )
```

will print 10 elements of the array on each line, before starting a new line of output.

指定数组元素的方法是使用隐式 do- 循环；节 32.3 :

```
print '(10f7.3)', (A(i),i=1,size(A))
```

或对于多维情况：

```
do row=1,size(A,2)
  print '(10f7.3)', (A(i,row),i=1,size(A,1))
end do
```

在这个例子中，如果行长度超过 10 个元素怎么办？你不能参数化格式，但指定比数组元素更多的格式没有坏处：

39.6 Operating onanarray

39.6.1 算术运算

对于形状相同的数组：

```
A =B+C
D =D*E
```

(其中乘法是逐元素的)。

39.6.2 内建函数

以下内建函数可用于数组：

- **Abs** 创建逐元素的绝对值矩阵。
- **MaxLoc** 返回最大元素的索引。
- **MinLoc** 返回最小元素的索引。
- **MatMul**返回两个矩阵的矩阵乘积。
- **Dot_Product**返回两个数组的点积。
- **Transpose** 返回矩阵的转置。
- **Cshift** 将数组中的元素旋转。

对数组本身进行归约操作：

- **MaxVal**在数组中找到最大值。
- **MinVal**在数组中找到最小值。
- **Sum** 返回所有元素的和。
- **Product**返回所有元素的乘积。

基于数组的掩码的归约操作：

- **All** 判断掩码是否对所有元素为真
- **Any** 判断掩码是否对任何元素为真
- **Count**统计掩码为真的元素数量

- 函数等**求和** 默认情况下对整个数组进行操作。
- 要限制此类函数到一个子维度，请添加一个关键字参数 **DIM**：

```
s = Sum(A,DIM=1)
```

  其中关键字是可选的。
- 同样地，操作可以限制到掩码：

```
s = 求和(A, 掩码=B)
```

代码：

```
1// /normij.
2   ！在 I 和 J 中求和
3   sums =Sum(A, dim)
4   print*,"行和："
5   print 10,sums
6
7   求和 =  Sum( A,dim=2 )
8   Print*,"Column sums:"
9   和
1010 格式 ( 4(i3,1x) )
```

输出 [数组 f] rowcolsum：

```
矩阵：
  0   1   2   3
  4   5   6   7
  8   9 10 11
 12 13 14 15
行总和：
  6 22 38 54
列总和：
 24 28 32 36
```

练习 39.5. 矩阵的 1- 范数定义为任何一列中绝对值和的最大值：

$$\|A\|_1 = \max_j \sum_i |A_{ij}|$$

而无穷范数定义为最大行和：

$$\|A\|_\infty = \max_i \sum_j |A_{ij}|$$

尽可能使用数组函数来计算这些范数，也就是说，尽量避免使用循环。

For bonus points, write Fortran **Function**s that compute these norms.

Exercise 39.6. Compare implementations of the matrix-matrix product.

1. 编写常规 i,j,k 实现，并将其作为参考。 2. 使用 **DOT_PRODUCT** 函数，它消除了 k 索引。时间变化如何？打印这个结果与参考结果之间的最大绝对距离。 3. 使用 **MATMUL** 函数。同样的问题。 4. 奖励问题：研究 j,k,i 和 i,k,j 变体。用数组切片和单独的数组元素分别编写它们。时间上有差异吗？

优化级别对时间有影响吗？

39.6.3 使用 `where` 进行限制

如果数组操作不应应用于所有元素，您可以使用一个 **where** 语句指定其应用的对象。

```
where ( A<0 ) B = 0
```

Full form:

```
WHERE (逻辑参数) 序列的数组语句

ELSEWHERE
   序列的数组语句
END
WHERE
```

39.6.4 全局条件测试

对所有数组元素进行测试的减少：all

REAL(8),dimension(N,N) :: A LOGICAL:: 正数 , 正数 _row(N), 正数 _col(N) 正数 = ALL(A>0) 正数 _row =ALL(A>0,1) 正数 _col = ALL(A>0,2)

练习 39.7。 使用数组语句（即，没有循环）用 0 到 1 之间的随机数填充二维数组 A。然后分别用 A 中大于 0。5 或小于 0。5 的元素填充两个数组 Abig 和 Asmall：

$$A_{\text{big}}(i,j) = \begin{cases} A(i,j) & \text{if } A(i,j) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

$$A_{\text{small}}(i,j) = \begin{cases} 0 & \text{if } A(i,j) \geq 0.5 \\ A(i,j) & \text{otherwise} \end{cases}$$

使用更多的数组语句，添加 Abig 和 Asmall，并测试它们的和是否足够接近 A。

类似于所有，有一个函数 **any**，用于测试是否任何数组元素满足测试条件。

```
if ( Any( Abs(A-B)>
```

39.7 数组操作

39.7.1 无循环的循环

除了普通的 do 循环之外，Fortran 还有一些机制可以节省你的输入，或在某些情况下更高效。

39.7.1.1 切片

如果你的循环从另一个数组赋值到数组，你可以使用区间表示法：

```
a(:) =b(:)c(1:n) =
d(2:n+1)
```

39.7.1.2 *'forall'* 关键字

The `forall`关键字也指示数组赋值：

```
forall <>(i=1:
n)a(i) =
b(i)c(i) =d(i
+1)end forall
```

你可以看出这是针对数组的，因为循环索引必须是每个赋值语句左侧的一部分。

如果你将 `forall` 应用于带有循环依赖的语句，会发生什么？首先考虑传统的循环

forallshift.F90

```
A = [1,2,3,4,5]
do i=1,4
    A(i+1) =A(i)
end do
print '(5(i2x))',A
```

forallshift.F90

```
A = [1,2,3,4,5]
do i=4,1,-1
    A(i+1) =A(i)
end do
print '(5(i2x))',
```

你能预测输出吗？现在考虑以下内容：

代码：

forallshift.F90

```
2  A = [1,2,3,4,5]
3  forall (i=1:4)
4      A(i+1) =A(i)
5  end forall
6  print '(5(i2x))',A
```

输出

[数组 f] forallf:

```
 1  1  2  3  4
```

代码：

forallshift.F90

```
2  A = [1,2,3,4,5]
3  do i=4,1,-1
4      A(i+1) =A(i)
5  end do
6  print '(5(i2x))',A
```

输出

[数组] forallb:

```
 1  1  2  3  4
```

这告诉你有关执行什么？

换句话说，这种机制容易引起误解，因此现在已弃用。它不是一个并行循环！为此，以下机制是首选。

39.7.1.3 执行并发

The *do concurrent* 是一个真正的 do- 循环。使用 `concurrent` 关键字，用户指定循环的迭代是独立的，因此可能可以并行执行：

```
do concurrent (i=1:n) a(i)
b(i) c(i) d(i+1) end do
```

(不要使用 `for` **所有**)

39.7.2 无依赖的循环

这里有一些使用上述机制进行简单数组复制的示例。

```
// /block.F90
  do i=2,n
      计数 (i)  = 2* 计数 (i-1)
  end do

原始          1   2   3   4   5   6   7   8   9  10
递归          0   2   4   6   8  10  12  14  16  18

//  / 块 .F90
  计数的 (2:n)   = 2*C计数 (1:n-1)

原始          1   2   3   4   5   6   7   8   9  10
章节          0   2   4   6   8  10  12  14  16  18

//  / block . F90
  forall (i=2:n)
      计数 (i) =     2*counting(i-1)
  end forall

Original      1   2   3   4   5   6   7   8   9  10
Forall        0   2   4   6   8  10  12  14  16  18

//  / block . F90
  do 并发 (i=2:n)
      计数 (i)  = 2* 计数 (i-1)
  end do

原始          1   2   3   4   5   6   7   8   9  10
并发          0   2   4   6   8  10  12  14  16  18
```

练习 39.8. 创建长度为 2N 的数组 A 、 C，以及长度为 N 的数组 B。现在实 t

$$B_i = (A_{2i} + A_{2i+1})/2, \quad i = 1, \ldots N$$

and

$$C_i = A_{i/2}, \quad i = 1, \ldots 2N$$

使用所有四种机制。确保每次都能得到相同的结果。

39.7.3 具有依赖关系的循环

对于循环的并行执行，所有迭代必须相互独立。如果循环具有 递归，则情况并非如此，在这种情况下，“do concurrent” 机制不适用。以下是上述四种结构，但应用于一个具有依赖关系的循环。

```
// /recur.F90
  do i=2,n
     counting(i) = 2*counting(i-1)
  end do

Original   1   2   3   4   5   6   7   8   9  10
Recursiv   1   2   4   8  16  32  64 128 256 512
```

这个切片版本的：

```
// /recur.F90 计数 (2:n) = 2* 计数
(1:n-1)

原始       1   2   3   4   5   6   7   8   9  10
章节       1   2   4   6   8  10  12  14  16  18
```

就好像右侧保存在临时数组中，然后分配给左侧。

使用 ‘forall’ 相当于切片：

```
// /recur。F90forall (i=2:n) 计数
(i)  = 2* 计数 (i-1)end forall


原始       1   2   3   4   5   6   7   8   9  10
forall     1   2   4   6   8  10  12  14  16  18
```

另一方面，“do concurrent” 不使用临时变量，因此它更像是顺序版本：

```
// /recur.F90
  do concurrent (i=2:n)
     counting(i) = 2*counting(i-1)
  end do

Original   1   2   3   4   5   6   7   8   9  10
Concurrent 1   2   4   8  16  32  64 128 256 512
```

请注意，结果不必等于顺序执行：编译器可以自由地以它认为合适的方式重新排列迭代。

39.8 复习问题

练习 39.9. 给定以下声明，并假设所有数组都已正确初始化：

```
real :: x, real, dimension(10) :: a,
b, real, dimension(10,10) :: c, d
```

评论以下行：它们是否合法，如果是，它们做什么？

1. `a = b`
2. `a = x`
3. `a(1:10) = c(1:10)`

你会如何：

1. 将 c 的第二行设置为 b？ 2. 将 c 的第二行设置为 b 的元素，从后到前？

第 40 章

指针

C/C++ 中的指针基于内存地址；而 Fortran 指针则更为抽象。

40.1 基本指针操作

Fortran 指针有点像 C 或 C++ 指针，但在很多方面也有不同。

- 像C 中的 ‘星’ 指针，而不像 C++ 中的 ‘智能’ 指针，它们可以指向任何东西。
- 与C 指针不同，你必须声明一个对象可以被指向。
- 与C/C++ 中的任何指针类型都不同，但与 C++ 引用类似，它们充当某种别名：没有显式的解引用。

We will explore all this in detail.

Fortran 指针类似于 ‘别名’：使用指针变量通常与使用它指向的实体相同。与实际使用变量的区别在于，你可以决定指针指向哪个变量。

Fortran 指针通常自动 解引用：如果你打印一个指针，你会打印它引用的变量，而不是指针的某种表示形式。

代码：

40.1 基本指针操作

```
real, target::x
real,pointer :: point_at_real

x = 1.2_
point_at_real => x
print *,point_at_real
```

Output

指针 f] 基本 p:

1.20000005

指针在指定类型的变量声明中定义，具有 指针 属性。示例：定义

```
real, 指针 ::point_at_real
```

声明了一个可以指向 real 变量的指针。如果没有进一步指定，该指针目前不指向任何东西，因此它是未定义的。

- 您必须声明一个变量是可以指向的：

```
real,target :: x
```

- 声明一个指针：

```
real, 指针 :: 指向 _at_real
```

- 使用 => 符号设置指针（新！注意！）：

```
指向 __real =>x
```

现在使用 point_at_real 与使用 x 是相同的。

```
print*,point_at_real ！将打印 x 的值
```

指针不能指向任何东西：被指向的对象需要被声明为 target

```
real,target :: x
```

并且你使用 => 运算符来让指针指向一个目标：

```
point_ 指向 _ 的实 => x
```

如果你使用指针，例如用来打印它

```
print *,point_at_real
```

它表现得好像你在使用它指向的值 ts at.

代码：

```
realp.
real,</target ::x,y
real,po n er :: that_real

x = 1.2
y 那个实 =>x_
print *, 那个 _ 实
 那个 _ 实 =>y
print *, 那个 _ 实
y= x
print *,that_real
```

输出

[指针 f] 实数 p：

```
1.20000005
2.40000010
1.20000005
```

1. that_real points at x, so the value of x is printed.
2. p y, p 2. y 的值被改变，并且由于 that_real 仍然指向 y，这个改变后的值是被打印。

40.2 组合指针

如果你让一个指针指向另一个指针会发生什么？C/C++ 中的指针到指针的概念不存在：相反，两个指向同一事物的指针。

如果你有两个指针

```
real,pointer :: point_at_real,also_point
```

你可以让其中一个的目标也成为另一个的目标：

```
point_ 指向 _ 实数 => x 也指向 =>
点 _ 指向 _ 实数
```

请注意，第二个指针也被分配了 => 符号。这不是一个指向指针的指针：它将右侧的目标分配给左侧的目标。

What happens if you want to write `p2=>p1`
but you write `p2=p1`?
The second one is legal, but has different meaning:

分配底层变量：

```
//assignequals.F90l t t
rea , arge  :: x,y
real,</pointer>::p1,p2

x = 1.2
p1 => x
p2 => y
p1 = p  ! same as y=x
print *,p2 ! same as print y
```

崩溃因为 `p2` 指针未关联：

```
//assignwrong.
real, target:: x
real,</pointer>::p1,p2

x = 1.2
p1 => x
p2 = p1
print *,p2
```

Exercise40.1. 编写一个例程，该例程接受一个数组和指针，并在返回时使该指针指向数组中的最大元素：

代码：

```
1//arpointf.
real,dimension(10),target :: array &
     = [1.1, 2.2, 3.3, 4.4, 5.5,          &
        9.9, 8.8, 7.7, 6.6, 0.0]
real, pointer:: biggest_element

print '(10f5.2)',array
call setpointer(array,biggest_element)
print*,"Biggest element
  是", 最大的 _ 元素
print *,"checking pointerhood:",&
  关联 (biggest_ 元素)
最大的 _ 元素  = 0
print '(10f5.2)',array
```

输出 [指针 f] arpointf:

```
1.10 2.20 3.30 4.40 5.50 9.90
  8.80 7.70 6.60 0.00<i>
B ggest element is   9.89999962
检查指针状态 :T
1.10 2.20 3.30 4.40 5.50 0.00
  8.80 7.70 6.60 0.00
```

Y你可以使用thi些常见的代码片段。tf F90 i th it

40.3 指针状态

指针可以有三种状态：

1. 指针在被首次创建时是未定义的，2. 如果显式设置，它可以为 null，3. 或者它可以被关联，如果它已经指向某个东西。

作为一种常识策略，不要担心未定义状态：在 40.5 节的示例中 40.5 中的指针很快就会被设置为 null。

- **Nullify**: zero a pointer
- **Associated**: test whether assigned

代码：

```
1  //  /ptptr.F90
2
3  real,pointer :: realp
4
5  print *,"Pointer starts as not set"
6  if (.not.associated(realp)) &
7     print*,"指针未关联"1 2
8  x =  .
9  print*,"设置指针"
10 realp => x
11 if (associated(realp)) &
12    print*,"指针指向"
13 print*,"未设置指针"
14 nullify<0xE3><0x80><0x82><0xE3><0x80><0x83>realp
15 if (.not.associated(realp)) &
16    print*,"指针未关联"
```

输出

指针 f] 状态 p:

Pointer starts as not set
指针未关联
设置指针
指针指向
取消设置指针
指针未关联

你也可以专门测试

- 关联 (p,x): 指针是否与变量关联，或
- 关联 (p1,p2): 两个指针是否关联到同一个目标。

如果你想让指针指向某个东西，但不需要为这个东西定义变量：

Code:

```
1// /allocptr.F90 2 Real,pointer :: x_
ptr,y_ptr 3 allocate(x_ptr) 4 y_ptr =>
x_ptr 5 x_ptr = 66 print *,y_ptr
```

Output

[pointerf] allocptr:

6.00000000

(比较使 _ 在 C++ 中共享)

40.4 指针和数组

您可以设置一个指针指向数组元素或整个数组。

```
real<sup>(8)</sup>,dimension<sup>(10)</sup>,target :: array_
ptr<sup>real</sup><sup>(8)</sup>,pointer :: element_
ptrptr<sup>real</sup><sup>(8)</sup>,pointer,dimension<sup>(:)</sup> ::
array_ptr element_ptr<sup>array</sup>=>
ptr<sup>array</sup><sup>(2)</sup>array_ptr<sup>array</sup>=>
ptr<sup>array</sup>
```

更令人惊讶的是，你可以将指针设置为数组的一部分：

```
数组 _ 指针 => 数组 <sub>2:)</sub> 数组 _ 指针 => 数组
<sub>1:</sub><sub>size</sub><sub>(</sub> 数组
<sub>):2)</sub>
```

如果你想知道，这不会创建临时数组，但编译器会向指针添加描述，以自动将代码转换为变长索引。

You can use the **allocate** statement for pointers to arrays:

```
整数 , 指针 , 维度 (:):: 数组 _ 点
分配 ( 数组 _ 指针 (100))
```

This is automatically deallocated when control leaves the scope. No memory leaks.

作为指向数组部分的指针的一个更有趣的例子，让我们考虑求平均操作

$$x_{i,j} = (x_{i+1,j} + x_{i-1,j} + x_{i,j+1} + x_{i,j-1})/4.$$

我们需要指向内部及其四个偏移量的指针：

```
// /interior.F90
  real(4),target,allocatable,dimension(:,:) :: grid
  real(4),pointer,dimension(:,:) :: interior,left,right,up,down

  分配 ( grid(N,N) ) /* ...*/ 内部 =>
  grid(2:N-1,2:N-1) 上 =>
  grid(1:N-2,2:N-1) 下 =>
  grid(3:N,2:N-1) 左 =>
  grid(2:N-1,1:N-3) 右 =>
  grid(2:N-1,3:N)
```

求平均操作是一个数组语句：

```
// / 内部 .F90 内部 = ( 上 + 下 + 左 + 右 ) /4
```

40.5 示例：链表

有关链表的图片，请参见第 66.1.2 节。

- 线性数据结构
- 在插入 / 删除方面比数组更灵活

- . . . 但访问速度较慢

使用指针的一个标准示例是 链表。这是一种比数组更灵活的动态一维结构。动态扩展数组需要重新分配，而在链表中可以插入元素。

练习 40.2. 使用链表可能比使用数组更灵活。另一方面，在链表中访问元素的成本更高，无论是绝对值还是按列表大小的数量级。

使这个论点精确化。

40.5.1 类型定义

列表基于一个简单的数据结构 —— 节点，该节点包含一个值字段和一个指向另一个节点的指针。列表数据结构本身只包含一个指向列表中第一个节点的指针。

- 节点：值字段，以及指向下一个节点的指针。
- 列表：指向头节点的指针。

```
// /listfappendalloc.F90
  类型 节点
     integer :: value
     类型 ( 节点 ), 指针 :: 下一个
  end type node

  类型 列表
     type(node),pointer :: head
  end type list
```

举例来说，我们创建一个按大小排序的动态整数列表。为了保持排序状态，我们需要根据需要添加和插入节点。

Our main program will create three nodes, and append them to the end of the list:

Code:

```
1// /listfappendalloc.
2   整数 , 参数 :: listsize=7
3   类型 ( 列表 ) :: 的 _ 列表
4   整数 , 维度 ( 列表大小 ):: 输入                    =
      &
5         [ 62, 75, 51, 12, 14, 15, 16 ]
6   整数 :: 输入 , 输入 _ 值
7
8   nullify(the_list%head)
9   do input=1,listsize
10     input_value = inputs(input)
11     call attach(the_list,input_value)
12  end do
```

输出

[指针 f] 列表追加:

```
列表 :  [ 62,75,51,12,14,15,16,]
```

40.5.2 在末尾连接阳极

首先，我们编写一个 `attach` 例程，它接受一个节点指针并将其连接到列表末尾，而不进行任何排序。

```
// /listfappendalloc.F90subroutine
attach( the_list,new_value)implicit
none! 参数type(list),intent(inout) ::the_
listinteger,intent(in) ::new_value
```

我们区分两种情况：列表为空时和列表不为空时。最初，列表为空，这意味着 “head” 指针未关联。第一次将元素添加到列表时，我们将节点分配为列表的头部：

```
//listfappendalloc.F90 ! 如果列表没有头节点，则连接新节点 if
(.not.associated(the_list%head)) then allocate( the_list%head ) the list%
head%value= new value_ _else call node_
attach(the_list%head,new_value ) end if
```

新元素已附加在末尾。

```
//listfappendalloc.F90 递归子程序 node_attach(the_node,new_value ) /
* ... */ if ( .not.associated(the_node%next) ) then allocate( the_node%
next ) the node%next%value= new value_ _
else call node_attach(the_node%next,new_value ) end if
```

练习 40.3. 将附加元素的递归代码转换为迭代版本，即使用一个 `while` 循环向下遍历列表直到末尾。

你可以将整个东西放在列表头部的 `attach` 例程中。

40.5.3 在排序顺序中插入阳极

如果我们想保持列表排序，在许多情况下需要在列表末尾之前的位置插入新节点。这意味着，我们不是遍历到末尾，而是遍历到新节点需要附加的第一个节点。

几乎和之前一样，但现在保持列表排序：

Code:

```
1 /// /listfinsertalloc.
2     以 C++17/Fortran2008 进行科学编程的艺术，第 3 卷 40.5.3 在 sortorder 中插入阳极
3         in_value= inputs(
4         调用 插入（the_ 列表，在 _ 值）
5         调用 print （_ 列表）
6     end do
```

输出

```
[指针 f] 列表插入:
List:  [  62  ]
List:  [  62  75  ]
List:  [  51  62  75  ]
List:  [  12  51  62  75  ]
List:  [  12  14  51  62  75  ]
List:  [  12  14  15  51  62  75
    ]
List:  [  12  14  15  16 51    62
    75  ]
```

练习 40.4。将 `attach` 例程复制到 `insert`，并修改它，以便插入值时保持列表有序。

您可以根据代码仓库中的文件 *listfappendalloc.F90* 进行参考

练习 40.5. 修改您的代码，从练习 40.4 开始，以便新的节点不在主程序中分配。

相反，仅传递整数参数，并在需要时使用 **`allocate`** 来创建新节点。 **缺失片段 flistinsertalloc**

Exercise 40.6. Write a **`print`** function for the linked list.
For the simplest solution, print each element on a line by itself.

更高级的：使用 **`Write`** 函数和 **`advance`** 关键字 :

write(*, '(i1",")', , advance="no") current%value

练习 40.7. 为链表编写一个 `length` 函数。尝试使用循环和递归两种方法进行测试。

第 41 章

输入 / 输出

41.1 Types of I/O

Fortran 可以处理 ASCII 格式的输入 / 输出，这被称为格式化 I/O，以及二进制，或非格式化 I/O。格式化 I/O 可以使用默认格式，但你可以在 I/O 语句本身中指定详细的格式说明，或在单独的**格式**语句中指定。

Fortran I/O 可以描述为 < 样式 id='1'> 表导向 I/O</ 样式 >：输入和输出命令都采用一个项目列表，可能带有格式说明。

- <样式 id='2'> 打印 </ 样式 > 简单输出到终端
- <样式 id='2'> 写入 </ 样式 > 输出到终端或文件（ ‘ 单元 ’ ）
- **从**终端或文件 <code> 读取 </code> 输入
- **打开** , **关闭**文件和流
- **格式化**可在多个语句中使用的格式规范

- 格式化输出：ASCII。这适用于报告，但不适用于数值数据存储。
- 未格式化输出：二进制输出。适用于进一步处理输出数据。
- 注意：二进制数据是依赖于机器的。使用 *hdf5* 进行可移植的二进制。

41.2 打印到终端

输出数据的 simplest 命令是 **print**。

```
print *,"The result is",result
```

在其最简单形式中，您使用星号来表示您不关心格式；在逗号之后，您可以有任意数量的用逗号分隔的字符串和变量。

41.2.1 在一行上打印

The statement

```
print *,item1,item2,item3
```

将打印出所有项目，并在一行内显示，直到行长度允许为止。

使用隐式 *do* 循环进行参数化打印：

```
print *,( i*i,i=1,n)
```

所有值将打印在同一行上。

41.2.2 打印数组

如果你打印一个数组变量，所有它的元素将被打印，如果数组是多维的，则按 列主序 排序。

您也可以通过使用隐式 *do* 循环来控制数组的打印：

```
print *,( A(i),i=1,n)
```

41.3 格式化 I/O

默认格式使用相当多的位置来表示可能很小的数字。为了明确指示格式，例如限制用于数字的位置数，或实数的整数和小数部分，您可以使用格式字符串。

For i$_{\text{nstance}}$ ，您可以使用一个字母后跟一个数字来控制格式 matting width:

代码：

```
1// /format.
2   i = 56
3   print *,i
4   print '(i4)',i
5   i
6   print '(i1)',i
7   i = i*i
8   print '("fit <",i0,"> ted")',i
```

输出 [iof] i4:

```
          56
  56
56
*
拟合 <3136>ted
```

(在最后一行中，格式说明符的宽度不足以容纳数据，因此使用了一个 星号 作为输出。)

让我们半系统地来处理这个问题。

41.3.1 格式字母

41.3.1.1 Integers

整数可以用 *in* 设置。

- 如果 $n > 0$，则使用该数字右对齐的那么多位置；除外
- 如果数字不适合 n 位置，则渲染为星号。
- 要使用精确所需的位置数，请使用 *i0*。

代码:

```
1// /format.
2   i = 56
3   print *,i
4   print '(i4)', i
5   print '(i2)', i
6   print '(i1)', i
7   i = i*i
8   print "("fit <",i0,"ted)" " ,  ",i"
```

输出
[iof] i4:

```
          56
  56
56
*
fit <3136> ted
```

41.3.1.2 Strings

字符串有两种处理方式:

1. 它们可以是格式字符串的一部分: `print '(i2,"--",i2)',m,n`2. 它们可以用 `an` 指定符: `print '(a5,2)',somestring,someint`3. 要使用精确的位置数, 使用 `aprint'(a,i0,a)', str1,int2,str3`

41.3.1.3 浮点数

- '`fm.n`' 指定一个实数的定点表示法, 有 m 总位置 (包括小数点) 和 n 小数部分的位数。
- '`em.n`' 指数表示法。

41.3.1.4 其他

x : 一个空格 x5 : 五
个空格 b : 二进制 o :
八进制 z : 十六进制

41.3.2 重复和分组

If y您想要显示相同类型的项, 可以使用 repeat count:

代码:

```
1// / 格式。F902 i = 12;
j  = 343  print'
(2i4)',i,j4  print'
(2i2)',i,j
```

Output
[iof] ii:

```
  12  34
1234
```

您可以混合不同类型的变量，以及字面量字符串，通过用逗号分隔它们。并且您可以使用括号将它们分组，并在这些组上再次指定重复次数：

Code:

```
1 // /format.F90
2   i = 23; j = 45; k = 67
3   print '(i2,1x,i2)',i,j
4   print '("Numbers:",3(1x,i2,"."))',
      i,j,k
```

Output

```
[iof] ij:
23 45
Numbers: 23. 45. 67.
```

在单个指定符前面放置一个数字表示它将被重复。

如果数据占用的位置比格式指定符允许的位置多，则会打印出一串星号：

Code:

```
1 //  / 星号 .F90
2   do ipower=1,5
3      print '(i3)',number
4      number  = 10*number
5   end do
```

Output

```
[fio] asterisk:
  1
 10
100
***
***
```

如果你发现自己多次使用相同的格式，你可以给它一个标签 :

```
      print 10,"result:",
    x,y,z10 格式 ('(a6,3f5.3)')
```

https://www.obliquity.com/computer/c++/format.html

```
print'( 3i4 )',, i1,, i2,, i3print'( 3(i2,":",f7.4))',, i1,, r1,,
i2,, r2,, i3,, r2
```

- If *abc* 是一个格式字符串，那么 10(*abc*) 给出 10 次重复。没有换行。
- 如果数据量超过格式中指定的数量，则格式会在新的打印语句中重复使用。这会导致换行。

- The / (斜杠) 指定符会导致换行。
- 输出行可能有 80 个字符的限制。

Exercise 41.1. Use formatted I/O to print the number $0 \cdots 99$ as follows:

```
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40 41
42 43 44 45 46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63 64 65 66 67
68 69 70 71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89
```

```
 90 91 92 93 94 95 96 97 98 99
```

41.4 文件和流 I/O

如果您想将输出发送到终端屏幕以外的任何地方，您需要使用 写 语句，该语句如下：

```
write (unit,format) data
```

其中 格式 和 数据 如上所述。新元素是 单位，它是一个表示输出设备的数值指示符，例如文件。

41.4.1 单位

对于文件 I/O，您写入一个单位编号，该编号通过一个打开 语句与文件关联。

文件处理完成后，你关闭 它。

```
Open(11)
```

这将生成一个文件，其名称通常为 fort11。要为其指定自定义名称：

```
Open(11,FILE="filename")
```

还有许多其他错误处理选项、新旧文件等。

之后，一个写入 语句可以引用该单元：

```
写入 (11,fmt) 数据
```

再次提供错误处理等选项。

41.4.2 其他写入选项

默认情况下，每个 `Write` 语句，像 `Print` 语句一样，会写入到新的一行（在 Fortran 术语中称为 ‘记录’ ）。要阻止这种情况，

```
write(unit, fmt, ADVANCE="no") 数据
```

不会发出换行。

41.5 转换到 / 从字符串

前面，你看到了在终端和文件输出上下文中的 `Print` 、 `Write` 和 `Read` 命令。然而，对于 `Write` 和 `Read` ，字符串处理有一种第二种用法，即字符串和数值之间的转换。

41.5.0.1 字符串转换为数值

将字符串转换为数值量，执行读取操作：

```
Read( some_string, some_format ) bunch, of, quantities
```

示例：

代码：

```
// /读写。F90
字符(len=8):: 日期
整数:: 年，月，日
日期="20221027"
读取( 日期,'(i4,i2,i2)' ) &
    年，月，日
  /* ... */
打印*,"日期:",日期
print '( "Year=",i4,", mo=",i2,",
  天=",i2 )', &
    年，月，天
```

输出
[stringf] 日期：

```
日期:20221027
年=2022,月=10, 日=27
```

41.5.0.2 数值转字符串

相反地，要从某些量构造一个字符串，你需要执行一个写操作：

```
写( 某些_字符串，某些_格式) 一串, 的, 量                    s
```

代码：

```
// /读写.F90
字符(len=10):: 长日期/      /
    * ... *
写入( 长日期,&
      '(i4,"/",i2,"/",i2)'      &
      ) 年，月，日
打印*,"长日期:",长日期
```

输出
[stringf] 斜杠：

```
长日期:2022/10/27
```

41.6 无格式输出

到目前为止，我们研究了 ASCII 输出，这对于人类来说看起来很美观，但不是将数据传递给另一个程序的正确媒介。

- ASCII 输出需要耗时的转换。
- ASCII 渲染会导致精度损失。

因此，如果你希望输出稍后由程序读取的数据，最好使用二进制输出或无格式输出，有时也称为原始输出。

Indicated by lack of format specification:

写（*）**数据**

Note: may not be portable between machines.

41.7 打印到打印机

在 Fortran2003 之前的 Fortran 标准中，输出行的第 1 列具有特殊含义，对应于 <code>line printer</code> 牵引控制。众所周知，此处有一个字符会移动到新的一页。虽然这个特性已经从标准中移除，但你可能仍然会在输出中看到黑色第一列，而无需指定这种情况。

41. 输入 / 输出

第 42 章

遗留话题

42.1 接口

如果您想在主程序中使用一个过程，编译器需要知道该过程的签名：有多少参数、什么类型以及什么意图。您已经看到了如何使用 `contains` 子句来为此目的使用，如果过程和主程序位于同一个文件中。

如果过程位于一个单独的文件中，编译器不会在一个地方看到定义和使用。为了允许编译器进行正确的使用检查，我们可以使用一个 **接口** 块。这放置在调用位置，声明过程的签名。

Main program:

```
// /interface .F90
  接口
     函数 f(x,y)
        real*8 :: f
        real*8, intent(in) :: x, y
     end function f
  end interface

  real*8 :: in1=1.5, in2=2.6, result

  result = f(in1,in2)
```

Procedure:

```
// /interfunc.F90
function f(x, y)
  implicit none
  real*8 ::f
  real*8 inten    t(in) :: x,y
```

The **接口** 块不是必需的（有一个更旧的 **外部** 机制用于函数），但推荐使用。如果函数接受函数参数，则需要使用它。

42.1.1 多态

可以使用 **接口** 块来定义一个通用函数：

```
接口 f 函数 f1(.....) 函数
f2(.....)endinterface
f
```

其中 f1 、 f2 是可以通过它们的参数类型来区分的函数。通用函数 f 然后根据调用它的参数类型变成 f1 或 f2。

42.2 随机数

在本节中，我们简要讨论 Fortran 随机数生成器。基本机制是通过库子程序 random_ 数，它有一个类型为 REAL 的参数 INTENT(OUT)：

```
real(4):: randomfractioncall
random_数(randomfraction)
```

结果是来自 $[0,1)$ 上均匀分布的随机数 .

设置随机种子 稍微复杂。存储种子所需的存储量可能依赖于处理器和实现，因此子程序 random_ 种子 可以有三种类型的命名参数，任何一次只能指定其中之一。关键字可以是：

- SIZE for 查询 种子的尺寸；
- *PUT* for 设置 种子；和
- *GET* for 查询 种子。

设置 种子的典型片段会是：

```
整数 :: 种子大小整数 , 维度 (:), 可分配 :: 种子


调用随机数 <code>_</code> 种子 <code>seed</code>(<code>size</code><code>=</code><code>seedsize</code>)

allocate(seed(seedsize))seed(:) = ! 你
的整数种子在这里 call random_seed(put=
seed)
```

42.3 计时

计时是通过系统 _ 时钟 例程完成的。

- 该调用返回一个整数，表示时钟滴答数。
- 要转换为秒，它还可以告诉你每秒有多少滴答数：它的 计时器分辨率。

```
整数 :: clockrate,clock_start,clock_end

调用系统 _ 时钟 ( 计数 _ 频率 = 时钟频率 )
打印 *," 每秒滴答数 :", 时钟频率

调用系统 _ 时钟 ( 时钟 _ 开始 )
! 代码
调用系统 _ 时钟 ( 时钟 _ 结束 )
打印 *," 时间 :", ( 时钟      _end-clock_start)/REAL(clockrate)
```

42.4 Fortran standards

- 第一个 Fortran 标准只是简单地称为 ‘Fortran’。
- Fortran4 是一个流行的下一个标准。
- Fortran66 也很常见。它非常基于 goto 语句，因为没有块结构。
- Fortran77 是一种更结构化的语言，包含现代 do 和 if 块语句。然而，内存管理仍然是完全静态的，递归过程不存在。

- Fortran88 并未及时出现以证明其名称，即使称之为 Fortran8X 也无济于事：它最终成为 Fortran90。该标准引入了

  – 模块，这使得公共块不再需要。– the 隐式无指定 ,–

  动态内存分配。– 递归。

Fortran95 是对 Fortran90 的澄清。

- Fortran2003 （及其改进版 Fortran2008 ）引入了：– 面向对象。–

这也是 Co-array Fortran (CAF) 成为语言一部分的时候。

- Fortran2018 引入了子模块、更多并行性和更多 C 互操作性。

42. 剩余主题

第 43 章

Fortran 复习问题

43.1 Fortran 与 C++

Exercise 43.1. For each of C++, Fortran, Python:

- 给出一个该语言适合的应用或应用领域的例子，并
- 给出一个该语言不太适合的应用或应用领域的例子。

43.2 基础

Exercise 43.2.

- What does the **参数**keyword do? Give an example where you would use it.
- Why would you use a **模块** ?
- Intent 关键字的作用是什么?

43.3 数组

练习 43.3. You are looking at historical temperature data, say a table of the high and low tem- perature at January 1st of every year between 1920 and now, so that is 100 years.

Your program accepts data as follows:

```
整数 : : 年份 ， 高 ， 低

!! 代码省略

读取 *, 年份 , 高 , 低
```

其中温度值四舍五入到最接近的度数（摄氏度或华氏度由你决定。）

考虑两种场景。对于这两种场景，分别给出以下代码行：1. 用于存储数据的数组，2. 将值插入数组的语句。

- Store the raw temperature data.

- 假设您想了解某些高温 / 低温发生的频率。例如，‘多少年有 32F /0 C 的高温’。

43.4 子程序

Exercise 43.4. Write the missing procedure `pos_input` that

- 从用户读取一个数字
- 返回它
- 并返回该数字是否为正

以使此代码正常工作：

代码：

```
1// /looppos.
2 program looppos 3

    implicit none
4   real(4) :: userinput
5   do while (pos_input(userinput))
6      print &
7        ’("Positive input:",f7.3)’,
8        userinput
9   end do
10  print &
11       ’("输入非正：",f7.3)’,&
12       用户输入
13    /* ... */
14 end program looppos
```

输出 [f f]

```
  unc    looppos:

使用以下
     输入：

5
1
-7.3

/bin/sh:./looppos: 不存在
     文件或目录
make[2]: *** run_ looppos]
     错误 127
```

给函数参数赋予正确的意图指令。

提示：pos_input 是 SUBROUTINE 还是 FUNCTION？如果是后者，函数结果的类型是什么？否则它有多少参数？变量 user_input 如何获取其值？那么函数的参数类型是什么？

第四部分

练习和项目

第 44 章

项目提交风格指南

计算的目的在于洞察，而非数字。*(* 理查德 · 汉明 *)*

你的项目文档与代码同等重要。以下是一些撰写优秀文档的常识性指南。然而，并非所有部分都适用于你的项目。请运用你的良好判断力。

44.1 一般方法

一般而言，将编程视为一门实验科学，将你的文档视为你已完成的某些测试的报告：解释你正在解决的问题、你的策略、你的结果。

请以 PDF 格式提交报告（ Word 和文本文档不可接受），该报告应是由文本处理程序生成的（最好使用 LaTEX。有关教程，请参阅教程书籍 [9], 第 -15 节。

44.2 Style

您的报告应符合正确的英语规范。

- 正确拼写和语法是理所当然的。
- 使用完整的句子。
- 尽量避免使用贬损性或不宜使用的措辞。 *Google* 开发者文档风格指南 [13] 是一个很好的资源。

44.3 你的报告结构

将这个项目报告视为练习撰写科学文章的机会 .

从显而易见的事情开始。

- 你的报告应该有一个标题。不是 ‘ 项目 ’ ，而是类似 ‘Chronosyn- clastic Enfundibula 的模拟 ’ 这样的标题。
- 作者和联系信息。这因情况而异。这里的信息是：你的名字、 EID 、 TACC 用户名和电子邮件。
- 引言部分，非常高级别：问题是什么，你做了什么，你发现了什么。

- 结论：你的发现意味着什么，有哪些局限性，未来扩展的机会。
- 参考文献。

44.3.1 简介

你的文档的读者不需要熟悉项目描述，甚至不需要了解它所解决的问题。指出问题是什么，如果合适，给出理论背景，可能勾勒一个历史背景，以及用全球术语描述你如何着手解决问题，以及你的发现简短陈述。

44.3.2 详细介绍

See section 44.5 below.

44.3.3 讨论 和 总结

将详细说明与最后的讨论分开：你需要有一个简短的最终部分来总结你的工作和发现。你也可以讨论你工作可能的扩展，以涵盖未覆盖的情况。

44.4 实验

你不应该期望你的程序运行一次并给你一个关于你研究问题的最终答案。
问问自己：哪些参数可以变化，然后变化它们！这允许你生成图表或多维图形。

如果你改变一个参数，想想你使用的粒度是什么。十个数据点足够吗，还是使用 10,000 你能获得洞察力?
首先：计算机非常快，每秒可以进行十亿次操作。所以不要害怕使用长时间运行的程序。你的程序不是一个计算器，按一下按钮就能立即得到答案：你应该预期程序运行需要几秒钟，也许是几分钟。

44.5 详细展示你的工作

你的工作的详细展示是代码片段、表格、图表以及这些的描述的组合。

44.5.1 数值结果的展示

你可以将结果作为图表 / 图形或表格展示。选择取决于因素，例如你有多少数据点，以及是否在图形中有一个明显的关系。
图表可以通过多种方式生成。如果你能弄清楚 LaTeX `tikz` 包，那真了不起，但 Matlab 或 Excel 也是可以接受的。不过不要截图。
为你的图表 / 表格编号，并在文中引用编号。给图表一个清晰的标签，并标注坐标轴。

44.5.2 代码

你的报告应以全局方式描述你开发的算法，并且你应该包含相关的代码片段。如果你想要包含完整的列表，将其移至附录：文本中的代码片段仅应用于说明特别突出的点。

不要使用你的代码的屏幕截图：至少使用等宽字体，例如 verbatim 环境，但使用 listings 包（本书中使用）是非常推荐的。

44. 项目提交的风格指南

第 45 章

素数

在本章中，你将进行一系列关于素数的练习，这些练习是相互关联的。每个部分都列出了所需的先决条件。相反，这里的练习也参考了前面的章节。

45.1 算术

在执行本节之前，请确保你学习了 *4.5.*

练习 45.1。 读取两个数字并打印出它们的模数。模运算符是 x%y。

- 你能不使用运算符来计算模数吗?
- 对于负数输入，两种情况下你得到的结果是什么?
- 在输出它们之前，将所有结果分配给一个变量。

45.2 条件

在进行本节之前，请确保你学习了第 *5.1* 节。

练习 45.2。 读取两个数字并打印一条消息，说明第二个数字是否是第一个数字的除数:

Code:

```
// /divisiontest.cpp
  int number,divisor;
  bool is_a_divisor;
    /* ... */
  if (
    /* ... */
      ) {
    cout << "Indeed, " << divisor
         << " is a divisor of "
         << number << '\n';
  } else {
    cout << "No, " << divisor
         << " is not a divisor of "
         << number << '\n';
  }
```

Output

[primes] division:

```
( echo 6 ; echo 2 ) |
    divisiontest
Enter a number:
Enter a trial divisor:
Indeed, 2 is a divisor of 6

( echo 9 ; echo 2 ) |
    divisiontest
Enter a number:
Enter a trial divisor:
No, 2 is not a divisor of 9
```

45.3 循环

在进入本节之前，请确保您已学习节 *6.1*。

练习 45.3. 读取一个整数，并将一个布尔变量设置为真以确定它是否为质数，方法是测试较小的数字是否可以整除该数。

打印最终消息

您的数字是质数

or

`Your number is not prime: it is divisible by ....`

你只需报告找到一个因子。

向屏幕打印消息几乎从来不是严肃程序的目的。在之前的练习中，因此我们假设一个数的素数（或非素数）属性将在程序的其他部分使用。所以你想存储这个结论。

练习 45.4。 使用布尔变量重写之前的练习，以表示输入数的素数。

练习 45.5。 读取一个整数 r。如果它是素数，打印一条消息说明这一点。如果它不是素数，找到整数 $p \leq q$ 使得 $r = p \cdot q$ 并且使得 p 并且 q 尽可能接近。例如，对于 $r = 30$ 你应该打印出 `5,6`，而不是 `3,10`。你可以使用函数 *sqrt*。

45.4 函数

在执行本节之前，请确保你学习了 **??** 节。

chapter]ch:function

你之前写了几行代码来测试一个数是否为质数。现在我们将这段代码转换为一个函数。

练习 45.6. 编写一个函数 `is_prime`，它有一个整数参数，并返回一个布尔值，对应参数是否为质数。

```
int main() {bool ispri
me;isprime= is_prime(13);}
```

编写一个主程序，读取输入的数，并打印布尔值的值。（布尔值如何渲染？参见第 12.2.2 节。）

您的函数有一个或两个 **返回** 语句吗？您能想象其他可能性是什么吗？您对它有什么支持或反对的理由？

45.5 While loops

在进入本节之前，请确保您学习了节 *6.3*。

练习 45.7。 使用您的素数检测函数 `is_prime`，编写一个程序来打印多个素数：

- 从输入中读取一个整数 `how_多少`，表示应该打印多少个（连续的）素数。
- 打印那么多连续的素数，每个素数占一行。
  （提示：保持一个变量 number，每当找到一个新质数时，该变量的值就会增加。）

45.6 类和对象

在开始这一节之前，请确保你已经学习了第 *9.1* 节。

练习 45.8。 编写一个类 `primegenerator`，它包含：

- 方法 *number_of_primes_found*and *nextprime*;
- 同时编写一个不需要在类中的函数 `isprime`。

你的主程序应如下所示：

```
//6primesbyclass.cppcin >>nprimes{v6};primegenerator
sequence{v10};;while{(sequence{v20}.number_of
primes{v28}found{v31}()<nprimes{v36}) {int number=
sequence{v45}.nextprime{v49}();cout << "Number " << number << "
is prime"<< '\n'{v65};}
```

在之前的练习中，你定义了 `primegenerator` 类，并且创建了一个该类的对象：

```
primegenerator sequence;
```

但你可以让多个生成器，每个生成器都有自己的内部数据，因此它们相互独立。

练习 45.9。 哥德巴赫猜想 说，从 *4* 开始的每个偶数都是两个质数的和 $p+q$。 编写一个程序来测试这个对于你读入的上限内的偶数。使用 `primegenerator` 类你在练习 45.8<style id='20'> 中开发的。

这是一个自顶向下方法的绝佳练习！

1. 对偶数 e 进行外部循环。 2. 对于每个 e，生成所有质数 p。 3. 从 $p+q=e$ 可以得出 $q=e-p$ 是质数：测试那个 q 是否是质数。

对于每个偶数 `e` 则打印 `e,p,q`，例如：

```
数字 10 是  3+7
```

如果存在多个可能性，仅打印您找到的第一个。

哥德巴赫猜想的一个有趣的推论是，每个素数（从 5 开始）与其他两个素数的距离相等。

哥德巴赫猜想 *Goldbach conjecture* 指出，每个偶数 $2n$ （从 4 开始），是两个素数 $p+q$ 的和：

$$2n = p + q.$$

换句话说，每个数字 n 与两个素数的距离相等：

$$n = \frac{p+q}{2} \qquad \text{or} \qquad q - n = n - p.$$

特别是这对每个素数都成立：

$$\forall_{r\,\text{prime}} \exists_{p,q\,\text{prime}} : r = (p+q)/2 \text{ is prime.}$$

练习 45.10。

编写一个测试这个的程序。你需要至少一个循环来测试所有素数 r；对于每个 r，你还需要找到与它等距的素数 p,q。你需要用两个生成器来做这个，还是一个就足够了？对于 p,q,r，你需要三个吗？

对于每个 r 值，当程序找到的 p,q 值时，打印出 p,q,r 三元组，然后继续下一个 r。

45.6.1 异常

在开始这一节之前，确保你学习了节 *23.2.2*。

练习 45.11. 重新审视素数生成器类（练习 45.8 ）并在候选数过大时抛出异常。（你可以硬编码这个最大值，或使用一个限制；章节 24.2.）

代码：

```
1// /genx.cpp2
    try {
3     do {
4       auto cur = primes.nextprime();
5       cout <<cur << " \n" ;
6     } while (true);
7   } catch(    tring s ) {
8     cout << s << '\n';
9   }
```

输出
[素数] 生成器：

```
9931
9941
9949
9967
9973
达到最大整数
```

45.6.2 Prime numberdecomposition

在进行本节之前，请确保您已学习 *24.3.1*。

设计一个类 `Integer`，它将存储其值为其质因数分解。例如，

$$180 = 2^2 \cdot 3^3 \cdot 5 \quad \Rightarrow \quad \texttt{[2:2, 3:2, 5:1]}$$

您可以自己实现这种分解，将其作为 *vector*（第 i 个位置存储第 i 个质数的指数），但让我们使用一个 `map` 代替。

练习 45.12. 编写一个 `Integer` 的构造函数 从 *int*，以及方法 <style id='13'>as_`int` / `as_string`，将分解转换回经典形式。首先假设每个质因数只出现一次。

代码：

```
1// /decomposition.cpp
2   整数 i2(2);i2_
3   cout <<    .as strin    << ": "
4             g()i2. asint() '\n';
5
6   整数 i6(6);
7   cout <<i6. as string() ":              "
8         << i6.as_int() << '\n';
```

输出[
i pr mes] decomposition26:

```
2^1 : 2
2^1 3^1 : 6
```

练习 45.13。 将之前的练习扩展到质因数具有 > 1 的重复性。

代码：

```
1 // /decomposition.cpp
2   Integer i180(180);
3   cout << i180.as_string() << ": "
4        << i180.as_int() << '\n';
```

输出
[质数] 分解 180：

```
2^2 3^2 5^1 : 180
```

为整数实现加法和乘法。

实现一个 *Rational* 类用于表示有理数，有理数由两个 *Integer* 对象表示。该类应该有加法和乘法的方法。如果你学过运算符重载，请通过运算符重载来编写这些方法。

确保分子和分母中总是约去公因数。

45.7 范围

在完成这一节之前，请确保你学习了节 *14.1*。

Exercise 45.14. Write a range-based code that tests

$$\forall_{\text{prime } p} : \exists_{\text{prime } q} : q > p$$

练习 45.15. 重写练习 45.10，仅使用范围表达式，不使用循环。

练习 45.16. 在上述哥德巴赫练习中，你可能需要两个素数序列，然而它们却不是从同一个数开始的。你能让你的代码读取

```
all_of( primes_from(5) /* et cetera */
```

45.8 其他

以下练习需要 std:: 可选，你可以在第 24.6.2 节中学习它。

练习 45.17. 编写一个函数 *first_因子*，它可选地返回给定输入的最小因子。

```
// /optfactor.cppauto factor= first_因子(number);if
(factor.has_值())cout << "找到因子："<<factor.值() << '
\n';
```

45.9 埃拉托斯特尼筛法

埃拉托斯特尼筛法是一种用于素数的算法，它逐步筛选出所找到的任何素数的倍数。

1. 从整数 2 开始：2, 3, 4, 5, 6, ..2. 第一个数，2，是素数：记录它并移除所有倍数，得到

   3, 5, 7, 9.11, 13, 15, 17 . . .

3. 第一个剩余的数，3，是素数：记录它并移除所有倍数，得到

   5, 7, 11, 13, 17, 19, 23, 25, 29 . . .

4. 第一个剩余的数，5，是素数：record itand remove all multiples, giving

   7, 11, 13, 17, 19, 23, 29, . . .

45.9.1 数组实现

The sieve can be implemented with an array that stores all integers.

练习 45.18. 读取一个整数，表示你想要测试的最大数。创建一个同样长度的整数数组。将数组元素设置为连续的整数。应用筛法算法来找到质数。

45.9.2 流实现

使用数组的缺点是我们需要分配一个数组。更重要的是，数组的大小是由我们想要测试的整数数量决定的，而不是我们想要生成的质数数量。我们将采用上述生成器对象的想法，并将其应用于筛法算法：现在我们将有多个生成器对象，每个对象以先前对象为输入，并从中删除某些倍数。

练习 45.19. 编写一个流 类，该类生成整数，并通过指针使用它。

代码：

```
// /ints.cpp
  for(int i=0; i<7; i++i)
    cout<< "下一个整数： "
    h  i_<< t e  nts->next() << '\n';
```

输出

```
{s1} 筛 {s2} 整数：
下一个整数： 2
下一个整数：3
下一个整数： 4
下一个整数： 5
下一个整数： 6
下一个整数： 7
下一个 int： 8
```

接下来，我们需要一个流，它接受另一个流作为输入，并从中过滤出值。

练习 45.20. 编写一个类过滤流，其构造函数为 _

```
过滤 _ 流 (int filter, shared_ptr<流 > 输入 );
```

that

1. 实现 `next`，提供过滤后的值，2. 通过调用输入流的 `next` 方法并过滤出值。

代码：

```
// /odds.cpp
  auto 整数 =
    make_ 共享<流 >();
  auto 奇数 =
    共享 _ 指针 < 流 >
     ( new 过滤 _ 流 (2, 整数 ) );(i t t 0 t
  for5  n+ts) ep= ; s ep  ;   s ep
    cout << "next odd: "
         << odds->next() << '\n';
```

输出

```
{s1} 筛 {s2} 奇数：
下一个奇数： 3
next odd:  5
next odd:  7
下一个奇数: 9
下一个奇数： 11
```

Now you can implement the Eratosthenes sieve by making a `filtered_stream` for each prime number.

练习 45.21. 编写一个程序，按如下方式生成质数。

- 维护一个当前 流，即初始为质数流。
- 重复执行：
  - 记录当前流中的第一个元素，它是一个新的质数；- 并将当前 设置为一个新的流，该流以当前 为输入，并过滤掉刚刚找到的质数的倍数。

45.10 范围实现

在进行本节之前，请确保您已学习 *14.6.2* 节。

如果我们写出素数定义

$$\begin{aligned} D(n,d) &\equiv n|d=0 \\ P(n) &\equiv \forall_{d\leq\sqrt{N}}: \neg D(n,d) \end{aligned}$$

我们会发现这涉及两个我们迭代的流：

1. 首先存在所有 d 的集合，使得 $d^2 \leq n$；然后 2. 我们有测试这些 d 值是否为因子的布尔集合。

练习 45.22。 使用 `iota` 范围视图生成从 2 到无穷大的所有整数，并找到一个范围视图，在最后一个可能的因子处截断序列。

然后使用 `all_` 的 或 `any_` 的 范围化算法来测试这些潜在因子是否实际上是因子，因此判断您的数字是否为素数。

Exercise 45.23. Use the `filter` view to filter from an `iota` view those elements that are prime.

练习 45.24。 创建一个 `primes` 类，使其可遍历：

代码：

```
// g++ -std=c++20 -o allprimes allprimes.cpp
for ( auto p : allprimes ) {
  cout << p << ", ";
  if (p>100) break;
}
cout<< '\n';
```

输出

```
[primes] 范围：
缺失片段 /
code/primes/range.runout
```

45.11 用户友好性

使用 `cxxopts` 包（第 63.2 节）为某些素性测试程序添加命令行选项。

练习 45.25。 将你的旧素数测试程序，并添加命令行选项：

- -h 选项应打印出使用信息；

- 指定单个 int-- 测试 1001 应该打印出该数字以下的所有质数；
- 指定一组 int-- 测试 57,125,1001 应该测试这些数字的质数性。

45. 素数

第 46 章

几何

在这个练习集中，你将编写一个小型的 ‘几何学’<code> 包 </code>：操作点、线、形状的代码。这些练习主要使用第 <code>9</code> 节的内容。

46.1 基本函数

练习 46.1。 编写一个函数，输入为（ <code>float</code> 或 <code>double</code> ） x, y，返回点 (x,y) 到原点的距离。

测试以下几对：1, 0; 0, 1; 1, 1; 3,4。

练习 46.2。 编写一个函数，输入为 x, y, θ，该函数会改变 x 和 y，以对应将点 (x,y) 绕角度 θ 旋转。

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

你的代码应该表现得像：

代码：

```
// /rotate.cpp
  const float pi = 2*acos(0.0);
  float x{1.}, y{0.};
  rotate(x,y,pi/4);
  cout << "Rotated halfway: ("
       << x << "," << y << ")" << '\n';
  rotate(x,y,pi/4);
  cout << "Rotated to the y-axis: ("
       << x << "," << y << ")" << '\n';
```

Output

```
[geom] rotate:
```

旋转一半：

(0.707107,0.707107) 旋转到 y- 轴：

(0,1)

46.2 类

在进行本节之前，请确保您已学习 *9.1* 节。

类可以包含基本数据。在本节中，您将创建一个 Point 类，该类模拟笛卡尔坐标和定义在坐标上的函数。

练习 46.3. 创建一个带有构造函数的 Point 类

```
Point( float xcoordinate, float ycoordinate );
```

编写以下方法：

- distance_to_origin 返回一个 float。
- angle 计算向量 (x,y) 与 x- 轴的夹角。

练习 46.4. 扩展上一练习中的 Point 类，添加一个方法：distance，用于计算该点与另一个点之间的距离：如果 p,q 是 Point 对象，

p. 距离 (q)

计算它们之间的距离。

练习 46.5. 编写一个方法 *halfway*，给定两个 Point 对象 p,q，构造中点，即 $(p+q)/2$:

```
点 p(1,2.2),q(3.4,5.6); 点
h= p. 中点 (q);
```

你可以直接编写这个函数，或者你可以编写函数 Add 和 Scale 并将它们组合起来。（稍后你将学习运算符重载。）

你会如何打印一个 *Point* 以确保正确计算中点?

练习 46.6. 为点类编写一个默认构造函数：

```
Point() { /* default code */ }
```

你可以这样使用：

Point p;

但它表明它是未定义的：

代码：

```
// /linear.cpp  Pointp3; cout <<"未初始化的点 :"
        <<'\n';
p3.printout(); cout <<"使用未初始化的点 :"
        <<'\n'; auto p4 = Point(4,5)+p3;
  p4.printout();
```

输出
[geom] linearnan:

未初始化的点 :*Point*: *nan,nan*

使用未初始化的点 :*Point*:

nan,nan

提示：参见第 26.3.3 节。

练习 46.7. 回顾练习 46.2 使用 `Point` 类。现在您的代码应该像这样：

```
newpoint = point.rotate(alpha);
```

练习 46.8. 高级。你能创建一个可以容纳任意空间维度的 `Point` 类吗？提示：使用 `vector`；第 10.3 节。你能创建一个不需要显式指定空间维度的构造函数吗？

46.3 在另一个类中使用一个类

在执行本节之前，请确保您学习了第 *9.2* 节。

Exercise 46.9. Make a class `LinearFunction` with a constructor:

LinearFunction (Point input_p1 , Point input_p2) ;

和成员函数

float evaluate_at(float x);

你可以将其用作：

```
线性函数 line(p1,p2);cout << "在 4.0 处的值："<< line.evaluate_在
(4.0) << endl;
```

Exercise 46.10. Make a class `LinearFunction` with two constructors:

线性函数 (点输入 _p2) ; 线性函数 (点输入 _p1 , 点输入 _ p2) ;

其中第一个表示一条通过原点的直线。
再次实现 evaluate 函数，以便

```
LinearFunction line(p1,p2);
cout << "Value at 4.0: " << line.evaluate_at(4.0) << endl;
```

练习 46.11。 重新审视练习 46.2 和 46.7，引入一个 `Matrix` 类。您的代码现在可以像

```
新点 = 点 < 样式 id='2'>.</ 样式 > 应用 < 样式 id='4'>(</ 样式 > 旋转 _ 矩阵
< 样式 id='7'>);</ 样式 >
```

or

```
新点 = 旋转 _ 矩阵 . 应用 ( 点 ) ;
```

你能为其中任何一个进行辩护吗？

假设你想编写一个 Rectangle 类，该类可以有诸如 float Rectangle::area() 或 bool Rectangle::contains(Point) 的方法。由于矩形有四个角，你可以在每个 Rectangle 对象中存储四个 Point 对象。但是，存在冗余

那里：你只需要三个点来推断第四个。让我们考虑一个矩形的情况，其边是水平的和垂直的；那么你只需要两个点。

Intended API:

```
float Rectangle::area();
```

存储宽度和高度会很方便；对于

```
bool Rectangle::contains(Point);
```

存储 bottomleft/topright 点会很方便 ts.

练习 46.12。

1. 创建一个类 Rectangle （边平行于坐标轴），并使用构造函数：

   ```
   矩形 (Point botleft,float width,float height);
   ```

   逻辑实现是存储这些量。实现方法：

   float area(); float rightedge_x(); float topedge_y();

   and write a main program to test these.
2. 添加第二个构造函数

   ```
   矩形（点左下，点右上）;
   ```

   你能想出如何使用成员初始化列表为构造函数吗?

练习 46.13. 复制你上一个练习的解决方案，并重新设计你的类，以便它存储两个 Point 对象。你的主程序不应更改。

上一个练习说明了一个要点：对于设计良好的类，你可以更改实现（例如出于效率的考虑），而使用该类的程序不会更改。

46.4 是 - 关系

在执行本节之前，请确保你学习了节 *9.3*。

练习 46.14。 将定义矩形（ Rectangle ）的代码，从一个点、宽度和高度开始。

创建一个从 Rectangle 继承的类 Square。它应该有一个从 Rectangle 继承的 area 函数定义。

First ask yourself: what should the constructor of a `Square` look like?

练习 46.15。 重新审视 LinearFunction 类。添加 slope 和 intercept 方法。

现在将 LinearFunction 一般化为 StraightLine 类。这两个类几乎相同，除了垂直线。斜率和截距不适用于垂直线，因此设计 StraightLine 以便它内部存储定义点。让 LinearFunction 继承。

46.5 指针

在进入本节之前，请确保您已学习第 *16.1* 节。

以下练习有点人为性。

练习 46.16. 创建一个 `DynRectangle` 类，该类由两个指向 `Point` 对象的共享指针构造：

```
// /dynrectangle.cpp
  auto
    origin  = make_shared<Point>(0,0),
    fivetwo = make_shared<Point>(5,2);
  DynRectangle lielow( origin,fivetwo );
```

计算面积，缩放右上角点，并重新计算面积：

代码：

```
1// /dynrectangle.cpp"
2   cout << Area:    << lielow. 区域 () <<
      \n;
3     * ... */
4   // 将 'fivetwo' 点放大两倍
5   cout <<"Area: " << lielow.{area}() '\ '"
        n ;
```

输出

```
[ 指针 ]  动态矩形 :
面积 : 10
面积 : 40
```

您可以根据代码仓库中的文件 *pointrectangle.cxx* 来进行参考

46.6 更多内容

在执行此部分之前，请确保您已学习 *15.3*。

Rectangle 类最多存储一个角，但有时有一个包含所有四个角的数组会更方便。

练习 46.17. 添加一个方法

const vector<Point>< &corners()

到 Rectangle 类。结果是所有四个角的数组，顺序任意。通过编译器错误证明该数组不能被修改。

在进行本节之前，请确保您已学习 *9.5.7*。

练习 46.18. 回顾练习 46.5 并用重载运算符替换 `add` 和 `scale` 函数。

Hint: for the `add` function you may need '**`this`**'.

46. 几何

第 47 章

零点查找

47.1 二分法求根

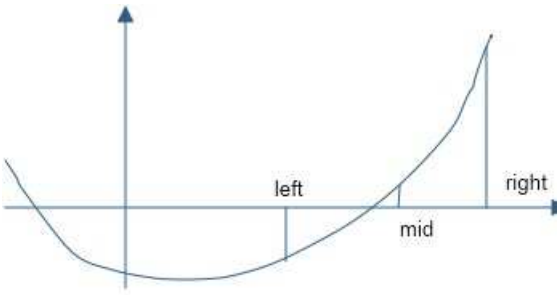


图 47.1：区间二分法求根

对于许多函数 f，求它们的零点，即满足 $f(x) = 0$ 的值 x，通常无法解析求解。这时你需要采用数值 求根 方法。在这个项目中，你将逐步开发更复杂的 二分法 求根方法的实现。

在这个方法中，你从一个函数值异号的两个点开始，将左点或右点移动到中点，具体取决于函数在该点的符号。见图 47.1。

在章节 47.2 中，我们将然后研究牛顿法。

在这里，我们将不关心方法之间的数学差异，尽管这些差异很重要：我们将使用这些方法来练习一些编程技巧。

47.1.1 简单实现

> 在进行本节之前，请确保您学习了章节 7。在进行本节之前，请确保您学习了章节 *10*。

让我们逐步开发第一个实现。为了确保我们代码的正确性，我们将使用测试驱动开发（TDD）方法：在我们将其集成到更大的代码中之前，我们会为每个功能编写一个测试来确保其正确性。（有关更多关于 TDD 的信息，以及特别是 Catch2 框架的信息，请参阅章节 68.2。）

47.1.2 多项式

首先，我们需要一种表示多项式的方法。对于一个 d 次多项式，我们需要 $d+1$ 个系数：

$$f(x) = c_0 x^d + \cdots + c_{d-1} x^1 + c_d \tag{47.1}$$

我们通过将系数存储在 向量 <double> 中来实现这一点。我们做出以下任意决定

1. 让这个向量的第一个元素是最高次项的系数，2. 为了使系数正确定义一个多项式，这个首项系数必须非零。

让我们从一个固定的测试多项式开始，由函数 `set_coefficients` 提供。为了使这个函数提供正确多项式，它必须满足以下测试：

```
// /testzeroarray.cppTEST_CASE( "coefficients represent polynomial" )
{vector<double> coefficients= { 1.5, 0., -3 } ;REQUIRE(
coefficients.size()) ;) ;REQUIRE( coefficients.front() !) ;. ) ; }
```

练习 47.1。编写一个设置系数例程，构建一个系数向量 :_

```
// /findzeroarray.cppvector<double> coefficients; set coefficients();
```

and make it satisfy the above conditions.

首先编写一组硬编码的系数，然后尝试从命令行读取它们。

练习 47.2。 附加题：使用 cxxopts 库（第 63.2.2 节）在命令行中指定系数。

前面我们假设了数组需要满足两个条件才能作为多项式的系数。你的代码可能会对此进行测试，所以让我们引入一个布尔函数 `is_proper_polynomial`：

- 此函数返回 `true` 如果数组满足两个条件；
- 如果任何一个条件不满足，它返回 `false`。

In order to test your function `is_proper_polynomial` you should check that

- 它能识别正确多项式，并且
- 对于不正确定义多项式的系数，它失败。

练习 47.3。 编写一个函数 `is_proper_polynomial` 如所描述，并为其编写单元测试，包括通过和失败的情况：

```
vector<double> 良好 = /* 正确的系数 */ ;REQUIRE( 是 _ 正确的
_ 多项式 ( 良好 ) ) ;vector<double> 不太 = /* 不正确的系数 */ ;
REQUIRE( 不是_ 正确的 _ 多项式 ( 不太 ) ) ;
```

接下来我们需要多项式求值。我们将构建一个函数 `evaluate_at`，其定义如下：

```
// /findzerolib.hpp
double evaluate_at( const std::vector<double>& coefficients,double x);
```

你可以以至少两种方式解释系数数组，但根据方程（47.1）我们规定了特定的解释方式。

因此我们需要一个测试，以验证系数确实是以首项系数优先的方式解释的，而不是以首项系数最后的方式解释的。例如：

```
// /testzeroclass.cpp
  polynomial second( {2,0,1} );
  // correct interpretation: 2x^2 + 1
  REQUIRE( second.is_proper() );
  REQUIRE( second.evaluate_at(2) == Catch::Approx(9) );
  // wrong interpretation: 1x^2 + 2
  REQUIRE( second.evaluate_at(2) != Catch::Approx(6) );
```

（此处省略了 `TEST_CASE` 标题。）

现在我们编写一个通过这些测试的函数：

练习 47.4. 编写一个函数 `evaluate_` 在该函数计算

$$y \leftarrow f(x).$$

并确认它通过上述测试。

```
double evaluate_在(多项式系数,double x);
```

For bonus points, look up *Horner's rule* and implement it.

多项式函数实现后，我们可以开始着手算法。

47.1.3 左 / 右搜索点

假设 x_-,x_+ 满足

$$x_- < x_+, \quad \text{and} \quad f(x_-) \cdot f(x_+) < 0,$$

也就是说，左点和右点的函数值符号相反。那么在该区间内存在零点 (x_-,x_+)；参见图 47.1。

但是如何找到搜索的外部边界呢?

如果多项式的次数是奇数，可以通过从任意两个起始点向左和向右移动足够远来找到 x_-,x_+。

对于偶数次多项式，没有这样的简单算法（实际上可能不存在零点），所以我们放弃尝试。

我们首先编写一个函数 `is_odd` 来测试多项式的次数是否为奇数。

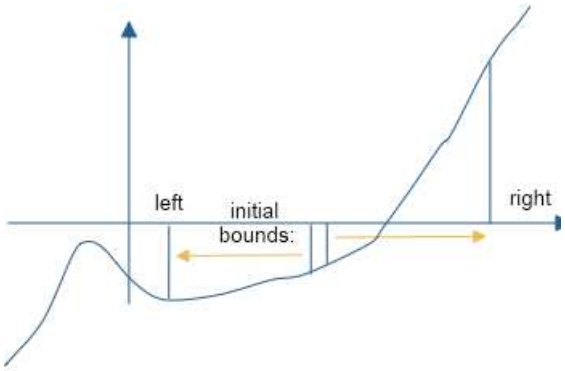


图 47.2：设置初始搜索点

练习 47.5。 使以下代码工作：

```
// /findzeroarray.cpp
  if ( not is_odd(coefficients) ) {
    cout << "This program only works for odd-degree polynomials\n";
    exit(1);
  }
```

你可以这样测试上述内容：

```
// /testzeroarray.cpp多项式二次{2,0,1}; // 2x^
2 + 1REQUIRE( notis_ 奇数(二次) ); 多项式三次
{3,2,0,1}; // 3x^3 + 2x^2 + 1REQUIRE( 是_ 奇数 (
三次 ));
```

现在我们可以找到 x_-,x_+: 从某个区间开始，移动端点直到函数值具有相反的符号。

练习 47.6。 编写一个函数 *find_ 初始 _ 边界*，该函数计算 x_-,x_+ 使得

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

解决以下问题：

1. 函数的良好原型是什么？ 2. 如何将点移动足够远以满足此条件？ 3. 你能更紧凑地计算上述测试吗？

由于对于偶次多项式，在中间找到一个零点并不总是可能的，我们完全拒绝这种情况。在以下测试中，对于偶次多项式，我们抛出异常（参见第 23.2.2 节，尤其是 23.2.2.3 节）：

```
// /testzeroarray.cppright = left+1;多项式第二{2,0,1}; //
2x^2 + 1REQUIRE_THROWS( find_ 初始_ 边界
(second,left,right));
```

```
三次多项式{3,2,0,1}; // 3x^3 + 2x^2 + 1REQUIRE_NOTHROW(
find_initial_bounds(三次, 左, 右));REQUIRE( 左<右 );
```

确保您的代码通过这些测试。您需要为函数值添加什么测试?

47.1.4 Root finding

求根过程整体如下:

- 您从函数值符号相反的点 x_-,x_+ 开始;然后您知道它们之间存在一个零点。
- 用于查找该零点的二分法观察中点,并根据中点的函数值:
- 将其中一个边界移动到中点,使得函数在 left 和 right 搜索点再次具有相反的符号。

代码的结构如下:

```
double find_zero( /* something */ ) {
  while ( /* left and right too far apart */ ) {
    // move bounds left and right closer together
  }
  return something;
}
```

同样,我们分别测试所有功能。在这种情况下,这意味着移动边界应该是一个可测试的步骤。

Exercise 47.7. Write a function `move_bounds_closer` and test it.

void move bounds closer_ _(
std::vector<double>< coefficients, double& left,
double& right);

在这个函数上实现一些单元测试。

最后,我们在顶层函数 `find_zero` 中把所有东西放在一起。

练习 47.8. 使这个调用工作:

// /findzeroarray.cppauto zero =find_zero{(coefficients, 1.e-8);cout "<<Found root <<
zero<< with value << evaluate_at{(coefficients, zero) }<< '\n'{v51};

设计单元测试,包括对达到的精度的测试,并确保你的代码通过它们。

47.1.5 对象实现

重新审视第 47.1.1 节的练习,并引入一个多项式类来存储多项式系数。现在几个函数变成了这个类的成员。

同时更新单元测试。

一些进一步的建议：

1. 你能让你的多项式类看起来像一个函数吗？

类 多项式 {/*... */}main() { 多项式 p;float y= p(x);} 参见第 9.5.7.1 节。 2. 你能泛化多项式类，例如推广到特殊形式的情况，如 $(1+x)^n$？ 3. 将你的多项式模板化：参见下一小节。

47.1.6 模板化

在目前的实现中我们使用了 `double` 作为数值类型。制作一个模板化的版本，使其既适用于 `float` 也适用于 `double`。

你能看到这两种类型在可达到的精度上有何不同吗？

47.2 牛顿方法

在进入本节之前，请确保你学习了第 *13* 节。

在本节中，我们来看 牛顿方法 。这是一种迭代方法，用于求解函数 f 的零点，即它计算一系列值 $\{x_n\}_n$，使得 $f(x_n) \to 0$。该序列由

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

与 x_0 任意选择。详情请参见 HPC 书籍 [11]，第 18 .

虽然在实际应用中牛顿方法用于复杂函数，但在这里我们将看一个简单的例子，该例子实际上在计算历史中有其基础。早期的计算机没有用于计算平方根的硬件。相反，它们使用了牛顿方法。

假设你有一个正值 y，并且你想计算 $x = \sqrt{y}$。这相当于找到的零点

$$f(x) = x^2 - y$$

其中 y 是固定的。为了表示这种对 y 的依赖关系，我们将写 $f_y(x)$。然后，牛顿法通过计算找到零点

$$x_{\text{next}} = x - f_y(x)/f'_y(x)$$

直到猜测足够准确，即直到 $f_y(x) \approx 0$。

我们将不讨论如何选择一个复杂的迭代停止测试；那是数值分析课程的内容，而不是编程课程的内容。

47.2.1 函数实现

当然，编写这个特定案例很简单；大约需要 10 行代码。然而，我们希望有一个通用的代码，它接受任何两个函数 f, f'，然后使用牛顿法来找到我们特定函数 f 的零点。

练习 47.9。

- 编写函数 f(x,y) 和 deriv(x,y)，它们计算 $f_y(x)$ 和 $f'_y(x)$，用于上述 f_y 的定义。
- 读取一个值 y 并迭代，直到 $|f(x,y)| < 10^{-5}$。打印 x。
- 第二部分：编写一个名为 newton_root 的函数来计算 $\sqrt{y}$。

47.2.2 使用 lambda 函数

上面你编写了符合以下规范的函数：

```
// /newton-fun.cppdouble
f(double x);double
fprime(double x);
```

以及算法：

```
// /newton-fun.cpp
  double x{1.};
  while ( true ) {
    auto fx = f(x);
    cout << "f( " << x << " ) = " << fx << '\n';
    if (std::abs(fx)<1.e-10 ) break;
    x = x - fx/fprime(x);
  }
```

练习 47.10. 将你的代码改写为使用 lambda 函数来处理 `f` 和 `fprime`。

您可以根据代码仓库中的文件 *newton.cxx* 作为基础

接下来，我们通过编写一个通用的函数 `newton_root` 来使代码模块化，该函数包含上一练习中的牛顿方法。由于它必须适用于任何函数 f, f'，您需要将目标函数和导数作为参数传递：

```
double root = newton_root( f,fprime );
```

练习 47.11. 重写上述牛顿练习，使其使用一个函数 t is used as:

```
double root = newton_root( f,fprime );
```

调用函数

1. 首先使用你已经创建的 lambda 变量；2. 但在一个更好的变体中，直接使用 lambda 表达式作为参数，也就是说，不将它们赋值给变量。

接下来我们扩展功能，但不是通过改变根查找函数：相反，我们使用一种更通用的方式来指定目标函数和导数。

Exercise 47.12. 将牛顿练习扩展到循环中计算根：

```
// /newton-lambda.cppfor(int n=2; n<=8;
n++n{v20}) {cout << "sqrt(" << n << ") "
= "\n" << newton root_({v46})<< ' \n'
{v51};
```

没有 lambda，你需要定义一个函数

```
double squared_minus_n( double x,int n ) {
  return x*x-n; }
```

然而，`newton_root` 函数只接受一个实参的函数。使用捕获使 `f` 依赖于整数参数。

Exercise 47.13. 您不需要梯度作为显式函数：您可以将其近似为

$$f'(x) = \big(f(x+h) - f(x)\big)/h$$

for 某个 h 的值。

编写一个只接受目标函数的根查找函数：

```
double newton_root( function< double(double)> f )
```

您可以使用固定值 `h=1e-6`。

不要重新实现整个 newton 方法：相反，为梯度创建一个 lambda 并将其传递给您之前编写的 `newton_root`函数。

练习 47.14. 附加：你能通过牛顿法计算对数吗?

47.2.3 模板实现

牛顿法对复数和实数同样适用。

练习 47.15. 重写你的牛顿程序，使其适用于复数：

```
// /newton-complex.cpp
  complex<double> z{.5,.5};
  while ( true ) {
    auto fz = f(z);
    cout << "f( " << z << " ) = " << fz << '\n';
    if (std::abs(fz)<1.e-10 ) break;
    z = z - fz/fprime(z);
  }
```

你可能会遇到一个问题，即你不能立即在复数和一个 **float** 或 **double** 之间进行操作。使用 **static_转换**；参见第 26.2.1 节。

那么你不得不写两个独立的实现，一个用于实数，一个用于复数？（而且你可能需要为 `float` 和 `double`！）
这就是模板发挥作用的地方；章节 22。

你可以将你的牛顿函数和导数模板化：

```
// /newton-double.cpp模板<
typename T>T f(T x) {return
x*x - 2; }; 模板 <typename T>T
fprime(T x) { return 2* x; };
```

然后写

```
// /newton-double.cpp
  double x{1.};
  while ( true ) {
    auto fx = f<double>(x);
    cout << "f( " << x << " ) = " << fx << '\n';
    if (std::abs(fx)<1.e-10 ) break;
    x = x - fx/fprime<double>(x);
  }
```

练习 47.16。 使用模板更新你的牛顿程序。如果你已经让它适用于 `double`，尝试使用 `complex<double>`。它有效吗？

练习 47.17。 使用你的复数牛顿方法来计算 $\sqrt{2}$。它有效吗？$\sqrt{}$

How about -2?

练习 47.18。 你能将使用 lambda 表达式的牛顿代码模板化吗？你现在函数的头部将是：

```
// /lambda-complex.cpp
template<typename T>
T newton_root_
   ( function< T(T) > f,
     function< T(T) > fprime,
     T init) {
```

例如，你会计算 $\sqrt{2}$ 为：

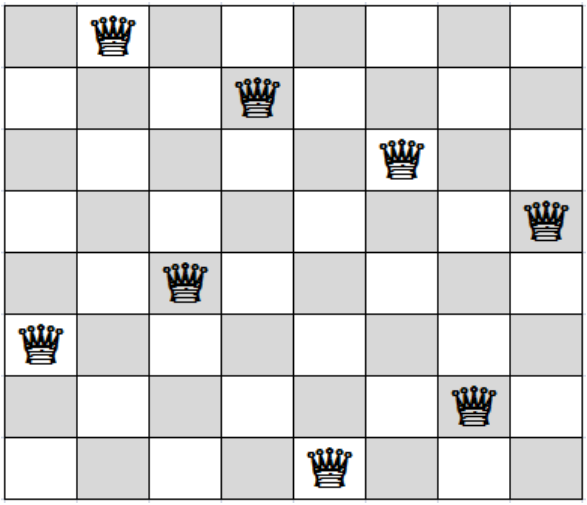
```
// /lambda-complex.cpp
  cout << "sqrt -2 = " <<
       newton_root<complex<double>>
            ( [] (complex<double> x) {
                 return x*x + static_cast<complex<double>>(2); },
              [] (complex<double> x) {
                return x * static_cast<complex<double>>(2); },
              complex<double>{.1,.1}
```

```
                    )
            << '\n';
```

第 48 章

八皇后问题

一个著名的递归编程练习是 <code> 八皇后 </code> 问题：是否可以在棋盘上放置八个皇后，使得根据棋规，没有两个皇后会互相 ‘攻击’？



48.1 问题陈述

‘八皇后问题’ 的精确陈述是：

- 在 8×8 棋盘上放置八个棋子，不在同一个方格上；使得
- 没有两枚棋子位于同一行，
- 没有两枚棋子位于同一列，并且
- 没有两枚棋子位于同一斜线。

一个系统性的解决方案会运行：

1. 在第一行的任何位置放置一枚棋子；2. 对于第一行的每个选择，尝试第二行的所有位置；3. 对于前两行的所有选择，尝试第三行的所有位置；4. 当你在所有八行都放置了棋子时，评估棋盘是否满足条件。

练习 48.1。 这个算法将生成所有 8^8 个棋盘。你至少能想到一种加速搜索的方法吗?

由于目前数字八是固定的，你可以将这段代码写成一个八层嵌套的循环。然而，这并不优雅。例如，上述解释中数字 8 的唯一原因是这是棋盘的传统大小。将问题更抽象地表述为在 n 个皇后放置在 $n \times n$ 个棋盘上，它有 $n \geq 4$ 个解决方案。

48.2 解决八皇后问题，基本方法

这个问题需要你了解数组 / 向量；第 10 章。另外，请参阅第 68 章关于 TDD。最后，请参阅第 24.6.2 节关于 std:: 可选。

基本策略将是按行填充，每次指示下一个皇后将占据的列。使用面向对象（ OO ）策略，我们创建一个类 *ChessBoard*，它包含一个部分填充的棋盘。

The basic solution strategy is recursive:

- 令 *current* 为当前部分填充的棋盘；
- 我们调用 *current*。*place_* 皇后 () 来尝试完成棋盘；
- 然而，使用递归，这个方法只能填充一行，然后调用 *place_queens* 在这个新棋盘上。

所以 *ChessBoard*::*place_queens*() {// 对于 c = 1 ... 列数： // 复制棋盘 // 在副本的下一行、列 c 放置一个皇后 // 然后调用 place_queens() 在副本上； // 调查结果 .....}

这个例程返回一个解决方案，或者一个表示没有可能找到解决方案的指示 .

在下一节中，我们将以 TDD 的方式系统地开发一个解决方案。

48.3 通过 TDD 开发解决方案

我们现在逐步使用测试驱动开发来开发八皇后问题的面向对象解决方案。

棋盘 我们首先构建一个棋盘，其构造函数仅指示问题的规模：

```
// /queens.hppChessBoard(
int n );
```

这是一个大小为 $n \times n$ 的 “ 通用棋盘 ”，并且最初应该是空的。

练习 48.2. 编写这个构造函数，用于创建一个大小为 $n \times n$ 的空棋盘。

请注意，棋盘的实现完全取决于你。在以下内容中，你会得到需要满足功能的测试，但任何使这一点成立的实现都是正确的解决方案。

记账：下一行是什么？ 假设我们按行填充棋盘，我们有一个辅助函数，它返回要填充的下一行：

```
// /queens.hppint next_row_to_be_
filled()
```

这给了我们第一个简单的测试：在一个空棋盘上，要填入的行是第 0 行。

Exercise 48.3. Write this method and make sure that it passes the test for an empty board.

```
// /queentest.cppTEST_CASE( "空棋盘","[1]")
{constexprint n=10;ChessBoard
empty(n);REQUIRE( empty.next_row_to_be_
filled()==0);}
```

根据 TDD 的规则，你可以实际编写这个方法，使其仅满足空棋盘的测试。稍后，我们将测试这个方法在填入几行后是否能给出正确的结果，当然，你的实现需要是通用的。

放置一个皇后 接下来，我们有一个函数来放置下一个皇后，无论这是否能给出一个可行的棋盘（即没有棋子可以互相攻击）：

```
// /queens.hppvoid place_next_queen_at_
列(inti);
```

这种方法首先应该捕获错误的索引：我们假设放置例程对无效的列号抛出异常。

```
ChessBoard::place_next_queen_at_column( intc )
{if ( /* c 是棋盘外的 */ )throw(1); // 或者其他异常。
```

(假设你没有测试错误的索引。你能构造一个在任何大小下都简单的 ‘ 作弊 ’ 解决方案吗？)

练习 48.4。 编写这个方法，并确保它通过以下对有效和无效列号的测试：

```
// /queentest.cpp
  REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
  REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
  REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
  REQUIRE( empty.next_row_to_be_filled()==1 );
```

(从现在开始，我们只会给出测试的主体。)

现在，是时候开始写一些严肃的内容了。

是否可行？ 如果你有一个棋盘，即使是部分的，你想要测试它是否可行，这意味着已经放置的皇后不能互相捕获。

这个方法的原型是：

```
// /queens.hppbool
可行()
```

这个测试首先必须适用于简单情况：空棋盘是可行的，只有一个棋子的棋盘也是可行的。

```
// /queentest.cpp
  ChessBoard empty(n);
  REQUIRE( empty.feasible() );
```

// /queentest.cpp
ChessBoard one _empty; 一个 . 放置 _ 下一个 _ 皇后 _ 在 _ 列 *(0);* 要求一个 . 下一个 _ 行 _ 要被 _ 填充 *();* 要求一个 . 可行的 *());*

练习 48.5。 编写方法，并确保它通过这些测试。

我们不应该只做成功的测试，有时也被称为代码中的 ‘ 快乐路径 ’ 。例如，如果我们把两个皇后放在同一列中，thetest 应该失败。

练习 48.6。 以上在位置 (0,0) 的初始尝试中，在下一行的列零中添加另一个皇后。检查它是否通过测试：

```
// /queentest.cppChessBoard collide= 一个;//
在 ‘ 冲突 ’ 位置放置一个皇后 collide。放置 _ 下一个
_ 皇后 _ 在 _ 列 (0);// 并测试这是不可行的 REQUIRE(
不 collide。可行 () );
```

添加一些你自己的测试。（这些不会被提交脚本执行，但你可能仍然会发现它们有用。）

测试配置 如果我们想测试非平凡配置的可行性，一个好的想法是能够 ‘ 创建 ’ 解决方案。为此，我们需要第二种构造函数，我们从棋子的位置构造一个完全填满的棋盘。

// /queens.hpp
ChessBoard(int n, vector<int> cols);
ChessBoard(vector<int> cols);

- 如果构造函数仅使用一个向量调用，这描述了一个完整的棋盘。
- 添加一个整数参数表示棋盘的大小，而向量仅描述已填满的行。

Exercise 48.7. 编写这些构造函数，并测试显式给出的解决方案是否是一个可行的棋盘：

```
// /queentest.cppChessBoard five(
{0,3,1,4,2} );REQUIRE( five.feasible() );
```

对于优雅地实现这一点，请参见委托构造函数 s ； 第 9.4.1 节。

最终我们必须编写棘手的部分。

48.4 递归解法

主函数

```
// /queens.hpp
可选 optional<ChessBoard> place_queens()
```

它接受一个棋盘，无论是否为空，并试图填充剩余的行。

一个问题在于，这种方法需要能够传达，给定某些初始配置，没有解决方案是可能的。为此，我们让位置 _ 皇后 是 可选 < 棋盘 >：

- 如果可以完成当前棋盘以得到一个解决方案，我们返回那个填满的棋盘；
- 否则我们返回 {}，表示没有可能的解决方案。

根据第 48.2 节中讨论的递归策略，这种放置方法大致有以下结构：

```
放置 _ 皇后 () {for ( int 列 =0; 列 <n; 列 ++ ) { 棋盘 next_
*this;// 在 'next' 棋盘的列 col 放置一个皇后 // 如果这是可行的并且
填满，我们就有一个解决方案 // 如果它是可行的但没有填满，递归 }}
```

The line

```
ChessBoard next = *this;
```

复制你所处的对象。

Remark 37 *Another approach would be to make a recursive function*

```
boolplace_queen( const ChessBoard& current, ChessBoard &next );//
如果可能则为 true，不可能则为 false
```

最终步骤 Above you coded the method `feasible` that tested whether aboard is still a candidate for a solution. Since this routine works for any partially filled board, you also need a method to test if you're done.

练习 48.8. 写一个方法

bool filled();

并为其编写测试，包括正向和反向测试。

现在你已经能够识别解决方案了，是时候编写解决方案例程了 .

练习 48.9. 编写方法

```
// /queens.hpp 可选<ChessBoard> 放置_
queens()
```

Because the function *place_queens* is recursive, it is a little hard to test in its entirety.

我们从一个更简单的测试开始：如果你几乎得到了解决方案，它就能完成最后一步。

练习 48.10. 使用构造函数

```
ChessBoard( int n,vector<int> cols )
```

生成一个除了最后一行外所有行都已填满且仍然可行的棋盘。测试你是否能找到解决方案：

```
// /queentest.cppChessBoard almost( 4, {1,3,0} );
auto solution= almost.place_queens();REQUIRE(
solution.has_value() );REQUIRE( solution-filled() );
```

由于这个测试只填充最后一行，它只做一次循环，所以可以打印出诊断信息，而不会被大量的输出所淹没。

解决方案和非解决方案 现在你已经有了解决方案例程，测试它从空棋盘开始是否工作。例如，确认没有 3×3 解决方案：

```
// /queentest.cppTEST_CASE( "没有 3x3 解
决方案","[9]") {ChessBoard
three(3);auto solution= three.place_
queens();REQUIRE( notsolution.has_
value() );}
```

另一方面，4×4 解决方案确实存在：

```
// /queentest.cppTEST_CASE( "有 4x4 解决方案","
[10]" ){ChessBoard four(4);auto solution=
four.place_queens();REQUIRE( solution.has_
value());}
```

练习 48.11. (可选) 你能修改你的代码，使其计算所有可能的解决方案吗?

Exercise 48.12. (可选) 随着 n 的变化，求解时间如何表现?

第 49 章

传染病模拟

本节包含一系列练习，逐步构建起一个较为真实的传染病传播模拟。

49.1 模型设计

可以通过统计方法模拟疾病传播，但在这里我们将构建一个显式模拟：我们将维护对人口中所有人的显式描述，并跟踪每个人的状态。

我们将使用一个简单的模型，其中一个人可以是：

- sick: 当它们生病时，可以感染其他人；
- susceptible: 它们是健康的，但可能被感染；
- recovered: 它们曾经生病，但现在不再携带疾病，且不能再次被感染；
- vaccinated: 它们是健康的，不携带疾病，且不能被感染。

在更复杂的模型中，一个人可能只在生病的一部分时间内具有传染性，或者可能发生其他疾病的继发性感染，等等。在这里我们保持简单：任何生病的人在他们生病时都可以感染其他人。

在下面的练习中，我们将逐步开发一个模型，描述疾病如何从最初的感染源传播。然后程序将跟踪人口从一天到下一天，无限运行，直到人口中没有任何生病的人。由于没有再感染，运行将始终结束。稍后我们将向模型添加突变，这可以延长流行病的持续时间。

49.1.1 其他建模方式

与其在代码中捕获每一个人，可以使用 “ 接触网络 ” 模型，采用常微分方程方法进行疾病建模。这样，就可以通过单个标量来模拟感染者的百分比，并推导出该标量及其他标量的关系 [2, 15]。

这是一种 “ 隔室模型 ”，其中三个 SIR 状态（易感者、感染者、移除者）都是隔室：人群的一部分。接触网络和隔室模型都捕捉了部分真相。事实上，它们可以结合。我们可以将一个国家视为一组城市，

人们可以在任意两座城市之间旅行。然后我们在城市内部使用隔室模型，并在城市之间使用接触网络。

在这个项目中，我们只会使用网络模型。

49.2 编码

以下部分是该项目代码的逐步开发。稍后我们将讨论如何将此代码作为科学实验运行。

注意 38 在许多地方你需要一个随机数生成器。你可以使用 *C* 语言的随机数生成器（第 *24.7.5* 节），或者使用第 *24.7* 节中的新 *STL* 生成器。

49.2.1 人员基础

疾病模拟的最基本组件是将一个人感染疾病，并观察该感染的时序发展。因此你需要一个人员类和一个疾病类。

在开始编码之前，问自己这些类需要支持哪些行为 .

- Person。人员的的基本方法是 1. 感染； 2. 接种疫苗；和 3. 每日进展。此外你可能需要查询人员的状态：他们是健康的、生病的还是康复的?

- Disease。目前，疾病本身并没有做什么。（在项目后期你可能希望它有一个方法 `mutate`。）然而你可能需要查询某些属性：1. 传播几率；和 2. 感染时人员生病的天数。

对单个人员的测试可以输出如下内容：

在第 <code style id='1'>10</code> 天，Joe 易感在第 <code style id='3'>11</code> 天，Joe 易感在第 <code style id='5'>12</code> 天，Joe 易感在第 <code style id='7'>13</code> 天，Joe 易感在第 <code style id='9'>14</code>，Joe 生病 <code style id='11'>(5</code> 天 <code style id='13')></code> 即将康复在第 <code style id='15'>15</code>，Joe 生病 <code style id='17'>(4</code> 天 <code style id='19')></code> 即将康复在第 <code style id='21'>16</code>，Joe 生病 <code style id='23'>(3</code> 天 <code style id='25')></code> 即将康复在第 <code style id='27'>17</code>，Joe 生病 <code style id='29'>(2</code> 天 <code style id='31')></code> 即将康复在第 <code style id='33'>18</code>，Joe 生病 <code style id='35'>(1</code> 天 <code style id='37')></code> 即将康复在第 <code style id='39'>19</code>，Joe 康复

<code style id='1'> 练习 49.1. </code> 编写一个 <code style id='3'>Person</code> 类，包含以下方法：

- <code style id='2'> 状态 </code>_<code style id='5'>string</code><code style id='7'>() </code><code style id='9'>: 返回描述人物状态的 </code><code style id='11'>string</code><code style id='13'>;</code>
- <code style id='2'> 一天 </code>_<code style id='5'> 之后 </code>_<code style id='8'> 天 </code><code style id='10'>() </code>: 将人物状态更新到下一天;
- <code style id='2'> 感染 </code><code style id='4'>(</code><code style id='6'>s</code><code style id='8'>)</code>: 用疾病对象感染一个人，其中疾病对象

Disease s<code style

i

d='1'>(</code>n<code

style id='3');</code>

指定运行 n 天。

您的主程序可以像这样：

```
// /person.cpp
  for (int step  = 1; ; step ++)            {

    joe.<one>_<more>_<day>(); //
     * ... *
    cout << "On day " <<step <<j       ", Joe is "
    _    i()<   oe.status str ng    << '\n';
    if (joe.is_recovered())
      break;
  }
```

其中感染部分被遗漏了。

您的主要关注点是如何对一个人的内部状态进行建模。这实际上是两个独立的问题：

1. 状态，以及 2. 如果生病了，需要多少天才能康复。

您可以用一个整数来实现这一点，但最好使用两个。此外，编写足够多的支持方法，例如 `is_recovered` 测试。

49.2.1.1 Person tests

编写看似正确但实际上并非在所有情况下都表现正常的代码很容易。因此，对代码进行一些系统性的测试是一个好主意。

确保您的 `Person` 对象通过这些测试：

- 感染了 100% 传染性疾病后，它们应该注册为生病。
- 如果它们接种了疫苗或康复，并且接触了这种疾病，它们将保持原始状态。

- 如果一种疾病的传染概率为 50%，并且有大量人接触了它，大约有一半的人应该生病。这个测试可能有点难以编写。

你能使用 *Catch2* 单元测试框架吗？参见章节 63.3。

49.2.2 交互

接下来我们模拟人与人之间的交互：一个人健康，另一个人感染，当两者接触时，疾病可能会传播。

```
// /interaction.cpp// 感染者 // , // 健康
者 // ; // 感染 // （// 流感 // ） ; /* ...
*/// 健康者 // 。// 触摸 // （// 感染者 // ） ;
```

疾病传播有一定的概率，因此你需要指定这个概率。你可以让声明为：

```
Disease flu( 5, 0.3 );
```

where the first parameter is the number of days an infection lasts, and the second the transfer probability.

练习 49.2. 为 疾病 类添加传播概率，为 人类添加一个 接触方法。设计和运行一些测试。

练习 49.3. 加分项：你能让以下疾病规范工作吗？

```
// /interaction.cpp疾病 流感 ; 流感 .
持续时间 ()  = 20; 流感 . 传播 _ 概率 () =
p;
```

为什么你可以认为这比之前建议的语法更好 ？

49.2.2.1 交互测试

将上述测试进行修改，但现在一个人与感染者接触，而不是直接接触疾病。

49.2.3 种群

接下来我们需要一个 种群 类，其中种群包含一个向量，该向量由 *Person* 对象组成。最初我们只感染一个人，并且没有疾病的传播。

种群 类至少应该有以下方法：

- 随机 _ 感染以从一个受感染的种群部分开始；
- 随机接种以从一个受感染的个体数量开始 ._
- 计数函数 *count_* 感染 和 *count_* 接种。

要运行一个逼真的模拟，您还需要一个 一个 _ 更多 _ 天 方法来使人口经历一天。这是您代码的核心，我们将在下一节逐步开发它。

49.2.3.1 人口测试

大多数人口测试将在下一节进行。目前，请确保您通过以下测试：

- 在疫苗接种率为 100% 的情况下，确实应该每个人都接种。

49.3 流行病模拟

为了模拟疾病在人群中的传播，我们需要一个更新方法来使人群推进一天：

- 患病者会与人群中的其他成员接触；
- 并且每个人都变老一天，这意味着患病者离康复更近一天。

我们分几个步骤来开发这个方法。

49.3.1 无接触

首先假设人们之间没有接触，因此疾病将随着最初的感染者结束。

跟踪输出应该看起来像这样：

```
Size of
population?
In step   1 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   2 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   3 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   4 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   5 #sick:    1 :  ? ? ? ? ? ? ? ? ? + ? ? ? ? ? ? ? ? ?
In step   6 #sick:    0 :  ? ? ? ? ? ? ? ? ? - ? ? ? ? ? ? ? ? ?
Disease ran its course by step 6
```

备注 39 这样的显示有助于检查程序行为。如果你在报告中包含这样的显示，请确保使用等宽字体，并且不要使用需要换行的种群大小。在进一步的测试中，你应该使用大种群，但不要包含这些显示。

Exercise 49.4. Program a population without infection.

- 编写 种群类。构造函数接受人数：

  Population population(npeople);

- 编写一个方法，感染一定数量的随机人群 e:

  // /pandemic.cpp

  *population.random_* 感染（发烧，初始_感染）;

- Write a method `count_infected` that counts how many people are infected.
- Write an `one_more_day` method that updates all persons in the population.
- Loop the `one_more_day` method until no people are infected: the `Population::one_more_day` method should apply `Person::one_more_day` to all person in the population.

编写一个例程，显示人群的状态，例如：？表示易感人群，+ 表示感染人群，- 表示康复人群。

49.3.1.1 测试

测试在疾病期间，感染人数保持不变，并且健康人数和感染人数之和等于人口规模。

49.3.2 传播

过去的练习过于简单：唯一的零号病人是唯一一个生病的人。现在让我们加入传播因素，并研究从单个感染者开始传播疾病的情况。

我们从一个非常简单的感染模型开始。

练习 49.5. 编写一个模拟程序，其中在每一步中，感染者的直接邻居现在也可能生病。

使用不同的人口规模和传染概率运行多次模拟。是否存在人们能够逃脱生病的情况?

49.3.2.1 测试

进行一些合理性测试:

- 如果一个人感染了具有 $p = 1$ 的疾病，那么第二天应该有 3 个人生病。除非感染者是第一个或最后一个:那么就有两个。
- 如果个体 0 被感染，并且 $p = 1$，模拟应该运行相当于人口规模那么多的天数。
- 如果 $p = 0.5$，之前的案例会怎样?

49.3.3 疫苗接种

练习 49.6。 加入疫苗接种：读取另一个数字，代表已接种疫苗人群的百分比。随机选择人群中的成员。

Describe the effect 已接种疫苗人群对疾病传播的影响。为什么是 t his model unrealistic?

49.3.4 传播

为了使模拟更加真实，我们让每个生病的人每天与固定数量的随机人接触。这给我们提供了或多或少的结果 *SIR* 模型； https://en.wikipedia.org/wiki/Epidemic_ 模型。

将一个人每天接触的人数设置为 6 左右。（你也可以让这个值作为随机值的上限，但这并不改变模拟的本质。）你已经编写了当一个人接触到感染者时自己生病的概率。再次从单个感染者开始运行模拟。

练习 49.7。 编写随机交互。现在运行一些模拟，变化

- 接种疫苗的人群百分比，和
- 接触时疾病传播的概率。

记录疾病在人群中持续的时间。在固定的接触次数和传播概率下，这个数字如何随接种疫苗的百分比变化?

将此函数报告为表格或图形。确保你有足够的数据点以得出有意义的结论。使用一个现实的种群大小。你也可以进行多次运行并报告平均值，以平衡随机数生成器的影响。

练习 49.8。 调查 “ 群体免疫 ” 的问题：如果足够多的人接种了疫苗，那么一些未接种疫苗的人仍然永远不会生病。假设你想让未接种疫苗但从未生病概率超过 95%。调查所需疫苗接种百分比作为疾病传染性的函数。

如前一个练习一样，确保你的数据集足够大。

备注 40 你上面使用的屏幕输出对于小问题进行合理性检查是好的。然而，对于现实的模拟，你必须思考什么是现实的种群规模。如果你的大学校园是一个随机的人可能互相遇见的种群，那么建模的种群规模应该是多少？你所在的城市呢？

同样地，如果你测试不同的疫苗接种率，你使用什么粒度？随着 *5%* 或 *10%* 的增加，你可以将所有结果打印到屏幕上，但你可能会错过一些东西。不要害怕生成大量数据并将它们直接输入到绘图程序。

49.3.5 突变

新冠疫情表明了原始病毒突变的重要性。接下来，你可以在你的项目中包含突变。我们这样建模：

- 每隔一段时间，病毒就会变异成新的变种。
- 从一种变种中康复的人仍然容易感染其他变种。
- 为了简化，假设每个变种使一个人生病的时间相同，并且
- 疫苗接种是全有或全无的：一剂疫苗足以保护所有变种；
- 另一方面，从一种变种中康复并不能保护免受其他变种感染。

从实现的角度来说，我们这样建模。首先，我们需要一个 *疾病* 类，这样我们就可以用明确的病毒感染一个人；

```
// /infect_lib.hpp
void touch(const Person&,<long int ps>=0 );void
infect(const Disease& );
```

一个 `Disease` 对象现在携带了如传播几率或一个人在生病期间停留多长时间等信息。建模变异有点棘手。你可以这样做：

- 有一个用于新病毒变异的全局 `variants` 计数器，以及一个用于传播的全局 `transmissions` 计数器。
- 每当一个人感染另一个人时，新感染的人会得到一个新的 `Disease` 对象，其中包含当前的变异版本，并且传播计数器会更新。
- 有一个参数决定了在传播多少次后疾病会变异。如果发生变异，全局 `variants` 计数器会更新，从那时起，每次感染都会使用新的变异版本。（注意：这并不太现实。你可以自由地提出一个更好的模型。）
- 每个 *人* 对象都有一个他们康复过的变异株向量；从一种变异株康复只会让他们对该特定变异株免疫，而不是对其他变异株。

练习 49.9. 给你的模型添加变异。尝试变异率：随着变异率的增加，疾病应该在种群中持续更长时间。你之前观察到的与疫苗接种率的关系是否改变？

49.3.6 没有疫苗的疾病：埃博拉和新冠病毒

本节是可选的，用于加分

迄今为止，该项目适用于有疫苗可用的疾病，例如麻疹、腮腺炎和风疹的 MMR 疫苗。如果没有疫苗，例如 埃博拉 和 新冠病毒 （截至本文写作时的情况），分析就会有所不同。

相反，您需要在代码中包含 “ 社交距离 ”： 人们不再与随机他人接触，而只与非常有限社交圈中的人接触。设计一个模型距离函数，并探索各种设置。

The difference between Ebola and Covid-19 is how long an infection can go unnoticed: the *incubation period*. With Ebola, infection is almost immediately apparent, so such people are removed from the general population and treated in a hospital. For Covid-19, a person can be infected, and infect others, for a number of days before they are sequestered from the population.

将此参数添加到您的模拟中，并探索疾病行为随其变化的情况。

49.4 伦理

传染病和疫苗接种的主题充满了伦理问题。主要问题是 发生在我身上的事情的机会非常小，为什么我不能稍微违反一下规则？ 这种推理通常应用于疫苗接种，其中人们出于某种原因拒绝接种疫苗。

探索这个问题以及其他你可能想到的问题：很明显，每个人都违反规则将会带来灾难性的后果，但如果只有少数人这样做呢?

49.5 奖励：测试

阅读 https://sinews.siam.org/Details-Page/the-mathematics-of-mass-testing-for-cov 你能测试那篇文章的假设吗?

49.6 项目报告和提交

49.6.1 Program files

在这个项目中，你已经编写了多个主程序，但一些代码在多个程序之间是共享的。将你的代码组织为每个主程序一个文件，以及一个包含类方法的单个 “ 库 ” 文件。

你可以用两种方法来做：

1. 您创建一个 “ 库 ” 文件，例如 infect_lib.cc，并且您的每个主程序都有一个行

   #include "infect"_lib.cc

   This is not the best solution, but it is acceptable for now.
2. 更好的解决方案要求您使用分离编译 来构建程序，并且您需要一个 头 文件。现在您将拥有 infect_lib.cc，它被单独编译，以及 infect_lib.h，它被包含在库文件和主程序中：

#include "infect"_lib.h

参见第 19.2.2 节了解更多信息。

提交所有源文件，并说明如何构建所有主要程序。你可以将这些说明放在一个具有描述性名称的文件中，例如 README 或 INSTALL，或者你可以使用一个 *makefile*。

49.6.2 写作

在写作中，描述你所进行的 ‘实验’ 以及你从中得出的结论。上面的练习为你提供了一些需要回答的问题。

对于每个主程序，包含一些示例输出，但请注意，这并不能替代用完整句子写出您的结论。

第 49.3.4 节中的练习要求您将程序行为作为一个或多个参数的函数进行探索。包含一个表格来报告您发现的行为。您可以使用 Matlab 或 Python 中的 Matplotlib （甚至 Excel ）来绘制您的数据，但这不是必需的。

49.7 附加：数学分析

SIR 模型也可以通过耦合差分或微分方程进行建模。

1. 在时间 S_i 时易感人群的数量 i 减少了一个分数

$$S_{i+1} = S_i(1 - \lambda_i\, dt)$$

其中 λ_i 是感染人数和一个反映会议次数和疾病传染性的常数的乘积。我们写：

$$S_{i+1} = S_i(1 - \lambda I_i\, dt)$$

2. 感染人数同样增加 $\lambda S_i I_i$，但也因康复者（或死亡者）而减少：

$$I_{i+1} = I_i(1 + \lambda S_i\, dt - \gamma\, dt).$$

3. 最后， ‘移除’ 的人数等于该最后一项：

$$R_{i+1} = R_i(1 + \gamma I_i).$$

Exercise 49.10. Code this scheme. What is the effect of varying dt?

练习 49.11。 为了使疾病成为流行病，新感染的人数必须大于康复的人数。也就是说，

$$\lambda S_i I_i - \gamma I_i > 0 \Leftrightarrow S_i > \gamma/\lambda.$$

你能在你的模拟中观察到这一点吗?

参数 γ 有一个简单的解释。假设一个人在康复前会生病 δ 天。如果 I_t 相对稳定，这意味着每天感染的人数与康复的人数相同，因此每天有 $1/\delta$ 比例的人康复。因此，γ 是给定个体感染持续时间的倒数。

49. 传染病模拟

第 50 章

Google PageRank

50.1 基本概念

我们将模拟互联网。特别是，我们将通过 *Pagerank* 算法模拟 *Google* 确定网页重要性的方法。

让我们从一些基本类开始：

- 一个 `Page` 包含一些信息，例如其标题和在 Google 数据中心的全局编号。它还包含一个链接集合。
- 我们用指向 `页面` 的指针来表示一个链接。理论上，我们可以有一个 `链接` 类，其中包含进一步的信息，例如被点击的概率或被点击的次数，但暂时一个指针就足够了。
- 最终我们希望有一个 `网络` 类，其中包含多个页面及其链接。网络对象最终还将包含页面的相对重要性等信息。

这个应用程序是使用指针的一个自然场景。当你在网页上点击一个链接时，你会从浏览器中的一个页面切换到另一个页面。你可以通过指向一个页面的指针来实现这一点，点击会更新该指针的值。

练习 50.1。 创建一个 `页面` 类，初始时只包含页面的名称。编写一个方法来显示页面。由于我们将频繁使用指针，让这个成为测试的预期代码：

```
// /web2.cpp
auto homepage= make_shared<页面>("我的主页");cout <<"主
页目前还没有链接："<< '\n';cout <<homepage->as_
string() << '\n';
```

接下来，向页面添加链接。链接是指向另一个页面的指针，由于可能有任意数量的链接，因此您需要一个 `向量` 来存储它们。编写一个方法 `click` 来跟随链接。预期代码：

```
// /web2.cpp
auto utexas= make_shared<Page>("University Home Page");
homepage->add_link(utexas); auto searchpage= make_shared<Page>("
google"); homepage->add_link(searchpage); cout <<homepage->as_
string() << '\n';
```

练习 50.2。 为你的主页添加更多链接。为 `Page` 类编写一个 `random_click` 方法。预期代码：

```
// /web2.cppfor(int iclick=0;iclick<20;++iclick{v20}) {auto
newpage =homepage{v31}->random_click();cout << "To: "<<
newpage{v49}->as{v52}string{v55}() << '\n';}
```

如何处理没有链接的页面？

50.2 Clicking around

练习 50.3。 现在创建一个 `Web` 类，它首先包含一组（技术上：一个 向量 ）页面。或者更准确地说：指向页面的指针。因为我们不想手动构建整个互联网，让我们有一个 `create_random_links` 方法，它创建指向随机页面的随机数量的链接。预期代码：

```
// /web2.cpp
    Web 互联网（网络规模）；互联网。创建 _ 随机 _
    链接（平均链接）；
```

现在我们可以开始我们的模拟。编写一个方法 `Web::random_` 随机游走，它接受一个页面和游走长度，并模拟在当前页面上随机点击该次数的结果。（当前页面。不是起始页面。）

让我们开始朝着 PageRank 努力。首先，我们看看是否有比其他页面更受欢迎的页面。你可以通过在每个页面上进行一次随机游走来做到这一点。或者也许几次。

练习 50.4。 除了你的互联网规模之外，你的测试还有哪些其他设计参数？你能给出一个粗略估计它们的影响吗？

练习 50.5。 你的第一个模拟是在每个页面上多次开始，并计算它最终会落在哪里。预期代码：

```
//web2.cpp vector<int> landing_counts(internet.number_of_pages(),0); for ( auto
page : internet.all_pages() ) { for (int iwalk=0; iwalk<5; ++iwalk) { auto endpage=
internet.random_walk(page,2*avglinks,tracing); landing_counts.at(endpage->global_
ID())++; } }
```

显示结果并进行分析。你可能会发现你在某些页面上访问次数过多。发生了什么？修复它。

50.3 图算法

有许多算法依赖于逐步遍历网络。例如，任何图都可以连通。您通过

- 取一个任意的顶点 v。创建一个 ‘可达集合’ $R \leftarrow \{v\}$。
- 现在看看您可以从您的可达集合中到达哪里：

$$\forall_{v \in V} \forall_{w \text{ neighbour of } v} : R \leftarrow R \cup \{w\}$$

- 重复上一步，直到 R 不再改变。

在此算法完成后， R 是否等于您的顶点集？如果是，则您的图称为（完全）连通的。如果不是，则您的图有多个组件。

练习 50.6。 编写上述算法，跟踪到达每个顶点 w 所需的步骤数。这是单源最短路径 算法（对于无权图）。

直径被定义为最大最短路径。编写这个。

50.4 页面排名

PageRank 算法现在询问，如果您随机点击，最终停留在某个页面的可能性分布是什么。我们通过概率分布来计算这个：我们为每个页面分配一个概率，使得所有概率之和为 1。我们从随机分布开始：

代码：

```
// /web2.cpp
 概率分布

random_state(internet.number_of_pages());
 随机_状态。设置_随机（）；
cout<< "初始分布:                    " <<
随机_状态。作为字符串 () '\n' ;
```

输出
[google] pdfsetup:

```
初始分布 :
      0:0.00, 1:0.02, 2:0.07,
      3:0.05, 4:0.06, 5:0.08
      6:0.04, 7:0.04, 8:0.04,
      9:0.01, 10:0.07, 11:0.05,
      12:0.01, 13:0.04,
      14:0.08, 15:0.06,
      16:0.10, 17:0.06,
      18:0.11, 19:0.01,
```

练习 50.7。 实现一个类 `ProbabilityDistribution`，它存储一个浮点数向量。编写方法：

- 访问特定元素，
- 将整个分布设置为随机，并且
- 归一化，使得概率之和为 1。
- 一种将分布渲染为字符串的方法也可能很有用。

接下来我们需要一个方法，给定一个概率分布，给出对应于执行一次点击的新分布。（这与马尔可夫链有关；参见 HPC 书籍 [11]，第 10.2.1 节。

练习 50.8. 编写方法

```
概率分布 Web::全局点击（ 概率分布 当前状态）;
```

Test it by

以一个在恰好一页上非零的分布开始;
- 打印对应一次点击的新分布;
- 对几页执行此操作并直观检查结果。

然后从随机分布开始并运行几次迭代。这个过程收敛得有多快？将结果与上面的随机游走练习进行比较。

练习 50.9. 在随机游走练习中，你必须处理一些页面没有出站链接的情况。在这种情况下，你转换到了一个随机页面。在全局点击方法中缺乏这种机制。想出一种方法来整合这一点。

让我们模拟一些简单的 “ 搜索引擎优化 ” 技巧。

练习 50.10. 添加一个你人为使其看起来很重要的页面：添加一些都链接到此页面的页面，但没有页面链接到它们。（由于随机点击，它们有时仍然会被访问。）

计算人为夸大页面的排名。你是否成功使谷歌将此页面排名靠前？你添加了多少链接?

示例输出:

```
互联网有 5000 页 Top 得分 : 109:0.0013, 3179:0.0012, 4655:0.0010, 3465:0.0009, 4298:0.0008
带有虚假页面 : 互联网有 5051 页 Top 得分 : 109:0.0013, 3179:0.0012, 4655:0.0010,
5050:0.0010, 4298:0.0008 在 4 处炒作页面得分
```

50.5 图和线性代数

概率分布本质上是一个向量。你也可以将网络表示为一个矩阵 W with$w_{ij} = 1$，如果页面 i 链接到页面 j。在这些术语中，你如何解释 全局点击 方法?

练习 50.11. 添加 `Web` 对象的矩阵表示，并重新实现 全局点击方法。测试正确性。

进行时间比较。

你上面进行的迭代以找到稳定概率分布对应于线性代数中的 “ 幂方法 ”。查阅 Perron-Frobenius 理论，看看它对页面排名意味着什么。

第 51 章

重新划分选区

在这个项目中，你可以探索 “ 杰利蝾螈 ” （ gerrymandering ），即通过策略性地划分选区，使少数人口获得多数选区的现象。1

51.1 基本概念

我们正在处理以下概念：

一个州被划分为选区，选区信息是已知的。根据人口普查数据（收入、种族、中位年龄），通常可以很好地猜测这样一个选区的整体投票情况。

- 存在一个固定的国会选区数量，每个选区由普查选区组成。国会选区不是随机的集合：普查选区必须是连续的。
- 每隔几年，为了适应人口变化，选区边界会被重新绘制。这被称为选区重划。

选区重划的方式有很大的自由度：通过移动（国会）选区的边界，可以使一个在整体上占少数的人口获得多数选区。这被称为杰利蝾螈。

背景阅读请参见 https://redistrictingonline.org/。

为了进行小规模的计算机模拟杰利蝾螈，我们做出了一些简化的假设。

- 首先，我们放弃了普查选区：我们假设一个选区直接由选民组成，并且我们知道他们的党派归属。在实践中，人们依赖于代理指标（如收入和教育水平）来预测党派归属。
- 接下来，我们假设一个一维状态。这足以构建能够揭示问题本质的示例：考虑一个有五个选民的状态，我们将他们的选票指定为 AAABB。将他们分配到三个（连续的）选区可以表示为 AAA|B|B，其中有一个 ‘A’ 选区和两个 ‘B’ 选区。
- 我们还允许选区可以是任何正大小，只要选区的数量是固定的。

1. 这个项目显然基于北美政治体系。希望这里的解释足够清晰。如果您知道其他国家也有类似的体系，请联系作者。

51.2 基本功能

51.2.1 投票者

我们放弃人口普查选区，用投票者来表示一切，我们假设投票行为是已知的。因此，我们需要一个投票者类，该类将记录投票者 ID 和政党归属。我们假设有两个政党，并留出未决定的选择。

练习 51.1。 实现一个投票者类。你可以例如让 ± 1 代表 A/B，0 代表未决定。

```
// /linear.cppcout << "投票者 5 是积极的: "<<'
\n'; 投票者 nr5(5,+1);cout << nr5.print() <<'
\n';

// /linear.cpp
    cout << "Voter 6 is negative:"<<'\n';Voter
    nr6(6,-1);cout << nr6.{v20}print() <<'\n';


// /linear.cpp
    cout << "Voter 7 is weird:"<< ' \n' ;Voter
    nr7(7,3);cout << nr7.{style id='22'}() <<' \n' ;
```

51.2.2 种群

练习 51.2. 实现一个 District 类，该类对选民群体进行建模。

- 你可能希望从一个选民或一组选民中创建一个选区。拥有一个接受字符串表示的构造函数会很有用。
- 编写方法 majority 来给出确切的多数或少数，以及 lean 来评估该选区整体是否应计为 A 党或 B 党。
- 编写一个子方法来创建子集 .

```
District District::sub(int first,int last);
```

- 为了调试和报告，有一个方法可能是个好主意

```
string District::print();
```

代码：

```
// /linear.c
        ppcout<< "使用一个 B 创建区域 "
      " 选民 "<<'\n';
      选民编号 5(5,+1);
      区域九 ( nr5);
      cout<< ".. size:"      " <<九. 个 () <<
      ' \n' ;
      cout<< ".   . lean: " << nine.lean() <<
      \n;/*
          ... */
      cout << "Making district ABA"<<' \n' ;
      第九区 (          vector<Voter>
                        {{1,-1},{2,+1},{3,-1}}
      } );
      cout << ".. size:   " <<九. 大小 () <<
      ' \n' ;
      cout << ".. lean: " << nine.lean() <<
      ' \n' ;
```

输出
[gerry] 区：

创建一个有 *B* 选民的区
.. *size: 1*
.. *lean*: -1

生成区域 *ABA*
.. *size: 3*
.. *lean*: -1

51.3. 实现一个 Population 类，它将最初模拟一个完整的状态。

代码：

```
1// /linear.cpp2 stringpns(
                        "-++--" );
3     为一些 pns 赋值;
4        cout<< "Population from string"pns<< ' \n' ;" <<

5     cout<< ".. size:  《C++17/fortran2008 科学编程艺
      ' \n' ;           术，第 3 卷》51.2.2 种群
6     cout << ".. lean:  《C++17/fortran2008 科学编程艺
      ' \n' ;           术》第 3 卷 51.2.2 种群
7     种群组 = 一些。子 (1,3) ; t << " b l ti 1 3"<<
8     cou      su  popu a  on  --      '\n';
9     cout<< ".. '\"  size: " << group.size()
      <<   n ;
10    cout<<    .. lean: "<< group.lean()
      << '\n';
```

输出
[gerry] 种群：

来自 字符串 -++--
.. 大小：5
.. 精简：-1li
su popu at on 1--3
.. size: 2
.. 精简：1

除了显式创建之外，还编写一个指定人数和多数派的构造函数：

```
// /linear.cpp
  Population( int population_size,int majority,bool trace=false )
```

使用随机数生成器来实现精确指示的多数派。

51.2.3 区域划分

下一个复杂程度是拥有一个区域集。由于我们将逐步创建它，我们需要一些方法来扩展它。

练习 51.4. 编写一个类 Districting，用于存储一个 District 对象的向量。编写 size 和 lean 方法：

代码：

```
// /linear.cpp
    cout<< "创建单个选民
    populat \n B" <<    n ;
    人口 people( v          ector<选民>{
    选民(0,+1) } );cout<< "
                ·· size:    " << people.size()
    << '\n';
    cout << ". . lean: " << people.lean()
    << ' \n' ;

    区域划分 gerry;t<< "St t ith

    cou          ar  w     empty
    区域划分 :"<<'\n';
    cout<< ".. 区域数量:                    " <<
    gerry.size() << '\n';
```

输出

```
[gerry] gerryempty:

Making single voter population B
.. size: 1
.. 精简 : 1 与
Star  w    cm t  districting:
    py.. 区域数量 :0
```

练习 51.5. 编写方法以扩展区域 :

```
// /linear.cpp
    cout << "添加一个 B 选民 :"<< '\n'; gerry= gerry.extend_with_ 新 _ 区域 (
    people.at(0) ); cout << ".. 区域数量 : " << gerry.size() <<'\n'; cout << ".. 
    lean: " <<gerry.lean() << '\n'; cout << " 添加 A A:" <<'\n'; gerry=
    gerry.extend_ 最后一个 _ 区域 ( Voter(1,-1) ); gerry= gerry.extend_ 最后一
    个 _ 区域 ( Voter(2,-1) ); cout << ".. 区域数量 : "<< gerry.size() << '\n'; cout
    << ".. lean: " <<gerry.lean() << '\n';


    cout << "添加两个 B 区域 :" <<'\n'; gerry= gerry.extend_with_ 新 _ 区域 (
    Voter(3,+1) ); gerry= gerry.extend_with_ 新 _ 区域 ( Voter(4,+1) ); cout <<
    ".. 区域数量 : "<< gerry.size() << '\n'; cout << ".. lean: " <<gerry.lean() << '
    \n';
```

51.3 方法

现在我们需要一个对人口进行划分的方法：

```
划分人口 :: 少数民族 _ 规则 ( int ndistricts );
```

与其生成所有可能的人口划分方案，我们采用增量方法（这与称为动态规划的解决方案策略相关）：

- 基本问题是如何在 n 个选区中最佳地划分人口；
- 我们通过递归方式首先解决一个子人口在 $n-1$ 个选区中的划分问题，
- 并扩展剩余人口为一个区。

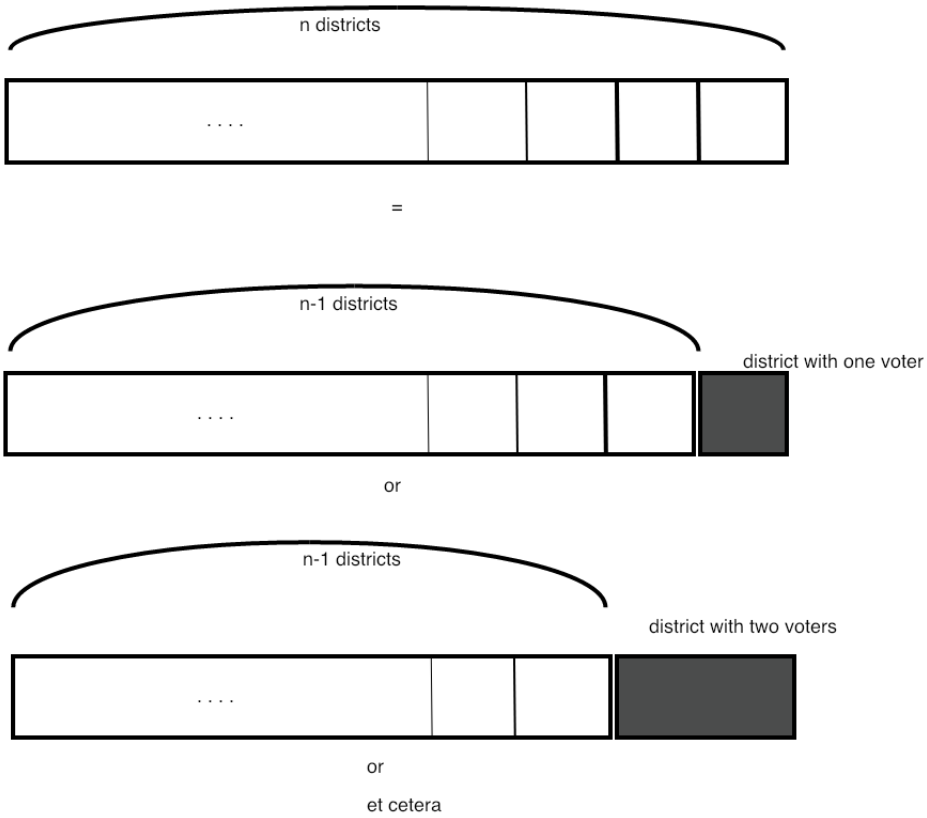


Figure 51.1: Multiple ways of splitting a population

这意味着你需要考虑将 ‘剩余’ 人口全部迁入一个区的方法，这意味着你将有一个循环来处理所有的人口分割方式，这些方式位于递归之外；参见图表 51.1。

- 对于所有 $p = 0, \ldots n-1$ 考虑将州分割为 $0, \ldots, p-1$ 和 $p, \ldots, n-1$.
- 使用第一组的最优区域划分，并将最后一组划分为一个单一区域。
- 保持对所有 p 值都给出最强少数规则的区域划分。

你现在可以理解上述简单示例：

AAABB => AAA |B |B

Exercise51.6. 实现上述方案。

Code:

```
// /linear.cpp
    Population five("+++--");
    cout << "Redistricting population: "
    << '\n'
         << five.print() << '\n';
    cout << ".. majority rule: "
         << five.rule() << '\n';
    int ndistricts{3};
    auto gerry =
    five.minority_rules(ndistricts);
    cout << gerry.print() << '\n';
    cout << ".. minority rule: "
         << gerry.rule() << '\n';
```

Output

```
[gerry] district5:
Redistricting population:
[0:+,1:+,2:+,3:-,4:-,]
.. majority rule: 1
[3[0:+,1:+,2:+,],[3:-,],[4:-,],]
.. minority rule: -1
```

注意：上面给出的 p 范围并不完全正确：例如，初始人口需要足够大以容纳 $n-1$ 名选民。

练习 51.7。 测试多种人口规模；在仍然给予 A 党多数的情况下，你可以给 B 党多少多数。

51.4 效率：动态规划

如果你思考一下刚刚实现的算法，你可能会注意到初始部分的划分被重新计算了很多次。优化这种策略的方法称为备忘录化。

练习 51.8。 通过存储和重用初始子人口的结果来改进你的实现。

在某种意义上，我们反向解决了程序：我们考虑将最后若干选民划分为一个选区，然后递归地为前若干选民解决一个更小的问题。但在那个过程中，我们决定如何为前 1 名选民、前 2 名选民、前 3 名选民等等分配选区。实际上，对于超过一名选民的情况，比如五名选民，我们找到了最佳少数派规则的结果，即将这五名选民分配到一个、两个、三个、四个选区。

计算 ‘最佳’ 选区划分的过程，被称为 动态规划。这里的根本假设是，你可以使用中间结果并将其扩展，而无需重新考虑早期的问题。

例如，考虑你已经考虑了将十名选民划分为最多五个选区的情况。现在，十一名选民和五个选区的多数是

- 十名选民和五个选区，以及新选民被添加到最后一个选区；或
- 十个选民和四个选区，新选民将成为一个新的选区。

练习 51.9。 编写一个动态规划解决方案来解决重新划分选区的红问题。

51.5 扩展

到目前为止，该项目有几个简化的假设。

- 国会选区需要大约相同的大小。你能对大小之间的比例设限吗？少数派还能获得多数吗？

练习 51.10。 最大的假设当然是我们考虑了一个一维状态。有两个维度，你就有更多的自由度来塑造选区。实现一个二维方案；使用一个完全平方的状态，其中人口普查选区形成一个规则的网格。将国会选区的形状限制为凸形。

The 效率差距 是一个衡量一个州选区 ‘公平性’ 的指标。

练习 51.11。 查找效率差距（和 ‘浪费的选票’ ）的定义，并在你的代码中实现它。

51.6 伦理

重新划分选区的活动旨在给予人们公平的代表。在其退化的形式 —— 杰利蝾螈中，这个公平的概念被违反了，因为明确的目标是给予少数派多数选票。探索如何纠正这种不公平。

在你上面的探索中，选民的唯一特征是他们支持 A 党或 B 党的偏好。然而，在实践中，选民可以被看作是社区的一部分。《投票权法案》关注 ‘少数派选票稀释’ 。你能举例说明无色盲的选区划分会对某些社区产生负面影响吗？

51. 重新划分选区

第 52 章

亚马逊配送卡车调度

本节包含一系列练习，逐步构建起配送卡车调度的模拟。

52.1 问题陈述

调度配送卡车的路线是一个研究已久的问题。例如，最小化卡车需要行驶的总距离对应于旅行商问题 *(TSP)*。然而，在亚马逊配送卡车调度的情况下，这个问题有一些新的方面：

- 客户承诺在交付可以进行的日期窗口内。因此，卡车可以将地点列表拆分成子列表，总距离比一次遍历列表更短。
- 不过，亚马逊 *Prime* 客户需要确保他们的送货次日到达。

52.2 编码基础知识

在我们尝试找到最佳路线之前，让我们先建立基础知识，以便至少有一个路线。

52.2.1 地址列表

你可能需要一个类 `Address` 来描述一个需要送货的房屋位置。

- 为了简化，让我们给一个房屋 (i, j) 坐标。
- 我们可能需要一个`距离` 函数来计算两个地址之间的距离。我们可以假设在两栋房子之间可以直线旅行，或者假设城市是构建在一个网格上的，并且你可以应用所谓的曼哈顿距离。
- 地址可能还需要一个字段来记录最后可能的配送日期。

练习 52.1. 编写一个具有上述功能的 `Address` 类，并进行测试。

52. 亚马逊配送卡车调度

代码：

```
// /route.C
        pp 地址一 (1.,1.),

          两 (2.,2.);<< "
Dit
   cerr          s ance: "
          << one. 距离(two)'\ '

          <<    n ;
```

输出

[亚马逊] 地址：

地址
距离 : 1.41421
.. 地址
地址 1 应该离得最近
仓库 . 检查 :1

从仓库到仓库:
(0,0) (2,0) (1,0) (3,0)
(0,0)
有长度 8: 8
贪婪调度： (0,0)(1,0)
(2,0) (3,0) (0,0)
应具有长度 6:6

Square5
按顺序旅行： 24.1421
平方根 : (0,0) (0,5)
(5,5) (5,0) (0,0)
长度为 20
.. *square5*

原始列表 : (0,0) (-2,0)
(-1,0) (1,0) (2,0) (0,0)
length=8
翻转中间两个地址：
(0,0) (-2,0) (1,0) (-1,0)
(2,0) (0,0)
长度 =12
更好 : (0,0) (1,0) (-2,0)
(-1,0) (2,0) (0,0)
长度 =10

百户
按顺序的路线有长度
25852.6
基于简单列举的 *TSP* 有
长度： 2751.99 超过朴素
25852.6
单路径长度： 2078.43
.. 接受新路径与
长度 2076.65
最终路线长度 2076.65
比初始 2078.43 长
TSP 路线长度 1899.4
比初始 2078.43 长

两条路线
Route1 :(0 0) (2 0) (3 2) ,
(2,3) (0,2) (0,0)
route2: (0,0) (3,1) (2,1)
(1,2) (1,3) (0,0)
total length 19.6251
start with 9.88635,9.73877
Pass 0
.. *down to* 9.81256,8.57649
Pass 1
Pass 2
Pass 3
Pass 4
TSP Route1: (0,0) (3,1) (3,2)
(2,3) (0,2) (0,0)
route2: (0,0) (2,0) (2,1)
(1,2) (1,3) (0,0)
total length 18.389

接下来我们需要一个类 *AddressList*，它包含地址列表。

练习 52.2. 实现一个类 *AddressList*；它可能需要以下方法：

- *add_* 地址用于构建列表；
- *length* 用于给出按顺序访问所有地址所需的距离；
- *index_* 最近的 _ 到，它给出列表中离另一个地址最近的地址，该地址可能不在列表中。

52.2.2 添加 depot

接下来，我们建模路线需要从 depot 开始和结束这一事实，我们将其任意放置在坐标 (0,0)。我们可以构建一个 *AddressList*，使其第一个和最后一个元素都是 depot，但这可能会引发问题：

- 如果我们重新排序列表以最小化驾驶距离，第一个和最后一个元素可能不会保持在原位。
- 我们可能希望列表中的元素是唯一的：地址重复意味着在同一个地址有两个配送，所以 *add_address* 方法会检查地址是否已经在列表中。

我们可以通过创建一个类 *Route* 来解决这个问题，该类继承自 *AddressList*，但其方法会保持列表的第一个和最后一个元素在原位。

52.2.3 贪心构建路线

接下来我们需要构建一条路线。与其解决完整的 TSP 问题，我们首先采用一个贪婪搜索 策略：

> 给定一个点，通过某种局部最优性测试（例如最短距离）找到下一个点。永远不要回头重新访问你已经构建的路线。

这种策略可能会带来改进，但很可能不会给出最优路线。

Let's write a method

```
路线 :: 路线贪婪 _ 路线 ();
```

构建一个新的地址列表，其中包含相同的地址，但排列方式使得旅行距离更短。

练习 52.3. 为 *AddressList* 类编写贪婪 _ 路线 方法。

1. 假设路线从仓库开始，其位置位于 (0,0)。然后通过以下方式逐步构建一个新的列表：
2. 维持一个地址 变量 *we_* 是 _ 在这里 当前位置； 3. 重复查找离 *we_* 是 _ 在这里最近的地址。

将其扩展为路线 类的方法，通过处理不包含最后一个元素的子向量来工作。

在这个示例上测试它：

Code:

```
1 // /route.cpp
2     路径配送;
3     配送 .add_ 地址 ( Address(0,5)
      );
4     配送 .add_ 地址 ( Address(5,0)
      );
5     deliveries.add_address( Address(5,5)
      );
6     cerr << "Travel in order: " <<
      配送 . 长度 () <<'\n';
7     assert deliveries.size();
8     自动 route =
      交付 . 贪婪 _ 路线 ();
9     assert(route.size(==5);
10    自动 . 路径长度 ();
11    cerr << "Square route: " <<
      路线 . 作为 _ 字符串 ()
12          << "\n 有长度 " <<len <<
      \n;
```

输出

[亚马逊] 方块 5:

```
Travel in order: 24.1421
方形路线 :          (0,0) (0,5)
      (5,5) (5,0) (0,0)
 具有长度 20
```

重新组织列表可以通过多种方式完成。

首先，您可以尝试就地修改。这会遇到一个问题，即您可能想要保留原始列表；此外，虽然交换两个元素可以通过使用插入和删除方法来完成，但更复杂的操作则比较棘手。

- 或者，您可以逐步构建一个新列表。现在的主要问题是跟踪原始列表中哪些元素已被处理。您可以通过为每个地址提供一个布尔字段 `done` 来做到这一点，但您也可以复制输入列表，并删除已处理的元素。为此，请研究 `erase` 方法用于 `vector` 对象。

52.3 优化路径

上述每次寻找最近地址的建议被称为 贪婪搜索 策略。它不会给你 TSP 的最优解。找到 TSP 的最优解很难编程 —— 你可以递归地做 —— 并且随着地址数量的增加需要花费大量时间。事实上，TSP 可能是 *NP-hard* 问题类别中最著名的问题之一，这些问题通常被认为具有比问题规模多项式增长更快的运行时间。

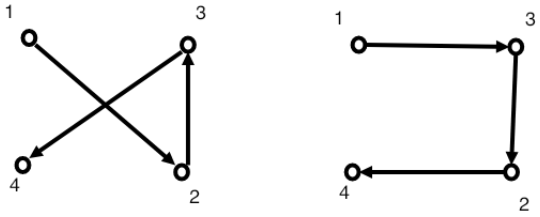


图 52.1: ‘opt2’ 思想的插图，即反转路径的一部分

然而，你可以启发式地近似解。一种方法，Kernighan-Lin 算法 [19], 基于 *opt2* 思想：如果你有一个 ‘交叉’ 的路径，你可以通过反转

部分内容。图 52.1 显示路径 $1--2--3--4$ 可以通过反转部分内容来缩短，得到 $1--3--2--4$。由于识别路径自交的位置可能很困难，甚至对于没有笛卡尔坐标关联的图来说是不可能的，我们采用一种尝试所有可能反转的方案：

> 对于路径上的所有节点 m<n `[1..N]`: 从`[1..m-1] + [m--n].reversed + [n+1..N]`创建一条新路径。如果新路径更短，则保留它

练习 52.4。 编写 opt2heuristic 的代码：写一个方法来反转路径的一部分，并编写一个循环来尝试多个起始和结束点。在一些简单的测试用例上运行它，以证明你的代码按预期工作。

让我们探讨复杂度问题。（有关复杂度计算的介绍，请参阅 HPC 书 [11], 第 15 节。）旅行商问题（TSP）是 NP 完全问题中的一类问题，非常非正式地说，这意味着除了尝试所有可能性之外没有更好的解决方案。

练习 52.5. 使用 opt2 的启发式解的运行时复杂度是多少？考虑所有可能性来寻找最佳解的运行时复杂度会是多少？对两种策略在一些问题规模上的运行时间做一个非常粗略的估计：$N = 10$, 100, 1000, ... 。

练习 52.6. 之前你已经编写了贪婪启发式算法。比较从 opt2 启发式算法中获得的改进，分别从给定的地址列表开始，以及从贪婪遍历它开始。

为了现实性，你会在你的路线上放置多少个地址？一个配送司机在非典型的一天会配送多少个地址？

52.4 多辆卡车

如果我们引入多辆配送卡车，我们得到 ‘多旅行商问题’ [5]。有了这个，我们可以模块多辆卡车在同一天外出配送的情况，或者一辆卡车在多天分摊配送的情况。目前我们不区分这两种情况。

第一个问题是如何分配地址。

1. 我们可以使用某种几何测试将列表分成两部分。这对于多辆卡车在同一天出勤的情况是一个很好的模型。然而，如果我们使用这个模型来表示同一辆卡车在多天出勤的情况，我们会忽略这样一个事实：第一天可以添加新的地址，这会打乱整齐分开的路线。 2. 因此，实际上假设所有卡车都得到一个本质上随机的地址列表可能是合理的。

我们能否将 opt2 启发式算法扩展到多路径的情况？参考一下图 52.2：与其修改一条路径，我们可以在一条路径和另一条路径之间切换位。在编写代码时，请考虑另一条路径可能会反向运行！这意味着根据第一条和第二条路径中的分割点，你知道有四个结果修改后的路径需要考虑。

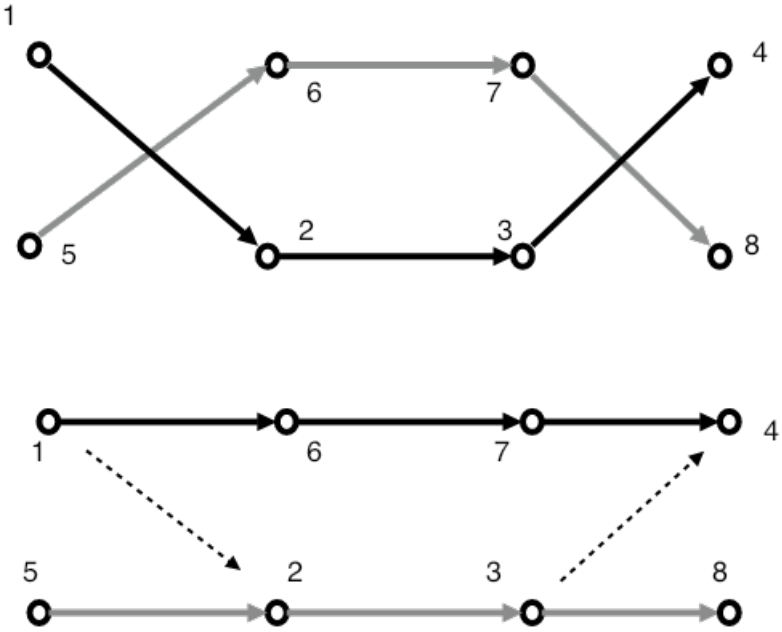


图 52.2：将 ‘opt2’ 思想扩展到多条路径

练习 52.7。 编写一个函数，同时优化两条路径，使用 opt2 启发式的多路径版本。测试用例参见 52.3。

在这里，你有相当大的自由度：

- 两个段的起点应该独立选择；
- 长度可以独立选择，但不必；最后
- 每个片段都可以反转。

更高的灵活性也意味着你的程序运行时间更长。这值得吗？做一些测试并报告结果。

根据上述描述，将会有很多代码重复。确保为各种操作引入函数和方法。

52.5 Amazon prime

在 52.4 节中，你假设包裹的交付日期无关紧要。这随着 *Amazon prime* 而改变，其中包裹必须在第二天保证交付。

练习 52.8。 探索一个场景，其中有两辆卡车，每辆卡车都有一些地址不能与其他路线交换。总距离会多长？尝试调整质地址和非质地址的比例。

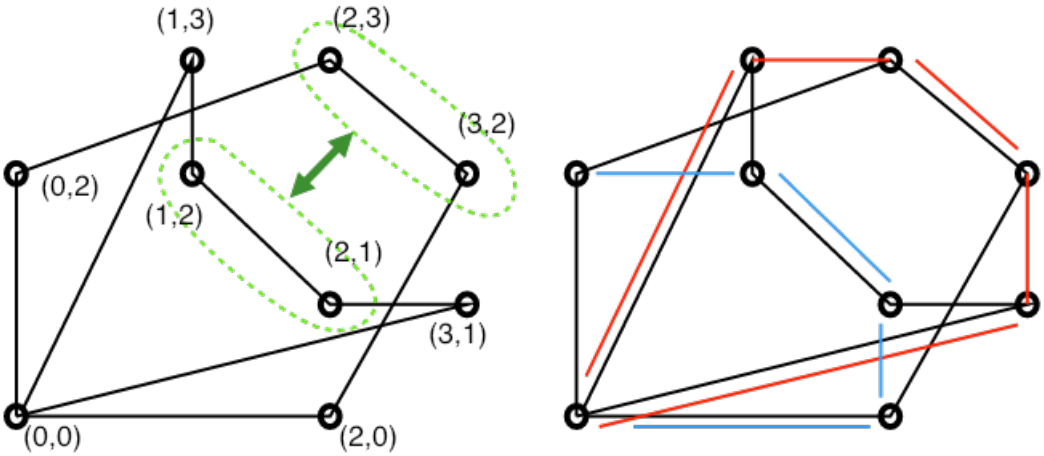


图 52.3：多路径测试用例

52.6 动态性

到目前为止，我们假设要配送的地址列表是给定的。这当然不正确：新的配送需要被持续调度。

练习 52.9。 实现一个场景，每天随机数量的新配送被添加到列表中。探索策略和设计选择。

52.7 伦理

人们有时批评亚马逊的劳动政策，包括对其司机的规定。你能从你在这方面的模拟中得出任何观察结果吗?

第 53 章

高性能线性代数

线性代数是计算科学的基础。涉及偏微分方程（ PDEs ）的应用最终归结为求解大型线性方程组；固体物理学涉及大型特征值系统。但在工程应用之外，线性代数也很重要：深度学习（ DL ）网络的主要计算部分涉及矩阵 - 矩阵乘法。

线性代数操作（如矩阵 - 矩阵乘法）可以用简单的方式编写代码。然而，这不会带来高性能。在这些练习中，你将探索高性能策略的基础。

注意 41 你将在本项目中开发的算法是 缓存无关编程的一个例子；参见 *HPC* 书籍 *[11]*，第 *6.1.12* 节。虽然这比简单算法的性能高得多，但要获得最高性能，需要更复杂的方法，这涉及大量的调优，以及一些 汇编编程 *[14]*。

53.1 数学预备知识

矩阵 - 矩阵乘法 $C \leftarrow A \cdot B$ 定义为

$$\forall_{ij} : c_{ij} \leftarrow \sum_k a_{ik} b_{kj}.$$

Straightforward code for this would be:

```
for (i=0; i<a.m; i++) for (j=0; j<b.n;
j++) s = 0; for (k=0; k<a.n;k++) s
+=a[i,k] * b[k,j]; c[i,j] =s;
```

然而，这不是执行此操作的唯一方法。循环可以交换顺序，总共提供六种实现。

练习 53.1。 编写其中一个置换算法并测试其正确性。如果上述参考算法可以被称为 ‘基于内积’，那么你会如何描述你的变体?

另一种实现基于块划分。让 A, B, C 在 2×2 块形式上分割:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21}, \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22}, \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21}, \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{53.1}$$

说服自己这实际上计算了相同的产品 $C = A \cdot B$。有关块算法的更多信息，请参阅 HPC 书籍 [11]，章节 5.3.6。

练习 53.2。 编写一个带有乘法例程的矩阵类:

Matrix Matrix::MatMult(Matrix other);

首先实现传统的矩阵 - 矩阵乘法，然后使其递归。对于递归算法，需要实现子矩阵处理：你需要提取子矩阵，并将子矩阵写回到周围的矩阵中。

53.2 矩阵存储

存储一个 $M \times N$ 矩阵最简单的方法是将其存储为长度为 MN 的数组。在这个数组中，我们可以决定按行连续存储，或按列存储。虽然这个决定显然对库来说具有实际重要性，但从性能角度来看，这没有区别。

注意 42 历史上，线性代数软件（如基本线性代数子程序（*BLAS*））使用列存储，这意味着元素 (i, j) 的位置计算为 $i + j \cdot M$（在本项目中，我们将始终使用零基索引，无论是代码还是数学表达式。）这种做法的原因源于 *BLAS* 在 *Fortran* 语言中的起源，*Fortran* 语言使用数组元素的列主序。另一方面，*C/C++* 语言中的静态数组（如 `x[5][6][7]`）使用行主序，其中元素 (i, j) 存储在位置 $j + i \cdot N$。

前面，你看到了块算法的概念，它需要提取子矩阵。为了提高效率，我们不想将元素复制到新数组中，因此我们希望子矩阵对应于子数组。
现在我们有一个问题：只有由一列序列组成的子矩阵是连续的。如果矩阵是更大矩阵的一个子块，那么元素位置公式 $i + j \cdot M\,(i, j)$ 就不再正确。

出于这个原因，线性代数软件通过三个参数来描述子矩阵 M, N, LDA，其中 ‘LDA’ 代表 ‘A<style id='13>’ 的 ‘主维度’（参见 BLAS [16],<style id='16> 和 Lapack [1]<style id='19>）。这在图 53.1。中进行了说明。

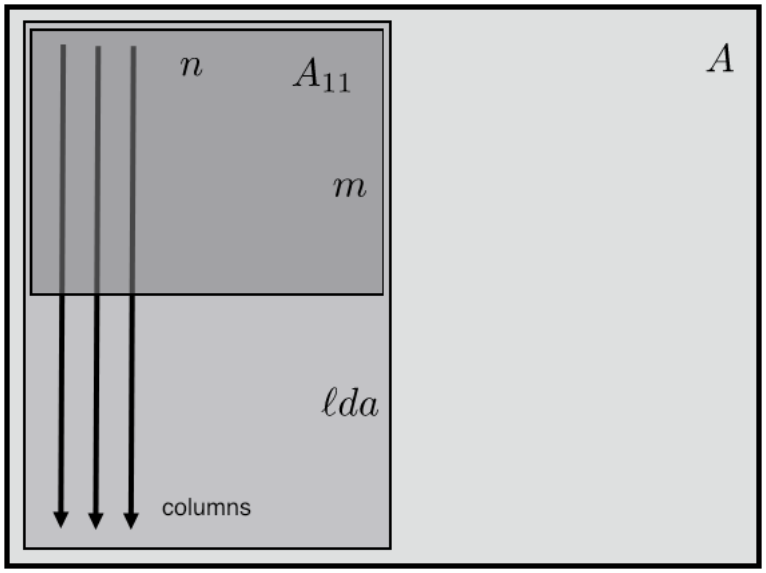


图 53.1：矩阵中的子矩阵， M, N， LDA 指示子矩阵的局部密度自适应

练习 53.3. 在 M, N, LDA 的术语下，(i, j) 元素的位置是什么?

从实现角度来看，我们也有一个问题。如果我们使用 `std::vector` 进行存储，则无法获取子数组，因为 C++ 坚持认为向量拥有自己的存储。解决方案是使用 `span` ； 章节 10.9.6。
我们可以有两种类型的矩阵：顶层矩阵存储一个 `vector<double>`，以及存储一个 `span<double>` 的子矩阵，但这会带来很多复杂性。这可以使用 `std::variant`(章节 24.6.4)，但让我们不这么做。

相反，让我们采用以下惯用法，其中我们在顶层创建一个向量，然后从其内存中创建矩阵。

```
// /lapack_alloc.cpp // M 、 LDA 、 N 的示例值 M
  = 2; LDA =M+2; N  = 3; // 创建一个包含数据的向量
vector<double>  one_data(LDA*N,1.); // 使用向量数
据创建矩阵 Matrix one(M,LDA,N,one_data.data());
```

（如果你之前没有用 C 语言编程，你需要习惯 <code>double</code><code>* </code> 机制。参见第 <code>10.10</code> 节。）

Exercise 53.4. Start implementing the `Matrix` class with a constructor

```
矩阵::矩阵(int m,int lda,int n,double*data)
```

and private data members:

```
// /lapack_alloc.cpp
private:
  int m,n,lda;
```

```
    span<double> data;
```

编写一个方法

```
double& Matrix::at(int i,int j);
```

作为安全访问元素的方式。

练习 53.5. 使用 C++23`mdspan` 结构执行上述练习。

让我们从简单的操作开始。

练习 53.6. 编写一个用于矩阵加法的 <code> 方法 </code>。在具有相同 M, N，但不同 LDA 的矩阵上测试它。

使用 `at` 方法对于 <code> 调试 </code> 很有用，但它效率不高。使用预处理程序（章节 21 ）引入替代方案:

```
#ifdef DEBUG
  c.at(i,j) += a.at(i,k) * b.at(k,j)
#else
  cdata[ /* expression with i,j */ ] += adata[ ... ] * bdata[ ... ]
#endif
```

你在那里直接访问数据，使用

```
//lapack_alloc.cppautoget_
double_data()
{double*adata;adata=
data.data();returnadata;}
```

练习 53.7. 实现这个。使用 `#define` 预处理程序指令用于优化的索引表达式。（见 <code>53.2.2</code> 节）

53.2.1 子矩阵

接下来我们需要支持构建实际的子矩阵。由于我们主要目标是 2×2 块形式的分解，因此只需编写四个方法:

```
矩阵左 (int j); 矩阵右
(int j); 矩阵上 (int i);
矩阵下 (int i);
```

其中，例如 `Left(5)` 给出包含 $j < 5$ 的列。

练习 53.8. 实现这些方法并测试它们。

53.3 乘法

You can now write a first multiplication routine, for instance with a prototype

```
void Matrix::MatMult( Matrix& other,Matrix& out );
```

或者，你也可以这样写

```
矩阵矩阵 ::MatMult( Matrix&other );
```

但我们希望将对象的创建 / 销毁数量保持在最低限度。

53.3.1 单级分块

接下来，编写

```
void Matrix::BlockedMatMult( Matrix& 其他,Matrix& out );
```

这使用了上述的 2×2 形式。

53.3.2 递归分块

最后一步是使分块递归。

练习 53.9。 编写一个方法

```
void RecursiveMatMult( Matrix& 其他 ,Matrix& out );
```

该

- 执行 2×2 块积，再次使用 `RecursiveMatMult` 进行块处理。
- 当块足够小时，使用常规 `MatMult` 积。

53.4 性能问题

如果你稍微尝试一下常规矩阵 - 矩阵乘法和递归矩阵 - 矩阵乘法之间的截止点，你会发现你可以获得很好的性能提升因子。为什么会这样?

矩阵 - 矩阵乘法是科学计算中的一个基本操作，人们已经投入了很多精力来优化它。一个有趣的事实是，在求和运算中，它几乎是可优化性最高的操作之一。其原因简而言之是，它涉及对 $O(N^3)$ 数据进行 $O(N^2)$ 次操作。这意味着，原则上每次获取的元素都会被多次使用，从而克服了 内存瓶颈 。

练习 53.10。 确保你的代码有计时器；参见章节 **??**。你能看到原始代码和优化代码之间的性能差异吗？你的处理器的峰值速度是多少，你接近它有多近？这个数字是否取决于你的矩阵的大小?

要理解与硬件相关的性能问题，你需要阅读一些资料。HPC 书籍 [11]，章节 1.3.5 解释了至关重要的 缓存 概念。

练习 53.11。 论证朴素矩阵 - 矩阵乘法实现不太可能实际重用数据。

解释为什么递归策略确实会导致数据重用。

在上面，你设置了一个切换点，用于在递归和常规乘法之间切换。

练习 53.12。 论证一旦乘法包含在缓存中，继续递归将不会带来太多好处。你的处理器的缓存大小是多少?

使用不同的切换点进行实验。你能将其与缓存大小联系起来吗?

53.4.1 并行（可选）

方程的四个子句 53.1 矩阵中的独立区域，因此它们可以在任何至少有四个核心的处理器上并行执行。 C 。

探索 OpenMP 库

以并行化 `BlockedMatMult`。

53.4.2 比较（可选）

最终问题是：你离最佳速度有多近？不幸的是，你还有很长的路要走。你可以按以下方式探索。

您的计算机很可能有一个优化的实现，可以通过：

```
#include <cblas.h>

cblas_dgemm(CblasColMajor, CblasNoTrans,
CblasNoTrans, m, other, n, alpha, adata, lda, bdata,
other, lda, beta, cdata, out, lda);
```

它计算 $C \leftarrow \alpha A \cdot B + \beta C$。

练习 53.13。 使用另一个 cpp 条件语句通过调用 `cblas_dgemm` 实现 `MatMult` 。您现在获得什么性能?

您可以看到您的递归实现比朴素实现更快，但几乎不像 CBlas 那样快。这是因为

- CBlas 的实现可能基于完全不同的策略 [14]，和
- 它可能涉及一定量的汇编编程。

第 54 章

巨大的垃圾带

本节包含一系列练习，逐步构建出一个 元胞自动机 模拟，模拟海洋中乌龟的移动、对它们致命的垃圾以及清理这些垃圾的船只。想了解更多信息： https://theoceancleanup.com/。

感谢 TACC 的 Ernesto Lima 提供了这个练习的想法和初始代码。

54.1 问题与模型解决方案

海洋中漂浮着大量的塑料，这对鱼类、乌龟和鲸类有害。在这里，你可以模拟它们之间的相互作用

- 塑料，随机分布；
- 游动的乌龟；它们繁殖缓慢，并且因摄入塑料而死亡；
- 清理海洋中的塑料垃圾的船只。

Th我们使用的模拟方法是元胞自动机：

- We have a grid of cells;
- Each cell has a ‘state’ associated with it, namely, it can contain a ship, a turtle, or plastic, or be empty; and
- On each next time step, the state of a cell is a simple function of the states of that cell and its immediate neighbors.

这个练习的目的是模拟多个时间步长，并探索参数之间的相互作用：乌龟将因多少垃圾而灭绝，需要多少船只才能保护乌龟。

54.2 程序设计

基本思路是创建一个包含海龟、垃圾和船只的 海洋 对象。您的模拟将使海洋经历多个时间步：

```
for(int t=0; t<时间_步; t++)海
洋.更新();
```

最终您的目的是研究海龟种群的演变：是否稳定？是否会灭绝？

虽然您可以针对此问题编写一个 ‘临时’ 解决方案，但您在现代化 / 干净的 C++ 编程技术方面的使用也将被部分评判。以下提供了一些建议。

54.2.1 网格更新

这里有一个需要注意的点。你能看出完全就地更新有什么问题吗:

```
for ( i )
  for ( j )
    cell(i,j) = f( cell(i,j), .... other cells ... );
```

?

54.3 测试

测试这个程序的正确性可能很复杂。你能做的最好的事情是尝试多种场景。为此，最好让你的程序输入灵活：使用 `cxxopts` 包 [8], 并从 shell 脚本中驱动你的程序。

这里有一份你可以测试的事项清单。

1. 从只有一些船开始；检查在 1000 个时间步之后你仍然有相同数量的船。 2. 同样地，对于乌龟；如果它们不繁殖也不死亡，检查它们的数量是否保持不变。 3. 只有船和垃圾，它们是否都被清理了？ 4. 只有乌龟和垃圾，它们是否都死光了?

要测试你的乌龟和船是否不会 ‘传送’ 到周围，但只会移动到相邻的单元格，这更困难。为此，使用视觉检查；参见第 54.3.1 节。

54.3.1 动画图形

这个程序的输出是可视化的理想候选。事实上，一些测试（ ‘确保乌龟不会传送’ ）除了通过查看输出外很难完成。首先，制作一个海洋网格的 ASCII 渲染，如图 54.1 所示。

最好有一些动画输出。然而，并非所有编程语言都同样容易生成视觉输出。在 C++ 中有非常强大的视频 / 图形库，但这些也很难使用。有一个更简单的方法。

对于此程序产生的简单输出，您可以制作一个简单的低成本动画。太阳下的每个终端 都支持 *VT100* 光标控制 [1]：您可以将某些魔法输出发送到您的屏幕以控制光标定位。

在每个时间步中，您会

1. 将光标发送到 home 位置，通过此魔法 o utput:

```
// /pacific.cpp
#include <cstdio>
    /* ... */
    // ESC [ i ; j H
    printf( "%c[0;0H",(char)27);
```

2. 显示您的网格，如图 54.1 所示； 3. 等待一小段时间；请参阅第 25.1.4 节。

1. https://vt100.net/docs/vt100-ug/chapter3.htm [1]

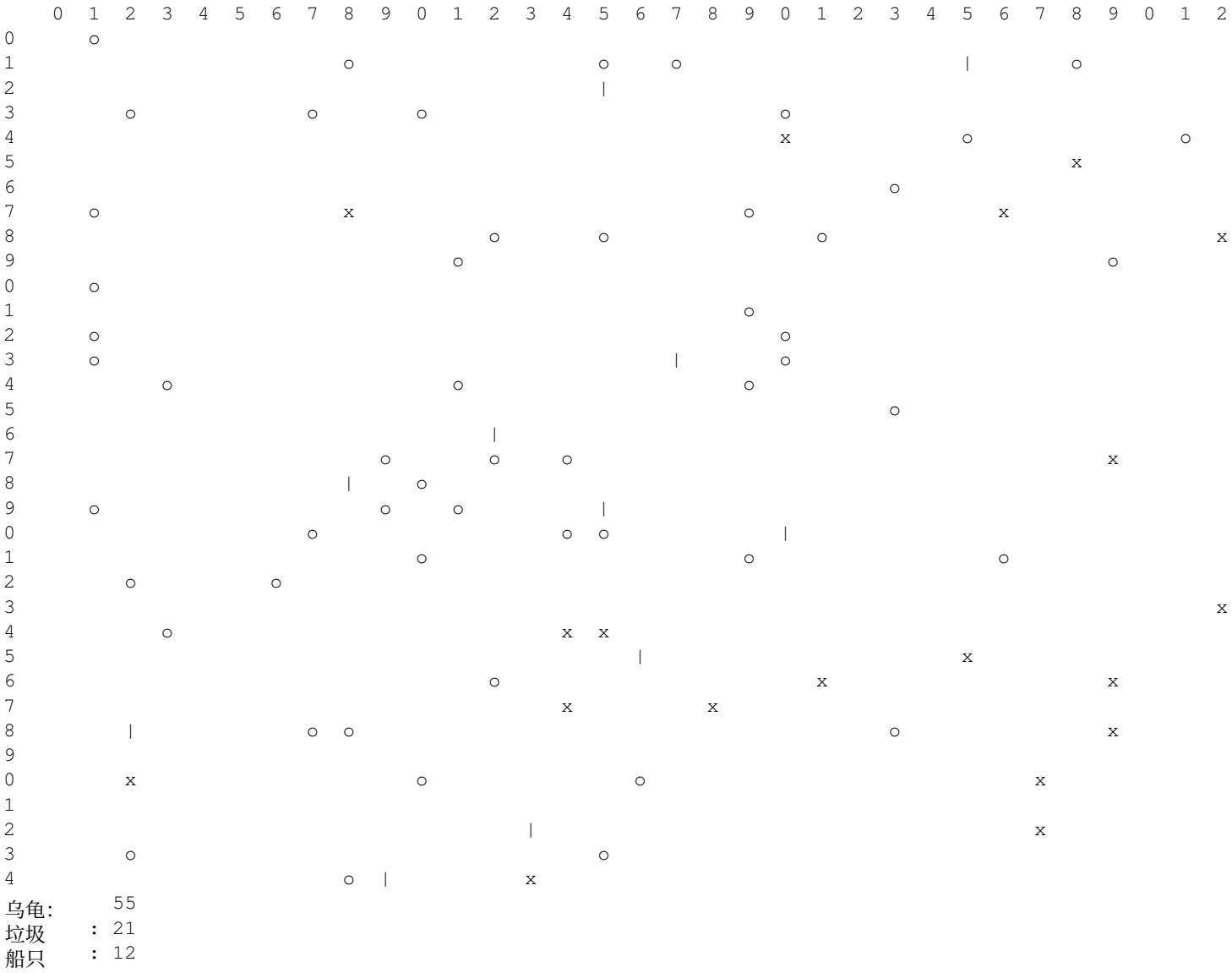


图 54.1：时间步的 Ascii 艺术打印输出

54.4 现代编程技术

54.4.1 面向对象编程

虽然只有一个海洋，但你仍然应该创建一个 `海洋` 类，而不是使用一个全局数组对象。然后所有函数都是你创建的那个类的 <code> 方法 </code>。

54.4.2 数据结构

忽略海洋的深度和海岸线的形状，我们几乎不会损失普遍性，并将海洋建模为一个 2D 网格。

如果你编写一个索引函数 `cell(i,j)` 你可以让你的代码在很大程度上独立于实际选择的数据结构。论证为什么 `vector<vector<int>>` 不是最佳存储。

1. 你用什么代替？ 2. 当你有一个工作代码时，你能通过计时来证明你的选择确实更优吗？

54.4.3 类类型

在代码中包含 ‘ 魔法数字 ’ （ 0 = 空， 1 = 海龟，等等）是不优雅的。创建一个 `enum` 或 `enum class`(参见第 24.10 节) 这样你就有你单元格中的名称：

```
// /pacific.cppcell(i,j)= 占用::
海龟;
```

如果你想打印出你的海洋，如果你能直接 `cout` 你的单元格可能会很好：

```
// /pacific.cpp for (int i=0; i<iSize; ++i) { cout
<<i%10 << " "; for (int j=0; j<jSize; ++j) { cout
<< setw(hs) <<cell(i,j); } cout <<’\n’; }
```

54.4.4 Rangingover theocean

It is easy enough to write a loop as

for(int i=0; i{i14}iSize; i{i21})for(int j=0; j{j37}jSize; j{j44}) ... cell(i,j) ...

然而，始终如此有秩序地掠过你的领域可能并非好主意。你能实现这个吗：

```
for ( auto [i,j] : permuted_indices() ) {
    ... cell(i,j) ...
```

? 请参阅第 24.5 节关于结构化绑定的内容。

同样地，如果你需要计算乌龟周围有多少垃圾，你能让这段代码工作吗：

```
//pacific.cpp int count_around( int ic,int jc,occupy typ ) const{ int
count=0; for ( auto [i,j] : neighbors(ic,jc) ) { if (cell(i,j) == typ)++
count; } return count; };
```

54.4.5 随机数

对于船舶和海龟的随机移动，你需要一个随机数生成器。不要使用旧的 C 生成器，而要使用新的随机 一个；章节 24.7。

尝试找到一个解决方案，以便你在所有需要随机数的地点使用恰好一个生成器。提示：将生成器**静态** 在你的类中。

54.5 探索

与其让船只随机移动，你能给它们一个朝向最近垃圾带的优先方向吗？这能改善海龟种群的生存状况吗？

你能通过让船只在你的网格中占据一个 2×2 块来考虑船和海龟的相对大小吗？

到目前为止，你一直让垃圾保持原地。如果有洋流呢？你能让垃圾 ‘粘性化’ 吗，这样当垃圾颗粒接触时就会开始作为一个整体移动？

海龟吃沙丁鱼。（不，它们不吃。）如果海龟灭绝了，沙丁鱼种群会发生什么？你能提出与稳定生态或非稳定生态相对应的参数值吗？

54.5.1 代码效率

调查你的 **枚举** 在 54.4.3 部分中的实现 是否有任何影响计时。解析 24.10 部分的细则。

你可能会注意到，在广阔而空旷的海洋上航行可能效率不高。你可以考虑维护一个 ‘活动列表’，记录海龟等生物的位置，并且只在这个列表上进行循环。你会如何实现这一点？你期望在时间上看到差异吗？实际情况确实如此吗？

选择海洋的 vector-of-vectors 实现方式如何影响运行时；参见 54.4.2。

54. 巨大的垃圾带

第 55 章

图算法

在这个项目中，你将探索一些常见的图算法及其各种可能的实现方式。这里的主要主题是，常见的教科书算法表述方式不一定是计算上的最佳表述方式。

作为这个项目的背景知识，你被鼓励阅读 HPC 书籍 [11], 第 10 章；对于图的基础教程，请参阅 HPC 书籍 [11]，第 20 章。

55.1 传统算法

我们首先实现两种 单源最短路径 *(SSSP)* 算法的 “ 教科书 ” 形式：无权图和有权图。在下一节中，我们将考虑基于线性代数的表述方式。

为了开发实现，我们首先从一些必要的预备知识开始。

55.1.1 代码准备

55.1.1.1 邻接图

我们需要一个类 `Dag` 来表示有向无环图（ DAG ）：

```
// /dijkstra1.cppclass Dag{private::vector< vector<
int> > dag;;public:// Make Dagof ‘n’ nodes, no
edges for now.Dag(int n ) : dag( vector< vector< int> >
(n) ) {};}
```

最好有一个函数

```
// /dijkstra1.cpp
  const auto& neighbors( int i ) const { return dag.at(i); };
```

给定一个节点，返回该节点的邻居列表。

练习 55.1. 完成 `Dag` 类。特别是，添加一个生成示例图的方法：

- 为了测试 ‘循环’ 图，通常很有用：连接边

$$0 \to 1 \to \cdots \to N-1 \to 0.$$

- 也许还有一个带有随机边的图是个好主意。

编写一个显示图的方法。

55.1.1.2 节点集

SSSP 算法的经典公式，例如 *Dijkstra* 最短路径算法 (参见 HPC 书籍 [11], 第 10.1.3 节) 使用逐渐构建或消耗的节点集。

You could implement that as a vector:

```
vector<int >类_的_节点(nnodes);for(
int inode=0;inode<nnodes; inode++)// 将inode 标
记为距离未知: 类_的_节点.at(inode) =inf;
```

其中，您可以使用某种约定，例如负距离，来表示一个节点已从集合中移除。

然而，C++ 具有一个实际的 类容器，其中包含添加元素、查找元素和删除元素的方法；参见第 24.3.2 节。这使得我们的算法表达更加直接。在我们的案例中，我们需要一个 int/int 或 int/float 对的集合，具体取决于图算法。（也可以使用 `map`，使用 int 作为查找键，int 或 float 作为值。）

对于无权图，我们只需要一个已完成的节点集合，并将节点 0 作为我们的起点插入：

```
// /queuelevel.cpp
  使用 节点  _     = std::pair<unsigned,unsigned>;
  std::set< node_info > distances;
  distances.insert( {0,0} );
```

对于 Dijkstra 算法，我们需要一个已完成节点的集合，以及我们仍在处理的节点。我们再次设置起始节点，并将所有未处理节点的距离设置为无穷大：

```
// /queuedijkstra.cp const unsigned inf= std::numeric_limits<unsigned>::
max(); using node_info= std::pair<unsigned,unsigned>; std::set< node_info
> distances,to_be_done; to_be_done.insert( {0,0} ); for (unsigned n=1; n<
graph_size; ++n) to_be_done.insert( {n,inf} );
```

(为什么在这里我们需要第二个集合，而在无权图的情况下则不需要？)

练习 55.2. 编写一个代码片段来测试一个节点是否在距离 中 t.

- 当然，您可以为此编写一个循环。在这种情况下，要知道遍历集合会为您提供键值对。使用结构化绑定 ; 章节 24.5。

- 但最好使用一个 ‘算法’，在技术意义上的 ‘标准库中内置的算法’。在这种情况下，查找。
- ... 但使用 查找 时，你必须搜索一个确切的键 / 值对，而这里你想搜索的是：‘这个节点是否在 距离 集合中，无论其值是什么’。使用查找 _ 如果 算法；节 24.3.2。

55.1.2 Level set algorithm

我们从一个简单的算法开始：无权图中的单源最短路径算法；参见 HPC 书籍 [11], 节 10.1.1 以获取详细信息。等效地，我们在图中查找 levelsets。

对于无权图，距离算法相当简单。归纳地：

- 假设我们有一个最多在 n 步内可达的节点集，
- 那么它们的邻居（尚未在这个集合中）可以在 $n+1$ 步内到达。

算法概述是

```
// /queuelevel.cpp for (;;) { if (distances.size()==graph_size) break; /* * 遍历所有
已经完成的节点 */ for ( auto [node,level] : distances ) { /* * 将节点的邻居设置为
距离 ‘level + 1’ */ const auto& nbors =graph.neighbors(node); for ( auto n:
nbors ) { /* * 检查 ‘n’ 是否有一个已知的距离，* 如果没有，则使用 level+1 将其添
加到 ‘distances’ 中 */ /* ... */ { cout <<"node " << n << " level " << level+1 << ’
\n’ ; distances.insert( {n,level+1} ); }
```

Exercise 55.3. Finish the program that computes the SSSP algorithm and test it.

这段代码有一个明显的低效之处：对于每个层级，我们遍历所有已完成的节点，即使它们所有的邻居可能已经处理过了。

练习 55.4。 维护一组 “ 当前级别 ” 节点，并且仅调查这些节点以找到下一级别。在几个大型图上测试这两种变体。

55.1.3 Dijkstra 算法

在 Dijkstra 算法中，我们维护两组节点：一组已经找到最短距离的节点，以及一组仍在确定最短距离的节点。注意：a 暂时的最短距离对于

节点可能被更新多次，因为可能存在多条路径到达该节点。在权重方面的 ‘最短’ 路径可能不是经过边数最少的路径！

主循环现在看起来像这样：

```
// /queuedijkstra.cpp for (;;) { if (to_be_done.size()==0) break; /* * 找到距离最小的节点 */ /* ... */ cout << "min: "<< nclose << " @ " << dclose << ’\n’; /* * 将该节点移动到 done, */ to_be_done.erase(closest_node); distances.insert( *closest_node); /* * 将 nclose 的邻居设置为该距离 + 1*/ const auto& nbors= graph.neighbors(nclose); for ( auto n : nbors ) { // 在 distances 中查找 ‘n’ /* ... */ { /* * 如果 ‘n’ 没有已知的距离, * 查找它在 ‘to_be_done’ 中的位置并更新 */ /* ... */ to_be_done.erase( cfind ); to_be_done.insert( {n,dclose+1} ); /* ... */ }
```

(注意我们在 *to_be_done* 集合中删除一条记录，然后重新插入相同的键和新的值。如果我们使用 *map* 而不是 *set*，本可以执行简单的更新。)

在完成 / 未完成的集合中找到节点的各种位置由你自行实现。你可以使用简单的循环，或者使用 *find_if* 来查找与节点编号匹配的元素。

练习 55.5. 完善上述概述以实现迪杰斯特拉算法。

55.2 线性代数公式

在这个项目部分，你将探索如何使图算法看起来像线性代数。

55.2.1 代码预备知识

55.2.1.1 数据结构

你需要一个矩阵和一个向量。向量很简单：

```
// /graphmvpdijkstra.cpp 类向量 {private: 向量 < 向量值 > 值 ;public: 向量 (int n ): 值 ( 向量 < 向量值 >(n, 无限 ) ) {};
```

对于矩阵，最初使用密集矩阵：

```
//graphmvpdijkstra.cpp 类 邻接矩阵 { private: 向量 < 向量 < 矩阵值 > > 邻接; public: 邻接矩阵 (int n) : 邻接 ( 向量 < 向量 < 矩阵值 >>( n, 向量 < 矩阵值 >(n, 空 ) ) ) { };
```

但稍后我们将优化它。

注意 43 一般来说，将矩阵存储为向量 - 向量的形式并不是一个好主意，但在这种情况下我们需要能够返回矩阵行，所以这样更方便。

55.2.1.2 矩阵向量乘法

我们来编写一个例程

```
向量 邻接矩阵 :: 左乘 ( const 向量 & 左 );
```

这是最简单的解决方案，但不一定是最高效的，因为它为每次矩阵 - 向量乘法创建一个新的向量对象。

正如理论背景中所述，图算法可以表述为具有非典型加 / 乘操作的矩阵 - 向量乘法。因此，乘法例程的核心可能如下所示

```
//graphmvp.cpp for (int row=0; row<n; ++row ) { for ( int col=0; col<n; ++col) { result[col] = add( result[col], mult( left[row] ,adjacency[row][col] ) ); } }
```

55.2.2 无权图

练习 55.6. 实现 `add` / `mult` 例程，使无权图上的 SSSP 算法能够工作。

55.2.3 迪杰斯特拉算法

例如，考虑以下邻接矩阵：

```
.  1  .  .  5
.  .  1  .  .
.  .  .  1  .
.  .  .  .  1
1  .  .  .  .
```

最短距离 $0 \to 4$ 是 4，但在第一步中发现了一个较大的距离 5。您的算法应显示类似以下内容，以显示已知最短距离的连续更新：

输入：0 . . . . 第 0 步：

0 1 . . 5 第 1 步：0 1 2 .

5 第 2 步：0 1 2 3 5 第

3 步：0 1 2 3 4

练习 55.7。 实现新的 `add` / `mult` 例程，使矩阵 - 向量乘法对应于加权图上的单源最短路径 (SSSP) 的迪杰斯特拉算法。

55.2.4 稀疏矩阵

上述描述的矩阵数据结构可以通过仅存储非零元素来使其更加紧凑。实现这一点。

55.2.5 进一步探索

通过运算符重载，你可以让你的代码多么优雅？

你能编写所有对最短路径算法吗？

你能将 SSSP 算法扩展为生成实际路径吗？

55.3 测试和报告

你现在有两个完全不同的图算法实现。生成一些大型矩阵并测试这些算法。

讨论你的发现，注意执行的工作量和所需的内存量。

第 56 章

拥塞

本节包含一系列练习，逐步构建起汽车交通的模拟。

56.1 问题陈述

汽车交通由个体行为决定：驾驶员加速、制动、决定走哪条路。由此，涌现现象出现，最明显的是交通拥堵。

56.2 代码设计

在这个项目中，我们将探索所谓的 *actor* 模型：模拟由独立的 actor 组成，它们会对他人的行为做出反应。

我们只需要两个基本类：

1. 一个 `Car` 类，其中对象有一个位置和速度，并且它们可以对他前面的车做出反应。（为了简化，我们假设司机只看他们正前方的车。） 2. 一个 `Street` 类，其中每条街道都是一个包含若干车的容器。

我们可以用多种方式运行模拟（使用 *thread*，由 *interrupt* 驱动），但为了简化，我们考虑离散的时间步长。这意味着车类和街道类都有一个 `progress` 方法，该方法将对象推进一个时间步。

56.2.1 Cars

你可以对车要求的主要事情是让它移动一个时间步。这意味着你需要它的位置、速度和方向。对于车来说，知道自己的速度和位置是有意义的，但方向来自街道。因此，街道可以像这样更新车：

```
// /traffic_lib.cppfor (auto c: cars()) {c->
progress(unit vector()); }
```

并且汽车会以方向作为输入。

56.2.2 街道

让我们从单行道开始。一条街道是一系列车的集合，但在它们开始相互通行之前，它们实际上是按顺序排列的。向量适合这种情况。然而，一辆车可以移动到不同的街道，甚至如果它在交叉口，可以同时出现在两条街道上。出于这个原因，将街道建模为

```
类街道 {private:vector<shared_
ptr<Car>> cars;
```

一旦模拟开始运行，车将对他人的行为做出反应，特别是它们前方车辆的行为。有各种方法可以建模这种情况。

1. 例如，街道可以将一辆车的信息传递给下一辆车。2. 可能更优雅的是，每辆车都有一个指向它前面和后面的指针。

56.2.3 单元测试

在这个级别上，你可以进行一些单元测试来确保你的代码按预期运行。

1. 定义一条街道，并测试其单位向量。2. 在不同的位置放置一辆车，包括街道上和街道之外或旁边。为这些情况编写测试方法，例如 **missingsnippetstreettestbeyond**3. 如果你把车放在街道上，它的速度保持不变，并在可预测的时间内离开街道：

```
// /street.cpp REQUIRE_NOTHROW( main_street.insert_with_speed(speed) ); float ti
me=0.f; for ( int t=0; t<static_cast<int>(main_street.length()); ++t ) { // 额外一步 INFO(
format( "at t={}, #cars={}",t,main_street.size() ) ); if ( main_street.empty() ) break; if
(global_do_vis) { // 动画： main_street.display(); std::this_thread::sleep_for(
seconds{2}/10. ); cout<< '\n'; printf( "%c[1A",(char)27); // 再次对齐。 } auto car= main_
street.at(0); REQUIRE( car->speed()==Catch::Approx(speed) ); REQUIRE_NOTHROW(
main_street.progress() ); } REQUIRE(main_street.empty() );
```

第 57 章

DNA 测序

在这个练习集中，你将编写 DNA 测序的机制。

57.1 基本函数

参考第 24.5 节。

首先，我们设置一些基本机制。

练习 57.1。 有四种碱基，A 、 C 、 G 、 T，每种都有一个互补碱基：A ↔ T，C ↔ G。通过 amap 实现，并编写一个函数

```
char BaseComplement(char);
```

练习 57.2。 编写代码将 *Fasta* 文件读入一个字符串。第一行，以 > 开头，是注释；其他所有行应连接成一个字符串表示基因组。

读取文件 lambda_virus.fa 中的病毒基因组。

用两种方法统计基因组中的四种碱基。首先使用 map。记录这需要多长时间。然后用一个长度为四的数组和一个条件语句做同样的事情。

附加：尝试想出一种更快的方法来统计。使用一个长度为 4 的 vector，并想出一种直接从字母 A 、 C 、 G 、 T 计算索引的方法。提示：研究 ASCII 码，可能还需要位操作。

57.2 Denovoshotgun assembly

一种生成基因组的办法是将其切割成片段，然后算法性地将它们重新粘合在一起。（对短读的碱基测序比对非常长的基因组测序要容易得多。）

如果我们假设我们有足够的读数，使得每个基因组位置都被覆盖，我们可以查看读数之间的重叠。然后，一种启发式方法是找到最短公共超集（ SCS ）。

57.2.1 Overlaplayout 共识

1. 构建一个图，其中读取为顶点，如果它们重叠则顶点相连；重叠量是边的权重。2. SCS 是此图中的一个哈密顿路径 —— 这已经是一个 NP 完全问题。3. 此外，我们优化总重叠量最大 .⇒ 旅行商问题 (TSP)，NP 难。

与其寻找最优超集，我们可以使用贪婪算法，每次找到重叠量最大的读取。

重复序列通常是问题。另一个是测序错误产生的虚假子图 s.

57.2.2 De Bruijn 图组装

57.3 ‘ 读取 ’ 匹配

一个 ‘ 读取 ’ 是 DNA 的一个短片段，我们希望将其与基因组进行匹配。在本节中，你将探索这种类型的匹配算法。

虽然在这里我们主要考虑基因组学的上下文，但这类算法有其他应用。例如，在网页中搜索一个单词本质上就是同一个问题。因此，这个主题有相当长的历史。

57.3.1 简单匹配

我们首先探索一个简单匹配算法：对于基因组中的每个位置，查看读取是否匹配。

```
ATACTGACCAAGAACGTGATTACTTCATGCAGCGTTACCAT
  ACCAAGAACGTG
    ^ 不匹配

ATACTGACCAAGAACGTGATTACTTCATGCAGCGTTACCAT
ACCAAGAACGTG 完全匹配
```

练习 57.3. 编写用于匹配读的朴素算法。在通过从基因组中复制子字符串获得的假读上测试它。使用基因组文件 inphix.fa。

现在读取 *Fastq* 文件 ERR266411_1.first1000.fastq。Fastq 文件包含四行一组的数据：每组中的第二行包含读。有多少个这些读与基因组匹配?

读不一定是完全匹配的；事实上，fastq 文件中的每组的第四行给出了相应读的 “ 质量 ” 指示。如果你取每个读的前 30 个或更多字符的子字符串，你会得到多少个匹配?

57.3.2 Boyer-Moore 匹配

The *Boyer-Moore* 字符串匹配算法 [6] 比朴素匹配快得多，因为它使用两个巧妙的技巧来排除那些不会匹配的比较。

Bad character rule

在朴素匹配中，我们从左到右确定匹配位置，然后尝试从左到右匹配。在 Bowers-Moore （ BM ）中，我们仍然从左到右查找匹配位置，但我们的匹配是从右到左进行的。

vvvv 匹配位置
antidisestablishmentarianism
blis ^bad character

不匹配是一个模式中的 ‘l’，它与文本中的 ‘d’ 不匹配。由于模式中根本没有 ‘d’，我们将模式完全移过不匹配位置：

vvvv 匹配位置
antidisestablishmentarianism blis

事实上，我们将其进一步移动到模式的首个字符的第一个匹配位置：

vvvv 匹配位置
antidisestablishmentarianism blis ^ 首个
字符匹配

当我们有一个不匹配，但文本中的字符确实出现在模式中的情况稍微有点棘手：我们在模式中找到不匹配字符的下一个出现位置，并使用该位置来确定偏移距离。

shoobeedoobeeboobah
edoobeeboob ^ 不匹配 ^ ‘d’ 的
其他出现位置

请注意，这可以是一个比前一个情况小得多的偏移量。

v shoobeedoobeeboobah
edoobeeboob ^ 匹配坏字符 ‘d’ ^ 新位置

练习 57.4。 讨论您期望这种启发式方法在基因组学相对于文本搜索的上下文中的效率。(见上文。)

好的后缀规则

‘好的后缀’ 由坏字符之后的匹配字符组成。在移动读取时，我们尝试保持好的后缀完整：

desistrust

listrest ^^ 好后缀规则 listrest ^^ 后缀的下一个出现

第 58 章

内存分配

此项目尚未准备就绪

https://www.youtube.com/watch?v=R3cBbvIFqFk

单调分配器

- base 和 free 指针，
- 始终从空闲位置分配
- 只有当所有内容都已释放时才释放。

适当：

- 视频处理：释放用于一帧的所有内容
- 事件处理：释放用于处理事件的所有内容

栈分配器

58. 内存分配

第 59 章

Ballistics calculations

此项目尚未准备好投入生产

59.1 简介

从 https://encyclopedia2.thefreedictionary.com/ballistics

弹道学

弹道学研究火炮弹丸、迫击炮弹、航空炸弹、火箭炮弹丸和导弹、鱼叉等的运动。弹道学是一门基于一系列物理学和数学学科的技术军事科学。内弹道学与外弹道学有所区别。

内弹道学涉及弹丸（或其他机械自由度受特定条件限制的物体）在炮管内受火药气体影响的运动，以及炮管或火药火箭燃烧室在发射过程中发生的其他过程的规则。内弹道学将发射视为火药化学能快速转化为热能，再转化为推动弹丸、装药和枪身后坐部分的机械工作的复杂过程。在内弹道学中，发射过程中区分的不同时期包括预备期，从火药燃烧开始到弹丸开始运动；第一（主）期，从弹丸开始运动到火药燃烧结束；第二期，从火药燃烧结束到弹丸离开炮管（气体绝热膨胀期）；以及火药气体对弹丸和炮管的后效期。与最后一期相关的过程的规律由弹道学的特殊分支 —— 中间弹道学处理。弹丸后效期的结束将内弹道学和外弹道学研究的现象分开。

内弹道学的主要分支包括 “ 火药静力学 ” 、 “ 火药动力学 ” 和弹道枪设计。火药静力学是研究火药在恒定体积中燃烧时粉末燃烧和气体形成规律的科学，它确定了火药的化学成分及其形状和尺寸对燃烧和气体形成规律的影响。火药动力学关注

通过研究发射过程中枪膛内发生的过程和现象，以及确定枪膛设计特征、加载条件与发射过程中发生的各种物理化学和机械过程之间的关系。基于对这些过程以及作用在弹丸和枪管上的力的考虑，建立了一个描述发射过程的方程组，包括描述内弹道的基本方程，该方程将药燃部分的大小、枪膛内火药气体的压力、弹丸的速度以及弹丸已行进的路径长度联系起来。求解该方程组，发现火药气体压力 ρ 的变化、弹丸速度 v 以及弹丸在路径 l 上行进时其他参数之间的关系，是第一个主要的（直接的）

为了解决这个问题，使用解析方法、数值积分方法（包括基于计算机的方法）和表格方法。鉴于发射过程的复杂性以及某些因素研究的不足，做出了一些假设。内弹道的修正公式具有很大的实际意义；它们使得在加载条件发生变化时，能够确定弹丸初速的变化和枪膛内的最大压力。

弹道枪设计是内弹道学的第二个主要（相关）问题。通过它确定了枪管的 设计规范以及在给定口径和质量的弹丸在飞行中获得指定（枪口）速度的装填 条件。在设计过程中，计算了所选枪管在枪管长度和时间上气体压力变化和弹丸速度变化的曲线。这些曲线是设计整个炮兵系统和其弹药 的初始数据。内弹道学还包括在步枪中使用特种和组合装药、锥形枪管系统以及 火药燃烧时气体逸出的系统（高低压枪和无后坐力炮、步兵迫击炮）中的发射 过程的研究。另一个重要分支是火药火箭的内弹道学，它已发展成为一种专门的 科学。火药火箭内弹道学的 主要分支是半封闭空间的火药燃烧静力学，它考虑在相对较低和恒定压力下的 火药燃烧规律；火药火箭内弹道学的基本问题的求解，即确定（在设定装填条件 下）燃烧室中火药气体压力随时间变化的规律，以及确定保证所需火箭速度所 需推力变化的规律；火药火箭的弹道设计，它涉及确定火药的能量产生特性、装 药的重量和形状，以及确保在火箭战斗部指定重量下工作时产生必要推力的喷管 设计参数。

外弹道学是研究无制导弹道（迫击炮弹、子弹等）在离开炮管（或发射器）后运动以及影响这种运动的因素。它基本上包括对弹道所有运动元素的研究，以及作用在其飞行中的力（空气阻力、重力、反作用力、产生的力

后效期等）；研究抛射体质心运动以计算其弹道（见图 2），当存在设定的初始条件和外部条件时（外弹道学的基本问题）；以及确定抛射体的飞行稳定性和散布。外弹道学有两个重要的分支：修正理论，它发展了评估决定抛射体飞行因素对其弹道性质影响的方法；以及编制射表的技术和在火炮系统设计中寻找最优外弹道变化的技术。抛射体运动问题和修正理论问题的理论解法归结为建立抛射体运动方程、简化这些方程和寻求求解这些方程的方法。计算机的出现使这一工作变得显著容易和快速。为了确定获得给定弹道所需的初始条件 —— 即初始速度和射向角、抛射体的形状和质量 —— 外弹道学中使用特殊表格。编制射表技术的制定涉及确定理论研究和实验研究的最佳组合，以便以最短的时间获得所需精度的射表。外弹道学的方法也用于研究航天器运动规律（在不受控制力和力矩影响时的运动）。随着导弹的出现，外弹道学在飞行理论的形成和发展中发挥了重要作用，并成为该理论的特例。

弹道学的出现可追溯到 16 世纪。关于弹道学的最早著作是意大利人 N. 塔尔塔利亚的《新科学》（ 1537 年）和《问题》。

弹道学的出现可追溯到 16 世纪。最早的弹道学著作是意大利人 N. 塔尔塔利亚的《新科学》（ 1537 年）和《与火炮射击相关的问题与发现》（ 1546 年）。在 17 世纪，伽利略、意大利人 E. 托里拆利和法国人 M. 梅森等人确立了外弹道学的基本原理，梅森提出将抛射体运动科学称为弹道学（ 1644 年）。I. 牛顿首次研究了考虑空气阻力的抛射体运动（《自然哲学的数学原理》，1687 年）。在 17 世纪和 18 世纪，荷兰人 C. 惠更斯、法国人 P. 瓦里农、英国人 B. 罗宾斯、瑞士人 D. 伯努利、俄国科学家 L. 艾勒等人研究了抛射体的运动。内弹道学的实验和理论基础在 18 世纪的罗宾斯、 C. 哈顿、伯努利等人的著作中奠定。在 19 世纪，空气阻力的规律被确立（ N. V. 迈耶夫斯基和 N. A. 扎布兹基定律、哈弗定律和 A. F. 西阿奇定律）。

对弹道计算的数值分析在 *ENIAC* 中描述了 [17]。

59.1.1 物理

这些是控制方程：

$$\begin{aligned} x'' &= -E(x' - w_x) + 2\Omega \cos L \sin \alpha y' \\ y'' &= -Ey' - g - 2\Omega \cos L \sin \alpha x' \\ z'' &= -E(z' - w_z) + 2\Omega \sin Lx' + 2\Omega \cos L \cos \alpha y' \end{aligned} \tag{59.1}$$

where

- x, y, z 是感兴趣的数量：距离、高度、横向位移；
- w_x, w_z 是风速；
- E 是 y 的复杂函数，涉及空气密度和声速；
- α 是方位角，即发射角；
- 其他所有量对于物理真实性是必需的，但在本次编程练习中将不设置 $\equiv 1$。

59.1.2 数值分析

这使用了一个三阶的欧拉 - 麦克劳林方案：

$$f_1 - f_0 = \frac{1}{2}(f'_0 + f'_1)h + \frac{1}{12}(f'_0 - f'_1)h^2 + O(h^5) \tag{59.2}$$

计算结果为

$$\begin{aligned} \bar{x}'_1 &= x'_0 + x''_0 \Delta t \\ x_1 &= x_0 + x'_0 \Delta t \\ x'_1 &= x'_0 + (x''_0 + \bar{x}''_1) \frac{\Delta t}{2} \\ x_1 &= x_0 (x'_0 + \bar{x}'_1) \frac{\Delta t}{2} + (x''_0 - \bar{x}''_1) \frac{\Delta t^2}{12} \end{aligned} \tag{59.3}$$

第 60 章

密码学

60.1 基础

虽然浮点数可以跨越相当大的范围 —— 对于双精度数来说，大约可以达到 10^{300} 左右 —— 但整数的范围要小得多：大约到 10^9。这对于处理更大数字的密码学应用来说是不够的。（为什么不能用浮点数？）

因此，第一步是编写没有这种限制的类 `Integer` 和 `Fraction`。使用运算符重载使简单表达式能够工作：

```
Integer big=2000000000; // 20 亿 big *
= 1000000; bigger =big+1;Integer one =
bigger % big;
```

练习 60.1。 编写法雷序列代码。

60.2 密码学

https://simple.wikipedia.org/wiki/RSA_ 算法

https://simple.wikipedia.org/wiki/ 指数 _ 乘方 _

60.3 区块链

实现一个区块链算法。

60. 密码学

第 61 章

气候变化

> 气候已经变化，并且一直在变化。
>
> 拉杰 · 沙赫，白宫新闻发言人主管

气候总是变化的说法远非严谨的科学主张。如果我们将其解释为关于气候统计行为的陈述，在这种情况下以平均全球温度来衡量，我们可以赋予其意义。在这个项目中，你将处理真实温度数据，并对其进行一些简单的分析。（这个项目的灵感来自 [18]。）

理想情况下，我们会使用来自世界各地各种测量站的数据集。然后 Fortran 是一种很棒的语言，因为它具有数组操作（见第 39 章）：你可以用一行代码处理所有独立测量值。为了简化，我们在这里使用一个包含 1880 年至 2018 年每月数据的数据文件。然后，我们将使用各个月份作为 ‘ 假设 ’ 的独立测量值。

61.1 读取数据

在代码仓库中，你找到两个文本文件

GLB.Ts+dSST.txt GLB.Ts.txt

这些文件包含与 1951-1980 年平均值的温度偏差。偏差按年每月给出，时间为 1880-2018 年。这些数据文件和其他更多内容可以在 https://data.giss.nasa.gov/gistemp/。

练习 61.1。 首先列出可用年份，并创建一个大小为 $12 \times$ nyears 的数组 monthly_deviation，其中 nyears 是文件中的完整年数。使用格式和数组表示法。

文本文件包含一些你不需要的行。你在程序中过滤它们，还是使用脚本？提示：合理使用 grep 将使 Fortran 代码更容易。

61.2 统计假设

我们假设 mr Shah 真正的意思是气候具有 ‘ 平稳分布 ’，这意味着高低值具有一个与时间无关的概率分布。这意味着在 n 数据点中，

每个点有 $1/n$ 的概率成为最高记录。由于超过 $n+1$ 年，每年都有 $1/(n+1)$ 的概率，第 $n+1$ 年有 $1/(n+1)$ 的概率成为最高记录。

我们得出结论，作为一个关于 n 的函数，最高记录（或最低记录，但让我们坚持最高记录）的概率随着 $1/n$ 的减少，并且连续最高记录之间的差距大约是年份的线性函数 [1]。

这是我们可以测试的。

练习 61.2。 创建一个与 monthly_deviation 形状相同的 previous_record 数组。这个数组记录（对于每个月，我们将其视为独立测量）该年是否为记录，如果不是，则记录前一个记录发生的时间：

$$\text{PrevRec}(m,y) = \begin{cases} y & \text{if } \text{MonDev}(m,y) = \max_{m'}\big(\text{MonDev}(m',y)\big) \\ y' & \text{if } \text{MonDev}(m,y) < \text{MonDev}(m,y') \\ & \text{and } \text{MonDev}(m,y') = \max_{m''<m'}\big(\text{MonDev}(m'',y)\big) \end{cases}$$

同样，使用数组表示法。这也是使用 Where 子句的绝佳地方。

练习 61.3. 现在取每个月，并找出记录之间的差距。这给你两个数组：gapyears 表示记录最高值之间开始出现差距的年份，gapsizes 表示该差距的长度。

这个函数，因为它被单独应用于每个月，所以不使用数组表示。 on.

假设是现在，差距的长度是年份的线性函数，例如以与起始年份的距离来衡量。当然它们并不是完全的线性函数，但我们能否通过线性回归拟合一个线性函数。

练习 61.4. 复制代码来自

http://www.aip.de/groups/soe/local/numres/bookfpdf/f15-2.pdf 并适应我们的目的：找到描述记录之间差距的线性函数的最佳斜率和截距。

你会发现差距明显不是线性增加的。那么这个负面结果就是故事的结局，还是我们可以做更多?

练习 61.5. 你能把这个练习变成一个全球变暖的测试吗？你能把偏差解释为年度温度增加加上一个平稳分布的总和，而不是仅仅是一个平稳分布吗?

1. 从技术上讲，我们处理的是温度的均匀分布，这使得最大值和最小值呈贝塔分布。

第 62 章

桌面计算器解释器

在这个练习集中，你将编写一个 “ 桌面计算器 ”： 一个小型交互式计算器，结合了数值和符号计算。

这些练习主要使用第 36 章、 41 章、 35 章的内容。

62.1 Named variables

我们从 “ 命名变量 ” 开始：`namedvar` 类型将一个字符串与一个变量关联：

```
//varhandling.F90type namedvarcharacter(len=20) ::
expression = ""integer ::valueend typenamedvar
```

一个命名变量有一个值，和一个字符串字段，该字段是生成该变量的表达式。当你创建变量时，表达式可以是任何东西。

```
//varhandling.F90type(na
medvar) :: x ,y ,z ,a x= namedvar("
x",1 )y= namedvar("yvar",2 )
```

接下来我们将用这些类型对象进行计算。例如，相加两个对象

- 将它们的值相加，并且
- 将它们的 `expression` 字段连接起来，给出对应于和值的表达式。

你的第一个作业是编写 `varadd` 和 `varmult` 函数，使以下程序按照指定的输出运行。这使用了来自章节 35.3 和 41.5.

Exercise 62.1. The following main program should give the corresponding output:

Code:

```
1 //varhandling.
2   打印 *,x
3   print *,y
4   z = varadd(x,y)
5   打印 *,z
6   a = varmult(x,z)
7   打印 *,a
```

Output
t s ruc f] varhandling:

```
x                          1
yvar                       2
(x)+(yvar)                 3
(x)*((x)+(yvar))           3
```

（明确：这两个例程需要执行数值和字符串的 ‘加法’ 和 ‘乘法’ 。）

您可以根据代码仓库中的名为 *var.F90* 的文件进行参考

62.2 首次模块化

让我们通过引入模块来组织现有的代码；参见第 37 章。

练习 62.2。 创建一个模块（建议名称：`VarHandling`），并将 `namedvar` 类型定义以及例程 `varadd`、`varmult` 移入其中。

练习 62.3。 还创建一个模块（建议名称：`InputHandling`），其中包含第 35 章字符练习中的例程 islower 、isdigit 以及用于识别算术运算的 `isop` 例程。

62.3 事件循环和堆栈

在编写解释器的过程中，我们将编写一个 “事件循环”：一个不断接受单个字符输入并处理它们的循环。输入 `"0"` 将表示终止进程。

练习 62.4. 编写一个循环，接受字符输入，并且只打印出遇到的字符类型：一个小写字母、一个数字，或一个表示算术运算 `+-*/` 的字符。

代码:

```
1// /structf.F902   do3
read*,input4 if (input .eq.'0')
then5          exit6 else if
(isdigit(input) ) then
```

Output
[structf] interchar:

```
Inputs: 4 x 3 + 0
 4 is a digit
 x is a lowercase
 3 is a digit
 + is an operator
```

使用上面介绍的 `InputHandling` 模块。

62.3.1 栈

接下来，我们将要在 `namedvar` 类型上栈中存储值。一个 栈 是一种数据结构，新元素会放在顶部，因此我们需要用一个栈指针 来指示顶部元素。等效地，栈指针指示了已经有多少元素：

```
// / 解释。F90 类型 ( 命名变量 ), 维度 (10)::
栈整数 :: 栈指针 =0
```

由于我们使用模块，让我们将栈从主程序中分离出来，并将其放入适当的模块中。

练习 62.5. 向 *VarHandling* 模块添加栈 变量和栈指针。

Since Fortran uses 1-based indexing, a starting value of zero is correct. For C/C++ it would have been −1.

接下来，我们将开始实现栈操作，例如将命名变量对象放在栈上。

62.3.2 栈操作

我们扩展了上述事件循环，该事件循环仅识别输入字符，通过实际包含动作。也就是说，我们反复

1. 从输入中读取一个字符；2. 0 导致事件循环退出；否则：3. 如果它是数字，则在堆栈顶部创建一个新的 *namedvar* 条目，该值既作为 *value* 字段的数值，也作为 *expression* 字段中的字符串。你可能会在主程序中编写以下内容：

```
/// 解释 .F90if (isdigit()) thenstackpointer= stackpointer + 1read(
input,'( i1 )') stack(stackpointer)%value stack(stackpointer)%
expression= trim(input)
```

(你已经在练习 *isdigit* 中编写了 35.1。) 但更简洁的设计使用对 *VarHandling* 模块中的方法进行函数调用来实现：

```
///internum.
      else if    ( isdigit(input) )
    then
        call 栈 _ 压入 ( 输入 )
```

```
// / 解释m.F90
子程序 栈 _ 压入 ( 输入 )
  implicit none
  字符 , 意图 ( 输入 ) :: 输入
```

注意，栈 _ 压入 子程序没有栈或栈指针作为参数：因为它们都在同一个模块中，所以可以作为全局变量访问。

最后，

4. if it is a letter indicating an operation $+, -, \times, /$,
   (a) 从栈中取出两个顶部的条目，降低栈指针；(b) 将该操作应用于操作数；以及 (c) 将结果压入栈中。

辅助函数 栈 _ 显示 有点棘手，所以你在这里得到它。这使用字符串格式化（第 41.3 节）和隐式 do 循环（第 32.3 节）：另外，请注意 栈 数组和 栈指针 像全局变量一样。

```
// /internum.F90
subroutine stack_display()
  implicit none
  ! local variables
  integer :: istck

  if (stackpointer.eq.0) return
  print '( 10( a,a, a,i0,"; ") )', ( &
       " expr=",trim(stack(istck)%expression), &
       " val=",stack(istck)%value, &
       istck=1,stackpointer )

end subroutine stack_display
```

让我们将各种选项添加到事件循环中。

Exercise 62.6. Make your event loop accept digits, creating a new entry:

Code:

```
// /internum.F90
     else if ( isdigit(input) ) then
        call stack_push(input)
```

Output

```
[structf] internum:
Inputs: 4 5 6 0
 expr=4 val=4;
 expr=4 val=4;  expr=5 val=5;
 expr=4 val=4;  expr=5 val=5;  expr=6 val=6;
```

接下来我们集成操作：如果输入字符对应于算术运算符，我们调用栈 _op 以该字符。该例程反过来调用适当的操作，具体取决于字符是什么。

Exercise 62.7. Add a clause to your event loop to handle characters that stand for arithmetic operations:

代码:

```
// /internum.F90 else if (isop(输入)) then call 栈_op(输入)
```

输出

```
[structf] internumop:
Inputs: expr=4;expr=4;expr=5;expr=4;expr=5;expr=6;
expr=4;expr+(5);val=11;expr_(4);val=15((5);(6));
```

62.3.3 项目重复

最后，我们可能希望多次使用一个栈条目，因此我们需要复制栈条目的功能。

为此我们需要能够引用一个栈条目，因此我们添加一个字符标签字段：`namedvar` 类型现在存储

1. 一个字符 id，2. 一个整数值，和

3. 其生成表达式作为字符串。`// /`

```
vartype.F90type
namedvarcharacter::
idcharacter(len=20) ::
expressioninteger ::valueend
typenamedvar
```

练习 62.8. 为 `namedvar` 添加 `id` 字段，并确保您的程序仍然可以编译和运行。

事件循环现在增加了一个额外的步骤。如果输入字符是小写字母，它被用作 `id` 的 `namedvar`，如下所示。

- 如果已经存在一个具有该 `id` 的栈条目，它将在栈顶被复制；
- 否则，栈顶条目的 `id` 将被设置为该字符。

以下是新的 `stack_print` 函数的相关部分：

```
// /interprets.F90
  print '( 10( a,a1, a,a, a,i0,"; ") )', ( &
       "id:",stack(istck)%id, &
       " expr=",trim(stack(istck)%expression), &
       " val=",stack(istck)%value, &
       istck=1,top )
```

练习 62.9. 在事件循环中编写缺失的函数及其子句：

Code:

```
1 // /interpret.F90
2          stacksearch =
       find_on_stack(stack,stackpointer,input)
3          if ( stacksearch>=1 ) then
4             stackpointer = stackpointer+1
5             stack(stackpointer) =  stack(stacksearch)
```

输出
[structf] stackfind:

```
Inputs: 1 x 2 y x y + z 0
id:. expr=1 val=1;
id:x expr=1 val=1;
id:x expr=1 val=1; id:. expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2;
id:x expr=1 val=1; id:y expr=2 val=2; id:x expr=1 val=1;
id:x expr=1 val=1; id:y expr=2 val=2; id:x expr=1 val=1; id:y expr=2
    val=2;
id:x expr=1 val=1; id:y expr=2 val=2; id:. expr=(1)+(2) val=3;
id:x expr=1 val=1; id:y expr=2 val=2; id:z expr=(1)+(2) val=3;
```

（这个条件语句的 else 部分包含什么？）

62.4 模块化

使用您已开发的模块和函数，您现在拥有一个非常简洁的主程序：

```
// /intermod.F90
  do
     call stack_display()
     read *,input
     if (input .eq. '0') exit
     if ( isdigit(input) ) then
        call stack_push(input)
     else if ( isop(input) ) then
        call stack_op(input)
     else if ( islower(input) ) then
        call stack_name(input)
     end if
  end do
```

您看到，通过将栈移入模块，栈变量和栈指针都不再在主程序中可见。

但这种设计有一个重要的局限性：只有一个栈，被声明为一种全局变量，通过模块访问。

全局数据是否是良好实践是另一回事。在这种情况下它是可以辩护的：在计算器应用中只有一个栈。

62.5 面向对象

但也许有时我们需要超过一个栈。让我们将栈数组和栈指针打包到一个新的类型中：

```
// / 类间。F90 类型 stackstruct 类型 ( 命名变量 ), 维度 (10) :: 数
据 integer ::top=0 包含过程 , 公共 :: 显示 , 查找 _id, 名称 , 操作 ,
推送 end typestackstruct
```

练习 62.10. 更改事件循环，使其调用 *stackstruct* 类型的对象的方法，而不是接受栈作为输入的函数。

例如，推送 函数的调用方式如下：

```
// / 类间 .F90
     if ( isdigit(input) ) then
         调用 thestack%push( 输入 )
```

62.5.1 运算符重载

The *varadd* 和类似算术例程使用函数调用来实现我们希望写为算术操作的逻辑。

练习 62.11. 在运算符重载 的 *varop* 函数中使用 :

```
//interpretov.F90if (op=="+"
")thenvarop= op1 + op2
```

等等。

62. 计算器解释器

第五部分

高级主题

第 63 章

外部库

如果你有一个 C++ 编译器，你可以自己编写尽可能多的软件。然而，你工作中可能需要的一些东西已经被其他人写好了，而且如果运气好的话，那个其他程序员已经将代码发布为一个库。

本章教你如何安装和使用库，并将讨论在科学计算上下文中有用的几个库。

63.1 什么是软件库?

在本章中，你将学习关于使用 软件库：这种软件不是作为一个独立的包来编写的，而是以一种方式编写，使你可以在自己的程序中访问其功能。

软件库可能非常庞大，科学库尤其如此，它们通常是多人多年合作的项目。另一方面，其中许多只是由单个程序员编写的简单工具。在后一种情况下，您可能需要担心该软件的未来支持。

63.1.1 使用外部库

使用软件库通常意味着

- 您的程序有一行 `#include "fancylib.h"`
- 并且您编译和链接为：icpc -c yourprogram.cpp -I/usr/include/fancylib icpc -o yourprogram yourprogram.o \ -L/usr/lib/fancylib -lfancy

您将在下文中看到具体的示例。

如果你现在担心每次编译都要打很多字，

- 如果你使用 IDE，通常可以在选项中一次性添加库；或者
- 你可以使用 *Make* 或 *CMake* 来自动化你的程序的构建过程。

63.1.2 获取和安装外部扩展

有时软件扩展可以通过 包管理器 获取，但我们打算用老方法：自己下载和安装。

一个流行的可下载软件位置是 `github.com`。然后你可以选择是否

- 克隆代码仓库，或
- 下载所有文件到一个文件中，通常带有 .tgz 或 .tar.gz 扩展名；在这种情况下，你需要用 tar fxz fancylib.tgz 解包它

  这通常会给你一个名为 fancylib-1.0.0 的目录

  其中包含库的源代码和文档，但不包含任何二进制文件或特定于机器的文件。

无论如何，从现在起，我们假设你已经有一个包含下载的包的目录。

There are two main types of installation:

- 基于 *GNU autotools*，你可以通过存在一个 *configure* 程序来识别它；configure ## 许多选项使 make install 或

- 基于 *Cmake*，你可以通过存在 CMakeLists.txt 文件来识别它：cmake ## 许多选项 make make install

63.1.2.1 Cmake 安装

使用 cmake 安装包最简单的方法是创建一个构建目录，放在源代码目录旁边。从该目录发出 cmake 命令，并引用源代码目录：

```
mkdir build cd build
cmake ../fancylib-1.0.0
make make 构建
```

有些人把构建目录放在源代码目录内部，但这是一种不好的做法 .

除了指定源代码位置外，你还可以给 cmake 提供更多选项。最常见的是

- 指定安装位置，例如因为你没有 超级用户 权限；或者

- 指定编译器，因为 Cmake 会默认使用 *gcc* 编译器，但你可能想要使用 Intel 编译器。

```
CC=icc CXX=icpc
cmake -D CMAKE_INSTALL_PREFIX:PATH=$
 {HOME}/mylibs/fancy \ ../fancylib-1.0.0
```

63.2 选项处理 :cxxopts

假设你有一个处理大型数组的程序，并且你希望能够更改数组大小。你可以

- 你可以每次重新编译你的程序。
- 你可以让你的程序解析 *argv*，并希望你能精确记住你的命令行选项是如何被解释的。
- 你可以使用 *cxxopts* 库。这就是我们现在将要探索的内容。

63.2.1 传统命令行解析

命令行选项通过 (可选的) *argv* 和 *argc* 选项传递给程序，这些选项属于 *main* 程序。前者是一个字符串数组，后者是长度:

```
int main( int argc,char **argv ) { /* 程序 */ };
```

对于简单的情况，您可以自己解析这些选项:

代码:

```
1 // /argcv.cpp
2   cout << "Program name: " << argv[0]
      << '\n';
3   for (int iarg=1; iarg<argc; ++iarg)
4     cout << "arg: " << iarg
5          << argv[iarg] << " => "
6          << atoi( argv[iarg] ) << '\n';
```

Output
[args] argcv:

```
./argcv 5 12
Program name: ./argcv
arg 1: 5 => 5
arg 2: 12 => 12
./argcv abc 3.14 foo
Program name: ./argcv
arg 1: abc => 0
arg 2: 3.14 => 3
arg 3: foo => 0
```

但是，1. 这很快就会变得繁琐 2. 难以使其健壮。因此，我们将现在看到一个使这些选项处理相对容易的库。

63.2.2 Thecxxopts library

The *cxxopts* 'commandline argument parser' 可以在 https://github.com/jarro2783/cxxopts。在 *Cmake* 安装后，它是一个 '仅包含头文件' 的库。

- 包含头文件

```
#include "cxxopts.hpp"
```

which requires a compile option:

```
-I/path/to//cxxopts/installdir/include
```

- 声明选项对象：`cxxopts::Optionsoptions("programname", "Programdescription");`

- 添加选项：// 定义 ‘-n 567’ 选项：options.add_options() ("n,ntimes"," 次数 ", cxxopts::value<int>()-> 默认 _ 值 ("37")) ; /* ... */ // 读取 ‘-n’ 选项并使用：auto number_of_times= result ["ntimes"].as<int>(); cout << " 使用次数：" << number_of_times << ’ \n’ ;

- 添加数组选项

  // 定义 ‘-a 1,2,5,7’ 选项：options.add_options() ("a,array"," 值数组 ", cxxopts::value< vector<int> >()-> 默认 _ 值 ("1,2,3")) ; /* ... */ auto array = result ["array"].as<vector<int>>(); cout << " 数组：" ; for (auto a : array) cout << a << ", "; cout<< ’ \n’ ;

- 添加位置参数：

  // 定义 ‘ 位置参数 ’ 选项：options.add_options() ("keyword","whatever keyword", cxxopts::value<string>()) ; options.parse_positional({"keyword"}); /*... */ // 读取关键字选项并使用：auto keyword = result ["keyword"].as<string>(); cout << "Found keyword: " <<keyword <<’ \n’ ;

- 解析选项：`auto 结果 = options.解析(argc,argv);`

- 获取帮助标志：`options.添加_options()("h,help","使用信息");/*... */auto 结果 = options.解析(argc,argv);`

```
if (result.count("help")>0) {
  cout << options.help() << '\n';
  return 0;
}
```

- 获取选项和参数的结果值： **missing snippet cxxoptget**

选项可以按常规方式指定：

我的程序 -n 10 我的程序 --nsize 100
我的程序 --nsize=1000 我的程序 --
array 12,13,14,15

练习 63.1. 将此包集成到素性测试中：练习 45.25.

选项解析可能抛出一个 *cxxopts*::异常::选项_没有_值 异常。

63.2.3 Cmake 集成

The *cxxopts* 包可以通过 *CMake* 通过 *pkgconfig*:

*find_* 包（*PkgConfig REQUIRED*）*pkg_* 检查_模块（ *CXXOPTS REQUIRED cxxopts*）目标_包含_目录（ *${PROJECT_NAME}* *PUBLIC ${CXXOPTS_INCLUDE_DIRS}*）

63.3 Catch2 单元测试

测试一个简单的函数

```
// /require.cppint five()
{return 5;}
```

测试成功：

代码：

```
1// /require.cpp_
2 TEST CASE( "needs to be 5","[1]" ) {
3     REQUIRE( five()==5 );
4 }
```

Output

```
[catch] require:

Filters: [1]
===============================================================
All tests passed (1
    assertion in1 测试用例）
```

测试失败：

Code:

```
1 // /require.cpp
2 TEST_CASE( "not six","[2]" ) {
3     REQUIRE( five()==6 );
4 }
```

Output
[catch] requirerr:

```
require.cxx:30: FAILED:
  REQUIRE( five()==6 )
with expansion:
  5 == 6

===============================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

抛出异常的函数:

```
    // /require.cppvoid
even( inte ) {if (e%2==1)
throw(1);cout<< "偶数:"<<
e<< '\n';}
```

Test that it throws or not:

代码:

```
1  /// cpp2TEST_CASE( "even fun", "" ) {
2      REQUIRE_NOTHROW(even(2));
3      要求_抛出(偶数(3));
4  }
```

输出
[catch] 要求偶数:

```
过滤器: [3]
偶数: 2
===============================================================
所有测试通过(2
    断言在 1 测试用例)
```

对一组数字运行相同的测试:

代码:

```
1 // /require.cpp
2 TEST_CASE( "even set","[4]" ) {
3     int e = GENERATE( 2,4,6,8 );
4     REQUIRE_NOTHROW(even());
5 }
```

输出
[catch] requiregen:

```
过滤器: [4]
偶数: 2
偶数: 4
偶数: 6
偶数: 8
===============================================================
所有测试通过(4
    在 1 个测试用例中)
```

这与使用循环有什么不同? 使用 *GENERATE* 将每个值作为单独的程序运行。

变体:

```
int i = GENERATE( range(1,100) );
int i = GENERATE_COPY( range(1,n) );
```

63.4 线性代数库

线性代数运算在许多科学应用中都有体现。例如，许多偏微分方程问题需要求解线性方程组，或特征值问题。近年来，机器学习又为矩阵运算提供了另一种应用。

因此，许多软件库实现了常见的线性代数运算也就不足为奇了。这些库不仅节省了大量输入工作，而且通常比你自己实现的版本具有更高的性能和更好的数值特性。

一些常见的库有：

- *BLAS*。这个库包含对矩阵和向量的相对简单的操作。请注意，‘BLAS’ 更是一个接口规范而不是一个实际的库。Fortran 中有一个 ‘参考实现’，但更高质量的实现提供了开源的 *BLIS* 包，以及像 Intel 的 *MKL* 这样的专有包。参考实现是 Fortran 的事实意味着从 C++ 调用 BLAS 操作可能会显得不必要地复杂。
- *Lapack*。基于 BLAS 操作，这提供了求解密集线性系统和特征值求解器的解决方案，同样，有一个 Fortran 的参考实现。
- *Eigen*。这是一个完全基于 C++ 的库，提供了对 BLAS 和 Lapack 功能的优雅访问，以及稀疏矩阵操作。对于严肃的科学计算，其缺点是对并行性的支持较差。
- 最后，有一些设计为并行执行的库：*PETSc* 、 *Trilinos* 、 *Hypre* 、 *Mumps*。它们不一定用 C++ 编写，因此可能会感觉有点不地道。然而，它们的功能足够有价值，以至于你会愿意忍受这一点。

63. 外部库

第 64 章

编程策略

64.1 编程哲学

Yes, your code will be executed by the computer, but:

- 你需要能够在一个月或一年后理解自己的代码。
- 其他人可能需要理解你的代码。
- ⇒ 让你的代码易于阅读，而不仅仅是高效

- 不要浪费时间让你的代码高效，直到你知道那段时间实际上会得到回报。

- 克努特：‘过早优化是万恶之源’。
- ⇒ 首先让你的代码正确，然后再担心效率

- Variables, functions, objects, form a new ‘language’:
  code in the language of the application.
- ⇒ your code should look like it talks about the application, not about memory.
- Levels of abstraction: implementation of a language should not be visible on the use level of that language.

64.2 编程：自顶向下与自底向上

第 45 章的练习是按复杂度递增的顺序排列的。你可以想象以这种方式编写程序，这在形式上被称为自底向上 编程。

然而，要以这种方式编写一个复杂的程序，你真的需要对整个程序的结构有一个整体的构想。

也许反过来做更有意义：从最高级别的描述开始，逐步细化到最低级别的构建模块。这被称为自上而下编程。

https://www.cs.fsu.edu/~myers/c++/notes/stepwise.html

示例：

64. 编程策略

运行模拟变为运行模拟：设置
数据和参数 直到收敛：执行时
间步变为

运行模拟：设置数据和参数：分
配数据结构 设置所有值 直到收敛：
执行时间步：计算雅可比矩阵 计
算时间步 更新

你可以将这些细化步骤在纸上完成，最终得到完成的程序，但每个被细化的步骤也可以是一个子程序。

我们已经进行了一些自顶向下的编程，当素数练习要求你编写函数和类来实现给定的程序结构时；例如练习 45.8。

自顶向下编程的问题在于，你必须在到达代码的基本部分之前完成所有步骤才能开始测试。而自底向上则更容易开始测试。这让我们想到了 ...

64.2.1 已解决的示例

查看练习 6.13。我们将逐步解决此问题。

1. 陈述问题： // 查找最长序列

2. 通过引入循环进行细化 // 查找最长序列： // 尝试所有起始点 //
如果它给出更长的序列则报告

3. 引入实际循环： // 尝试所有起始点 for (int starting=2; starting<1000;
starting++) { // 如果它给出更长的序列则报告 }

4. 记录长度：
// 尝试所有起始点 int 最大 _ 长度
=-1;

for (int starting=2; starting<1000; starting++) { // 如果从 ‘start’ 开始的序列能生成更长的序列报告 : int length=0; // 从 ‘start’ 计算序列 if (length>maximum_length) { // 报告这个序列为最长 } }

5. 精炼计算序列 : // 从 ‘start’ 计算序列 int current= starting; while (current!=1) { // 更新当前值 length++; }

6. 精炼更新当前值 : // 更新当前值 if (current%2==0) current /= 2; else current= 3*current+1;

64.3 编程风格

在您编写完代码后，就会有 代码维护 的问题：您将来可能需要更新代码或修复某些问题。您甚至可能需要修复别人的代码，或者别人将不得不处理您的代码。因此，编写清晰的代码是个好主意。

命名 使用有意义的变量名：record_number 而不是 rn 或 n。这有时被称为 “ 自文档化代码 ”。

注释 插入注释来解释代码的非平凡部分。**重用** 不要重复编写相同的代码片段：使用宏、函数、类。

64.4 文档

看看 Doxygen。

64.5 最佳实践：C++ 核心指南

C++ 语言很大，某些特性的组合并不建议使用。大约在 2015 年制定了一系列核心指南 ，这将大大提高代码质量。请注意，这并非关于性能：指南基本上没有性能影响，但能带来更好的代码。

例如，当处理多个构造函数时，指南建议尽可能使用默认值：

```
class Point { // 不要这样 private: double d; public:
Point( double x, double y, double fudge ) { auto d
= ( x*x +y*y ) * (1+fudge); }; Point( double x,
double y ) { auto d = ( x*x +y*y ); }; };
```

这是由于代码重复而不好。稍好一点 ：

```
class Point { // 不要这样 private: double d; public:
Point(double x, double y, double fudge ) { auto d = (
x*x +y*y ) * (1+fudge); }; Point(double x, double y) :
Point(x,y,0.) {}; };
```

如果 fudge 为零，这会浪费几个周期。最好：

```
类 Point{ // not this wayprivate:double
d;public:Point(double x,double y,double
fudge=0. ) {auto d = ( x*x +y*y ) * (1+
fudge); };};
```

第 65 章

性能优化

本节讨论随机游走练习中的性能和代码优化问题。

65.1 问题陈述

1904 年，发现疟疾由蚊子传播的生物学家罗纳德 · 罗斯爵士，发表了一篇题为 ‘ 蚊虫减少卫生政策的逻辑基础 ’ 的演讲。在演讲中，他考虑了蚊子能飞多远的问题，因此，需要将所有可能孳生蚊子的池塘排干多远。

我们可以将蚊子建模为在其生命周期中的每个时间周期内飞行某个单位距离，比如一天。然而，由于蚊子随机飞行，它在 N 天的生命周期内不会覆盖 N 的距离。那么，从统计学的角度来看，它能飞多远呢?

Ross 只能计算一维蚊子的情况，也就是说，只能决定在线上前进或后退的蚊子。在这种情况下，蚊子平均会以 $\sqrt{N}$ 的距离远离它开始的地方。

更一般的问题是由卡尔 · 皮尔逊于 1905 年引入数学家的注意，并且发现早在 1880 年就被雷利勋爵解决了。现在这被称为一个 随机游走 问题。有点令人惊讶的是，在 2D 中，蚊子也预计会旅行 $\sqrt{N}$，其中 N 是时间步数。

一般的 d- 维问题稍微难一些，蚊子行进的距离略小于 $\sqrt{N}$。让我们以所有普遍性来编写代码。

65.2 编码

主程序设置:

代码:

```
// /walk_vec.cpp
   float avg_dist{0.f};
   for ( int x=0; x<experiments; ++x
) {
      蚊子 m(dim);
      for (int step=0; step<steps;
++步)
         m.步();
      avg_dist+= m.距离();
   }
   avg_dist /= experiments;
```

输出 [rand] vec:

```
D=3 after 10000 steps,
    p距离 = 83.7997

D=3 after 100000 steps,
    距离 = 224.372
D=3 after 1000000 步,
    距离 = 922.599
产品耗时: 2776
    毫秒
```

where the `Mosquito` 类存储其位置:

```
// /walk_lib_vec.cpp类 蚊子{私有: 向
量<浮点数> 位置; 公共: 蚊子( 整数 d ): 位
置( 向量<浮点数>(d,0.f)){};
```

并且 步骤方法更新这个:

```
// /walk_lib_vec.cpp void 步骤 () { 整数
d = 位置 .size(); 自动 增量 = 随机_步骤
(d); for( 整数 id=0;id<d; ++id) 位置 .
at(id) += 增量 .at(id); };
```

随机步骤方法生成一个随机坐标，归一化到单位圆上。这里有一个小问题：如果我们在一个单位立方体内生成一个随机坐标，然后归一化它，它将偏向立方体的角。因此，我们迭代直到得到一个单位圆内的坐标，并使用那个进行归一化:

```
// /walk_lib_vec.cpp 向量<浮点数>随机_坐标(整数 d){自动 v= 向
量<浮点数>(d); for(自动 & e: v) e= 随机_浮点数 (); 返回 v; }; // /
walk_lib_vec.cpp 向量<浮点数> 随机_步骤(整数 d){for (;;){自动
步骤 = 随机_坐标 (d); if(自动 l= 长度(步骤); l<=1.f){if(l==0.f){/*
* 零长度可能会发生，对于 d==1
```

```
    * 但不应用于更高的 d。 */assert(d==1);}
else {normalize(step,l);return
step;}}}};
```

练习 65.1。 取基本代码，并基于

```
模板<int d>类蚊子{ /*...
*/
```

这简化了你的代码多少？你获得了任何性能提升 ？

你可以基于代码仓库中的文件 *walk_vec.cxx*

65.2.1 优化：在分配时保存

可能的主要问题是每个步骤都会创建多个向量。这种内存管理相对昂贵，尤其是由于对其中每个向量执行的操作非常少。

因此我们将向量的创建移到计算例程之外。随机坐标现在写入一个作为参数传递的数组中：

```
    //walk_lib_pass.cppvoidrandom_
coordinate( vector<float>& v ) {for ( auto&
e : v)e= random_float();};
```

同样地，随机步长：

```
    //walk_lib_pass.cppvoid random_step(
vector<float>&step ) {for(;;) {random_
coordinate(step);
```

这个过程在将数组传递给步骤方法时停止，我们希望保持该方法无参数。因此，我们在构造函数中添加了一个缓存选项，以存储步向量以及位置：

代码:

```
// /walk_lib_pass.cpp
class Mosquito {
private:
  vector<float> pos;
  vector<float> inc;
  bool cache;
public:
  Mosquito( int d,bool cache=false )
    : pos( vector<float>(d,0.f)
    ),cache(cache) {
    if (cache) inc =
    vector<float>(d,0.f);
  };
```

Output
[rand] pass:

```
D=3 after 10000 steps,
    distance= 76.7711
D=3 after 100000 steps,
    distance= 257.19
D=3 after 1000000 steps,
    distance= 956.122
run took: 2852 milliseconds
D=3 after 10000 steps,
    distance= 87.034
D=3 after 100000 steps,
    distance= 256.655
D=3 after 1000000 steps,
    distance= 912.033
run took: 1762 milliseconds
```

// /walk_lib_pass.cpp void step() {
intd = pos.size(); if (缓存) {
random_step(inc); step(inc); }
else { vector<float> incr(d);
random_step(incr); step(incr); } };

65.2.2 静态向量中的缓存

仍然存在长度 计算问题。由于 “ 平方和 ” 没有归约运算符，我们需要为平方创建一个临时向量，以便对其进行加法归约。

练习 65.2。 探索这个临时的选项。讨论最优雅的方法，并测量性能提升。

- This temporary can be passed in as a parameter;
- it can be stored in a global variable;
- or we can declare it `static`.
- With the C++20 standard, you could also use the `ranges` header.

65.3 向量 vs 数组

在这个模拟中，我们将主要在 2D 空间工作，因此我们可以使用静态分配的数组 对象，而不是使用 向量。这允许更优雅的功能接口:

```
// /walk_lib_
arr.cpp模板<int d>
```

*float length(const array<float,d>& step) { array<float,d> square =step; for_ each(square.begin(),square.end(),[] (float& x) { x *= x; }); auto l= sqrt (std::accumulate(square.begin(),square.end(),0.f)); return l; };*

虽然我们在上面去除了所有不必要的分配，但通过编译器知道数组的长度，我们获得了额外的性能提升。因此，编译器可能会用两个显式指令来替换长度为两个的循环。

代码：

```
1 // /walk_arr.cpp
2       float av {}; for(int x=0;x< 实验 ; ++x
3
      ) {
4         蚊子 < 尺寸 > 米
5         for(int step=0; step< steps;
      ++ 步)
6           . 步 ();
7         avg_dist+= m. 距离 ()
8       }
9       avg_dist/= 实验 ;
```

输出 [rand] arr:

```
D=3 经过 <10000> 步 <, >
    距离 = 76.3221
D=3 after 100000 steps,
    距离 = 247.5
D=3 经过 1000000 步,
    距离 = 959.735
产品用时 : 358
    毫秒
```

65. 性能优化

第 66 章

关于算法和数据结构的简要介绍

66.1 数据结构

到目前为止，你看到的主要数据结构是数组。在本节中，我们简要概述一些更复杂的数据结构。

66.1.1 栈

A *stack* is a data 一种有点像数组的数据结构，但只能 see the last element:

- 您可以检查最后一个元素；
- 您可以移除最后一个元素；和
- 您可以添加一个新元素，然后它将成为最后一个元素；之前的最后一个元素变得不可见：如果新的最后一个元素被移除，它又会变得可见。

添加和移除最后一个元素的操作分别称为 *push* 和 *pop* 。

Exercise 66.1. Write a class that implements a stack of integers. It should have methods

```
void push(int value);
int pop();
```

66.1.2 链表

在进行本节之前，请确保您已学习第 *16* 节。

数组不够灵活：您不能在中间插入元素。相反：

- 分配一个更大的数组，
- 复制数据（并插入），
- 删除旧数组存储

这很昂贵。（这是 C++ 向量；第 10.3.2 节中发生的情况。）

如果你需要做大量的插入操作，请使用链表。基本数据结构是一个 节点，它包含

1. 信息，可以是任何内容；和 2. 一个指向下一个节点的指针（有时称为 ‘链接’ ）。

如果没有下一个节点，指针将是 *null*。每种语言都有其表示空指针的方式；C++ 有 `nullptr`，而C 使用的是 `NULL` ，它不过是零值的同义词。

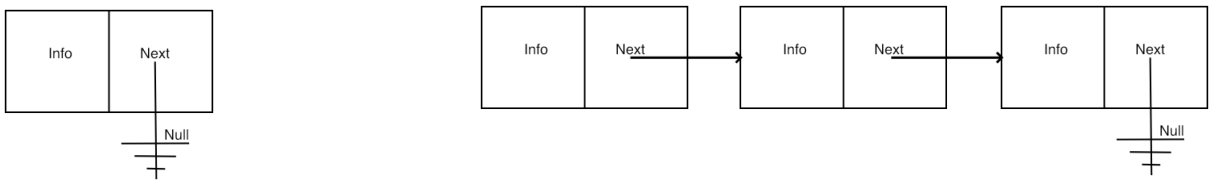


图 66.1：节点数据结构和节点链表

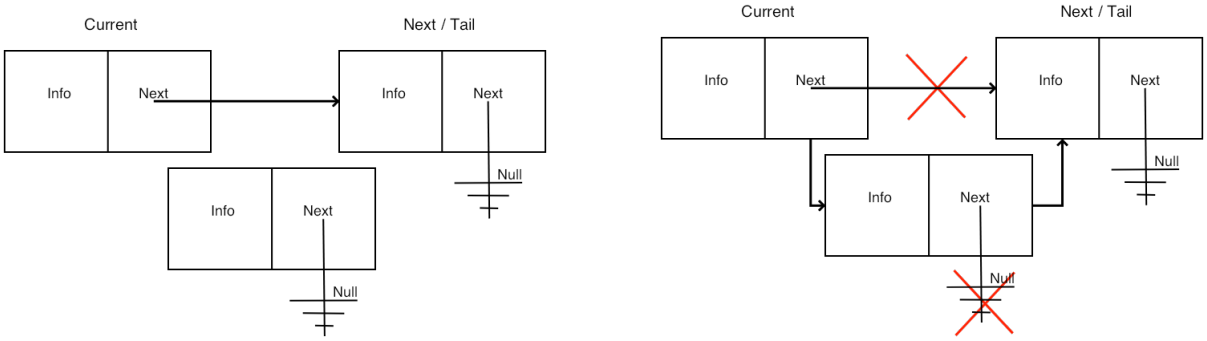


图 66.2：链表中的插入

我们在图中说明这一点 66.1。

我们的主要关注点将是实现报告列表某些统计信息的操作，例如其长度，测试列表中是否存在信息，或更改列表，例如通过插入一个新节点。见图 66.2。

66.1.2.1 数据定义

在 C++ 中，你可以选择指针类型。目前，我们将使用共享 `_ptr` ；稍后我们将使用唯一 `_ptr`。

我们声明基本类。首先声明 `List` 类，该类有一个指针，对于空列表为空，否则指向第一个节点。

链表只有一个成员，即指向节点的指针：

```
        // /linkshared.cppclass List
{private:   shared_ptr<Node>
head{v22}{      nullptr   };public:
List() {};
```

对于空列表，初始为空。

接下来，`Node` 有一个信息字段，我们这里选择它为一个整数，并有一个计数器来记录某个数字出现的次数。Node 还有一个指向下一个节点的指针。这个指针对于列表中的最后一个节点再次为空。

一个节点包含信息字段，并指向另一个节点的链接：

```
1 //linkshared.cpp 2 类 Node { 3 private: 4 int
datavalue{0},datacount{0}; 5 shared_ptr<Node>next{nullptr}; 6
public: 7 Node() {} 8 Node(int value,shared_ptr<Node> next=
nullptr) 9 : datavalue(value),datacount(1),next(next) {}; 10 int
value() { 11 return datavalue; }; 12 auto nextnode() { 13 return
next; };
```

一个空指针表示链表的尾部。

我们现在将为 `List` 和 `Node` 类开发方法，以支持以下代码。

列表测试和修改。

```
List mylist;
cout << "Empty list has length: "
     << mylist.length() << '\n';

mylist.insert(3);
cout << "After one insertion the length is: "
     << mylist.length() << '\n';
if (mylist.contains_value(3))
  cout << "Indeed: contains 3" << '\n';
```

让我们从一些简单的函数开始。

66.1.2.2 简单函数

对于许多算法，我们可以在迭代版本和递归版本之间进行选择。递归版本更容易编写，但迭代解决方案可能更高效。

我们从打印链表的简单工具函数开始。（这个函数有点粗糙；更好的解决方案使用了第 12.4 节中的策略。）这个实现展示了递归策略。

辅助函数，以便我们可以跟踪我们在做什么。

打印列表头:

```
// /linkshared.cpp
void print() {
   cout<< "List:";
   if (has_list())
      cout "<<"=> ">print();t'<< '
   cou        n ;
};
```

打印节点及其尾部:

```
// /linkshared.cpp
void print() {
   cout << datavalue<<:          <<
   datacount;
   if (has_next()) {
      cout <<, ; next - print();
   }
};
```

对于递归计算列表长度的场景，我们采用相同的递归方案 .

对于列表:

```
// /linkunique.cpp
int 递归 _ 长度 ()          {
   if (!has_list())
      返回 0;
   else
      return head->listlength();
};
```

对于节点:

```
// /linkunique.cpp
int listlength_ recursive() {
   if (!has_next()) return 1;
   else return  1+next- listlength_recursive();
};
```

一个迭代版本使用一个指针沿着列表向下移动，在每一步中递增一个计数器。

使用一个共享指针沿着列表向下移动:

```
// /linkshared.cpp int length_ iterative() { int count  = 0; if (has_ list()) {
auto current_ node = head; while (current_ node-> has_ next()) { current_
node =  current_ node-> nextnode(); count  += 1; } } return count; };
```

(有趣的练习：可以做一个迭代释放列表的函数?)

练习 66.2. 写一个函数

```
bool List::contains_value(int v);
```

用于测试一个值是否存在于列表中。

This can be done recursive and iterative.

66.1.2.3 修改函数

当然，有趣的 方法 是那些会改变列表的。在列表中插入一个新值基本上有两种情况：

1. 如果列表为空，创建一个新节点，并将列表的头设置为该节点。2. 如果列表不为空，我们有几个更多的情况，取决于值是放在列表头、列表尾，还是列表中间。并且我们需要检查值是否已经在列表中。

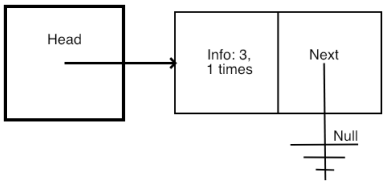
我们将编写函数

void List::insert(int value);void Node::insert(int value);

那些将值添加到列表中的。The`List::insert` 值可以将一个新节点插入到第一个节点之前；the`Node::insert` 假设该值大于或等于当前节点的值。

这里有大量情况。你可以通过一种称为测试驱动开发（ TDD ）的方法来尝试：首先你确定一个测试，然后 you write thecodethatcovers that case。

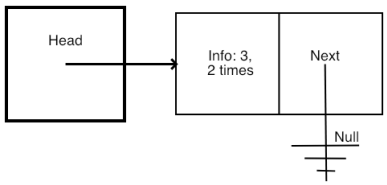
步骤 ***1***：插入第一个元素 将第一个元素添加到空列表很简单：我们需要头节点的指针指向一个 `Node`。



练习 66.3. 接下来编写处理空列表的 `Node::insert` 情况。你还需要一个方法 `List::contains` 来测试一个项目是否在列表中。

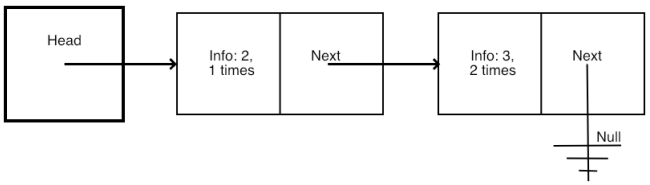
```
// /linkshared.cpp mylist.insert(3); cout <<"After
inserting 3 the length is: "<<mylist.length() << ’ \n’; if
(mylist.contains_value(3)) cout << "Indeed: contains 3"
<< ’ \n’; else cout << "Hm. Should contain 3" <<’ \n’;
if (mylist.contains_value(4)) cout << "Hm. Should not
contain 4"<< ’ \n’; else cout << "Indeed: does not
contain 4" <<’ \n’; cout <<’ \n’;
```

步骤 ***3***：插入一个已存在的元素 如果我们尝试向一个已存在的列表中添加一个值，插入操作不会做任何事；如果需要，我们可以增加包含该值的 节点 中的计数器。



练习 66.4. 将一个已经在列表中的值插入意味着节点的计数值需要增加。更新你的插入方法以使这段代码工作：

```
// /linkshared.cpp mylist.insert(3); cout <<" 插入相同的项目给出长
度 : "<<mylist.length() << ’\n’; if (mylist.contains_value(3)) {
cout << " 确实 : 包含 3"<< ’\n’; auto headnode =
mylist.headnode(); cout << " 头节点有值 " <<headnode->value()
<< " 和计数 "<< headnode->count() << ’\n’; } else cout << " 嗯。
应该包含 3" <<’\n’; cout <<’\n’;
```

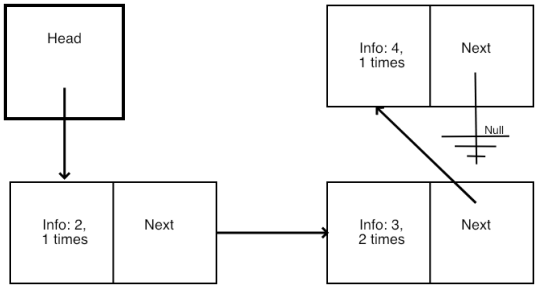


步骤 ***4:*** 在头部插入一个元素

练习 66.5. 插入的一个情况是元素位于头部。更新你的插入方法以使其工作：

```
// /linkshared.cpp mylist.insert(2); cout <<" 插入 2 位于头部；\n 现在长度是：
"<<mylist.length() << ’\n’; if (mylist.contains_value(2)) cout << " 确实：
包含 2"<< ’\n’; else cout << " 嗯。应该包含 2" <<’\n’; if
(mylist.contains_value(3)) cout << " 确实：包含 3"<< ’\n’; else cout <<
" 嗯。应该包含 3" <<’\n’; cout <<’\n’;
```

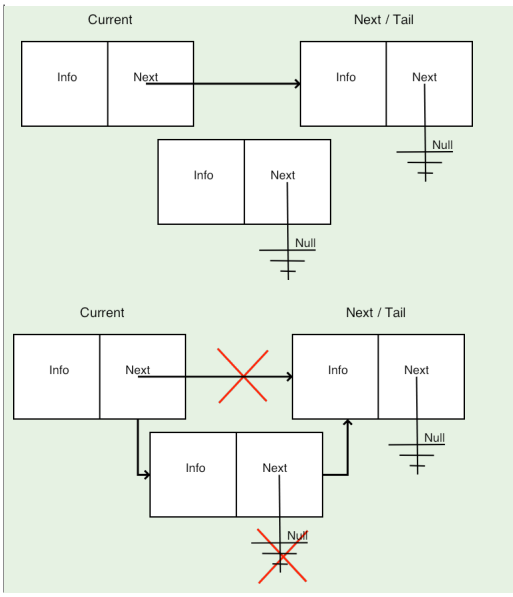
步骤 ***5***：在末尾插入一个元素 向尾部添加元素需要遍历整个列表。



练习 66.6. 如果一个元素放在列表的末尾：

```
// /linkshared.cpp mylist.insert(6); cout <<" 插入 6 放在尾部；\n 现在长度是：
"<< mylist.length()<< ’ \n’; if (mylist.contains_value(6)) cout << " 确实：
包含 6"<< ’ \n’; else cout << " 嗯。应该包含 6" <<’ \n’; if
(mylist.contains_value(3)) cout << " 确实：包含 3"<< ’ \n’; else cout <<
" 嗯。应该包含 3" <<’ \n’; cout <<’ \n’;
```

步骤 *6*：在中间插入元素 最复杂的情况是在列表的中间某个位置插入元素。现在你需要比较当前元素和下一个元素，以决定是插入元素还是继续移动到尾部。



练习 66.7. 更新你的插入例程，以处理需要插入到中间位置的元素。

```
// /linkshared.cpp mylist.insert(4); cout <<" 插入 4 到中间；\n 现在长度是 : "
<< mylist.length()<< ’\n’; if (mylist.contains_value(4)) cout << " 确实：
包含 4"<< ’\n’; else cout << " 嗯。应该包含 4" <<’\n’; if (
mylist.contains_value(3)) cout << " 确实：包含 3"<< ’\n’; else cout << " 嗯。
应该包含 3" <<’\n’; cout <<’\n’;
```

66.1.2.4 高级：使用唯一指针

从概念上讲，我们可以说列表对象拥有第一个节点，每个节点拥有下一个节点。因此，最合适的指针类型是唯一的 _ptr。

我们也可以用唯一指针来完成这个操作：

链表只有一个成员，即指向一个节点的指针：

```
///linkunique.cpp 类 List{ 私有 :
    唯一 _ 指针 < 节点 > 头 {nullptr};
公有 :    List() {};
```

对于空链表，初始值为空。

节点包含信息字段，并指向另一个节点。 de:

```
1 // /linkunique.cpp 2 类 Node { 3 朋友类 List; 4 私有 : 5 int
datavalue{0},datacount{0}; 6 唯一 _ 指针 < 节点 >next{nullptr}; 7 公有 : 8 朋友类
List; 9 Node() {} 10 Node(int value, 唯一 _ 指针 < 节点 > tail=nullptr) 11 :
datavalue(value),datacount(1),next(move(tail)) {};12 ~Node() { cout <<" 删除节
点 " << datavalue << ’\n’; };
```

一个空指针表示列表的尾部。

Above we formulated an iterative and a recursive way of computing the length of a list. The iterative

代码有一个共享指针指向连续的列表元素。我们不能用唯一指针这样做。相反，这是一个使用裸指针的好地方。

Use a *bare pointer*, which is appropriate here because it doesn't own the node.

```
// /linkunique.cppint listlength() {Node *walker = ;
walker->next().get();int len = 1;;while ( walker!=
nullptr ) {walker =walker{v47}-next.>next().get(); len++
len{v61};}return len;};
```

(如果你尝试将 `walker` 设为智能指针，你会得到一个编译器错误：你不能复制唯一指针。)

- 使用智能指针表示所有权
- 使用裸指针表示指向但不拥有。
- 这是一个效率方面的论点。我并不完全信服。

66.1.3 树

在进入本节之前，请确保您已学习完第 *16* 节。

树可以递归地定义：

- 树为空，或
- 树是一个节点，带有若干子树。

让我们设计一个用于存储和计数整数的树：每个节点都有一个标签，即一个整数，以及一个计数值，用于记录我们见过该整数的次数。

我们的基本数据结构是节点，并且我们递归地定义它最多有两个子节点。这是一个问题：你不能写出

```
类
Node{private:Node
left, right; }
```

因为那样会递归地需要无限内存。所以我们使用指针。

```
// /binary.cpp 类 Node { private: int key{0},count{0};
shared_ptr<Node> left,right; bool
hasleft{false},hasright{false}; public: Node() {} Node(int i,i
nt init=1 ) { key = i;count = 1; }; void addleft(int value) {
left = make_shared<Node>(value); }
```

```
    hasleft =true; }void addright(int value )
{right make sharedNode(value);hasright tr
ue; };/* ... */};
```

我们记录下来，已经看到了整数零零次。

树上的算法通常是递归的。例如，节点总数是从根节点计算的。在任何给定节点，该连接的子树中的节点数是左子树和右子树节点数之和加一。

```
// /binary.cpp int number_of_nodes() { int count
 = 1; if (hasleft) count+= left->number_of_
nodes(); if (hasright) count+= right->number_of_
nodes(); return count; };
```

同样地，树的高度是通过左子树和右子树的递归最大值计算的：

```
// /binary.cpp intdepth() { int d
 = 1, dl=0,dr=0; if (hasleft) dl=
left->depth(); if (hasright) dr=
right->depth(); d =max(d+dl,d+
dr); return d; };
```

现在我们需要考虑如何实际插入节点。我们编写一个在节点处插入项的函数。如果该节点的键是项，我们就增加计数器的值。否则我们确定是将项添加到左子树还是右子树。如果没有这样的子树，我们就创建它；否则我们进入适当的子树，并进行递归插入调用。

```
// /binary.cpp void insert(int value)
{ if (key== < value++) count++; else
if (value< < key) { if (hasleft) left->
insert(value); else addleft(value); }
else if (value> > key) { if (hasright)
right->insert(value); else
```

```
    addright(value);} elsethrow(1);// 不
应该发生 };
```

66.1.4 其他图

树中的节点具有从父节点到子节点的关系，因此它们是 有向图 的一个特例。

练习 66.8. 如果你了解图与其 邻接矩阵 之间的关系，你能用矩阵属性来表达有向性吗?

有向图的一个重要子集，即 DAG，具有树所没有的属性：对于某些节点对，它们之间可能有超过一条路径。

更多详情，请参阅 HPC 书籍 [11], 章节 20。

66.2 算法

这真的 **真的** 超出了这本书的范围。

- 简单的：数值
- 与数据结构相关：搜索

66.2.1 排序

与上述树形算法不同，后者使用了非直观的数据结构，排序算法是简单数据结构（主要是数组）与复杂算法行为分析的典型结合示例。我们简要讨论两种算法。

标准库内置了排序例程；参见第 14.3.6 .

66.2.1.1 冒泡排序

长度为 n 的数组 a 被排序当且仅当

$$\forall_{i<n-1}: a_i \leq a_{i+1}.$$

一种简单的排序算法立即显现：如果 i 满足 $a_i > a_{i+1}$，则反转数组中的 i 和 $i+1$ 位置。

```
    //bubble.cppvoid swapij(vector< int>  &
array{v17}, int{v21} i{v23}) {   int t= ; array[i];
array[i] = ; array[i+1]; array[i+1] =; t{v60};}
```

(为什么数组参数是通过引用传递?)

如果你遍历数组一次，交换元素，结果可能未排序，但至少最大的元素会移到末尾。你现在可以再遍历一次，将下一个最大的元素放在合适的位置，依此类推。

这个算法被称为 冒泡排序。它通常被认为不是一个好的算法，因为它的时间复杂度（第 70.1.2节）为 $n^2/2$ 交换操作。排序需要 $O(n\log n)$ 操作，而冒泡排序远远超出了这个限制。

66.2.1.2 快速排序

一个可以取得最优复杂度（但不必如此；见下文）的流行算法是 快速排序：

- 找到一个称为枢轴的元素，它大约等于中值。
- 重新排列数组元素，以给出三个连续存储的集合：所有小于、等于和大于枢轴的元素分别。
- 将快速排序算法应用于第一个和第三个子数组。

这种算法最好递归编程，甚至可以为其并行执行辩护：每次你找到一个枢轴时，都可以将活动处理器的数量加倍。

练习 66.9。 假设，由于不幸，你的枢轴每次都是数组中最小的元素。由此产生的算法的时间复杂度是多少?

66.2.2 图算法

我们简要讨论了一些图算法。进一步讨论请参见 HPC 书籍 [11], 第 10 章。

首先考虑 SSSP 问题：给定一个图和一个起始节点，到其他每个节点的最短路径是什么。最初，我们考虑一个 无权图，或者等价地，将任何一对连通节点之间的距离设置为 1。

该算法按层级进行：每个下一层级包含已探索节点的邻居。

- 将起始节点的距离设置为 0。
- 直到完成,
- 遍历所有已知距离的节点，并
- 对于其所有邻居，– 除非邻居已经有一个已知的

  距离 – 将距离设置为 $d+1$,

我们使用一个简单的数据结构来存储距离：

```
map<int, int> distances; intstarting
node_; node = 0; distances[starting
node_; node] = 0;
```

直到所有节点都被映射，我们迭代所有已知距离的节点，并设置其邻居的距离：

```
// while not donefor(;;) {int
updates{0};// 迭代所有已映射的节点
```

```
    for (auto [节点, 距离] : 距离) {// 对每个节点，遍历其
所有邻居 }}}
```

假设图数据结构支持获取节点的邻居列表：

```
for ( const auto& neighbor : graph.neighbors(node) ) { if ( auto find_
neighbor= 距离 .find(neighbor) ; find_neighbor== 距离 .end() ) { 距离
[neighbor] = 距离 +1; 更新 ++; } }
```

由于我们使用的是 until-done 循环，我们需要显式跳出循环。一个简单的测试条件可以是‘如果映射的节点数量等于图中的节点数量’。

练习 66.10. 你能发现这个问题吗？有什么解决方法吗？

上述算法有些浪费：在每次遍历中它会遍历所有映射的节点，但我们只需要新添加的那些。因此我们添加：

```
set<int> current_front; current_front.insert(starting_node); for (;;) { set<
int> next_front; // 遍历 current_front for ( auto node :current_front ) { //
遍历邻居 for (const auto& neighbor : graph.neighbors(node) ) { // 如果邻居
尚未映射 if ( /* ... */ ) { distances [neighbor] =dist_to_this_node+1; next_
front.insert(neighbor); } } } current_front= next_front; }
```

66.3 编程技巧

66.3.1 Memoization

在 7.6 节 中你看到了一些递归的例子。阶乘的例子可以用循环来写，并且无论是 for 还是 do...while 都可以这样做。

斐波那契的例子更微妙：它不能立即转换为迭代形式，但简单的递归形式确实存在一些浪费，需要消除。我们可以使用的技术称为 备忘录化：存储中间结果以防止它们被重复计算。

这里有一个大纲。

```
// /fibomemo.cpp intfibonacci(int n) { vector<int> fibo_values(n); for (inti=0;i<
n; ++i) fibo_values [i] = 0; fibonacci_memoized(fibo_values,n-1); return fibo_
values [n-1]; } intfibonacci_memoized( vector<int> &values, inttop) { int
minus1 =top-1, minus2 =top-2; if (top<2) return 1; if (values [minus1]==0)
values [minus1] =fibonacci_memoized(values,minus1); if (values [minus2]==0)
values [minus2] =fibonacci_memoized(values,minus2); values [top] =values [
minus1]+values [minus2]; //cout<<"set f("<<top<< ") to "<<values [top] <<'\n';
return values [top]; }
```

第 67 章

可证明正确的程序

编程常常更像是一种艺术，甚至是一种黑色艺术，而不是一门科学。然而，人们已经尝试用系统的方法来确保程序的正确性。我们可以区分

- 证明一个程序是正确的，或
- 编写一个程序，使其保证是正确的。

这种区别只是想象的。一个更有成效的方法是让证明来指导编码。正如 E.W. Dijkstra 指出的那样

> 提高程序置信度的唯一有效方法是为其提供令人信服的正确性证明。但不应先编写程序再证明其正确性，因为那样只会增加糟糕程序员的负担。相反：程序员应让正确性证明和程序同步发展。

我们将看到几个这样的例子。

67.1 循环作为量词

通常，算法可以用数学表达。在这种情况下，你的程序应该看起来像数学。这里我们考虑一个例子，指出 for 循环和数学量词之间的联系。

67.1.1 Forall-quantor

考虑一个简单的例子：测试一个数是否为质数。谓词 ‘isprime’ 可以表示为：

$$\text{isprime}(n) \equiv \forall_{2 \leq f < n} : \neg \, \text{divides}(f, n)$$

您看到，证明原始 `isprime` 谓词对于某个值 n 现在涉及

1. 一个关于新变量 f 的量词 —— 我们称之为 ‘绑定变量’；2. 以及一个新的谓词 `divides`，它涉及原始变量 n 和由量词绑定的变量 f。

我们现在将 ‘全称量词’ 迭代地拼写为 asaloop，其中每次迭代都需要为真。也就是说，我们对某个与迭代相关的结果进行 ‘与’ 归约。

$$\neg\,\mathrm{divides}(2,n) \cap \ldots \cap \neg\,\mathrm{divides}(n-1,n)$$

并且这个 ‘与’ 合取的序列可以被编程：

```
for (int f=2; f<n; f++)
  isprime = isprime && not divides(f,p)
```

你注意到循环变量是量词引入的变量 f。

现在我们唯一担心的是如何初始化 `isprime`。初始值对应于一个空集上的 ‘与’ 合取，它为真，所以：

```
bool isprime{true};
for (int f=2; f<n; f++)
  isprime = isprime && not divides(f,p)
```

Exercise67.1. 现在你已经有一个计算正确结果的循环，你可以开始担心性能了。在某些情况下，循环能否过早终止？你会如何编程实现？

67.1.2 存在量词

如果我们把素数性表示为：

$$\mathrm{isprime}(n) \equiv \neg\exists_{2\leq f<n}\colon\ \mathrm{divides}(f,n)$$

为了得到一个纯粹的量词，而不是一个否定的量词，我们编写：

$$\mathrm{isnotprime}(n) \equiv \exists_{2\leq f<n}\colon\ \mathrm{divides}(f,n)$$

把存在量词拼写为

$$\mathrm{isnotprime}(n) \equiv \mathrm{divides}(2,n) \cup \ldots \cup \mathrm{divides}(n-1,n)$$

我们看到我们需要一个循环，其中测试任何迭代是否满足谓词。也就是说，我们对每次迭代的结 果进行 ‘或’ - 归约。如前所述，循环变量是量词引入的变量。

for (int f=2; f<n; f++) isnotprime isnotprime or
divides(f, p) bool isprime not isnotprime;

此外，如前所述，我们小心地正确初始化归约变量：将 $\exists_{s\in S}P(s)$ 应用于空集 S 是 **false**：

```
bool isnotprime{false};
for (int f=2; f<n; f++)
  isnotprime = isnotprime or divides(f,p)
bool isprime = not isnotprime;
```

练习 67.2. 与 ‘forall’ 量词相同的问题：这个循环能否提前终止？你会如何编写代码？

67.1.3 通过范围量词

我们有以下对应关系：

∀ all_of
∃ any_of

假设 S 是一个范围，P 是一个谓词，那么

$$\forall_{n \in S} : P(n)$$

可以表示为

```
all_of( S,[] { auto n } -> bool { return P(n); } );
```

同样地，

$$\exists_{n \in S} : P(n)$$

可以表示为

```
any_of( S,[] { auto n } -> bool { return P(n); } );
```

在以下示例中，我们 “ 证明 ” 一个关于整数的简单陈述。使用 `iota(1)` 生成所有整数会导致无限运行时间，因此我们使用第二个参数进行截断。

示例：

$$\forall_n : \exists_m : m > n$$

```
// /rangepred.cpp
  all_of( ranges::views::iota(1,20),
          [] ( auto n ) -> bool {
            return any_of( ranges::views::iota(n+1),
                           [n] ( auto m ) -> bool {
                             if (m>n) {
```

遍历更复杂的集合：

$$\forall_{\substack{n \in S \\ Q(n)}} : P(n)$$

可以表示为

```
all_of( S | filter( [] (auto n) -> bool { return Q(n) } ),
        [] { auto n } -> bool { return P(n); } );
```

练习 67.3。 编写基于范围的代码来证明

$$\forall_n : \exists_{m>n} : \text{even}(m)$$

如果你正在进行素数项目，现在可以做一些练习 45.15 。

67.2 谓词证明

对于具有清晰循环结构的程序，您可以采用类似于进行 “ 归纳法证明 ” 的方法。

Let us consider the Collatz conjecture again, where for brevity we define

$$c(\ell) = \text{ the length of the Collatz sequences, starting on } \ell.$$

现在我们考虑将 Collatz 猜想作为证明谓词

$$P(\ell_k, m_k, k) = \begin{cases} \ell_k < k \text{ is the location of the longest sequence:} \\ c(\ell_k) = m_k \text{: the length of sequence } \ell_k \text{ is } m_k \\ \text{all other sequences } \ell < k \text{ are shorter} \end{cases}$$

形式上：

$$P(\ell_k, m_k, k) = \left[\begin{array}{ll} & \ell_k < k \\ \wedge & c(\ell_k) = m_k \text{(only if } k > 0) \\ \wedge & \forall_{\ell<k} : c(\ell) \leq m_k \end{array} \right.$$

for $k = N$.

我们开发使这个谓词归纳为真的代码。我们从

$$\ell_0 = -1, \quad m_0 = 0 \Rightarrow P(\ell_0, m_0, 0).$$

归纳证明对应于一个循环：

- 我们假设在 k- 次迭代的开始时 $P(\ell_k, m_k, k)$ 为真；
- 迭代体是，在 k- 次迭代的结束时 $P(\ell_{k+1}, m_{k+1}, k+1)$ 为真；
- 这当然是在下一次迭代的开始设置谓词。

循环结构是：

k=0;{$P(l_k, m_k, k)$}*while (*
k<N) {{$P(l_k, m_k, k)$} 更新;
{$P(l_{k+1}, m_{k+1}, k+1)$}*k*=
k+1*; }*

更新必须将谓词从 k 扩展到 $k+1$。让我们考虑它的部分。

We need to establish

$\forall_{\ell<k+1}$: $c(\ell) \leq m_{k+1}$ 我们将范围 $\ell < k+1$ 分成 $\ell < k$ 和 $\ell = k$:

- 第一部分 $\forall_{\ell<k}$: $c(\ell) \leq m_{k+1}$ 如果 $m_{k+1} \geq m_k$ 则为真;
- 部分 $\ell = k$: $c(\ell) \leq m_{k+1}$ 声明 $m_{k+1} \geq c(k)$。

两者结合起来，我们得到

$$m_{k+1} \geq \max(m_k, c(k))$$

最后，子句

$$c(\ell_{k+1}) = m_{k+1}$$

可以满足:

- 如果 $c(k) > m_k$，我们需要设置 $m_{k+1} = c(k)$ 和 $\ell_{k+1} = k$。
- (严格来说，存在 $m_{k+1} > c(k)$ 的可能性。这是不可能的，因为我们不能对任何 k 满足 $m_{k+1} = c(\ell_k)$。)
- 如果 $c(k) \leq m_k$，我们需要设置 $m_{k+1} \geq m_k$。同样，$m_{k+1} > m_k$ 不能被任何 ℓ_{k+1} 满足，所以我们得出 $m_{k+1} = m_k$。

67.3 Flame

前面，你看到了 Dijkstra 的一句名言，他认为测试不足以证明程序的正确性。那么，Dijkstra 是如何设想确保正确性的呢？这可以在他的名言的第二部分中找到:

> 提高程序置信度的唯一有效方法就是给出其正确性的令人信服的证明。但是，不应该先编写程序，然后再证明其正确性，因为那样的话，提供证明的要求只会增加糟糕程序员的负担。相反：程序员应该让正确性证明和程序一起成长。

让我们用矩阵 - 向量乘法作为一个简单的例子来发展这一点：我们将与它的正确性证明一起推导出算法。

许多线性代数算法都是基于循环的，而正确性证明的基础是 循环不变式 的推导：一个被归纳证明在每次循环迭代中都为真的谓词，从而保证整个算法的正确性。

67.3.1 常见算法的推导

在推导循环不变式之前，我们首先考虑分区形式下的计算结果。

$$y = Ax$$

分区：

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Two equations:

$$\begin{cases} y_T = A_T x \\ y_B = A_B x \end{cases}$$

归纳证明的关键在于采用这种分区形式，并假设它只是部分满足。

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

Assume only equation

$$y_T = A_T x$$

已满足。

现在我们接近归纳证明了：我们将算法视为旨在增加谓词为真的块的大小。如果这个块等于问题的规模，我们就有了完整结果的正确性。

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

While T 不是整个系统 谓词： $y_T = A_T x$ 为真 更新：将 T 块增加一个 谓词： $y_T = A_T x$ 对于新的 / 更大的 T 块

注意初始和最终条件。

现在我们比较一次迭代中的真实陈述和下一次中的陈述，并比较两个谓词为真的块。

这里是大技巧
Before

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

split:

$$\begin{pmatrix} y_1 \\ \cdots \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ \cdots \\ A_2 \\ A_3 \end{pmatrix} (x)$$

然后是更新步骤，和
After

$$\begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \\ \cdots \\ A_3 \end{pmatrix} (x)$$

以及未拆分

$$\begin{pmatrix} y_T \\ y_B \end{pmatrix} = \begin{pmatrix} A_T \\ A_B \end{pmatrix} (x)$$

比较这两个块给我们提供了额外的谓词，该谓词是为了满足我们考虑的迭代而设计的：

更新前：

$$\begin{pmatrix} y_1 \\ \cdots \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ \cdots \\ A_2 \\ A_3 \end{pmatrix} (x)$$

so

$$y_1 = A_1 x$$

是正确的
然后是更新步骤，并且
之后

$$\begin{pmatrix} y_1 \\ y_2 \\ \cdots \\ y_3 \end{pmatrix} = \begin{pmatrix} A_1 \\ A_2 \\ \cdots \\ A_3 \end{pmatrix} (x)$$

so

$$\begin{cases} y_1 = A_1 x & \text{我们有了这个,} \\ y_2 = A_2 x \\ \text{我们需要这个} \end{cases}$$

这个额外的谓词可以轻易地转换为基本的计算，因此我们得到完整的算法:

当 T 不是整个系统时谓词: $y_T = A_T x$ true 更新:
$y_2 = A_2 x$ 谓词: $y_T = A_T x$ true 对于新的 / 更大
的 T 块

67.3.1.1 按列推导算法

在上一节中，我们推导了矩阵 - 向量乘法，以通常的方式表示：输出向量的每个元素都是矩阵行和输入向量的内积。你可能会认为这对于不太多的结果来说太多了。

考虑一下，我们可以使用相同的一般原则推导出矩阵 - 向量乘法的其他算法。我们的基本假设是将矩阵水平分成两个块行。如果我们将矩阵垂直分成两个块列，会发生什么?

我们将矩阵分成两个块列，并表达矩阵 - 向量乘法的结果。

$$y = Ax$$

分区:

$$(y) = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

公式:

$$\begin{cases} y = A_L x_T + A_R x_B \end{cases}$$

我们再次对一个部分完成的计算做出声明。

$$(y) = \begin{pmatrix} A_L & A_R \end{pmatrix} \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

Assume

$$y = A_L x_T$$

被构建，并增长 T 块。

我们再次比较在一次迭代中分裂矩阵和在下一次迭代中分裂矩阵，并比较部分谓词：

Before

$$(y) = (A_L \quad A_R) \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

split:

$$(y) = \begin{pmatrix} A_1 & \vdots & A_2 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ \cdots \\ x_2 \\ x_3 \end{pmatrix}$$

然后是更新步骤，和
After

$$(y) = \begin{pmatrix} A_1 & A_2 & \vdots & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_3 \end{pmatrix}$$

以及未分裂

$$(y) = (A_L \quad A_R) \begin{pmatrix} x_T \\ x_B \end{pmatrix}$$

这通过从另一个谓词中 ‘减去’ 一个谓词，给了我们由单次迭代推导出的额外结果，因此，在该迭代中完成的计算。

更新前：

$$(y) = \begin{pmatrix} A_1 & \vdots & A_2 & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ \cdots \\ x_2 \\ x_3 \end{pmatrix}$$

so

$$y = A_1 x_1$$

is true
然后是更新步骤，和
After

$$(y) = \begin{pmatrix} A_1 & A_2 & \vdots & A_3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_3 \end{pmatrix}$$

so

$$y = A_1 x_1 + A_2 x_2$$

in other words, we need

$$y \leftarrow y + A_2 x_2$$

因此我们得到矩阵 - 向量乘积的第二种形式。

While T is not the whole system

谓词 : $y = A_L x_T$ true 更新 : $y \leftarrow y + A_2 x_2$ 谓词 :

$y = A_L x_T$ true for new/bigger T block

在准 -Matlab 记号中，我们表示这两种算法:

for $r = 1{,}m$

$y_r = A_{r,*} x_*$

$y \leftarrow 0$

for $c = 1{,}n$

$y \leftarrow y + A_{*,c} x_c$

第 68 章

单元测试与测试驱动开发

在一个理想的世界里，你会证明你的程序是正确的，但在实践中这并不总是可行的，或者至少：不这样做。大多数情况下，程序员通过测试来验证他们代码的正确性。

是的，有一个艾德格 · 迪科斯彻 的名言说：

> 如今一个常见的做法是编写程序然后测试它。但是：程序测试可以是一个非常有效的方法来展示错误的存在，但对于展示它们的缺失却是绝望地不足够。（笑声）

但这并不意味着你不能通过测试至少对你的代码获得一些信心。

68.1 测试类型

测试代码是一门艺术，甚至比编写代码本身更重要。但这并不意味着你不能系统地对待它。首先，我们将一些基本的测试类型区分开来：

- 单元测试 通过单独测试程序的一小部分；
- 系统测试 测试整个软件系统的正确行为；和
- 回归测试 确认程序的行为在添加或改变其方面后没有发生变化。

在本节中，我们将讨论单元测试。

一个以足够模块化方式编写的程序允许其组件在不等待整个程序的全有或全无测试的情况下进行测试。因此，测试和程序设计在它们的利益上是一致的。事实上，在编写程序时考虑到它需要可测试性，可以导致更干净、更模块化的代码。

在这种极端形式下，你会通过测试驱动开发（TDD）来编写你的代码，其中代码开发和测试是同步进行的。基本原理可以表述如下：

- 整个代码及其部分应该始终是可测试的。
- 在扩展代码时，只做允许测试的最小更改。
- 在每次更改前后进行测试。
- 在添加新功能之前确保正确性。

在严格的解释下，你甚至需要对程序的每个部分首先编写它将满足的测试，然后才是实际的代码。

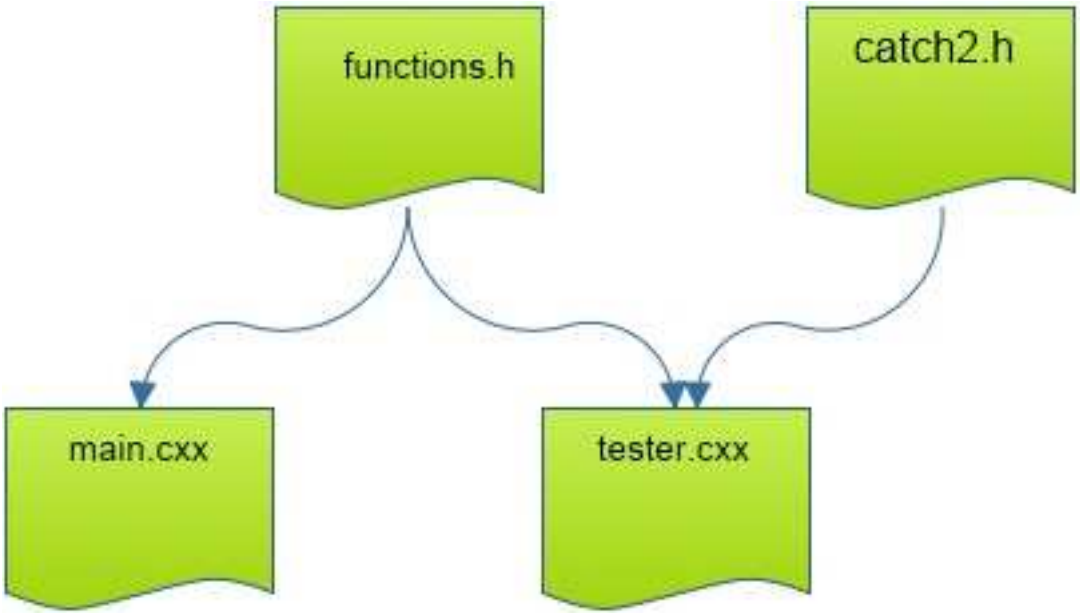


图 68.1：单元测试的文件结构

68.2 单元测试框架

有几个 ‘框架’ 可以帮助你进行单元测试。在本章的其余部分，我们将使用 *Catch2*，这是 C++ 中最常用的框架之一。

See section 68.6 for the practical matters of how to obtain, install, and compile with Catch2.

68.2.1 测试用例

一个测试用例是一个作为独立主程序运行的短程序。在上述建议的设置中，你将所有单元测试放在测试器主程序中，即包含

```
#define CATCH_配置_MAIN#
include "catch2/catch_all.hpp"
```

magic lines.

每个测试用例都需要有一个唯一的名称，当测试失败时将打印该名称。你可以选择性地为测试用例添加键，以便从命令行选择测试。

```
TEST_CASE( "此测试的名称") {// 内容 }TEST_CASE( "此测
试的名称","[键 1][键 2]" ){// 内容 }
```

测试用例的主体本质上是一个主程序，其中一些语句被封装在测试宏中。最常用的宏是 `REQUIRE`，用于要求某些条件正确。

Tests go in `tester.cpp`:

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    REQUIRE( f(n)>0 );
}
```

- *TEST_CASE* 像独立的 main 程序一样。测试器文件中可以有多个 case
- *REQUIRE* 像 assert 但更复杂

练习 68.1。

1. 编写一个函数

   ```
   double f(int n) { /* .... */ }
   ```

   that takes on positive values only.
2. Write a unit test that tests the function for a number of values.

您可以根据代码仓库中的 *tdd.cxx* 文件进行参考

布尔值:

```
REQUIRE(一些_测试(一些_输入));
REQUIRE(不一些_测试(其他_输入));
```

整数:

```
REQUIRE(整数_函数(1)==3);
REQUIRE(整数_函数(1)!=0);
```

注意浮点数:

```
REQUIRE(real_function(1.5)==Catch::Approx(3.0));
REQUIRE(real_function(1)!=Catch::Approx(1.0));
```

In general exact tests don't work.

对于失败的测试，框架将提供测试的名称、行号以及被测试的值。

运行测试器:

```
-------------------------------
test the increment function
-------------------------------
test.cpp:25
...............................

test.cpp:29: FAILED:
  REQUIRE( increment_positive_only(i)==i+1 )
with expansion:
  1 == 2
```

```
        ================================
        test cases: 1 | 1 failed
        assertions: 1 | 1 failed
```

在上述情况下，错误信息打印出了有问题的值 `f(n)`，而不是它发生的值 `n`。要确定这一点，请插入 *INFO* 规范，这些规范只有在测试失败时才会打印出来。

INFO: print out information at a failing test

```
    TEST_CASE( "test that f always returns positive" ) {
      for (int n=0; n<1000; n++)
        INFO( "function fails for " << n );
        REQUIRE( f(n)>0 );
    }
```

如果你的代码抛出异常（第 23.2.2 节）你可以测试这些。

假设函数 g(n)

- 在输入 $n > 0$ 时成功
- fails for input

$n \leq 0$: throws 异常

```
    TEST_CASE( "test that g only works for positive" ) {
      for (int n=-100; n<+100; n++)
        if (n<=0)
          REQUIRE_THROWS( g(n) );
        else
          REQUIRE_NOTHROW( g(n) );
    }
```

在单元测试中，常见的情况是具有公共设置或拆除的多个测试，使用您在单元测试中有时会遇到的术语。Catch2 支持这一点：您可以在设置和拆除之间的部分创建 “ 部分 ”。

如果测试具有共同的 intro/outtro，请使用 SECTION：

TEST_CASE("commonalities") { // 公共设置： double x,y,z; REQUIRE_NOTHROW(y = f(x)); // 两个独立的测试： SECTION("g function") { REQUIRE_NOTHROW(z =g(y)); } SECTION("h function") { REQUIRE_NOTHROW(z =h(y)); } // 公共后续 REQUIRE(z>x); }

(有时称为设置 / 拆除)

68.3 示例：二分法求零点

二分法求零点算法的开发可以在第 47.1 节中找到 .

68.4 一个例子：二次方程的根

我们重新审视练习 24.6，它使用了 `std::variant` 来返回一个二次方程的 0,1,2 个根。在这里我们使用 TDD 来得到代码。

Throughout, we represent the polynomial

$$ax^2 + bx + c$$

as

```
using quadratic = tuple<double,double,double>;
```

当需要时，您可以用

```
auto [a,b,c] = coefficients;
```

(你能想到一个断言 语句在这里可能有用吗？)

练习 68.2. 编写一个函数

```
double 判别式 ( 二次系数 )                    ;
```

该函数计算 $b^2 - 4ac$，并测试：

```
// /quadtest.cpp
TEST_CASE( "判别式" ) {
  REQUIRE( 判别式 (make_tuple(0., 2.5, 0.) ) ==Catch::Approx(6.25) );
  REQUIRE( 判别式 (make_tuple(1., 0., 1.5 ) ) == Catch::Approx(-6.) );
  REQUIRE( 判别式 ( make_tuple(.1, .1, .1*.5 )== Catch::Approx(-.01));
}
```

要说明如果省略近似相等测试会发生什么，可能很有说明性：

```
要求 ( 判别式 ( 创建_元组 (.1, .1, .1*.5 ) ) == -.01 );
```

有了这个函数，很容易检测无根的情况：判别式 $D < 0$。接下来我们需要有单根或双根的标准：如果 $D = 0$，则有一个单根。

练习 68.3。 编写一个函数

```
bool discriminant_zero( 二次系数                    s );
```

通过测试

```
// /quadtest.cpp
二次系数= 使_元组 (a,b,c);
d= 判别式 ( 系数);
z= 判别式_零 ( 系数);
INFO( a<< "," << b << "," << c << " d=" << d);
REQUIRE(z);
```

例如使用以下值:

```
a = 2; b = 4; c = 2;
a = 2; b = sqrt(40); c = 5; // !!!
a = 3; b = 0; c = 0.;
```

这个练习是我们第一次遇到数值细节的地方。第二个测试值集的判别式在精确算术中为零，但在计算机算术中不为零。因此，我们需要测试它是否足够小，与 b 相比。

Exercise 68.4. Be sure to also test the case where `discriminant_zero` returns false.

既然我们已经检测到一个根，我们需要计算它的函数。这一个没有细节。

练习 68.5. 编写函数 `simple_root`，它返回单个根。为了确认，测试

```
// /quadtest.cpp auto r= simple_root(系数);REQUIRE( evaluate(系数,r)==
Catch::Approx(0.).margin(1.e-14));
```

剩余的两种不同根的情况是通过排除法得出的，唯一需要做的是编写一个返回它们的函数。

练习 68.6. 编写一个函数，将两个根作为
indexcstdpair:

```
pair<double,double> double_root( quadratic coefficients );
```

Test:

1// /quadtest.cpp 2 二次系数= make_tuple(a,b,c); 3 auto [r1,r2] =double_root(coefficients); 4 auto 5 e1 =evaluate(coefficients,r1), 6 e2 =evaluate(coefficients,r2); 7 REQUIRE(evaluate(coefficients,r1)==Catch::Approx(0.).margin(1.e-14)); 8 REQUIRE(evaluate(coefficients,r2)==Catch::Approx(0.).margin(1.e-14));

最后一段代码是测试有多少个根的函数，并将它们作为 `std::variant`。

练习 68.7. 编写一个函数

```
变体<bool,double, pair<double,double> >
compute_roots(二次系数);
```

测试:

```
1// /quadtest.cpp2TEST_
CASE( "full"
                        test" ) {
3   double a,b,c; int index;
4   章节 ( "无根") {
5      =2.0; =1.5; =2.5
6      index = 0;
7   } 章节 ( "单根") {
8
9      =1.0; =4.0;=4.0;
10     index = 1;
11  }
12  SECTION( "double root" )      {
13     =2.2; =5.1; =2.5
14     index = 2;
15  } 二次系数 =
16
     make_tuple(a,b,c);
17  auto 结果 =
     compute_roots ( 系数 );
18  REQUIRE ( 结果 .index ()==index);
19 }
```

68.5 八皇后示例

See 48.3 .

68.6 Catch2 的使用实践

68.6.1 安装 Catch2

您可以在 https://github.com/catchorg 找到 Catch2 的代码。您可以下载发布版本，或者克隆代码仓库。这里我假设版本 3.1.1。

// 下载发布版本 wget https://github.com/catchorg/Catch2/archive/refs/tags/v3.1.1.tar.gz // 解压 tar fxz v3.1.1.tar.gz // 创建构建和安装目录 mkdir -p build-catch2 mkdir -p installation-catch2 // 进入构建目录 cd build // 构建安装 cmake -D CMAKE_INSTALL_PREFIX=../installation-catch2\ -D BUILD_SHARED_LIBS=TRUE \ ../Catch2-3.1.1 make make install

68.6.2 两种使用模式

假设你有一个文件结构，其中包含

- 一个非常简短的主程序，以及
- 一个库文件，其中包含主程序使用的所有函数。

为了测试这些函数，你需要提供一个另一个主程序，该程序只包含单元测试；这如图 68.1 所示。

实际上，使用 Catch2 时，你的主文件实际上没有 `main` 程序：这个程序由框架提供。在测试器主文件中，你只需要放置测试用例。

框架提供自己的 main：

```
#define CATCH_配置 _MAIN
#include "catch2/catch_all.hpp"

#include "library_functions.h"
/*
   here follow the unit tests
*/
```

一个重要的问题是包含哪个头文件。你可以做

```
#include "catch.hpp"
```

这是 ‘仅头文件’ 模式，但这会导致编译非常慢。因此，我们将假设您已通过 *Cmake* 安装了 Catch，并且您包含

```
#include "catch2/catch_all.hpp"
```

注意：截至 2021 年 9 月，这需要代码仓库的开发版本，而不是任何 2.x 版本。

68.6.3 编译

上述设置要求您向您的设置中添加编译和链接标志。这是与系统相关的。

One-line solution:

```
icpc -o tester test_main.cpp \ -I${TACC_CATCH2_INC} -
L${TACC_CATCH2_LIB} \ -lCatch2Main -lCatch2
```

Variables for a Makefile:

```
INCLUDES= -I${TACC_CATCH2_INC} EXTRALIBS = -L${TACC_
CATCH2_LIB} -lCatch2Main -lCatch2
```

在 CMake 中，可以通过 *pkgconfig* 发现 Catch2：

用于 CMake 配置的行：

```
find_包 ( PkgConfig REQUIRED ) pkg_检查 _模块 ( CATCH2
REQUIRED catch2-with-main ) 目标 _包含 _目录 ( ${PROGRA
M_NAME} 公开 ${CATCH2_INCLUDE_DIRS} ) 目标 _链接 _目录
( ${PROGRAM_NAME} 公开 ${CATCH2_LIBRARY_DIRS} )
```

68. 单元测试和测试驱动开发

第 69 章

使用 gdb 进行调试

69.1 一个简单的示例

以下程序没有任何错误；我们使用它来展示一些 gdb 的基础知识。

```
    // /hello.cpp void say(int n) { cout <<"
hello world " << n << ’ \n’ ; } intmain() {
for(inti=0;i<10; ++i) { int ii; ii =i*i;++ii;
say(ii); } return 0; }
```

69.1.1 调用调试器

在您编译完程序后，不是按正常方式运行它，而是调用 *gdb*：gdb myprogram

这会带您进入一个环境，可通过 (gdb) 提示符识别：GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-115.el7[stuff](gdb)

在那里，您可以使用 *run* 命令来对程序进行受控运行：

```
    (gdb) 运行 启动程序：/
home/eijkhout/gdb/hello hello world 1 hello
world 2
```

hello world 5 hello world 10 hello world 17 hello world 26 hello world 37 hello world 50 hello world 65 hello world 82[从属进程 1 (进程 30981) 正常退出]

69.2 示例：整数溢出

以下程序展示了整数溢出。(我们使用 **short** 来强制这种情况尽快发生。)

代码：

```
1 //overflow.c
2 《C++17/Fortran2008 科学编程艺术，第 3 卷》引言 69.2 示例：整数溢出
pp 2 void say(short n) { 3 cout <<"
hello wor                          ld " << n << '\n';
4 }
5
6 int main() {
7
8   for(short i=0; ; i+=20) {
9     短 ii;
10    ii =i*i;
11    ++i;
12    say(ii);
13  }
14
15  返回 0;
16 }
```

C++17/Fortran2008 高性能计算艺术，第 3 卷：科学编程入门 69.2 示例：整数溢出

[gdb] hello:

```
你好，世界 1
你好，世界 401
你好，世界 1601
你好，世界 3601
你好，世界 6401
你好，世界 10001
你好，世界 14401
你好，世界 19601
你好，世界 25601
你好世界 32401
你好世界 -25535
你好世界 -17135
你好世界 -7935
你好世界 2065
你好世界 12865
你好世界 24465
你好世界 -28671
你好世界 -15471
你好世界 -1471
你好世界 13329
```

69.3 More gdb

69.3.1 Runwith commandlinearguments

此程序是自包含的，但如果你有一个接受 *commandline arguments* 的程序：

```
./myprogram 25
```

你可以通过 gdb 提供这些：

```
(gdb) run 25
```

69.3.2 源代码列表和正确编译

Inside gdb, you can get a source listing with the *list* command.

让我们再试一下我们的程序：

[ ]icpc -o hello hello.cpp[ ]gdb hello GNU gdb (GDB) Red HatEnterprise Linux 7.6.1-115.el7 版权所有 (C) 2013 Free Software Foundation, Inc. 正在读取 / home/eijkhout/gdb/hello... (未找到调试符号)... 完成。 (gdb) list 没有加载符号表。请使用 "file" 命令。

看到对 ‘symbols’ 的重复引用了吗？您需要为 符号表 提供 - g 编译器选项，以便将其包含在二进制文件中：

[ ]icpc-g-o hello hello.cpp[ ]gdb hello GNUgdb (GDB) Red Hat EnterpriseLinux7.6.1-115.el7[stuff] 从 / home/eijkhout/gdb/hello 读取符号 ... 完成。 (gdb) list 13 using std::cout; 14 using std::endl;[等等]

(如果你现在按回车，list 命令会重复执行，你会得到下一块行。执行 list - 会给你当前所在位置的上一块行。)

69.3.3 单步调试源代码

现在让我们对程序进行更受控的运行。在源代码中，我们看到第 22 行是第一个可执行行：

```
20      int main() {
21
22        for (int i=0; i<10; i++) {
23          int ii;
24          ii = i*i;
...
```

我们使用 *break* 命令设置一个断点：

```
(gdb) break 22
Breakpoint 1 at 0x400a03: file hello.cpp, line 22.
```

(如果你的程序分布在多个文件中，你可以指定文件名：break otherfile.cpp:34。)

现在如果我们运行程序，它将停在那一行：

(gdb) 运行 启动程序：/ home/eijkhout/gdb/hello 断点 1，main() at hello.cpp:22 22 for(int i=0;i<10;i++){

更精确地说：程序在执行这一行之前的状态下停止。

现在我们可以使用 *cont*（用于“继续”）来让程序继续运行。由于没有其他断点，程序将运行到结束。这并不太有用，所以让我们改变断点的位置：如果执行在每次迭代的开始处停止，那就更有用了。

回想一下断点的编号是 1，所以我们使用 *delete* 来删除它，而是在循环体内设置断点，然后继续直到我们遇到它。

```
(gdb) delete 1
(gdb) break 23
Breakpoint 2 at 0x400a29: file hello.cpp, line 23.
(gdb) cont
Continuing.
Breakpoint 2, main () at hello.cpp:24
24          ii = i*i;
```

（注意，第 23 行不可执行，所以执行会在该行之后的第 1 行停止。）

现在如果我们继续执行，程序会运行到下一个断点：

(*gdb*) *cont* 继续。*hello world*5 断点 1，*main* () 在 *hello.cpp*:2424 *ii*= *i***i*;

要进入下一条语句，我们使用 *next*：

(gdb) next 25 ii ++; (gdb)

按下回车键会重新执行上一个命令，所以我们转到下一行：

(gdb) 26 say(ii); (gdb) hello world 2

```
Breakpoint 1, main () at hello.cpp:24
24          ii = i*i;
```

您观察到函数调用

1. 被执行，如 hello world 1 的输出所示，但
2. 在调试器中未详细显示。

结论是，next 会跳转到当前子程序中的下一个可执行语句，而不是进入它调用的函数和类似的东西。

如果你想进入函数 say，你需要使用 <code>step</code>：

(gdb) next 25 ii++; (gdb) next 26 say(ii); (gdb) step say(n=10) at hello.cpp:17 17 cout << "helloworld"<<n<< endl;

调试器报告函数名称，以及参数的名称和值。另一个 ‘ 步进 ’ 执行当前行并将我们带到函数的末尾，下一个 ‘ 步进 ’ 将我们带回到主程序：

(gdb) hello world 10 18 } (gdb) main ()at hello.cpp:24 24 ii =i*i;

69.3.4 检查值

当执行在某一行停止时（记住，这意味着在执行之前！）你可以检查该子程序中的任何值：

```
24          ii = i*i;
(gdb) print i
$1 = 4
```

你甚至可以让表达式使用局部变量进行求值：

(gdb) 打印 2*i $ 2 = 8

你可以通过查看值和使用断点来组合这种方式。假设你想知道变量 ii 何时超过 40：

(gdb) break 26 if ii>40 断点 1 在 0x4009cd：文件 hello.cpp，第 26 行。(gdb) run 启动程序：/home/eijkhout/intro-programming-private/code/gdb/hello hello world 1

```
hello world 2
hello world 5
hello world 10
hello world 17
hello world 26
hello world 37
```

断点 1，main () at hello.cpp:26 26 say(ii); 缺少单独的调试信息，使用：debuginfo-install glibc-2.17-292.el7.x86 (gdb) print i $1 = 7

69.3.5 一个 NaN 示例

以下程序：

```
17 float root(float n) 18 { 19 float r; 20 float n1 =
n-1.1; 21 r =sqrt(n1); 22 return r; 23 } 24 25 int
main() { 26 float x=9,y; 27 for (int i=0; i<20; i++) {
28 y= root(x); 29 cout <<"root: " << y << endl; 30
x-= 1.1; 31 } 32 33 return 0; 34 }
```

打印一些 ‘不是数字’ 的数字：

```
[] ./root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214
root: -nan root: -
nan root: -nan
```

假设你想弄清楚为什么会这样。

打印 ‘nan’ 的那行是 29，所以我们要在那里设置一个断点，最好是一个条件断点。但如何测试 ‘nan’？这需要一点技巧。

(gdb) break 29 if y!=y 断点 1 at 0x400ea6:fileroot.cpp,line28. (gdb) run 启动程序 : /home/eijkhout/intro-programming-private/code/gdb/root root: 2.81069 root: 2.60768 root: 2.38747 root: 2.14476 root: 1.87083 root: 1.54919 root: 1.14018 root: 0.447214 断点 1, main ()atroot.cpp:29 29 cout<< "root:"<<y<<endl;

我们发现这是第几次迭代： (gdb) print i $1 = 8

现在我们可以重新运行程序，并调查那个特定的迭代：

(gdb) break 28 if i==8 断点 2 at 0x400eaf:fileroot.cpp, 第 28 行。(gdb) run 正在调试的程序已经启动。要从头开始启动吗? (y/n)y .

```
Starting program: /home/eijkhout/intro-programming-private/code/gdb/root
root: 2.81069
root: 2.60768
root: 2.38747
root: 2.14476
root: 1.87083
root: 1.54919
root: 1.14018
root: 0.447214

Breakpoint 2, main () at root.cpp:28
28          y = root(x);
```

我们现在进入根例程，看看那里出了什么问题：

(gdb) 单步执行 root(n=0.200000554) at root.cpp:20 20 float n1 = = n - 1.1; (gdb) 21 r = =sqrt(n1); (gdb) 打印 n $2 = 0. 200000554 (gdb) 打印 n1 $3 = - 0.89999944 (gdb) 下一步 22 return r; (gdb) 打印 r $4 = -nan(0x400000)

在那里我们发现了问题：我们的输入 n 用来计算另一个数 n1，我们计算它的平方根，有时这个数会变成负数。

69.3.6 断言

与其运行程序并在发现问题时调试它（请注意，这并不总是会发生！），你还可以通过包含断言来使你的程序更健壮。这些是你根据你对所解决问题了解而知道应该是正确的事情。

例如，在之前的示例中有一个平方根函数，并且你只是 ‘知道’ 输入总是正的。因此，你可以按照以下方式编辑你的程序：

```
// 允许断言：#include <cassert>
float root(float n){float
r;float n1 = n-1.1;assert(n
1>=0); //注意！ r=
sqrt(n1);return r;}
```

现在如果你运行你的程序，你会得到：

[] ./assert 根：
2.81069 根：
2.60768 根：
2.38747 根：
2.14476 根：
1.87083 根：
1.54919 根：
1.14018 根：
0.447214

assert: assert.cpp:22: float root(float): 断言 ‘n1>=0’ 失败。终止 (核心转储)

这会给你什么结果?

- 它只告诉你断言失败了，但没有提供具体值;
- 它不会提供回溯信息；另一方面
- 断言可以帮助你检测可能被忽略的错误条件!

69. 使用 gdb 进行调试

第 70 章

复杂度

70.1 算法的复杂度

70.1.1 Theory

关于复杂度理论，请参阅 HPC 书籍 [11], 第 15 节。

70.1.2 时间复杂度

练习 70.1。 对于从 1 到 100 的每个数字 n，打印从 1 到 n 的所有数字的和。

这个问题有几种可能的解法。假设你不知道求和公式 $1 \ldots n$，你可以有一个保持运行总和的解法，以及一个带有内部循环的解法。

练习 70.2. 这两个解法中，作为 n 的函数，执行了多少次操作?

70.1.3 空间复杂度

练习 70.3. 读取用户输入的数字；当用户输入零或负数时停止读取。将所有正数加起来并打印它们的平均值。

这个问题可以通过将数字存储在 std::vector 中来解决，但也可以保持运行总和和计数。

练习 70.4. 这两个解法需要多少空间?

70. 复杂度

第 71 章

支持工具

仅仅知道如何编写程序是不够的：你需要各种工具才能成为一名优秀的程序员，或者成为一名程序员。

71.1 编辑器和开发环境

简单的程序，例如本书中的大多数练习，可以使用简单的编辑器编写。传统上，程序员使用 `emacs` 或 `vi` (或其衍生版本，如 vim)。

更复杂的是开发环境，例如 *Microsoft Visual Studio Code* 或 *CLion*。

71.2 编译器

对于本书中的简单练习，您需要知道的关于编译器的所有内容就是命令行

icpx - myprogram myprogram.cpp

(或者您使用的编译器名称和程序名称)

编译器还有更多需要了解的内容；参见教程书籍 [9]，章节 -2.

71.3 构建系统

对于比单个源文件更复杂的程序（甚至在这种情况下），你明智地使用某种形式的构建系统。

- The simplest and oldest solution is `Make`; see Tutorials book [9], chapter-3.
- More modern, more powerful, yet also in a way more complicated, is `CMake`; see Tutorials book [9], chapter-4.
- Make and Cmake are often integrated into the above-mentioned development environments.

71.4 调试器

如果你的代码行为不正常，有一个好的 调试器 可以救命。传统的调试器是 `gdb` (参见教程书籍 [9]，章节 -11) 但是，再次强调，这通常集成到构建环境中。

71. 支持工具

第六部分

索引和其他

第 72 章

C++ 关键字索引

第 73 章

Fortran 关键字索引

索引

第 74 章

参考文献

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: 一种用于高性能计算机的可移植线性代数库 . 在《超级计算会议论文集 *'90* 》 中 , 第 2–11 页 . IEEE 计算机协会出版社 , 加利福尼亚州洛斯阿尔 amos, 1990.

[2] Roy M. Anderson 和 Robert M. May. 传染病种群生物学：第一部分 . *Nature*, 280:361–367, 1979. doi:10.1038/280361a0. 483

[3] Satish Balay 、 William D. Gropp 、 Lois Curfman McInnes 和 Barry F. Smith。面向对象数值软件库中并行性的高效管理。在 E. Arge 、 A. M. Bruaset 和 H. P. Langtangen 编辑的《现代科学计算软件工具》中，第 163–202 页。Birkhäuser 出版社，1997 年。

[4] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. PETSc 网页。http://www.mcs.anl.gov/petsc,2011。 332

[5] Tolga Bektas. 多重旅行商问题：公式和求解方法的概述。 *Omega*, 34(3):209 – 219, 2006。 510

[6] Robert S. Boyer 和 J. Strother Moore. 一种快速字符串搜索算法。 *Commun. ACM*, 20(10):762–772,1977 年 10 月。 534

[7] Yen-Lin Chen, Chuan-Yen Chiang, Yo-Ping Huang, 和 Shyan-Ming Yuan. 一个基于项目的 C++ 面向对象编程课程。在 *2012* 年 *9* 届国际普适计算与计算会议和 *9* 届国际自主与可信计算会议论文集 , UIC-ATC ’12, 第 667–672 页，华盛顿特区，美国，2012 年。IEEE 计算机协会。 29

[8] https://github.com/jarro2783/cxxopts. 520

[9] Victor Eijkhout. HPC carpentry. https://theartofhpc.com/carpentry.html. 262, 272, 447, 621

[10] Victor Eijkhout. MPI 和 OpenMP 中的并行编程。http://theartofhpc.com/pcse.html. 215

[11] Victor Eijkhout. 计算科学 . http://theartofhpc.com/istc.html. 20,46, 298, 333, 334, 472, 495, 510, 513, 514, 517,525, 526, 527, 585, 586, 619

[12] Barbara J. Ericson 、 Lauren E. Margulieux 和 Jochen Rick。解决 parsons 问题与修复和编写代码。在《第 17 届 KoliCalling 国际计算教育研究会议论文集》， Koli Calling 17，第 20-29 页，纽约，纽约州，美国，2017 年。美国计算机协会。

[13] Google. Google 开发者文档风格指南。https://developers.google.com/style/

[14] Kazushige Goto 和 Robert A. van de Geijn. 高性能矩阵乘法的解剖。*ACM Trans. Math. Softw.*, 34(3):1–25,2008. 513,518

[15] https://mathworld.wolfram.com/Kermack-McKendrickModel.html. 483

[16] C. L. Lawson, R. J. Hanson, D. R. Kincaid, 和 F. T. Krogh. 用于 Fortran 使用的线性代数子程序。*ACM Trans. Math. Softw.*, 5(3):308–323, 1979 年 9 月 . 514

[17] Harry L. Reed. 在 ENIAC 上的发射表计算。在 *1952* 年 *ACM* 全国会议（匹兹堡）的论文集 , ACM ' 52, 第 103?106 页，纽约，纽约，美国，1952 年。Association for ComputingMachinery. 541

[18] Juan M. Restrepo, , 和 Michael E. Mann. 这就是 .. 气候如何一直在变化 ..。 *APS* 物理，*GPS* 新闻简报 , 2018 年 2 月 . 545

[19] Lin S. and Kernighan B. 一个有效的启发式算法用于旅行商问题。运筹学 , 21:498–516,1973. 509